Revisiting Hyper Binary Resolution*

Marijn J. H. Heule[1,3], Matti Järvisalo[2], and Armin Biere[3]

[1] Department of Computer Science, The University of Texas at Austin, United States
[2] HIIT & Department of Computer Science, University of Helsinki, Finland
[3] Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria

Abstract. This paper focuses on developing efficient inference techniques for improving conjunctive normal form (CNF) Boolean satisfiability (SAT) solvers. We analyze a variant of hyper binary resolution from various perspectives: We show that it can simulate the circuit-level technique of structural hashing and how it can be realized efficiently using so called tree-based lookahead. Experiments show that our implementation improves the performance of state-of-the-art CNF-level SAT techniques on combinational equivalent checking instances.

1 Introduction

Boolean satisfiability (SAT) solvers provide the crucial core search engines for solving problem instances arising from various real-world problem domains. This paper focuses on developing efficient inference techniques to improve the robustness of conjunctive normal form (CNF) SAT solving techniques. Especially, our goal is to improve CNF-level techniques on instances of *miter-based combinational equivalence checking* which is an important industrially-relevant problem domain. The main motivation behind this work is to take notable steps towards the ambitious goal of making CNF-level approaches competitive with circuit-level techniques for equivalence checking. This goal is important as it would notably simplify the current state-of-the-art techniques applied in the industry which require alternating between circuit-level techniques and CNF-level SAT solving. To this end, we identify how known CNF-level SAT solving techniques can simulate the circuit-level technique of *structural hashing*—which plays an integral role in solving miter instances—purely on the level of a standard CNF encoding of Boolean circuits. As the main CNF-level approach, we study a variant of hyper binary resolution (HBR), which can be used to learn non-transitive hyper binary resolvents, and analyze this technique from various perspectives. While this variant or HBR has already been studied and implemented previously within the HYPRE [1] and HYPERBINFAST [2] CNF simplifiers, we extend this previous work both from the theoretical and practical perspectives.

Our main theoretical observations include: (i) explanations for how and to what extent the CNF techniques HBR, clause learning, and ternary resolution can simulate structural hashing; (ii) that HBR can be focused in a beneficial way to produce only

* The first author is supported by DARPA contract number N66001-10-2-4087. The first and third authors are supported by Austrian Science Foundation (FWF) NFN Grant S11408-N23 (RiSE), and the second author by Academy of Finland (grants 132812 and 251170).

non-transitive resolvents that increase transitive reachability of the underlying binary implications; and (iii) providing an explicit quadratic worst-case example on the number of binary clauses added, which applies to all known implementations of HBR, and that has not been explicitly provided before. As the main practical contribution, we show how this variant of HBR can be realized efficiently using so called tree-based lookahead [3]. In fact, the tree-based lookahead algorithm described in this work is a substantially simplified version of the original idea, and is also of independent interest due to its much more general applicability for instance within CDCL SAT solvers. We show experimentally that our *TreeLook* implementation of HBR using tree-based lookahead clearly outperforms state-of-the-art CNF-level SAT techniques on instances encoding on miter-based equivalence checking CNF instances.

The rest of this paper is organized as follows. After preliminaries (Sect. 2), we discuss possibilities of simulating structural hashing on the CNF-level (Sect. 3). Then the considered variant of hyper binary resolution is defined and analyzed (Sect. 4), followed by an in-depth description of tree-based lookahead (Sect. 6) that enables implementing hyper binary resolution efficiently. Before conclusions, experimental results are presented (Sect. 7) and related work is discussed (Sect. 8).

2 Preliminaries

For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\neg x$. A *clause* is a disjunction of literals and a 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 (partial) truth assignment for a CNF formula F is a function τ that maps (a subset of) the literals in F to $\{0,1\}$. If $\tau(x) = v$, then $\tau(\neg x) = 1 - v$. A clause C is satisfied by τ if $\tau(l) = 1$ for some literal $l \in C$. A clause C is falsified by τ if $\tau(l) = 0$ for every literal $l \in C$. An assignment τ satisfies F if it satisfies every clause in F. We denote by $\tau(F)$ the reduced formula for which all satisfied clauses by τ and all falsified literals by τ are removed.

Two formulas are *logically equivalent* if they are satisfied by exactly the same set of assignments. A clause of length one is a *unit clause*, and a clause of length two is a *binary clause*. For a CNF formula F, F_2 denotes the set of binary clauses, and $F_{\geq 3}$ denotes the set of clauses of length three and larger.

Binary Implication Graphs Given a CNF formula F, the unique *binary implication graph* $\mathsf{BIG}(F)$ of F has for each variable x occurring in F_2 two vertices, x and $\neg x$, and has the edge relation $\{\langle \neg l, l' \rangle, \langle \neg l', l \rangle \mid (l \vee l') \in F_2\}$. In other words, for each binary clause $(l \vee l')$ in F, the two implications $\neg l \to l'$ and $\neg l' \to l$, represented by the binary clause, occur as edges in $\mathsf{BIG}(F)$. A node in $\mathsf{BIG}(F)$ with no incoming arcs is a *root* of $\mathsf{BIG}(F)$ (or, simply, of F_2). In other words, literal l is a root in $\mathsf{BIG}(F)$ if there is no clause of the form $(l \vee l')$ in F_2. The set of roots of $\mathsf{BIG}(F)$ is denoted by $\mathrm{RTS}(F)$.

BCP, Failed Literal Elimination (FLE**), and Lookahead** For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) propagates all unit clauses, i.e., repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove from $F \setminus \{(l)\}$ all clauses that contain the literal l, and remove the literal $\neg l$ from all clauses

in F, resulting in the formula $\mathrm{BCP}(F)$. A literal l is a *failed literal* if $\mathrm{BCP}(F \cup \{(l)\})$ contains the empty clause, implying that F is logically equivalent to $\mathrm{BCP}(F \cup \{(\neg l)\})$. FLE removes failed literals from a formula, or, equivalently, adds the complements of failed literals as unit clauses to the formula, until a fixpoint is reached. Failed literal elimination is sometimes also referred to as *lookahead*, and is often applied in non-CDCL DPLL solvers (*lookahead solvers* [4]).

Equivalent Literal Substitution (ELS**)** The strongly connected components (SCCs) of $\mathsf{BIG}(F)$ represent equivalent classes of literals (or simply *equivalent literals*) in F_2 [5]. *Equivalent literal substitution* refers to substituting in F, for each SCC G of $\mathsf{BIG}(F)$, all occurrences of the literals occurring in G with the representative literal of G. ELS is confluent, i.e., has a unique fixpoint, modulo variable renaming.

Transitive Reduction (TRD**)** A directed acyclic graph G' is a *transitive reduction* [6] of the directed graph G provided that (i) G' has a directed path from node u to node v if and only if G has a directed path from node u to node v, and (ii) there is no graph with fewer edges than G' satisfying the condition (i). For a CNF formula F, a binary clause $C = (l \vee l')$ is *transitive* in F if l' is reachable from $\neg l$ (equivalently, l is reachable from $\neg l'$) in $\mathsf{BIG}(F \setminus C)$. Applying TRD on $\mathsf{BIG}(F)$ amounts to removing from F all transitive binary clauses in F. TRD is confluent for the class of CNF formulas F for which $\mathsf{BIG}(F)$ is acyclic. This is due to the fact that the transitive reduction of any directed acyclic graph is unique [6]. For directed graphs with cycles, TRD is unique modulo node (literal) equivalence classes.

The main inference rule of interest in this work is the *hyper binary resolution rule*.

Hyper Binary Resolution (HBR**)** The resolution rule states that, given two clauses $C_1 = \{l, a_1, \ldots, a_n\}$ and $C_2 = \{\neg l, b_1, \ldots, b_m\}$, the clause $C = C_1 \bowtie C_2 = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, called the *resolvent* $C_1 \bowtie C_2$ of C_1 and C_2, can be inferred by *resolving* on the literal l. Many different simplification techniques are based on the resolution rule. In this paper of interest is *hyper binary resolution* [7]. Given a clause of the form $(l \vee l_1 \cdots \vee l_k)$ and k binary clauses of the form $(l' \vee \neg l_i)$, where $1 \leq i \leq k$, the hyper binary resolution rule allows to infer the *hyper binary resolvent* $(l \vee l')$ in one step. HBR is confluent since it only adds clauses to CNF formulas.

3 Simulating Structural Hashing on CNF

In this section we show that hyper binary resolution is surprisingly powerful in that it implicitly—purely on the CNF-level—achieves *structural hashing*, i.e., sharing of equivalent subformula structures, over disjunctive and conjunctive subformulas. This is surprising, as structural hashing is often considered one of the benefits of representing propositional formulas on the higher level of *Boolean circuits* rather than working on the flat CNF form. This result implies that structural hashing can be achieved also during the actual CNF-level solving process by applying HBR on the current CNF formula.

Boolean Circuits are a natural representation form for propositional formulas, offering *subformula sharing* via structural hashing. A Boolean circuit over a finite set $\mathcal{G}$ of *gates* is a set $\mathcal{C}$ of equations of form $g := f(g_1, \ldots, g_n)$, where $g, g_1, \ldots, g_n \in \mathcal{G}$ and f :

$\{1,0\}^n \to \{1,0\}$ is a Boolean function, with the additional requirements that (i) each $g \in \mathcal{G}$ appears at most once as the left hand side in the equations in $\mathcal{C}$, and (ii) the underlying directed graph

$$\langle \mathcal{G}, E(\mathcal{C}) = \{\langle g', g\rangle \in \mathcal{G} \times \mathcal{G} \mid g := f(\ldots, g', \ldots) \in \mathcal{C}\}\rangle$$

is acyclic. Each gate represents a specific subformula in the propositional formula expressed by the set of Boolean equations. If $g := f(g_1, \ldots, g_n)$ is in $\mathcal{C}$, then g is an f-gate (or of type f), otherwise it is an *input gate*. The following Boolean functions are some which often occur as gate types: $\text{NOT}(v)$ (1 if and only if v is 0), $\text{OR}(v_1, \ldots, v_n)$ (1 if and only if at least one of $v_1, \ldots, v_n$ is 1), $\text{AND}(v_1, \ldots, v_n)$ (1 if and only if all $v_1, \ldots, v_n$ are 1), $\text{XOR}(v_1, v_2)$ (1 if and only if exactly one of v_1, v_2, is 1), and $\text{ITE}(v_1, v_2, v_3)$ (1 if and only if (i) v_1 and v_2 are 1, or (ii) v_1 is 0 and v_3 is 1). The standard "Tseitin" encoding of a Boolean circuit $\mathcal{C}$ into a CNF formula $\text{TST}(\mathcal{C})$ works by introducing a Boolean variable for each gate in $\mathcal{C}$, and representing for each gate $g := f(g_1, \ldots g_n)$ in $\mathcal{C}$ the logical equivalence $g \leftrightarrow f(g_1, \ldots g_n)$ with clauses.

3.1 Structural Hashing on the CNF-Level via HBR

Structural hashing is a well-known technique for factoring out common sub-expression. It is an integral part of many algorithms for manipulating different data structures representing circuits [8,9,10,11,12].

Given a circuit $\mathcal{C}$ with $g := f(g_1, \ldots, g_n), g' := f(g_1, \ldots, g_n) \in \mathcal{C}$, *structural hashing* removes $g' := f(g_1, \ldots, g_n)$ from $\mathcal{C}$, i.e., detects that g and g' label the same function $f(g_1, \ldots, g_n)$ in $\mathcal{C}$. A Boolean circuit $\mathcal{C}$ is structurally hashed if g and g' are the same gate whenever $g := f(g_1, \ldots, g_n), g' := f(g_1, \ldots, g_n) \in \mathcal{C}$.

Proposition 1. *Let $\mathcal{C}$ be an arbitrary Boolean circuit. Assume that there are two distinct gates $g := f(g_1, \ldots, g_n)$ and $g' := f(g_1, \ldots, g_n)$ in $\mathcal{C}$, where $f \in \{\text{NOT}, \text{AND}, \text{OR}\}$. Then* HBR *applied to* $\text{TST}(\mathcal{C})$ *will produce the clauses $(\neg g \vee g')$ and $(g \vee \neg g')$ representing the fact that g and g' label the same function $f(g_1, \ldots, g_n)$ in $\mathcal{C}$.*

Basically the binary clauses in $\text{TST}(\mathcal{C})$ associated with $g := f(g_1, \ldots, g_n)$ together with a clause of arity $(n+1)$ associated with $g' := f(g_1, \ldots, g_n)$ always produce the binary clause equivalent to one of the directions of the bi-implication $g \leftrightarrow g'$. The binary clauses in $\text{TST}(\mathcal{C})$ associated with $g := f(g_1, \ldots, g_n)$ together with a clause associated with $g' := f(g_1, \ldots, g_n)$ will produce the other direction of the bi-implication.

Proof (Proof of Proposition 1). Assume that we have $g := \text{AND}(g_1, \ldots, g_n)$ and $g' := \text{AND}(g_1, \ldots, g_n)$. On the CNF-level we have the clauses $(\neg g \vee g_i), (g \vee \neg g_1 \vee \cdots \vee \neg g_n)$ and $(\neg g' \vee g_i), (g' \vee \neg g_1 \vee \cdots \vee \neg g_n)$, where $i = 1..n$. Now the hyper binary resolution rule allows to derive $(\neg g \vee g')$ in one step from $(\neg g \vee g_1), \ldots, (\neg g \vee g_n), (g' \vee \neg g_1 \vee \cdots \vee \neg g_n)$, and similarly $(\neg g' \vee g)$ in one step from $(\neg g' \vee g_1), \ldots, (\neg g \vee g_n), (g \vee \neg g_1 \vee \cdots \vee \neg g_n)$. The cases $f \in \{\text{NOT}, \text{OR}\}$ are similar. □

Especially, by Proposition 1 hyper binary resolution can achieve the same effect purely on the CNF-level as circuit-level structural hashing on *And-Inverter Graphs* (AIGs) [9] which are often used for representing circuit-level SAT instances. We say that HBR can hence *simulate* structural hashing of AIGs.

However, HBR is not strong enough to simulate structural hashing for XOR and ITE gates on the standard CNF encoding, simply because the CNF clauses produced by the standard CNF encoding for XOR and ITE gates do not include any binary clauses.

Observation 1 *Given a Boolean circuit $\mathcal{C}$ with two gates $g := f(g_1, \ldots, g_n)$ and $g' := f(g_1, \ldots, g_n)$. Assume g and g' label the same function $f(g_1, \ldots, g_n)$. If $f \in \{\text{XOR}, \text{ITE}\}$ then HBR cannot in general derive $g \leftrightarrow g'$ (i.e., establish that g and g' label the same function) from $\text{TST}(\mathcal{C})$.*

3.2 Other Approaches to Structural Hashing on the CNF-Level

Structural Hashing and CDCL Interestingly, CNF-level *conflict-driven clause learning* (CDCL) SAT solvers can *in principle* simulate structural hashing by learning the bi-implication $g \leftrightarrow g'$. By "in principle" we mean that this requires a CDCL solver to assign the "right" values to the "right" variables in the "right" order, and to restart after each conflict (and possibly to postpone unnecessary unit propagations).

Observation 2 *CDCL can in principle simulate structural hashing of any Boolean circuit $\mathcal{C}$ on $\text{TST}(\mathcal{C})$, assuming that the solver assigns variables optimally, restarts after every conflict, and can postpone unit propagation at will.*

The intuition behind this observation is the following. Given any Boolean circuit $\mathcal{C}$ containing two gates g and g', where $g := f(g_1, \ldots, g_n)$ and $g' := f(g_1, \ldots, g_n)$. For simplicity, let us assume $g := \text{AND}(g_1, \ldots, g_n)$ and $g' := \text{AND}(g_1, \ldots, g_n)$. Now apply CDCL as follows on $\text{TST}(\mathcal{C})$. First, assign $g = 0$. Notice that unit propagation does not assign values to any g_i based on $g := \text{AND}(g_1, \ldots, g_n)$. Then assign $g' = 1$. Now unit propagation assigns $g_i = 1$ for all $i = 1..n$, resulting in a conflict with $g = 0$. The key observation is that the standard 1-UIP clause learning scheme will now learn the clause $(g \vee \neg g')$, since this is the only 1-UIP conflict clause derivable from the conflict graph restricted to the clauses associated with $g := \text{AND}(g_1, \ldots, g_n)$ and $g' := \text{AND}(g_1, \ldots, g_n)$. Then let the solver restart, and afterward assign similarly first $g' = 0$ and then $g = 1$ in order to learn the clause $(g' \vee \neg g)$.

A similar argument goes through also for XOR and ITE but needs one more decision to learn one auxiliary clause for each of the two implications. Consider for instance $g := \text{XOR}(g_1, g_2)$ and $g' := \text{XOR}(g_1, g_2)$. Assigning $g = 0$, $g' = 1$ and then $g_2 = 0$ allows learning the clause $(g \vee \neg g' \vee g_2)$. After backtracking, unit propagation on this clause assigns $g_2 = 1$ which results in another conflict, from which one of the two implications $(g \vee \neg g')$ is learned. The other implication can be derived in a similar way.

From the practical point of view, however, it is unlikely that CDCL solver implementations would behave in the way just described.

Structural Hashing using Ternary Resolution Further we claim that another way of achieving structural hashing of XOR and ITE on the CNF-level is to apply *ternary resolution*, originally suggested in [13] and subsequently applied as an inference technique

in the contexts of both complete [14] and local search methods [15] for CNF SAT. Ternary resolution refers to restricting the resolution rule between two ternary clauses so that only a ternary or binary resolvent are inferred (i.e., added to the CNF).

Proposition 2. *Ternary resolution simulates structural hashing of* ITE *and* XOR.

Proof. Consider the clauses for two ITE gates $x := \text{ITE}(c, t, f)$ and $y := \text{ITE}(c, t, f)$:

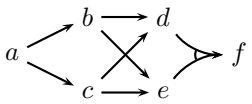
$$(\neg x \vee \neg c \vee t) \wedge (\neg x \vee c \vee f) \wedge (x \vee \neg c \vee \neg t) \wedge (x \vee c \vee \neg f)$$
$$(\neg y \vee \neg c \vee t) \wedge (\neg y \vee c \vee f) \wedge (y \vee \neg c \vee \neg t) \wedge (y \vee c \vee \neg f)$$

Using ternary resolution, $(\neg x \vee y \vee \neg c) = (\neg x \vee \neg c \vee t) \bowtie (y \vee \neg c \vee \neg t)$ and $(\neg x \vee y \vee c) = (\neg x \vee c \vee f) \bowtie (y \vee c \vee \neg f)$ can be inferred. These resolvents can be combined to $(\neg x \vee y) = (\neg x \vee y \vee \neg c) \bowtie (\neg x \vee y \vee c)$. In a similar fashion, the other binary clause can be obtained: $(x \vee \neg y \vee \neg c) = (x \vee \neg c \vee \neg t) \bowtie (\neg y \vee \neg c \vee t)$ and $(x \vee \neg y \vee c) = (x \vee c \vee \neg f) \bowtie (\neg y \vee c \vee f)$. Now using these resolvents, we get $(x \vee \neg y) = (x \vee \neg y \vee \neg c) \bowtie (x \vee \neg y \vee c)$. A similar argument applies to XOR. □

4 Capturing Non-transitive HBR

For the following, given a CNF formula F and two literals l and l' that occur in F, we say that l' *dominates* l (or l' is a *dominator* of l) in F if there is a clause $C = (l \vee l_1 \vee \cdots \vee l_k) \in F_{\geq 3}$ such that $(\neg l_1), \ldots, (\neg l_k) \in \text{BCP}(F_2 \cup \{(l')\})$. In other words, l' dominates l in F if there is such a clause C for which each of the literals $\neg l_i$ are reachable from l' in $\mathsf{BIG}(F)$. This implies that by assigning $l' = 1$, unit propagation on F will assign $l = 1$ based on only F_2 and the clause C.

Example 1. Consider the formula $F = (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee d) \wedge (\neg b \vee e) \wedge (\neg c \vee d) \wedge (\neg c \vee e) \wedge (\neg d \vee \neg e \vee f)$. A part of $\mathsf{BIG}(F)$ with a hyperedge on the right showing the ternary clause $(\neg d \vee \neg e \vee f)$ can be illustrated as:



By assigning $a = 1$, unit propagation on F_2 and $(\neg d \vee \neg e \vee f) \in F_{\geq 3}$ will assign $d = 1$ and $e = 1$, and hence also $f = 1$. Thus a dominates f. The literal f has two other dominators: b and c, both of which are implied by a. ■

Given a CNF formula F and a literal l in F, the set of *non-transitive hyper binary resolvents* $\text{NHBR}(F, l)$ of F w.r.t. l is the set S of binary clauses arising from the following fixpoint computation. Let $\tau := \{l = 1\}$ and $S := \{\}$. Apply the following (non-deterministic) steps repeatedly until fixpoint:

1. While there is a unit clause $(x) \in \tau(F_2 \cup S)$, let $\tau := \tau \cup \{x = 1\}$.
2. If there is a unit clause $(y) \in \tau(F_{\geq 3})$ and literal l' with $\tau(l') = 1$ that dominates y in $F \cup S$, let $S := S \cup \{(\neg l' \vee y)\}$.

Step 1 corresponds to applying unit propagation under τ on the current set $F_2 \cup S$ of binary clauses. In step 2, it is checked whether a dominator of y has been assigned to true where y is part of a non-binary clause in F that is reduced to the unit (y) under τ. Notice that there is always at least one dominator for each $(y) \in \tau(F_{\geq 3})$, namely l; however, this is not in general the only dominator. Still, only one clause is added per execution of step 2.

Computing $\mathrm{NHBR}(F, l)$ using l as dominator was proposed in [16], while [2] discusses the use of alternative dominators. It should be noted that the above-defined construction algorithm is very similar to the one proposed in [1]. To our best understanding, the main difference is that our definition restricts step 2 to consider only units *in* $\tau(F_{\geq 3})$ in contrast to considering any units inferred by applying BCP on $\tau(F_{\geq 3})$.

In essence, the construction of $\mathrm{NHBR}(F, l)$ consists of applying lookahead on the literal l restricted to F_2, and checking for dominators w.r.t. non-binary clauses in F whenever a BCP fixpoint in reached. Notice that τ may become conflicting (i.e., both $l' = 1$ and $l' = 0$ would be assigned for some literal l') during the computation of $\mathrm{NHBR}(F, l)$. This implies that l is a failed literal, which can in practice be detected on-the-fly during the computation of $\mathrm{NHBR}(F, l)$. For the following analysis, we will always assume that l is not a failed literal.

We call a binary clause C a *non-transitive hyper binary resolvent* w.r.t. a CNF formula F if $C \in \mathrm{NHBR}(F, l)$ for some literal l in F. Given a CNF formula F, the procedure NHBR applies the following until fixpoint: while there is a non-transitive hyper binary resolvent $C \in \mathrm{NHBR}(F, l)$ w.r.t. F for some l, let $F := F \cup \{C\}$. A formula resulting from NHBR is denoted by $\mathrm{NHBR}(F)$. However, this fixpoint is not unique in general, and hence NHBR is not confluent, as will be shown in the following. Among other observations, we will also show that any $C \in \mathrm{NHBR}(F, l)$ for any F, l is indeed *non-transitive* in F, which implies that NHBR can *increase reachability* in the binary implication graph.

4.1 Understanding NHBR

Proposition 3. *For a CNF F and literal l, F is logically equivalent to $F \cup \mathrm{NHBR}(F, l)$.*

Proof. Any assignment that satisfies $F \cup \mathrm{NHBR}(F, l)$ also satisfies F. Now, assume that F is satisfiable, and fix an arbitrary truth assignment τ that satisfies F. Take an arbitrary clause $(\neg l' \vee y) \in \mathrm{NHBR}(F, l)$ with l' being a dominator of y. Notice that $l' \rightarrow y$, so $\neg y \rightarrow \neg l'$. So either $\tau(y) = 1$ or $\tau(y) = \tau(l') = 0$. Both satisfy $(\neg l' \vee y)$. Thus τ satisfies $\mathrm{NHBR}(F, l)$. □

Example 2. Let $F = (a \vee b) \wedge (a \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee e) \wedge (\neg b \vee c)$. We have $\mathrm{NHBR}(F, \neg a) = \{(a, d), (\neg b, e)\}$, which means that both of these non-transitive hyper binary resolvents can be added to F while maintaining logical equivalence. ■

The following proposition shows that all clauses in $\mathrm{NHBR}(F, l)$ for any literal l are indeed non-transitive in F.

Proposition 4. *For any CNF F, literal l, and clause $C \in \mathrm{NHBR}(F, l)$, we have that C is not transitive in F.*

Proof. Consider the first $C = (\neg l' \vee y) \in \mathrm{NHBR}(F, l)$ added to S during the computation of $\mathrm{NHBR}(F, l)$. By definition, l' dominates y in F (recall step 2 of the computation of $\mathrm{NHBR}(F, l)$), S being the empty set. Assume that C is transitive in F. It follows that there is a path from l' to y in $\mathsf{BIG}(F)$. However, by step 1 in the computation of $\mathrm{NHBR}(F, l)$, we would have $\tau(y) = 1$ after step 1, and hence $(y) \notin \tau(F_{\geq 3})$, and thus C would not be added to S.

The claim follows by induction using a similar argument for the $i + 1$ clause added to S assuming that the i clauses added before to S are not transitive in F. □

This implies that, in case NHBR can add clauses to a CNF formula F, NHBR will increase reachability in the implication graph of F.

Corollary 1. *If $\mathrm{NHBR}(F) \setminus F \neq \emptyset$, then it holds that there are two literals l, l' such that (i) there is a path in $\mathsf{BIG}(\mathrm{NHBR}(F))$ from l to l', and (ii) there is no path in $\mathsf{BIG}(F)$ from l to l'.*

However, as an additional observation, we note by adding a clause $C \in \mathrm{NHBR}(F, l)$ to F, some clauses in F_2 may become transitive in the resulting $F \cup \{C\}$.

Example 3. Consider the formula $F := (a \vee b) \wedge (a \vee c) \wedge (a \vee \neg b \vee d) \wedge (c \vee \neg d)$. Notice that $(a \vee d) \in \mathrm{NHBR}(F, \neg a)$. After adding $(a \vee d)$ to F, the clause $(a \vee c)$ is transitive in the resulting formula $F \cup \{(a \vee d)\}$. ■

The following clarifies the connection between hyper binary resolvents and non-transitive hyper binary resolvents: in essence, NHBR is a refinement of HBR that focuses on adding the most relevant hyper binary resolvents that improve reachability in the implication graph and hence can contribute to additional unit propagations.

Proposition 5. *Given a CNF formula F, and a hyper binary resolvent C w.r.t. F, it holds that C is transitive in F, or that $C \in \mathrm{NHBR}(F, l)$ for some literal l.*

Proof. Take an arbitrary hyper binary resolvent $C = (l \vee \neg l')$ w.r.t. a CNF formula F and let $D = (l \vee l_1 \vee \cdots \vee l_k)$ be the longest clause used in the hyper binary resolution rule to infer C. Clearly, if D is binary, then C is transitive. Now assume that $D \in F_{\geq 3}$. Because $(l \vee \neg l')$ is a hyper binary resolvent, unit propagation on $F_2 \cup \{(l')\}$ assigns all literals $l_1, \ldots, l_k$ to false. Assume that C is not transitive in F. In this case unit propagation on $F_2 \cup \{(l')\}$ will not assign l to true. Hence, after unit propagation on $F_2 \cup (l')$, $D \in F_{\geq 3}$ becomes the unit clause (l), and hence $(l \vee \neg l') \in \mathrm{NHBR}(F, l')$. □

As for the number of produced hyper binary resolvents, NHBR does not escape the quadratic worst-case, which, as we show, holds for all known implementations of HBR.

Proposition 6. *For CNF formulas over n variables,* NHBR *adds $\Omega(n^2)$ hyper binary resolvents in the worst-case. This holds even for formulas with $\mathcal{O}(n)$ clauses.*

Proof. There are $2n(n-1)$ different non-tautological binary clauses over n variables. So clearly NHBR adds only $\mathcal{O}(n^2)$ resolvents. As a worst-case example, consider the formula $F = (x_i \vee v) \wedge (x_i \vee w) \wedge (\neg v \vee \neg w \vee y_j)$ with $i, j \in \{1, \ldots, k\}$ having $2k+2$ variables and $3k$ clauses. Since all $(x_i \vee y_j) \in \mathrm{NHBR}(F, \neg x_i)$, NHBR will add $\Omega(k^2)$ resolvents. □

We will now address the question of confluence of NHBR.

Proposition 7. NHBR *is not confluent.*

Proof. Consider the formula $F := (a \vee b \vee c) \wedge (\neg b \vee c) \wedge (a \vee \neg d) \wedge (c \vee d \vee e) \wedge (d \vee \neg e)$. Notice that $(a \vee c) \in \mathrm{NHBR}(F, \neg c)$ and $(c \vee d) \in \mathrm{NHBR}(F, \neg d)$. Furthermore, $(c \vee d) \in \mathrm{NHBR}(F \cup \{(a \vee c)\}, \neg d)$, but $(a \vee c) \notin \mathrm{NHBR}(F \cup \{(c \vee d)\}, \neg c)$. Therefore, the resulting formula could only contain $(a \vee c)$ if this resolvent is added before $(c \vee d)$, The reason for the non-confluence in this example is that $(a \vee c)$ is transitive in $F \cup \{(c \vee d)\}$. □

Example 4. Recall step 2 of the computation of $\mathrm{NHBR}(F, l)$. While l is always guaranteed to be a dominator of $(y) \in \tau(F_{\geq 3})$, there can be other dominators as well (recall Example 1). In case there is a dominator $l' \neq l$, then it is preferable to add $(\neg l' \vee y)$ to S instead of $(\neg l \vee y)$ in the sense that $(\neg l' \vee y)$ is not transitive in $F \cup \{(\neg l \vee y)\}$, while $(\neg l \vee y)$ is transitive in $F \cup \{(\neg l' \vee y)\}$. Recall the formula F in Example 1. The dominators of f are $\neg a$, d, and e (recall Example 1), and b and c are implied by a. Hence $\mathrm{NHBR}(F, a) = \{(\neg a \vee f), (b \vee f), (c \vee f)\}$. Hence, instead of adding $(\neg a \vee f)$, one can add $(b \vee f)$ or $(c \vee f)$. ■

Although NHBR in itself is not confluent, interestingly, when combining NHBR with ELS and TRD, a unique fixpoint is reached (modulo variable renaming within literal equivalence classes). A similar observation has been previously made in [1, Theorem 1] for the combination of HBR and ELS alone without TRD.

Proposition 8. *For any CNF formula F,* NHBR *followed by the combination of* ELS *and* TRD *until fixpoint is confluent (modulo variable renaming).*

Proof. (sketch) Given any CNF formula F, the implication graph of $\mathrm{ELS}(F)$ is acyclic, and hence $\mathrm{TRD}(\mathrm{ELS}(F))$ is unique (modulo variable renaming). Now assume that there are two literals l, l' and clauses C, C' such that $C \in \mathrm{NHBR}(F, l)$ and $C' \in \mathrm{NHBR}(F, l')$. Assume that C' is transitive in $F \cup \{C\}$ and that C is not transitive in $F \cup \{C'\}$. If NHBR adds the clauses to F in the order C', C, TRD will afterwards remove the transitive C' from $F \cup \{C', C\}$, resulting in $F \cup \{C'\}$ to which NHBR would not add C. Finally, since NHBR can only increase reachability in the implication graph, NHBR will not re-introduce any previously added clauses that may have been afterwards removed by TRD. □

Example 5. As a concrete example, recall that the reason for the non-confluence in the proof of Proposition 7 is that $(a \vee c)$ is transitive in $F \cup \{(c \vee d)\}$. However, a unique result is obtained by applying TRD after NHBR.

5 Realizing Non-transitive HBR

Apart from the classical BCP that removes satisfied clauses and falsified literals, the variant $\mathrm{BCP}_{\mathrm{NHBR}}$ efficiently adds *non-transitive* hyper binary resolvents by prioritizing binary clauses during propagation. The fact that hyper binary resolution can be

achieved through unit propagation is due to [1] and has been extended in [2]. Another extension, called lazy hyper binary resolution (LHBR) [17] is discussed in Sect. 8.

The pseudo-code of $\mathrm{BCP}_{\mathrm{NHBR}}$ is shown in Fig. 1. Besides a formula F and a literal l, it takes a truth assignment τ (here interpreted as a *stack* of variable-value assignments) as input. For wellformedness, it is required that all assignments in τ are implied by $l = 1$ using only binary clauses. That is, all literals in τ can be reached from l in $\mathsf{BIG}(F)$.

```
    BCP_NHBR (formula F, truth assignment τ, literal l)
1     τ.push(l = 1)
2     while τ(F) contains unit clauses do
3       while (l') ∈ τ(F_2) do τ.push(l' = 1)
4       if (l'') ∈ τ(F_≥3) then F := F ∪ {(¬l ∨ l'')}
5     return ⟨F, τ⟩
```

Fig. 1. Pseudo-code of the $\mathrm{BCP}_{\mathrm{NHBR}}$ procedure.

First, the input literal l is set to true on the assignment stack τ (line 1). As long as unit clauses exist (line 2), propagation of binary clauses is prioritized (line 3). If there are only unit clauses left originating from $F_{\geq 3}$, then a random one is selected and converted into a non-transitive hyper binary resolvent (line 4). In the end, the resulting formula and extended assignment are returned (line 5).

In practice, implementing $\mathrm{BCP}_{\mathrm{NHBR}}$ can be expensive. To reduce the computational costs, [2] proposes two optimizations. The first, using alternative dominators is discussed in Sect. 4. The second is restricting computation to $C \in \mathrm{NHBR}(F, l)$ with $l \in \mathrm{RTS}(F)$ (i.e., starting only from literals that are roots in the implication graph). This restriction reduces the costs significantly. For FLE, starting only from literals $l \in \mathrm{RTS}(F)$, will not change the fixpoint [18,19]. Yet, this is not the case for NHBR.

Proposition 9. *By restricting* NHBR *to add only* $C \in \mathrm{NHBR}(F, l)$ *with* $l \in \mathrm{RTS}(F)$*, some non-transitive hyper binary resolvents will not be added.*

Proof. Consider formula $F = (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg c \vee d) \wedge (b \vee \neg c \vee \neg d)$. Notice that $(b \vee \neg c) \in \mathrm{NHBR}(F, c)$, while for all $l \in \mathrm{RTS}(F)$ holds that $\mathrm{NHBR}(F, l) = \emptyset$. □

In the following section, we will discuss an alternative technique, namely, *tree-based lookahead*, that can be used to efficiently compute $\mathrm{NHBR}(F)$ till fixpoint.

6 Tree-Based Lookahead

Tree-based lookahead originates from [3] but has not been properly described in the literature yet. It is a technique to reduce the computational cost to find failed literals and non-transitive hyper binary resolvents by reusing propagations. For some intuition about how this technique works, consider a CNF formula F which contains a binary clause $(\neg a \vee b)$ and several other clauses. Due to the presence of $(\neg a \vee b)$, we know that when propagating $a = 1$, b is forced to 1 as well as all variables that would have been forced by $b = 1$. It is possible to reuse the propagations of $b = 1$ (i.e., without rerunning

BCP), by assigning $a = 1$ afterwards *without* unassigning the forced variables. If there is another binary clause $(\neg c \vee b)$, then the effort of propagating $b = 1$ can additionally be shared with the effort of propagating $c = 1$ after backtracking over $a = 1$ and then assigning $c = 1$ without backtracking the assignments implied by $b = 1$.

This concept can be generalized by decomposing $\mathsf{BIG}(F)$ into in-trees: trees in which edges are oriented so that the root is reachable from all nodes (the root has out-degree 0 and other nodes have out-degree 1). For each implication $x \to y$ in the in-trees, y is assigned before x. Note that in-trees in the in-tree decomposition are almost never induced subgraphs, e.g. they are missing some edges; there are edges of $\mathsf{BIG}(F)$ that are not part of any in-tree, and even might connect two different in-trees.

The first step in tree-based lookahead is to create the in-trees, which is realized by the *getQueue* procedure, shown in Fig. 2. First queue Q is initialized and all cycles in $\mathsf{BIG}(F)$ are removed using ELS. Note that applying ELS once to F might produce new binary clauses by shrinking longer clauses, and even introduce new cycles. We thus have to run this process until completion.

Afterwards a random depth-first search is applied starting from the leafs of $\mathsf{BIG}(F)$. Notice that if $\neg l \in \mathrm{RTS}(F)$, l is a leaf. In the *enqueue* procedure, first l is added to Q followed by a recursive call for all literals that imply l and are not in the queue yet. The procedure ends adding the special element $\triangledown$ to Q that denotes that the algorithm should backtrack if that element is dequeued. The resulting Q contains each literal l in F exactly once, and for each literal occurring in Q the special element $\triangledown$ occurs once.

```
getQueue (F)
1  Q := {}
2  while ELS(F) ≠ F do
3     F := ELS(F)
4  foreach ¬l ∈ RTS(F) do
5     Q := enqueue(F, Q, l)
6  return Q
```

```
enqueue (F, Q, l)
1  Q.enqueue(l)
2  foreach (l ∨ ¬l') ∈ F2 do
3     if l' ∉ Q then
4        Q := enqueue(F, Q, l')
5  Q.enqueue(▽)
6  return Q
```

Fig. 2. Left: the *getQueue* procedure. Right: the *enqueue* sub-procedure.

The *TreeLook* algorithm (Fig. 3) uses the queue Q to compute failed literals and non-transitive hyper binary resolvents efficiently. After initialization (line 1 and 2), it dequeues elements from Q until it is empty (line 3 and 4). In case the current literal l is not $\triangledown$ (line 5), the decision level is increased by pushing $*$ on the assignment stack τ (line 6). If l is assigned to 0 or the current assignment τ falsifies F then the failed literal $(\neg l)$ is found (line 7). Otherwise, if l is still unassigned (line 8), then it is assigned to 1, followed by $\mathrm{BCP}_{\mathrm{NHBR}}$ prioritizing binary clauses, under which unit clauses that originate from non-binary clauses are transformed into a non-transitive hyper binary clause (line 9). If BCP results in a conflict, then a failed literal is found (line 10). Each time the element $\triangledown$ is dequeued, the algorithm backtracks one level, by popping elements from τ until it removes $*$ (line 11). Finally, the resulting F, simplified with failed literals and strengthened by non-transitive hyper binary resolvents (which may be trivially unsatisfiable (line 12)), is returned (line 13).

Example 6. Consider $F = (\neg a \vee \neg b) \wedge (b \vee \neg c \vee e) \wedge (b \vee c) \wedge (c \vee d) \wedge (a \vee \neg d \vee \neg e)$. NHBR can add two clauses to F: $\mathrm{NHBR}(F, b) = \{(b \vee e)\}$ and $\mathrm{NHBR}(F, c) = \{(c \vee \neg e)\}$. The

```
TreeLook (formula F)
1   τ := {}
2   Q := getQueue(F)
3   while Q is not empty do
4     l := Q.dequeue()
5     if l ≠ ▽ then
6       τ.push(∗)
7       if τ(l) = 0 or ∅ ∈ τ(F) then F := BCP(F ∪ {¬l})
8       else if τ(l) ≠ 1 then
9         ⟨F, τ⟩ := BCP_NHBR(F, τ, l)
10        if ∅ ∈ τ(F) then F := BCP(F ∪ {¬l})
11    else while τ.pop() ≠ ∗
12    if ∅ ∈ F then break
13  return F
```

Fig. 3. The *TreeLook* algorithm.

TreeLook (F) algorithm can find them as follows. Assume that the result of *getQueue* (F) is $Q = \{c, \neg b, a, \triangledown, \triangledown, \neg d, \triangledown, \triangledown, d, \neg c, \triangledown, \triangledown, \neg a, b, \triangledown, \triangledown\}$, visiting the leafs in the order c, d, $\neg a$. This Q partitions $\mathsf{BIG}(F)$ (see Fig. 4) in three in-trees by removing the dotted edge $\neg c \dashrightarrow b$. After initialization, τ is extended by pushing $*$ and $c = 1$. This does not result in any units. Now, τ is extended with $*$ and $b = 0$. The clause $(b \vee \neg c \vee e) \in F_{\geq 3}$ becomes unit. Therefore $(b \vee e)$ is added to F, which is unit (e) by construction under τ. Hence τ is extended by $e = 1$. Afterwards, $*$ and $a = 1$ are pushed to τ. No new units exist in $\tau(F)$ and the next element in Q is dequeued which is $\triangledown$. This causes popping $a = 1$ and $*$ from τ as the first backtracking step. The next element is also zero, which pops $e = 1$, $b = 0$, and $*$ from τ. Extending the shrunken τ by pushing $*$ and $d = 0$, does not result in any unit in $\tau(F)$. The first in-tree is now finished and the algorithm will pop all elements from τ due to the double $\triangledown$ element dequeued from Q. The NHBR $(c \vee \neg e)$ is found in the second in-tree: after $d = 1$ and $c = 0$, τ is extended by $b = 1$ and $a = 0$. Now $(a \vee \neg d \vee \neg e) \in F_{\geq 3}$ is unit $(\neg e)$ under τ. The third in-tree does not add any clause to F. Notice that after adding both binaries to F, $(b \vee c)$ becomes redundant (transitive) as well as $(b \vee \neg c \vee e)$ (subsumed). ■

7 Experiments

The *TreeLook* algorithm (Fig. 3) is implemented in the MARCHRW SAT solver [20]. The SAT Competition version of MARCHRW runs FLE until completion in each node of the search-tree. We slightly modified the code such that it runs NHBR until comple-

Fig. 4. $\mathsf{BIG}(F)$ in Example 6 before (left) and after (right) applying NHBR on F. Both graphs have three leafs: $\neg a$, c, and d. The dotted edge $\neg c \dashrightarrow b$ is not in the in-tree decomposition.

tion. In other words, lookahead runs until no new non-transitive hyper binary resolvent is found, instead of no new failed literal. The resulting version, called MARCHNH (benchmarks, sources and logs at `http://fmv.jku.at/treelook`), also has the ability to output the formula after preprocessing, so the result is similar to existing implementations of HBR, HYPRE [1] and HYPERBINFAST [2]. The experiments were done on a cluster of computing nodes with Intel Core 2 Duo Quad Q9550 2.8-GHz processors, 8-GB main memory, under Ubuntu Linux. Memory was limited to 7 GB and a timeout of 10 h was enforced for each run.

As benchmarks we used all 818 sequential circuits of the Hardware Model Checking Competition 2010 `http://fmv.jku.at/hwmcc10`. A miter was constructed from each circuit by connecting the inputs (and latches) of two copies of the same circuit, and by constraining outputs and next state functions to be pairwise equivalent. We used `aigmiter` for constructing the miters, and translated them to CNF with `aigtocnf`. Both tools are available from `http://fmv.jku.at/aiger`. Note that these benchmarks are trivial on the AIG level and can simply be solved by structural hashing. Actually, a non-optimized implementation of structural hashing needs less than 13 seconds for all 818 benchmarks, and less than half a second for the most difficult one (intel048 with 469196 variables and 1300546 clauses).

Running times for the hardest benchmarks using logarithmic scale are shown in Fig. 5: NHBR through tree-based lookahead (MARCHNH with TreeLook) can solve all of benchmarks on its own (i.e., without any additional search). Switching off tree-

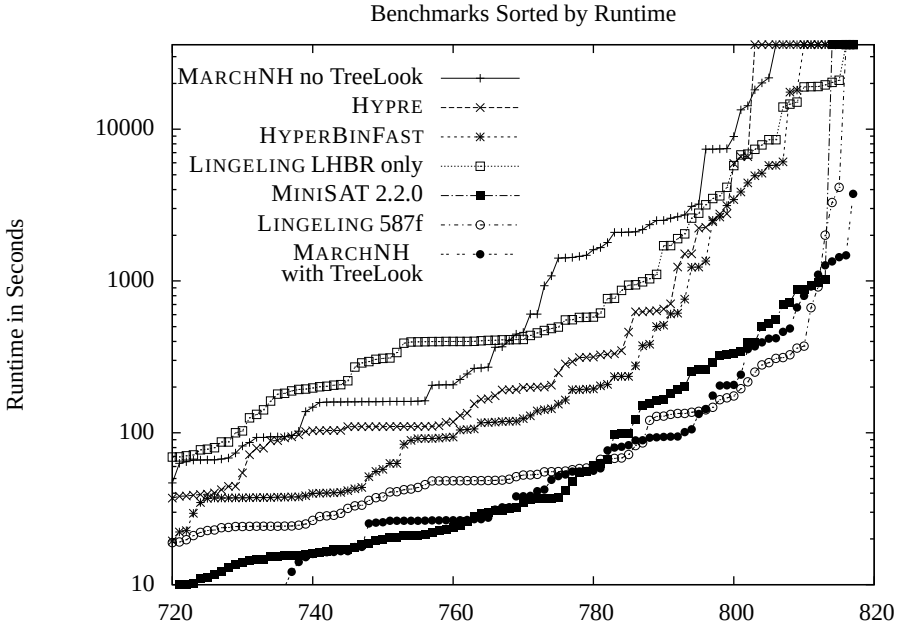


Fig. 5. Runtimes of CNF solving tools on 818 instances generated from HWMCC 2010. The plot starts at 750 because many instances could be solved easily. Notice that only LINGELING and MINISAT perform search. The other tools run NHBR till unsatisfiability is detected.

based lookahead (MARCHNH no TreeLook), i.e. always applying $\mathrm{BCP}_{\mathrm{NHBR}}(F, \tau, l)$ with $\tau = \emptyset$, the timeout is reached on eleven benchmarks and is two orders of magnitude slower. In between are the results of the previous implementations of HBR including LHBR (see next Sect. 8), which take much more time and memory, even though they use ELS. HYPRE hits the memory limit on 15 miters, HYPERBINFAST runs out of memory on eight, and LINGELING (LHBR only) runs out of time on two. Surprisingly state-of-the-art CDCL SAT solvers such as LINGELING 587F and MINISAT 2.2.0 can not solve some of the miters even within 10 hours of search (LINGELING could not solve two miters, MINISAT four).

Although not the main focus here, we also measured the effect of applying NHBR as a preprocessing technique for SAT Competition 2011 application instances. For a clean experiment, we compared plain Lingeling (no pre- and inprocessing) with and without NHBR. With NHBR, Lingeling solved 7 more instances.

8 Related Work and Existing Implementations

A version of the SAT solver PRECOSAT [17] submitted to the SAT Competition 2009 contained an algorithm for cheaply computing hyper binary resolvents *on-the-fly* during BCP in a standard CDCL solver on all decision levels. This method was called *lazy hyper binary resolution* (LHBR), and a preliminary version was implemented in PICOSAT [21] before. It has since then been ported to many other recent SAT solvers, including CIRCUS [22], LINGELING [23], and CRYPTOMINISAT [24]. Extensions of LHBR including a detailed empirical analysis of its benefits, can be found in [22].

The basic idea of LHBR is to restrict the implication graph, made of assigned literals and their forcing antecedents resp. reason clauses, to binary clauses. The implication graph is in general a DAG and the restriction to binary clauses turns it into a forest of trees, which we call *binary implication forest*. This allows us to save for each assigned variable the root of its binary implication tree. If a literal is implied by a non-binary clause, and all its antecedent literals in this clause are in the same tree, or equivalently they have the same root, a binary clause through LHBR is obtained. This can be checked by scanning the forcing non-binary clause, and checking whether all its variables, except the implied one, have the same root. If this is the case, the closest dominator of the antecedents can be computed as least-common ancestor in the tree.

The binary clause derived through LHBR is used as reason instead of the originally forcing non-binary clause, which extends the binary implication tree of the antecedents. It adds an edge from the dominator to the newly forced literal. To avoid adding too many transitive clauses, propagation over binary clauses is run until completion for all assigned literals before non-binary clauses are considered for propagation. This form of LHBR adds a negligible overhead to BCP, because checking for a common root among antecedent literals is cheap and only has to be performed if a non-binary clause becomes forcing. Thus, from the point of view of effectiveness, ease of implementation, and overhead, LHBR is comparable to on-the-fly *self*-subsumption [25]. One difference though is, that the former is implemented as part of BCP and the latter in the analysis algorithm for learning clauses from conflicts.

In practice we observed that the vast majority of binary clauses derived through LHBR are obtained during failed literal probing resp. lookahead at decision level 0

anyhow. Thus a simpler implementation similar to the one used in lookahead solvers *including tree-based lookahead* already gives the largest benefit without the need to store roots of the binary implication forest. The additional advantage of using LHBR even during search is to cheaply learn binary clauses at all decision levels, which are valid globally and can be added permanently. In lookahead solvers binary clauses learned through LHBR have to be removed during backtracking.

The competition version LINGELING 587f used in Sect. 7 uses LHBR during failed literal probing. This time-limited lookahead is one of the many implemented pre- resp. inprocessing techniques [26]. We patched LINGELING to run LHBR until completion (`http://fmv.jku.at/lingeling/lingeling-587f-lhbrtc.patch`) on these instances, but as shown in the experiments the run-times were much worse, even with (full) ELS and (time-limited) TRD.

Recursive Learning [27] and Stålmarck's method [12] work on circuits resp. on data structures (triplets) close to circuits and can easily be combined with structural hashing. This leads to an algorithm similar to congruence closure algorithms used in SMT solvers [28]. There are versions of both Stålmarck's method and Recursive Learning working directly on CNF [29,30]. In both cases only boolean constants are propagated and not equivalences as in the original method of Stålmarck. We conjecture that a combination of these CNF techniques with equivalence reasoning would also simulate structure hashing, but we are not aware of published results along this line.

9 Conclusions

We focused on understanding how non-transitive hyper binary resolvents can be efficiently exploited on the CNF-level. We explained how hyper binary resolution can be implemented through tree-based lookahead, which allows to simulate structural hashing on the CNF-level also in practice much more efficiently than previous CNF-level solutions. As a side-result, we believe our explanation of tree-based lookahead is of independent interest, providing an efficient way of implementing lookahead, which is important for example in the recently proposed *cube & conquer* approach [31].

The motivation for tree-based look-ahead was originally twofold. First, it provides an efficient implementation technique for failed literal probing during pre- and inprocessing [26]. This was the focus of this paper. Second, tree-based look-ahead can also be used to efficiently compute look-ahead heuristics, such as the number of clauses reduced to binary clauses after assuming and propagating a literal. It is unclear at this point whether the second motivation is really important, or whether other cheaper-to-compute metrics could also be used.

While our *TreeLook* implementation significantly improves over existing CNF-level approaches, there is still a large gap between the efficiency of circuit-level structural hashing and of using CNF reasoning alone for identifying equivalences. Future work consists of closing this gap further. As a final remark, as also pointed out by anonymous reviewers, it should be possible to reformulate tree-based lookahead for applying singleton arc consistency in CP and probing in MIP solvers.

Acknowledgements. We thank Donald Knuth for detailed comments and suggestions on a draft version of this paper.

References

1. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Proc. SAT 2003. Volume 2919 of LNCS., Springer (2004) 341–355
2. Gershman, R., Strichman, O.: Cost-effective hyper-resolution for preprocessing CNF formulas. In: Proc. SAT. Volume 3569 of LNCS., Springer (2005) 423–429
3. Heule, M.J.H., Dufour, M., van Zwieten, J., van Maaren, H.: March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In: SAT'04. Volume 3542 of LNCS. (2005) 345–359
4. Heule, M.J.H., van Maaren, H. In: Handbook of Satisfiability, Chapter 5: Look-Ahead Based SAT Solvers. IOS Press (2009) 155–184
5. Van Gelder, A.: Toward leaner binary-clause reasoning in a satisfiability solver. Annals of Mathematics and Artificial Intelligence **43** (2005) 239–253
6. Aho, A., Garey, M., Ullman, J.: The transitive reduction of a directed graph. SIAM Journal on Computing **1**(2) (1972) 131–137
7. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: Proc. AAAI, AAAI Press (2002) 613–619
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691
9. Kuehlmann, A., Krohm, F.: Equivalence checking using cuts and heaps. In: Proc. DAC, ACM (1997) 263–268
10. Williams, P.F., Andersen, H.R., Hulgaard, H.: Satisfiability checking using boolean expression diagrams. STTT **5**(1) (2003) 4–14
11. Abdulla, P., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT-solvers. In: TACAS. Volume 1785 of LNCS. (2000) 411–425
12. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck's proof procedure for propositional logic. Formal Methods in System Design **16**(1) (2000) 23–58
13. Billionnet, A., Sutter, A.: An efficient algorithm for the 3-satisfiability problem. Operations Research Letters **12**(1) (1992) 29–36
14. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: CP. Volume 1330 of LNCS., Springer (1997) 341–355
15. Anbulagan, Pham, D.N., Slaney, J.K., Sattar, A.: Old resolution meets modern SLS. In: Proc. AAAI. (2005) 354–359
16. Heule, M.J.H.: March: Towards a lookahead SAT solver for general purposes (2004) MSc thesis.
17. Biere, A.: P{re,i}coSAT@SC'09. In: SAT 2009 Competitive Event Booklet. (2009)
18. Boufkhad, Y.: Aspects probabilistes et algorithmiques du problème de satisfaisabilité (1996) PhD thesis, Univertsité de Paris 6.
19. Simons, P.: Towards constraint satisfaction through logic programs and the stable model semantics (1997) Report A47, Digital System Laboratory, Helsinki University of Technology.
20. Mijnders, S., de Wilde, B., Heule, M.J.H.: Symbiosis of search and heuristics for random 3-SAT. In: Proc. LaSh. (2010)
21. Biere, A.: (Q)CompSAT and (Q)PicoSAT at the SAT'06 Race (2006)
22. Han, H., Jin, H., Somenzi, F.: Clause simplification through dominator analysis. In: Proc. DATE, IEEE (2011) 143–148
23. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series TR 10/1, JKU, Linz, Austria (2010)
24. Soos, M.: CryptoMiniSat 2.5.0, SAT Race'10 solver description (2010)
25. Han, H., Somenzi, F.: On-the-fly clause improvement. In: Proc. SAT. Volume 5584 of LNCS., Springer (2009) 209–222

26. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Proc. IJCAR. Volume 7364 of LNCS. (2012) 355–370
27. Kunz, W., Pradhan, D.K.: Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. IEEE T-CAD **13**(9) (1994)
28. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C. In: Handbook of Satisfiability, Chpt. 26: SMT Modulo Theories. IOS Press (2009)
29. Marques-Silva, J., Glass, T.: Combinational equivalence checking using satisfiability and recursive learning. In: Proc. DATE. (1999)
30. Groote, J.F., Warners, J.P.: The propositional formula checker HeerHugo. J. Autom. Reasoning **24**(1/2) (2000) 101–125
31. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: HVC 2011 Revised Selected Papers. Volume 7261 of LNCS., Springer (2012) 50–65