Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas

Roman Gershman[1] Ofer Strichman[2]

[1] Computer Science, Technion, Haifa, Israel.
[2] Information Systems Engineering, Technion, Haifa, Israel.
Emails: gershman@cs.technion.ac.il ofers@ie.technion.ac.il

Abstract. We present an improvement to the HYPRE preprocessing algorithm that was suggested by Bacchus and Winter in SAT 2003 [2]. Given the power of modern SAT solvers, HYPRE is currently one of the only cost-effective preprocessors, at least when combined with some modern SAT solvers and on certain classes of problems. Our algorithm, although it produces less information than HYPRE, is much more efficient. Further, we implemented HYPERBINFAST as an *anytime* algorithm, which means that it can be interrupted before termination with a timeout and still get useful information. We also describe a heuristic for automatic setting of timeouts: it detects whether preprocessing produces enough interesting information, and if not, it terminates. Experiments on a large set of industrial Benchmark sets from previous SAT competitions show that HYPERBINFAST is always faster than HYPRE (sometimes an order of magnitude faster on some of the bigger CNF formulas), and achieves faster total run times, including the SAT solver's time. The experiments also show that HYPERBINFAST is cost-effective when combined with three state-of-the-art SAT solvers.

1 Introduction

Given the power of modern SAT solvers, most CNF preprocessing algorithms [5, 11] are mostly not cost-effective time-wise. Since these solvers are so effective in focusing on the important information in a given CNF, it is particularly challenging to find the right balance between the amount of effort invested in preprocessing and the quality of information gained, in order to positively impact the overall solving time.

One of the only preprocessors that succeeds to do so, at least when combined with *some* of the modern SAT solvers, is HYPRE [2]. An early version of HYPRE was implemented in 2CLS+EQ as an inference rule with impressive success (it solved instances that could not be solved by any other solver in the SAT'02 competition). There it was invoked in each node of the decision tree, hence making it part of the solver rather than a preprocessor. But given the very large SAT instances that now solvers need to cope with, this approach was too costly in practice. In [2] Bacchus and Winter improved the implementation of this algorithm and tried it very successfully as a preprocessor. Their experiments

show that in most cases there is a benefit in using this preprocessor prior to invoking a state-of-the-art solvers like Berkmin [6] and zChaff [8]. It seems, however, that on larger CNF formulas this is no longer true: running HYPRE on the benchmarks given in the SAT'04 competition, which were larger on average than the benchmark set attempted in [2], we noticed that it is not cost-effective in most cases.

HYPRE processes a CNF instance in three interconnected ways: 1) it adds binary clauses using Hyper Resolution [2] 2) it finds *failed literals* (variables with forced value) and propagates them, and 3) identifies equivalent variables by a traversal of the binary implications graph (a graph in which the edges correspond to binary clauses) and performs substitutions accordingly. Overall we refer to these actions as deriving *auxiliary information* from the CNF instance that simplifies its solution later by the SAT solver. One of the major reasons for the success of HYPRE comparing to previous preprocessors like 2-SIMPLIFY is that it is more selective in the information that it adds, and take less time to generate it.

In this paper we present HYPERBINFAST, which is similar to HYPRE, but improves it in two dimensions. First, to avoid cases in which preprocessing takes disproportional time, HYPERBINFAST is an *anytime* algorithm, which means that it produces meaningful auxiliary information even when interrupted before termination. This enables us to control the amount of preprocessing, and in particular, to allocate a certain percentage of the overall solving time to preprocessing. Second, HYPERBINFAST sacrifices some of the preprocessing power in order to terminate faster. We consider this as an adaptation to the new reality of SAT solvers being so efficient by themselves. Our experiments, reported in Section 5, show that these two improvements not only make it better in general than HYPRE, but also make HYPERBINFAST cost-effective when combined with state-of-the-art solvers like zChaff 2004, Siege_v1 and our own solver HaifaSat.

In the next section we describe the technical details of HYPRE relevant for understanding the difference from HYPERBINFAST. In Section 3 we describe HYPERBINFAST, in Section 4 we discuss the differences between the two algorithms and describe our solution to the problem of determining a time bound, and finally in Section 5 we present experimental results and our conclusions from them.

2 The HYPRE algorithm

2.1 Definitions and motivation

We begin with several definitions.

Definition 1 (Binary Implications graph). *Given a CNF formula φ with a set of binary clauses B, a* Binary Implications Graph *is a directed graph $G(V, E)$ such that $v \in V$ if and only if v is a literal in φ, and $e = (u, v)$ is an edge if and only if B contains a clause $(\overline{u}, v)$.*

A Binary Implications Graph allows us to follow implications through binary clauses. Note that for each binary clause (u, v), both $(\overline{u}, v)$ and $(\overline{v}, u)$ are edges in this graph (thus, the total number of edges in the initial graph, before further processing, is twice the size of B). For this reason we say that Binary Implications Graphs are *symmetric*.

Definition 2 (Binary Transitive Closure of a literal). *Given a literal v, a set of literals denoted by $BTC(v)$ is the* Binary Transitive Closure of v *if it contains exactly those literals that are implied by v through the Binary Implications Graph.*

Definition 3 (Failed literal). *A literal v is called a* Failed Literal *if setting its value to $TRUE$ and applying UP causes a conflict.*

Definition 4 (Propagation closure of a literal). *Given a non-failed literal u, a set of literals denoted by $UP(u)$ is the* Propagation closure of u *if it contains exactly those literals that are implied through UP by u in the given CNF (not only the binary clauses).*

It is easy to see that $BTC(v) \subseteq UP(v)$ for every literal v, because $UP(v)$ is not restricted to what can be inferred from binary clauses. Note that $v \in UP(u)$ implies that $u \rightarrow v$ and hence $\overline{v} \rightarrow \overline{u}$, but it is not necessarily the case that $\overline{u} \in UP(\overline{v})$, due to the limitations of UP. For example, in the set of clauses $(\overline{x} \vee y), (\overline{x} \vee z), (\overline{y} \vee \overline{z} \vee w)$, it holds that $x \rightarrow w$ and hence $\overline{w} \rightarrow \overline{x}$, but UP detects only the first direction. It disregards $\overline{w} \rightarrow \overline{x}$ because $\overline{w}$ does not invoke any unit clause. Hence, UP lacks the symmetry of Binary Implications Graphs.

Definition 5 (The HyperBinRes Hyper resolution rule). *The HyperBinRes inference rule:*

$$\frac{(l_1, \ldots, l_n) \quad (\bar{l}_1, l), \ldots, (\bar{l}_{n-1}, l)}{(l, l_n)} \qquad \textit{for } n \geq 2$$

HyperBinRes is a hyper resolution rule (resolution from more than two clauses). It is possible to compute the HyperBinRes *closure* (add all possible clauses according to this rule) in polynomial time, by analyzing the binary sub-theory of the formula, and invoking UP.

Hyper resolution can be thought of as a shortcut to a long sequence of standard resolution steps, that results only in the last clause in this sequence rather than in all intermediate results. It allows to retrieve much more information than otherwise possible by pure binary reasoning, while staying polynomial.

In general, any sequence of implications $l_1 \rightarrow \ldots \rightarrow l_n$ on the Binary Implications Graph that does not end with a failed literal, can potentially lead to hyper resolution through a standard application of UP on the given formula (including all clauses). The following lemma proves this:

Lemma 1. *Given a CNF formula F and a set $UP(v)$ as defined before, there is a sequence of HyperBinRes derivations for every literal $u \in UP(v)$ that proves $(\overline{v}, u)$.*

Proof. By induction on the literals that are propagated with UP. First, UP is seeded by v itself and, of course, $v \in UP(v)$. Clause $(\overline{v}, v)$ is universally true in the trivial way.

Suppose now that the clause $(l_1, \ldots, l_n, u)$ propagates u and the lemma holds for all literals propagated before u. It must be that l_i is FALSE for $1 \leq i \leq n$. Since all propagated literals are a result of setting v to TRUE, then it must be that $\bar{l}_i \in UP(v)$. We have by induction that there exists a sequence S_i of HyperBinRes derivations that proves $(\overline{v}, \bar{l}_i)$. Now, we use HyperBinRes again to derive $\frac{(\overline{v},\bar{l}_1),\ldots,(\overline{v},\bar{l}_n),(l_1,\ldots,l_n,u)}{(\overline{v},u)}$ and add it to the sequence $S_1 \ldots S_n$, which results in $(\overline{v}, u)$. □

Hyper Resolution can solve the problem of lack of symmetry in UP. In the example above, in which UP cannot detect that $\overline{w} \rightarrow \overline{x}$, applying Hyper Resolution on the given clauses produces the new binary clause $(\overline{x} \vee w)$. Clearly, now both $w \in UP(x)$ and $\overline{x} \in UP(\overline{w})$. In addition, realizing this new clause as a syntactic primitive allows algorithms that work on Binary Implications Graphs to use the fact that $x \rightarrow w$ or $\overline{w} \rightarrow \overline{x}$. For example, an algorithm that looks for equivalences between literals searches for cycles in this graph, and this new clause can lead to additional such cycles.

Each added binary clause (recall that this corresponds to two edges in the Binary Implications Graph) adds implicants not only to $BTC(v)$ for some v, but also to $UP(v)$ because of the symmetry discussed above of binary clauses. The goal of HYPRE is to add such clauses until both of these sets are equal, or, in other words:

> Goal: *Build a Binary Implications Graph such that $BTC(v) = UP(v)$ for every literal v.*

2.2 The HYPRE algorithm

HYPRE is a recursive algorithm, which processes a node v in the Binary Implications Graph only after returning from processing its children, i.e. in a *post-order*. It then performs UP in order to compute the propagation closure of v unless v is a Failed Literal. If v is a Failed Literal then a new unit clause $(\overline{v})$ is added to the formula, and unit propagation is applied. Otherwise, the set $NewDescendants := UP(v) \setminus BTC(v)$ is computed. For each literal $u \in NewDescendants$, a new binary clause $(\overline{v}, u)$ is added to the formula and $NewDescendants$ is updated to be $NewDescendats \setminus BTC(u)$ (this is an optimization, aimed at reducing *forward edges*, i.e., edges to nodes to which we already have a path from v). When HYPRE leaves v, it is guaranteed that either v is a Failed Literal or $UP(v) = BTC(v)$.

We call a node v *strong* if HYPRE already concluded that $BTC(v) = UP(v)$ with respect to the current formula. There is nothing to be done for strong nodes before the formula changes again. A *weak* node is a node that is not strong. Note that it is invariantly true that if a node v is weak, then all of its ancestors are also weak. HYPRE begins by marking all nodes as weak, and then gradually processes them and changes their marking to strong. When a binary clause $(\overline{v}, u)$ is added

as described above, $\overline{u}$ and its ancestors have to be marked as weak, which means that HYPRE has to process them again. Note that v should not be marked as weak, since HYPRE currently processes it.

To detect equivalent literals, occasionally HYPRE searches for Strongly Connected Components (SCC) in the graph. It is easy to see that literals in a cycle are equivalent. Each SCC is replaced with a single node (a 'representative literal'), not only in the binary clauses, rather in the entire formula. This operation, as well as unit propagation (when a new unit clause is added), can simplify clauses (shorten them) which means that HYPRE needs to reconsider nodes that were already visited before. When an n-ary clause $(l_1, \ldots, l_n)$ is shortened to, e.g., $(l_1, \ldots, l_k), 2 \leq k < n$, then the literals $\bar{l}_1 \ldots \bar{l}_k$ and all their ancestors are marked as weak, since further processing of these nodes may lead to more hyper resolutions given the new shortened clause. A more elaborated justification of this step can be found in [2].

The main computational cost of HYPRE is due to the need to perform UP on each node at least once, but on average many more times, due to the iterative nature of the algorithm. It is still far more efficient than previous approaches like 2-SIMPLIFY and 2CL_SIMP that stored the full transitive closure of the binary sub-theory and are therefore incapable of handling large instances.

HYPRE also preserves an interesting optimality property: If $y \in UP(x) \cap UP(z)$ and $x \xrightarrow{*} z$ (read: there is a path from x to z in the binary Implications Graph), then HYPRE guarantees, by the post-order in which it progresses, that it adds the edge $z \rightarrow y$ and not $x \rightarrow y$. The former is a stronger implication, because from this implication it is possible to infer the latter implication through the Binary Implications Graph.

3 The HYPERBINFAST algorithm

As stated in the introduction, our algorithm HYPERBINFAST builds upon and improves HYPRE in two dimensions: it is capable of giving useful auxiliary information even when stopped before termination, and it is more efficient for the price of generating less information. Our experiments, as shown in Section 5, prove that this shifting of emphasis is worth while: although the SAT solving time after HYPRE can be smaller, the total time is typically larger.

Algorithm 1 gives a bird-eye view of HYPERBINFAST. HYPERBINFAST iterates over all root nodes in the Binary Implications Graph ($roots(v)$ denotes the set of all ancestor roots of v in such a graph). It has two main stages. In the first stage (lines 4 - 5) it iteratively finds equal literals (by detecting SCCs and unifying their vertices to a single 'representing literal'), propagates unit clauses, and simplifies the clauses in the formula. Simplification in this context corresponds to substituting literals by their representative literal in all clauses (not only binary), removing literals that are evaluated to FALSE and removing satisfied clauses. The simplification may result in shortening of some n-ary clauses to binary clauses, which change the Binary Implications Graph. In line 6 we perform a restricted version of what HYPRE does in such cases: while HYPRE

Algorithm 1

```
HyperBinFast
Mark all root vertices as weak;
while there are weak roots, unit clauses, or binary cycles do
    while there are unit clauses or binary cycles do
        Detect all SCCs and collapse each one of them to a single node;
        Propagate all unit clauses and simplify all clauses accordingly;
        For each new binary clause (u, v) mark as weak roots(ū) and roots(v̄);
    end while
    Choose a weak root node v;
    FailedLiteral = FastVisit (v);
    Undo assignments caused by BinaryWalk and clear nQueue;
    if FailedLiteral ≠ undefined then        ▷ FailedLiteral holds a failed literal
        Add unit clause (¬FailedLiteral);
    end if
    Mark v as strong;
end while
```

Algorithm 2

```
procedure BinaryWalk (Literal t, Antecedent clause C)
    if value(t)=True then
        return undefined;
    end if
    if value(t)=False then
        return t̄;                    ▷ a failed literal which was marked before
    end if
    value(t) ← TRUE;
    antecedent(var(t)) ← C;
    Put t on assignment stack;
    Put t into nQueue;
    for each binary clause (t̄, u) in 2-CNF sub-theory do
        res ← BinaryWalk (u, (t̄, u));
        if res ≠ undefined then
            return res;
        end if
    end for
    return undefined;
end procedure
```

Algorithm 3

```
procedure FastVisit (Literal v)
    res ← BinaryWalk (v, NULL);
    if res ≠ undefined then
        return res
    end if
    while !nQueue.empty() do
        Literal p ← nQueue.pop_front();
        for each n-ary clause ∈ watched(p) do                    ▷ n > 2
            if clause is conflict then
                Literal fUIP ← FindUIP (clause);
                return fUIP;
            else if clause is unit then
                Literal toLit ← undefined literal from clause.
                Literal fromLit ← FindUIP (clause \ {toLit});
                Add clause (¬fromLit, toLit)
                Mark roots(¬toLit) ∪ roots(fromLit) as weak
                res ← BinaryWalk(toLit, (¬fromLit, toLit));
                if res ≠ undefined then
                    return res
                end if
            end if
        end for
    end while
    return undefined;
end procedure
```

Algorithm 4

```
procedure FindUIP (Literal set S)
Require:  implication graph is binary only.
Ensure:   res is first UIP of S
    mark all variables in S;
    count ← |S|;
    while count > 1 do
        v ← latest marked variable in the assignment stack
        unmark v and decrease count by one.
        Let (u, L) be antecedent clause of v, s.t. var(L) = v.
        if var(u) not marked then
            mark var(u) and increase count by one.
        end if
    end while
end procedure
res ← last marked literal in assignment stack.
unmark var(res);
return res;
```

marks as weak all ancestor nodes, HyperBinFast only marks root ancestor nodes. Further, while Hypre invokes this process every time an n-ary clause is being shortened, HyperBinFast only does so for clauses that become binary. The reduced overhead due to these changes is clear. In the second stage (line 9), we invoke FastVisit for some weak root node, a procedure that we will describe next. FastVisit can change the graph as well, so HyperBinFast iterates until convergence.

Computing the Binary Transitive Closure Before describing FastVisit, we concentrate on the auxiliary function BinaryWalk, which FastVisit calls several times. The goal of BinaryWalk is to mark all literals that are in TBC(v) or return a failed literal, which can be either v itself or some descendant of v. It also updates a queue, called *nQueue* with those literals in TBC(v) for future processing by FastVisit. BinaryWalk performs DFS from a given literal on the Binary Implications Graph. In each recursive-call, if t is already set to FALSE (i.e. $\bar{t}$ is already set to TRUE in the current call to FastVisit), it means that there is a path in the binary implication graph from $\bar{t}$ to t, and hence $\bar{t}$ is a failed literal. This is a direct consequence of the following lemma:

Lemma 2. *In a DFS-traversal on a Binary Implications Graph from a literal u that marks all nodes it visits, if when visiting a node t another node $\bar{t}$ is already marked, and this is the* first time *such a 'collision' is detected, then $\bar{t} \xrightarrow{*} t$.*

Proof. Falsely assume that there is no path $\bar{t} \xrightarrow{*} t$. The fact that traversal from u leads to marking of both $\bar{t}$ and t, implies that $u \xrightarrow{*} \bar{t}$ and $u \xrightarrow{*} t$. By the symmetry of Binary Implications Graphs, there is also a path $\bar{t} \xrightarrow{*} \overline{u}$. Since the DFS traversal visits $\bar{t}$ before t then it must visit $\overline{u}$ before leaving $\bar{t}$. Moreover, it cannot visit t before leaving $\bar{t}$ because otherwise this would contradict our assumption that no such path exists. This leads to a collision between u and $\overline{u}$, before a collision is detected between t and $\bar{t}$, which contradicts our assumption that the latter was detected first. Hence, $\bar{t} \xrightarrow{*} t$. □

When BinaryWalk detects such a failed literal it returns $\bar{t}$ all the way out (due to lines 14-15) and back to FastVisit and then to HyperBinFast.

The other case is when t does not have a value yet. In this case BinaryWalk sets it to TRUE and places it in *nQueue*, which is a queue of literals to be propagated later on by FastVisit. It also places t in the (global) assignment stack, and stores for $var(t)$ its antecedent clause (the clause that led to this assignment), both for later use in FindUIP.

¿From Binary Transitive Closure to Propagation Closure We now describe FastVisit. Recall that FastVisit is invoked for each root node in the Binary Implications Graph. FastVisit combines Unit Propagation with Binary Learning based on single assignments, i.e. learning of new clauses by propagating a single decision at a time. It relies on the simple observation that if $u \in UP(v)$ then $v \rightarrow u$. It is too costly to add an edge for every such pair v, u, because this

corresponds to at least computing the transitive closure [2]. Since our stated goal is to form a binary graph in which $UP(v) = BTC(v)$ for each root node, it is enough to focus on a vertex u only if $u \in UP(v)$ but $u \notin BTC(v)$. Further, given such a vertex u, although adding the edge $v \rightarrow u$ achieves this goal, we rather find a vertex w, a descendant of v that also implies u, in the spirit of the First Unique-Implication-Point (UIP) scheme that is used by most modern SAT solvers. The FindUIP function (see Algorithm 4) called by FastVisit can in fact be seen as a variation of the standard algorithm for finding first UIPs [9]: unlike the standard usage of such a function in analyzing conflicts, here there are no decision levels and the clauses are binary. On the other hand it can receive as input an arbitrary set of assigned literals, and not just a conflict clause. In contrast to Hypre, which performs unit propagation from each node in a post order, HyperBinFast only propagates from the roots, and the edges that it adds depend on the specific DFS run it performs. It therefore cannot guarantee the optimality property discussed in the end of the previous section. Invoking FindUIP attempts to compensate on this fact, but it cannot guarantee it.

In line 6 FastVisit starts to process the literals in $nQueue$. For each literal p in this queue, it checks all the n-ary clauses ($n > 2$) watched by p. As usual, each such clause can be of interest if it is either conflicting or unit. If it is conflicting, then FastVisit calls FindUIP, which returns the first UIP causing this conflict. This UIP is a failed literal and is returned to HyperBinFast, which adds its negation as a unit clause in line 12. If the processed clause is a unit clause, the unassigned literal, denoted by $toLit$, is a literal implied by v that is not in $BTC(v)$ (otherwise it would be marked as TRUE in BinaryWalk). In other words, $toLit \in UP(v)$ and $toLit \notin BTC(v)$, which is exactly what we are looking for. At this point we can add a clause $(\overline{v}, toLit)$ but rather we call FindUIP, which returns a first UIP denoted by $fromLit$. The clause $(\overline{fromLit}, toLit)$ is stronger than $(\overline{v}, toLit)$ because the former also adds the information that $\overline{t}olit \rightarrow fromlit$. Note that this is an unusual use of this function, because *clause* is not conflicting. Because the addition of this clause changes the Binary Implications Graph, we need to mark as weak all the ancestor nodes of $fromLit$ and of $\overline{toLit}$, and to continue with BinaryWalk from $toLit$. This in effect continues to compute $BTC(v)$ with the added clause.

4 discussion

4.1 Differences from Hypre

One of the differences between fast and Hypre can be demonstrated with the following example. Suppose we have a direct sub-graph with n vertices, rooted at a node r. Hypre goes recursively over all descendants of r and apply UP at each one of them and finds *all edges* that are needed to achieve the stated goal of making the binary graph represent the propagation closure of r and its descendants. There can be n propagations, which can many times find only a very small number of edges, sometimes even zero such edges. Moreover, it can also generate edges which connect descendants of r between them. HyperBinFast,

on the other hand, applies only one UP at the root r, and finds *all literals* that should have new edges leading to them. It will not connect existing descendants of r but concentrate on those which do not have any path from r to them.

HYPRE generally adds more information than HYPERBINFAST. For example, if two descendants of v, p and s prove a new literal t, then HYPERBINFAST finds only one of them, e.g. it could add $p \rightarrow t$. In other words, it finds only one edge for each UP-implied literal. Also, while FINDUIP tries to find the best binary clauses to add, it does not guarantee optimality. It could add $p' \rightarrow t$, where p' is some literal between v and p.

HYPERBINFAST can be seen as a restricted version of DPLL with non-conflict learning. Indeed, it "decides" on the root variables and never goes below decision level 1. Also, it learns a new clause every time an implication by an n-ary clause occurs. It is possible to change HYPERBINFAST so it uses decision heuristics other than just choosing the roots. We leave this option for future research.

4.2 Bounding the runtime of HYPERBINFAST

Given that HYPERBINFAST is an anytime algorithm, there are various strategies to decide when to stop it. The most naive method, of course, is to use a time-limit. While being extremely simple, it still enables us to balance between the preprocessing and the solving stage, according to the efficiency of the SAT solver. One somewhat unexpected disadvantage of this method is that it can lead to a nondeterministic solving process and run time[1]. The reason for this is the imperfection of measuring time: differences in mili-seconds as measured by the computer clock can stop the preprocessor in different stages and hence lead to a different instance for the SAT solver. We therefore developed a heuristic function that decides when to stop the preprocessor according to its progress. In particular, it measures the number of deduced unit clauses and equivalent literals, versus decaying rate of the weak nodes. This way we do not allow HYPERBINFAST to work for a long time without producing evidence of its efficiency. If the weak nodes are not decreased fast enough comparing to what is expected from the number of reduced variables, HYPERBINFAST is stopped and its output is forwarded to the SAT solver. HYPERBINFAST performs this check every constant number of invocations of FASTVISIT. There are numerous possible heuristics to perform this check. Currently our heuristic, which is still under development, is better or equally good as a fixed time-out of 300 seconds, although in a few instances it is worse (it stops the preprocessing too early). Altogether for the set of benchmarks reported in the next section the total solving time of the automated and fixed timeouts are comparable. We believe that with further experiments and tuning we will be able to make this technique dominant over a fixed timeout strategy.

[1] It is the policy of the SAT competition's organizers to not accept non-deterministic solvers.

5 Experiments, conclusions and directions for future research

We ran HyperBinFast and the original preprocessor Hypre [1] with several SAT solvers: siege_v1 [7], zChaff 2004 and our experimental SAT solver HaifaSat. We do not present results for the latest version of Siege, siege_v4, because we do not know if this undocumented solver already uses similar learning rules internally. Nevertheless, we ran siege_v4 with both Hypre and HyperBinFast and saw that it is not cost-effective to run either one of them. The overhead of HyperBinFast, however, was much smaller than that of Hypre.

Table 1 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux. The benchmark set is comprised of 165 industrial instances used in various SAT competitions. In particular, *fifo8, bmc2, CheckerInterchange, comb, f2clk, ip, fvp2, IBM02 and w08* are hard industrial benchmarks from SAT02; *hanoi* and *hanoi03* participated in SAT02 and SAT03; *pipe03* is from SAT03 and *01_rule, 11_rule_2, 22_rule, pipe-sat-1-1, sat02, vis-bmc, vliw_unsat_2.0* are from SAT04 [10, 3, 4]. The number in brackets for each benchmark set denotes the number of instances. The timeout for each instance was set to 3000 seconds. When relevant, the timeout for the preprocessor itself was set to 300 seconds and the timeout for the SAT solver was dynamically reduced to 3000 minus the time spent during preprocessing. All times in the table include preprocessing time when relevant. We count each failure as 3000 seconds as well.

The table shows that:

1. HyperBinFast helps each of the tested solvers to solve more instances in the given time bound.
2. When the instance is solvable without HyperBinFast, still HyperBinFast typically reduces the overall run time.
3. Whenever HyperBinFast does not help, its overhead in time is relatively small.
4. It is very rare that an instance can be solved without HyperBinFast but cannot be solved with HyperBinFast.
5. On average, the total gain in time is about 20-25%.

Table 2 compares the performance of Hypre and HyperBinFast. Since Hypre is not implemented as an anytime algorithm, and as a preprocessor it always takes more time than HyperBinFast (empirically), its only advantage when compared to HyperBinFast can be that it produces better information that compensates on the extra preprocessing time. In order to test this possibility we ran Hypre with a timeout of 3000 seconds and the SAT solver with a timeout of 3000 seconds minus the time spent by Hypre. In other words we compared three configurations for each SAT solver: plain SAT, SAT solver+Hypre, and SAT solver+HyperBinFast, all with a global timeout of 3000 seconds per instance. The time policy of HyperBinFast was left as in the previous experiment. Table 2 compares these configurations with both HaifaSat and Siege_v1. The times include both the preprocessing and the SAT solving run times. The table shows

that sometimes HyperBinFast is not 'strong' enough (it does not simplify the formula enough), so the SAT solver fails on the corresponding instance but succeeds after applying Hypre. Nevertheless, the total time is always smaller with HyperBinFast. Moreover, it can be seen that with HaifaSat, Hypre is not cost-effective, neither in the total number of failures or the total run time, while HyperBinFast reduces HaifaSat's failures by 35% and reduces its total solving time by 25%.

More data: For the above benchmark, it took HaifaSat 97,909 seconds after HyperBinFast and only 54,567 seconds after Hypre, which indicates that indeed the quality of the CNF generated by Hypre is better, as expected. But these numbers may mislead because, recall, the timeouts for the two preprocessors are different, which, in turn, is because HyperBinFast is an anytime algorithm. In Table 3 we list several benchmarks for which both preprocessors terminated before their respective timeouts, together with the time it took the preprocessor and then HaifaSat to solve them. To the extent that these instances are representative, it can be seen that typically the solving time is longer after HyperBinFast, but together with the SAT solver time it is more cost effective than Hypre.

Conclusions and directions for future research. Preprocessing can be cost-effective when combined with modern SAT solvers, as is evident from our experiments with 165 industrial CNF instances from previous SAT competitions. We pointed to two directions for future research: develop more efficient dynamic strategies for determining the amount of time spent for preprocessing, and make preprocessing *decide* on the set of variables from which it begins its traversal of the Binary Implications Graph (and not just choose all the root nodes as we do now). This concept can be generalized to preprocessing in general: while SAT solvers focus on the *semantics* of the formula, that is, they attempt to find the 'important' variables, preprocessors focus on the *syntactical* characteristics of the formula, and are therefore much more sensitive to its size. Hence, attempting to build a *semantic preprocessor* seems like a worth while direction to pursue next: the CNF instances that are hard to solve with modern solvers become larger every year, so becoming less affected by their sheer size seems like the only way for preprocessors to stay in the game.

References

1. http://www.cs.toronto.edu/∼fbacchus.
2. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT 2003*, volume 2919 of *Lect. Notes in Comp. Sci.*, pages 341–355, 2003.
3. D. Le Berre and L. Simon. The essentials of the sat'03 competition. In A. Tacchella E. Giunchiglia, editor, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, volume Lecture Notes in Computer Science 2919, pages 452–467. June 2003.

SAT solver →	HaifaSat				Siege_v1				zChaff 2004			
Preprocessor →	—		H-B-Fast		—		H-B-Fast		—		H-B-Fast	
Benchmark:	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	**7,379**	0	20,730	4	**11,408**	1	20,779	4	19,196	5
11_rule_2(20)	22,975	6	**7,491**	0	29,303	8	**17,733**	2	36,042	10	**27,500**	8
22_rule(20)	27,597	8	**22,226**	5	31,839	10	29,044	9	31,279	9	**25,377**	6
bmc2(6)	1,262	0	**81**	0	3,335	1	**85**	0	3,174	1	**83**	0
CheckerI-C(4)	**682**	0	902	0	4,114	0	**3,541**	0	816	0	**703**	0
comb(3)	4,131	1	4,171	1	5,679	1	6,027	1	6,363	2	6,237	2
f2clk(3)	4,059	1	4,060	1	6,105	2	6,063	2	6,090	2	6,047	2
fifo8(4)	1,833	0	**554**	0	5,555	1	**2,420**	0	5,206	1	**3,390**	1
fvp2(22)	1,995	0	2,117	0	1,860	0	2,009	0	7,078	0	**3,830**	0
hanoi(5)	**131**	0	285	0	**357**	0	1,231	0	**2,151**	0	2,435	0
hanoi03(4)	**427**	0	533	0	6,026	2	6,028	2	6,022	2	6,016	2
IBM02(9)	**3,876**	0	5,070	0	10,442	4*	**7,881**	0	9,596	3	**8,132**	1
ip(4)	203	0	**172**	0	630	0	**548**	0	**1,065**	0	3,538	1
pipe03(3)	1,339	0	1,266	0	2,006	0	**1,275**	0	4,106	1	**1,254**	0
pipe-sat-1-1(10)	**3,310**	0	5,147	0	**2,445**	0	5,249	0	**4,568**	0	8,950	0
sat02(9)	17,330	4	**14,797**	4	24,182	7	**18,843**	5	23,632	7	**20,333**	6
vis-bmc(8)	13,768	3	**10,717**	2	10,449	2	**6,989**	1	18,358	6	**13,389**	4
vliw_unsat_2(8)	19,425	5	19,862	6	16,983	6*	17,891	6*	21,867	7	21,364	6
w08(3)	2,681	0	**1,421**	0	4,387	1	**1,711**	0	4,316	1	**2,223**	0
Total(165)	146,194	30	**108,251**	19	186,426	49	**145,978**	29	212,508	56	**179,997**	44

Table 1. Run-times (in seconds) and failures (denoted by 'F') for various SAT solvers with and without HyperBinFast. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded. Failures denoted by * are partially caused by bugs in the SAT solver.

SAT solver →	HaifaSat						Siege_v1					
Preprocessor →	—		HYPRE		H-B-FAST		—		HYPRE		H-B-FAST	
Benchmark:	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	10,758	1	**7,379**	0	20,730	4	**5,318**	0	11,408	1
11_rule_2(20)	22,975	6	21,247	0	**7,491**	0	29,303	8	20,178	1	**17,733**	2
22_rule(20)	27,597	8	**16,825**	2	22,226	5	31,839	10	**17,510**	3	29,044	9
bmc2(6)	1,262	0	163	0	**81**	0	3,335	1	156	0	**85**	0
CheckerI-C(4)	**682**	0	989	0	902	0	4,114	0	**2,492**	0	3,541	0
comb(3)	4,131	1	4,056	1	4,171	1	5,679	1	**4,439**	1	6,027	1
f2clk(3)	4,059	1	**3,448**	1	4,060	1	6,105	2	**5,078**	1	6,063	2
fifo8(4)	1,833	0	1,756	0	**554**	0	5,555	1	**1,159**	0	2,420	0
fvp2(22)	1,995	0	3,288	0	2,117	0	1,860	0	2,431	0	2,009	0
hanoi(5)	131	0	119	0	285	0	357	0	802	0	1,231	0
hanoi03(4)	**427**	0	979	0	533	0	6,026	2	6,014	2	6,028	2
IBM02(9)	**3,876**	0	11,072	3	5,070	0	10,442	4	12,653	3	**7,881**	0
ip(4)	203	0	365	0	**172**	0	630	0	**349**	0	548	0
pipe03(3)	1,339	0	1,809	0	1,266	0	2,006	0	1,822	0	**1,275**	0
pipe-sat-1-1(10)	**3,310**	0	27,130	10	5,147	0	2,445	0	30,029	10	5,249	0
sat02(9)	17,330	4	16,669	4	**14,797**	4	24,182	7	17,662	4	18,843	5
vis-bmc(8)	13,768	3	10,139	2	10,717	2	10,449	2	**5,715**	0	6,989	1
vliw_unsat_2(8)	19,425	5	20,421	6	19,862	6	16,983	6	20,375	6	17,891	6
w08(3)	2,681	0	2,899	0	**1,421**	0	4,387	1	2,726	0	**1,711**	0
Total(165)	146,194	30	154,132	30	**108,251**	19	186,426	49	156,910	31	145,978	29

Table 2. Run-times (in seconds) and failures (denoted by 'F') for HaifaSat and Siege_v1, without preprocessing and when combined with HYPRE and HYPERBINFAST. All run-times include both the preprocessing and the SAT solving times. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded.

	HYPRE		HYPERBINFAST	
Benchmark:	HYPRE	SAT	HYPERBINFAST	SAT
01_rule.k95.cnf	377	1679	4	1504
11_rule2.k70.cnf	1387	47	71	285
22_rule.k70.cnf	671	251	51	1410
fifo8_400.cnf	164	1226	12	309
7pipe.cnf	651	258	147	416
ip50.cnf	109	82	6	79
w08_14.cnf	1231	5	267	298
Total:	4590	3548	558	4301

Table 3. Few representative instances for which both HYPRE and HYPERBINFAST terminated before their respective (different) timeouts. The SAT times refer to HaifaSat's solving time. It can be seen that typically the solving time is longer after HYPERBINFAST, but together with the SAT solver time it is more cost effective than HYPRE.

4. D. Le Berre and L. Simon. Fifty-five solvers in vancouver: The sat 2004 competition. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, volume Lecture Notes in Computer Science. 2004. Accepted for publication.
5. Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the International Joint Conference on Artifical Intelligence*, 2001.
6. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.
7. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, 2004.
8. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
9. J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.
10. L. Simon, D. Le Berre, and E. Hirsch. The sat2002 competition. *Accepted for publication in Annals of Mathematics and Artificial Intelligence (AMAI)*, 43:343–378, 2005.
11. Allen Van Gelder and Yumi K. Tsuji. Satisfiability Testing with More Reasoning and Less Guessing. Number UCSC-CRL-95-34, pages 559–586, 1995.