Solving Non-clausal Formulas with DPLL search*

Christian Thiffault[1], Fahiem Bacchus[1]**, and Toby Walsh[2]***

[1] Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada
[cat|fbacchus]@cs.toronto.edu
[2] Cork Constraint Computation Center,
University College Cork, Ireland.
tw@4c.ucc.ie

Abstract. Great progress has been made on DPLL based SAT solvers operating on CNF encoded SAT theories. However, for most problems CNF is not a very natural representation. Typically these problems are more easily expressed using unrestricted propositional formulae and hence must be converted to CNF before modern SAT solvers can be applied. This conversion entails a considerable loss of information about the problem's structure. In this work we demonstrate that conversion to CNF is both unnecessary and undesirable. In particular, we demonstrate that a SAT solver which operates directly on a propositional formula can achieve the same efficiency as a highly optimized modern CNF solver. Furthermore, since the original formula remains intact, such a solver can exploit the original problem structure to improve over CNF solvers. We present empirical evidence showing that exploiting the original structure can yield considerable benefits.

1 Introduction

State of the art SAT solvers typically solve CNF encoded SAT theories using DPLL based algorithms [1]. However, many problems are more naturally expressed by arbitrary propositional formulas or Boolean circuits. Hence in order to use modern SAT solvers these problems must be converted into CNF. Converting to a simple and uniform representation like CNF provides conceptual and implementational simplicity. Indeed, a number of key techniques for improving the effectiveness and efficiency of DPLL solvers were originally designed to exploit the simple structure of CNF. However, such a conversion also entails considerable loss of information about the problem's structure, information that could be exploited to improve search efficiency.

In this paper, we argue that conversion to CNF is both unnecessary and undesirable. In particular, we have implemented NoClause, a non-CNF DPLL like solver that achieves a raw efficiency very similar to modern highly optimized CNF solvers by employing techniques very similar to those used in modern CNF solvers. Furthermore, we demonstrate how the additional structure present in the original propositional formula can be exploited to achieve significant gains in solving power, to the point where on various benchmarks NoClause outperforms the CNF solver it was based on.

---

* An extended abstract on this topic was presented at the SAT-2004 conference.
** Supported by Natural Science and Engineering Research Council of Canada.
*** Supported by Science Foundation Ireland.

The performance of our non-CNF solver is particularly encouraging for two other reasons. First, our implementation, although carefully constructed, does not employ any cache level optimizations. Nevertheless its raw performance is still close to that of the highly optimized CNF solver ZCHAFF [2]. Hence, there does not seem to be any intrinsic reason why a non-CNF solver cannot be as efficient as a CNF solver given equal engineering effort. Second, there are many other potential ways of exploiting the structure of the original propositional formula that we have not yet experimented with. It seems likely that some of these possibilities could yield additional performance improvements.

We begin by discussing CNF based SAT solvers, the way in which CNF encodings are generated, and the inherent disadvantages of CNF encodings. Then we present a method for performing DPLL search with a non-clausal encoding, and discuss the implementation techniques we utilized to obtain efficient inference on the non-clausal encoding. To go beyond mimicking current levels of performance we then present two techniques for exploiting the extra structure present in the non-clausal encoding. Empirical evidence shows that these techniques yield significant increases in performance. There has been some earlier work in the verification and theorem proving communities on formula based (or circuit based) solvers. We discuss this previous work pointing out the differences and similarities with our approach in various sections of the paper. Finally we close with some concluding remarks.

2 SAT solving using CNF

Many problems are more naturally described using arbitrary propositional formulas rather than clausal form. For example, hardware verification problems are often initially expressed in non-clausal form. To check the satisfiability of such formulas, the standard technique is to convert them to CNF and utilize a CNF SAT solver. Conversion to CNF is typically achieved using Tseitin encodings [3]. It is useful to review this encoding to better understand the correspondence between a non-clausal solver and a CNF solver.

Tseitin encodings work by adding new variables to the CNF formula, one new variable for every subformula of the original propositional formula, along with clauses to capture the dependence between these new variables and the subformulas. This is best illustrated by an example. Consider the propositional formula $\big(A \Rightarrow (C \land D)\big) \lor \big(B \Rightarrow (C \land E)\big)$. The Tseitin encoding would introduce the new variable F_1 to represent the subformula $C \land D$ and the new clauses $(\neg F_1, C)$, $(\neg F_1, D)$, and $(\neg C, \neg D, F_1)$ to capture the relation $F_1 \equiv (C \land D)$. Similarly we would have $F_2 \equiv (C \land E)$ and the clauses $(\neg F_2, C)$, $(\neg F_2, E)$, and $(\neg C, \neg E, F_2)$. With these new variables we would now have $A \Rightarrow (C \land D) \equiv A \Rightarrow F_1 \equiv \neg A \lor F_1$, and $B \Rightarrow (C \land E) \equiv \neg B \lor F_2$. Now two more new variables would be introduced $F_3 \equiv \neg A \lor F_1$ and $F_4 \equiv \neg B \lor F_2$ with the clauses $(\neg F_3, \neg A \lor F_1)$, $(A \lor F_3)$, $(\neg F_1, F_3)$, $(\neg F_4, \neg B \lor F_2)$, $(B \lor F_4)$, and $(\neg F_2, F_4)$. Finally, we introduce one more new variable $F_5 \equiv F_3 \lor F_4$ with the clauses $(\neg F_5, F_3, F_4)$, $(\neg F_3, F_5)$, and $(\neg F_4, F_5)$.[1]

Tseitin CNF encodings are linear in the size of the original formula as long as the Boolean operators that appear in the formula have linear clausal encodings. For ex-

[1] It is possible to build a more optimal encoding that only imposes the condition $F_5 \Rightarrow F_3 \lor F_4$ rather than equivalence as long as F_5 is not the descendant of an equivalence operator [4].

ample, the operators *and, or, not, nand, nor*, and *implies* all have linear sized clausal encodings. The k-ary *and* operator $A = A_1 \wedge \cdots \wedge A_k$ can be represented with a set of clauses of length $O(k)$ over the propositional variables A_i. Operators that do not have linear clausal encodings include k-ary biconditionals, k-ary counting operators (e.g., exactly 3 of the k inputs are true), and k-ary parity operators. The CNF encoding also retains some of the structure of the original formula. For example, any truth assignment to the variables of the original formula generates a truth assignment to every subformula; i.e., every subformula evaluates to TRUE or FALSE under this truth assignment. It is not difficult to see that a setting of the original variables will force a corresponding setting of the "subformula" variables in the CNF encoding.

The CNF encoding has two main disadvantages. The first, and most fundamental problem is that a great deal of structural information is lost: the clauses no longer directly reflect the structure of the original circuit. For example, it is not immediately obvious that the F_i variables represent derived signals rather than input signals, that F_4 is upstream of F_2 in the original circuit, or that F_4 encodes an *or* gate while F_1 encodes an *and* gate. In this simple example, some of this information can be computed from the clausal encoding. In general, however, whilst some of this information can be computed from the clausal encoding, some of it is intractable to compute. For example, it is intractable to determine which variables represent derived signals and which represent the original variables in an arbitrary CNF encoded formula [5].

The second problem is that the CNF theory contains more variables, which means that the space of truth assignments from which a solution must be found has been enlarged by an exponential sized factor. This does not necessarily mean that in practice the search for a solution is any harder. Nevertheless, as we shall explain, the difficulty of searching this larger space is exacerbated by the first problem, the lack of structural information.

Loss of Structural Information. A number of works show that the structural information lost in a CNF encoding can be used to give significant performance improvement. For example, the EqSATZ solver [6] achieves significant gains by extracting and exploiting biconditionals from the CNF theory. Until very recently, it was the only solver able to complete the par32 family of problems which contain many biconditionals. More recently, the Lsat solver [7] has shown that extracting even more extensive structural information can allow some problems, that are very hard for clausal solvers, to be solved quite easily. Given that these solvers have to utilize specialized (and incomplete) methods to extract the structural information they need, and given that many problems start off with a structure rich non-clausal encoding, it is natural to see if we can solve the problem more efficiently and effectively in its original non-clausal encoding. In addition to this empirical evidence, recent theoretical results show that on some problems structure can be exploited to derive branching decisions that reduce the size of the search space exponentially [8].

The Added Variables. The second problem, that of additional variables in the CNF encoding, is an issue that has been the subject of some previous work. The main approach taken, e.g., [9, 10, 7], has been to annotate the CNF theory to distinguish between the original variables (the primary inputs) and the derived variables (the internal signals).

It is assumed that either the annotation is supplied with the CNF encoding (thus a small amount of additional structural information is preserved) or is approximated by examining the CNF encoding [7]. Given this annotation, the recommendation is then to restrict the DPLL search from branching on any derived variable: once all of the primary inputs have been set all of the derived signals can be determined by unit propagation. The benefit of this technique is that now the CNF encoding can be solved by searching in the same sized state space: the set of assignments to the original propositional variables.

Unfortunately, there is compelling empirical and theoretical evidence that this simple technique is not robust. For example, the Lsat solver uses this technique of branching only on input variables. It displays impressive performance on a number of problems, but very disappointing performance on an even wider range of problems. The most robust and powerful SAT solvers do not restrict their branching decisions in this manner. From a theoretical point of view it can be shown that restricting the solver to branching only on the input variables entails a reduction in the power of the proof system it implements. A number of results of this form have been given in [11]. These results show that there exist families of Boolean circuits on which a DPLL solver that branches only on the input variables (in the clausal encoding) will always explore an exponentially sized search tree (irrespective of how it chooses which of the input variables it wants to branch on), while a DPLL solver that is allowed to branch on the derived variables can construct a constant sized refutation tree.

Theorem 1. *There exists families of Boolean circuits such that a short resolution proof of unsatisfiability exists if and only if branching on derived variables is allowed [11].*

This result shows that we want to branch on the derived variables, so as not to suffer a loss in power of the proof system. Hence, it is useful to analyze more carefully possible sources of inefficiency that such branching can produce. First, it should be noted that it is not necessarily the case that a DPLL procedure will search a larger space when additional variables are introduced. For example, if we add the new variables $Y_1, \ldots, Y_n$ to a theory containing the variables $X_1, \ldots, X_n$, but also include the clauses $(\neg X_i, Y_i)$ and $(\neg Y_i, X_i)$ making each Y_i equivalent to its corresponding X_i, then there will be no effect on the size of the DPLL search tree: each time Y_i or X_i is set the other variable will be set by unit propagation.

A major source of inefficiency introduced when branching on derived variables arises from subsequent branching on variables that have become *don't care* due to previous decisions. Consider for example a formula of the form $PHP^n \vee (q \wedge p)$, where PHP^n is an unsatisfiable formula requiring an exponentially sized resolution refutation (e.g., the pigeon hole problem with n pigeons), and q and p are propositional variables. The clausal encoding of this formula contains the added variables $B_1 \equiv PHP^n$, $B_2 \equiv (q \wedge p)$, $B_3 \equiv (B_1 \vee B_2)$, and other variables added by the clausal encoding of PHP^n. If the solver first assigns $B_3 = \text{TRUE}$, then $B_2 = \text{TRUE}$ both q and p will be unit propagated to TRUE. This set of assignments satisfies the formula. However, the clausal theory will still contain the clauses encoding the subformula $B_1 \equiv PHP^n$ so the solver's job will not yet be completed. If the solver was then to set the input variables of PHP^n, any such setting would force the setting $B_1 = \text{FALSE}$ and the solver would be finished. Similarly, if the solver was to set $B_1 = \text{FALSE}$ then it could find a setting of the variables in PHP^n that falsifies PHP^n and again it would be finished. However, if it made the wrong decision of first setting $B_1 = \text{TRUE}$, then it would be faced with having to produce an exponentially size refutation of PHP^n in order to backtrack to reset $B_1 = \text{FALSE}$. All of this work is unnecessary, but in the clausal encoding it is difficult to detect that the work is not needed.

In this example, we do not need to branch on any of the variables encoding PHP^n. This part of the formula has become irrelevant and the variables in it have become don't care since their values cannot affect the value of the formula. How often CNF DPLL solvers branch unnecessarily on don't care variables, and how much search is wasted by doing so is an empirical question. We present results which along with previous evidence [12, 13] indicates that the amount of wasted time is significant.

3 DPLL without conversions

Our approach is designed to decide if a propositional formula is satisfiable. A propositional formula can be represented as an operator tree, where each internal node is a Boolean operator and its children are subtrees representing its operands. After inputing the formula we first compress it by converting the tree representation into a directed acyclic graph (DAG) in which all duplicates of a sub-formula are merged. For example, in the formula $\big(A \Rightarrow (C \wedge D)\big) \vee \big(B \Rightarrow (C \wedge D)\big)$ the DAG would contain only one instance of the subformula $(C \wedge D)$. Propositional formulas represented as DAGS are often called Boolean circuits. The conversion to a Boolean circuit can be done bottom up using hashing to identify common sub-formulas.

Once the DAG representation is computed we store it in a contiguous section of memory and associate with each node of the DAG (gate) the following data:

1. A unique identifier.
2. A list of parent nodes (in a DAG a node might have many parents).
3. A list of children nodes.
4. The type of the node (e.g., the node might be a propositional variable, an *and* gate, an *or* gate, etc.).
5. A truth value (TRUE, FALSE, *don't care*, or *unknown*.)
6. The decision level at which the node's truth value changed from *unknown*.
7. The reason that a node's truth value changed from *unknown* (either the trigger of a propagation rule or a conflict clause).

Given this representation our task is to consistently label the nodes with truth values such that the top level node (representing the entire formula) is labeled TRUE. A labeling is consistent if it respects the logic of the node types. For example, if an *and* node is labeled TRUE all of its children must be labeled TRUE, if a *not* node is labeled FALSE its child must be labeled TRUE, etc. We try to find a consistent labeling, or prove that one does not exist, using a backtracking tree search (i.e., a DPLL search) on the truth values assigned to each node. Such an approach has been used in previous work on circuit-based solvers, e.g., [14–16].

Our backtracking search procedure chooses an unlabeled node to label, labels it TRUE or FALSE, propagates the consequences of that label, and then recursively tries to label the remaining nodes. If that fails it backtracks and tries the opposite label. Note that the search can choose to label (branch on) any unlabeled node in the DAG. Choosing an internal node corresponds to branching on the truth value of a subformula of the input, while choosing a leaf node corresponds to branching on the truth value of one of the inputs of the circuit.

Once a node is labeled with a truth value, the consequences of that truth value are propagated through the DAG. Propagation utilizes the node's data to propagate other labels through the DAG, e.g., if a node is labeled FALSE then FALSE is propagated to all of its parents that are *and* gates; if it is labeled TRUE and it is an *and* gate then TRUE is propagated to all of its children, etc. Propagation of labels goes up and down the DAG guided by a simple set of propagation rules. Similar propagation rules were used in the works cited above.

A contradiction is detected when a node gets both a TRUE and a FALSE label. Once we have a contradiction we must backtrack and try a different labeling. It is not difficult to see that setting the truth value of a node corresponds precisely to setting the variables identified with the subformula headed by that node in the Tseitin CNF encoding. Similarly, propagation of labels corresponds to unit propagation in the CNF theory.

Proposition 1. *If assigning a variable v the truth value x in the Tseitin CNF encoding of a circuit causes another variable v' to be assigned the truth value y by unit propagation, then assigning the node corresponding to v the value x will cause the node corresponding to v' to be assigned the value y by applying our propagation rules.*

As in [15], for each propagated label we remember the set of node labels that caused the propagation. For example, if we propagate TRUE to an *and* because all of its children were set to TRUE we would associate the TRUE labels of all of the children as the reason for the TRUE labeling of the *and* node. If the *and* node is subsequently labeled FALSE because of propagation from one of its parents, we would have another set of node labels as the reason for the FALSE label. We can combine these labels to obtain a conflict set. The negation of the labels in the conflict set is a conflict clause just like those constructed by CNF solvers. In fact, by successively replacing the most recently propagated label by its reason until we have only one label at the current decision level, we can implement precisely 1-UIP learning [17]. As in [15], we discover and then store such conflicts in a clausal database. These conflict clauses are then unit propagated using standard techniques (assigning two node labels as the watch labels). Thus nodes in the DAG are labeled by unit propagation from the conflict clauses as well as by propagation in the circuit DAG.

Efficient Propagation in the DAG The main difference in our implementation and the previous circuit based SAT solvers cited above is that we adopt the watch literal technique from CNF solvers and apply it to our circuit representation. Watches are used wherever they can make a propagation rule more efficient. Propagation through an *and* gate provides a typical example. There are four rules for propagating *and* gates:

1. If the *and* becomes TRUE propagate TRUE to all of its children.
2. If a child becomes FALSE propagate FALSE to the *and* node.
3. If all of the children become TRUE propagate TRUE to the *and* node.
4. If the *and* node is FALSE and all but one of the children are TRUE then propagate FALSE to the unlabeled child.

Watches do not help the first two rules. In fact in the clausal encoding the first two rules would correspond to unit propagation through the binary clauses $(\neg A, C)$, where A is literal corresponding to the *and* node and C is one of its children. Watches do not aid in the efficiency of binary clauses either. To make the second rule efficient we divide the parent list of a node into separate lists based on the parent's type. So we would have a separate list of *and* parents, another list of *or* parents, etc. Thus when a node is labeled FALSE we can efficiently propagate FALSE to all of its *and* parents.

Watches offer significant improvement for the third and fourth rules. For every *and* node we assign two children to be TRUE watches, and for every node we maintain a list of parents it serves as a TRUE watch for (the node might also be a FALSE watch for some *or* nodes). We maintain the invariant that neither watch should be assigned TRUE unless we have no other choice, or the other watch is already FALSE. When a node is assigned TRUE we examine each of the parents for which it is a TRUE watch. For each parent we first look at the other watch child, if that child is already FALSE we do not need to do anything. If it is TRUE then we know that every child of the parent is now true, and we can activate the third rule propagating TRUE to the parent. Otherwise we look to see if we can find another child of the parent that is currently unassigned or FALSE and make that the new watch. If we cannot find an alternative watch we leave the current watch intact, obtaining one TRUE watch and one unassigned watch, and check the *and* node to see if it is currently FALSE. If it is then we activate rule four and propagate FALSE to the sole unassigned watch child. Finally, whenever we label an *and* node FALSE, we look at its two watch children. If one of these is TRUE we know that the other is the only remaining unassigned child, and we activate rule four propagating FALSE to that child.

Previous circuit based solvers (e.g., [14, 15]) have restricted themselves to binary Boolean operators and have used tables to perform label propagation in the DAG. Although table lookup is fast, to propagate TRUE to an *and* node containing thousands of children (and then converted to a tree of binary *and* nodes) requires a table lookup every time one of the children is labeled. With the watch child technique, we only need to perform some computation when a watch child is labeled. One of the suites we experimented with (VLIW-SAT.1.1 due to M. Velev) contained *and* nodes with an average of 15.9 children, and had some *and* nodes with over 100,000 children. The other suites also contained some *and* nodes with thousands of children. As a result we found that implementing the watch child technique for triggering propagation in the DAG yielded very significant gains in efficiency. Besides watches for the *and* propagation rules we were able to use an analogous set of watches for the *or*, *iff*, and *xor* propagation rules. Watches were also used in don't care propagation.

As a result, conversion to CNF does not seem to be necessary. A non-clausal DPLL SAT solver can duplicate the search performed by a CNF solver: labeling nodes corresponds to making literals in the CNF encoding TRUE, propagation in the DAG corresponds to unit propagation in the CNF encoding, conflicting labels corresponds to conflicting assignments to a variable in the CNF encoding, and conflicts consisting of sets of labels correspond to conflict clauses in the clausal encoding. Furthermore, propagation in the DAG can be made just as efficient as unit propagation in the CNF encoding by using watched children. Besides being able to duplicate the behavior of CNF solvers, non-CNF solvers also have the advantage of being able to exploit additional structural information not present in a CNF encoding. In the next section we present some simple techniques for exploiting this structure.

4 Exploiting Structure

4.1 Complex Gates

It is well known that every DPLL search tree for an UNSAT problem can be converted to a similarly sized resolution refutation of the problem. Hence, a clausal DPLL solver is inherently limited by the power of resolution. Non-clausal DPLL solvers on the other hand can employ complex gates and special purpose propagators for those gates that in certain cases, can circumvent the limitations imposed by resolution. In fact, any subcircuit can be encoded as a single gate and propagation through that subcircuit realized by a specialized propagation algorithm.

For example, the SAT problem might be an encoding of a CSP problem containing an all-different constraint. In general, reasoning with all-different requires exponentially sized resolution proofs (e.g., the pigeon hole problem can be encoded with a single all-different constraint over a set of multi-valued variables). Hence, the causal encoding to SAT will suffer an exponential loss of efficiency over polytime all-different propagators. In a non-clausal solver, on the other hand, a single gate can be used to encode an all-different constraint and a polytime propagator can be used to propagate truth values through the gate.

4.2 Don't Care Propagation

The problem described earlier where a clausal solver might branch on a don't care variable is easily addressed using the circuit structure. Two techniques have previously been described in the literature for exploiting don't cares. Gupta et al. tag each variable with fanin and fanout information from the original circuit [13]. Using this information they are able to detect when a clause encodes part of the circuit that no longer influences the output, given the variable assignments we have already made. Such clauses are tagged as being inactive and are restored to active status when backtracking makes them relevant again. The detection of inactive clauses requires a sweep through all of the active clauses in the theory. This sweep must be performed at every node in the search tree.

Safarpour et al. use a different technique [12]. They maintain the original circuit and use it to mark variables that dynamically become don't cares (*lazy* in their notation). Then they prohibit the CNF solver from branching on don't care variables. They scan the entire circuit to detect don't care variables at every node of the search tree.

Like Safarpour et al. we also use a variable marking technique. However, we have gained efficiency by using watches and by not having to maintain both a CNF encoding as well as a circuit description. To understand how watches improve efficiency consider a typical example when a node is the child of an *and* node that has been labeled FALSE. The node's label is then irrelevant with respect to its impact on this particular *and* parent. However, the node might still be relevant to the label of its other parents. Hence, a node's value becomes irrelevant to the circuit as a whole only when for each of its parents it is either irrelevant to that parent's value, or that parent has itself become irrelevant to the circuit's output.

To perform efficient propagation of don't care values through the DAG we use a single don't care watch parent for each node. The invariant for a don't care watch parent is that the parent should not be a don't care and that the child it is watching should not be irrelevant to its value. Whenever a node is assigned at truth value that makes its watched children irrelevant, or when a don't care value is propagated to it we search for a new don't care watch parent for each watched child. If we fail to find one we can propagate a don't care to the child and then perhaps subsequently to the child's children, etc. Our use of watches means that computation is required only when the watch parent is modified, changes to the other parents do not require any computation. In the approaches described above a node will be checked every time one of its parents is modified.

Some of the problems we experimented with contained nodes with over 8,000 parents, and an average of 23 parents per node. Many other problems contained nodes with over a 1,000 parents. Hence our watch technique yielded significant gains. As described below, on some problems we obtained a speedup of 38 times using don't care propagation. The above cited works report speedups from don't care propagation in the order of only 3 to 7 times. This also provides evidence that CNF solvers are wasting a significant amount of time by branching on don't care variables, and are thus suffering from the loss of structural information in the CNF encoding.

4.3 Conflict Clause Reduction

Another structure based technique we have implemented is conflict clause reduction. To the best of our knowledge this technique is new to this work. The idea is simple. When we learn a conflict clause it will contain some set of node labels. We examine these labels to see if any of them are "locally" redundant given the circuit structure, and if they are we remove them. We say that label ℓ makes label ℓ' *redundant* if one of the DAG propagation rules generates ℓ' from ℓ. For example, if n is an *and* node and n' is one of its children, then $n = \text{FALSE}$ makes $n' = \text{FALSE}$ redundant. In a conflict clause we can remove any redundant labeling. For example, if we have the conflict clause $(n = \text{FALSE}, n' = \text{FALSE}, x = \text{TRUE}, \ldots)$ we can reduce this clause to $(n = \text{FALSE}, x = \text{TRUE}, \ldots)$. This corresponds to a resolution step: we have that $n' = \text{FALSE} \Rightarrow n = \text{FALSE} \equiv (n' = \text{TRUE}, n = \text{FALSE})$, which resolved against the conflict clause yields the reduced clause. In addition to removing any label made redundant by another label, we can transitively remove all labels made redundant by the redundant label. Since redundancies are defined with respect to local DAG propagation rules, all redundancies can be efficiently checked by examining the parents and children of the node in the label.

The above resolution of $(n' = \text{TRUE}, n = \text{FALSE})$ and $(n = \text{FALSE}, n' = \text{FALSE}, x = \text{TRUE}, \ldots)$ is known as Krom Subsumption. Krom subsumption has been utilized before in clausal SAT solvers [18]. The difficulty lies in detecting that the binary clause $(n' = \text{TRUE}, n = \text{FALSE})$ holds. van Gelder and Tsuji [18] utilized extensive binary clause reasoning to discover a large number of binary clauses at each stage of the search. These binary clauses were then used to perform Krom subsumption (however their solver did not generate new learned clauses). The advantage of the non-clausal representation is that a very local test can be performed (running the local propagation rules) to effectively circumscribe the computation required to detect the needed binary clauses.

We experimented with various uses of conflict clause reduction and found empirically that the most effective use was to employ reduction on shorter conflict clauses,

	ZCHAFF				NOCLAUSE			
Benchmark	**Time**	**Decisions**	**Impl/s**	**Size**	**Time**	**Decisions**	**Impl/s**	**Size**
sss-sat-1.0 (100)	128	2,970,794	728,144	70	225	1,532,843	616,705	39
vliw-sat-1.1 (100)	3,284	154,742,779	302,302	82	1,033	4,455,378	260,779	55
fvp-unsat-1.0 (4)	245	3,620,014	322,587	326	172	554,100	402,621	100
fvp-unsat-2.0 (22)	20,903	26,113,810	327,590	651	4,104	5,537,711	267,858	240

Table 1. Comparison between ZCHAFF and NoClause on 4 benchmark suites. Time: CPU seconds to solve all problem in suite. Decision: number of branches in the search tree. Impl/s: number of unit implications per second during search. Size: average size of a learned conflict clause.

length 100 or less. For longer conflict clauses, the clause remained long even after reduction, whereas on the shorter clauses the reduction produced more useful clauses. It should also be noted that conflict clause reduction has a cumulative effect: conflict clauses produce new conflict clauses, so shorter conflict clauses produce new conflict clauses that are themselves shorter.

5 Empirical Results

Our non-clausal DPLL solver NOCLAUSE uses the ideas described above. We represent the input as a propositional formula in ISCAS format, convert it to a non-redundant Boolean circuit, perform 1-UIP clause learning at failures, use ZCHAFF's VSIDS heuristic to guide branching, perform don't care propagation, and use the circuit structure to reduce all learned clauses of size 100 or less.

We designed our solver to perform a carefully controlled experimental comparison with the ZCHAFF solver. ZCHAFF is no longer the fastest SAT solver, but its source code is available. Hence, we were able to have better control over the differences between our solver and ZCHAFF. In particular, we duplicated as much as possible ZCHAFF's branching heuristic, clause learning, and clause database management techniques by careful examination of the ZCHAFF code. Hence we were able to build NOCLAUSE so that the differences with ZCHAFF are mainly dependent on NOCLAUSE's use of the circuit structure. This allows us to assess more accurately the benefits of using a non-CNF representation.

For this reason we compare only with the ZCHAFF solver. Our aim is to demonstrate the specific benefits of a non-CNF representation. Other more recent solvers, e.g., BerkMin and Siege, employ different branching heuristics from ZCHAFF and to some extent different clause learning techniques, and are often able to outperform ZCHAFF with these new techniques. However, as explained in Sec. 3 a non-CNF solver can be made to duplicate the search performed by a CNF solver. Thus it should be possible to implement the same branching and clause learning techniques employed in these other solvers with a commensurate gain in efficiency. It seems plausible that at least some of the gains we obtain from exploiting structural information would be preserved under these alternate branching and clause learning strategies. Unfortunately, the exact nature of the strategies employed in many of these solvers remains undisclosed so it is difficult to test such a conjecture.

Another restriction in our experimental results is that our solver requires non-CNF input. It was quite difficult to obtain non-CNF test problems, and the only ones that were

		ZCHAFF				NOCLAUSE			
Problem	**# Vars.**	**Time**	**Decisions**	**Impl/s**	**Cls Size**	**Time**	**Decisions**	**Impl/s**	**Cls Size**
2pipe	892	0.14	6,362	1,156,271	35	0.27	4,880	1,133,000	17
2pipe_1	834	0.17	5,254	1,075,924	32	0.13	3,323	925,923	13
2pipe_2	925	0.25	6,664	1,042,740	38	0.31	5,697	828,923	18
3pipe	2,468	2.74	39,102	865,566	88	1.45	14,898	702,202	24
3pipe_1	2,223	2.43	25,939	724,459	87	7.93	39,859	419,688	48
3pipe_2	2,400	3.80	35,031	723,537	93	5.99	31,622	414,157	36
3pipe_3	2,577	6.94	53,806	653,575	105	7.10	37,258	427,852	53
4pipe	5,237	188.89	541,195	467,001	253	9.87	41,637	509,433	40
4pipe_1	4,647	26.55	131,223	512,108	158	35.52	114,512	327,098	77
4pipe_2	4,941	49.76	210,169	482,896	186	36.50	112,720	327,298	84
4pipe_3	5,233	144.34	392,564	424,551	254	62.03	169,117	316,049	108
4pipe_4	5,525	93.83	295,841	470,936	228	42.26	122,497	326,186	112
5pipe	9,471	54.68	334,761	526,457	258	33.34	102,077	409,154	93
5pipe_1	8,441	126.11	381,921	425,921	273	116.18	255,894	280,758	140
5pipe_2	8,851	138.62	397,550	437,166	276	177.24	362,840	279,298	165
5pipe_3	9,267	137.70	385,239	441,319	271	134.08	292,802	295,976	165
5pipe_4	9,764	873.81	1,393,529	370,906	406	284.62	503,128	270,234	208
5pipe_5	10,113	249.11	578,432	456,400	324	137.09	283,554	298,903	172
6pipe	15,800	4,550.92	5,232,321	322,039	619	297.13	435,781	288,855	232
6pipe_6	17,064	1,406.18	2,153,346	402,301	469	1,056.56	1,326,371	267,207	309
7pipe	23,910	12,717.00	12,437,654	306,433	900	1,657.70	1,276,763	244,343	336
7pipe_bug	24,065	128.90	1,075,907	266,901	393	0.29	481	403,148	10

Table 2. Comparison between ZCHAFF and NOCLAUSE on the complete fvp-unsat-2.0 benchmark suite. #Vars: number of variables in problem instance. Other columns as in Table 1.

able to obtain had already suffered some loss of structural information by been encoded into ISCAS format which contains only *and*, *or*, and *not* gates. We expect to see even better performance on problems which have not been so transformed. All experiments were run on a 2.4GHz Pentium IV machine with 3GB of RAM.

Table 1 shows the performance of ZCHAFF and NOCLAUSE on four different benchmark suites containing a total of 226 problems. These suites were the only difficult problems we found that were available in both CNF and non-CNF formats. The table shows the total run time (all times in CPU seconds) to solve the suite, the total number of decisions (branches) over all of the search trees explored, and the rate of unit propagations per second achieved. It also shows the average size of the conflict clauses learned over the suite. We see that NOCLAUSE is faster on all but the easiest of suites (sss-sat-1.0, where each problem took ZCHAFF an average of 1.3 seconds to solve). NOCLAUSE is significantly faster on the hardest of the suite fvp-unsat-2.0. We also see that it makes far fewer decisions and learns shorter conflict clauses. Furthermore, its raw performance, measured in terms of implications per seconds is comparable with ZCHAFF.

Table 2 shows the runtimes of ZCHAFF and NOCLAUSE in more detail on the fvp-unsat.2.0 suite. On the larger problems, ZCHAFF is learning very long conflict clauses,

	NoClause					NoClause without don't cares			
Benchmark	**Time**	**Decisions**	**Step**	**Impl/s**	**DC/s**	**Time**	**Decisions**	**Step**	**Impl/s**
sss-sat-1.0 (100)	225	1,532,843	4.20	411,760	204,945	272	3,095,245	6.75	652,927
vliw-sat-1.1 (100)	1,033	4,455,378	6.32	175,995	84,784	2,120	13,208,363	10.86	381,188
fvp-unsat-1.0 (4)	172	554,100	3.95	212,012	190,609	494	3,442,123	12.42	295,179
fvp-unsat-2.0 (22)	4,104	5,537,711	3.03	186,603	81,255	30,934	20,382,047	3.18	335,242

Table 3. Analysis of don't care propagations in NoClause on 4 benchmark suites. DC/s: number of don't care implications per second. Step: average number of levels backtracked over on backtrack.

	NoClause					NoClause without don't cares		
Problem	**Time**	**Decisions**	**Impl/s**	**DC/s**	**Total/s**	**Time**	**Decisions**	**Impl/s**
4pipe	9.87	41,637	337,405	172,028	509,433	57.68	198,828	526,014
4pipe_1	35.52	114,512	236,336	90,762	327,098	62.65	159,049	413,037
4pipe_2	36.50	112,720	234,174	93,124	327,298	94.46	212,986	438,386
4pipe_3	62.03	169,117	233,951	82,098	316,049	213.27	365,007	404,224
4pipe_4	42.26	122,497	244,725	81,460	326,186	318.64	525,623	412,208
5pipe	33.34	102,077	281,338	127,816	409,154	246.93	650,312	559,266
5pipe_1	116.18	255,894	194,417	86,341	280,758	300.59	489,825	380,509
5pipe_2	177.24	362,840	190,681	88,617	279,298	360.67	585,133	392,928
5pipe_3	134.08	292,802	206,226	89,750	295,976	387.65	593,815	405,352
5pipe_4	284.62	503,128	198,872	71,362	270,234	2097.31	1,842,074	360,270
5pipe_5	137.09	283,554	212,402	86,501	298,903	379.19	543,535	448,226
6pipe	297.13	435,781	194,494	94,361	288,855	10,241.64	4,726,470	283,592
6pipe_6	1,056.56	1,326,371	192,991	74,216	267,207	3,455.35	2,615,479	374,503
7pipe	1,657.70	1,276,763	163,934	80,409	244,343	12,685.59	6,687,186	343,710
7pipe_bug	0.29	481	345,986	57,162	403,148	1.00	2,006	481,415

Table 4. Analysis of the don't care propagations in NoClause on the non-trivial problems from the fvp-unsat-2.0 benchmark. Total/s: number of implications per second plus the number of don't care propagations per second.

much longer than those learned by NoClause. We also see that NoClause displays more uniform scaling behavior.

Table 3 shows the effect of don't care propagation on NoClause's performance. We see that don't care propagation is a major contributor to its performance. Without don't care propagation NoClause for the most part has inferior performance to ZChaff being slower on two of the benchmarks and slightly faster on the other two. The table also shows the average number of don't care implications per second on these suites, and the average number of backjump levels on detecting a conflict. We see that without don't cares the solver will jump back further on average. This is because the solver is able to jump back over decision levels where don't care variables were branched on. Nevertheless, despite the ability of conflict clauses to jump back over don't care decisions, don't care decisions still have a significant negative impact on performance.

Table 4 shows in more detail the results of don't care propagation on the non-trivial problems in the fvp-unsat-2.0 suite. We see that don't care propagation has its largest

	NoClause						NoClause without reductions			
Benchmark	**Time**	**Decisions**	**Impl/s**	**Size**	**Exam**	**Rem**	**Time**	**Decisions**	**Impl/s**	**Size**
sss-sat-1.0	225	1,532,843	616,705	39	90%	12%	228	1,624,312	628,953	52
vliw-sat-1.1	1,033	4,455,378	260,779	55	88%	11%	984	4,322,679	281,017	90
fvp-unsat-1.0	172	554,100	402,621	100	73%	11%	402	820,582	311,127	119
fvp-unsat-2.0	4,104	5,537,711	267,858	240	33%	16%	5,675	7,614,898	246,498	418

Table 5. Analysis of clause reductions in NoClause on 4 benchmark suites. Exam: percentage of conflict clauses examined. Rem: percentage of literals removed from the examined clauses.

	NoClause						NoClause without reductions			
Problem	**Time**	**Decisions**	**Impl/s**	**Size**	**Exam**	**Rem**	**Time**	**Decisions**	**Impl/s**	**Size**
4pipe	9.87	41,637	509,433	40	90%	18%	30.50	105,543	384,022	148
4pipe_1	35.52	114,512	327,098	77	73%	16%	45.13	146,250	324,710	132
4pipe_2	36.50	112,720	327,298	84	71%	16%	54.97	160,462	311,666	132
4pipe_3	62.03	169,117	316,049	108	55%	15%	69.70	189,004	320,735	162
4pipe_4	42.26	122,497	326,186	112	59%	15%	80.13	204,747	303,505	157
5pipe	33.34	102,077	409,154	93	63%	17%	16.29	67,868	437,006	113
5pipe_1	116.18	255,894	280,758	140	42%	14%	195.81	410,159	289,525	204
5pipe_2	177.24	362,840	279,298	165	33%	15%	159.45	348,772	297,682	199
5pipe_3	134.08	292,802	295,976	165	32%	13%	154.02	331,226	294,891	218
5pipe_4	284.62	503,128	270,234	208	30%	13%	504.27	806,083	269,251	264
5pipe_5	137.09	283,554	298,903	172	32%	11%	216.11	424,870	305,280	237
6pipe	297.13	435,781	288,855	232	37%	16%	647.78	834,635	251,934	540
6pipe_6	1,056.56	1,326,371	267,207	309	14%	10%	1,421.42	1,648,993	253,759	380
7pipe	1,657.70	1,276,763	244,343	336	28%	22%	2,053.92	1,778,995	201,923	761
7pipe_bug	0.29	481	403,148	10	100%	0%	0.29	481	403,148	10

Table 6. Analysis of clause reductions in NoClause on the non-trivial problems from the fvp-unsat-2.0 benchmark. Exam: percentage of NoGoods examined; Rem: percentage of literals removed from the examined clauses.

impact on the hardest problems. For example, we obtain a speed up factor 34 on the 6pipe instance. We also see that the total number of propagations per second (implications plus don't cares indicated in the Total/s column) remains fairly similar with the addition of don't care propagation. However, some of that raw speed must now be used to perform don't care propagation which results in a lower rate of implications per second. Nevertheless, this more than pays for itself in the reduced number of decisions.

Table 5 shows the effect of conflict clause reduction on performance. The size column shows the average size of a conflict clause learned. We see that without reduction the conflicts are significantly larger. Note that the size of the clause was measured prior to being reduced. The table also shows the percentage of clauses that are examined for reduction (only clauses of length 100 or less are reduced) and for these the percentage reduction achieved. We see that for the easier suites, sss-sat-1.0, and vliw-sat-1.1, clause reduction does not help—the clauses are already quite short. For the two harder suites clause reduction does provide useful performance gains, although the gain is less significant that that achieved with don't cares. We also see that as the problems get harder, the conflict clauses get longer and a smaller percentage of them are examined for reduction (the examined column). However, despite only reducing 33% of the conflicts in the fvp-unsat-2.0 suite, we still cut the average size of the conflicts by almost half. This shows that reducing only some of the conflicts can still have a significant impact on the other learned conflicts.

Table 6 shows in more detail the results of conflict clause reduction on the non-trivial problems in the fvp-unsat-2.0 suite. We see that clause reduction does not always improve performance, e.g., 5pipe and 5pipe_2. However, overall it appears to offer a useful performance enhancement. Clause reduction also influences the branching heuristic (as mentioned above we use ZCHAFF's VSIDS heuristic which counts how many times a literal appears in recent conflict clauses, see [2] for details), and we see that on the fvp-unsat-2.0 suite performance improvement is correlated with the number of decisions. Further analysis of the interaction between the branching heuristic and clause reduction is a topic for future investigation.

6 Conclusion

Our results demonstrate that conversion to CNF is unnecessary. A DPLL like solver can reason with Boolean circuits just as easily as with a clausal theory. We have implemented NOCLAUSE, a non-CNF DPLL like solver with similar raw efficiency to highly optimized clausal DPLL solvers. Reasoning with Boolean circuits offers a number of advantages. For example, we can support much more complex inference like formula rewriting, as well as propagation rules for complex gates like counting gates. We can also use the circuit structure to simplify learned clauses, and to inform branching heuristics. NOCLAUSE is related to a number of previous works on circuit based Boolean solvers. Its main innovations are (a) greater efficiency through adaptation of the watch literal technique and (b) its new technique of conflict clause reduction. It is also the first circuit based solver that performs don't care propagation (previous uses of don't care reasoning have built on top of CNF solvers). We have demonstrated empirically that don't care propagation has a very significant impact on performance, and that conflict clause reduction can offer useful performance improvements.

Our experimental results are very promising. We often outperform a highly optimized solver like ZCHAFF. We expect that the results would be even more favorable if the benchmarks available to us had not already lost some of their structure. As we explained before, the ISCAS format only contains *and*, *or*, and *not* gates. There are many other ways in which we expect performance could be further improved. For example, more complex preprocessing of the input circuit, as in BCSat [16], is likely to offer major efficiency gains. Most interesting, however, is that we have only scratched the surface with respect to using structure to perform more sophisticated clause learning, branching, and non-chronological backtracking. Future work on these topics has the potential to deliver significant performance improvements.

References

1. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **4** (1962) 394–397

2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proc. of the Design Automation Conference (DAC). (2001)
3. Tseitin, G.: On the complexity of proofs in poropositional logics. In Siekmann, J., Wrightson, G., eds.: Automation of Reasoning: Classical Papers in Computational Logic 1967–1970. Volume 2. Springer-Verlag (1983) Originally published 1970.
4. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2** (1986) 293–304
5. Lang, J., Marquis, P.: Complexity results for independence and definability in propositional logic. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. (1998) 356–367
6. Li, C.M.: Integrating equivalence reasoning into Davis-Putnam procedure. In: Proceedings of the AAAI National Conference (AAAI). (2000) 291–296
7. Ostrowski, R., Grégoire, E., Mazure, B., Sais, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Principles and Practice of Constraint Programming. Number 2470 in Lecture Notes in Computer Science, Springer-Verlag, New York (2002) 185–199
8. Beame, P., Kautz, H., Sabharwal, A.: Using problem structure for efficient clause learning. In: Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). Number 2919 in Lecture Notes In Computer Science, Springer (2003) 242–256
9. Giunchiglia, E., Sebastiani, R.: Applying the Davis-Putnam procedure to non-clausal formulas. In: AI*IA 99: Advances in Artificial Intelligence: 6th Congress of the Italian Association for Artificial Intelligence. Volume 1792 of Lecture Notes in Computer Science., Springer (2000) 84–95
10. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables for propositional satisfiability. In: Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA). Volume 2424 of Lecture Notes in Computer Science., Springer (2002) 23–26
11. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. In: AI&M 2004, 8th International Symposium on Artificial Intelligence and Mathematics. (2004) Available on-line at http://rutcor.rutgers.edu/ amai/aimath04/.
12. Safarpour, S., Veneris, A., Drechsler, R., Lee, J.: Managing don't cares in Boolean satisfiability. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume I (DATE'04), IEEE Computer Society (2004) 10260
13. Gupta, A., Gupta, A., Yang, Z., Ashar, P.: Dynamic detection and removal of inactive clauses in SAT with application in image computation. In: Proceedings of the 38th conference on Design automation, ACM Press (2001) 536–541
14. Circuit-based Boolean Reasoning. In: Proceedings of the 38th conference on Design automation, ACM Press (2001)
15. Ganai, M.K., Ashar, P., Gupta, A., Zhang, L., Malik, S.: Combining strengths of circuit-based and cnf-based algorithms for a high-performance SAT solver. In: Proceedings of the 39th conference on Design automation, ACM Press (2002) 747–750
16. Junttila, T., Niemelä, I.: Towards an efficient tableau method for Boolean circuit satisfiability checking. In: Computational Logic - CL 2000; First International Conference. Volume 1861 of Lecture Notes in Computer Science., Springer (2000) 553–567
17. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design, IEEE Press (2001) 279–285
18. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. In Johnson, D., Trick, M., eds.: Cliques, Coloring and Satisfiability. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996) 559–586