New Advances in Compiling CNF to Decomposable Negation Normal Form

Adnan Darwiche[1]

Abstract. We describe a new algorithm for compiling conjunctive normal form (CNF) into Deterministic Decomposable Negation Normal (d-DNNF), which is a tractable logical form that permits model counting in polynomial time. The new implementation is based on latest techniques from both the SAT and OBDD literatures, and appears to be orders of magnitude more efficient than previous algorithms for this purpose. We compare our compiler experimentally to state of the art model counters, OBDD compilers, and previous CNF2dDNNF compilers.

1 INTRODUCTION

A tractable logical form known as *Deterministic, Decomposable Negation Normal Form,* d-DNNF, has been proposed recently, which permits some generally intractable logical queries to be computed in time polynomial in the form size [4, 5, 6]. These queries include clausal entailment; model counting; model minimization based on model cardinality; model enumeration; and probabilistic equivalence testing. Most notably, d-DNNF is a strict superset of, and more succinct than, OBDDs [2], which are popular in supporting various AI applications, including diagnosis and planning. Moreover, although OBDDs are more tractable than d-DNNFs (support more polytime queries), the extra tractability does not appear to be relevant to some of these applications.

An algorithm has been presented in [4, 5] for compiling Conjunctive Normal Form (CNF) into d-DNNF. The algorithm is structure–based in two senses. First, its complexity is dictated by the connectivity of given CNF, with the complexity increasing exponentially with increased connectivity. Second, it is insensitive to non–structural properties of the given CNF: two CNFs with the same connectivity are equally difficult to compile by the given algorithm. However, many CNFs of interest—including random CNFs and those that arise in diagnosis, formal verification and planning domains—tend to have very high connectivity and are therefore outside the scope of this structure–based algorithm. This problem has been addressed in [6], which presents a CNF2dDNNF compiler that is structure-based, yet is sensitive to the non–structural properties of a CNFs. The compiler is based on the one presented in [4] but incorporates a combination of additional techniques, some are novel, and others are well known in the SAT and OBDD literatures. The compiler of [6] was the first CNF to d-DNNF compiler that practically matched some of the expectations set by theoretical results on the comparative succinctness between d-DNNFs and OBDDs [7].

We present a third–generation compiler in this paper for converting CNF into d-DNNF, which incorporates two key new techniques: conflict-directed backtracking and a new method for caching intermediate results. We show that the new compiler can be orders of magnitude more efficient than the compiler of [6] on problems that have been solved before. We also point to a number of CNF benchmarks that could be compiled for the very first time using the new compiler.

[1] Computer Science Department, University of California, Los Angeles, CA 90095, USA, email: darwiche@cs.ucla.edu

2 Deterministic DNNF

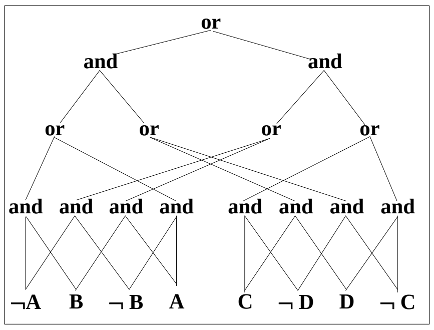
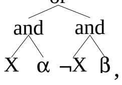


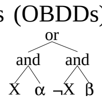
Figure 1. A negation normal form.

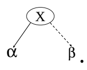
A negation normal form (NNF) is a rooted directed acyclic graph in which each leaf node is labeled with a literal, $true$ or $false$, and each internal node is labeled with a conjunction $\wedge$ or disjunction $\vee$. Figure 1 depicts an example. For any node n in an NNF graph, $Vars(n)$ denotes all propositional variables that appear in the subgraph rooted at n, and $\Delta(n)$ denotes the formula represented by n and its descendants. A number of properties can be stated on NNF graphs:

- Decomposability holds when $Vars(n_i) \cap Vars(n_j) = \emptyset$ for any two children n_i and n_j of an and-node n. The NNF in Figure 1 is decomposable.
- Determinism holds when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j of an or-node n. The NNF in Figure 1 is deterministic.
- Decision holds when the root node of the NNF graph is a decision node. A *decision node* is a node labeled with $true$, $false$, or is an or-node having the form , where X is a variable, α and β are decision nodes. Here, X is called the *decision variable*

of the node. The NNF in Figure 1 does not satisfy the decision property since its root is not a decision node.

- Ordering is defined only for NNFs that satisfy the decision property. Ordering holds when decision variables appear in the same order along any path from the root to any leaf.

Satisfiability and clausal entailment can be decided in linear time for decomposable negation normal form (DNNF) [4]. Moreover, its models can be enumerated in output polynomial time, and any subset of its variables can be forgotten (existentially quantified) in linear time. Deterministic, decomposable negation normal form (d-DNNF) is even more tractable as we can count its models given any variable instantiation in polytime [5, 7]. Decision implies determinism. The subset of NNF that satisfies decomposability and decision (hence, determinism) corresponds to Free Binary Decision Diagrams (FBDDs) [8]. The subset of NNF that satisfies decomposability, decision (hence, determinism) and ordering corresponds to Ordered Binary Decision Diagrams (OBDDs) [2, 7]. In OBDD notation, however, the NNF fragment 

is drawn more compactly as 

. Hence, each non-leaf OBDD node generates three NNF nodes and six NNF edges.

Immediate from the above definitions, we have the following strict subset inclusions OBDD $\subset$ FBDD $\subset$ d-DNNF $\subset$ DNNF. Moreover, we have OBDD $>$ FBDD $>$ d-DNNF $>$ DNNF, where $>$ stands for “less succinct than.”[2] OBDDs are more tractable than DNNF, d-DNNF and FBDD. General entailment among OBDDs can be decided in polytime. Hence, the equivalence of two OBDDs can be decided in polytime. The equivalence of two DNNFs cannot be decided in polytime (unless P=NP). The equivalence question is still open for d-DNNF and FBDD, although both support polynomial probabilistic equivalence tests [1]. For a comprehensive analysis of these forms, the reader is referred to [7].

3 THE COMPILER

We will discuss in this section an algorithm for converting a CNF into an equivalent d-DNNF. We will refer to our algorithm as a “compiler” since the resulting d-DNNF can be used for efficient, mostly linear, inference for answering a wide variety of queries, including model counting.

The two main advances underlying our proposed compiler are (1) the use of conflict-directed backtracking with clause learning [11, 10], and (2) a new method for characterizing the state of clauses, to be used as keys for indexing into a cache. We will first describe both of these techniques and then show how they are integrated into the new compiler.

3.1 A Recursive DPLL Implementation

Our new compiler utilizes a DPLL–based SAT solver that we developed especially to support the compilation task. The solver is based on state–of–the–art solvers, zchaff in particular [10], but the basic DPLL procedure is implemented recursively instead of iteratively. Iterative implementations of DPLL are now the standard in SAT solvers, but we found the utilization of iterative implementations for our purpose to be more complex.

Our implementation of DPLL is based on the following primitive procedures: decide(l), undo-decide(l), at-assertion-level(), and assert-cd-literal(), where l is a literal. Algorithm 1 illustrates these procedures by using them to implement a SAT solver that utilizes conflict–directed backtracking with clause learning. Here’s a brief description of these primitives:

Algorithm 1 sat(V)

```
1: if $V$ is empty then
2:     return $true$
3: choose a variable $v$ from $V$
4: choose literal $l$ of variable $v$
5: if decide($l$) and sat($V \setminus \{v\}$) then
6:     undo-decide($l$)
7:     return $true$
8: undo-decide($l$)
9: if at-assertion-level() then
10:    return assert-cd-literal() and sat($V \setminus \{v\}$)
11: return $false$
```

- decide(l) will set literal l to true and mark the variable of l as a *decision* variable, and assign it a *decision level:* a number which is incremented each time a new decision is made. decide(l) will then apply unit resolution which would potentially *imply* other literals. decide(l) succeeds if no contradiction is discovered by unit resolution, otherwise, it will fail after having constructed a conflict–driven clause as described in [10]. The mentioned method will construct a conflict–driven clause which is also an asserting clause, in the sense that adding this clause to the knowledge base will lead to implying the negation of literal l, $\neg l$, which is known as a conflict–driven assertion. Another side effect of a failing decide(l) is to compute the *assertion level*, which is the second largest level for any literal in the conflict–driven clause.
- undo-decide(l) will erase the decision l and all other literals that were derived by unit resolution after having asserted l. The current decision level will also be decremented.
- at-assertion-level() is a predicate that succeeds if the current decision level equals the assertion level computed by the last call to decide(l)
- assert-cd-literal() will add the conflict–driven clause constructed by the last call to decide(l). This will in turn lead to implying $\neg l$, the conflict–driven assertion. Unit resolution will then be applied which would potentially imply new literals. This may also lead to discovering a contradiction in which case assert-cd-literal() will fail, after having constructed a conflict–driven clause and computed a new assertion level (just like a call to decide()).

Some key points to observe regarding the above implementation. First, every set literal is either set by a decision or by an implication. This distinction is crucial for our caching scheme which we discuss later. Second is the notion of a “current decision level” which is the level assigned to the last decision made by the algorithm. All literals implied by unit resolution based after a decision is made will assume the same level of that decision (although the level will be called an implication level instead of a decision level).

3.2 Decomposing CNFs

The main technique that underlies our CNF to d-DNNF compiler is that of decomposition. Specifically, given a CNF Δ, we partition its clauses into Δ^l and Δ^r. If Δ^l and Δ^r share no variables, we can then compile them independently and simply conjoin the results. Suppose, however, that the two sets turn out to share a variable v. We will then

[2] That DNNF is strictly more succinct than d-DNNF assumes the non-collapse of the polynomial hierarchy [7].

use what is known as Boole's or Shannon's expansion:

$$\Delta = v \wedge ((\Delta^l \wedge \Delta^r)|v) \bigvee \neg v \wedge ((\Delta^l \wedge \Delta^r)|\neg v),$$

where $(\Delta^l \wedge \Delta^r)|v$ represents the result of replacing the occurrences of variable v with $true$ in $\Delta^l \wedge \Delta^r$, and $(\Delta^l \wedge \Delta^r)|\neg v$ represents the result of replacing the occurrences of variable v with $false$ in $\Delta^l \wedge \Delta^r$.

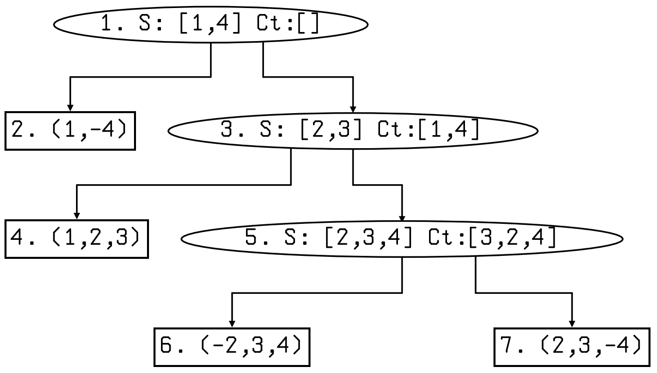


Figure 2. A decomposition tree for a CNF. Each leaf node is annotated with a distinct clause in the CNF. Each internal node n is annotated with a separator S (variables shared between clauses in left subtree of n and clauses in the right subtree of n), and a context Ct (variables shared between clauses below node n and clauses not below node n).

To control this decomposition process, we use a *decomposition tree* (dtree). A dtree for a given CNF is a binary tree whose leaves are tagged with the CNF clauses—see Figure 2. Our compiler constructs a decomposition tree for the given CNF as discussed in [6]. A dtree provides a recursive decomposition of the given CNF: the root t of the dtree partitions the CNF into two sets of clauses, those corresponding to its left and right children, $t.left$ and $t.right$. Each one of these children is then a root of a smaller dtree which recursively partitions its corresponding set of clauses.

The *separator* of dtree t is the set of all variables that are shared by clauses in $t.left$ and $t.right$. The main function of our compiler is to instantiate enough of the separator variables for t so as to decompose the clauses in $t.left$ and $t.right$; that is, get to a point where no variables are shared between these sets of clauses. At this point, the compiler can compile the two sets independently and simply conjoin the results to obtain a compilation for the clauses in t. Two important points to observe here. First, there is no need to set all variables in the separator for t to decompose the clauses in $t.left$ and $t.right$. Specifically, after setting some of these variables, enough clauses may be subsumed that the rest of the separator variables may no longer be shared between $t.left$ and $t.right$. This observation was indeed made in [6], who called for the re-computation of the separator for t each time a new variable is decided. We do indeed follow this strategy here as it is extremely effective. However, the addition of conflict–driven clauses does complicate this strategy a bit: the number of conflict–driven clauses maybe too large (at least in comparison to the initial set of clauses) that we cannot afford to perform the computation based on original and added clauses. In fact, a dtree is constructed once for the initial set of clauses and it would be computationally prohibitive (and algorithmically complex) to construct a new dtree each time a conflict–driven clause is added.

The good news, however, is that the added conflict–driven clauses are redundant and can therefore be ignored as long as one is careful in handling the following potential complication. Suppose that we have a dtree t, where y is the only variable shared between the children $t.left$ and $t.right$. If we set enough variables and subsume enough clauses, then y may no longer be shared between the children of t (the separator of t becomes empty). One outcome of this decomposition is this: when applying unit resolution to clauses in $t.left$ we can never derive literals that appear in $t.right$ and vice versa. This actually will no longer be true in the presence of conflict–driven clauses. For example, suppose that conflict–directed backtracking ends up adding the clause $x \vee z$ to the above example, where x appears in $t.left$ and z appears in $t.right$. This clause will not appear in the dtree and will therefore not be accounted for by separator decomposition. Now, if in the process of recursing on $t.left$, we set x to $false$, unit resolution will derive z which appears in $t.right$.[3] This is taken care of on Line 1 of Algorithm 2—to be discussed later—which states that when working on a dtree, we should collect literals that only pertain to that dtree, to avoid collecting literals that are derived by conflict–driven clauses about other dtrees (that would have been disconnected from the current dtree if we were not to have conflict–driven clauses).

3.3 CNF Caching

The last key topic we need to discuss before presenting our compiler is the way we cache our results. Specifically, our compiler will be trying to compile the same subset of clauses, say $\Delta = \alpha_1, \ldots, \alpha_m$, but under different variable settings, say, $\gamma = l_1, \ldots, l_m$, where each l_i is a literal. We will use $\Delta|\gamma$ to denote the set of clauses Δ after having set the values of variables as given by γ. The key point here is that even though we may have two different variable settings γ and γ', the CNFs $\Delta|\gamma$ and $\Delta|\gamma'$ may be equivalent. Recognizing this equivalence is critical to the efficiency of our compiler as it will allow it to avoid redundant compilations. The compiler given in [4, 5] uses the identity of variable instantiations γ and γ' as the key to the cache. This clearly fails to recognize some equivalences, yet it was sufficient to prove some useful bounds on the complexity of proposed algorithm. An improvement on this was given in [6], where the CNFs $\Delta|\gamma$ and $\Delta|\gamma'$ where each captured using a bit vector, with one bit for each literal in a CNF. If a clause is subsumed, all its bits are set to 0, otherwise, only the bits of resolved literals are set to 0. The bit vector is then used as the key to a cache. This more sophisticated scheme was claimed to be a critical reason for the efficiency of the compiler given in [6], allowing it to compile CNFs that could never be compiled before. We will now actually describe an even more sophisticated caching scheme which significantly improves the scalability of our compiler.

The new caching scheme will be described in three steps. First, suppose that we have a set of clauses Δ and a specific set of variables V, and let γ and γ' be two instantiations of all variables in V. Consider now the CNF $\Delta|\gamma$, which results from replacing variables V in Δ with $true/false$ according to instantiation γ. Our first goal is to find a way to efficiently test whether $\Delta|\gamma$ and $\Delta|\gamma'$ are logically equivalent. We first observe that each clause α in Δ can be in only one of two states in $\Delta|\gamma$. That is, either $\alpha|\gamma$ is satisfied (reduces to $true$) if one of the literals of α is satisfied by γ, or $\alpha|\gamma$ is not satisfied in which case all literals of α whose variables are in V must be falsified by γ and removed from α. The key point is that the state of clause

[3] In this case, the literal z must be implied by the current clauses in $t.right$ and, hence, the derivation of z by unit resolution on conflict–driven clauses can be safely ignored. Recall that conflict–driven clauses are redundant: anything that is implied using these clauses is also implied using only the original clauses.

$\alpha|\gamma$ can be uniquely determined by knowing only whether $\alpha|\gamma$ reduces to $true$ and without knowing the identity of instantiation γ: all we need is the identity of variables V. This implies that we can capture the state of CNF $\Delta|\gamma$ using a bit vector, with one bit per clause α in Δ to indicate whether $\alpha|\gamma$ is satisfied. We will use $bv1(\Delta,\gamma)$ to denote that bit vector. Our first technique is then to use this bit vector as the key to a cache. That is, to test whether $\Delta|\gamma$ is equivalent to $\Delta|\gamma'$ we just need to test whether $bv1(\Delta,\gamma) = bv1(\Delta,\gamma')$, assuming that γ and γ' are complete instantiations of variables V.[4]

This technique would work only if γ and γ' both instantiate the same set of variables V. This may not be true in our compiler as we shall see next, since γ and γ' may instantiate different subsets of V. We can address this by including another bit vector in the key which includes the state of variables V, whether instantiated or not (the specific values do not matter). We will use $bv2(V,\gamma)$ to denote the additional bit vector and we can now state that $\Delta|\gamma$ is equivalent to $\Delta|\gamma'$ if $bv2(V,\gamma) = bv2(V,\gamma')$ and $bv1(\Delta,\gamma) = bv1(\Delta,\gamma')$.

Our algorithm actually takes the above technique a step further by taking advantage of unit resolution. We first note that if β is a set of literals implied by $\Delta|\gamma$, then $\Delta|\gamma \equiv \beta \wedge \Delta|\gamma\beta$. Our algorithm will actually compile $\Delta|\gamma\beta$ instead of $\Delta|\gamma$, where β is the set of literals derived by unit resolution from $\Delta|\gamma$, and then conjoin the result with β. That is, before it tries to compile a subset of clauses Δ which are known to have been conditioned on γ, the algorithm will first collect all literals β that are derived by unit resolution from $\Delta|\gamma$. It will then lookup the cache using the keys $bv2(V,\gamma\beta)$ and $bv1(\Delta,\gamma\beta)$. If an entry is found, it will be conjoined with β and returned as the result of compiling $\Delta|\gamma$.

3.4 From CNF to d-DNNF

We are now ready to present our compiler which is given in Algorithm 2. Let's ignore Line 1 for now and consider Line 2 where the separator s for dtree t is computed. If this is empty, the dtree t is already decomposed (current clauses in $t.left$ and $t.right$ share no variables) and recursive calls are made on Line 4. If the separator is not empty, a variable v is chosen from s on Line 6 (we choose the variable that appears in the largest number of un-subsumed clauses). We then consider two cases, corresponding to the positive and negative literals of variable v. Lines 7-15 and Lines 16-24 are basically symmetric except that each deals with one literal of v. Consider now the positive literal v. We first call decide(v) on Line 8, which asserts the literal as a decision and runs unit resolution. If that call succeeds we make a recursive call on Line 9. The test $p = false$ on Line 11 will succeed for two reasons: either because decide(v) failed on Line 8 or because we got a failure deeper in the DPLL tree after making the recursive call on Line 9. In either case, we know that a conflict–driven clause and an assertion level have been computed. Line 12 checks whether we are currently at the assertion level, in which case the conflict–driven clause is added to the CNF, the conflict–driven literal is implied, and unit resolution is applied again to imply any further literals. If all of this succeeds, we try again to compile the dtree, except that we have implied some more literals now, which may in turn simplify the separator further.

Note that Line 25 will be reached only if the two cases for variable v have succeeded, in that each have returned a d-DNNF which is not $false$. If either case fails, however, backtracking takes place. This is very different from conflict–directed backtracking in classical SAT solvers, which would not backtrack if one of the cases have

[4] See [9] for another use of this technique for compiling CNFs into OBDDs.

Algorithm 2 cnf2ddnnf(dtree t): returns the root of a d-DNNF which is equivalent to the conjunction of clauses stored at leafs of dtree t.

```
1:  $term$ = all newly implied (not decided) literals of dtree $t$
2:  compute current separator $s$ of dtree $t$
3:  **if** $s$ empty **then**
4:    return conjoin($term$,cnf2-aux($t.left$),cnf2-aux($t.right$))
5:  **else**
6:    select a variable $v$ from separator $s$
7:    $p = false$          /* positive case */
8:    **if** decide($v$) **then**
9:      $p$ =cnf2ddnnf($t$)
10:   undo-decide($v$)
11:   **if** $p = false$ **then**
12:     **if** at-assertion-level() and assert-cd-literal() **then**
13:       return cnf2ddnnf($t$)          /* try again */
14:     **else**
15:       return $false$          /* backtracking */
16:   $n = false$          /* negative case */
17:   **if** decide($\neg v$) **then**
18:     $n$ =cnf2ddnnf($t$)
19:   undo-decide($\neg v$)
20:   **if** $n = false$ **then**
21:     **if** at-assertion-level() and assert-cd-literal() **then**
22:       return cnf2ddnnf($t$)          /* try again */
23:     **else**
24:       return $false$          /* backtracking */
25:   return conjoin($term$,disjoin(conjoin($v$,$p$),conjoin($\neg v$,$n$)))
```

succeeded. The reason we backtrack in this case, even if one of the cases have succeeded, is due to our desire to assert the conflict–driven clause, and any literals that may be implied by it, and this will have to be done at the assertion level [10].

Note also that after calling decide(), unit resolution may imply further literals. All newly derived literals (those resulting from the last call to decide()) are collected on Line 1 into a term, which is conjoined with the results on Line 4 or Line 25, depending on which case materializes. Note here that only literals set by an implication are collected on Line 1, since literals set by a decision are accounted for on Line 25.

It is important to note two key points about caching as handled by the proposed algorithm. First, the cache is checked only when a dtree node is first visited; that is, the cache is not checked each time a separator variable is set as that leads to overhead that turns out not to be worthwhile. Second, inconsistent CNFs (those that compile into $false$) are not cached. There are two reasons for this: (1) most of this type of caching is handled by conflict–driven clauses which provide a form of caching as they allow unit resolution to detect inconsistent CNFs that may have not been detectable before the clauses were added; (2) the algorithm's correctness depends on the assumption that whenever p on Line 9 or n on Line 18 are set to $false$, a conflict–driven clause and an assertion level are computed, which are referenced on Lines 12 and 21.

We finally point out that the functions conjoin() and disjoin() use the unique-node technique from the OBDD literature in the sense that they will not construct a new and/or-node which already exists (have the same children). This is done by keeping unique-node tables for and-nodes and for or-nodes. We actually do the same for term-nodes: an and-node whose children are all literals. In fact, our implementation has three types of nodes: binary and-nodes (two children), binary or-nodes and term-nodes.

Algorithm 3 cnf2-aux(dtree t)

```
1:  if $t$ is a leaf dtree then
2:      return a d-DNNF corresponding to current clause at $t$
3:  else
4:      compute key $key$ of CNF corresponding to dtree $t$
5:      if cache($key$) != null then
6:          return $cache(key)$
7:      else
8:          $r$ = cnf2ddnnf(t)
9:          if $r! = false$ then
10:             cache($key$)=$r$
11:         return $r$
```

4 EXPERIMENTAL RESULTS

We now consider experimental results which illustrate the effectiveness of our proposed compiler. Our implementation is in Standard C, and our experiments were run on a 1.6GHz Intel processor, with 2GB of RAM. We will compare our compiler to the state–of–the–art compiler presented in [6] for the same purpose. Since the successful compilation of CNF into d-DNNF allows us to count the models of given CNF, our compiler is already a model counter. We therefore compare the performance to the state-of-the-art model counter, RELSAT[5]. The succinctness of d-DNNF in comparison to OBDDs has already been discussed in [6], which presents a number of benchmarks for which an OBDD could not be constructed using a state–of–the–art OBDD package (CUDD *http://vlsi.colorado.edu/ fabio/CUDD/*), yet could be compiled into d-DNNFs.

We start with Table 1 which lists some benchmarks from SATLIB.[6] These are the most difficult ones from SATLIB that [6] reported on—results from [6] are in parentheses. The table shows that the new compiler is an order of magnitude more effective on flat200, and about three times more effective on uf200.[7] RELSAT average for flat200 was 8 seconds, and for uf200 was 2 seconds and, hence, does better on these problems.

Name	Vars/Clause	d-DNNF edges	Time (sec)
uf200	200/860	5774 (19273)	13 (37)
flat200	600/2237	7923 (46951)	50 (636)

Table 1. CNF benchmarks from SATLIB. uf200 and flat200 contains a 100 instances each; we report the average over all instances.

We now turn to Table 2 which lists some very difficult CNFs obtained from the ISCAS85[8] and ISCAS89[9] circuit suites. These circuits were reported on in [6], but our proposed compiler is at least an order of magnitude more efficient on them. Except for c432, which it finishes in 1 second, RELSAT could not count the models of any of the other CNFs within a cutoff time of few hours. We also point the reader to [3], which employs the presented compiler in the domain of reasoning with probabilistic relational models. Most of the CNFs reported in [3], some having more than $100,000$ clauses, could only be handled using the compiler presented in this paper as they are not accessible to the compiler of [6].

[5] *http://www.almaden.ibm.com/cs/people/bayardo/vinci/index.html*
[6] *http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/*
[7] We have excluded the time for dtree generation as is done in [6]. The dtree generation algorithm employs randomness, so we generate a number of dtrees and pick the one with the smallest width.
[8] *http://www.cbl.ncsu.edu/www/CBL_Docs/iscas85.html*
[9] *http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html*

Name	Vars/Clause	d-DNNF edges	Time (sec)
c432	196/514	13767 (19779)	0 (6)
c499	243/714	2214814 (2919960)	6 (448)
c880	443/1112	20676927 (7949684)	80 (1893)
c1355	587/1610	2748340 (3295293)	15 (809)
c1908	913/2378	18376664 (12363322)	187 (5712)
s1423	748/1821	467935 (1132322)	3 (162)
s3330	1961/4605	2496907 (8889410)	13 (5853)

Table 2. CNFs for ISCAS85 and ISCAS89 circuits.

5 CONCLUSION

We presented a compiler for converting CNFs into d-DNNFs: a tractable logical form which allows polytime algorithms for clausal entailment, model counting, enumeration and minimization, in addition to a probabilistic test for equivalence. The new compiler utilizes non–chronological backtracking and learning of conflict–driven clauses, in addition to a new technique for characterizing the states of instantiated CNFs to be used in the process of caching compilations. Our compiler is shown to be orders–of–magnitude more efficient than the state–of–the–art compiler on some benchmarks, and was able to compile some CNFs that could not be compiled before.

REFERENCES

[1] Manuel Blum, Ashok K. Chandra, and Mark N. Wegman, 'Equivalence of free Boolean graphs can be decided probabilistically in polynomial time', *Information Processing Letters*, **10**(2), 80–82, (1980).

[2] R. E. Bryant, 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers*, **C-35**, 677–691, (1986).

[3] Mark Chavira, Adnan Darwiche, and Manfred Jaeger, 'Compiling relational Bayesian networks for exact inference', Technical Report D–139, Computer Science Department, UCLA, Los Angeles, Ca 90095, (2004). Submitted to PGM'04.

[4] Adnan Darwiche, 'Decomposable negation normal form', *Journal of the ACM*, **48**(4), 608–647, (2001).

[5] Adnan Darwiche, 'On the tractability of counting theory models and its application to belief revision and truth maintenance', *Journal of Applied Non-Classical Logics*, **11**(1-2), 11–34, (2001).

[6] Adnan Darwiche, 'A compiler for deterministic, decomposable negation normal form', in *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*, pp. 627–634, Menlo Park, California, (2002). AAAI Press.

[7] Adnan Darwiche and Pierre Marquis, 'A knowlege compilation map', *Journal of Artificial Intelligence Research*, **17**, 229–264, (2002).

[8] J. Gergov and C. Meinel, 'Efficient analysis and manipulation of OBDDs can be extended to FBDDs', *IEEE Transactions on Computers*, **43**(10), 1197–1209, (1994).

[9] Jinbo Huang and Adnan Darwiche, 'Using DPLL for efficient OBDD construction', in *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, (2004).

[10] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, 'Chaff: Engineering an efficient SAT solver', in *39th Design Automation Conference, Las Vegas*, (2001).

[11] Joao Silva and Karem Sakallah, 'Grasp—a new search algorithm for satisfiability', in *Proceedings of the International Conference on Computer Aided Design*, (1996).