BDD Construction for All Solutions SAT and Efficient Caching Mechanism

Takahisa Toda^{*}

Graduate School of Information Systems,
the University of Electro-Communications
1-5-1 Chofugaoka, Chofu, Tokyo Japan
takahisa.toda@is.uec.ac.jp

Koji Tsuda

Graduate School of Frontier Sciences, The
University of Tokyo, Kashiwa, Japan
Computational Biology Research Center,
National Institute of Advanced Industrial Science
and Technology (AIST), Tokyo, Japan
ERATO MINATO Discrete Structure Manipulation
System Project, Japan Science and Technology
Agency, Sapporo, Japan
tsuda@k.u-tokyo.ac.jp

ABSTRACT

We improve an existing OBDD-based method of computing all total satisfying assignments of a Boolean formula, where an OBDD means an ordered binary decision diagram that is not necessarily reduced. To do this, we introduce lazy caching and finer caching by effectively using unit propagation. We implement our methods on top of a modern SAT solver, and show by experiments that lazy caching significantly accelerates the original method and finer caching in turn reduces an OBDD size.

1. INTRODUCTION

Boolean satisfiability (*SAT*) is to decide if a Boolean formula is satisfiable. *SAT* is ubiquitous in computer science. In practice, one often wants to know not only a decision result, but also a solution, i.e., a satisfying assignment. Thus, most *SAT* solvers return a decision with a solution. Its fundamental task is to solve as many instances as possible in a realistic amount of time. To this end, various useful techniques such as unit propagation, conflict-driven clause learning, non-chronological backtracking have been developed.

The problem of computing all solutions, called *ALL-SAT*, is also important in various practical applications. Examples include unbounded model checking [9], reachability analysis [5], prime implicant generation [11], and so on. In recent years, a data mining framework based on the reduction to an *ALL-SAT* problem has been proposed [7]. Most *ALL-SAT* solvers are implemented on top of a *SAT* solver that searches one solution. Specifically, when a satisfying assignment is found, a solver is enforced not to halt and continue

searching until the formula becomes unsatisfiable. In order to prevent a solver from rediscovering satisfying assignments, *blocking clauses* are computed from assignments by taking their complement, and added to a clause database. Since the number of total satisfying assignments can be exponential, it is usual to consider partial ones by removing irrelevant literals.

In some applications such as data mining, computing all solutions is not an end in itself and it is necessary to perform some processing over generated solutions: say, one may want to select solutions at random, to restrict solutions to optimal ones with respect to some preference [3], or to perform set operations such as union, intersection, and difference. It is thus necessary to represent generated solutions so that they can be efficiently manipulated in a subsequent processing.

We focus on a *binary decision diagram (BDD)*, which is a graphical representation of Boolean functions in a compressed form [2]. It is space-efficient in practice, and furthermore it allows various logical operations and queries to be performed efficiently. Our approach is to construct a BDD directly from a Boolean formula in a *conjunction normal form (CNF)*, which is a standard input in *ALL-SAT*. The constructed BDD represents all total satisfying solutions of a given formula in a compressed form.

Existing BDD construction methods fall into two categories: BDD operation-based construction and top-down construction. The former method is classical yet major, and it is easy to implement, however it tends to explode on intermediate results when applying conjunction operations repeatedly. In recent years, the latter method has been studied intensively in different areas [1] [6]. In the context of knowledge compilation, Huang and Darwiche proposed a top-down construction method that can be implemented on top of a modern *SAT* solver [6]. Their basic idea is that while constructing a BDD downward, it uses a clever caching, called *cutset caching*, to decide if a new node can be safely merged with an existing node. However, a serious drawback is that their method requires frequent cache operations.

In this paper, we improve the BDD construction method of Huang and Darwiche by reducing as many cache operations as possible in a systematic way. This significantly reduces overhead introduced by cache operations. We furthermore propose an alternative cutset caching, called *finer*

^{*}The author is supported by JSPS KAKENHI Grant Number 26870011.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'15 April 13-17, 2015, Salamanca, Spain.

Copyright 2015 ACM 978-1-4503-3196-8/15/04...\$15.00.

<http://dx.doi.org/10.1145/2695664.2695941>

cutset caching, that uses the information provided by unit propagation, and present complexity results. To evaluate our method, we conduct experiments and show that lazy caching significantly accelerates the original method and finer caching in turn reduces an OBDD size.

This paper is organized as follows. We introduce the basic notions of BDDs in Section 2 and classify BDD construction methods in Section 3. We then present two efficient caching mechanisms in Sections 4 and 5. We present experimental results in Section 6, and Section 7 provides concluding remarks.

2. BINARY DECISION DIAGRAMS

A *binary decision diagram* (BDD) is a graphical representation of Boolean functions [2]. Figure 1 shows an example of a BDD. Exactly one node has indegree 0, which is called the *root* and displayed at the top. Each internal node f has a label and two children, which are indicated by the three fields $V(f)$, $LO(f)$, $HI(f)$ associated with f . Each node f has a variable index as its label $V(f)$. The children indicated by $LO(f)$ and $HI(f)$ are called the *LO child* and *HI child* of f , respectively. The arc to a LO child is called a *LO arc* and illustrated by a dashed arrow, while the arc to a HI child is called a *HI arc* and illustrated by a solid arrow. There are only two terminal nodes, denoted by $\top$ and $\perp$.

We assume that BDDs satisfy the following two conditions. They must be *ordered*: if a node u points to an internal node v , then $V(u) < V(v)$. They must be *reduced*: no further application of the following reduction operations is possible.

1. If there is an internal node u whose arcs both point to v , then redirect all the incoming arcs of u to v , and then eliminate u (Fig. 2(a)).
2. If there are two internal nodes u and v such that the subgraphs rooted by them are equivalent, then merge them (Fig. 2(b)).

In this paper, ordered reduced BDDs are simply called *BDDs*. Ordered BDDs that need not be reduced are distinguished with ordinary BDDs by calling them *OBDDs*.

Given a BDD, paths from the root to terminal nodes mean assignments to Boolean variables and the value of a Boolean function. That is, 1 is assigned to x_i if the HI arc of a node of label i is selected; otherwise, 0 is assigned to x_i . If a given path leads to $\top$, then the function that is represented by a BDD has value 1; otherwise, 0. According to the node elimination rule of BDDs, it follows that if there is no node of label i in a path, then the function value is independent of the value of x_i .

Remarkable properties of BDDs are that if a set S of variables and its order are fixed, then every Boolean function over S has a canonical BDD representation, meaning that different functions correspond to different BDDs. Thanks to the BDD reduction rules, BDDs tend to have much smaller size than other representations such as CNFs and DNFs. Once BDDs of given Boolean functions are constructed, equivalence checking and many other useful operations such as logical conjunction, disjunction and negation can be done in constant time or in time related to the size of an input BDD (see [2][8]).

3. BDD CONSTRUCTION

We consider the problem of constructing the BDD that represents a given CNF. Even deciding if a CNF is satisfiable is NP-complete. Since the constructed BDD represents all satisfying assignments, BDD construction turns out to be a hard problem. Nevertheless, it is still important to consider a practical method that can solve as many instances as possible in a realistic amount of time, because once BDDs are constructed, one can efficiently solve various hard problems such as optimization and enumeration over BDDs [8, pp.209-212].

Existing methods fall into two categories. A classical yet major method is to construct BDDs using BDD operations. Since a clause corresponds to a path-shaped BDD, we construct the BDD for each clause in a bottom-up fashion. This can be done in linear time. After that, we repeatedly apply conjunction operation to these BDDs, and finally obtain the BDD for a given CNF. Taking a single conjunction operation only requires time proportional to the product of input sizes on average [2], however its repetition tends to produce intermediate BDDs of huge size even though a final BDD has a much smaller size. Thus, it is likely that computation gets stuck with no output, meaning that we obtain no satisfying assignment of a CNF. On the other hand, a strong point is ease of implementation. It is known that variable ordering and the order of applying conjunction operation to clauses both significantly affect computation performance, and several heuristics of deciding a good ordering have been proposed.

The other method is to construct BDDs in a top-down fashion. It stands in contrast with the method above in several aspects. The construction proceeds as if a final BDD is directly constructed in depth-first order (or in breadth-first order) without computing intermediate BDDs. While a BDD is constructed downward, in order that a BDD is reduced, one has to instantly decide if a new node can be merged with some existing node, even though at this time its subgraph is not yet constructed. Since exact decision is hard, a weaker condition for equivalence is often considered instead. Accordingly, the constructed BDD is not fully reduced, which means that an OBDD is constructed.

Since we improve the top-down method proposed by Huang and Darwiche [6] in later sections, we illustrate their algorithm with examples. Suppose that we are given the CNF $\psi = C_1 \wedge \dots \wedge C_5$ illustrated in Fig. 3. For simplicity, we assume that a variable ordering of BDD follows that of the indices of given variables. We first attempt to assign 0 to x_1, x_2, x_3 in this order, which corresponds to creating the left-most path in Fig. 3. After the value of x_3 was assigned, the clause C_2 turns out to be unsatisfiable. Thus the LO arc of ③ must point to $\perp$. We then proceed to the next assignment in which the value of x_3 is changed to 1, and for the same reason, the HI arc of ③ also points to $\perp$.

Likewise, we repeat to examine assignments and extend an OBDD if they are satisfying assignments; otherwise, redirect to $\perp$. During the repetition, we have to take care when a new node is added, because it may be able to be merged with an existing node as shown in the gray area of Fig. 3. Consider that when we go along the path ① $\rightarrow$ ② $\dashrightarrow$ ③ $\rightarrow$, the clauses C_1, C_2, C_3 are all satisfied, and it suffices to consider only C_4 and C_5 in assignment of the 4-th or larger variables. The same situation happens in the path ① $\dashrightarrow$ ② $\rightarrow$ ③ $\rightarrow$. This implies that the two paths can be safely merged at

Figure 1: The BDD for the parity function $x_1 \oplus x_2 \oplus x_3$

Figure 2: The reduction rules

Figure 3: The CNF ψ with $C_1 = x_1 \vee x_2 \vee \neg x_3$, $C_2 = x_2 \vee x_3$, $C_3 = x_1 \vee x_3 \vee x_4 \vee x_5$, $C_4 = x_4 \vee \neg x_5 \vee x_6$, $C_5 = x_5 \vee \neg x_6$ and the incomplete OBDD. Cutsets are associated with arcs, where satisfied clauses are underlined.

node of label 4. On the other hand, when we go along the path $\textcircled{1} \dashrightarrow \textcircled{2} \rightarrow \textcircled{3} \dashrightarrow$, the clause C_3 is not yet satisfied, and we have to take C_3 into consideration as well. The only difference of these three cases is whether C_3 is satisfied or not yet satisfied. Note that C_4 and C_5 are excluded because no assignment with the 3-rd or less variables assigned values affect their satisfiability, and one can focus only on C_3 . This equivalence criterion can be formulated using the following notion.

Definition 1. The i -th *cutset* of a CNF ψ is the set of clauses C in ψ such that C has variables of indices j and k with $j \leq i < k$. The *cutwidth* of ψ is the maximum size of a cutset of ψ .

The equivalence criterion is as follows: given partial assignments with the $(i-1)$ -th or less variables assigned values, if they have the same set of satisfied clauses in the $(i-1)$ -th cutset, then one can safely merge the corresponding paths in an OBDD at node of label i . It should be noted that the criterion is sound but not complete: if two nodes meet the criterion, then the constructed subgraphs are equivalent; however, the reverse direction is not true. For example, consider the two nodes of label 4 in Fig. 3. They are pointed to by arcs with different tags, thereby do not meet the criterion, however after constructing their subgraphs, it turns out that they can be safely merged.

Algorithm 1 shows a pseudo code of the Huang-Darwiche method. The notation is summarized below. The cutset^{i-1}

denotes the $(i-1)$ -th cutset, and the value (cutset^{i-1}) is the bit-vector that represents the ordered partition of satisfied clauses and the other clauses in the $(i-1)$ -th cutset, where an *ordered partition* is a partition in which the order of blocks in a partition is important. Bit-vectors of level $i-1$ are stored in the same cache, denoted by cache_{i-1} . The symbol $\psi|_{x_i=0}$ denotes the CNF ψ after instantiating x_i with 0. The function $\text{node}(i, lo, hi)$ returns a new node f with $V(f) = i$, $LO(f) = lo$, and $HI(f) = hi$.

Algorithm 1 Construct an OBDD from a CNF.

```

function OBDD(CNF  $\psi$ , variable index  $i$ )
  if there is an inconsistent clause in  $\psi$  then
    return  $\perp$ 
  end if
  if there is no uninstantiated variable in  $\psi$  then
    return  $\top$ 
  end if
  if ( $r := \text{cache}_{i-1}[\text{value}(\text{cutset}^{i-1})] \neq \text{nil}$ ) then
    return  $r$ 
  end if
   $r := \text{node}(i, \text{OBDD}(\psi|_{x_i=0}, i+1), \text{OBDD}(\psi|_{x_i=1}, i+1))$ 
   $\text{cache}_{i-1}[\text{value}(\text{cutset}^{i-1})] := r$ 
  return  $r$ 
end function

```

Algorithm 1 runs in $O(sn2^w)$ time, where s denotes the size of a CNF, n the number of variables, w the cutwidth of

a CNF [6]. Furthermore, the reduction of an OBDD can be done in linear time [12].

A strong point of the Huang-Darwiche method is that it can return satisfying assignments that have been discovered in the form of an incomplete OBDD even if it is interrupted at any time before it ends. Furthermore, it can be implemented on top of a modern SAT solver [6], and we can benefit from SAT technologies such as unit propagation, conflict-driven clause learning, non-chronological backtracking. On the other hand, implementation becomes complicated. In each call of OBDD, it is necessary to create the bit-vector value (cutset^{i-1}) and perform cache lookup, which is expected to introduce large overhead.

4. LAZY CACHING

As mentioned in the previous section, Algorithm 1 requires frequent cache operations. This can cause serious performance deterioration. Indeed, practical instances tend to have a large number of variables, and a large portion of a vast search space is occupied with unsatisfying assignments. For such instances, we may have to perform cache operations, say, for thousands of variables in billions of unsatisfying assignments.

Algorithm 2 Return 1 if CNF is satisfiable; otherwise 0.

```

function DPLL(CNF  $\psi$ , assignment  $\nu$ )
  ( $\psi, \nu$ ) := propagate( $\psi, \nu$ )
  if  $\psi$  has empty clause then
    return 0
  else if  $\psi$  has no clauses left then
    return 1
  end if
  choose an unassigned variable  $x$ 
  return DPLL( $\psi|_{x=0}, \nu|_{x=0}$ )  $\vee$  DPLL( $\psi|_{x=1}, \nu|_{x=1}$ )
end function

```

In this section, we present a caching mechanism, called *lazy caching*, to reduce as many cache operations as possible. Our method exploits unit propagation, which is an essential technology in modern SAT solvers. We summarize necessary notions and terminology in advance. Algorithm 2 shows DPLL procedure that is a basic framework to solve satisfiability, where $\psi|_{x=v}$ denotes the ψ after instantiating x with v , and $\nu|_{x=v}$ the map obtained by defining as $\nu(x) := v$. We omit other commonly used SAT techniques for simplicity. Algorithm 2 receives a CNF and an empty assignment, and returns either 0 or 1. Like the previous OBDD construction, it searches candidate solutions while assigning values to variables, although this procedure halts when one solution is found. Look at Fig. 3. Suppose that we now have the partial assignment $\nu: x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0$. The C_3 then has all but one of literals assigned 0 and the remaining one, i.e. x_5 , is unassigned. Such a clause is called *unit clause*. Since we are interested in satisfying assignments, we have no choice but to assign 1 to x_5 . Since C_4 then becomes a unit clause, we have to assign 1 to x_6 in turn. This process is called *unit propagation*. The function propagate updates an assignment and simplifies a CNF by applying unit propagation. Variables assigned values as a result of unit propagation are called *implied variables*, while those assigned values through a variable selection heuristic are called *decision variables*.

We are now ready to describe our lazy caching mechanism. Since our method requires unit propagation, we implement Algorithm 1 on top of a SAT solver with unit propagation. The only difference from Algorithm 1 is the timing of performing cache operations. Specifically, we create bit-vectors and perform cache lookup only at the previous variable of each decision variable in the variable ordering of a BDD. Accordingly, when extending an OBDD by adding new nodes, we insert only bit-vectors that were created in lookup phase into cache. It should be noted that when evaluating clauses for computing the bit-vector value (cutset^{i-1}), we must not use the values of implied variables with indices larger than $i - 1$. Because, as mentioned in [6], the correctness of cutset caching hinges on the fact that other variables remain free when variables $x_1, \dots, x_{i-1}$ are instantiated (see Section 5 for details). If we take care of this, the correctness of our method is clear because we only reduce cache operations, and its effect is at best to increase the size of an OBDD.

It is expected that the more unit propagation takes place, the more implied variables increase. Accordingly, the number of cache operations is significantly reduced. A natural question is to what extent the size of an OBDD increases. To see this, we conducted experiments using commonly used SAT instances, where experimental results are provided in Section 6.

5. CACHING IN MORE CONSIDERATION OF UNIT PROPAGATION

Unit propagation plays an essential role in lazy caching, however the cutset caching itself does not use the information provided by unit propagation. In this section, we present an alternative cutset caching with unit propagation.

Let us first see an example in which considering unit propagation in cutset caching in a naive manner leads to a wrong result. In Fig. 3, let ν_2 and ν'_2 be the partial assignments defined as $\nu_2: x_1 \mapsto 1, x_2 \mapsto 0$ and $\nu'_2: x_1 \mapsto 1, x_2 \mapsto 1$. The ordered partitions of the 2-nd cutsets in ν_2 and ν'_2 respectively correspond to $\{\underline{C_1}, C_2, C_3\}$ and $\{\underline{C_1}, C_2, C_3\}$, where satisfied clauses are underlined. If we consider unit propagation in evaluation of these clauses, then C_2 becomes satisfied under ν_2 because C_2 is a unit clause. Thus, ν_2 and ν'_2 yield the same partition, however the corresponding paths must not be merged. Suppose for contradiction that these paths are merged. Consider the assignment that extends ν'_2 by further defining as $x_3 \mapsto 0, x_4 \mapsto 1, x_5 \mapsto 1$. This assignment clearly satisfies all clauses. Since the paths are merged, it follows that the assignment extending ν_2 in the same way becomes a satisfying assignment. However, this contradicts what unit propagation implies, that is, x_3 must have value 1 under ν_2 .

A lesson learnt from the example above is that we must consider not only partitions of cutsets but also the value of implied variables, and actually it is sufficient. However, since there can be a large number of implied variables, it seems impractical to consider all possible implied variables at each level. We thus restrict implied variables to those that are essential. Let ν_{i-1} be an assignment with the $(i - 1)$ -th or less variables assigned values. Observe that unit propagation under ν_{i-1} first takes place for clauses that have all but one literals assigned values by ν_{i-1} and the remaining one is unassigned. Subsequently, unit propagation repeatedly takes place as long as variables that are implied at the

previous stage yield unit clauses. Thus, implied variables at the first stage can be considered as essential ones because they completely determine all implied variables under ν_{i-1} . Note that the number of implied variables at the first stage is at most the size of the $(i-1)$ -th cutset.

We now present an alternative cutset caching, which is called *finer cutset caching* for convenience (see Fig. 4). We partition the $(i-1)$ -th cutset into two: clauses that are satisfied either by the values of the $(i-1)$ -th or less variables or by those of implied variables at the first stage of unit propagation, and the other clauses, where blocks are ordered in this order. We claim that this ordered partition (B_1, B_2) together with the values of implied variables at the first stage of propagation have sufficient information for a sound equivalence criterion. Indeed, by performing unit propagation for B_2 using the values of implied variables at the first stage of propagation, we obtain the values of all other implied variables and the ordered partition of satisfied clauses and not yet satisfied clauses, which are all sufficient information. For example, in Fig. 4, we have $(\{C_1, C_2, C_3\}, \{C_4, C_5\})$ and $x_4 \mapsto 0, x_6 \mapsto 1$. Since C_4 is a unit clause under the values of x_4 and x_6 , we obtain $x_5 \mapsto 1$, and from C_5 we obtain $x_7 \mapsto 0$.

The following two proposition states that our finer cutset caching only requires as much cost as the original cutset caching does except for a constant factor.

PROPOSITION 1. *Let ν_{i-1} be an assignment with the $(i-1)$ -th or less variables assigned values. The pair of an ordered partition and the values of implied variables at the first stage of unit propagation under ν_{i-1} in finer cutset caching can be encoded into a bit-vector of length at most twice the size of the $(i-1)$ -th cutset.*

PROOF. Let (B_1, B_2) be the ordered partition of the $(i-1)$ -th cutset under ν_{i-1} , and let S be the set of implied variables at the first stage of unit propagation. Introduce arbitrary total order in the $(i-1)$ -th cutset. For each variable $x \in S$, let $C(x)$ be the first clause in B_1 such that the variable with the maximum index in $C(x)$ is x . Such $C(x)$ exists in B_1 because by definition, there exists a unit clause having x as a variable with the maximum index, although $C(x)$ need not be a unit clause. For example, let us see Fig. 4, where clauses are ordered with respect to their indices. Under the assignment given in the caption, the unit clause C_2 is a candidate for x_4 , however since the preceding clause C_1 also has x_4 , we obtain $C(x_4) = C_1$. Selecting not C_2 but C_1 is important because we do not distinguish unit clauses with satisfied clauses, which makes our caching finer. In order to encode, assign one of four integers to each clause C as follows. Assign 0 if $C \in B_2$, assign 1 if $C = C(x)$ for some $x \in S$ assigned 0, assign 2 if $C = C(x)$ for some $x \in S$ assigned 1, and assign 3 otherwise. Clearly, this sequence of integers can be encoded into a bit-vector of length at most twice the size of the $(i-1)$ -th cutset. It should be noted that the sequence is uniquely determined, and there is no loss of information through the encoding, meaning that both (B_1, B_2) and S with assigned values can be recovered from the sequence. $\square$

PROPOSITION 2. *The bit-vector in Proposition 1 can be computed in $O(k)$ time, where $k = \sum_{C \in \text{cutset}^{i-1}} |C|$.*

PROOF. When we compute cutsets in the preprocessing phase, sort clauses in each cutset in increasing order with

respect to the maximum variable index in a clause. Since this is done only in the preprocessing phase, we do not include its cost. In order to assign one of the four integers to each clause as in the proof of Proposition 1, it is sufficient to scan all clauses once. Indeed, since clauses are sorted, clauses with the same maximum variable index are placed consecutively. While scanning clauses with the same maximum variable index, we assign 0 to a clause C if $C \in B_2$; otherwise, assign 3. We furthermore keep track of the first clause C in B_1 and the existence of a unit clause. We do this in order to change the assignment of C after we encounter with the last clause with same maximum variable index so that we reassign 1 or 2 to C if a unit clause has ever appeared. $\square$

THEOREM 1. *An output OBDD size of Algorithm 1 with finer cutset caching is less than or equal to that of the original cutset caching in the same variable ordering.*

PROOF. It is sufficient to prove that if the original cutset caching decides that two nodes can be safely merged, so does our cutset caching. Let (S_1, S_2) be the ordered partition of the $(i-1)$ -th cutset under an assignment such that S_1 consists of clauses that are satisfied by the values of the $(i-1)$ -th or less variables and S_2 consists of the other clauses. One can locate all clauses in S_2 such that all but one of literals are assigned 0 by the $(i-1)$ -th or less variables and the remaining one has an index larger than $i-1$. By moving all these clauses to S_1 , we obtain an ordered partition of finer cutset caching and the values of implied variables at the first stage of unit propagation. This means that the decision of the original cutset caching always implies that of finer cutset caching. $\square$

Figure 4: Under $x_1 \mapsto 1$, $x_2 \mapsto 1$, $x_3 \mapsto 0$, the C_1 is the only satisfied clause, and C_2 and C_3 are unit clauses before unit propagation. At the first stage of unit propagation, x_4 and x_6 are assigned 0 and 1, respectively.

6. EXPERIMENTAL RESULTS

To implement programs, we decided to use MiniSat [4], although Huang and Darwiche used Zchaff SAT solver [10]. This is because MiniSat is widely known as a minimal reference implementation of major techniques, which allows easier development not only in the current experiments but also in future work. Although we use a different solver, we follow the same implementation steps as instructed in [6], which consists of fixing a variable ordering, instructing the program to find all solutions by introducing blocking clauses, augmenting an OBDD whenever a solver finds a solution, and performing cache operations in place. The only exception is a cache insertion. It was mentioned in [6] that we should only cache nodes whose construction is complete, as there are also nodes that are partially constructed. An advantage of following this suggestion would be that equivalent nodes can be merged during compilation. Indeed, since cached nodes are all complete, their labels and children must be determined when they are cached, and thus in inserting a node, a hashing technique allows deciding if an equivalent node is cached. Hashing is useful in practice yet requires in worst-case $O(N)$ time, where N is the number of cache entries. Since we did not want to include any stochastic factor, we did not follow their suggestion. Since partially constructed nodes become complete later, our implementation instead caches them at the first time they are created, which does not affect performance.

Our programs are all deterministic and each result was taken in a single execution. All experiments were performed on a 2.13GHz Xeon®E7-2830 with 512GB RAM, running Red Hat Enterprise Linux 6.3. We compiled our code [13] with version 4.4.7 of the gcc compiler. We used 12 instances: 10 instances were those used in the experiment of [6], and the other 2 instances iscas89-s1196 and s1238 were newly added in order to compare using large instances. These instances were taken from SATLIB and ISCAS 89 circuits. As in [6], variable ordering of each instance was determined before experiments by using MINCE. Since MINCE is a heuristic method, the computed variable ordering is not exactly the same ordering, thereby OBDD sizes vary from those in [6].

According to Table 1, lazy caching significantly reduces cache lookup operations, while it keeps as many cache hits as original. This effect manifests itself as the acceleration in running time, shown in Table 2, and the suppression of rapid increase in an OBDD size, shown in Table 3. The rate

Table 1: The numbers of cache hits / cache lookups

CNF	original		lazy	
flat75-1	297 /	14,100	297 /	788
flat75-2	792 /	23,316	795 /	2,568
flat75-3	2,466 /	82,372	1,629 /	6,279
flat100-1	3,260 /	182,057	3,263 /	27,069
flat100-2	1,280 /	118,269	1,273 /	13,084
flat100-3	523 /	18,895	524 /	2,164
iscas89-s344	1,072 /	17,829	1,032 /	2,110
iscas89-s386	2,140 /	133,126	1,635 /	4,403
iscas89-s510	2,057 /	38,174	2,035 /	4,177
iscas89-s953	33,168 /	1,788,637	31,774 /	65,527
iscas89-s1196	65,924 /	2,893,527	66,452 /	140,142
iscas89-s1238	150,196 /	9,667,161	150,699 /	308,210

Table 2: Time comparison (sec)

CNF	original	lazy	finer	lazy+finer
flat75-1	0.07	0.01	0.11	0.01
flat75-2	0.06	0.02	0.11	0.02
flat75-3	0.33	0.05	0.53	0.06
flat100-1	0.87	0.24	1.56	0.26
flat100-2	0.44	0.17	0.76	0.19
flat100-3	0.07	0.02	0.13	0.03
iscas89-s344	0.02	0.01	0.04	0.01
iscas89-s386	0.54	0.03	1.00	0.04
iscas89-s510	0.15	0.01	0.29	0.02
iscas89-s953	10.95	0.64	19.50	0.81
iscas89-s1196	19.38	1.60	25.37	1.87
iscas89-s1238	99.67	4.31	104.90	3.25

of increase is at most around two or three times. On the other hand, although finer caching is a bit slower than the original method, the combination of lazy and finer caching is as fast as lazy caching only. According to Table 3, in comparison to the original method, OBDD sizes are halved by finer caching and not largely changed by the combination of lazy and finer cachings.

7. CONCLUDING REMARKS

We improved the BDD construction method of Huang and Darwiche by reducing as many cache operations as possible in a systematic way. We furthermore proposed an al-

Table 3: OBDD size comparison

CNF	original	lazy	finer	lazy+finer
flat75-1	13,250	23,661	11,701	21,525
flat75-2	19,502	44,256	17,558	44,256
flat75-3	67,118	123,235	48,744	88,407
flat100-1	113,407	361,736	102,818	360,293
flat100-2	48,309	97,534	43,428	95,342
flat100-3	13,446	30,261	10,875	21,801
iscas89-s344	16,758	24,988	15,131	24,988
iscas89-s386	129,303	208,920	116,985	197,292
iscas89-s510	35,907	55,025	33,426	52,591
iscas89-s953	1,751,688	2,774,968	1,497,155	2,336,533
iscas89-s1196	2,797,970	6,763,654	1,801,324	5,564,335
iscas89-s1238	9,489,858	17,157,980	4,685,484	9,430,235

ternative cutset caching that uses the information provided by unit propagation, and presented complexity results. We evaluated efficiency of our method by conducting experimental comparison and showed that lazy caching significantly accelerates the original method and finer caching in turn reduces an OBDD size.

8. REFERENCES

- [1] H. Andersen, T. Hadzic, J. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In C. Bessière, editor, *Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 2007.
- [2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, 1986.
- [3] E. Di Rosa, E. Giunchiglia, and M. Maratea. Computing all optimal solutions in satisfiability problems with preferences. In P. Stuckey, editor, *Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 603–607. Springer Berlin Heidelberg, 2008.
- [4] N. Eén and N. Sörensson. The MiniSat Page: MiniSat-C_v1.14.1. <http://minisat.se/MiniSat.html>. accessed on 15 Dec. 2014.
- [5] O. Grumberg, A. Schuster, and A. Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In A. Hu and A. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 275–289. Springer Berlin Heidelberg, 2004.
- [6] J. Huang and A. Darwiche. Using DPLL for efficient OBDD construction. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, pages 157–172, 2005.
- [7] S. Jabbour, L. Sais, and Y. Salhi. Boolean satisfiability for sequence mining. In *Proceedings of the 22Nd ACM International Conference on Conference on Information; Knowledge Management, CIKM '13*, pages 649–658, New York, NY, USA, 2013. ACM.
- [8] D. Knuth. *The Art of Computer Programming Volume 4a*. Addison-Wesley Professional, New Jersey, USA, 2011.
- [9] K. McMillan. Applying sat methods in unbounded symbolic model checking. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer Berlin Heidelberg, 2002.
- [10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference, DAC '01*, pages 530–535, New York, NY, USA, 2001. ACM.
- [11] C. Pizzuti. Computing prime implicants by integer programming. In *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pages 332–336, Nov 1996.
- [12] D. Sieling and I. Wegener. Reduction of obdds in linear time. *Information Processing Letters*, 48(3):139–144, 1993.
- [13] T. Toda. cnf2obdd version 1.0.0. <http://www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html>. accessed on 15 Dec. 2014.