Don't-Care Computation using k-clause Approximation

K. L. McMillan
Cadence Berkeley Labs

Abstract

Computation of the satisfiability and observability care sets for a sub-circuit in a Boolean network is essentially a problem of quantifier elimination in propositional logic. In this paper, we introduce a method of approximate quantifier elimination that computes the strongest over-approximation expressible using clauses of a given length. The method uses a Boolean satisfiability solver in a machine-learning framework. Experiments using the SIS system show that the method can produce useful care set information in cases where earlier approaches are prohibitively costly.

1 Introduction

Environmental don't-care information is an important source of flexibility in optimizing two-level logic functions within a larger circuit. The *care set* of a sub-circuit is the set of valuations of the sub-circuit's inputs that are produced by some primary input pattern, such that the sub-circuit's output is observable at the circuit primary outputs. Computing care sets is essentially a problem of *image computation.* We can characterize by a Boolean formula the set of primary input patterns that make a given sub-circuit output observable. We will call this the *observability condition.* The care set is the image of the observability condition with respect to the sub-circuit input function [9].

The problem of computing the image of a set with respect to a (vector) Boolean function has been extensively studied [3, 2, 1, 6]. Existing methods derive primarily from model checking applications and are based either on Binary Decision Diagrams (BDD's) or Boolean satisfiability (SAT) solvers. Despite considerable study, however, the computation is still prohibitively expensive for large circuits and for sub-circuits with many inputs.

To avoid the expense of an exact image computation, we propose a method of approximating the care set using clauses of up to a fixed length k. These clauses correspond to short cubes in the don't-care space. The rationale behind the method (for which we provide some experimental evidence) is that short don't-care cubes are more useful than longer ones in two-level logic optimization. To avoid generating and testing all the cubes of length k, we investigate a machine-learning approach that uses random sampling and SAT to synthesize the prime implicates of the care set with up to k literals. Experiments show that this approach is more robust than existing exact methods based on BDD's and SAT, and that it can produce significant reductions in literal count in cases where other methods are prohibitively expensive.

Related work. The SIS system computes don't-care information using an image computation approach based on BDD's [9]. This method computes *compatible observability don't care* (CODC) sets, which allow every node (sub-circuit) in a multi-level network to be optimized independently, but sacrifices optimization flexibility. Mishchenko and Brayton [7] introduced a technique of don't-care approximation by "windowing" (restricting the sub-circuit environment to a local subset of the entire circuit). This restriction makes it feasible evaluate the care set of every node independently, and avoids the need for CODC's. They also introduce the use of SAT-based quantifier elimination [6], augmented with random sampling, which they show is more robust than BDD-based quantifier elimination if the number of node inputs is less than about ten. However, because this method enumerates the minterms of the care set, it cannot be applied to nodes with a large number of inputs. The method presented here approximates the care set by restricting the length of clauses rather than the size of the environment (although it can be combined with windowing). This makes the method relatively insensitive to the number of node inputs. We show experimentally that it can extract useful don't-care information in cases where both the CODC method and the minterm enumeration method are impractical.

Notations We use P, Q to stand for Boolean formulas or circuits, and U, W to stand for sets of Boolean variables. We will consistently confuse a formula or circuit over variables U with with its truth function (a function in $(U \rightarrow \{\textsc{False}, \textsc{True}\}) \rightarrow \{\textsc{False}, \textsc{True}\})$ and also with its set of satisfying assignments (a subset of $U \rightarrow \{\textsc{False}, \textsc{True}\}$). We use $\neg P$ for the negation of P, $P \wedge Q$ for the conjunction of P and Q and $P \vee Q$ for the disjunction of P and Q. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals, while a *cube* is a conjunction of literals. A k-clause is a disjunction of k literals. A *minterm* over U is a cube containing exactly one literal over each variable in U. We will generally confuse a minterm with its unique satisfying truth assignment. We use l for literals, c for clauses, and σ for truth assignments or minterms. Otherwise, capital letters generally represent sets, while lower case letters represent individuals.

2 Approximate image computation

The image computation problem can be reduced to the problem of Boolean quantifier elimination or *projection*. That is, suppose we are given a Boolean function P over a set of Boolean variables U, and a vector Boolean function $f = (f_1, \ldots, f_n)$ also over U. The *image* of P with respect to f is the set of valuations of f over all the satisfying assignments of P. We can characterize this set as a Boolean function P' over a vector of variables $W = (w_1, \ldots, w_n)$, representing the outputs of the function:

$$P' \doteq \exists U.\ P \wedge (\bigwedge_{i=1}^{n} (w_i \iff f_i))$$

That is, P' is true of an assignment to W when there exists a valuation of U (the input to f) such that the value of f is equal to W. The problem of characterizing the image as a Boolean formula is to construct a formula equivalent to the above without quantifiers.

Notationally, it will be more convenient to speak of *projection* rather than quantification. That is, given a Boolean function Q and a set of variables W, we will say that $Q \downarrow W$, the projection of Q onto W, is $\exists W^c.Q$. That is, the projection is the result of existentially quantifying all Boolean variables *not* in W, leaving a formula dependent only on W. Another way to view projection is that $Q \downarrow W$ is the strongest Boolean formula over W that is implied by Q. Thus, the problem of computing $Q \downarrow W$ can be reduced to the computation of the *prime implicates* of Q over W.

An *implicate over* W of a Boolean formula Q is a clause over W that is implied by Q. A *prime implicate over* W is an implicate over W that is not implied by any other implicate over W. It is straightforward to show that the conjunction of the prime implicates over W of P is equivalent to $P \downarrow W$.

In this work we will be concerned with computing prime implicates having at most k literals. Since this is a subset of the prime implicates, it is an over-approximation of $P \downarrow W$ – the strongest over-approximation expressible as a conjunction of k-clauses.

A brute-force approach to this problem would be to enumerate all the k-clauses over W, testing whether each is implied by P using a SAT solver. Clearly, this will only be effective if $|W|$ is small and k is very small. Here, we will attempt to avoid this enumeration by applying the machine learning paradigm. In this approach, we have a teacher that produces a set of samples S from the solution space of P, and a learner that produces a theory T that accounts for these samples. The function of the teacher is to choose samples that contradict the learner's theory, and thus to force the learner to modify its theory.

In our case, the learner's theory T is the set of prime implicates over W of S, with k literals or fewer. In particular, if the set S of samples is empty, the learner's theory is the proposition $\textsc{False}$. The teacher may use multiple strategies. The simplest strategy is to generate random samples satisfying P. Since the learner's theory T is initially far too strong, many of these random samples will falsify T, and thus cause the learner to update T. A more sophisticated strategy uses a Boolean satisfiability (SAT) solver to generate satisfying assignments to $P \wedge \neg T$. The cost of finding such a sample is higher, but it is guaranteed to cause an update of T. The process ends when $P \wedge \neg T$ becomes unsatisfiable. At this point, T is the strongest conjunction of k-clauses over W implied by P.

To carry out this computation efficiently, we require an efficient means of updating T when a truth assignment is added to the sample set S. For this purpose, we use a trie representation of T. This allows us to quickly find the clauses that conflict with a new sample and to test whether a clause is subsumed by existing clauses.

In addition, when the theory T is updated, we need to be able to quickly update our representation of $P \wedge \neg T$ in the SAT solver. To do this, we use an incremental SAT solver, which allows us to copy updates to the trie structure representing T directly into the clause database of the SAT solver.

2.1 Prime implicates computation

In a classical prime implicates computation, we generate new clauses by resolving existing clauses (applying the consensus rule). A trie structure can be used to determine whether the new clause is subsumed (implied) by an existing clause, and to find those clauses that are subsumed by the new clause and must therefore be removed. Here, however, we do not use resolution. This is because we expect the formula P in clause form to contain hundreds to thousands of Boolean variables (one for each gate in a circuit). This makes a resolution approach impractical.

Instead, we generate a set S of samples from the satisfying assignments of P, and compute the prime implicates of this sample set. Since we are interested only in clauses over W, we project the samples onto W. Thus, each sample in S is a minterm over W, and we can think of the sample set S as a sum-of-products over W. Our problem is thus to incrementally compute the set of prime implicates of a sum-of-products as new minterms are added. Moreover, we wish to restrict the computation to prime implicates having k literals or fewer.

As in standard approaches to prime implicates computation, we will use a trie-like structure to represent the set of prime implicates. Typically, it is required that tries be *ordered*, in that the words they generate must be strictly increasing according to some total order on the alphabet. In our application, the tries are unordered. Moreover, we view the trie as a deterministic finite automaton with a tree-like state graph (not to be confused with a tree automaton!). In the usual definition of a trie, the terminal states of this automaton are exactly the leaves of the tree. Here, we allow leaves that are not terminal states. This makes it possible to represent the empty set of clauses, which otherwise cannot be represented.

For our purposes, a *trie* over W is a finite automaton $(\Sigma, V, E, \epsilon, F)$ where Σ, the alphabet, is the set of literals over W, V is the set of states, $E \subseteq V \times \Sigma \times V$ is the transition relation, $\epsilon \in V$ is the initial state and $F \subseteq V$ is the set of terminal states. We further require that the labeled directed graph (V, E) be a tree rooted at ϵ.

Every word over Σ can be viewed as a clause. Thus, we can think of the language of a trie T as a set of clauses. We will use the notation $[T]$ for the set of clauses $l_1 \vee \cdots \vee l_n$ such that the word $l_1 \cdots l_n$ is in the language of T. We will say that trie T is *reduced* when no clause in $[T]$ is implied by another clause in $[T]$ (that is, when all clauses in $[T]$ are prime with respect to $[T]$).

A fundamental operation on tries is to determine whether a given clause is implied (subsumed) by some clause in $[T]$. An efficient procedure for checking subsumption in ordered tries is given, for example, in [5]. Figure 1 shows a similar procedure for unordered tries. Here, c is a clause represented by the set of its literals. The ability to check subsumption efficiently will allow us to maintain a trie in reduced form.

Now suppose we have a set S of samples (minterms over W) and a trie T representing the prime implicates of S. Given a new sample σ, we wish to compute a trie T' representing the prime implicates of $S' = S \cup \{\sigma\}$. Figure 2 shows a procedure UPDATE for this purpose. In this procedure both truth assignments and clauses are represented as sets of literals. The recursive procedure UPDATEVTX searches the trie for conflicts – clauses in $[T]$ that are false under σ. Each time we encounter a conflict c, we replace c with the set of prime implicates of S' implied by c. This is done by the procedure EXTEND. We note that every implicate of S' must contain some literal in σ, since it must be true under σ. Thus, the only candidates for prime implicates implied by c are of the form $c \vee l$ where l is a literal in σ. Of course, if $\neg l$ is in clause c, $c \vee l$ is a tautology, so we discard this clause. Note also that we never generate subsumed clauses or clauses of length greater than k.

```
function SUBSUMEDVTX($T, c, v$)
    if $v \in F_T$ return true
    for all edges $(v, l, v') \in V_T$ such that $l \in c$ do
        if SUBSUMEDVTX($T, c, v'$) return true
    done
    return false

function SUBSUMED($T, c$)
    return SUBSUMEDVTX($T, c, \epsilon_T$)
```

Figure 1: Checking subsumption of a clause in a trie.

```
procedure EXTEND($T, v, L$)
    remove $v$ from $F_T$
    for each literal $l \in L$ do
        let $v'$ be a new vertex
        add $v'$ to $V_T$ and $F_T$
        add edge $(v, l, v')$ to $E_T$
    done
```

Figure 3: Extending a trie terminal.

This procedure maintains the invariant that all implicates of S' are implied by some clause in $[T]$. When the procedure completes, there are no conflicts, thus all clauses in $[T]$ are implicates, thus $[T]$ is the set of prime implicates of S'.

In the actual implementation of this procedure, we represent each non-terminal state v in the tree by a list of the pairs (l, v') such that $(v, l, v') \in E_T$. This makes it straightforward to recur over the tree. Sets of literals are represented by bit vectors. Also, in practice we can save space by deleting any non-

```
procedure UpdateVtx($T$,$\sigma$,$v$,$c$)
    if $v \in F_T$ then
        if $|c| < k$ then
            let $L = \{l \in \sigma \mid \neg l \notin c$ and not Subsumed$(T, c \cup \{l\})\}$
        else let $L = \emptyset$
        Extend$(T, c, L)$
    else for all edges $(v, l, v') \in V_T$ in $T$ do
        if $\neg l \in \sigma$ then UpdateVtx($T$,$\sigma$,$v'$,$c \cup \{l\}$)
    done
end UpdateVtx

procedure Update($T$,$\sigma$)
    UpdateVtx($T$,$\sigma$,$\epsilon_T$,$\emptyset$)
end Update
```

Figure 2: Trie update procedure.

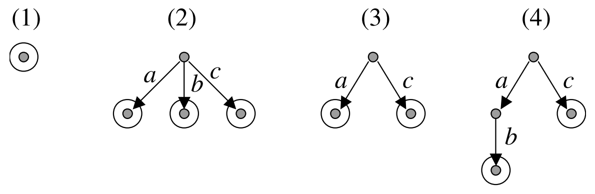

Figure 4: Updating the trie.

terminal state with an empty set of children, since such a state does not contribute to the language of T (and note the deletion of one state can lead to deletion of its parent). This is not shown in the figure for the sake of clarity.

As an example, Figure 4 shows a sequence of tries obtained by the procedure Update. We let $k = 2$. Trie (1) represents the set containing the empty clause (False). We add the sample $\{a, b, c\}$. Since the clause False is a conflict, we call Extend with the literals a, b and c from the sample, obtaining trie (2). This represents the three unit clauses (a), (b) and (c). Now we add the sample $\{a, \neg b, c\}$. The clause (b) is a conflict. In this case, however, there are no extensions, since the clauses $(b \vee a)$ and $(b \vee c)$ are subsumed (while $(b \vee \neg b)$ is a tautology). Thus, the clause (b) becomes a non-terminal with no children and we delete it, yielding trie (3). Finally, we add the sample $\{\neg a, b, c\}$. Now clause (a) is a conflict. We extend clause (a) only with b in this case, since $(a \vee c)$ is subsumed, yielding trie (4). This tree represents $(a \vee b) \wedge (c)$. Note, however, that if we had $k = 1$, this final extension would have been disallowed, resulting in the deletion of clause (a).

2.2 Generating samples

The procedure Update provides the learner in our teacher/learner framework. For the teacher, we need some means of generating useful samples. A useful sample is one that induces at least one conflict in T. Early in the computation, it is typically possible to generate useful samples randomly. However, as the learner's theory is refined, random samples become less useful. Thus, we use a SAT solver for sample generation. A naïve approach to this would be to simply convert the formula $P \wedge \neg T$ into a satisfiability equivalent CNF formula (here we let T stand for the conjunction of its clauses). We can then use a SAT solver to find a satisfying assignment. The projection of this assignment onto W is guaranteed to be a useful sample.

A more efficient approach is to use an incremental SAT solver, and to update the clause representation of $P \wedge \neg T$ "in place" to reflect changes in T. In this way, we avoid reconstructing the clause representation for every sample, and the SAT solver can reuse conflict clauses learned when computing previous samples.

We now define a representation Cnf of the trie T in conjunctive normal form. The clause set Cnf is satisfiable for a given assignment σ to W exactly when some clause in $[T]$ is false under σ (*i.e.*, when there is some conflict). For each vertex $v \in V_T$, we introduce fresh Boolean variables s_v and p_v. The variable s_v will be implied to be true when all the clauses represented by the subtree rooted at v are true. The variable p_v will be implied to be true when s_v is true, or when the label of the incoming edge of v is true (intuitively, p_v is the contribution of v to its parent).

The set CNF consists of the following clauses:

- For every non-terminal $v \in V$, the clause

$$(\bigvee\{\neg p_{v'} \mid (v,l,v') \in E\}) \vee s_v$$

- For every non-root $v' \in V$, with incoming edge (v,l,v'), the clauses $(\neg s_{v'} \vee p_{v'})$ and $(\neg l \vee p_{v'})$.
- The clause $(\neg s_\epsilon)$.

We can show by induction on the height of the tree that, for a given assignment to W, there is an assignment to the s_v and p_v variables satisfying CNF exactly when some clause in $[T]$ is falsified by W.

Now suppose we extend a terminal state $v \in V$ by a literal set L. The state v becomes non-terminal, and a new edge (v,l,v_l) is added for each $l \in L$. Figure 5 shows a procedure EXTENDCNF that updates CNF to reflect this change. The parameter E' is the set of new edges. Notice that this only involves adding clauses to CNF, not deleting clauses. Thus, we do not require from the incremental SAT solver the ability to delete clauses. Let us assume that the procedure EXTEND from the previous section is modified so that it also calls EXTENDCNF, passing the set E' of newly added edges. In the actual implementation, adding clauses is accomplished by calls to an incremental SAT solver. Note it is also possible to delete the clauses in CNF corresponding to deleted states of T, if the SAT solver supports this.

```
procedure ExtendCnf($T, v, E'$)
    for each $(v,l,v') \in E'$ do
        add clauses $(\neg s_{v'} \vee p_{v'})$, $(\neg l \vee p_{v'})$ to Cnf
    add clause $(\bigvee\{\neg p'_v \mid (v,l,v') \in E'\}) \vee s_v$
            to Cnf
```

Figure 5: Adding an edge to the CNF representation of the trie T.

Figure 6 shows a procedure SAMPLE that generates a new sample based on P and CNF. It assumes that the clause representation of P has been added to CNF. Thus CNF is satisfiable exactly when $P \wedge \neg T$ is satisfiable. A procedure RANDOMSAT is used to generate random satisfying assignments of P. The difficulty of this depends on the structure of P. Thus we postpone a discussion of generating random samples to the application section. The procedure SAT represents the SAT solver, and returns either a satisfying assignment for its argument, or the token UNSAT. The choice of whether to generate a random sample or apply the SAT solver is arbitrary. Ideally, this choice should be made so as to roughly balance the computational effort applied in the two methods. In the current implementation, the random approach is applied until 320 consecutive samples produce no conflict.

```
procedure Sample($P$, Cnf)
    let $\sigma$ = either RandomSat($P$) or Sat(Cnf)
    if $\sigma$ = Unsat return Unsat
    return $\sigma \downarrow W$.
```

Figure 6: Sample generation.

Now we are ready to put our teacher and our learner together into a procedure for computing the k-clause restricted prime implicates. The overall procedure is shown in figure 7. Our initial theory T contains only the empty clause, representing the proposition FALSE. The function TOCNF uses standard methods to convert a formula into a linear-size satisfiability-equivalent set of clauses. We initialize CNF so it contains the clause representation of P and $\neg T$. Then we loop, generating samples, and updating T until $P \wedge \neg T$ becomes unsatisfiable. Note that the procedure UPDATE has a side-effect on CNF, updating it to reflect any changes in T. As an optimization, we can postpone constructing CNF until the first SAT solver call. This avoids adding many unnecessary clauses during the random simulation phase.

```
procedure ApproxProject($P, W, k$)
    let $T$ be a one-state trie with $F = \{\epsilon\}$
    let Cnf = ToCnf($P$)
    add the clause $\neg s_\epsilon$ to Cnf
    while true do
        let $\sigma$ = Sample($P$, Cnf)
        if $\sigma$ = Unsat return $[T]$
        Update($T, \sigma$)
    done
```

Figure 7: Procedure for k-clause over-approximate projection.

3 Application to care set computation

Suppose we are given Boolean circuit over a set of primary input variables U, with output function vector $g = (g_1, \ldots, g_m)$. We identify a single-output subcircuit ϕ, with input function vector $f = (f_1, \ldots, f_n)$.

The observability condition P of the sub-circuit is

$$P \doteq \bigvee_{i=1}^{m} (g_i|_\phi \oplus g_i|_{\neg\phi})$$

That is, P is true whenever substituting true and false for the sub-circuit output yields differing values at the primary outputs. The care condition P' at the sub-circuit inputs is the image of P with respect to f, which is characterized over the variables $W = (w_1, \ldots, w_n)$ by

$$P' \doteq (P \wedge \bigwedge_{i=1}^{n} (w_i \iff f_i)) \downarrow W$$

We use the method of the previous section to compute a k-clause over-approximation of this projection. To generate random samples, we simply choose valuations of the inputs U with a uniform distribution. The valuation of W is then computed from the sub-circuit input function f. Input valuations that do not satisfy the observability condition are discarded. The usefulness of random sampling is thus determined by the density of the observability space. If ϕ is observable under a very small fraction of the input assignments, then random sampling will not be useful. In practice, however, we find random sampling to quite effective, producing the vast majority of the theory updates.

3.1 Empirical evaluation

The k-clause approximation method has been implemented in the SIS environment. The derived observability care set information is used to optimize the nodes of a multi-level network, using Espresso [8]. The dual of each clause in the care set approximation yields a cube in the don't-care set. We do not construct compatible observability don't care sets (CODC's). Rather, we visit each node in turn, computing a care set approximation for that node based on the current state of the network, and then optimizing the node before moving on to the next node.

As in [7], we use a "window" surrounding each node as its environment for approximating the observability care set. According to Mishchenko and Brayton's terminology, an $N \times N$ window contains nodes up to a distance N from the node to be optimized in both the forward (toward outputs) and backward (toward inputs) directions. Our procedure for generating an $N \times N$ window is precisely as in [7]. For benchmarks, we use the larger MCNC sequential benchmarks [10] (names beginning with s) and a subset of the ITC'99 benchmarks [4] (names beginning with b). These designs were pre-processed by running of version of *script.rugged* from the SIS distribution, leaving out the final step, which applies don't-care optimization using the *full_simplify* command. This script typically produces a few nodes with too many fan-ins to be processed using the *espresso* two-level logic minimizer [8]. For this reason, the SIS *elim* command was modified so that node x is never substituted into node y when y has more that 20 fan-ins. This results in nodes with up to about 30 fan-ins, and makes it possible to pre-process all the designs.

We perform two experiments on the pre-processed designs. In experiment 1, we make one pass over the network, simplifying each node with respect to the computed don't care cubes, with a given window size and a given value of k. Table 1 shows run times and resulting number of SOP literals as we increase k, for a 4×4 window. The column labeled "orig." gives the number of literals after pre-processing. We observe that k values greater than 2 produce no significant additional improvement. For each node, we put a limit on the SAT solver of of 10^7 total assignments to Boolean variables (either by decisions or by BCP). If this limit is reached, the care-set computation for the node is abandoned. In experiment 1, this occurred for approximately 0.003% of nodes.

We also observe that a robust ODC computation allows us to apply don't-care optimization in ways that would otherwise be too costly. In our second experiment, we use the same pre-processed benchmarks. These are then processed using a modified version of *script.rugged*. In the original script, the first two of the three simplification steps do not use don't-care information. In the modified script, all three simplification steps have been replaced with the k-clause limited ODC method. As a result, ODC simplification is applied both before and after algebraic factorization and resubstitution. Table 2 shows the results of experiment 2, again for a 4×4 window. Here, in a few cases, we observe a non-negligible improvement for $k > 2$. In this experiment, the rate of abandoned nodes is approximately 0.02%.

Table 3 compares the results obtained with the results of experiments 1 and 2 for a 4×4 window and $k = 4$. In the table *new 1* represents experiment 1 (one-pass simplification after preprocessing), while *new 2* represents experiment 2 (modified *script.rugged* after pre-processing). As we can see, the additional flexibility for simplification sometimes results in significantly reduced area in experiment 2, though run times are correspondingly higher (the run times include the entire script, though processing time is dominated by simplification).

By way of comparison, the columns labeled *fs* in

Name	SOP literals				Run time (s)		
	orig.	$k = 2$	$k = 3$	$k = 4$	$k = 2$	$k = 3$	$k = 4$
b14	9290	8783	8776	8798	38.8	42.9	47.9
b15	13411	12938	12873	12855	183.9	163.9	191.1
b17	47145	45586	44641	44785	397.3	418.1	462.5
b20	19946	18769	18541	18691	122.3	130.9	135.8
b21	22267	19884	20262	20752	140.3	163.6	180.9
b22	28519	27174	26591	27156	195.6	197.1	224.3
s13207.1	2852	2779	2782	2782	3.4	4.3	6.0
s15850.1	4241	4191	4180	4181	4.4	5.2	6.1
s35932	11222	11222	11222	11222	5.7	5.7	5.7
s38417	15470	15287	15266	15190	20.0	21.5	22.6
s38584.1	13056	12943	12872	12823	96.6	78.6	98.1

Table 1: One-pass simplification with 4×4 window.

Name	SOP literals				Run time (s)		
	orig.	$k = 2$	$k = 3$	$k = 4$	$k = 2$	$k = 3$	$k = 4$
b14	9290	7504	7443	7473	112.9	130.6	143.3
b15	13411	10428	10360	10396	343.2	436.5	629.9
b17	47145	36982	36017	36110	1470.5	1658.2	2121.0
b20	19946	15809	15453	15525	283.5	313.3	345.9
b21	22267	15994	15702	15628	301.8	357.7	406.8
b22	28519	24190	23759	23451	588.1	703.5	716.0
s13207.1	2852	2602	2556	2570	6.8	8.1	12.1
s15850.1	4241	3984	3993	3972	10.6	12.1	15.3
s35932	11222	11221	11221	11221	34.6	35.4	34.8
s38417	15470	14293	14223	13157	60.5	62.2	60.9
s38584.1	13056	12513	12478	12433	302.1	179.0	227.1

Table 2: Modified *script.rugged* with 4×4 window.

Name	SOP literals						Run time (s)				
	orig.	*fs*	*new 1*	*new 2*	*mt 1*	*mt 2*	*fs*	*new 1*	*new 2*	*mt 1*	*mt 2*
b14	9290	–	8798	7473	9024	7828	–	47.9	143.3	44.8	120.0
b15	13411	–	12855	10396	17695	11021	–	191.1	629.9	153.3	500.8
b17	47145	–	44785	36110	58491	37717	–	462.5	2121.0	496.9	1660.5
b20	19946	–	18691	15525	18566	16642	–	135.8	345.9	131.1	353.0
b21	22267	–	20752	15628	19370	17067	–	180.9	406.8	134.1	405.5
b22	28519	–	27156	23451	28077	25379	–	224.3	716.0	206.9	558.3
s13207.1	2852	2685	2782	2570	3162	2663	3.0	6.0	12.1	3.5	6.5
s15850.1	4241	4141	4181	3972	4130	4012	29.4	6.1	15.3	5.5	15.9
s35932	11222	10838	11222	11221	11222	11221	5182.4	5.7	34.8	5.5	33.2
s38417	15470	–	15190	13157	16234	13236	–	22.6	60.9	28.1	67.2
s38584.1	13056	12754	12823	12433	12868	12551	2960.9	98.1	227.1	8.2	54.6
ave.	17038	–	16294	13812	18076	14485	–	125	428	110	343

Table 3: Comparison of methods.

the Table 3 show the corresponding results for the SIS *full_simplify* command on the pre-processed designs. This command computes CODC's using Binary Decision Diagrams. A dash indicates that the method aborted because the number of BDD nodes exceeded the fixed limit of 4.8×10^5. As is clear from the table, *full_simplify* is only effective for the smallest designs. In one case (*s35932*) the BDD method produces a better result than experiment 2, though at a very high run time cost.

Finally, we also applied the exact SAT-based method of extracting don't care information as described in [7]. This method enumerates the minterms of the care set using random simulation and a SAT solver (and is essentially the same as the present method if we let $k = \infty$). It failed to terminate within two hours on any of the benchmark designs, even for a 2×2 window. This is not surprising, since a node with 30 inputs may have on the order of 10^9 minterms in its care set. As Mishchenko and Brayton observe, minterm enumeration is only feasible for nodes having up to about 10 inputs. In our last experiment, we restrict the SIS *elim* command to produce nodes with no more than 10 inputs. We then rerun experiments 1 and 2 (including pre-processing), using the exact minterm enumeration method. The results are shown in Table 3 in the columns labeled *mt 1* and *mt 2*. This shows that the ability to handle large fan-in nodes can reduce overall literal count.

4 Conclusions

We have described a machine-learning technique for k-clause approximate image computation, and applied this to approximate ODC computation in logic synthesis. Since this method does not require enumeration of the minterms of the care set, it can be applied when the number of node inputs is relatively large. On the other hand, it is essentially equivalent to the minterm enumeration method when the number of node inputs is less than or equal to k, so no precision is lost for nodes with few inputs. Experimentally, we observe that the method is robust in cases when the BDD-based image computation and minterm enumeration methods fail. We find that increasing k produces diminishing returns in terms of area minimization.

We should note that ODC computation is only one of many applications of Boolean image computation. By iterating the image operation, we can compute an over-approximation of the reachable states of a system (to be more precise, the strongest inductive invariant of the system expressible using $k-$clauses). This invariant could be used to derive sequential don't-care information, or as an aid in formal verification.

References

[1] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *VLSI '91*, Edinburgh, Scotland, August 1991.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[3] O. C., C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[4] F. Corno, M. Sonza Reorda, and G. Squillero. Rt-level itc 99 benchmarks and first atpg results. *IEEE Design and Test of Computers*, pages 44–53, July-August 2000.

[5] J. de Kleer. An improved incremental algorithm for generating prime implicates. In *AAAI '92*, 1992.

[6] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer-Aided Verification (CAV 2002)*, pages 250–264, 2002.

[7] A. Mishchenko and R. K. Brayton. SAT-based complete don't-care computation for network optimization. In *IWLS 2004*, pages 353–360, 2004.

[8] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla minimization. In *ICCAD '86*, pages 352–355, 1986.

[9] Hamid Savoj, Robert K. Brayton, and Hervé J. Touati. Extracting local don't cares for network optimization. In *ICCAD '91*, pages 514–517, 1991.

[10] S. Yang. Logic synthesis and optimization benchmarks, verion 3.0. Technical report, Microelectronics Center of North Carolina.