SAT Sweeping with Local Observability Don't-Cares

Qi Zhu[1] Nathan Kitchen[1] Andreas Kuehlmann[1,2] Alberto Sangiovanni-Vincentelli[1]

[1] University of California at Berkeley, CA, USA
[2] Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

SAT sweeping is a method for simplifying an AND/INVERTER *graph (AIG) by systematically merging graph vertices from the inputs towards the outputs using a combination of structural hashing, simulation, and SAT queries. Due to its robustness and efficiency, SAT sweeping provides a solid algorithm for Boolean reasoning in functional verification and logic synthesis. In previous work, SAT sweeping merges two vertices only if they are functionally equivalent. In this paper we present a significant extension of the SAT-sweeping algorithm that exploits local observability don't-cares (ODCs) to increase the number of vertices merged. We use a novel technique to bound the use of ODCs and thus the computational effort to find them, while still finding a large fraction of them. Our reported results based on a set of industrial benchmark circuits demonstrate that ODC-based SAT sweeping results in significantly more graph simplification with great benefit for Boolean reasoning with a moderate increase in computational effort.*

Categories and Subject Descriptors:
B.6.3 [Logic Design]: Design Aids—*verification*
General Terms: Algorithms, Verification
Keywords: And/inverter graphs, SAT sweeping, observability

1 Introduction

Boolean reasoning is a key part of many tasks in computer-aided circuit design, including logic synthesis, equivalence checking, and property checking. Circuit graphs such as AND/INVERTER graphs (AIGs) [1] are often used to represent Boolean functions because their memory complexity compares favorably with other representations such as binary decision diagrams. In many Boolean reasoning problems, circuit graphs have a high degree of structural redundancy [2]. The redundancy can be reduced by the application of *SAT sweeping* [3]. SAT sweeping is a method for simplifying an AND/INVERTER graph by systematically merging graph vertices from the inputs towards the outputs using a combination of structural hashing, simulation, and SAT queries. Due to its robustness and efficiency, SAT sweeping provides a solid algorithm for Boolean reasoning in functional verification and logic synthesis.

In previous work, SAT sweeping merges two vertices only if they are functionally equivalent. However, functional differences between vertices are not always observable at the outputs of a circuit. This fact can be exploited to increase the number of vertices merged. In this paper we present a significant extension of the SAT-sweeping algorithm that uses observability don't-cares (ODCs) for greater graph simplification. Taking observability into account increases the effectiveness of SAT sweeping but also increases its computational expense. In order to find a balance between effectiveness and efficiency, we introduce the notion of *local observability*, in which only paths of bounded length are considered.

When observability is taken into account, the equivalence-class refinement scheme of the original SAT-sweeping algorithm cannot be used. We introduce a new method of comparing simulation vectors to identify merging candidates. Despite the quadratic theoretical complexity of our algorithm, our results show an approximately linear complexity in practice.

This paper is structured as follows: Previous work is discussed in Section 2. Section 3 introduces some preliminary concepts, including AIGs and SAT sweeping. Section 4 explains the details of SAT sweeping with local ODCs. Section 5 presents the experimental results and Section 6 contains conclusions.

2 Previous Work

The combined application of random simulation and satisfiability queries for finding functionally equivalent circuit components has been proposed in multiple publications [4, 5, 6, 7, 8]. A particular implementation on AIGs, that combines these methods with structural hashing and circuit rewriting, was described in [3] for simplifying the transition relation for bounded model checking (BMC).

In [9] the use of observability don't-cares was proposed to simplify the functionality of internal circuit nodes by exploiting non-observability of their input assignments [9]. Using BDDs for their representation, the don't-cares are systematically computed from the outputs towards the inputs and then applied for node resynthesis. Recently, SAT-based methods have been suggested for the same application [10, 11]. To reduce the huge computational cost of computing ODCs, existing methods use compatible ODCs [12, 13] to avoid frequent recomputations or windowing [14] to limit the problem size. In contrast, the ODC-based SAT sweeping method presented in this paper encodes observability conditions directly in the check for mergeability of two vertices. Random simulation is utilized to effectively reduce the number of merge candidates. Furthermore, our approach is made robust by limiting the path length considered for observability.

Other previous work utilizes ODCs for CNF-based SAT checking. In [15] additional clauses which encode ODC conditions are added to the CNF formula of the circuit with the goal to improve the solver performance. In [16], a similar approach is presented that also introduces additional literals to speed up the SAT search. The aim of these approaches is to speed up the SAT search, whereas our methods is targeted to simplify the circuit representation itself.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

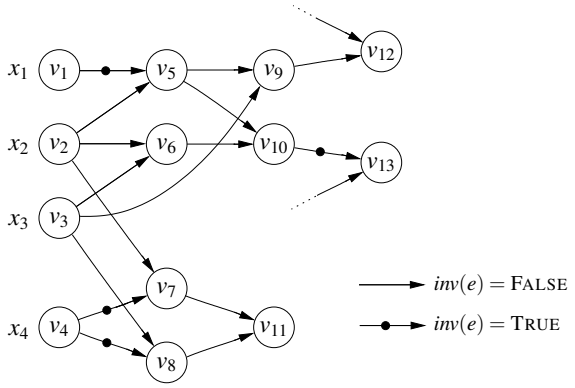

Figure 1: Example of AND/INVERTER graph: $f(v_1) = x_1$, $f(v_2) = x_2$, $f(v_3) = x_3$, $f(v_4) = x_4$, $f(v_9) = (\neg x_1 \wedge x_2) \wedge x_3$; $f(v_{11}) = (x_2 \wedge \neg x_4) \wedge (x_3 \wedge \neg x_4)$; $f((v_{10}, v_{13})) = \neg(\neg x_1 \wedge x_2) \wedge (x_2 \wedge x_3)$.

This simplification can be used in multiple application domains, including equivalence checking [2], property verification [3] or logic synthesis.

3 Preliminaries

For completeness, this section briefly outlines the concept of AND/INVERTER graphs and the SAT-sweeping algorithm.

3.1 AND/INVERTER Graphs

Let $C = (V, E)$ denote an AIG with the set of vertices $V = \{v_0\} \cup X \cup G$ and set of directed edges $E \subseteq V \times V$. Vertex v_0 represents the logical constant 0, X is the set of inputs, G is the set of AND gates, and $O \subseteq G$ is the set of outputs. v_0 and all inputs $x \in X$ have no predecessor, whereas the gates $g \in G$ have exactly two incoming edges denoted by $right(g)$ and $left(g)$. Let $ref(e)$ denote the source vertex of edge e and $FO(v)$ be the set of successor vertices of v, i.e. $v' \in FO(v) \Leftrightarrow (v, v') \in E$. $TFO(v)$ refers to the set of vertices in the transitive fanout of v. Moreover, let $other(v', v)$ denote the incoming edge of vertex $v' \in FO(v)$ which does not come from v, i.e.

$$other(v', v) = \begin{cases} right(v') & \text{if } v = ref(left(v')) \\ left(v') & \text{if } v = ref(right(v')). \end{cases} \quad (1)$$

For the definition of the AIG semantics, we assume that each input $v \in X$ is assigned a Boolean variable x_v and each gate $v \in G$ computes the conjunction of the incoming edge functions. For each edge $e \in E$ an attribute $inv(e)$ is used to indicate whether the function of the source vertex is complemented. More formally, the functions of a vertex $v \in V$ and an edge $e \in E$ of an AIG are defined as:

$$f(v) = \begin{cases} 0 & \text{if } v = v_0 \\ x_v & \text{if } v \in X \\ f(left(v)) \wedge f(right(v)) & \text{otherwise} \end{cases} \quad (2)$$

$$f(e) = f(ref(e)) \oplus inv(e).$$

Figure 1 illustrates part of an AIG. Inverted edges are shown with dots. v_1, v_2, v_3, and v_4 are inputs; their functions are x_1, x_2, x_3, and x_4. The function of v_9 is $(\neg x_1 \wedge x_2) \wedge x_3 = \neg x_1 \wedge x_2 \wedge x_3$. The function of the inverted edge (v_{10}, v_{13}) is $\neg((\neg x_1 \wedge x_2) \wedge (x_2 \wedge x_3)) = \neg(\neg x_1 \wedge x_2 \wedge x_3) = x_1 \vee \neg x_2 \vee \neg x_3$. Note that $f((v_{10}, v_{13}))$ is also the complement of v_9's function, so this AIG is redundant.

3.2 SAT Sweeping

Algorithm 1 outlines the basic flow of the SAT-sweeping algorithm as presented in [3]. The goal of the algorithm is to systematically identify all pairs of functionally equivalent vertices and merge them in the graph. For this the algorithm uses a classical partition-refinement approach using a combination of functional simulation, SAT queries, and structural hashing. For simplicity, Algorithm 1 does not outline the handling of equivalence modulo complementation, i.e., the assignment of two vertices to the same equivalence class if their simulation vectors are identical or complements.

In Algorithm 1 we denote by $F(v)$ the simulation vector of vertex v. The value of $F(v)$ is computed by bitwise application of the semantics given in (2).

During the first iteration of the outer loop the randomly initialized input vectors are simulated to generate a first partitioning of the graph vertices into equivalence classes $\{V_1, V_2, \ldots V_c\}$. To refine the initial partitioning, a SAT solver is applied to check for functional equivalence of the shallowest vertices in all shallow classes containing more than one vertex. If they are indeed functionally equivalent, they are merged. Otherwise, the simulation vectors are updated with the counterexample given by the satisfying input values. This ensures that this class is broken apart during the next iteration. The reader is referred to [3] for more implementation details on the SAT-sweeping algorithm.

Algorithm 1 BASIC SAT SWEEPING

```
1:  {Given: AIG C = (V,E) with inputs X}
2:  randomly initialize simulation vectors F(x) for all x ∈ X
3:  Classes := {V}    {Initially all vertices in single class}
4:  while (1) do
5:    simulate C to update all simulation vectors F(v)
6:    refine all classes V_i s.t. ∀u,v ∈ V_i : F(u) = F(v)
7:    if ∀i : |V_i| = 1 ∨ (UsedResource > ResourceLimit) then
8:      return
9:    else
10:     for all shallow classes i with |V_i| > 1 do
11:       v := argmin_{v'∈V_i} LEVEL(v')
12:       u := argmin_{u'∈V_i\{v}} LEVEL(u')
13:       res := SAT-CHECK (f(u) ⊕ f(v))
14:       if res = SAT then
15:         extend simulation vectors F(x) by SAT counterexample
16:       else if res = UNSAT then {u and v equivalent}
17:         MERGE (u,v)
18:         remove u from V_i   {v is representative of u in V_i}
```

When SAT sweeping is applied to the AIG in Figure 1, v_9 and v_{10} are identified as functionally equivalent and merged, resulting in the graph shown in Figure 2. The outputs of v_{10} are moved to v_9. The rest of the AIG is unchanged; no other vertices can by merged by basic SAT sweeping.

4 SAT Sweeping with Observability Don't-Cares

4.1 Motivating Example

Functional equivalence is a sufficient condition to prevent changes in overall circuit behavior when vertices are merged. However, it is not a necessary condition. Some functional differences between vertices are never observed at the outputs of the circuit. For example, in Figure 1 v_6 and v_8 are not functionally equivalent, since $f(v_6) = x_2 x_3$ and $f(v_8) = x_3 \bar{x}_4$. However, their values only

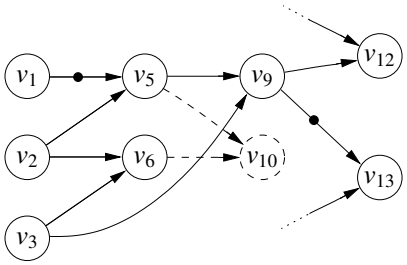

Figure 2: Subgraph of AIG from Figure 1 after SAT sweeping: v_{10} is merged onto v_9.

differ when $x_3 = 1$ and $x_2 = x_4$, which implies $f(v_7) = 0$. Since v_7 provides a controlling value to the input of v_{11}, this difference is not observable for v_8. Therefore, v_8 can safely be merged onto v_6.

4.2 Observability Don't-Cares

For a given AIG, the *observability* $obs(v)$ of a vertex $v \in V$ that has no reconverging path in its transitive fanout, denoted by $TFO(v)$, is defined as follows:

$$obs(v) = \begin{cases} 1 & \text{if } v \in O \\ \bigvee_{v' \in FO(v)} \ (obs(v') \wedge f(other(v',v))) & \text{otherwise} \end{cases} \tag{3}$$

In other words, the value of vertex v is not observable iff for each of its fanouts either the value of the fanout vertex itself is not observable or its other input evaluates to 0, thus blocking the logical path. In the presence of reconvergent paths beginning at v, the recursive computation of $obs(v)$ given above may lead to incorrect results [12]. It may exclude input assignments for which a change of the value at v can propagate to an output through simultaneous switching of multiple paths. To avoid this problem, we use the following modified computation of $obs(v)$ which over-approximates the observability for reconverging structures:

$$obs(v) = \overline{obs}(v,v)$$

$$\overline{obs}(v,u) = \begin{cases} 1 & \text{if } v \in O \\ \bigvee_{v' \in FO(v)} \ (\overline{obs}(v',u) \wedge & \\ \qquad g(other(v',v),u)) & \text{otherwise} \end{cases} \tag{4}$$

$$g(v,u) = \begin{cases} 1 & \text{if } v = u \ \vee \ v \in TFO(u) \\ f(v) & \text{otherwise} \end{cases}$$

The concept of *k-bounded observability* or *local observability* is based on limiting the length of the paths being considered for observability to reduce the effort for its computation. The approximate k-bounded observability, $obs(v,k)$ is defined as:

$$obs(v,k) = \overline{obs}(v,v,k)$$

$$\overline{obs}(v,u,k) = \begin{cases} 1 & \text{if } v \in O \ \vee \ k = 0 \\ \bigvee_{v' \in FO(v)} \ (\overline{obs}(v',u,k-1) \wedge & \\ \qquad g(other(v',v),u)) & \text{otherwise} \end{cases} \tag{5}$$

For example, for $k = 0$ vertex v is always observable, whereas $obs(v,1)$ considers only the possible blockings at the immediate fanouts of v.

Clearly, every input assignment that results in $obs(v,k) = 0$ is an ODC for v. The idea of using local observability in SAT sweeping is to exploit the fact that v can be merged onto u if the functions of u and v are equal for all k-bounded observable input assignments, i.e., u and v have to be equal only if $obs(v,k) = 1$. More formally:

Theorem 1 *For a given k, vertex v can be merged onto vertex u if,*

$$(f(u) \Leftrightarrow f(v)) \vee \neg obs(v,k) = 1$$

Sketch of Proof: It is easy to show that the original circuit is equivalent to the circuit after merging v onto u by arguing that for every input assignment either $f(u) = f(v)$ or $obs(v,k) = 0$, that is, either their functions are identical or no path from v to any vertex within k levels is sensitizable. Therefore, no differences can propagate to any circuit output.

4.3 Algorithm

The criterion for merging two vertices exploiting observability don't-cares as given in Theorem 1 can be evaluated using a combination of random simulation and SAT queries, similarly to the criterion of functional equivalence used in Algorithm 1. The challenge here is that the equivalence-class refinement method cannot be used, because the merging criterion of Theorem 1 is not an equivalence relation: It is neither symmetric nor transitive, so it is not possible to define equivalence classes of vertices such that each member of a class can be merged onto any other member. For example, in the graph in Figure 1, v_8 can be merged onto v_6 as stated in Section 4.1, but v_6 cannot be merged onto v_8—there is an input assignment, $\bar{x}_1 x_2 x_3 x_4$, for which $f(v_6) = 1$ and $f(v_8) = 0$, while $f(v_5) = 1$, a non-controlling value. If v_6 were merged onto v_8, the value of $f(v_{10})$ would be changed for this input assignment.

Instead of selecting candidate pairs for merging from equivalence classes, it is necessary to find independently for each vertex v the vertices onto which it may be merged. Possible candidates include vertices whose simulation vectors are identical to v's, but there may also be candidates whose simulation vectors differ from v's, as long as v is not observable for the input assignments resulting in the differing bits. In order to take observability into account when comparing simulation vectors, we compute *observability vectors* $OBS(v,k)$ from the simulation vectors $F(v)$ as follows:

$$OBS(v,k) = \begin{cases} [111\ldots11] & \text{if } v \in O \ \vee \ k = 0 \\ \bigvee_{v' \in FO(v)} \ (OBS(v',k-1) \wedge & \\ \qquad F(other(v',v))) & \text{otherwise} \end{cases} \tag{6}$$

For example, in Figure 1 if $F(v_2) = 110011$, $F(v_5) = 010010$, and $F(v_4) = 010101$, then $OBS(v_3,1) = (OBS(v_6,0) \wedge F(v_2)) \vee (OBS(v_9,0) \wedge F(v_5)) \vee (OBS(v_8,0) \wedge \neg F(v_4)) = F(v_2) \vee F(v_5) \vee \neg F(v_4) = 111011$.

The bits of $OBS(v,k)$ represent the values of $obs(v,k)$ for the input assignments in the simulation vectors. When the bit at position i of $OBS(v,k)$ (denoted $OBS(v,k)[i]$) is 1, the value of $F(v)[i]$ is observable, and it should be considered when comparing $F(u)$ to $F(v)$. When $OBS(v,k)[i] = 0$, we ignore any difference between $F(u)[i]$ and $F(v)[i]$. When $F(u)[i] = F(v)[i]$ or $OBS(v)[i] = 0$ for all i, we say that $F(u)$ is *compatible* with $F(v)$.

Note that for the bits of $OBS(v,k)$ we use an underapproximation of $obs(v,k)$ which may be incorrect for reconverging paths. However, as shown later in Algorithm 2, we only use the vectors as an initial filter for finding candidates for merging. Any false compatibilites are caught when we check the candidates against the criterion in Theorem 1.

In order to identify vertices with compatible simulation vectors, we build a dictionary of simulation vectors and utilize an efficient search routine that compares subsequences of the vectors using observability vectors as masks.

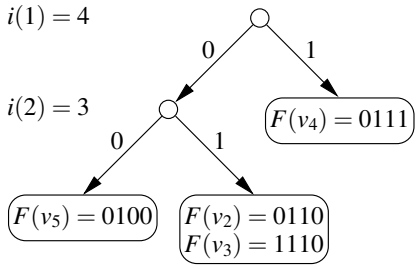

Figure 3: Illustration of trie containing AIG vertices with simulation vectors $F(v_2) = 0110$, $F(v_3) = 1110$, $F(v_4) = 0111$, and $F(v_5) = 0100$.

The overall flow of SAT sweeping with local ODCs is shown in Algorithm 2. After the simulation and observability vectors are initialized, they are used to build the dictionary. For each vertex v, the SEARCH routine returns the simulation vectors in the dictionary compatible with $F(v)$. For each compatible candidate u, the criterion of Theorem 1 is checked using a SAT query. If the SAT solver finds a violation of the criterion, the input assignment that distinguishes u and v is added to the simulation vectors. If the criterion holds, then the vertices are merged. Because u and v are not functionally equivalent, the merge changes the functions of the vertices formerly in the transitive fanout of v. The functional differences are not themselves observable, but the change in graph structure can change the observability of other vertices. Therefore, the simulation vectors must be updated.

Algorithm 2 SAT SWEEPING WITH LOCAL ODCS

```
1: {Given: AIG C = (V,E) with inputs X; bound k}
2: randomly initialize simulation vectors F(x) for all x ∈ X
3: simulate C to compute simulation vectors F(v) for all v ∈ V
4: compute observability vectors OBS(v,k) for all v ∈ V
5: insert each node v in dictionary D
6: for all v ∈ V do
7:    update OBS(v,k)
8:    for all u ∈ SEARCH(D, F(v), OBS(v,k)) do
9:       if (F(u) ⇔ F(v)) ∨ ¬OBS(v,k) = 1 {bitwise} then
10:         res := SAT-CHECK ((f(u) ⊕ f(v)) ∧ obs(v,k))
11:         if res = SAT then
12:            add SAT counterexample to vectors F(x)
13:            simulate C to update simulation vectors F(v)
14:         else if res = UNSAT then
15:            MERGE (v,u)
16:            simulate C to update simulation vectors F(v)
17:            go to next v
```

4.4 Implementation

To implement the search dictionary, we use a binary trie. For each level d of the trie, there is an associated bit index $i(d)$. Each internal node of the trie has two children. The two sub-tries rooted at the children of a node at level d contain AIG vertices whose simulation vectors have value 0 or 1 at bit position $i(d)$, respectively. Each leaf node corresponds to a bin of vertices. All the vertices in a bin have the same values for the sub-vectors indexed by the branching bit indices $i(1), i(2)$, etc. Figure 3 illustrates the trie structure.

To search the trie for vertices with compatible simulation vectors, we use a recursive routine, shown in Algorithm 3. At level d, if $F(v)[i(d)]$ is observed, it is used to choose a branch. If it is not observed, it does not affect the compatibility of other vectors with $F(v)$, so both branches are followed.

Algorithm 3 SEARCH$(T, F(v), OBS(v,k), d)$

```
1: {Given: (sub-)trie T; simulation vector F(v); observability
   vector OBS(v,k), depth d}
2: if T is a leaf then
3:    return V_T {vertices in bin of T}
4: else
5:    if OBS(v,k)[i(d)] = 1 then
6:       b := F(v)[i(d)]
7:       return SEARCH(T_b, F(v), OBS(v,k), d+1))
8:    else
9:       return SEARCH(T_0, F(v), OBS(v,k), d+1))
               ∪ SEARCH(T_1, F(v), OBS(v,k), d+1))
```

The efficiency of searching in the trie depends on the choice of branching bit indices. An uneven distribution of the vertices among the leaves will increase the complexity of the algorithm. In the extreme case, if all the vertices are in a single bin, then every pair of vertices is considered for merging, and the algorithm is quadratic in the graph size. To prevent this from happening, we set an upper limit on the size of the bins. Whenever a bin exceeds the size limit, we insert a new branch node. As a consequence, the number of branches along the path to the leaves increases. To keep the trie as shallow as possible for fast searches, the branching bits could be chosen so as to distribute the vertices evenly among the leaves.

Alternatively, the branching bit indices may be chosen to minimize the number of don't-care bits indexed during searches, since the number of leaves reached in a search is exponential in the number of unobservable bits indexed. In our implementation we chose this scheme, since we did not observe highly unbalanced distributions of vertices in our experiments. To minimize the number of unobservable bits in searches, we select the indices with a straightforward heuristic: For each index i, we compute the number of observed bits in the simulation vectors with index i, i.e., $\sum_v OBS(v,k)[i]$, and use the bits with the largest sums.

Updates to the simulation vectors are needed both when vertices are merged and when a SAT query proves that vertices are not mergeable. In the first case, the only vectors affected are those formerly in the transitive fanout of the merged vertex. (In fact, only k levels of transitive fanout are affected.) We take advantage of this fact by limiting vector propagation to the local fanout in order to minimize the cost of simulation vector updates.

We use another form of incremental simulation vector update when the SAT solver provides a new input assignment. In this case, the values propagated from previous input assignments are still valid, so there is no need to propagate them again. Instead, we update only the bits at the ends of the simulation vectors.

If a vertex v is merged onto a vertex u in its transitive fanout, a cycle will be created in the AIG. Both the original SAT-sweeping algorithm and our extension avoid this case conservatively by comparing the levels of the vertices: v can be merged onto u only if LEVEL$(v) \geq$ LEVEL(u). This constraint can be exploited for efficiency by iterating over the vertices in level order. Only the vertices at the current level and below are searched for merging candidates. The search dictionary need only include the vertices up to that level, so the search cost is reduced. In addition, we rebuild the trie at each level, so the choice of branching bit indices can vary as the algorithm progresses.. We included this refinement in our implementation, but it is omitted from Algorithm 2 for simplicity.

Design	AIG	Number of merges for each k					
	Verts	Basic	1	2	3	4	5
stepper.	215	8	20	24	27	30	31
ss_pcm	506	0	19	19	22	22	23
usb_phy	593	19	38	42	47	47	49
sasc	894	25	210	267	299	299	299
simple_spi	1183	38	266	348	383	387	388
i2c	1315	44	88	104	110	113	114
pci_spoci.	1467	125	275	352	420	439	456
systemcdes	3322	171	354	378	479	493	496
spi	4086	37	187	211	289	292	297
tv80	10026	360	945	1173	1378	1487	1609
systemcaes	13271	206	1609	2211	2464	2471	2483
ac97_ctrl	16553	49	1822	2096	2351	2403	2409
usb_funct	17755	407	1405	1680	1984	2040	2071
aes_core	22307	341	980	1032	1073	1102	1150
wb_conmax	49755	427	1538	3834	5304	5913	6189
Average % merged:		2.73	10.3	12.8	14.8	15.3	15.6

Table 1: Number of AIG vertices merged in OpenCores benchmarks by basic SAT sweeping and SAT sweeping with observability to k levels.

Design	Runtime (s)					
	Basic	1	2	3	4	5
stepper.	0.01	0.02	0.02	0.03	0.04	0.06
ss_pcm	0.01	0.02	0.04	0.04	0.05	0.06
usb_phy	0.02	0.05	0.07	0.09	0.09	0.10
sasc	0.05	0.08	0.11	0.13	0.99	1.00
simple_spi	0.09	0.17	0.24	0.27	1.35	1.46
i2c	0.07	0.29	0.39	0.41	0.88	1.04
pci_spoci.	0.32	0.61	0.76	0.93	1.05	1.42
systemcdes	0.51	1.12	1.48	1.72	2.16	2.24
spi	1.49	5.65	5.72	8.78	10.4	11.0
tv80	7.05	21.5	27.5	39.3	111	152
systemcaes	5.84	12.5	20.9	25.6	39.8	166
ac97_ctrl	5.74	11.3	18.1	22.3	32.9	35.1
usb_funct	14.1	29.8	53.2	71.8	99.6	149
aes_core	7.47	14.7	26.4	33.0	46.3	59.0
wb_conmax	21.9	58.8	115	169	196	237
Avg. relative time	1.00	2.40	3.48	4.41	8.39	11.3

Table 2: Runtime in seconds for basic SAT sweeping and SAT sweeping with observability to k levels on OpenCores benchmarks.

5 Results

We implemented SAT sweeping with local ODCs in OpenAccess [17, 18] with the OpenAccess Gear toolkit [19]. For comparison, we also implemented basic SAT sweeping in the same framework. In this section we present the results of our experiments on input circuits from the IWLS 2005 benchmark set [20]. In particular, we used IWLS benchmarks originating from the OpenCores repository [21], ranging in size from 200 to 48K AND vertices.

Our first set of experiments demonstrates the increased power of ODC-based SAT sweeping for graph simplification, the effect of varying levels of observability, and the scalability of the algorithm. In these experiments we applied ODC-based SAT sweeping to each design, varying the observability bound k from 1 to 5. As a preprocessing step, we applied basic SAT sweeping to remove functionally equivalent vertices, so that all reported merges are due to consideration of observability.

Table 1 shows the number of merges for each benchmark and observability bound k. The first column gives the names of the designs. The second column lists the number of vertices in the original AIGs. The third column lists the number of vertices merged by basic SAT sweeping. The fourth to eighth columns report the number of merges for each value of k. The average percentage of vertices merged is also given. Table 2 reports the runtimes for these experiments. Figures 4 and 5 show plots of the results. In Figure 4, the number of merges is shown as a percentage of the number of vertices in the original AIG. Figure 5 shows the runtimes relative to the runtime of basic SAT sweeping.

The results show that ODC-based SAT sweeping is able to merge many more vertices than basic SAT sweeping—on average, almost 4 times more when $k = 1$ and 5 times more when $k = 2$. Note that the greatest increases in the numbers of merges come with the first 2 levels of observability. For $k > 2$, the increases taper off quickly. The majority of the graph simplification is obtained at a moderate cost in runtime. In most cases, the runtime of ODC-based SAT sweeping with 2 levels of observability is between 2 and 4 times that of basic SAT sweeping. Note also that the runtime increases linearly for low k where the greatest gains are obtained.

Figure 6 shows how the runtime of ODC-based SAT sweeping increases with the size of the circuit graph for $k = 2$. For our benchmarks, the runtime follows a roughly linear trend. This implies that our trie implementation successfully limits the number of candidates for merging, avoiding the theoretical quadratic complexity of the algorithm.

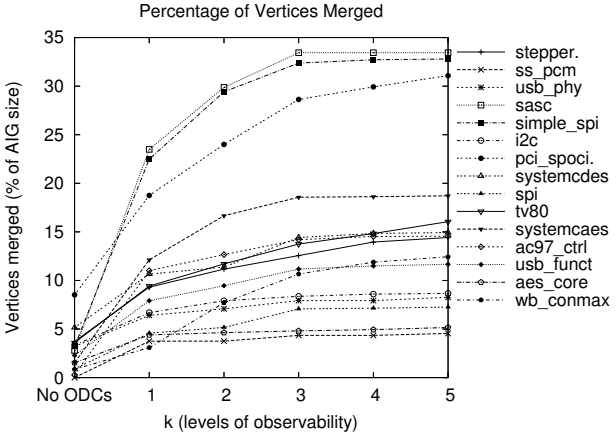

Figure 4: Percentage of AIG vertices merged in OpenCores benchmarks for each observability bound k.

To determine how much optimization potential we sacrifice by our conservative approximation of observability given by (5), we compare our results with a SAT sweeping approach that uses an exact computation of k-bounded observability based on Boolean difference. For $k = 2$, the number of vertices merged increased by less than 0.1% on average and at most 0.3%. For $k = 5$, the number of merges increased by 0.5% on average and at most 3%. These results show that our approximation is very tight in practice.

To illustrate the power of ODC-based SAT sweeping in an application setting, we performed experiments similar to the bounded property checking in [3] on a subset of our benchmarks. In each iteration, we ran SAT sweeping to simplify the latest frame, then unrolled it into a new frame. The results of our experiment are shown in Table 3. On average, ODC-based SAT sweeping achieves almost 20% more simplification than basic SAT sweeping.

6 Conclusions

We proposed an extension of SAT sweeping that considers observability don't-cares. We introduced the notion of local observability, which we exploit to reduce the cost of computing ODCs. Ours is the first work that bounds the use of ODCs by path length instead of windows. Our algorithm employs observability vectors that capture information about ODCs and a trie-based dictionary

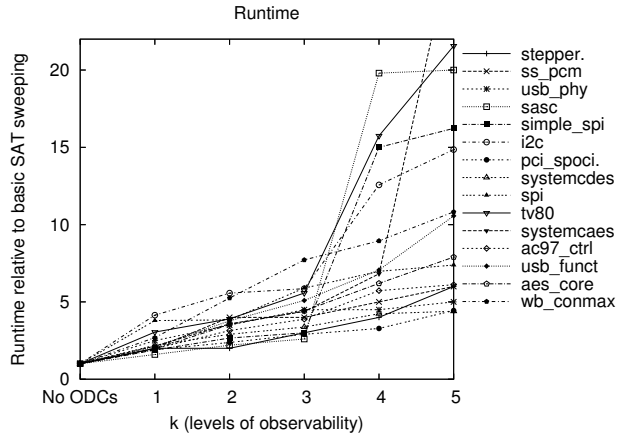

Figure 5: Runtimes of SAT sweeping with ODCs relative to basic SAT sweeping for each observability bound k.

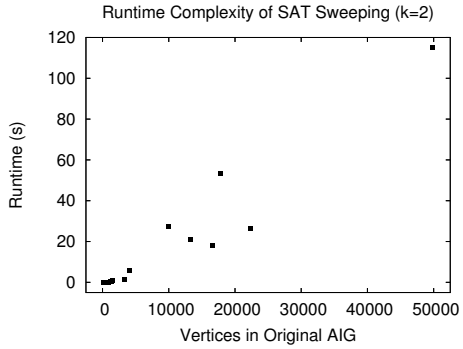

Figure 6: Number of AIG vertices versus runtime of SAT sweeping with 2 levels of observability.

that supports comparison of simulation vectors while taking don't-cares into account. Our experimental results show that SAT sweeping with 2 or 3 levels of observability finds several times more merges than basic SAT sweeping with a moderate increase in runtime, and that it scales well with circuit size. We also showed an increased benefit from ODC-based SAT sweeping in an application setting.

ODC-based SAT sweeping can be expected to have a great effect on other applications, such as equivalence checking. Miter structures in equivalence checking have many functionally equivalent points, which can be used as cutpoints for decomposition into subproblems in order to reduce the total complexity of the procedure. Our algorithm can provide more cutpoints while maintaining the locality of the subproblems because observability is tightly bounded. This is another way in which ODC-based SAT sweeping is an effective tool for increasing the power of Boolean reasoning.

References

[1] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. of the 34th Design Automation Conference*, pp. 263–268, June 1997.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design*, vol. 21, pp. 1377–1394, December 2002.

[3] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Digest of Technical Papers of the IEEE/ACM Int'l Conference on Computer-Aided Design*, pp. 50–57, November 2004.

Design	Algorithm	Number of vertices in each frame				
		1	2	3	4	5
stepper.	Basic	177	156	152	142	139
	ODC	160	114	101	107	105
ss_pcm	Basic	398	383	363	361	361
	ODC	379	349	341	349	349
usb_phy	Basic	459	451	450	450	449
	ODC	436	429	429	429	426
sasc	Basic	734	491	487	476	468
	ODC	492	361	336	336	336
simple_spi	Basic	995	786	775	771	774
	ODC	685	552	541	530	527
i2c	Basic	1122	1084	1073	1076	1076
	ODC	1062	1005	1002	1004	1008
pci_spoci.	Basic	1255	886	704	689	687
	ODC	1028	559	417	422	397
systemcdes	Basic	2827	2787	2786	2786	2786
	ODC	2620	2592	2586	2583	2581
spi	Basic	3771	3748	3744	3744	3744
	ODC	3597	3557	3543	3543	3543
ac97_ctrl	Basic	14219	13730	13226	13010	12888
	ODC	12172	11369	10374	10360	10937

Table 3: Number of AIG vertices in each unrolled frame of OpenCores benchmarks simplified by basic SAT sweeping and SAT sweeping with observability to 2 levels.

[4] D. Brand, "Verification of large synthesized designs," in *Digest of Technical Papers of the IEEE/ACM Int'l Conference on Computer-Aided Design*, pp. 534–537, November 1993.

[5] W. Kunz, "HANNIBAL: An efficient tool for logic verification based on recursive learning," in *Digest of Technical Papers of the IEEE/ACM Int'l Conference on Computer-Aided Design*, pp. 538–543, November 1993.

[6] W. Kunz, D. Stoffel, and P. Menon, "Logic optimization and equivalence checking by implication analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 266–281, March 1997.

[7] E. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT in combinational equivalence checking," in *Proceedings of IEEE/ACM Design Automation and Test in Europe Conference and Exposition*, pp. 114–121, March 2001.

[8] F. Lu, L.-C. Wang, K.-T. Cheng, and R. C.-Y. Huang, "A circuit SAT solver with signal correlation guided learning," in *Design Automation and Test in Europe*, pp. 892–897, March 2003.

[9] A. Saldanha, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Multi-level logic simplification using don't cares and filters," in *Proceedings of the 26th ACM/IEEE Conference on Design Automation*, pp. 277–282, 1989.

[10] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proceedings of Design, Automation and Test in Europe*, pp. 412–417, 2005.

[11] K. L. McMillan, "Don't-care computation using k-clause approximation," in *Int'l Workshop on Logic Synthesis (IWLS'05)*, (Lake Arrowhead, CA), 2005.

[12] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 297–301, June 1990.

[13] R. K. Brayton, "Compatible observability don't cares revisited," in *Proc. of the IEEE/ACM Int'l Conference on Computer-aided Design*, pp. 618–623, 2001.

[14] N. Saluja and S. P. Khatri, "A robust algorithm for approximate compatible observability don't care (CODC) computation," in *Proceedings of the 41st Design Automation Conference*, pp. 422–427, 2004.

[15] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee, "Managing don't cares in Boolean satisfiability," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, p. 260–265, 2004.

[16] Z. Fu, Y. Yu, and S. Malik, "Considering circuit observability don't cares in CNF satisfiability," in *Proceedings of Design, Automation and Test in Europe*, pp. 1108–1113, 2005.

[17] `http://openeda.si2.org`.

[18] T. Blanchard, R. Ferreri, and J. Whitmore, "The OpenAccess coalition: The drive to an open industry standard information model, API, and reference implementation for IC design data," in *Proceedings of the Int'l Symposium on Quality Electronic Design*, pp. 69–74, 2002.

[19] Z. Xiu, D. A. Papa, P. Chong, C. Albrecht, A. Kuehlmann, R. A. Rutenbar, and I. L. Markov, "Early research experience with OpenAccess Gear: An open source development environment for physical design," in *Proceedings of the ACM Int'l Symposium on Physical Design*, pp. 94–100, Apr 3–6 2005.

[20] `http://iwls.org/iwls2005/benchmarks.html`.

[21] `http://www.opencores.org`.