# A Study of Sweeping Algorithms in the Context of Model Checking

Zyad Hassan, Yan Zhang, and Fabio Somenzi
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Boulder, CO 80309

***Abstract*—Combinational simplification techniques have proved their usefulness in both industrial and academic model checkers. Several combinational simplification algorithms have been proposed in the past that vary in efficiency and effectiveness. In this paper, we report our experience with three algorithms that fall in the combinational equivalence checking (sweeping) category. We propose an improvement to one of these algorithms. We have conducted an empirical study to identify the strengths and weaknesses of each of the algorithms and how they can be synergistically combined, as well as to understand how they interact with `ic3` [1].**

## I. Introduction

Combinational simplification eliminates redundancies and increases sharing of logic in a design. It has been successfully employed in logic synthesis, equivalence checking, and model checking.

In the model checking context, combinational simplification often dramatically improves the performance of the proof engines. This has made it into a primary component in both industrial [2] and academic [3], [4] model checkers. Several combinational simplification algorithms have been proposed in the past, such as DAG-aware circuit compression [5], [6] and sweeping methods [7]–[9]. Sweeping methods merge functionally equivalent nodes. They include BDD sweeping [7], SAT sweeping [8], [10], and cut sweeping [9].

When designing a model checker, the strengths and weaknesses of each of the sweeping methods should be taken into account. To the best of our knowledge, no studies have been carried out so far to evaluate and compare the different sweeping methods, with the exception of a limited study reported in [9].

The effect of combinational simplification on several model checking engines has been studied in the past. In [8], SAT sweeping has been shown to positively affect bounded model checking (BMC) [11]. In [12], it is shown that combinational redundancy removal techniques benefit interpolation considerably. The recently introduced `ic3` [1] incrementally discovers invariants that hold relative to stepwise reachability information. Designing a model checking flow that involves `ic3` requires understanding how combinational simplification algorithms affect it.

This paper makes the following contributions:

- We carry out a comparative study of the different sweeping methods.
- We propose a BDD-based cut sweeping method that is more effective than the original cut sweeping.
- We propose a combined sweeping approach in which more than one sweeping method is applied. We show that the combined approach can achieve more simplification than any of the methods can achieve individually.
- We perform an empirical study of the effect of sweeping on `ic3`.

The rest of the paper is organized as follows. Section II contains preliminaries. In Section III, we introduce the BDD-based cut sweeping method. In Section IV, we explain the rationale behind the combined sweeping approach. In Section V we present the experimental results and in Section VI we conclude.

## II. Preliminaries

### A. AND-inverter-graph

The input and output of our sweeping algorithms are AND-inverter-graphs (AIGs). An AIG is a directed acyclic graph that has four types of nodes: source nodes, sink nodes, internal nodes and the constant TRUE node. A *primary input* (PI) is a source node in an AIG. A *primary output* (PO) is a sink node and has exactly one predecessor. The internal nodes represent 2-input AND gates. A node $v$ is a *fanin* of $v_0$ if there is an edge $(v, v_0)$; it is a *fanout* of $v_0$ if there is an edge $(v_0, v)$. $Left(v)$ and $Right(v)$ refer to the left and right predecessors of $v$.

$Fanin(v)$ and $Fanout(v)$ denote the fanins and fanouts of node $v$. An edge in an AIG may have the *INVERTED* attribute to model an inverter. The Boolean function of a PI is a unique Boolean variable. For an edge, it is the function of its source node if the edge does not have the INVERTED attribute, and the complement otherwise. For an internal node, it is the conjunction of the functions of its incoming edges. The Boolean function of a PO is that of its incoming edge.

A *path* from node $a$ to $b$ is a sequence of nodes $\langle v_0, v_1, v_2, \ldots, v_n \rangle$, such that $v_0 = a$, $v_n = b$ and $v_i \in Fanin(v_{i+1}), 0 \leq i < n$. The *height* of a node $v$, $h(v)$, is the length of the longest path from a PI to $v$. The *fanin (fanout) cone* of node $v$ is the set of nodes that have a path to (from) $v$. Two nodes are *functionally equivalent (complementary)* if they represent the same (complementary) Boolean function(s). Functionally equivalent (complementary) nodes can be merged transferring the fanouts of the redundant nodes to their representative. To simplify our presentation, in what follows we deliberately ignore detection of complementary functions.

### B. BDD Sweeping

BDD sweeping identifies two nodes as equivalent if they have the same BDD representation. The original algorithm of Kuehlmann and Krohm [7] works in two stages: equivalence checking and false negative detection. In the first stage, it builds BDDs for each node and merges nodes having the same BDD. The algorithm introduces an auxiliary BDD variable (cut point) for each node that has been merged, which potentially leads to false negatives. In the second stage, it takes the BDDs of the output functions and for each of them, replaces the auxiliary variables with their driving functions. The algorithm is complete in the sense that it can find all the equivalences in the circuit given a sufficiently large limit on the BDD sizes. However, a large limit often hurts efficiency. In this paper, we intend to use BDD sweeping in conjunction with SAT sweeping which is complete and avoids inherent BDD inefficiencies [8]. For that, we employ a version of BDD sweeping that is incomplete yet faster than the original.

The algorithm we adopt iterates the following steps on each node $v$ of the AIG in some topological order. It builds a BDD for $v$ and checks if there exists another node that has the same BDD. If so, it merges these two nodes and continues. Otherwise, if the BDD size exceeds a given threshold, the algorithm introduces an auxiliary BDD variables for $v$ to be used in place of the original BDD when dealing with the fanouts of $v$. An important point is that the original BDD is kept for equivalence checking, but is not used to produce new BDDs. The algorithm is complete if the threshold is so large that no auxiliary variable is introduced. In practice, however, this can be prohibitive.

### C. SAT Sweeping

The advancements in SAT solver technology over the past two decades has proliferated SAT-based reasoning methods. SAT sweeping is one such method proposed by Kuehlmann [8] for combinational redundancy identification. SAT sweeping queries the SAT solver to prove or refute the equivalence of two nodes in the circuit. The basic SAT sweeping algorithm works as follows. First, the circuit is simulated with random inputs, and candidate equivalence classes are formed, where nodes having the same simulation values are placed together. Next, for each two nodes belonging to the same class, the SAT solver is queried for their equivalence. If the SAT solver reports they are equivalent, one of them is merged into the other. Otherwise, the counterexample provided is simulated to break this equivalence class, and possibly rule out other candidate equivalences as well. This process is repeated until a resource limit is reached, or until all classes become singletons, indicating the absence of further equivalences.

In our implementation of SAT sweeping, several heuristics were applied. We mention each of them briefly.

*1) Simulating Distance-1 Vectors:* This heuristic was proposed in [13]. Instead of just simulating the counterexample to equivalence derived by the SAT solver, all distance-1 vectors, that have a single bit flipped, are simulated as well. Only the bits corresponding to the inputs that are in the fanin cone of the two nodes being checked for equivalence are flipped. We have found this heuristic to be very effective in practice. In [13], this process is repeated for vectors that were successful in breaking up equivalence classes until convergence. In our implementation, we only simulate the distance-1 vectors for the original counterexample: for the benchmark suite we experimented with, recurring on successful vectors is too expensive for the number of refinements it achieves.

*2) Clustering:* Simulating distance-1 vectors often results in considerable refinement of the equivalence classes. This is desirable, since an equivalence class is often broken up more cheaply by simulation than by the SAT solver. Moreover, we have observed that with distance-1 vector simulation, it becomes very likely that nodes remaining in an equivalence class are indeed equivalent. Therefore, rather than checking the equiva-

lence of two nodes at a time, we check the equivalence of all nodes in an equivalence class using a single SAT query. If they are all indeed equivalent, we find that using a single SAT query rather than $n-1$ queries where $n$ is the number of nodes in the class.

*3) Height-Based Merging:* When two nodes are proved equivalent, we merge the node with a larger height into the one with a smaller height, instead of merging based on a topological order as in [13]. The intuition being that a node having a larger height often has a larger fanin cone, which suggests that merging it would lead to a larger reduction. Nodes coming later in a topological order do not necessarily have a larger height than nodes coming earlier.

### D. Cut Sweeping

Cut sweeping [9] is a fast yet incomplete approach for combinational equivalence checking. It iteratively computes cuts for each AIG node and compares the functions associated to the cuts.

A *cut* is defined with respect to an AIG node, called *root*. A cut $C(v)$ of root $v$ is a set of nodes, called *leaves*, such that any path from a PI to $v$ intersects $C(v)$. A cut-set $\Phi(v)$ consists of several cuts of $v$. For cut-sets $\Phi(v_1)$ and $\Phi(v_2)$, the *merge operator* $\otimes$ is defined as

$$\Phi(v_1) \otimes \Phi(v_2) = \{C_1 \cup C_2 \mid C_1 \in \Phi(v_1), C_2 \in \Phi(v_2)\} \quad . \tag{1}$$

Assume $k \geq 1$. A *$k$-feasible* cut is a cut that contains at most $k$ leaves. A $k$-feasible cut-set is a cut-set that contains only $k$-feasible cuts. The *$k$-merge* operator, $\otimes_k$, creates only $k$-feasible cuts. *Cut enumeration* recursively computes all $k$-feasible cuts for an AIG. It computes the $k$-feasible cut-set for a node $v$ as follows:

$$\Phi(v) = \begin{cases} \{\{v\}\} & v \in PI \\ \{\{v\}\} \cup \Phi\left(Left(v)\right) \otimes_k \Phi\left(Right(v)\right) & v \in AND \end{cases} , \tag{2}$$

where *PI* and *AND* refer to the set of PIs and 2-input AND gates respectively. Note that cuts are not built for POs because they are never merged.

The function of a cut is the Boolean function of the root in terms of the leaves. It can be represented in different ways, for instance, using bit vectors or BDDs. Two cuts are equivalent if their cut functions are equal. Hence, two nodes are functionally equivalent if their cut-sets contain equivalent cuts.

Cut sweeping is parametrized by $k$ and $N$, the maximum cut size and the maximum cut-set size, respectively. For each node $v$ in some topological order of the AIG, the algorithm builds a $k$-feasible cut-set $\Phi(v)$. Each cut in $\Phi(v)$ is associated with a truth table. Next, it searches for a node equivalent to $v$ by looking for a cut equivalent to some cut in $\Phi(v)$. If it succeeds, the two nodes are merged. Otherwise, a heuristic is applied to prune $\Phi(v)$ to at most $N$ cuts. After pruning, the algorithm stores $\Phi(v)$ as the cut-set of $v$ and builds a cross-reference between each of its cuts and $v$.

The heuristic for pruning, which we call the *quality heuristic*, computes a value $q$ for each cut:

$$q(C) = \sum_{n \in C} \frac{1}{\mid Fanout(n) \mid} \quad . \tag{3}$$

The cuts with the smallest values of $q$ are kept. The intuition of the quality heuristic is two-fold. First, it tries to localize the equivalence and thus favors smaller cuts. Second, it normalizes cuts by attempting to express them with the same set of variables. The chosen variables are those that have a large fanouts, i.e., that are shared by many other nodes.

A good truth-table implementation is critical to the performance of cut sweeping. In [9], truth tables are implemented as bit vectors. An advantage of bit vectors is the constant-time cost of Boolean operations. On the other hand, bit interleaving is required to extend the bit vectors to the same length so that the corresponding bits represent the same minterm[1].

## III. BDD-based Cut Sweeping

Representing functions having a small number of inputs using bit vectors is very efficient. However, the number of bits required grows exponentially with the number of variables, which can easily lead to memory blow-up. As an alternative, BDDs, which are more suitable for large functions, can also be used to represent cut functions. Furthermore, the strong canonicity of BDDs makes it trivial to check for equivalence. The use of BDDs also enables a heuristic which we describe below.

The proposed algorithm differs from the original one in two aspects. First, we introduce a new parameter $s$, the maximum size of a BDD, to replace $k$. That is, instead of $k$-feasible cuts, we keep cuts whose functions contain at most $s$ BDD nodes. Node count, as opposed to number of inputs, is a more natural characterization of BDD size.

The second difference comes from the pruning heuristic. We define the height $h$ of a cut $C$ as the average height of its nodes:

$$h(C) = \sum_{v \in C} \frac{h(v)}{|C|} \quad . \tag{4}$$

[1]A good reference of bit-interleaving can be found at http://graphics.stanford.edu/~seander/bithacks.html.

A smaller $h$ indicates that the leaves in the cut are closer to the PIs. The *height heuristic* keeps at most $N$ cuts choosing the ones with smallest values of $h$.

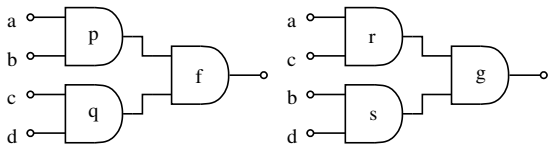


Fig. 1. Two implementations of a 4-input AND gate

A motivating example for the new heuristic is in Figure 1, which shows two different 4-input AND gates. Nodes $a$, $b$, $c$, and $d$ are PIs. Nodes $p$, $q$, $r$, and $s$ are internal nodes. Nodes $f$ and $g$ can only be merged if their cut-sets both contain $\{a, b, c, d\}$. However, if the internal nodes have many more fanouts than the PIs, the quality heuristic may select cuts containing the internal nodes instead, causing the merge to be missed.

As mentioned before, the quality heuristic tries to normalize the cuts on certain "attractors." This reduces the possibility that equivalent functions are represented differently. However, this might also lead to the loss of the opportunity to find equivalences that cannot be expressed by those "attractors," as in Figure 1.

On the other hand, the height heuristic tries to push the cut boundary as far as possible. A supporting argument is that, if a node is employed in equivalent cuts, then replacing it with its predecessors preserve equivalence. Furthermore, new merges that are otherwise undiscoverable (consider other equivalences that require $a$ and $b$ in the above example) may be found. The height heuristic does not attract cuts to certain nodes, which may result in different cuts for equivalent nodes. As shown in the experiments, the effectiveness of the height heuristic reduces as the height of nodes increases.

The two heuristics have their own strengths and weaknesses. A natural question is whether it is possible to combine them to benefit from their individual strengths. We can choose a few cuts with each heuristic. This may lead to more merges but may also worsen the efficiency if it significantly increases the number of cuts. To prevent such an increase, a combined heuristic only records height cuts for the lower nodes, while it keeps both types of cuts for the others.

There is some connection between cut sweeping with each of the two heuristics and BDD sweeping. With the height heuristic, cut sweeping tries to build cuts as large as possible, as BDD sweeping does. However, BDD sweeping can store cuts that exceed the threshold while cut sweeping only keeps those below the threshold. The quality heuristic tries to attract cuts on certain nodes, which is similar to the placement of auxiliary variables in BDD sweeping. Nevertheless, the number of "attractors" in the quality heuristic tends to be much larger than in BDD sweeping.

## IV. Combining Sweeping Methods

The idea of combining several simplification algorithms is not new. Many existing model checkers iterate several simplification algorithms before the problem is passed to the model checking engines. However, we are unaware of any studies that have been carried out to identify the best way they could be combined. In this section we attempt to give general guidelines to which a combined approach should adhere. We support our claims by empirical evidence collected in the experiments reported in Section V.

The problem we address is as follows: given a time budget for combinational simplification, how should it be allotted to the different algorithms? The sweeping algorithms discussed in previous sections vary in their completeness and speed, with cut sweeping being the most incomplete method yet the fastest of the three methods, SAT sweeping being a complete, yet the slowest, and BDD sweeping lying in between, both in terms of completeness and speed.

Possible solutions include allocating the whole time budget to a single algorithm, or dividing it among two or more algorithms. The fact that some methods are better in approaching certain problems than others, suggests that more than one method should be applied. If two or more methods are to be applied in sequence, the intuition suggests that the lower effort methods should precede the higher effort ones. The advantages of doing so are two-fold. First, although the higher-effort methods are likely to discover the merges that a lower-effort method discovers, in general, it will take them more time to do so. Second, preceding higher-effort methods by lower-effort methods is beneficial in having them present a smaller problem that is easier to handle.

Finally, the percentages of total time that should be allotted to each of the methods to yield the maximum possible reduction is studied in Section V.

## V. Results

The experiments are divided into three parts. The first part compares different variations of the cut sweeping

algorithm. The second part shows the results of the combined sweeping methods, and the third part is concerned with the effect of sweeping on `ic3`[2].

We use the benchmark set from HWMCC'10 [14], a set of 818 benchmarks in total. The experiments are run on a machine with a 2.8GHz Quad-Core CPU and 9GB of memory. We use CUDD [15] as our BDD package for all the BDD-related algorithms.

### A. BDD-based Cut Sweeping

Variations of cut sweeping are applied to the HWMCC'10 benchmark set. The differences between variations are two-fold. First, either the number of variables, $k$, or the number of BDD nodes, $s$, is used to drop over-sized cuts. Second, we experiment with several heuristics for pruning cut-sets: the quality heuristic, the height heuristic, and two combined heuristics. The naive combined heuristic ("combined-1") chooses one cut based on the height heuristic and the others based on the quality heuristic. The other heuristic ("combined-2") sets a threshold on the node height (350 in our experiments). For nodes that are below the threshold, it only keeps a height cut. For higher nodes, it produces cut-sets consisting of a height cut and two quality cuts. We denote a method by a $k$ or an $s$ followed by the the heuristic name. All the variations use BDDs to represent the cut functions.

The results are shown in Table I; they are aggregated over the 818 benchmarks. Based on experiments, both the threshold of BDD sweeping and $s$ in BDD-based cut sweeping are set to 250. The total number of AIG nodes before sweeping is 7.22M. "Final" is the size of AIGs after sweeping. "Generated" and "Kept" are the number of cuts generated and kept by the corresponding methods. For an individual benchmark, its "height" is the average height of all merged cuts. The "Height" column is computed by taking the average of the "height" of all the benchmarks. A smaller value indicates that more merges are found by cuts that are close to the PIs. Note that since we use BDDs, the results in terms of efficiency of bit-vectors based methods may not be as good as in [9]. Therefore, when dealing with them, we just compare the effectiveness.

Results indicate that the resulting AIGs are consistently smaller with $s$ than with $k$. There are a few interesting observations. First, the ratios $GeneratedCuts/Merge$ and $KeptCuts/Merge$ are improved significantly with $s$. This means that with $s$, each cut has a larger chance of resulting in a merge.

Second, while "k-quality" and "k-combined-1" have very close sweeping times, the latter achieves $19.8\%$ more merges. Furthermore, the decrease in the "Height" column reveals that the height cuts indeed lead to merges. Although "s-quality" is more effective than the two above methods, it is less efficient due to the larger cut sizes.

For the methods with $s$ (excluding "s-quality"), we observe that "s-height $N = 1$" is the fastest and produces a good number of merges. Increasing the number of height cuts to two triples the run time without gaining many more merges. Comparing it with "s-combined-1", an improvement on the merges is shown by the latter. This indicates that maintaining one height cut and one quality cut works better than two height cuts. For "s-combined-2", the number of merges is between the two above methods, but with lower run time. Furthermore, the numbers of generated cuts and kept cuts are even comparable to "s-height $N = 1$". That is, even though we keep three cuts for those nodes with height larger than 350, on average we compute only a few percent more cuts than we do in the case of one cut per node.

The "Height" values of the three methods confirm the assumption made in Section III: most merges produced by the height heuristic come from cuts close to the PIs. When the two heuristics are combined, a significant increase on the "Height" value is observed. In Figure 2, we show the number of merges found by "s-height $N = 1$" and "s-combined-2" on nodes within different height ranges. The plot is normalized to "k-quality" and has bin size of 50, i.e., a point at $(2, 1)$ indicates that the method finds the same number of merges as "k-quality" for nodes with height from 100 to 149. Obviously, the height heuristic works better on smaller height nodes, while the quality heuristic catches more on larger height ones. The combined heuristic takes the advantage of both and produces an even better profile on nodes with larger heights. Note that although the height heuristic works worse on the nodes with larger heights, it can still get more merges. This may be due to the fact that in this benchmark set, a large percent of equivalences are located at lower heights.

In our setup, cut sweeping is intended for usage as a fast method. Thus we consider "s-height $N = 1$" and "s-combined-2" to be the best variants. Compared to BDD sweeping, those two variants are faster because they create fewer BDD nodes than BDD sweeping, but are less effective since BDD sweeping may keep BDDs

[2]Detailed results for first and second parts can be found at http://eces.colorado.edu/~hassanz/sweeping-study

TABLE I
RESULTS OF CUT SWEEPING, BDD SWEEPING AND SAT SWEEPING ON THE HWMCC'10 SET. BY DEFAULT, $k = 8$ AND $s = 250$.

| Method | Final ($\times 10^6$) | Merges ($\times 10^5$) | Time (s) | Generated ($\times 10^7$) | Kept ($\times 10^7$) | Height |
|---|---|---|---|---|---|---|
| k-quality $N = 5$ | 6.82 | 2.62 | 123.26 | 5.83 | 1.92 | 10.20 |
| k-combined-1 $N = 5$ | 6.75 | 3.14 | 129.64 | 5.84 | 1.90 | 8.57 |
| s-quality $N = 5$ | 6.71 | 3.31 | 536.75 | 7.63 | 2.32 | 12.11 |
| s-height $N = 1$ | 6.55 | 4.07 | 58.99 | 1.07 | 0.54 | 3.19 |
| s-height $N = 2$ | 6.51 | 4.20 | 181.52 | 2.18 | 0.99 | 2.94 |
| s-combined-1 $N = 2$ | 6.48 | 4.42 | 181.21 | 2.29 | 1.02 | 12.86 |
| s-combined-2 | 6.52 | 4.28 | 74.64 | 1.10 | 0.54 | 12.72 |
| BDD Sweeping | 6.34 | 5.61 | 112.74 | – | – | – |
| SAT Sweeping | 6.10 | 6.37 | 2149.4 | – | – | – |

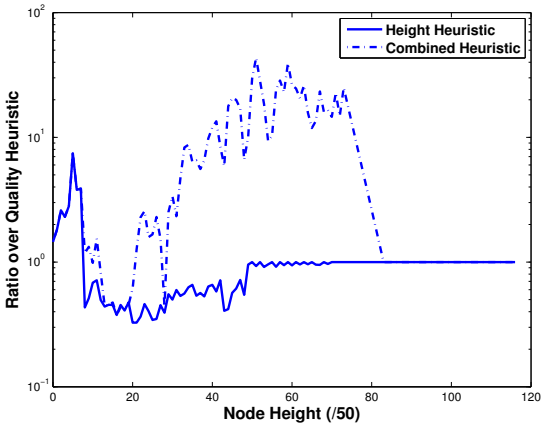


Fig. 2. Number of merges on nodes within different height ranges

that exceed the threshold.

### B. Combined Sweeping Methods

In this section, we show experimental evidence that supports the guidelines of combining sweeping methods presented in Section IV. In particular, we try several possibilities of allotting the budget to the different sweeping algorithms with the purpose of identifying empirically if they should be combined, and if so, in which way. In what follows we use the "s-height $N = 1$" variant of cut-sweeping since it is the fastest, and we simply refer to it as cut sweeping.

In our combined approach, SAT sweeping is always run last since it is the only complete method of the three, and should thus be given whatever time is left to find equivalences not discovered by the other methods. Also, the time not used by any of the preceding methods is passed to SAT sweeping. For instance, if cut sweeping is given a time budget of 4 seconds and only uses 3 of them, SAT sweeping gets to run for one extra second.

We compare the reduction measured in terms of the number of AIG nodes removed, and the total time spent in sweeping. Data are aggregated over the 818 benchmarks. The base case for our comparisons is the pure SAT sweeping case in which SAT sweeping gets the whole budget. The time budget used in our study is 10 seconds.

We consider the following policies: (a) allocating the budget to two methods, (b) allocating it to three methods, and (c) allocating the whole budget to SAT sweeping. For (a) and (b), we consider all the different permutations of assigning integer time values to each method, such that they sum up to 10 seconds. Note that if a sweeping algorithm times out, what it has achieved thus far is used in what follows. In all cases, a set of light-weight sequential simplification algorithms are applied before sweeping. This set of algorithms includes cone-of-influence reduction, stuck-at-constant latch detection, and equivalent latch detection. The total number of AIG nodes for all 818 benchmarks measured after the sequential simplification step is 6.1M.

Results are presented in Table II. The first column lists the methods, where the number before each sweeping method indicates the number of seconds given to it. The second column shows the number of AIG nodes removed. The third column shows the total time spent in sweeping. The methods are listed in order of decreasing reduction. The last row is for pure SAT sweeping. We only show the best three setups in terms of reduction for each of the possible orders of the method sequences.

Several observations can be made. First, when it comes to running two methods in sequence, BDD sweeping combined with SAT sweeping outperforms cut sweeping combined with SAT sweeping. The method that achieves maximum reduction (8 seconds of BDD sweeping followed by 2 seconds of SAT sweeping) removes 56K more nodes than pure SAT sweeping (7.7% more reduction). Second, more reduction is achievable by running three methods in sequence. As suggested in Section IV, ordering the methods by increasing effort

TABLE II
EFFECT OF BUDGET ALLOCATION ON REDUCTION.

| Method | Reduction | Total Sweeping Time (s) |
|---|---|---|
| 4 Cut, 5 BDD, 1 SAT | 801,932 | 518 |
| 2 Cut, 5 BDD, 3 SAT | 801,137 | 516 |
| 6 Cut, 3 BDD, 1 SAT | 801,119 | 522 |
| 4 BDD, 1 Cut, 5 SAT | 794,052 | 517 |
| 8 BDD, 1 Cut, 1 SAT | 793,921 | 515 |
| 7 BDD, 2 Cut, 1 SAT | 793,814 | 519 |
| 8 BDD, 2 SAT | 793,226 | 500 |
| 7 BDD, 3 SAT | 793,068 | 503 |
| 5 BDD, 5 SAT | 792,797 | 508 |
| 1 Cut, 9 SAT | 772,563 | 512 |
| 6 Cut, 4 SAT | 771,070 | 513 |
| 3 Cut, 7 SAT | 769,483 | 511 |
| 10 SAT | 736,594 | 619 |

TABLE III
EFFECT OF SWEEPING ON ic3'S PERFORMANCE.

| Seed Index | Solves (No Sweeping) | Solves (With Sweeping) | Runtime (s) (No Sweeping) | Runtime (s) (With Sweeping) |
|---|---|---|---|---|
| 1 | 693 | 698 | 96,297 | 91,762 |
| 2 | 689 | 699 | 95,629 | 90,341 |
| 3 | 691 | 699 | 95,050 | 92,714 |
| 4 | 696 | 697 | 93,691 | 91,141 |
| 5 | 693 | 698 | 95,007 | 89,656 |
| 6 | 690 | 695 | 96,270 | 91,559 |
| 7 | 693 | 699 | 94,784 | 92,056 |
| 8 | 690 | 701 | 94,351 | 90,837 |
| 9 | 693 | 693 | 95,491 | 92,847 |
| 10 | 690 | 693 | 95,124 | 93,048 |
| Average | 691.8 | 697.2 | 95,169 | 91,596 |

(or equivalently by increasing degree of completeness) achieves more reduction than otherwise. Here, the best method (4 seconds, 5 seconds, and 1 second for cut, BDD and SAT sweeping, respectively), has an edge of 65K nodes over pure SAT sweeping (about 8.9% more reduction). Third, in terms of sweeping time, it is clear that a large drop occurs ($>$ 100 seconds) when two or three methods are combined versus pure SAT sweeping, which is due to the often smaller time needed by BDD and cut sweeping to discover equivalences than SAT sweeping. Given an overall model checking budget, smaller sweeping time allows more time for the model checking engine, which is desirable.

The question of whether such difference has a considerable effect on the performance of the model checking engine is answered in the next section.

### C. *Effect on* ic3

The recently developed model checking algorithm, ic3 [1], has been regarded as the best standalone model checking algorithm developed up till now [16]. As the interaction of combinational simplification methods with different model checking algorithms has been studied in the past, we here aim to study how they interact with ic3. In particular, we would like to empirically find out if ic3 benefits from preprocessing the netlist with a simplification algorithm or not, and if it does, how sensitive it is to the magnitude of reductions achieved through simplification.

In the first experiment, we compare two runs of ic3, one that is preceded by SAT sweeping, and one that is not. The experimental setup is as follows. A total timeout of 10 minutes is used. The budget for SAT sweeping is 10 seconds. The light-weight sequential simplification algorithms referred to in Section V-B are applied once in the no-sweeping case, and twice (before and after sweeping) in the sweeping case. We compare the number of solves, and the aggregate runtime among all benchmarks.

It is important to note that the ic3 algorithm has a random flavor. In particular, the order by which generalization (dropping literals) is attempted is randomized. Also, since the algorithm is SAT-based, randomization occurs in the SAT solver decisions. To have reliable experimental results, each experiment is repeated with 10 different seeds, and the results are averaged over the different seeds.

Results are shown in Table III. The first column shows the seed index, the second and third columns show the number of solves without and with sweeping, and the fourth and fifth columns show the aggregate runtime without and with sweeping.

The results confirm a positive effect of sweeping on the performance of ic3. On average, five more solves are achieved with sweeping, and the aggregate runtime drops by 3.8%.

The enhancement in the performance of ic3 in presence of sweeping can be attributed to two factors. First, reduction in the number of gates caused by sweeping can result in the reduction in the SAT solver time. Second, simplification often results in dropping of latches (e.g., if it merges the next-state functions of two latches). For example, for the benchmark set used in our experiments, sweeping reduces the aggregate number of latches from 279,161 to 269,091 (3.6% decrease). This reduces the amount of work done by ic3 in generalization of counterexamples to induction.

We now repeat the previous experiment, but this time we compare the number of solves and the aggregate runtime between pure SAT sweeping and the empirically optimum combined sweeping scheme of Section V-B. The purpose of this experiment is to understand how

TABLE IV
OPTIMUM SWEEPING VERSUS PURE SAT SWEEPING.

| Seed Index | Solves (Pure SAT Sweeping) | Solves (Optimum Sweeping Scheme) | Runtime (s) (Pure SAT Sweeping) | Runtime (s) (Optimum Sweeping Scheme) |
|---|---|---|---|---|
| 1 | 698 | 696 | 91,762 | 91,567 |
| 2 | 699 | 696 | 90,341 | 91,113 |
| 3 | 699 | 697 | 92,714 | 92,373 |
| 4 | 697 | 702 | 91,141 | 90,478 |
| 5 | 698 | 697 | 89,656 | 90,327 |
| 6 | 695 | 699 | 91,559 | 90,606 |
| 7 | 699 | 697 | 92,056 | 91,498 |
| 8 | 701 | 699 | 90,837 | 92,228 |
| 9 | 693 | 696 | 92,847 | 91,252 |
| 10 | 693 | 697 | 93,048 | 92,663 |
| Average | 697.2 | 697.6 | 91,596 | 91,411 |

sensitive `ic3` is to the magnitude of reductions.

Results are shown in Table IV, where the second and third columns compare the number of solves for pure SAT sweeping and the optimum sweeping scheme, and the fourth and fifth columns compare the total runtime.

As the results indicate, `ic3` does not benefit much from the better reduction achieved by the combined sweeping scheme. The lack of performance enhancement on `ic3` can be attributed to the small improvement in reduction the combined sweeping approach achieves over pure SAT sweeping. In particular, while pure SAT sweeping removes 737K nodes out of the total 6.1M nodes in the 818 benchmarks (12.1% reduction), the combined approach removes 802K nodes (13.2% reduction); a mere 1.1% improvement. This suggests that `ic3` has a small sensitivity to the magnitude of reduction.

## VI. CONCLUSION

In this paper, we presented an empirical study of the different sweeping methods proposed in the past. We have shown that a combined sweeping approach outperforms any of the sweeping methods alone. We have also proposed a BDD-based cut sweeping method that is more effective than the original cut sweeping. Finally, we have studied the effect of sweeping on the new model checking algorithm, `ic3`, and investigated the causes of the better performance it experiences with sweeping. The goal of this analysis is to help designers of model checkers to make decisions regarding the incorporation of sweeping methods and to provide a deeper understanding of how sweeping methods interact with `ic3`.

## ACKNOWLEDGEMENTS

We thank Aaron Bradley who motivated this work and contributed to many discussions. We also thank the reviewers for their insightful comments regarding cut sweeping's pruning heuristics that prompted us to try the "combined-2" heuristic.

## REFERENCES

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *Proc. Int. Conf. on Verification, model checking, and abstract interpretation*, 2011, pp. 70–87.

[2] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design*, 2004, pp. 159–173.

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[4] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," in *Proc. Hardware Verification Workshop*, 2010.

[5] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Proc. Int. Conf. on Computer-aided design*, 2004, pp. 42–49.

[6] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. Design Automation Conference*, 2006, pp. 532–535.

[7] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. Design Automation Conference*, 1997, pp. 263–268.

[8] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proc. Int. Conf. on Computer-aided design*, 2004, pp. 50–57.

[9] N. Eén, "Cut sweeping," Cadence Design Systems, Tech. Rep., 2007.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., UC Berkeley, ERL, Tech. Rep., Mar 2005.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[12] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *Proc. Int. Conf. on Computer-aided design*, 2006, pp. 772–778.

[13] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. Int. Conference on Computer-aided design*, 2006, pp. 836–843.

[14] A. Biere and K. Claessen, "Hardware model checking competition," in *Hardware Verification Workshop*, 2010.

[15] F. Somenzi, *CUDD: CU Decision Diagram Package*, University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[16] R. Brayton, N. Een, and A. Mishchenko, "Continued relevance of bit-level verification research," in *Proc. Usable Verification*, Nov. 2010, pp. 15–16.