Cut Sweeping

Niklas Een
Cadence Research Labs, Berkeley, USA.

Abstract

This paper presents a light-weight sweeping method, similar to SAT- and BDD-sweeping. Performance are on the order of 10x to 100x faster than SAT-sweeping for large designs, while achieving about 50-90% of the reductions.

1 Introduction

In recent years, two methods for circuit minimization through detection of equivalent points, *BDD-sweeping* [4] and *SAT-sweeping* [3, 8], have become prevalent in logic synthesis and verification [6, 7]. Both methods are in principle complete, i.e. they find all existing pairs of equivalent points in the circuit, but are for reasons of scalability often made incomplete by introducing multiple sweeping layers (for BDD-sweeping) or time-bounded SAT-runs (for SAT-sweeping). Even so, these methods do not scale well beyond a million gates, and few applications have the time-budget to sweep such large designs.

For those cases, this paper proposes a new algorithm, *cut-sweeping*, that is inherently incomplete but scales to much larger designs than the previous methods. Experimental results show that for million-gate designs, runtime is on the order of seconds and minutes rather than hours and days, while still picking off 50-90% of all equivalences present in the circuit. This allows for the method to be used in applications such as hardware accelerated emulation or FPGA synthesis.

The cut-sweeping method is based on partially enumerating, and hashing, the *k-feasible cuts* of the circuit [5]. If two nodes try to hash the same cut, they are found equivalent. On a high level, the procedure resembles BDD-sweeping, but with the BDDs kept very small (actually represented as truth-tables), and the sweeping layers being determined individually for each node in a much finer grained fashion than for BDD-sweeping.

Cadence Technical Report CDNL-TR-2007-0510, May 21, 2007

2 Preliminaries

A combinational *boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The *primary inputs* (PIs) of the network are nodes without fanins. The *primary outputs* (POs) are nodes without fanouts. The PIs and POs define the external connections of the network.

A special case of a boolean network is the *and-inverter graph* (AIG), containing four node types: PIs, POs, two-input AND-nodes, and the constant TRUE modeled as a node with one output and no inputs. Inverters are represented as attributes on the edges, dividing them into *unsigned* edges and *signed* (or complemented) edges. An AIG is said to be *reduced and constant-free* if (1) all the fanouts of the constant TRUE, if any, feed into POs; and (2) no AND-node has both of its fanins point to the same node. Furthermore, an AIG is said to be *structurally-hashed* if no two AND-nodes have the same two fanin edges including the sign.

A *cut* C of node n is a set of nodes of the AIG, called *leaves*, such that any path from a PI to n passes through at least one leaf. A *trivial cut* of a node is the cut composed of the node itself. A cut is k*-feasible* if the number of nodes in it does not exceed k.

Cut Enumeration. Here we review the standard procedure for enumerating all k-feasible cuts of an AIG. Let Δ_1 and Δ_2 be two sets of cuts, and the merge operator $\otimes$ be defined as follows:

$$\Delta_1 \otimes \Delta_2 \;=\; \{\, C_1 \cup C_2 \;|\; C_1 \in \Delta_1,\ C_2 \in \Delta_2 \,\}$$

By $\Delta_1 \otimes_k \Delta_2$ we denote the set $\Delta_1 \otimes \Delta_2$ restricted to cuts with $\leq k$ leaves. Further, let n_1, n_2 be the first and second fanin of node n, and let $\Phi(n)$ denote all k-feasible cuts of n, recursively computed as follows:

$$\Phi(n) \;=\; \begin{cases} \Phi(n_1) & ,n \in \text{PO} \\ \{\{n\}\} & ,n \in \text{PI} \\ \{\{n\}\} \;\cup\; \Phi(n_1) \otimes_k \Phi(n_2) & ,n \in \text{AND} \end{cases}$$

This formula gives a simple procedure for computing all k-feasible cuts in a single topological pass from the PIs to the POs. Informally, the cut-set of an AND-node is the trivial cut plus the pair-wise unions of cuts belonging to the fanins, excluding those cuts whose size exceeds k.

3 Cut Sweeping

The basic procedure of cut-sweeping works as follows: A bottom-up k-feasible cut-enumeration, as described in the previous section, is performed on the AIG. The cut-width k

is determined by the user and is fixed throughout the procedure. For each cut, the functional relation between the inputs (leaves) and the output is computed and stored as a 2^k-bit truth-table (*FTB* – "function table") together with the cut. If the cut-sets for two nodes m and n have a common element, and the associated FTBs are the same, m and n are proven equivalent. The information is immediately used to rewrite the network by substituting either m for n or vice versa, after which the cut-enumeration is resumed.

The rationale behind the procedure is that most of the equivalent points detected by existing sweeping methods are local, and if we use wide enough cuts, we should be able to capture a big part of them.

3.1 Cut normalization

In a practical implementation of the procedure just outlined, an actual cut will be stored as a vector, i.e. an ordered sequence. The bit-pattern comprising the FTB will depend on the order, so to facilitate comparison between cuts (including their FTBs) the cut-inputs are kept sorted according to a global order on the gates (any order will do). Moreover, it is desirable to identify functions f and $\neg f$, because they can be used interchangeably in an AIG by adjusting the sign attributes on the arcs accordingly. As cuts are stored during the enumeration, we therefore enforce the lowest bit of the FTB to be zero by simply inverting the whole FTB if necessary, and remembering that the cut now represents $\neg f$ rather than f.

3.2 Cut merging

Combining two cuts C_1 and C_2 with their associated FTBs requires the following steps:

1. The inputs (stored as vectors) are merged and sorted. If the new input vector is larger than the cut-width k, the merge operation is aborted.

2. The FTBs of C_1 and C_2 are manipulated to reflect the new input order and the extra inputs from the other cut.

3. The two FTBs are bitwise-ANDed (there are only AND-nodes in an AIG) to produce the FTB for the new cut.

Additionally, a fourth step can be carried out: scanning the new FTB for *redundant inputs*. Although the cut *structurally* depends on all inputs of $C_1 \cup C_2$, due to reconvergence it often happens that inputs are *semantically* masked (i.e. the input does not affect the output). In the experiments (section 5), this step is performed.

3.3 Prioritizing cuts

Given the procedure for merging two cuts, implementing the $\otimes_k$ operation is straight-forward. To make the procedure feasible for larger cut-widths, however, some cuts have to be discarded. If two nodes are equivalent, it is enough to detect just *one* common cut, which motivates keeping only a few "good" cuts for each node. These "priority cuts" [1] should be chosen to maximize the likelihood of capturing equivalent nodes in the local transitive fanout. We make two observations:

1. Nodes with a single fanout need not be used as input to a cut. If two nodes m and n have a common cut, there must be a common cut where every node has fanout degree ≥ 2.

2. Nodes with a high fanout degree will occur in more cuts, and thus, cuts where *all* inputs have a high fanout degree should have a higher chance of occurring in multiple cut-sets.

From the second bullet we propose the following sorting criteria for cuts:

$$\textit{cutQuality}(C) \;=\; \sum_{n \in C} \frac{1}{\textit{nFanouts}(n)}$$

Here a *low value* means *high quality*. For each node, the N cuts with the highest quality are kept.

3.4 Equivalence detection and merging

Equivalent points are detected by storing cuts (including their FTBs) in a hash-table. The cut will be the hash-key, and the node from which we computed the cut will be the hash-value (i.e. a mapping from cuts to nodes). If, when hashing a cut for node n, we find that the cut is already present in the table with hash-value m, the two nodes have just been proven equivalent. As outlined in the beginning of this section, we choose to rewrite the network as soon as we find two points that are equivalent. Because of this, node m may actually already have been deleted by a previous transformation, in which case the hash-entry is stale and we should just replace it with n instead. If m does still exist, we rewire the network to use m in all places currently using n. Node n is then deleted, as well as any recursively redundant fanins of n.

In principle, it may sometimes be better to do the substitution in the other direction (keeping n and deleting m), but then m must not be in the transitive fanin of n.

4 Implementation

Figure 1 and *2* present the cut-sweep algorithm in pseudo code. The following types are used:

Name	Original Size	SAT-sweeping	Cut-sweeping (k=12 N=10)	Cut-sweeping (k=8 N=5)
design 1	678,396	502,871 — 2154 s	541,767 (78%) — 74 s	572,050 (61%) — 8 s
design 2	1,378,882	1,184,646 — 4021 s	1,209,155 (87%) — 110 s	1,223,025 (80%) — 14 s
design 3	2,174,188	1,871,434 — 7379 s	1,906,063 (89%) — 171 s	1,931,225 (80%) — 23 s
design 4	2,403,322	2,097,401 — 15,003 s	2,240,949 (53%) — 109 s	2,255,588 (48%) — 20 s
design 5	3,160,067	2,744,998 — 22,422 s	2,832,021 (79%) — 311 s	2,886,273 (66%) — 40 s
design 6	17,409,623	[$>$ 24 h]	17,130,262 — 1824 s	17,176,208 — 273 s

Table 1. *SAT- and cut-sweeping comparison.* Sizes are in number of AND-gates (left of "—"), run-times in seconds (right of "—"). Reduction rates for cut-sweeping are compared to SAT-sweeping and printed within parenthesis (if SAT-sweeping removes 500 gates, and cut-sweeping removes 400 gates, then 80% is printed).

Wire – A wire is a pointer to a gate plus a *sign*-bit (corresponding to an arc in the And-Inverter-graph).

Cut – A cut is a set of gates (represented as unsigned wires) plus an FTB. A type-declaration may look like:

```
struct Cut {
    Wire inputs[size]
    Bit  ftb    [2^size]
}
```

Cuts – A vector/array of cuts. Supports '*push*(elem)' operation for adding elements at the end, and '*size*()' for returning the current length of the vector.

Map⟨From,To⟩ – A mapping from type *From* to type *To*. Depending on *From*, it is implemented either by a hash-table or by a vector. Supports '*set*(key,value)' for adding or updating an entry in the map, and '*lookup*(key)' for retrieving the value stored for 'key'.

Furthermore, two functions are used without giving pseudo-code implementation:

- *merge*(***Cut*** c_0, ***bool*** inv_0, ***Cut*** c_1, ***bool*** inv_1, ***int*** k)
— Returns a *Cut* which is the merge of cuts c_0 and c_1 (including their associated FTBs) or the "null" cut if the number of inputs exceeds k. If inv_0 or inv_1 is true, the corresponding FTB is inverted before computing the output FTB.

- *strashedReplace*(***Wire*** old, ***Wire*** new)
— In a structurally hashed, reduced and constant-free network: replace all uses of the gate pointed to by 'old' with the gate pointed to by 'new' and incrementally update the network to a structurally hashed, reduced and constant-free state.

5 Experimental Results

The proposed method has been evaluated on 6 real-world designs from different sources, taken from the regression test-set for the Cadence' PALLADIUM compiler for hardware emulation. The results of cut-sweeping are compared against the free SAT-sweeper inside the tool ABC [2]. The command "fraig" was used in its default mode, which allows the internal SAT-solver to do 100 backtracks while trying to prove a hypothesized equivalence.

The results clearly show a big gap in runtime for these large designs. The reduction rates for cut-sweeping are unsurprisingly a bit smaller, but still within range. Tuning the SAT-sweeper to cope with these large benchmarks may close the run-time gap a bit; but then again, applying an equal amount of tender-loving-care to this first implementation of cut-sweeping may restore the gap.

6 Conclusions and Future Work

For typical designs occurring in the compilation flow of Cadence' PALLADIUM compiler, much of the benefits of SAT-sweeping can be achieved in a fraction of the time by proving points equivalent using cut-enumeration. Future work includes improved implementation and better prioritization of cuts, especially excluding subsumed cuts.

References

[1] S. Cho, S. Chatterjee, A. Mishchenko, and R. Brayton. Efficient FPGA mapping using priority cuts. In *FPGA'07*, 2007.

[2] B. L. S. Group. ABC: A system for sequential synthesis and verification. *http://www.eecs.berkeley.edu/~alanmi/abc/*.

[3] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *International Conference on Computer Aided Design (ICCAD-2004)*. IEEE/ACM, 2004.

[4] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference*, pages 263–268, 1997.

[5] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Cutless FPGA mapping. In *ERL Technical Report, EECS Dept., UC Berkeley*, 2007.

[6] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation.

[7] F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. *FMCAD*, 0:89–96, 2006.

[8] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli. SAT sweeping with local observability don't-cares. In *DAC '06*, New York, NY, USA, 2006. ACM Press.

```
cutEnum(Wire w, Map⟨Wire, Cuts⟩ cutset, Map⟨Cut,Wire⟩ unique, int k, int N)
{
    Cuts cuts                                              – Empty array; elements added by 'push()'
    Cuts $\Delta_0$ = cutset.lookup($w_{in0}$)
    Cuts $\Delta_1$ = cutset.lookup($w_{in1}$)

    – ENUMERATE CUTS:

    cuts.push({w})                                         – Add singleton cut
    forEach ($c_0$, $c_1$) $\in \Delta_0 \times \Delta_1$ do {
        if ($|c_0|$ == 1 && nFanouts($w_{in0}$) $<$ 2) continue   – Use only singleton cuts from
        if ($|c_1|$ == 1 && nFanouts($w_{in1}$) $<$ 2) continue     nodes with multiple fanouts
        Cut c = merge($c_0$, sign($w_{in0}$), $c_1$, sign($w_{in1}$), k)
        if (c.null()) continue                             – Resulting cut is wider than k

        if (c.size() == 0)                                 – Node is constant
            strashedReplace(w, (c.ftb == 0) ? WIRE_FALSE : WIRE_TRUE)
            return
        else if (c.size() == 1)                            – Node is equal to an input
            strashedReplace(w, (c.ftb == FTB_INVERSE) ? $\neg c_0$ : $c_0$)
            return

        cuts.push(c)
    }
    removeDuplicates(cuts)

    – LOOK FOR EQUIVALENT NODE:

    forEach c $\in$ cuts do {
        Wire v = unique.lookup(c)
        if (v $\neq$ WIRE_NULL && notDeleted(v))           – Found the same cut in hash-table
            Wire $w_{in0}$ = (c.ftb & 1) ? $\neg$w : w        – Normalization for negation
            strashedReplace($w_{in0}$, v)
            return
    }

    – STORE CUTS:

    if (cuts.size() $>$ N)  ⟨⟨ keep the N best cuts according to "cutQuality()" ⟩⟩
    cutset.set(w, cuts)
    forEach c $\in$ cuts do {
        Wire v = (c.ftb & 1) ? $\neg$w : w                  – Normalization for negation
        unique.set(c, v)
    }
}
```

Figure 1. *Cut-enumeration for one gate.* Wire 'w' points to an AND-gate with inputs w_{in0} and w_{in1}, for which sets of cuts has already been computed. The above procedure combines these sets to a cut-set for 'w', and at the same time looks for equivalences to use for rewriting.

```
cutEnum(Network network, int k, int N)
{
    Map⟨Wire,Cuts⟩ cutset                                  – default value is the empty set
    Map⟨Cut,Wire⟩ unique                                   – Uniqueness table for cuts

    topologicallyForEach w $\in$ network do                – bottom-up traversal from PIs to POs
        if (type(w) == GATETYPE_AND)
            cutEnum(w, cutset, unique, k, N)
}
```

Figure 2. *Cut-sweeping all nodes.* 'k' is the cut-width used, 'N' the number of cuts kept per node. The network is assumed to be structurally hashed, reduced and constant-free.