Efficient Redundancy Removal In OpenAccess

Donald Chai
donald@eecs.berkeley.edu

1 Introduction

Despite advances in logic synthesis, synthesized designs are often suboptimal because synthesis algorithms cannot take full advantage of don't-cares in circuits. For example, some NAND3 gates may be replaced by NAND2 gates. As a generalization, redundancy removal optimizing a circuit by replacing instance inputs with constants (when possible). The advantage of this optimization is that timing information is known, and replacing inputs with constants cannot degrade a circuit assuming reasonable delay and area models.

The basic redundancy removal algorithm is implemented as described in [1]. First, a circuit is converted into a simple representation for efficient reasoning, and a set of possible stuck-at-faults generated. For efficiency, a fault is not checked if it can be checked by other means, e.g. a stuck-at-0 on any AND inputs is testable iff the output is stuck-at-0 testable. Second, we simulate the circuit using a set of random vectors to eliminate easily tested faults. Finally, we use an automatic test pattern generator based on Boolean satisfiability to confirm whether the remaining faults are untestable.

It is important to note that once a fault site is replaced with a constant, previously untestable faults may become testable, and previously testable faults may become untestable (redundancy removal uses ODCs, and not *Compatible* ODCs). For efficiency, each fault is checked only once for testability. The set of fault sites is sorted, and the sites with highest priority are checked first.

2 Implementation

2.1 Architecture

In order to encourage code reuse and improve modularity, our module exports a clean API, in which each component does a single job (and does it well). The first three components are meant for reuse by other OAGear modules, while the last component is meant for user code and only expects OpenAccess objects. The dependencies are shown graphically in Figure 1.

oagRedun::Flattener This component converts an annotated oaDesign into a oagAi::Graph, while preserving instance boundaries with AIG (and-inverter graph) terminal nodes. Terminal nodes allow us to cleanly reason about specific inputs of specific gates. The flattener is implemented using oagFunc::QueryOcc to accommodate hierarchical designs, and thus requires that every wire has exactly one driver. State bits are converted into pairs of graph inputs and outputs.

oagRedun::RTG This component allows one to check if replacing a given AIG node with a given constant will change the function of the design.

oagRedun::CircuitSAT This component allows one to check if replacing a given AIG node with another (*not necessarily a constant*) will change the function of the design. The interface is defined intentionally so that oagSsw can use this component to check if node merges are feasible under don't-cares.

oagRedun:: The motivation behind the OpenAccess initiative is to provide clean programmatic interfaces, rather than requiring users to run an executable for each operation, reading and writing designs to disk between each command. In accordance with this goal, we provide C++ functions to remove unused logic and redundancies. The actual removal is customizable using callback functions (the callback can simply not remove redundancies at all, e.g. for testability analysis).

2.2 Details

For cleanliness, query routines check whether AIG nodes are replaceable, and not whether say, the left input of a given AIG node is replaceable. Thus, each top-level oaInstTerm corresponds to a unique AIG terminal. However, RTG and

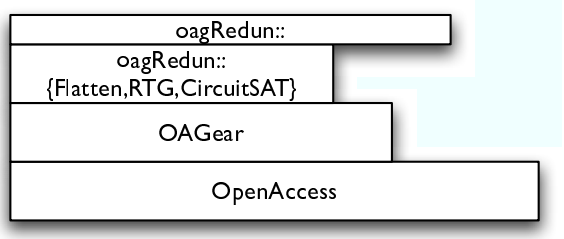

Figure 1: Dependencies and layers of abstraction

Benchmark	# Cells	Area	Flatten (s)	RTG (s)	ATPG (s)		# Redun	ΔArea(%)
					oagAi	oagRedun		
ac97_ctrl	11,855	674,737	0.95	0.52	0.49	0.37	85	0.04
aes_core	20,795	692,567	1.51	2.46	0.14	0.11	2	0.00
des_perf	98,341	4,002,720	7.82	4.86	0.78	0.74	0	0.00
ethernet	46,771	3,345,290	5.36	10.85	230.3	92.36	87	0.01
mem_ctrl	11,440	517,090	1.26	0.76	384.13	138.52	5798	20.66
usb_funct	12,808	625,340	1.38	1.18	14.61	6.27	261	0.32
vga_lcd	124,031	6,310,560	14.59	14.12	50009	2960.3	102	0.01
wb_conmax	29,034	1,049,350	3.51	3.84	43.4	19.56	4006	2.47

Table 1: Redundancy removal results

SAT algorithms generally perform faster with small AIGs. Therefore, when a fault is tested, we remove its corresponding terminal with the assumption that this oaInst boundary may be blurred. Afterwards, we rewrite (without don't-cares) all logic between AIG terminals. Our implementation currently uses oagAi::Graph::rehash.

The random test generation component is implemented using two simulators. First, we simulate the entire AIG using a cycle simulator with random inputs. Second, for each test site, we assume a given stuck-at-value and proceed with an event-driven simulator, checking if any changes are observed at graph outputs. Currently, the event queue is an STL set which stores DFS numbers of nodes to be simulated.

The ATPG component is performed on an AIG using an algorithm similar to that presented in [2]. Rather than using lookup tables to accommodate 9-value logic, we use 3-value logic and assign each node a pair of values. In the worst case, this results in a 2x slowdown. To avoid this, we assign both values simultaneously when we visit nodes *not* in the transitive fanout of the fault site under test. The benefit is that 3-value logic enables very fast inference routines and allows us to integrate conflict-based learning using CNF clauses.

We provide two implementations of ATPG. One natively uses OAGear data structures and is intended for applications with severely limited memory resources. The other uses an efficient data structure with the following features:

1. Data such as reference counts are not embedded in AIG nodes, thereby sparing applications from unnecessary cache pollution. These data are rarely used and should be stored separately by the AIG manager.

2. Since nodes are usually visited in DFS order, nodes are stored in DFS order for cache locality.

3. Fanouts are stored by embedding nodes in linked lists similar to those used in ABC[1]. In ABC, each node has references to its own first fanout, to the next fanout of its left input, and to the next fanout of its right input. We improve on this by marking each reference with one bit to indicate left/right input of the next fanout and another bit to indicate a node is complemented. This streamlines fanout traversal and inference routines.

[1] http://www.eecs.berkeley.edu/~alanmi/abc/

The second implementation synchronizes its internal data with OAGear and uses an identical ATPG algorithm, so that the two implementations differ *only* in performance.

3 Experimental Results

To test the performance of our implementation, we ran our code on large netlists from the IWLS'05 benchmarks. All code was compiled with GCC 3.3 using "-O2 -NDEBUG", and run on a Pentium-4 2GHz processor. All results were verified by an independently written equivalence checker.

The results are shown in Table 1; most columns are self-explanatory. The columns "oagAi" and "oagRedun" show runtimes for the two different ATPGs based on the native OAGear AIG and custom AIG, respectively. The runtimes show that the new data structure is about 2x faster due to better cache behavior. The column "ΔArea (%)" shows reductions from removing unused logic after redundancy removal, further reductions may be obtained with resynthesis.

4 Conclusion

We presented a set of reusable components used to build a combinational redundancy remover. We propose that oagAi be rewritten using a lightweight AIG and a translation layer, making methods such as oagAi::Graph::repackMemory trivial to implement.

References

[1] M. Berkelaar, K. van Eijk. Efficient and effective redundancy removal for million-gate circuits. *International Workshop on Logic Synthesis*, 2001.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, M. K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, 21(12):1377–1394, 2002.