Synthesis-Guided Partial Hierarchy Collapsing

Sayak Ray Baruch Sterin Alan Mishchenko Robert Brayton

Department of EECS

University of California, Berkeley

{sayak, sterin, alanmi, brayton}@eecs.berkeley.edu

ABSTRACT

This paper presents a framework for analyzing distribution of sequentially equivalent nodes in a hierarchical design. This information can be used for selectively collapsing hierarchical modules into 'super-modules' resulting in improved optimization and better placement decisions. Our framework is capable of comparing any two modules in the design hierarchy in terms of logic sharing. Our current implementation is based on ABC and Verific, and we are reporting preliminary result of our experiments.

1. INTRODUCTION

Electronic Design Automation (EDA) industry deals with computer designs that are specified, synthesized, verified, and finally implemented using one of the implementation technologies. Design automation is very important, given the ever increasing sizes and stringent time-to-market requirements of today's designs.

As part of the effort to automate specification and synthesis, design companies and EDA vendors deal with hierarchical design descriptions. A *hierarchy* can be seen as a set of communicating block, called *instances,* belonging to several types, called *modules*. Instances can be nested, in which case an instance that is higher in the hierarchy serves as an environment for any contained instance.

Hierarchy is natural for two reasons: (a) a design can be seen as a set of interacting components, each of which can be specified separately, and (b) the components are specified one at a time by human designers who mostly focus on each part separately.

The presence of a hierarchical description has two consequences:

Preserving the hierarchy is often desirable, because it allows the designer to track important signals during the design cycle. For example, verification, which compares the design before and after optimization, is greatly simplified, if the boundaries of most of the instances are preserved. In this case, correctness of each synthesized component implies correctness of the whole design.

However, the requirement to preserve the hierarchy may impose constraints on logic optimization. The boundaries between modules become separators between the regions optimized independently. The signals crossing the boundaries become primary inputs of the regions. The primary inputs are handled as free variables by the synthesis tools, which leads to the loss of optimization quality.

Several methods have been proposed to gather information from the surrounding modules and use it as constraints for the internal modules, e.g. [1][2]. Constraints allow for more optimal synthesis of the internal modules, because they express conditions when some combinations of primary inputs do not occur or have no impact on the functionality of the design. However, these methods are rarely used in practice because of their complexity and poor scalability. For example, [6] tries to overcome this limitation by imaging don't-cares across the partitions. The method works for smaller designs in the VIS benchmarks and may not be applicable to larger industrial designs.

In most cases, the problem is resolved by partial or complete collapsing of the hierarchy. Although complete collapsing is undesirable because it makes the internal signal untrackable, it is often performed in practice because it gives maximum flexibility for synthesis, provided that the synthesis tool is scalable enough to cope with the large size of the collapsed design.

A more practical method is a partial collapsing of the hierarchy. After partial collapsing, some information is preserved for tracking internal signals by the designer or by a verification tool, while large portions of the design can be handled flat by the synthesis tool, resulting in improved quality of optimization.

The decision on which part of hierarchy to collapse, is often done manually using the designer's knowledge about which parts of the designs are related. However, the designer may not be available when the synthesis tool is exercised or may not have good enough intuition about the degree of similarity of the components. Therefore, in practice, decisions about collapsing the hierarchy are often made ad-hoc or randomly.

In this paper, we propose an automatic way of determining what parts of the hierarchy should be collapsed as a high priority, to give maximum benefit to synthesis. This synthesis-guided collapsing relies on a scalable method of computing sequential equivalences across the signals in the design [3].

Sequential equivalences are objects (nodes/flip-flops) that always produce the same (or opposite) values on the reachable states. Sequential equivalences appear in industrial designs more frequently than combinational equivalences, which require the nodes to produce the same value in any state. However, they are difficult to reason about and are often missed.

The proof of sequential equivalences can be obtained by iterating SAT-based refinement of equivalence classes. This method scales to designs with millions of objects and hundreds of thousands of memory elements, as shown by the latest implementation of [3] in ABC [4]. In the course of our work on detecting and merging sequentially equivalent objects [3], we found that in typical industrial designs such equivalences are more abundant, compared to combinational equivalences. This is easy to understand if we recall that sequential equivalences are those that hold only in the reachable states, while combinational equivalences are a special case of sequential equivalences, which hold in any state.

Merging sequential equivalences typically allows us to reduce the design more than 10% in area, and more than 50% in extreme cases. In this work, we expect the design to have some sequential equivalences. In a typical case, about 5-10% of nodes/registers participate in one of the equivalence classes. This information can be used to detect functional relationships among the blocks in the hierarchy, as proposed in this paper.

Even if the design has no equivalent nodes, or existing equivalences are hard to prove, the proposed methodology allows us to detect 'nearly sequentially equivalent nodes', that is, nodes whose sequential behavior on reachable states is almost the same, or likely the same. Such nodes can also be used to measure the degree of relationship between different parts of the design. This also allows us to trade the runtime of equivalence detection for the quality of hierarchy collapsing.

The rest of this paper is organized as follows. Section 2 explains the objective and methodology of our framework. Section 3 discusses implementation details. Section 4 presents some preliminary results, and Section 5 concludes the paper.

2. OBJECTIVE AND METHODOLOGY

The objective of our framework is to explore the degree of functional similarity across different modules of a hierarchical design and to estimate the trade-off between logic optimization vs. preservation of hierarchy. We will illustrate our objective using a schematic module hierarchy shown in Figure 1. The same hierarchy is again shown as a box diagram in Figure 2.

Module `foo` instantiates three sub-modules, viz. `foo1`, `foo2` and `foo3`. It also contains logic defined using basic logic gates and flip-flops. We call this logic not instantiated as a separate module *glue logic*. This glue logic is labled "`Glue_Logic`" in Figure 2.

Module `foo_1` instantiates two other sub-modules, viz. `foo_1_1` and `foo_1_2`, and module `foo_2` instantiates one sub-module `foo_2_1`. Both `foo_1` and `foo_2` have their own glue logics. To keep this example simple, we assume that `foo_3`, `foo_1_1`, `foo_1_2` and `foo_2_1` contain only glue logic. We have omitted definitions of `foo_1_1`, `foo_1_2` and `foo_2_1` from Figure 1 for brevity.

```
module foo( … )
  … // glue logic
  foo_1 instance_of_foo_1(…);
  foo_2 instance_of_foo_2(…);
  foo_3 instance_of_foo_3(…);
endmodule

module foo_1( … )
  … // glue logic
  foo_1_1 instance_of_foo_1_1(…);
  foo_1_2 instance_of_foo_1_2(…);
endmodule

module foo_2(…)
  … // glue logic
  foo_2_1 instance_of_foo_2_1(…);
endmodule

module foo_3(…)
  … // glue logic
endmodule
```

Figure 1. An example of a typical module hierarchy.

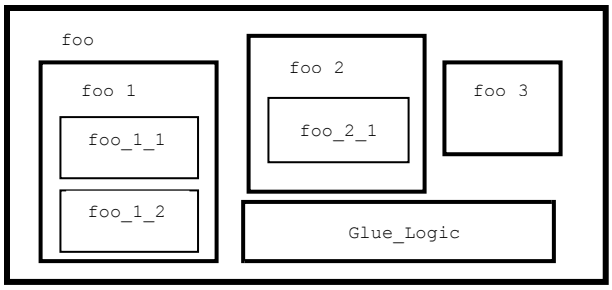

Figure 2. Box diagram of the module hierarchy of module `foo`

Our framework aims at measuring the degree of functional similarity shared by various sub-modules of `foo`. For example, our framework will answer how much functional similarity is shared between `foo_1_1` and `foo_2_1`, or between `foo_1` and `Glue_Logic`. As mentioned before, we have chosen sequential equivalence as the measure of functional similarity. So, in particular, our framework will determine n_1, the number of nodes in a module (say `foo_1_1`) and n_2, the number of nodes in another module (say `foo_2_1`) that are members of the same sequential equivalence class. We will henceforth use the notions of 'functional similarity' and 'sharing of sequential equivalence' interchangeably. Based on these numbers, our framework compares any two modules in the module hierarchy of `foo` and estimates the total amount of sequential equivalence they share with respect to all sequential equivalence classes.

The pair of modules under comparison need not be in the same level of the module hierarchy. For example, we can compare `foo_1_2` with `Glue_Logic`, or with `foo_3`, even though they are not in the same level of the hierarchy. But utility of our framework, as we envision it, is to help

making decisions on merging two or more modules into a single module for the subsequent runs of synthesis and place-&-route. In this regard, we will use our framework to compare only the modules that are instantiated by the top-level module. For example, we will use our framework to compare `foo_1`, `foo_2`, `foo_3` and `Glue_Logic` only.

This is motivated by the question whether we can merge any two modules in the hierarchy. Suppose we detect that module `foo_1_1` and `foo_2_1` share substantial sequential equivalence and decide to merge them together. But as evident from the hierarchy structure, we cannot merge them unless we merge `foo_1` and `foo_2` together. Therefore, it only makes sense to compare two modules which can be merged without disturbing other module boundaries.

Clearly, the candidate modules must be instantiated by the same container module. This also saves us from the combinatorial explosion of comparing any two pairs of modules in the module hierarchy. Said this, we introduce a notion of '*level-1*' module. We consider the top-level module as the level-0 module in the tree of module hierarchy (`foo` in our example) and call all modules instantiated by the level-0 module as level-1 modules (`foo_1`, `foo_2`, `foo_3` and `Glue_Logic` in our example).

We assume that there is always a single level-0 module and more than one level-1 modules. In our experiments, we will compare only the level-1 modules with each other to explore the opportunity of merger.

2.1. Methodology

Our framework follows a "*flatten–analyze–back-annotate"* approach to estimate the sharing of sequential equivalence among modules. We first flatten the hierarchical netlist into a monolithic AIG. A logic node belonging to the glue logic of any module can be considered as a leaf-node in the module hierarchy and such a leaf-level node can be uniquely mapped to a node of the monolithic AIG.

For example, we assumed that the module `foo_1_2` contains glue logic only. Hence, any logic node of `foo_1_2` will be considered as a leaf node in the hierarchy (that can be accessed by traversing the levels `foo`, `foo_1`, and `foo_1_2` in order), and a node from the flattened AIG can be uniquely identified that implements the same function as of this leaf-level node.

A bit-level logic synthesis engine like ABC can analyze the flattened AIG and compute the sequential equivalence classes for its nodes [3]. We can, therefore, annotate this information back to the leaf-nodes of the hierarchical design following the correspondence between the leaf-level nodes of the nodes of monolithic AIG. After back-annotation, our framework moves this information up in the hierarchy from the leaf-nodes to the level-1 modules.

So, in terms of Figure 1, for any sequential equivalence class E of the flattened AIG derived from `foo`, we can figure out how many nodes from each of the four level-1 modules belong to E. Once we have this information per level-1 module, we can immediately compute the (approximate) opportunity of logic reduction by merging two level-1 modules. Steps specific to ABC data structure involved in this "*flatten–analyze– back-annotate"* process are detailed in Section 3.

3. IMPLENTATION DETAILS

The flow of our framework is illustrated in Figure 3. The input is a hierarchical BLIF file, which is flattened into a monolithic netlist N. While flattening, a unique hierarchical name for each node is created. Next, the framework applies the following series of transformation on the flattened netlist N. First, N is converted into a logic network L, then L is *structurally hashed* into another network S, and finally S is converted into an AIG A, for which equivalences are computed. We analyze signal correspondences among the nodes in A and partition it into classes of sequentially equivalent nodes. Now, for each node n in N, we have a unique image $L(n)$ in L. $L(n)$ will have a unique image $S(L(n))$ in S and, subsequently, $S(L(n))$ will have a unique image $A(S(L(n)))$ in AIG A. Therefore, a set of nodes T in netlist N will be sequentially equivalent if the set of their image nodes $A(S(L(T)))$ belong to the same class of sequentially equivalent nodes in AIG A.

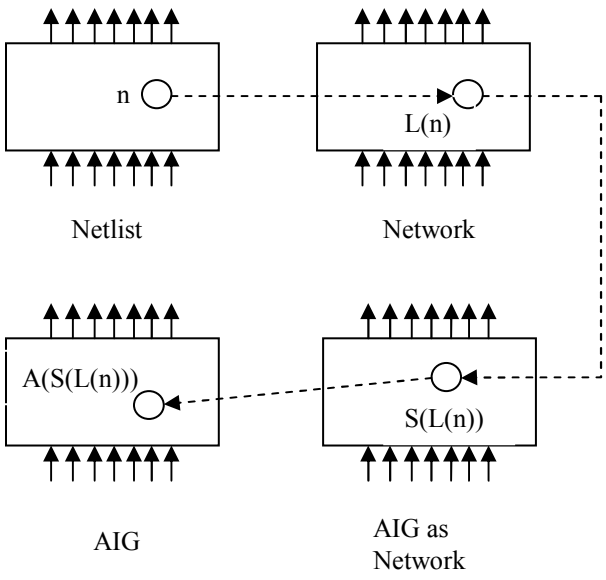

Figure 3. Mapping of nodes across multiple transformations from the hierarchical netlist into an AIG annotated with equivalences.

Now, from the hierarchical name of node n we can determine which level-1 module contains n. By iterating over nodes in N, we can calculate how many nodes from each equivalence class a level-1 module contains.

We gather this information for each level-1 module of the netlist N. Now, suppose there are k equivalence classes $\{E_1, E_2, ..., E_k\}$ for the whole netlist, and suppose M_1 and M_2 are two level-1 modules such that $M_i(E_j)$ is the number

of nodes from module M_i that belong to equivalence class E_j, where $i = 1,2$ and $j = 1, 2,\ldots, k$. If we merge M_1 and M_2, we can replace all $M_1(E_j) + M_2(E_j)$ sequentially equivalent nodes with a single representative node.

On the contrary, if we maintain M_1 and M_2 as two separate modules, we may replace only $M_1(E_j)$ nodes with one representative node inside module M_1 and only $M_2(E_j)$ nodes with one representative nodes inside module M_2. Therefore, merging M_1 and M_2 will allow us to remove $gain(M_1, M_2) = \Sigma_j(M_1(E_j) + M_2(E_j) - 1)$ nodes in total. Our tool analyzes the input circuit and computes $gain(M_i, M_j)$ for each pair of level-1 modules M_i and M_j of the circuit.

Remark: While traversing the sequence of node mappings from n to $A(S(L(n)))$, it should be noted that for some node n, $S(L(n))$ may be eliminated during the clean-up of dangling logic. Since such redundant nodes do not contribute to the metric *gain,* we can ignore such nodes.

We use Verific [5] to generate hierarchical BLIF files from Verilog design descriptions. A hierarchical BLIF network is read in into ABC as a netlist and flattened into another netlist. During flattening, we collapse white boxes and replace black boxes by additional primary inputs and outputs. We then convert it into a logic network, structurally hash it, remove dangling logic, convert it into an AIG and generate sequential equivalence information from that AIG, as described in Section 2.

Our current implementation keeps all the intermediate networks needed to perform the mapping sequence $n \rightarrow L(n) \rightarrow S(L(n)) \rightarrow A(S(L(n)))$, but a little more bookkeeping can be used to eliminate the intermediate networks if that is important for saving memory.

An additional data-structure is maintained to store the number of nodes per equivalence classes. We can pick any arbitrary module in the design hierarchy and store its equivalence class information. This enables easy comparison of any two modules in terms of their share of sequential equivalence classes.

4. EXPERIMENTS

We experimented on IWLS 2005 OpenCore benchmarks. Experiments are in progress. So far we have only preliminary results on a subset of OpenCore circuits.

Three designs were used: *ac97_ctrl*, *tv80*, and *vga_lcd*. We have first analyzed, elaborated, and then generated BLIF files for each design using Verific. The total of 28 level-1 modules were found in ac97_ctrl, 6 level-1 modules in tv80, and 6 level-1 modules in vga_lcd.

Our tool reports the metric *gain* (as defined in Section 2) for each pair of top-level modules of a design. Tables 1, 2 and 3 show the results for ac97_ctrl, tv80 and vga_lcd, respectively. Each level-1 module is given a unique integer id starting with 0. In each of Tables 1, 2 and 3, if we merge the module from the first column with that from the second column, the *gain* is shown in the third column.

Table 4 shows the number of AND nodes and flip-flops for the three designs and the time, in seconds, taken by three main subroutines of our framework. Column *eqv_t* shows the time consumed by the routine computing signal correspondence, column *dist_t* shows the time taken by the routine computing the distribution of various equivalence classes across different top-level modules, and column *gain_t* shows the time taken by the routine computing the metric *gain* for different pairs of top-level modules.

Table 4 shows that the overall time is dominated by the signal correspondence while the hierarchy analysis part (*dist_t* + *gain_t*) takes a negligible portion of time.

5. CONCLUSIONS

We presented a framework for detecting sequential equivalence across various modules in hierarchical designs. Using this information, designers may decide to collapse two modules to leverage better synthesis. The flow was tested on a subset of OpenCore benchmarks with very preliminary results so far. We are actively working on other OpenCore benchmarks and expect to have a more complete experimental evaluation in the future.

6. ACKNOWLEDGMENTS

This work is supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the industrial sponsors Abound Logic, Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

7. REFERENCES

[1] H.-Y. Wang and R. Brayton, "Input don't care sequences in FSM networks". *Proc. ICCAD'93*, pp. 321-328.

[2] V. Singhal, C. Pixley, A. Aziz, and R. Brayton, "Theory of safe replacements for sequential circuits". *IEEE TCAD'01*, Vol. 20(2), pp. 249-265.

[3] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[4] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*, Release 00127. http://www.eecs.berkeley.edu/~alanmi/abc/

[5] Verific Verilog Parser: http://www.verific.com/

[6] K. Gulati, M. Lovell, and S. Khatri. "Efficient don't care computation for hierarchical designs". Proc. ISCAS '06.

Table 1: Result for ac97_ctrl.

Module i	Module j	Gain
2	{13, 27}	3
14	{18, 19, 20, 21, 22, 23, 24, 25, 26}	19
17	{18, 19, 20, 21, 22, 23, 24, 25, 26}	7
18	{19, 20, 21, 22, 23, 24, 25, 26}	3
19	{20, 21, 22, 23, 24, 25, 26}	3
20	{21, 22, 23, 24, 25, 26}	3
21	{22, 23, 24, 25, 26}	3
22	{23, 24, 25, 26}	3
23	{24, 25, 26}	3
24	{25, 26}	3
25	{26}	3

Table 2: Result for tv80.

Module i	Module j	Gain
0	{1}	1220
0	{2}	1214
0	{3, 4,5}	4
1	{2,3,4,5}	1
3	{5}	1

Table 3: Result for vga_lcd.

Module i	Module j	Gain
0	{1}	3
1	{3, 5}	1
1	{4}	27
2	{4}	1
4	{5}	1

Table 4: Summary of results.

Design	#node	#ff	eqv_t	dist_t	gain_t
ac97_ctrl	17940	2345	11.14	0.41	0.67
tv80	14519	359	20.69	0.28	0.02
vga_lcd	158461	17102	360.47	33.92	0.34