Regularity Extraction Via Clan-Based Structural Circuit Decomposition

Soha Hassoun
Tufts University
Medford, MA 02155
soha@eecs.tufts.edu

Carolyn McCreary
Compaq Computer Corp.
Shrewsbury, MA 01545
mccreary@eecs.tufts.edu

Abstract – Identifying repeating structural regularities in circuits allows the minimization of synthesis, optimization, and layout efforts. We introduce in this paper a novel method for identifying a set of repeating circuit structures, referred to as *templates*, and we report on using an efficient binate cover solver to select an appropriate subset of templates with which to cover the circuit. Our approach is comprised of three steps. First, the circuit graph is decomposed in a hierarchical inclusion parse tree using a clan-based decomposition algorithm. This algorithm discovers *clans*, grouping of nodes in the circuit graph that have a natural affinity towards each other. Second, the parse tree nodes are classified into equivalence classes. Such classes represent templates suitable for circuit covering. The final step consists of using a binate cover solver to find an appropriate cover. The cover will consist of instantiated templates and gates that cannot be covered by any templates. We describe the results of applying this algorithm to several circuits, and show that the algorithm is effective in extracting structural regularity.

1 Introduction

With the increasing gap in design productivity, innovative methods are needed to minimize design and verification efforts. Regularity extraction, which identifies repeated structures within a circuit, minimizes optimization, sizing, and layout efforts. Recent research has shown substantial speed ups in logic optimization when using regularity extraction [6]. Regularity extraction can also be used in creating an abstraction of the netlist which, if done well, would allow for increased readability and understanding of the post-synthesis and layout circuit.

In this paper we discuss a novel way of identifying a set of repeating circuit structures, referred to as *templates*, and selecting a template subset with which to cover the circuit. Extracting templates is difficult because it corresponds to generating all equivalence classes of the circuit graph under isomorphism. One recent heuristic is based on growing templates by using hints from bus names and datapath features such as high-fanout control nets [1]. A more structured heuristic is based on identifying tree templates and single-principal output templates, where all outputs of a template are in the transitive fanin of a particular output [2]. Clustering, where a group of nodes are clustered around an initial node(s), is yet another approach to template generation [9, 1, 6]. Other approaches to extracting circuit regularity assume a given set of templates [10, 3].

The novelty of our approach to template generation and circuit covering is due to decomposing the circuit graph into an inclusion hierarchy of subgraphs called *clans*. The hierarchy forms a parse tree of the graph: the leaf nodes are gates in the netlist; intermediate nodes correspond to groupings of nodes; the root of the tree corresponds to the entire graph. Each internal tree node is labeled as linear, independent, or pseudo-linear to reflect the relationship among the node's subclans. This hierarchical decomposition facilitates (a) finding the templates because they can be easily identified by comparing a subset of the parse tree nodes, and then (b) choosing (instantiating) an appropriate subset of templates to generate a circuit cover.

The original clan-based decomposition algorithm was successfully used in graph drawing [7] and in code parallelizing [8]. Our contribution consists of adapting and modifying the original decomposition algorithm to template extraction, classifying the clans into the appropriate templates, and utilizing the parse tree structure to determine a covering.

We begin by introducing the notation and describing the details of clan-based decomposition. Next, we give an overview of our approach, and then we detail each step. We conclude with experiments and future work.

2 Background

2.1 Circuit Graph Definitions

We model a combinational circuit $\mathcal{C}$ as a directed graph, $G = (V, E)$. The vertex (node) set $V = V^C \cup V^I \cup V^O$ is partitioned into combinational logic, primary input, and

0-7803-5832-X /99/$10.00

primary output sets. Each combinational logic component in the circuit is modeled as a vertex $v \in V^C$. Each primary input is represented with a vertex $v \in V^I$. Similarly, a primary output is represented with a vertex $v \in V^O$.

An edge, (x, y), is an ordered pair of distinct vertices, and x is a *predecessor* or *parent* of y, and y is a *successor* or *child* of x. A *path* is a sequence of edges. Node z is a *source* node if z has no predecessors, and node z is a *sink* if it has no successors. If there is a path from u to v, then u is an *ancestor* of v, and v is a *descendent* of u. A *tree* is a connected directed graph where there is exactly one vertex, the root, r, for which there is no edge (x, r) and for every other vertex v there is exactly one edge (x, v).

2.2 Clan-Based Decomposition

A clan is a grouping of circuit nodes with common ancestors and descendents. More specifically, a *clan* is a subset of nodes where every element not in the subset is related in the same way (i.e. ancestor, descendant, or neither) to each member in the subset. A clan C in $G = (V, E)$ is thus a subgraph such that for all nodes x, y in C and all nodes z in $V - C$, z is an ancestor of x if and only if z is an ancestor of y, or z is a descendant of x if and only if z is a descendant of y. Figure 1(a) depicts a graph and its non trivial clans. Trivial clans include all singleton sets and the entire graph.

Clan-based decomposition is the process of decomposing a graph into a hierarchy of clans. A clan with nodes $n_1, n_2, \ldots, n_n$ is classified to be one of three types. A clan is:

- *independent* if it is a union of disconnected nodes, or
- *linear* if for every pair of nodes n_i, n_j in C, n_i is the ancestor of n_j or n_j is the descendent of n_i, or
- *primitive* if it is neither linear nor independent.

A simple independent clan consists of a number of disconnected nodes; a simple linear clan is a sequence of one or more consecutive nodes. The clan classification is hierarchical. Thus, the nodes $n_1, n_2, \ldots, n_n$ correspond to either vertices in the graph, or to already formed clans (subclans).

Deutz, Ehrehfeucht and Rosenberg show that a Hasse graph can yield a unique canonical decomposition into the three types of subgraphs (independent, linear, primitive) [5]. McCreary and Reed report an algorithm for (a) finding clans within a general directed graph and then (b) arranging them into a hierarchical tree structure referred to as a *parse tree* [8]. The parse tree represents an inclusion hierarchy of the clans in the graph. The root tree is the entire graph, and the leaves are individual nodes. The graph in Figure 1(a) can be parsed as shown in Figure 1(b). We summarize the algorithm. More details can be found in the original paper [8].

2.2.1 Recognizing Clans

A clan is found by identifying nodes with the same parents that will become the clan's sources and nodes with the same children that will become the clan's sinks. The clan will then include: the sources, sinks, and the nodes between them. An edge that enters a potential clan at a non-source node and one that leaves at a non-sink node violates the clan definition, thus forming an *illegal* clan. The algorithm for recognizing clans consists of: finding *siblings*, nodes with the same parents, and *mates*, nodes with the same children; identifying nodes that are descendants of the siblings and ancestors of the mates; and then checking that no illegal entries or exits occur. Adding an edge (e, g) to our example graph makes group $a - f$ an illegal clan. Recognizing clans is $O(V^3)$.

2.2.2 Building the Parse Tree

As clans are recognized, there is an attempt to classify them as linear, independent, or primitive. As clans are discovered, the parse tree is updated to reflect clan inclusion relationships. Clan sizes and overlapping clans are compared to place each clan in its proper location in the parse tree. If two clans overlap, but neither is a subset of the other, the clans must be linearly connected. In our example, clans $a - e$ and $d - f$ overlap, but neither one is a subset of the other. Thus, they are linearly connected. If a graph does not contain any primitives, then the classification of nodes along a path from the root to the leaf nodes will alternate between linear and independent.

2.2.3 Decomposing Primitive Clans

Arbitrarily large clans, and sometimes the whole graph, can be classified as primitives, thus preventing the use of the above algorithm for meaningful detailed decomposition.

McCreary suggests augmenting edges to decompose primitive clans [8, 7]. More specifically, edges from all the source nodes of a primitive to the union of the children of the sources are added. This creates a linear connection from the source nodes to the rest of the clan. The primitive clan is then decomposed into a linear clan of 2 subclans: an independent clan representing the source nodes, and a subclan containing the rest of the nodes in the original primitive clan. This latter subclan can then be re-parsed to determine if it is linear, independent, or a primitive. If primitive, it can be resolved using the same technique.

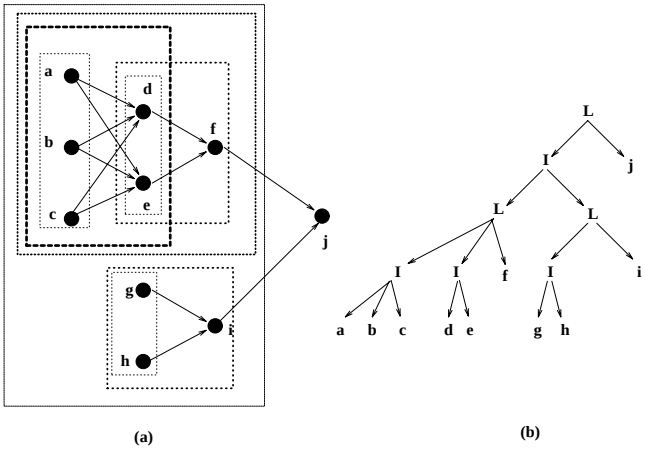

Figure 1: Example. (a) A graph and its non trivial clans. (b) The corresponding parse tree. I refers to an independent clan; L refers to a linear one.

Modifying the graph in Figure 1(a) by removing edge (a, e), causes nodes $a-f$ to be classified as a primitive. Thus, the clan representing nodes $a-f$ will be classified as a linear consisting of an independent clan of sources $a-c$, followed by the rest of the nodes $d-f$. The latter subclan is parsed as a linear composed of an independent, nodes d, e, followed by a linear, node f. The final parse tree, however, is identical to the one shown in Figure 1(b). This is because, when constructing the parse tree, subclan d, e and subclan f are inserted as children of the linear $a-f$ to ensure alternating between independent and linear classifications. When decomposing primitive clans, the algorithm becomes $O(V^4)$.

3 Approach

To realize a circuit covering using repeating circuit structures, two problems must be solved: template generation and graph covering.

The problem of template generation consists of finding for a given circuit graph a set of templates such that each template has at least two instances, or equivalently, finding a set of subgraphs that repeat more than once.

Our approach to template generation is based on using the clan-based decomposition algorithm to generate a parse tree. Only the nodes in the parse tree are potential templates. The parse tree is then examined to find clans that are structurally identical. However, because decomposing primitives sometimes results in identical parse trees even though the underlying structures are not isomorphic, we modify the parsing algorithm to allow us to identify such differences. We also modify the primitive decomposition algorithm to increase the algorithm's ability to build isomorphic clans.

A circuit cover consists of instantiated templates, each occuring more than once, and of gates that cannot be covered by such instantiations. The problem of finding a cover when given a template set, S, is essentially a binate covering problem, because to cover a node in the circuit using one template, precludes covering it using another. The set of such covering constraints can be derived from the parse tree structure. The appropriate cover can then be determined using an efficient solver such as Scherzo [4].

Our approach is comprised of three steps. First, the circuit is decomposed using a modified clan-based decomposition algorithm into a parse tree. Second, the template set is extracted from the parse tree. Third, a covering of the circuit using a subset of the templates is found. We discuss each of these steps.

3.1 Modified Graph Decomposition

As was explained in Section 2.2.3, a true linear clan and a primitive clan, or two primitive clans may be decomposed into identical parse subtrees. To ensure that template extraction distinguishes between such structures, we revise the primitive decomposition algorithm. We decompose primitives into pseudo-linear clans instead of linear clans. A pseudo-linear clan consists of an independent clan representing some of the primitive's source nodes, and a subclan containing the rest. When building the parse tree, pseudo-linear clans are not combined with other linear or pseudo-linear clans. The modified algorithm thus decomposes the graph in Figure 1 *without* edge (a, e) as shown in Figure 2, where PL refers to a pseudo-linear clan.

We further modify McCreary's primitive decomposition algorithm to enhance the ability of decomposing the circuit into isomorphic structures. The original algorithm removes the primitives sources all at once, thus ruining the chances of discovering large subclans that could be identified had there been a more gradual isolation of the sources nodes from the rest of the primitives. We thus rely on the maximum height from a primary output to decide which nodes (subclans) to isolate as the independent clan of the pseudo-linear clan. Our modifications slows the run time of the algorithm; however, it is still bounded by $O(V^4)$.

3.2 Template Extraction

To find templates, we must compare nodes in the parse tree. If two or more match, then they are instances of a template. Subtrees in the parse tree with identical descendents correspond to subgraphs of identical nodes; however, they may not be isomorphic because of the way edges are added to the original graph during primitive decomposition.

To determine if two nodes (clans) c and g in the parse

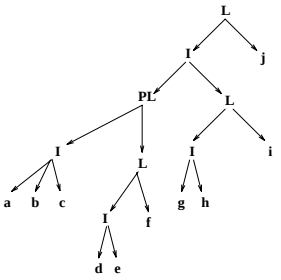

Figure 2: Parse tree of graph 1(a) without edge (a, e).

tree are *identical* (i.e. correspond to isomorphic subgraphs), then we verify the following:

- If c and g are leaf nodes, then they are identical if they have the same logic gate type and the same number of incident edges.
- If both c and g are independent clans, then for each child of c, there exists, an identical child of g.
- If both c and g are linear clans, then for each child c, there exists an identical child of g. Moreover, the sequence of the children of c is exactly the same sequence of the children of g.
- If both c and g are pseudo-linear clans, then we verify that they are equivalent linear clans and that the connectivity of g between the nodes of the independent subclan and the nodes in the rest of the pseudo-linear clan is identical to that of c.

A few observations improve the efficiency of finding identical clans. First, only nodes with identical levels (distance from leaf nodes) need to be compared. Second, by assigning ID numbers to the equivalence classes of nodes, we can hierarchically verify if two non-leaf linear or independent nodes are identical by comparing the of the equivalence classes of their immediate children. Sorting the IDs of subclans of independent clans make quick comparisons possible. Finally, comparing the connectivity of two pseudo-linear clans is made efficient by sorting the necessary information.

3.3 Circuit Covering

Once a template set S is identified, a subset of S must be chosen to cover the circuit such that each circuit gate is covered by exactly one template. This cover must minimize the design effort (cost). We first describe our objective function, and then describe how to generate covering constraints for the binate covering solver.

The design effort per template increases with the size of the template. The total design effort decreases with the frequency of usage of each template because the per template design effort becomes amortized over all instances of the template. We chose to assign the following simplified cost function to each clan in the parse tree. More complex cost functions, however, are possible. The cost assigned to a clan c is:

$$cost(c) = \begin{cases} 0.01 & \text{if } size(c) \text{ equals } ideal \ \& \ freq(c) > 1 \\ size(c) & \text{if } freq(c) = 1 \\ log|ideal - size(c)| \,/\, ln\,(freq(c)) & \text{otherwise} \end{cases}$$

Leaf nodes are assigned a frequency of 1, and *ideal* refers to an ideal template size that is provided by the user. The cost is minimal (0.01) if the clan is of an ideal size. It is proportional to the size of the clan for clans with a frequency of 1. The cost, otherwise, is proportional to the distance in the parse from an ideal clan; we thus use the log function of the absolute value of the difference between the idea and the clan size. This last cost is further reduced in case the clan structure appears frequently in the circuit.

To obtain a cover, we generate *covering constraints* from the parse tree. Two types of covering constraints are needed. The first type ensures that all leaf nodes (gates) are covered. The second ensures that each leaf node is either chosen once or covered by at most one ancestor. The constraints are:

- A leaf is covered if it or one of its tree ancestors is chosen.
- If a node in the tree is chosen, then none of its ancestors or descendents can be chosen.

We thus associate a variable with each node in the parse tree, and generate the constraints by traversing the tree appropriately. The constraints are then passed to Scherzo [4], which assigns a value to each variable, reflecting if a node should be included in the cover.

4 Experimental Results

We modified the original C++ code for clan-based decomposition, and implemented the template classification and constraint generation algorithms. Scherzo is used to solve the binary covering constraints. The input circuits are either from the ISCAS-85 benchmarks, which are mostly datapaths with some controllers, or generic examples.

Table 1 summarizes our results. We report the size (S) and frequency of occurrence (F) of the largest (column (c)) and most frequent (column (d)) templates in the parse tree before covering. We then report the covering results first assuming that the ideal size is equal to the size of the largest template in the parse tree (columns e-h), and then assuming that it is equal to the size of the most frequent template in the parse tree

		Parse Tree T		ideal size is size of largest template				ideal size is size of most frequent template			
ckt	size	largest S/F	most frequent S/F	largest S/F	most frequent S/F	regularity index	covering index	largest S/F	most frequent S/F	regularity index	covering index
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
alu4	116	9/2	3/7	9/2	4/4	2.50	69.83	9/2	4/4	2.57	69.83
mult	172	12/2	3/10	12/2	4/7	2.86	71.51	8/2	4/9	2.78	73.84
4cla	217	8/4	2/26	8/4	2/16	4.60	78.80	8/4	2/16	4.90	78.80
c432	189	3/9	2/9	3/9	3/9	6.33	49.21	3/9	3/9	6.33	49.21
c499	211	45/2	3/16	45/2	3/16	8.67	80.57	5/8	3/16	9.60	79.62
c880	417	33/2	3/21	33/2	3/13	4.78	50.36	14/4	3/13	3.69	55.64
c1355	555	109/2	2/104	109/2	12/16	8.67	92.613	109/2	12/16	8.67	92.61
4-b ripple	57	8/3	4/4	8/3	4/4	3.50	70.18	8/3	4/4	3.50	70.18
8-b ripple	113	8/7	4/8	8/7	4/8	7.50	77.88	8/7	4/8	7.50	77.88
16-b ripple	225	8/15	4/16	8/15	4/16	15.50	81.78	8/15	4/16	15.5	81.78

Table 1: Summary of Results. S refers to size of template and F refers to the related frequency.

(columns i-l). We summarize the quality of our covers using two indices. The *regularity index* is the number of templates instantiated in the cover divided by the number of unique used templates. A higher regularity index indicates that fewer unique templates were frequently used. The *covering index* reports the percentage of the circuit that was covered using instantiated templates.

From columns (c,d), we see that we are able to generate small templates that are frequent, or larger templates that are less frequent – an expected and intuitive result. We also see that we can cover large portions of the circuit using templates (columns h,l). Furthermore, for the ripple adder, we see that the quality of the cover improves with larger circuits (columns g, h and k, l). That is, the algorithm is able to identify existing circuit regularity as the circuit size and regularity is increased. The runtime for the algorithms was negligible on an loaded Ultrasparc for circuits with sizes less than 600 gates.

5 Conclusion

Our approach to regularity extraction and circuit covering is unique because it is based on a general decomposition algorithm. The decomposition is beneficial in: (a) effectively generating an appropriate and small, $O(V)$, number of templates, and (b) easily producing the needed binate covering constraints. The generated templates are appropriate because they correspond to groupings of nodes that designers would manually identify as appropriate. The results of applying the algorithm to benchmarks indicate that our algorithm is effective in extracting regularity. We are currently working on investigating other more efficient decomposition algorithms to enable the handling of large as well as cyclic graphs.

This is the first application of graph-grammar based linear/independent decomposition to VLSI CAD. We believe the decomposition will be useful in other applications such as automatically drawing schematics of synthesized transistors.

6 Acknowledgment

The authors wish to thank Olivier Coudert for allowing the use of his efficient code for solving binate covering.

References

[1] S. Arikati and R. Varadarajan. "A Signature Based Approach to Regularity Extraction". In *IEEE International Conference on Computer-Aided Design*, pages 542–5, 1997.

[2] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta. "A General Approach for Regularity Extraction in Datapath Circuits". In *IEEE International Conference on Computer-Aided Design*, pages 332–339, 1998.

[3] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey. "Perofrmance Optimization Using Template Mapping for Datapath-Intensive High-level Synthesis". *IEEE Transactions on Computer-Aided Design*, 15(8):877–87, August 1996.

[4] O. Coudert and J. Madre. "New Ideas for Solving Covering Problems". In *ACM-IEEE Design Automation Conference*, pages 641–6, 1995.

[5] A. Deutz, A. Ehrenfeucht, and G. Rozenberg. "Clans and Regions in 2-Structures". *Tehoretical Computer Science*, (129):207–262, March 1994.

[6] T. Kutzschenbauch. "Logic Optimization Using Regularity Extraction". In *"Proc. of the 1999 International Workshop on Logic Synthesis"*, 1999.

[7] C. McCreary, R. Chapman, and F.-S. Shieh. "Using Graph Parsing for Automatic Graph Drawing". *IEEE Transactions on Systems, Man, and Cypernetics – Part A: Systems and Humans*, 28(5):545–61, September 1998.

[8] C. McCreary and A. Reed. "A Graph Parsing Algorithm and Implementation". Technical Report TR-93-04, Auburn University, Auburn, AL, 1993.

[9] R. Nijssen and J. Jess. "Two-Dimensional Datapath Regularity Extraction". In *"Proc. of the 1996 ACM/SIGDA Physical Design Workshop"*, 1996.

[10] D. Rao and F. Kurdahi. "On Clustering for Maximal Regularity Extraction". *IEEE Transactions on Computer-Aided Design*, 12(8):1198 –1208, August 1993.