SEMI–SYNTHETIC CIRCUIT GENERATION USING GRAPH MONOMORPHISM FOR TESTING INCREMENTAL PLACEMENT AND INCREMENTAL ROUTING TOOLS

David Grant, Scott Chin, and Guy Lemieux

University of British Columbia
Vancouver, BC, Canada
email: [davidg,scottc,lemieux]@ece.ubc.ca

ABSTRACT

FPGA architects are always searching for more benchmark circuits to stress CAD tools and device architectures. In this paper we present a new method to generate benchmark circuits by removing part of a real circuit and replacing it with a synthetic clone. This replacement or stitching process can easily introduce combinational loops if the synthetic circuit contains an input-to-output dependence that was not in the original subcircuit it is replacing. We show that this can be expressed as the graph monomorphism problem, and that a solution to that problem gives a precise stitching assignment that is cycle-free. This technique can be used to create new benchmark circuits that are identical to the original circuit except for small, local changes. The resulting semi-synthetic benchmarks are ideal for testing incremental place and route tools.

1. INTRODUCTION

Incremental design changes arise for a number of reasons including debug changes, iterative design improvement, and physical resynthesis to meet timing closure. To test the incremental modes offered by CAD tools, *incremental circuits* are needed. These circuits should behave like real circuits, except that a large number of variations are needed to mimic the process of small, incremental design changes. To our knowledge, no incremental benchmark circuits exist to test FPGA tools. Good FPGA benchmark circuits are difficult enough to obtain by themselves – gathering incremental changes that represent the evolution of the circuit is even more challenging.

A number of methods exist to create *synthetic circuits* including stochastic generation using just a few parameters (e.g., `gnl` [1]), stochastic generation of clones based upon detailed characterization of real circuits (e.g., `ccirc+cgen` [2, 3]), and stochastic stitching together of real designs as subcircuits of a larger design [4, 5]. Ultimately, these generators are concerned with creating an entire benchmark circuit, making them unsuitable for creating incremental circuits where large parts of the circuit must remain the same.

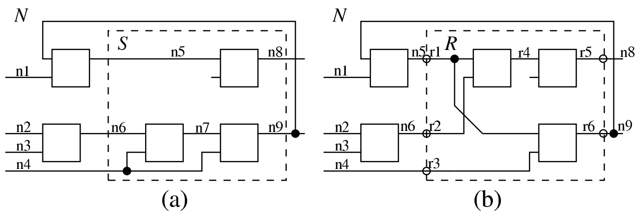

Fig. 1. A synthetic sub–circuit (clone) replacement that introduces a combinational loop.

This paper describes an approach for creating synthetic incremental circuits. The approach takes an original circuit (either real or synthetic) N, identifies a sub-circuit S, removes it to produce $N \backslash S$, and replaces S with a replacement R. Of these steps, identifying S and generating replacement R can be done easily. However, the process of stitching R into $N \backslash S$ is not robust and requires care to avoid the creation of combinational loops. We are unaware of any previous work done where part of a circuit (S) is removed from a larger circuit (N) and replaced with a synthetic clone (R).

Figure 1 illustrates the problem of combinational loops. S and R are identified in the dashed areas of Figures 1a and 1b respectively. We refer to R as a "clone" of S regardless of whether it exactly mimics all the characteristics of S. In this case, R contains the same number of inputs, outputs, and LUTs as S, however, the mapping of the input $r1$ to $n5$ and output $r6$ to $n9$ has created a combinational loop.

To create R, we build upon existing generators, particularly `ccirc+cgen` and `gnl`. These tools use techniques to avoid the creation of combinational loops within R, but they have no knowledge of the circuit in which R will be stitched. Ideally, generators would accept input/output dependence constraints that specify which inputs should **not** be connected to which outputs. If R was created under such constraints, the process of "stitching" it back into the original circuit would be trivial. However, existing generators do not accept such constraints.

Instead of creating a methodology which depends upon features of one specific tool, we show how a replacement circuit R can be stitched into an original circuit N to generate incremental benchmarks using any synthetic generation tool. We present two related problems in this paper: (a) generating a graph P of permissible input-to-output dependences from $N \backslash S$, and (b) assigning the inputs and outputs of R to specific cut points of $N \backslash S$ in a way that prevents combinational loops. We solve both of these problems using heuristic techniques.

2. PROBLEM FORMULATION

Given a directed acyclic graph representing netlist N, subcircuit S, and replacement R, the **circuit stitching problem** is to remove S from N (creating $N \backslash S$), and replace it with R in a way which prevents loops from being formed. To be feasible, the number of inputs (and outputs) to (from) S and R must be equal. When $N \backslash S$ is created, there are two sets of nets left dangling from the cut. These cut points are connected to two sets of new nodes: the set of nets I which are inputs to S, and the set of nets O which are outputs of S. From I, O, N, and S, one can construct loop graph L from $N \backslash S$. Using L, it is possible to constructively create R in a way which creates no loops when stitched. However in the case where R is pre-existing, *e.g.*, created using a synthetic generator such as `cgen` which is unaware of external loop constraints, we construct permission graph P from L and a dependence graph D from R. The circuit stitching problem is then reduced to finding a mapping which shows that D is monomorphic to P.

2.1. Loop Graph L of $N \backslash S$

Given $N \backslash S$ and its cutpoints I (outputs driving S) and O (inputs driven by S), we can construct the loop graph, $L(I_L, O_L, E_L)$ shown in Figure 2a. The loop graph summarizes which nets in O_L connect through combinational logic to nets in I_L. Hence, it is a bipartite graph with nodes I_L and O_L and edges E_L. The edge set contains a "back" edge from a node in O_L to a node in I_L if a combinational path exists between those nodes in $N \backslash S$. This edge set can be constructed by a simple traversal of the network. Since existing circuit generators are not aware of L, we must attempt to match the cutpoints of $N \backslash S$ with the inputs and outputs of R in a way which is cycle-free.

2.2. Dependence Graph D of R

Given R, we compute a dependence graph $D(I_D, O_D, E_D)$ of R. The dependence graph is a bipartite directed graph that contains a vertex for each input and each output in R. Inputs that drive outputs through combinational logic are represented by a directed edge from the corresponding input

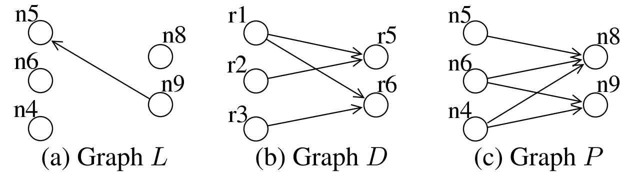

Fig. 2. (a) The loop graph, L, for the circuit $N \backslash S$ in Figure 1a. (b) The dependence graph, D, for the replacement circuit, R from Figure 1b. (c) The permissible graph, P, for the circuit $N \backslash S$ in Figure 1a.

vertex in I_D to the output vertex in O_D. The dependence graph for circuit R in Figure 1b is shown in Figure 2b.

The problem of stitching R into $N \backslash S$ is that of finding two 1–to–1 mappings $f_I : I_D \rightarrow I_L$ and $f_O : O_D \rightarrow O_L$ such that the merge of $D(I_D, O_D, E_D)$ with $L(f_I(I_D), f_O(O_D), E_L)$ contains no cycles. This problem is combinatorially complex and we were unable to precisely map it to any known problem. In order to solve it, we transform the problem into an approximation of the original form so that existing graph tools can be used to find a solution.

2.3. Permissible Graph P of $N \backslash S$

The transformation we use is to construct a permissible graph, $P(I_P, O_P, E_{P_i})$ from $L(I_L, O_L, E_L)$ as shown in Figure 2c. This graph contains the same nodes as L, but the edges of P represent a set of permissible forward edges that will not create a cycle if P is merged with L.

It is not sufficient to simply omit forward edges in P if there exists a corresponding back-edge in L. For example, for every pair of back edges in L that connect with distinct nodes, there is a corresponding "criss-crossing" pair of forward edges which cannot exist simultaneously in P. Figure 3 illustrates this situation by showing a complete loop graph, L, and two edges in a permissible graph, P, which cannot exist simultaneously. If the two edges in Figure 3 did exist in P, and a mapping made use of both the edges, then a combinational loop would be created. To be maximal in the number of edges, P must be constructed with only one of these forward edges but not both. Hence, P cannot be uniquely determined from L, and in fact there are numerous maximal P_i graphs which can be constructed from L, making the problem challenging. In our methodology, we construct just one P in a particular way as described in Section 3.2.

2.4. Graph Monomorphism Problem

We now express the problem as a graph monomorphism problem. Graph monomorphism is similar to graph isomorphism, except that the number of edges need not be the same. Given the replacement dependence graph, $D(I_D, O_D, E_D)$, and the permissible graph,

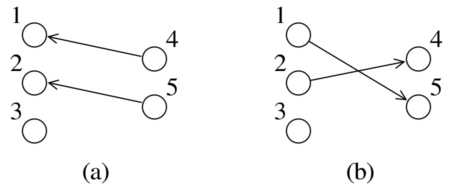

Fig. 3. (a) An example loop graph, L. (b) Two edges that cannot simultaneously exist in P given L.

$P_i(I_P, O_P, E_{P_i})$, a 1–to–1 function that maps the vertices from $I_D, O_D \rightarrow I_P, O_P$ is required ($f : I_D, O_D \rightarrow I_P, O_P$) such that $\{u, v\} \in E_D$ only if $\{f(u), f(v)\} \in E_{P_i}$. That is, we wish to determine if $D(I_D, O_D, E_D)$ is a monomorphic graph[6] to $P_i(I_P, O_P, E_{P_i})$. If we can do this for any of the possible maximal P_i graphs, then we have computed a cycle–free way to stitch R.

3. BENCHMARK GENERATION PROCEDURE

In this section, we describe the high level flow used to generate R and stitch it back into $N \backslash S$. The tool flow accepts a `BLIF` file as input, and produces a new `BLIF` file which can be used as a replacement for the original circuit.

3.1. Replacement Region Selection

First, a replacement region S is selected from the input `BLIF`. For our experiments, S is selected by clustering the circuit with `TV-Pack` and choosing all the nodes that fall within a specific cluster. All the LUTs, latches, and nets within S are then removed, which creates dangling nets in the original circuit. These nets will be un–dangled when R is stitched back into the circuit.

Given S, we also considered growing S to include all combinational logic surrounding S. An S with cutpoints only on input/output pin and latch boundaries would trivialize the stitching problem, since no edge assignment could create a combinational loop. However, we found that an expanded S in the MCNC benchmarks caused most or all of the original circuit to be selected. Such an expansion may be viable for large real circuits, but is impractical for the MCNC circuits we used.

3.2. Compute Permissible Graph

The nets cut in the removal of S from N are categorized as input or output depending on whether they are an input or an output to S. Recall these are the sets I and O of nets. In Figure 1a, $I = \{n4, n5, n6\}$ and $O = \{n8, n9\}$.

Finding a permissible graph, $P(I_P, O_P, E_P)$, with the maximum number of edges, is NP–Complete (see the "Feedback Arc Set" problem in [7]). We compute an approximation to a maximal P as follows. Begin with $N \backslash S$, and perform a breadth–first search beginning at O, attempting to reach nets in I. For every path from a net in O to a net in I, the corresponding edge from I to O is left unconnected in P, and every other edge from I to O is added. This produces an oversized graph because some edges in P will still create cycles in $N \backslash S \cup P$. To arrive at a valid permissible graph, we run a loop detection algorithm on $N \backslash S \cup P$ to find edges in P that are directly involved in a loop. We select one such edge and remove it, and repeat this process until $N \backslash S \cup P$ is cycle free.

3.3. Characterization and Generation

In order to generate a suitable replacement, S is characterized using `ccirc`. When generating R with `cgen`, the full `ccirc` profile is used. When using `gnl`, the number of inputs, outputs, LUTs, and latches are taken from the `ccirc` profile and used to create a `gnl` input file. Either `gnl` or `cgen` is then invoked to generate R.

3.4. Fixing Inputs and Outputs

Because of the generation methods used by `gnl` and `cgen`, R may not contain the exact number of inputs/outputs specified. Thus, R may not precisely match the hole left by $N \backslash S$.

To add inputs to the circuit, a vertex in I_D is selected. A new LUT is then inserted to drive the selected input, allowing additional inputs nets to be added to the circuit. The selected vertex should have a low number of connected edges because each new input duplicates all the dependence edges in E_D, and we wish to avoid unnecessarily complicating the graph monomorphism problem. Removing inputs is done by selecting two input vertices, again with a low number of connected edges, and merging the corresponding input nets into a single net. The creation and removal of outputs use similar techniques to merge output nets or create new ones.

3.5. Stitching

The permissible graph, P, and the dependence graph D of R are used to stitch R into $N \backslash S$. This step assigns the inputs and outputs of the replacement circuit to all dangling nets. Verification is also done to ensure the final circuit contains no loops and no dangling logic. The stitching procedure is described in Section 4. The output of this step is a `BLIF` suitable for input into any tool flow the original `BLIF` could have be used in.

4. STITCHING ASSIGNMENT

The procedure used to stitch a clone into the original circuit is to first solve the graph monomorphism problem. That is, determine if the graph $P(I_P, O_P, E_{P_i})$ is a monomorphic

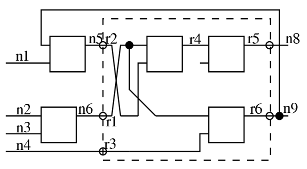

Fig. 4. A correct stitching for R in Figure 1.

graph to $D(I_D, O_D, E_D)$, and determine the vertex mappings. Using the vertex mappings, the inputs and outputs of R are assigned to the cut nets in the original circuit, and a new `BLIF` file is created.

The CP(Graph+Map)[8] tool solves the general graph monomorphism problem by using an exact search method to find a solution, but has several pruning strategies to reduce the problem search space. The solution is a list of vertex pairs (r_i, n_j), indicating that vertex r_i in graph $D(I_D, O_D, E_D)$ should be mapped to vertex n_j in graph $P(I_P, O_P, E_{P_i})$. It is possible that there is more than one valid mapping for a circuit. One mapping for the stitching problem in Figure 1 is: $(r1, n6), (r2, n5), (r3, n4), (r5, n8), (r6, n9)$. This solution is shown in Figure 4, and is cycle free. Many times, CP(Graph+Map) finds a solution quickly. In other cases, it runs for a long time searching for a solution. We find that stopping it after 20 seconds and restarting it with a different seed increases the success rate; this is an imprecise heuristic.

5. BENCHMARK GENERATION AND TESTING

Using the 20 largest circuits in the MCNC benchmark suite, we have generated new benchmarks for testing incremental CAD tools. For each original MCNC circuit (N), we clustered the circuit using `TV-Pack` with a cluster size of 64. We then selected 10 of the clusters to be used as the replacement region (S). For each region we generated 10 different clones (R) and stitched each one back into the original, creating 100 different variations of the original circuit. We then repeated this exercise using a cluster size of 128. For larger replacement regions where there could be several hundred inputs and outputs to match, finding the solution to the graph monomorphism problem becomes the bottleneck.

The results obtained by using `ccirc` on the generated incremental benchmark circuits showed the characteristics matched very closely with those of the original circuit, except for one major feature. The maximum depth of the circuit exhibited a two–to–three fold increase, which we traced to a single cause: no attempt is made to match the old depth of each path through the circuit as the clone is stitched in. The solution to the graph monomorphism problem often found a mapping solution that caused the critical path to re–enter R several times (while avoiding combinational loops), and thus considerably increasing the critical path.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new method for benchmark generation which is ideal for testing incremental place and route tools. We have formulated the exact stitching problem which is combinatorially complex, and showed how the problem can be approximated and solved with existing graph theory and tools. Using this process, a number of benchmarks were generated from the 20 largest MCNC benchmarks. Analysis of the benchmarks showed they were similar to the original circuit except for an increase in logic depth. The cause of this increase was identified and can potentially be addressed in future work. Testing also showed that no combinational loops were created by the stitching process, which was the focus of this paper.

We wish to use the procedure described in this paper to create more benchmarks by using additional cluster sizes or by varying other parameters. Of particular interest in testing incremental place–and–route tools is varying the size of R with respect to S, so the tools need to make room for a larger R or fill the hole left by a smaller R.

7. REFERENCES

[1] D. Stroobandt, P. Verplaetse, and J. van Campenhout, “Generating synthetic benchmark circuits for evaluating CAD tools,” *IEEE Trans. on CAD*, vol. 19, no. 9, pp. 1011–1022, 2000.

[2] M. Hutton, J. Rose, and D. Corneil, “Automatic generation of synthetic sequential benchmark circuits,” *IEEE Trans. on CAD*, vol. 21, no. 8, pp. 928–940, 2002.

[3] P. Kundarewich and J. Rose, “Synthetic circuit generation using clustering and iteration,” *IEEE Trans. on CAD*, vol. 23, no. 6, pp. 869–887, 2004.

[4] J. Pistorius, E. Legai, and M. Minoux, “Generation of very large circuits to benchmark the partitioning of FPGAs,” in *ISPD ’99: Proceedings of the 1999 International Symposium on Physical Design*, 1999, pp. 67–73.

[5] M. Tom and G. Lemieux, “Logic block clustering of large designs for channel-width constrained FPGAs,” in *DAC ’05: Proceedings of the 42nd Annual Conference on Design Automation*, 2005, pp. 726–731.

[6] S. Zampelli, Y. Deville, and P. Dupont, “Approximate constrained subgraph matching.” in *Principles and Practice of Constraint Programming*, 2005, pp. 832–836.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[8] Y. Deville, G. Dooms, S. Zampelli, and P. Dupont, “CP(Graph+Map) for approximate graph matching,” in *1st International Workshop on Constraint Programming Beyond Finite Integer Domains*, Oct. 2005, pp. 31–47.