Perturb+Mutate: Semisynthetic Circuit Generation for Incremental Placement and Routing

DAVID GRANT and GUY LEMIEUX
The University of British Columbia

CAD tool designers are always searching for more benchmark circuits to stress their software. In this article we present a heuristic method to generate benchmark circuits specially suited for incremental place-and-route tools. The method removes part of a real circuit and replaces it with an altered version of the same circuit to mimic an incremental design change. The alteration consists of two steps: *mutate* followed by *perturb*. The perturb step exactly preserves as many circuit characteristics as possible. While perturbing, reproduction of interconnect locality, a characteristic that is difficult to measure reliably or reproduce exactly, is controlled using a new technique, *ancestor depth control* (ADC). Perturbing with ADC produces circuits with postrouting properties that match the best techniques known to-date. The mutate step produces targetted mutations resulting in controlled changes to specific circuit properties (while keeping other properties constant). We demonstrate one targetted mutation heuristic, scale, to significantly change circuit size with little change to other circuit characteristics. The method is simple enough for inclusion in a CAD tool directly, and fast enough for use in on-the-fly benchmark generation.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; B.6.0 [**Logic Design**]: General

General Terms: Algorithms, Design, Experimentation, Measurement, Reliability, Verification

Additional Key Words and Phrases: Automated development tools, design automation, graph algorithms, hardware-supporting software, place and route, testing

ACM Reference Format:

Grant, D. and Lemieux, G. 2008. Perturb+Mutate: Semisynthetic circuit generation for incremental placement and routing. ACM Trans. Reconfig. Technol. Syst. 1, 3, Article 16 (September 2008), 24 pages, DOI = 10.1145/1391732.1391736 http://doi.acm.org/10.1145/1391732.1391736

This research has been made possible through the use of WestGrid computing resources, which are funded in part by the Canada Foundation for Innovation, Alberta Innovation and Science, BC Advanced Education, and the participating research institutions. WestGrid equipment is provided by IBM, Hewlett Packard and SGI. This work was also funded by Altera and NSERC.
Authors' address: D. Grant (corresponding author), G. Lemieux, Department of Electrical and Computer Engineering, The University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4; email: davidg@ece.ubc.ca.

1. INTRODUCTION

In-system reprogrammability with FPGAs gives system designers a key advantage over using ASICs. It enables the construction of a simple, bare-bones prototype which can be modified and extended until the final design is done. These incremental design changes can arise for a number of reasons, including additional functionality or late requirements changes (ECOs), bug fixes, workarounds for errors in the circuit board or other chips, and debugging by adding circuitry for observability, controllability, and logic analysis.

Although there are many types of incremental design changes, a few use-cases do not really need incremental CAD tools. For example, large-scale ECOs or generation of the final production bitstream almost always entails a full recompile. In contrast, debugging in-circuit or adding small features to a system under test greatly benefits from incremental CAD tools. In these very common use-cases, the tools produce a temporary bitstream to implement the change.

As a result, incremental place-and-route performance is particularly important for FPGAs. The incremental tools must recompile successfully and quickly, and meet all timing constraints. It is also expected that the incremental tools are *stable*. This means that a reasonable change to the input circuit (i.e., one which does not significantly alter the properties of the netlist) should recompile quickly and with a similar result to the original.

Since the bitstream is only for temporary use, incremental FPGA tools are not usually concerned with meeting power constraints or further improving timing. This is in contrast to ASIC tools, where incremental tools are more commonly used for *incremental improvement* of metrics such as wirelength, timing, or power [Cong and Sarrafzadeh 2000; Coudert et al. 2000]. While FPGA tools do need a similar incremental improvement mode, we are not concerned with that mode of usage in this work. Instead, we are concerned with the place-and-route tool performance in response to *incremental design changes*.

To assist with the development and evaluation of incremental place-and-route algorithms targeting design changes, a set of benchmark circuits is required. Such *incremental circuits* must be specified in two forms: an original form and a modified form. We are unaware of any benchmark circuits available for this purpose.[1] Existing circuit generation schemes produce entire *synthetic* circuits, but they do not produce incremental circuits. In fact, we were unable to easily modify them to do so: Our first attempt resulted in a scheme that is computationally expensive and unable to control logic depth [Grant et al. 2006].

To generate incremental circuits, we start with a real circuit. From this, the modified circuit is created by replacing a subcircuit of the original with a synthetically generated subcircuit. Since the modified version consists of both real and synthetic parts, we say these incremental circuits are *semisynthetic*.

[1]The difficulty of obtaining benchmark circuits for testing CAD tools is already widely acknowledged. Obtaining full design changes for a benchmark, which may expose the history of bugs or some potentially embarrassing or litigious artifact in the development process, is even more difficult.

In this article, we describe the *Perturb+Mutate* approach to generating synthetic or semisynthetic circuits, also called clones. The overall technique is simple, quick, and produces synthetic results which closely reproduce most netlist topological properties, as well as postrouting properties.

Perturb [Grant and Lemieux 2006] produces a variation of an original circuit with the objective of exactly preserving as many properties of the original netlist as possible. Perturbing is based on edge-swapping, a technique that has been used in several prior efforts [Ghosh et al. 1998; Coudert et al. 2000, Kundarewich and Rose 2004]. However, an important feature unique to our Perturb tool is ancestor depth control (ADC), a method used to preserve circuit locality during swapping. Ancestor depth control results in a clone circuit that reproduces the postrouting properties (channel width, wirelength, and critical-path delay) of the original circuit as well as *CGen* [Kundarewich and Rose 2004], the best generator known to-date.

Mutate is intended to controllably change one characteristic, such as the size of a circuit, while keeping all other properties intact. In molecular biology and genetics, this level of control is called a targetted mutation. It can be difficult to produce targetted mutants because one change to the primary characteristic often produces unintended side effects in other characteristics. However, targetted mutants are valuable for experimentation with CAD tools because they provide greater control by limiting the number of changed variables.

The key difference between our previous work and Perturb+Mutate is that the latter distinguishes between transformations that preserve characteristics (Perturb) of the circuit from those that controllably modify them (Mutate). Mutate can be used, for example, to test the ability of the incremental placer to create room for added debugging logic. In this article, we propose one mutation heuristic that first upscales a circuit via replication and then downscales by random subsampling. By following this with a Perturb step, the mutant can be further obscured from its original source.

Simplicity and speed allow Perturb+Mutate to be easily embedded directly into an incremental CAD tool so it can test itself thoroughly and quickly. For example, once the tool and original benchmark are loaded, a Perturb and incremental CAD pass can be tested without saving or reloading the netlist or restarting the tool. A predefined series of clones, each followed by an incremental update, can also be applied in rapid succession to thoroughly test the flow. This can also save a significant amount of time in overhead, for example, for rebuilding the routing graph of the architecture or the unchanged portion of the netlist.

The remainder of this work is organized as follows. Section 2 gives the background and previous work to the circuit generation approach. Section 3 provides terminology and an overview of Perturb+Mutate. Section 4 presents Perturb and ancestor depth control. Section 5 presents Mutate. Section 6 presents several additional techniques which were unsuccessful at controlling the postrouting results for Perturb. Directions for future work specific to Perturb+Mutate are presented in Section 7, and conclusions are given in Section 8.

The incremental circuits and generation tools described in this article are available online at `http://www.ece.ubc.ca/~lemieux/downloads`.

2. BACKGROUND AND PREVIOUS WORK

This section introduces previous work on incremental CAD for FPGAs and benchmark circuit generation.

2.1 Incremental CAD for FPGAs

There is very little published work on incremental place-and-route algorithms for FPGAs. One placement algorithm, known as ICP, is an incremental improvement flow [Singh and Brown 2002a, 2002b]. It is designed to apply small netlist changes to improve timing. We are not aware of any published algorithms that target design changes for FPGAs. However, both Altera and Xilinx support incremental placement and routing modes for design changes, suggesting that design-change flows are very important in practice.

2.2 Benchmark Circuit Generation

This article is concerned with generating benchmarks for incremental placement-and-routing tools used in design-change flows. Since there is no known prior work, a de facto benchmark suite does not yet exist for this purpose. We started this circuit generation effort to test our own incremental placement tool aimed at design changes [Leong 2006].

Netlist changes produced by incremental improvement flows are inappropriate as benchmarks for design-change flows. Improvement-based flows use automated methods such as retiming algorithms [Singh et al. 2005] to iteratively propose and apply numerous small netlist changes. The sequence of netlist changes, such as placement-moves or duplications, depends heavily upon the success of previous netlist changes and the current physical mapping information. These are often oriented towards correcting physical layout, such as straightening critical nets.

In contrast, design-change flows are intended for netlist changes made by the user, not those proposed by the CAD tool. As a result, the types of changes introduced by the user involve modifications of a larger scale that that do not rely upon details of the previous mapping solution. For example, design changes may involve significant rewiring to fix bugs, or the addition or removal of gates to change features. This assumption only affects the way in which we generate incremental circuits; it does not prevent the CAD tools from using the original mapping solution to reduce the runtime to produce the new mapping solution.

There are several studies published on generating whole synthetic circuits that possess the properties of real circuits. The *rmc* tool stochastically generates circuits in a top-down fashion with just a few parameters: the number of LUTs, total input pins used on all LUTs, primary inputs, primary outputs, and the Rent parameter [Darnauer and Dai 1996]. The *gnl* tool adds two parameters to this list, net degree distribution and terminals-per-block distribution, and generates synthetic circuits in a bottom-up clustering approach [Stroobandt et al. 2000]. In both of these methods, the Rent parameter captures locality information. However, they are not ideal: specifically, rmc may create combinational cycles, and gnl is unable to control delay characteristics.

The tools *CIRC* and *GEN* were created to first measure key circuit properties and then generate clones based on these properties [Hutton et al. 1998]. These tools define several circuit characteristics such as the circuit shape and the edge-length distribution. Definitions for these properties are given in Section 3.1.

However, Verplaetse et al. [2000] show that GEN does not preserve locality very well. In Hutton et al. [2002], the tools are extended to include sequential circuits. Refinements to improve the reproduction of locality were introduced in the *CCirc* and *CGen* tools [Kundarewich and Rose 2004]. CCirc first partitions a circuit and characterizes the partitions separately, then CGen generates clusters accordingly and joins them together. Another change in CGen is the use of iterative edge swaps to better match properties of the generated circuit to the specified characteristics. These changes make CGen dramatically better at preserving wirelength, routed channel width, and also delay. CGen is the best synthetic generator known to-date, but it is unable to scale circuit size and cannot generate incremental circuits.

Methods to promote greater realism were introduced in Pistorius et al. [1999] and Tom and Lemieux [2005]. These methods stitch together real circuits as if they are IP blocks or subcircuits within a larger design.

Another circuit generation approach involves perturbing a real circuit through a sequence of edge swaps to create a synthetic clone [Kapur et al. 1997; Ghosh et al. 1998].[2] The perturbations preserve certain wiring characteristics of a circuit, collectively called the wiring signature. To preserve the wiring signature, perturbations must abide by a set of 13 rules given in Ghosh et al. [1998]. This set of rules is larger than necessary and does not preserve important properties such as depth profile, fanout distribution, and edge-length distribution. This can negatively impact delay characteristics, which was untested in Ghosh et al. [1998]. More importantly, the perturbations also destroy interconnect locality [Verplaetse et al. 2000].

All past generation methods have focussed on generating an entire synthetic circuit. To solve the problem of creating *incremental* benchmark circuits for place and route, we have developed our Perturb method [Grant and Lemieux 2006] using a simplification of the method from Ghosh et al. We have also added ancestor depth control, a new mechanism to preserve locality. Our Perturb technique preserves more circuit characteristics than any previous circuit generation scheme.

Previous schemes such as GEN strive to reproduce topological netlist features as accurately as possible in the synthetic clone, but due to randomness and imprecise heuristics these techniques generate an approximate clone rather than a true clone. While unpredictable deviation from precise specifications can be argued to be a feature, it should be noted that the extent of these deviations are uncontrolled, even when the algorithm attempts to reduce them.

[2]In their work, Ghosh et al. call a perturbed circuit a "mutant". Throughout this article, we use the terms "perturb" or "mutate" to describe minor or major circuit changes, respectively, resulting in a synthetic circuit.

In contrast, the ability to preserve these properties precisely with Perturb helps to control the reproduction of circuit characteristics. However, it is also possible to modify the rules to allow it to alter some circuit characteristics in a controllable fashion; Mutate is an example of this. The ability to precisely control more circuit parameters than any other previous scheme makes this approach a better resource for observing the effect of each parameter on synthetic circuit quality.

2.3 Incremental Circuit Generation Objectives

The main goal of traditional full-circuit generators is to closely mimic the properties of real circuits so that the CAD tools are given a realistic workload. If it is unrealistic, the tools will be unable to exploit properties known to exist in real circuits. For example, a random graph is a poor synthetic circuit because there is no connection locality, making it difficult for placement tools to reduce congestion.

Similarly, an incremental circuit generator might be expected to produce incremental changes that mimic those made to a real circuit. Real changes may result in large, medium, or small alterations to the netlist. Large alterations are not good for testing incremental tools, since there is little similarity in the netlist to exploit. For example, consider a large alteration that significantly degrades routability or delay. This should not be used to benchmark an incremental tool because there is no reasonable way the tools can avoid the degradation. The tools are placed at an immediate disadvantage and will likely run slowly, fail to route, or fail to meet timing. Also, although medium alterations are interesting, it is difficult to define expectations. Is the problem easier or harder? Is the designer expecting good performance? Should a full recompile be done?

For small alterations, however, it is clear that the tools must always perform well. They must be stable, complete successfully and quickly, and meet timing. To ensure there is reasonable chance for success in these objectives, the incremental circuit should be similar in structure to the original. However, it should also be different enough to present a different problem to the tool (not just the same circuit). Hence, the objective of our incremental circuit generator is to closely mimic as many properties of the original circuit as possible while still exemplifying a change.

The modified form should have similar routability and delay to the original. If the incremental tools can preserve routability and delay across several different incremental changes, each with “similar difficulty” to the original, this gives confidence that the tool is stable. Then, for stable input changes provided by the designer (like a minor bugfix), the tool can be expected to produce a stable result.

We do not consider generating “more difficult” incremental circuits in this article because we believe the problem of generating “similar difficulty” incremental circuits to be more challenging and more useful. Likewise, we do not consider “easier” incremental circuits that are expected to significantly improve delay or routability. In those cases, a full place-and-route should be done to capture the full improvement; relying upon incremental tools to properly reflect

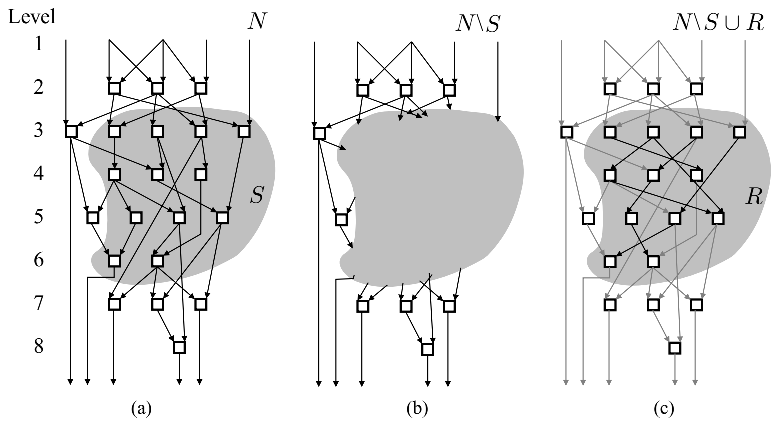


Fig. 1. Diagram for semisynthetic circuit generation: (a) the original circuit N, with a selected subcircuit S; (b) S is removed to produce $N\backslash S$; (c) R is generated using Perturb+Mutate and stitched into $S\backslash R$ to produce $N\backslash S \cup R$. Wires that remain in their original location are shown in gray.

this improvement is unwise, since it may be lost the next time a full place-and-route is performed. While future incremental circuit generators may wish to expand upon this method, we believe it to be a prudent starting point.

3. PERTURB AND MUTATE OVERVIEW

This section describes the general approach taken to solving the problem of producing an incremental benchmark circuit (Section 3.1). We then present our implementation of this general approach in Section 3.2, which focuses on two tools created for this purpose: Perturb and Mutate.

3.1 Terminology and Circuit Change Model

Our general approach for creating an incremental benchmark circuit starts with an original circuit which is represented by a directed acyclic graph (or DAG) N, as shown in Figure 1(a). To produce this graph, a sequential circuit is transformed into an equivalent network of combinational logic by cutting the graph so that all flip-flop outputs appear as virtual inputs to the circuit, and all flip-flop inputs appear as virtual outputs. The nodes of the graph are logic blocks (LUTs) of the combinational logic, while each directed edge represents a *single* fanout connection from source to sink. Multiple fanouts of a single net in the circuit are represented with multiple edges in the graph. A significant concern in this article is avoiding the creation of a cycle in the DAG when modifying the edges. Such a cycle would imply a combinational loop in the circuit, which is not allowed.

As originally defined in Hutton et al. [1998], the terms “circuit size”, “number of I/Os”, “delay level”, “shape”, “edge-length distribution”, and “fanout distribution” are used to describe properties of the circuit. The *size* is the number of nodes. Latches, inputs, and constant drivers are assigned a delay level of 1. A forward breadth-first traversal labels remaining nodes with a *delay level* equal to 1 plus the maximum level of its predecessors. The *shape* is the histogram of nodes per delay level. An *edge length* is the difference in delay levels of its source and sink nodes. The *edge-length distribution* is a histogram of edges per length. *Fanout distribution* is a histogram of nodes per fanout, where *fanout* is the total number of outgoing edges from a node.

For an incremental benchmark, a significant portion of the generated circuit should be identical to the original form. To do this, we consider a *region of change* in the DAG, which is a vertex-induced subgraph S of N. Since N is the input circuit, S is a subcircuit of N, as shown in Figure 1(a).

The subcircuit S is removed to produce $N \backslash S$ (Figure 1(b)) and then replaced with a replacement R (Figure 1(c)). The replacement R should be created using information in $N \backslash S$ to avoid creating combinational loops in the overall circuit.

We assume that R must interact with $N \backslash S$ through the same input and output signals that were used by S. This means that all changes are entirely localized to the selected subcircuit. This does not really affect the generality of our approach, since we can choose S to be as large as we wish to contain all changes. Also, the addition of new primary inputs or primary outputs to R at chip level does not present any difficulty for this model because these new signals do not interact with the rest of the circuit ($N \backslash S$). Although these new inputs or outputs would not be difficult to add, we do not capture the effect of these new signals with our methodology.

This model is convenient since it captures most types of design changes. For example, to model critical path changes, S could be selected to include the chain of logic along the critical path (or a portion thereof) and then altered accordingly. Note that the graph model only directly captures logic depth, so the critical path must be identified by traditional timing analysis. Alternatively, S could be an independent IP block with an added feature to produce R. In this second case, the predefined I/O interface to the IP block remains the same. To model situations where the IP-block interface must change, S could be considered to include the IP block plus all associated logic that is needed to keep the change contained.

Our early effort at producing incremental circuits tried to adapt standard full-circuit generation methods to create replacement R. This approach worked, but the process of stitching R back into $N \backslash S$ without forming combinational loops is nontrivial [Grant et al. 2006]. The loops arise because R is generated blindly, without knowledge of combinational paths in $N \backslash S$ that may connect outputs of R back to inputs of R. There is no easy way to constrain generators like gnl or CGen from creating input-output paths that would form a loop. Instead, we attempted to take the blindly generated R and stitch it back into $N \backslash S$ more intelligently to avoid loops. This requires solving a graph monomorphism problem, which is computationally challenging. Instead, we devised the Perturb method to directly avoid creating loops and simplify stitching by controlling how R is generated.

3.2 Perturb and Mutate Flow

This section outlines the tool flow for two tools, Perturb and Mutate, used in our implementation of the general approach presented the previous section. Our flow starts by computing the depth level of each node, as shown in Figure 1(a). This level information will be used later by Perturb to preserve the depth profile and avoid creating combinational loops in the final circuit.

Next, the subcircuit S is selected and removed from N (Figure 1(b)). There are many possible ways of selecting S, including random selection, placement-based selection, design hierarchy, top-down partitioning, or bottom-up clustering. For simplicity, our flow uses the bottom-up clustering tool *T-VPack* [Marquardt et al. 2000] to form large clusters containing hundreds of lookup tables, and we randomly select one of these clusters as S. Clustering ensures that highly connected logic is grouped together, and by selecting a single cluster for S we are more likely to select a module or a component that a user may be debugging or updating.

Last, S is passed through Mutate to produce a temporary circuit T, then altered with Perturb to produce R, and finally R is inserted back into the original circuit, as shown in Figure 1(c). Careful restrictions in the Perturb tool simplify stitching R and guarantee the final circuit is free of combinational loops.

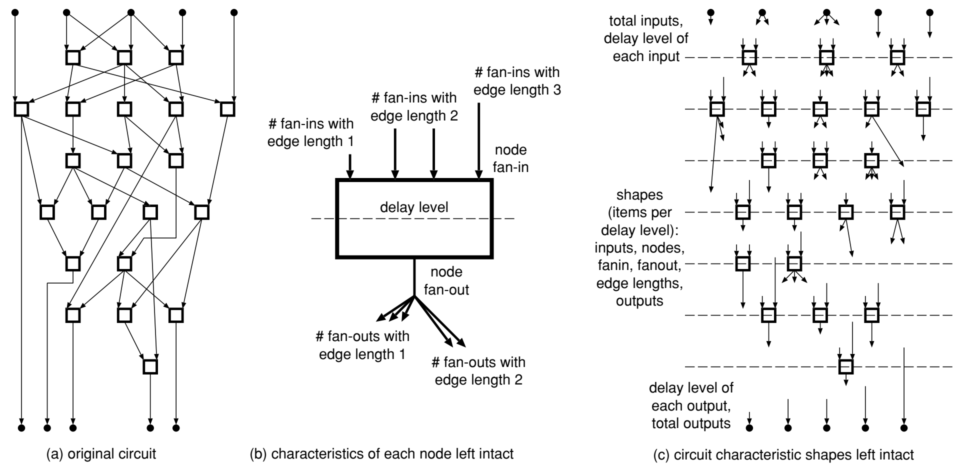
The Perturb tool creates R by using S or T directly. Instead of capturing various properties of S and using these to generate R (as done with approaches like CCirc+CGen), T is iteratively perturbed until a new circuit is produced. Each perturbation involves swapping some of the edges in T with other edges in T. These swaps are done under some simple restrictions, presented in Section 4.1, to preserve many of the circuit characteristics of T in R. Further, the input and output nets in T and R are identical, which trivializes the process of stitching R into $N\backslash S$.

The Mutate tool produces targetted mutations in S before it is perturbed into R. As an example, we implement a scaling mutation that changes the number of nodes in the circuit while keeping other circuit characteristics relatively unchanged. We use a two-step approach to achieve arbitrary circuit scaling using two simple scaling techniques. The circuit is first enlarged by creating multiple parallel copies of S and tying the inputs and outputs together with multiplexers. The circuit is then reduced by randomly selecting and deleting nodes under certain restrictions to maintain the validity of the circuit. A similar process can be developed to controllably modify other circuit characteristics, such as logic depth (placing more logic in series), shape (pushing nodes in some logic levels forward or backward), and so forth.

4. PERTURBING A CIRCUIT

This section describes the procedure we call "perturbing" a circuit to rapidly create a clone of this circuit. This procedure is implemented in the Perturb tool. In contrast, Section 5 presents the Mutate tool and performs testing with the combined Perturb+Mutate tool. Only the Perturb tool is considered in this section. Perturb can be used to modify a complete circuit or to modify just a part of it.

To visualize the types of properties of the circuit that are preserved by Perturb, consider Figure 2. The original circuit is shown levelized in Figure 2(a). There is a one-to-one mapping of nodes in the original and modified circuits:



total inputs, delay level of each input
# fan-ins with edge length 3
# fan-ins with edge length 2
# fan-ins with edge length 1
node fan-in
delay level
node fan-out
# fan-outs with edge length 1
# fan-outs with edge length 2
shapes (items per delay level): inputs, nodes, fanin, fanout, edge lengths, outputs
delay level of each output, total outputs
(a) original circuit
(b) characteristics of each node left intact
(c) circuit characteristic shapes left intact

Fig. 2. Circuit properties left intact by Perturb.

Each pair of nodes will be identical in the properties shown in Figure 2(b), namely in the delay level, fanin, fanout, and distribution of edge lengths among the fanins and fanouts. The original and modified circuits will also have identical shape characteristics to those shown in Figure 2(c), including node, input, output, fanin, fanout, and edge-length shapes. The directed edges in Figure 2(c) are drawn in proportion to their required edge length. The goal of Perturb is to generate modified circuits and perfectly preserve all of these properties by matching the input edges of the nodes with the output edges of other compatible nodes.

Of course, if so much of the original circuit is kept intact, one must also ask whether the perturbing procedure alters enough of the circuit to create a clone. Is the clone a fraternal twin or identical twin? We will address this question later when examining the place-and-route results of the clones.

4.1 Perturbation Procedure

The tool flow starts with the complete circuit, represented by N, and the induced subset S, $S \subseteq N$, which has been identified in N but not yet removed. N is first "levelized" to determine the delay level of each node.

Once the level of each node is known, a list containing all the edges for the nodes between any two given levels in S can be created. An edge represents a connection from one source to one sink. An *edge swap* is the exchange of sinks between the two source nodes. The perturbation method randomly selects two edges from this list and swaps them, subject to the following conditions.

(1) The two edges must come from different source nodes.
(2) The source and sink levels of both edges to be swapped must match. Swapping edges with mismatched source or sink levels may be valid, but would necessitate a recomputation of the levels of all nodes in the fanout cone of the edges swapped. The lists of edges between each pair of levels in the circuit would also need to be updated or rebuilt. For large circuits, this can

significantly increase the time to perturb the circuit. Ensuring that the source and sink levels match also preserves the edge-length distribution and depth profile of the circuit.

(3) The edge swap must not create a multigraph. In other words, the swap is rejected if it would result in multiple edges between the same source and sink.

(4) The source node cannot be level 1. A level-1 source is either an input to the original circuit or the direct output of a latch. In either case, we allow inputs and latch outputs to proceed through one level of logic, to reduce the probability of directing the signal to a completely different branch of logic in the circuit.

Only edges are considered for swapping. The nodes (LUTs and latches) in the circuit are left untouched because they do not need to be moved around. An incremental user change to a circuit may modify the contents of a LUT, but this is irrelevant to incremental placement and routing.

Perturb takes a single parameter, the *perturbation factor*, which is the percentage of edges to modify in S. All the results in this article use a perturbation factor of 25%. Tests using perturbation factors of 12% and 50% gave postrouting results that were not significantly different; however, at 50% Perturb required disproportionately more time to generate R, since edges were returned to the original position of a swapped edge with increasing frequency. Edges swapped into the original position of another edge, or to their own original positions, are not counted as swapped.

When Perturb is finished, the output R is stitched into the hole left by the removal of S from N. The stitching process simply matches the names of the nets in R with those which were cut when S was removed from N. The nodes and edges in R are copied into $N \backslash S$ and the fanins/fanouts for matching input/output edges in R are reconnected, creating a complete circuit.

By only swapping edges with matching source and sink levels, the level of each node in the circuit remains the same. This means the original levelization is preserved, and therefore no combinational loops have been introduced into the circuit and all shape characteristics are left intact.

Place-and-route tests using this method show promising results when S is small relative to N. When a larger S is used, however, the results become unacceptable compared to existing circuit generators. The problem is that the locality of the edges is not considered during swapping, causing R to become irregular and difficult to route (like a random circuit). In the next section we place an additional restriction on Perturb, with the goal of preserving locality and postrouting characteristics.

4.2 Ancestor Depth Control

The major problem with Perturb, as described before, is that locality is not considered during edge swapping. For example, two unrelated buses could easily end up "cross-connected." This destroys the "nice" regular features of the circuit and makes it harder to route.

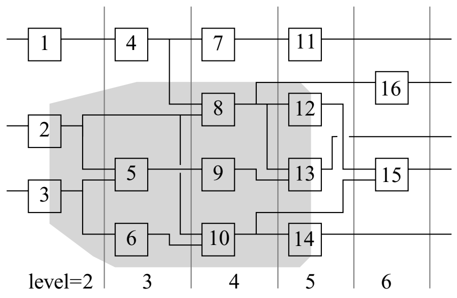


Fig. 3. Ancestor selection region (shaded) for the net connecting nodes 10 and 14 with $d = 2$.

To control the locality of perturbations done to the circuit, we restrict Perturb to only swap edges within related chains of logic. To do this, an additional restriction on the edge-swapping criteria is added, as follows.

(5) Both edges to be swapped must share a common ancestor through combinational logic within a certain ancestor depth. This is called *ancestor depth control (ADC)*.

The ADC is specified by a single parameter, d. When an edge in the circuit is selected, the edge has a *selected source* and a *selected sink*. A list of candidate edges for the swap is computed dynamically by first finding all the ancestors within d levels of the selected source, then walking forward through all fanouts from each ancestor back to the original depth. For each LUT visited, if the LUT level matches the level of the selected source, and this LUT contains a sink that matches the level of the selected sink, then the associated edge may be swapped with the selected edge.

Figure 3 shows part of an example network which can be perturbed. If the net between nodes 10 and 14 is selected by Perturb, the selected source is node 10 at level 4, and the selected sink is node 14 at level 5. For an ADC value $d = 2$, the list of candidate swap edges would be built by walking backwards from node 10, visiting nodes 6, 5, 3, and 2. Nodes 2 and 3 are two levels away from the starting node, so the backtrace stops and begins to follow all forward paths until each path reaches the original depth of 4. The procedure would find nodes 8 and 9 in addition to the nodes identified by the backtrace. The final step is to evaluate all outputs of all identified nodes and add those edges to the candidate list which meet the criteria identified in Section 4.1. In this example, the candidate netlist would include the edges between the following pairs of nodes: (8, 12), (8, 13), (9, 13). Of these edges, one would be randomly selected, say (8, 13), and the sinks would be swapped. The final circuit would thus contain edges between nodes (8, 14) and (10, 13). The original perturbation procedure would have also included the edge between nodes (7, 11) in the candidate list, but the ADC method excludes it.

It is worth noting that ADC does not perpetually restrict edge swaps between unrelated edges. Two edges which may not have had a close ancestor in the past may eventually find they have a close ancestor after a number of edge swaps have taken place. Although we have not formally measured the extent of this

Table I. Place-and-Route Results for Unmodified MCNC Circuits

	Nodes			CW Channel Width	CP Critical Path	WL Wire Length
Name	(4-LUTs)	(DFFs)	Edges	(tracks)	(ns)	(CLBs spanned)
alu4	1522	0	2533	33	12.0	9301
apex2	1878	0	4073	47	13.1	15794
apex4	1261	0	3626	49	12.1	11085
bigkey	1699	224	4945	46	6.0	9097
clma	8364	33	24958	67	24.4	83587
des	1591	0	3679	58	10.7	11050
diffeq	1494	377	5069	34	15.8	8979
dsip	1362	224	3383	42	5.9	6962
elliptic	3602	1122	12037	55	20.4	30388
ex1010	4598	0	15643	58	16.6	42961
ex5p	1064	0	3218	49	12.4	9525
frisc	3539	886	12730	54	26.8	30152
misex3	1397	0	3137	42	11.4	10164
pdc	4575	0	15654	67	25.8	49485
s298	1930	8	5806	28	21.2	9130
s38417	5974	1463	22294	41	15.6	38076
s38584.1	6192	1260	18641	43	12.8	40122
seq	1750	0	3807	45	14.9	14418
spla	3690	0	12658	58	15.5	33871
tseng	1046	385	3577	39	14.7	6689
			Average:	47.8	15.4	23542

effect, it seems a good strategy to gradually relax the depth control when a larger number of edge swaps is requested.

In the next section, we show that ADC produces a perturbed circuit with postrouting results which are similar to those of the original circuit.

4.3 Perturb Results

In this section we present the results of two experiments. The first experiment uses Perturb to generate complete circuits (synthetic clones) from an original circuit. This is something that would be done to test the stability of results from a full (nonincremental) place-and-route flow, for example, to eliminate noise in measured results. It shows that Perturb generates very effective clones using simpler heuristics than CCirc+CGen. The second experiment uses Perturb to generate incremental circuits with three different sizes for S. This demonstrates how well the postrouting characteristics are preserved after just a subcircuit change. As mentioned at the beginning of this section, Perturb preserves most characteristics of the circuit by design, including the number of nodes, number of edges, fanout distributions, and depth profile. Therefore, only the changes in postrouting results need to be examined in these experiments.

Table I shows the placement and routing results for the 20 largest MCNC benchmarks using *vpr* [Betz et al. 1999]. The "CW" column is the minimum channel width required to route the circuit. The "CP" column is the critical path, in nanoseconds, of routing the circuit using a channel width 20% larger than the value reported in the CW column. The "WL" column is the total wirelength of the final routed circuit. These numbers are the baseline values to be used in conjunction with the percentage-change results for the two experiments presented Tables II and III.

The first experiment compares the routing results of CCirc+CGen clones with those of Perturb with ancestor depth control. For these tests, Perturb was set

Table II. Percent Change in Place-and-Route Results with Fully Synthetic Circuits

	CCirc+CGen			Perturb, operating on entire circuit		
Name	CW %	CP %	WL %	CW %	CP %	WL %
alu4	-0.9 ± 4.3	0.8 ± 6.9	-5.0 ± 2.4	5.3 ± 3.1	6.0 ± 9.7	0.2 ± 1.9
apex2	7.2 ± 2.3	5.1 ± 1.5	5.6 ± 1.6	9.3 ± 1.9	14.5 ± 13.1	7.3 ± 1.2
apex4	-12.4 ± 2.6	0.0 ± 6.7	-11.9 ± 2.7	-1.0 ± 1.5	5.5 ± 7.0	-0.3 ± 1.3
bigkey	-5.4 ± 7.1	7.7 ± 4.7	25.3 ± 2.8	11.7 ± 2.6	8.0 ± 7.0	77.3 ± 2.4
clma	57.3 ± 3.4	14.7 ± 7.9	60.7 ± 2.8	47.2 ± 2.2	9.0 ± 2.5	40.6 ± 1.2
des	12.2 ± 6.7	3.8 ± 2.0	20.5 ± 1.7	-7.1 ± 4.7	1.0 ± 2.0	1.2 ± 1.6
diffeq	19.4 ± 4.0	3.9 ± 7.7	18.6 ± 4.7	9.2 ± 3.3	-4.0 ± 5.8	3.8 ± 3.1
dsip	2.9 ± 3.1	6.1 ± 4.8	21.9 ± 3.5	0.3 ± 4.3	4.7 ± 3.3	38.7 ± 2.3
elliptic	14.5 ± 2.4	-5.3 ± 6.1	26.0 ± 1.5	-0.7 ± 2.2	-0.2 ± 5.9	3.0 ± 3.6
ex1010	12.2 ± 2.4	-1.4 ± 1.7	12.4 ± 2.2	37.1 ± 2.9	2.2 ± 3.4	31.2 ± 1.0
ex5p	-9.6 ± 2.9	5.8 ± 7.6	-8.7 ± 2.6	5.1 ± 1.5	5.9 ± 5.1	4.2 ± 0.9
frisc	27.6 ± 3.3	-1.9 ± 4.1	30.7 ± 3.1	38.0 ± 22.2	7.2 ± 3.3	31.5 ± 2.0
misex3	-1.0 ± 2.0	2.0 ± 3.9	0.5 ± 1.4	3.6 ± 1.8	6.6 ± 9.9	3.6 ± 1.3
pdc	11.2 ± 1.9	-31.1 ± 3.8	12.6 ± 0.8	17.9 ± 1.1	-19.6 ± 15.4	16.7 ± 0.6
s298	-3.6 ± 5.8	0.0 ± 4.5	-7.5 ± 3.2	10.7 ± 2.7	18.2 ± 24.2	8.4 ± 3.4
s38417	100.7 ± 3.6	12.1 ± 4.3	115.6 ± 3.0	56.4 ± 2.4	21.5 ± 4.4	46.1 ± 2.5
s38584.1	64.9 ± 3.5	4.4 ± 4.4	73.8 ± 2.1	0.6 ± 4.8	-3.8 ± 3.4	3.4 ± 1.8
seq	1.6 ± 1.8	-19.4 ± 7.9	0.5 ± 1.6	6.9 ± 2.5	-19.1 ± 1.7	4.5 ± 1.9
spla	12.9 ± 2.0	10.0 ± 18.9	17.5 ± 1.9	19.0 ± 2.3	16.7 ± 21.4	22.2 ± 1.9
tseng	-10.8 ± 3.2	13.3 ± 4.2	2.7 ± 3.3	-0.3 ± 4.4	5.7 ± 4.2	3.9 ± 1.9
Worst Case:	100.7 ± 3.6	-31.1 ± 3.8	115.6 ± 3.0	56.4 ± 2.4	21.5 ± 4.4	77.3 ± 2.4
Absolute Average:	19.4 ± 3.4	7.4 ± 5.7	23.9 ± 2.4	14.4 ± 3.7	9.0 ± 7.6	17.4 ± 1.9

Table III. Percent Change in Place-and-Route Results with Semisynthetic Circuits

	Perturb, 5% cutout size			Perturb, 20% cutout size		
Name	CW %	CP %	WL %	CW %	CP %	WL %
alu4	4.2 ± 3.9	2.6 ± 7.4	-0.3 ± 1.4	4.5 ± 2.3	8.6 ± 9.9	0.8 ± 1.7
apex2	1.1 ± 2.5	4.2 ± 1.9	-0.3 ± 2.0	1.6 ± 2.2	4.4 ± 2.8	0.2 ± 1.7
apex4	0.0 ± 1.1	3.9 ± 4.4	-0.4 ± 1.3	1.5 ± 3.0	12.2 ± 22.5	1.1 ± 3.3
bigkey	10.1 ± 6.8	-1.9 ± 1.1	0.9 ± 2.0	12.5 ± 7.4	1.3 ± 3.1	0.1 ± 2.9
clma	-4.7 ± 1.2	-2.0 ± 2.7	-1.4 ± 1.2	6.2 ± 2.2	0.5 ± 2.4	5.9 ± 1.1
des	-11.4 ± 6.0	3.3 ± 2.6	-0.9 ± 1.7	-8.4 ± 5.9	1.3 ± 2.0	0.2 ± 2.6
diffeq	-1.8 ± 2.7	-2.1 ± 4.1	-6.1 ± 1.4	4.4 ± 5.2	-3.8 ± 3.2	0.0 ± 1.6
dsip	1.2 ± 5.4	1.1 ± 2.6	0.3 ± 3.0	3.6 ± 7.0	-0.1 ± 1.2	0.5 ± 1.3
elliptic	2.0 ± 3.4	-3.4 ± 2.7	0.6 ± 0.6	2.7 ± 1.4	7.9 ± 30.3	1.4 ± 1.2
ex1010	2.6 ± 3.9	1.0 ± 4.4	1.0 ± 2.4	18.8 ± 6.5	3.8 ± 5.7	14.0 ± 4.5
ex5p	0.5 ± 2.1	5.3 ± 6.9	0.7 ± 1.3	4.3 ± 1.7	14.1 ± 22.7	3.6 ± 1.1
frisc	3.0 ± 2.8	-1.4 ± 2.7	2.6 ± 2.1	3.7 ± 1.7	1.4 ± 4.4	3.6 ± 1.2
misex3	-0.6 ± 2.8	11.9 ± 15.0	-0.1 ± 2.4	3.9 ± 1.8	4.3 ± 7.1	3.5 ± 2.5
pdc	0.4 ± 1.3	-27.7 ± 5.9	0.2 ± 0.7	5.0 ± 1.9	-28.4 ± 4.5	3.4 ± 0.6
s298	5.4 ± 3.8	10.2 ± 4.2	4.4 ± 2.7	9.8 ± 3.2	7.4 ± 7.0	6.4 ± 2.7
s38417	-0.3 ± 1.6	1.7 ± 2.5	1.3 ± 1.2	21.3 ± 3.6	11.2 ± 8.3	14.8 ± 1.8
s38584.1	1.2 ± 2.8	1.5 ± 2.2	3.0 ± 1.5	10.5 ± 5.1	1.7 ± 2.7	7.7 ± 2.2
seq	0.3 ± 2.8	-21.6 ± 1.8	0.1 ± 1.1	4.2 ± 3.0	-19.1 ± 6.5	3.7 ± 2.2
spla	-0.6 ± 2.0	3.2 ± 4.3	1.8 ± 1.3	4.7 ± 1.2	39.2 ± 58.8	6.0 ± 0.7
tseng	0.6 ± 4.1	-0.2 ± 3.8	-1.2 ± 1.5	2.9 ± 5.0	4.0 ± 3.7	0.1 ± 3.4
Worst Case:	-11.4 ± 6.0	-27.7 ± 5.9	-6.1 ± 1.4	21.3 ± 3.6	39.2 ± 58.8	14.8 ± 1.8
Absolute Average:	2.6 ± 3.2	5.5 ± 4.2	1.4 ± 1.6	6.7 ± 3.6	8.7 ± 10.4	3.9 ± 2.0

to operate on the entire circuit ($S = N$ using the terminology from Section 3.1). The arithmetic average of results from ten different synthetic clones are given in Table II. All data is of the form *average percentage difference* ± *standard deviation percentage*, where both numbers are the percentage of the original MCNC result given in Table I. For example, in Table II, the first row (alu4) and the first column of data (CCirc+CGen, CW %) contains the data -0.9 ± 4.3. This means that the average channel width of the ten CCirc+CGen alu4 clones was 0.9% lower than the channel width of the original alu4 circuit. From Table I, the original channel width of alu4 was 33, so the average channel width observed is $33 - (33 * 0.009) = 32.7$. The standard deviation is reported as 4.3%, meaning that the actual standard deviation is $33 * 0.043 = 1.4$ tracks. We present the results as percentages to aid comparisons between the various circuits. At the bottom of the table, we summarize the worst-case row and the absolute average for all twenty rows in the table.

From the results, we see that Perturb produces clones which more closely reproduce the channel width and wirelength characteristics than does CCirc+CGen. Critical-path delay is also good: Although the average is slightly higher, the worst case is smaller for Perturb. The standard deviations in results for both schemes are similar, with values between 2% and 5% being common. In all but four cases, those circuits which are difficult for Perturb (results differ by $\geq$ 10%) are also difficult for CCirc+CGen. In contrast, there are thirteen cases which are difficult for CCirc+CGen but not difficult for Perturb. These results suggest that Perturb generates synthetic clones with excellent postrouting properties.

Although not shown, the results of using Perturb without ADC are much worse than the CCirc+CGen results. This tells us that it is possible to preserve many characteristics of the circuit, including wiring characteristics, but still end up with a circuit that does not behave like the original. Hence, circuit locality *must* be considered, but it is not properly captured by these metrics. It should be noted that the perturbations in Ghosh et al. [1998] do not consider locality at all.

The Perturb results in Table II use an ADC value of $d = 3$. The data in Table II (and Table III) was also generated for $d = 2$ and $d = 4$. A depth of 2 appeared to be too restrictive for finding candidate edges to swap, whereas a depth of 4 showed a significant step towards the results with no ADC for some circuits. Hence, we used $d = 3$ for all experiments.

The second experiment examines postrouting properties of incremental circuits. A subcircuit S is first identified to contain 5% or 20% of the LUT nodes in N. Although many schemes could be used to identify S, we use T-VPack with an extremely large cluster size and randomly select one of the clusters. This ensures that S is somewhat realistic: a fully connected group of LUTs and flip-flops that are selected without any regard for placed location or CLB boundaries. Then, S is perturbed and stitched back into the original circuit. The modified circuit is clustered and fully placed and routed with vpr. This helps assure us that new postrouting results are based upon the properties of the netlist and not on those of the original placement. We generated eight incremental semisynthetic clones, each time selecting a different random region for S.

The results in Table III show that Perturb is able to create new circuits that have very similar postrouting properties to the original. As expected, a 5% netlist change results in smaller changes to postrouting characteristics than a 20% change. Although not shown, the results for a 10% change are in between these results. Also as expected, the change in results for incremental circuits is smaller than that for fully synthetic circuits. This suggests the incremental changes are not too dramatic, and the magnitude of change roughly scales with the size of change.

In the end, we must also revisit the question as to whether Perturb sufficiently changes a circuit when producing a clone. This is a difficult question to address, but we offer the following insight. First, Perturb produces different clones with different random seed values; that is, the same clone is not being produced in all cases. Second, to ensure differentiation, Perturb always verifies that the required number of edges have changed from the original according to the perturbation factor (set to 25% for this work[3]). If insufficient edges have changed, it continues to run. Third, the technique generates clones that sometimes produce large changes to the average postrouting properties. This suggests that considerable disturbances to the original circuit can be made. Fourth, from the standard deviation results, the amount of variation within a family of clones roughly agrees with the amount of variation from CCirc+CGen. Also, the standard deviation is sometimes large and sometimes small, suggesting some variety across circuits.

It is also interesting to examine the runtime of Perturb. There are $3 * 8 * 20 = 480$ circuits which were generated, placed, and routed for Table III. On a Pentium 4 running at 3GHz with 1GB of RAM, the process of cutting out S from N, perturbing S into R, and stitching R into $N \backslash S$ was completed in approximately 8 minutes for all 480 circuits (approximately one circuit per second). This includes all I/O and spawning several Linux processes per clone.

5. MUTATING A CIRCUIT

Perturb provides a reliable starting point for generating synthetic circuits where key characteristics of S are exactly preserved; however, it may be desirable to alter some of these features in controlled ways. For example, an incremental user-design change to a circuit is likely to either increase or decrease the size of the circuit. To demonstrate how this can be done, we have added the Mutate preprocessing step to scale S before the Perturb step. By scaling *before* the circuit is altered with Perturb, we can be less concerned about the realism of the scaling mechanism.

There are two ways to scale a circuit: reduction and enlargement. Scaling is of particular interest in benchmarks for incremental place-and-route tools because the place-and-route tool must fill holes left by inserting a smaller R into $N \backslash S$ or must make room for a larger R. It is likely that a user change to a circuit will not be exactly the same size when the incremental place-and-route tools are called.

The Mutate tool performs the required scaling on S and produces T by first enlarging S beyond the required size and then reducing the circuit to achieve the required scaling. If a reduction in size is requested, the enlargement step is skipped, and if an enlargement of an exact integer factor is requested, the reduction step is skipped. T is then passed to Perturb, which generates R. The name "Mutate" arises due to the gross changes this tool makes to the input circuit, S.

[3]This means that 25% of the edges will be moved without another edge being swapped in where the original edge was located.

5.1 Circuit Reduction

To perform reduction, a "shotgun" approach is used. The required number of nodes to remove from the circuit is computed, and then nodes in S are randomly selected and deleted under the following restrictions.

- —If a selected node is the only source for another node, then both nodes must be deleted. This could lead to a cascade of removals to return the circuit to a valid state.
- —If a selected node is the only sink of another node, then both nodes must be deleted. Again, this could cascade.
- —The selected node must not be a primary input or a primary output.

Nodes (or chains of nodes) are removed until the required number of nodes have been successfully deleted. When complete, the circuit will be smaller without skewing many of the original characteristics.

5.2 Circuit Enlargement

Increasing the size of a circuit is a harder problem, complicated further by the need to preserve locality. Adding nodes using a shotgun approach, similar to the procedure used in circuit reduction for removing nodes, does not work well in highly connected circuits where there may be no nodes nearby with available inputs. The maximum number of inputs on a node is fixed for a particular FPGA architecture (in this research we have used 4 inputs), so we cannot arbitrarily add nodes wherever required. Hence, the shotgun approach would lead to "filling out" the circuit, where sparsely connected portions of the circuit would become highly connected, and highly connected portions would remain largely untouched. We felt this would not very well mimic a user change to a circuit.

Because S is nominally a relatively small part of N, we replicate S a number of times to achieve an enlargement. This also preserves the characteristics of S. These copies are placed in parallel and additional logic (LUTs) is added to properly multiplex the inputs and outputs of each copy together, creating T. This logic increases depth of the region by 2. It is possible to also place copies of S in series instead of in parallel, but this would have more dire consequences to maximum logic depth.

5.3 Mutate Results

To test the Mutate step, we tested the two components of the scaling operations (enlargement and reduction) separately. The 20 largest MCNC benchmarks were again used, and in each circuit a subcircuit containing 5%, 10%, and 20% of the nodes was scaled to 50%, 75%, 200%, and 400% of its original size. After scaling, the replacement circuit was sent to Perturb and stitched back in $N \backslash S$. Ten trials were performed at each size of S and each scaling factor. The average normalized postrouting results are shown for the 5% case in Figures 4, 5, and 6. In all three figures, the value 1.00 on the vertical axis represents the results without Mutate, taken from Table III.

Figure 4 shows the minimum routable channel width for a scaled 5% cutout size. It is difficult to predict the impact of scaling on channel width. On the one

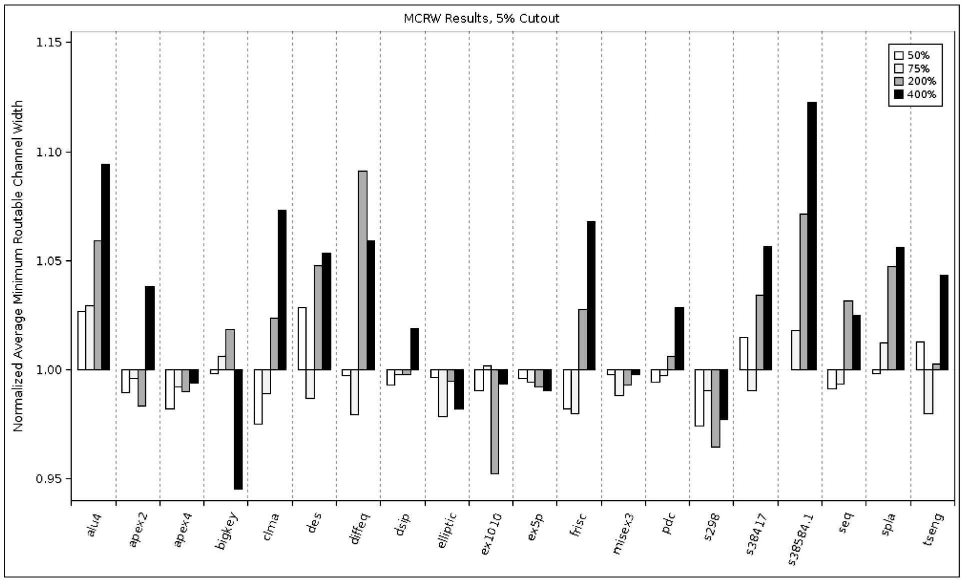


Fig. 4. Normalized channel width for a 5% cutout size.

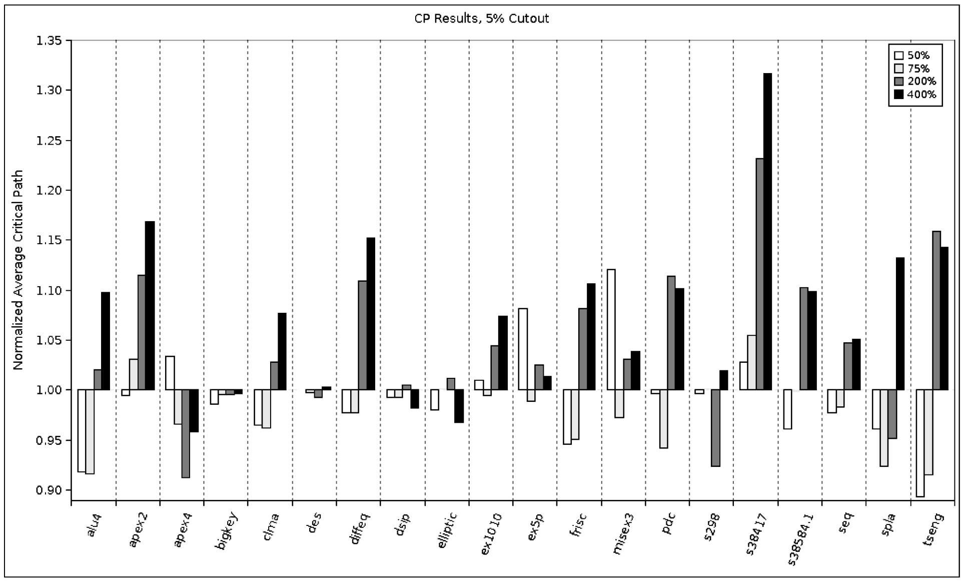


Fig. 5. Normalized critical-path delay for a 5% cutout size.

hand, we might expect that a size reduction in one region of a circuit will reduce the number of tracks required to route the circuit because the routing problem is being made easier. On the other hand, a size reduction may have the opposite effect by compacting the placement around an already congested region, causing further congestion. Similarly, an increase in circuit size may increase

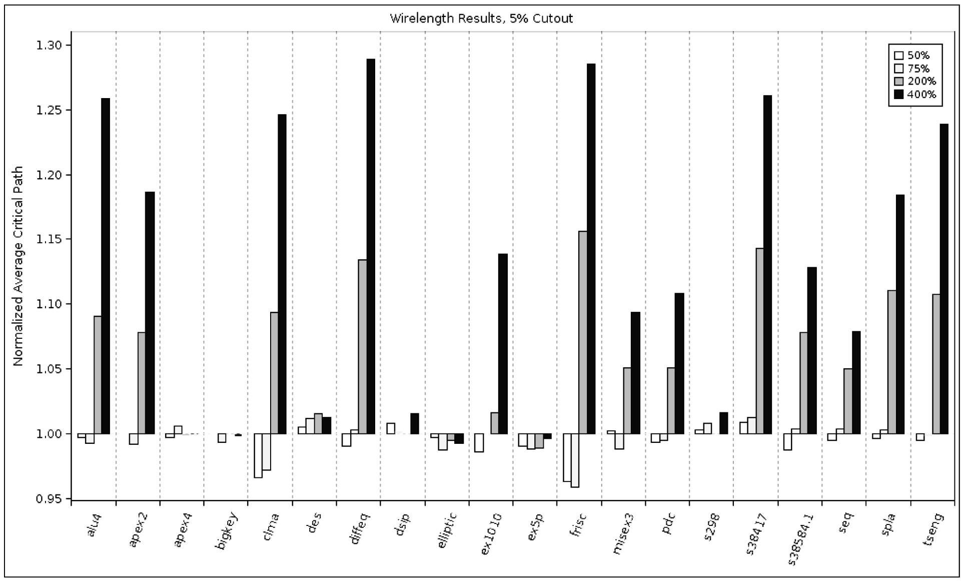


Fig. 6. Normalized wirelength for a 5% cutout size.

or decrease the minimum number of tracks required to route the circuit. In some large circuits (clma, pdc, and frisc), there is a positive correlation between channel width and size, but in others (s38584.1 and s38417) the trend is not apparent. For smaller circuits, the trend is less clear or not existent at all. Most importantly, however, Figure 4 shows that Mutate has increased the minimum channel width by at most 12% (s38584.1 at 400%).

Figure 5 shows the critical path for a scaled 5% cutout size. For most circuits, reducing the size of S results in a decrease in the critical path, which is expected. Removing logic from a circuit will decrease the critical path, either because the removed logic was part of the critical path or because the circuit is smaller after the reduction. For circuits where the critical path has increased, it is possible that Perturb has made the place-and-route problem more difficult, resulting in more “bad” moves.

For 200% and 400% increases, shown in Figure 5, there is an expected increase in the critical path. Since the amount of logic in the circuit has increased, finding a placement for the additional logic will naturally increase the critical path. Additionally, the enlargement process involves adding two additional logic levels to the circuit, which might also increase the critical path.

Despite the increases, most of the data falls within 15% of the original circuit, again showing that Perturb+Mutate is not significantly altering the postrouting characteristics of the circuit in unexpected ways.

Figure 6 shows the routed wirelength for a scaled 5% cutout size. The only feature that stands out on this graph is the noticeable increase in wirelength for most 200% and 400% scales. A 5% cutout region scaled to 400% would result in a 15% increase to the size of the circuit. Figure 6 shows that the wirelength for a 400% mutation increases by 10% to 25%, which is close to the expected amount. The postrouting wirelength results of Mutate show that it is again preserving the postrouting characteristics of the original circuit.

The results for mutations with 10% and 20% cutout sizes are not presented, but they show similar trends to the 5% case. However, the trends observed are more pronounced because a larger portion of the circuit is being mutated in each case. They also reinforce the observation that Perturb+Mutate does not change the postrouting characteristics of the circuit in unexpected ways.

6. PITFALLS

In this section, several unsuccessful methods of controlling locality, as evidenced by poor postrouting results, are presented. We include these for completeness and because they provided valuable insight into the behavior of Perturb. This insight led to the development of the ancestor-depth-control technique. Additionally, we describe some limitations of the Perturb+Mutate technique. In some of the situations presented next, we used information after placing the original circuit to capture locality.

6.1 Wirelength Control

In an effort to control the channel width and total routed wirelength in the circuit, we attempted to limit the wirelength during perturbation. The method uses the total Manhattan distance of all edges in the circuit to approximate the total wirelength in the circuit. During perturbation, if an edge swap lowers the total wirelength, the swap is accepted. If the wirelength increases, the move is probabilistically accepted, using an exponential function similar to that used in simulated annealing.

The location of each node in the circuit is taken from a placement of the original MCNC circuit. After the perturbation procedure is complete, the entire circuit is replaced and rerouted. When considering incremental place-and-route tools, it is reasonable to assume we have a previous placement of the circuit, so we can use that placement to drive this technique.

The results from experiments using this method showed very little difference over perturbation with no ancestor depth control. Hence, it was ineffective.

6.2 Bounding-Box Control

Instead of limiting the wirelength by using a Manhattan metric, the bounding box can also be used. The bounding box of a net is the half perimeter of the smallest box bounding all the fanouts of a net. Additionally, the bounding box would prevent random edge swaps over large distances. Consider Figure 3 where (10, 14) is again the selected edge. Suppose the (7, 11) edge is separated from the (10, 14) edge by a large distance; a bounding-box limitation would prevent these edges from being swapped.

Here, we again follow a procedure similar to that used in simulated annealing to always accept moves that lower the total cost, and probabilistically accept ones that do not. This technique marginally improved the results by only 1% to 2% compared to using no control at all. Hence, it was also ineffective.

6.3 Net-Swapping

Instead of swapping individual sinks from edges, a method of potentially preserving locality is to swap the entire net that an edge is in; in other words, this technique swaps the sources of nets instead of the sinks. This method, combined with the bounding box, aims to prevent nets from fanning out to all regions of the FPGA when it is routed.

This method further improved the results over the bounding-box control by an additional 1%. However, these results cannot be deemed significant, and are still far from the original postrouting results of the MCNC circuits. Only the ancestor depth control described in Section 4.2 was effective at capturing locality and preserving the channel width, critical-path delay, and total wirelength results of the original circuit.

6.4 Additional Limitations

The Perturb+Mutate approach does not properly handle two situations that may appear in practice. First, coarse-grain blocks are not considered at all. This makes it difficult to consider situations where coarse-grain blocks such as multipliers are changed to LUTs, or vice versa. Second, the use of design hierarchy may result in several replicated instances of one subcircuit; any change to this subcircuit will result in several parallel but seemingly unrelated (unconnected) changes throughout the netlist. In both of these situations, however, it is probably better to do a full recompile rather than an incremental one.

When verifying that a circuit has had sufficient modifications, Perturb is unable to consider logic equivalence or other symmetry. With sufficiently large and complex circuits, this should not present a problem. However, highly symmetric parity trees may not exhibit sufficient differentiation between the original and modified forms.

7. FUTURE WORK

Listed in the following are several directions for future research that can make Perturb even more useful in generating benchmarks.

7.1 Dynamic Ancestor Depth

For all experiments in this article, a static ADC value of $d = 3$ was used because it gave desirable results, without being overly restrictive in the choice of candidate edges to swap during perturbation. However, this value was chosen by observing postrouting results for different circuits under test; it is not ideal for all circuits. A dynamic method or heuristic to compute a good ancestor depth is the next logical step. For example, when building the list of candidate edges for swapping, an unbounded backwards search can be terminated when a "good" number of candidates is found, instead of using a fixed depth. Somehow, this method must still attempt to preserve locality.

Some further study is needed relating ADC to the Rent parameter.

7.2 Increasing the Critical Path

An additional useful feature for testing incremental place-and-route that no existing circuit generator possesses is the ability to controllably increase or decrease the critical path length through the circuit. Such a change to the circuit would force the incremental place-and-route tool to shuffle nodes along the critical path with minimal adjustment to the rest of the circuit, approximating an incremental improvement flow. A critical-path change is difficult to produce and test with real circuits, so a synthetic approach would be helpful in this area. In Section 5, circuit enlargement was done by duplicating S in parallel; instead, duplicating S in series may be a way to directly increase the critical path.

7.3 Breaking Correlations

Perturb identically preserves all node properties between both the original and modified circuits. To create a greater variety in the types of generated circuits, it may be useful to break the linkage between the fanin and fanout of each node, allowing high-fanout nets to inherit a different fanin distribution. Other correlations may similarly be broken.

7.4 Locality and Structure

Concerning locality and structure, Perturb does not understand structure that arises from bus and datapath connections in an array multiplier, for example. It isn't clear how to add such a discernment ability to Perturb but this would be useful to help preserve locality and routability of modified circuits.

8. CONCLUSIONS

In this article we have presented a simple new method for benchmark generation which is intended for testing incremental place-and-route tools. The perturbation and scaling methods are simple and effective and do not create combinational loops.

We have described a new technique that modifies a given circuit to generate semisynthetic clones of an original circuit. Perturb exactly preserves a number of key characteristic features of a circuit: the number of nodes, number of edges, fanout distribution, and depth profile. We found that this information does not sufficiently capture locality, so Perturb was extended to include ancestor depth control, thereby taking the locality of edge swaps into consideration. We have also presented Mutate, a tool that implements scaling the size of the circuit. In the future, these tools can be expanded to permit more circuit characteristics, such as the fanout distribution, to be controllably modified.

Two experiments were conducted with Perturb. The first validated the approach by comparing the postrouting results of Perturb operating on 100% of the circuit to the results of a synthetic circuit generator, CCirc+CGen. Results indicate that Perturb is effective at preserving postrouting characteristics of the original circuits. Although not directly shown in the article, we also witnessed the importance of ancestor depth control at controlling locality.

The second experiment used Perturb to modify a small portion of the MCNC benchmarks to create new incremental circuits. Like the full synthetic clones, these incremental circuits also exhibited postrouting properties very similar to the original. For the 5% cutout size, the channel width, critical path, and total wirelength were all within 5.5% of the original circuit, on average, with a standard deviation of no more than 4.2%.

A third set of experiments verified that Perturb+Mutate is able to add or remove logic to a circuit without significantly altering the postrouting characteristics in unexpected ways. In these experiments, Perturb+Mutate operated on 5% to 20% of the MCNC benchmarks and scaled the operating region to 50%, 75%, 200%, and 400% of the original size to create new circuits.

We believe these experiments have shown the perturbation and scaling techniques to be viable building blocks for generating incremental benchmark circuits. Finally, several directions for future work have been presented which would add additional functionality to the tools, allowing generation of benchmarks that test more sophisticated features of incremental place-and-route tools.

ACKNOWLEDGMENTS

Thanks to D. Leong for integrating the original Perturber in his incremental place-and-route toolflow, and generating many bug reports.

REFERENCES

Betz, V., Rose, J., and Marquardt, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic, Boston, MA.

Cong, J. and Sarrafzadeh, M. 2000. Incremental physical design. In *Proceedings of the International Symposium on Physical Design (ISPD)*. ACM Press, New York, 84–92.

Coudert, O., Cong, J., Malik, S., and Sarrafzadeh, M. 2000. Incremental CAD. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 236–243.

Darnauer, J. and Dai, W. 1996. A method for generating random circuits and its application to routability measurement. In *Proceedings of the 4th ACM/CIGDA Internation Symposium on FPGAs*. 66–72.

Ghosh, D., Kapur, N., Harlow, J. E., and Brglez, F. 1998. Synthesis of wiring signature-invariant equivalence class circuit mutants and applications to benchmarking. In *Proceedings of the Design Automation and Test in Europe (DATE)*, 663–671.

Grant, D., Chin, S., and Lemieux, G. 2006. Semi-Synthetic circuit generation using graph monomorphism for testing incremental placement and incremental routing tools. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 725–728.

Grant, D. and Lemieux, G. 2006. Perturber: Semi-Synthetic circuit generation using ancestor control for testing incremental place and route. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Bangkok, Thailand.

Hutton, M., Rose, J., and Corneil, D. 2002. Automatic generation of synthetic sequential benchmark circuits. *IEEE Trans. Comput.-Aided Des. 21,* 8, 928–940.

Hutton, M., Rose, J., Grossman, J. P., and Corneil, D. 1998. Characterization and parameterized generation of synthetic combinational circuits. *IEEE Trans. Comput.-Aided Des. 17,* 10, 985–996.

Kapur, N., Ghosh, D., and Brglez, F. 1997. Towards a new benchmarking paradigm in EDA: Analysis of equivalence class mutant circuit distributions. In *Proceedings of the International Symposium on Physical Design (ISPD)*, 136–143.

Kundarewich, P. and Rose, J. 2004. Synthetic circuit generation using clustering and iteration. *IEEE Trans. Comput.-Aided Des. 23,* 6, 869–887.

Leong, D. 2006. Incremental placement for FPGAs. M.S. thesis, Department of Electrical and Computer Engineering, University of British Columbia.

Marquardt, A., Betz, V., and Rose, J. 2000. Speed and area tradeoffs in cluster-based FPGA architectures. *IEEE Trans. Very Large 8,* 1, 84–93.

Pistorius, J., Legai, E., and Minoux, M. 1999. Generation of very large circuits to benchmark the partitioning of FPGAs. In *Proceedings of the International Symposium on Physical Design (ISPD)*. ACM Press, New York, 67–73.

Singh, D. P. and Brown, S. D. 2002a. Incremental placement for layout-driven optimizations on FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. ACM Press, New York, 752–759.

Singh, D. P. and Brown, S. D. 2002b. Integrated retiming and placement for field programmable gate arrays. In *Proceedings of the 10th International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM Press, New York, 67–76.

Singh, D. P., Manohararaja, V., and Brown, S. D. 2005. Incremental retiming for FPGA physical synthesis. In *Proceedings of the 42nd ACM IEEE Design Automation Conference (DAC)*. ACM Press, New York, 433–438.

Stroobandt, D., Verplaetse, P., and van Campenhout, J. 2000. Generating synthetic benchmark circuits for evaluating CAD tools. *IEEE Trans. Comput.-Aided Des. 19,* 9, 1011–1022.

Tom, M. and Lemieux, G. 2005. Logic block clustering of large designs for channel-width constrained FPGAs. In *Proceedings of the 42nd Annual Conference on Design Automation (DAC)*. ACM Press, New York, 726–731.

Verplaetse, P., van Campenhout, J., and Stroobandt, D. 2000. On synthetic benchmark generation methods. In *Proceedings of the IEEE International Symposium on Circuits and Systems, Volume IV 4*, 213–216.

Received December 2007; revised May 2008; accepted July 2008