Automatic Generation of Synthetic Sequential Benchmark Circuits

Michael D. Hutton, *Member, IEEE*, Jonathan S. Rose, *Member, IEEE*, and Derek G. Corneil

***Abstract*—The design of programmable logic architectures and supporting computer-aided design tools fundamentally requires both a good understanding of the combinatorial nature of netlist graphs and sufficient quantities of realistic examples to evaluate or benchmark the results. In this paper, the authors investigate these two issues. They introduce an abstract model for describing sequential circuits and a collection of statistical parameters for better understanding the nature of circuits. Based upon this model they introduce and formally define the signature of a circuit netlist and the signature equivalence of netlists. They give an algorithm (GEN) for generating sequential benchmark netlists, significantly expanding previous work (Hutton *et al.*, 1998) which generated purely combinational circuits. By comparing synthetic circuits to existing benchmarks and random graphs they show that GEN circuits are significantly more realistic than random graphs. The authors further illustrate the viabilty of the methodology by applying GEN to a case study comparing two partitioning algorithms.**

***Index Terms*—Benchmark, digital circuits, placement.**

I. Introduction

Most algorithms in computer-aided design (CAD) are heuristic and the only reasonable evaluation method is to use benchmark circuits. The design of programmable logic devices (PLDs) is similarly inexact and benchmarks are needed to evaluate competing routing and logic structures and to predict future wiring requirements. However, it is difficult to find such test circuits, particularly large ones.

Netlists are typically modeled as graphs, which can be characterized by their mathematical and topological properties. Furthermore, many algorithms have different behavior and quality for different sets of graphs and in fact some problems which are NP-hard on general graphs are easily solvable for specific graph subclasses. The goals of this paper are to understand the restrictions which are typical for the types of netlist graphs which occur in practice, characterize these mathematically and statistically, and generate new graphs which emulate the typical netlist in terms of their characteristics and behavior.

Manuscript received April 5, 2001; revised December 28, 2001. This work was supported in part by Hewlett Packard Corporation and the Actel Corporation. This paper was recommended by Associate Editor L. Stok.

M. D. Hutton is with the Department of Computer Science, University of Toronto, Ontario M5S 3G4, Canada, and the Altera Corporation, San Jose, CA 95134 USA (e-mail: mdhutton@cs.toronto.edu; mhutton@Altera.com).

J. S. Rose is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4 Canada (e-mail: jayar@eecg.toronto.edu).

D. G. Corneil is with the Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4 Canada (e-mail: ; dgc@cs.toronto.edu).

Publisher Item Identifier 10.1109/TCAD.2002.800456.

In earlier work [1], [2] we used this method to characterize and generate purely combinational netlists. Other previous work by Darnauer and Dai [3] generated a random graph subject to a partition tree following Rent's Rule [4], with the goal of studying routability. This method captures empirical notions of partition hierarchy which we do not attempt to address here but does not deal with the delay, fanout, and sequential correctness which we do model. Iwama and Hino [5] used random modifications of seed circuits to create test instances for logic synthesis.

In this paper, we extend the previous combinational characterization and generation efforts of [1] and [2] to the more difficult problem of sequential and hierarchical circuits. We use the approach illustrated in Fig. 1 to measure the quality of synthetic circuits. Given an industrial benchmark "seed" circuit, we use our software tools circ to extract its parameterization or *signature* and GEN to generate a *clone* circuit with the same signature. For comparison, we generate a random graph with the same number of nodes, edges, and I/Os, but otherwise unconstrained. We place and route all three circuits with an academic tool VPR [6] and with Altera Corporation's MaxPlus2 software [7]. By comparing physical postplacement and routing statistics we are able to show that our method generates circuits which are significantly more realistic than random graphs.

Though we use "cloning" as a validation method, we have used the MCNC circuits [8] to generate typical distributions of the input parameters to GEN. In fact, the only required parameter to GEN is the circuit size n; all other unspecified values can be determined from the default distributions. Users can also define parameters in terms of other values or using common statistical distributions using the specification language of GEN. For example, $\mathrm{unit\_delay} = \mathrm{gauss}(\log(n), 1)$ defines that the circuit's unit delay will be sampled from a normal distribution with mean $\log(n)$ and variance 1.

The structure of this paper is as follows. In Section II, we discuss combinational and sequential circuit characterization then formally define the signature of a circuit and signature-equivalence classes of circuits. Section III gives our generation algorithm. In Section IV, we perform the evaluation process just described. Section V presents an application of our methodology to comparing two partitioning algorithms. We conclude in Section VI and discuss planned future work.

The results here are an amalgamation of several preliminary papers. The sequential generation algorithm originally appeared in [9], the formalization of sequential signatures and the equivalence classes of clone circuits in [10], and the application to partitioning in [11].

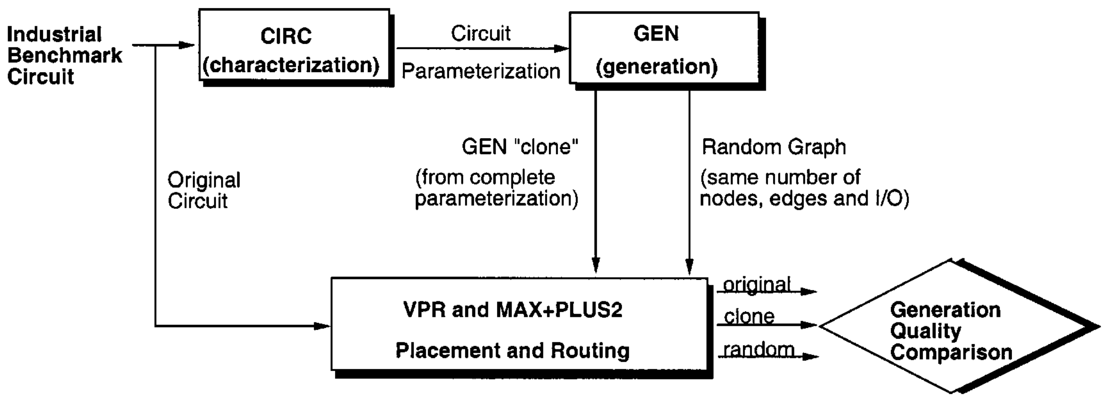

Industrial Benchmark Circuit
CIRC (characterization)
Circuit Parameterization
GEN (generation)
GEN "clone" (from complete parameterization)
Random Graph (same number of nodes, edges and I/O)
Original Circuit
VPR and MAX+PLUS2 Placement and Routing
original
clone
random
Generation Quality Comparison

Fig. 1. Measure circuit quality.

Since the publication of [2], [3], and [12] there has been large and growing interest in automatic benchmark generation, both in the realm of equivalence classes of circuits and in Rent-based approaches. Kapur *et al.* [13], Ghosh *et al.* [14], and Harlow and Brglez [15] addressed various methods for "mutating" seed circuit through local perturbations in order to effect a circuit with similar overall structure but differing local connectivity. A special session at the 1999 ISCAS conference was also dedicated to benchmark generation [10], [11], [16], [17]. Stroobandt*et al.* [18] extended the Rent-based approach significantly and applied the results to partitioning problems. Most recently, Pistorius *et al.* [19] classified circuits according to their functional type and generated and glued both random-logic circuits and parameterized modules such as memories, controllers, and multipliers together using a Rent's Rule hierarchy, again with high-density partitioning as a target application. This work used GEN as a subroutine for generating some logical subfunctions.

II. The Signature of a Circuit

Denote by n_{PI} and n_{PO} the number of primary inputs (PI) and outputs (PO) in netlist N, by n_{LOG} the number of logic nodes [four-input lookup tables (LUTs)] and n_{DFF} the number of flip–flops. Then n, the *size* of N, is $n_{\mathrm{PI}}+n_{\mathrm{DFF}}+n_{\mathrm{LOG}}$.[1] We consider only single-clock synchronous netlists with no bidirectional pins. For any node x, $fanin(x)$ is the number of edges entering x. Similarly, $fanout(x)$ is the number of edges leaving x and $max\_fanout(N)$ is $\mathrm{MAX}_{x\in N}\{\mathrm{fanout(x)}\}$. We assume that $\mathrm{fanin}(x)$ is always bounded by some constant k (typically 4), but that $\mathrm{max\_fanout}(N)$ is bounded only by n. Defining $fanouts[i]$, $\mathrm{i}=0\ldots\mathrm{max\_fanout}$, as the number of nodes in N with fanout i, we have the *fanout distribution* of N. The number of edges n_{edges} in N is the sum, over all x in N, of $\mathrm{fanin}(x)$ (equivalently the sum of $\mathrm{fanout}(x)$).

The remaining parameters defined on netlist N are related to combinational delay. Because delay can sometimes have subtle distinctions, we define these more formally.

Definition 2.1: Define the *unit delay* of a node x: If x is a PI or DFF, then $\mathrm{delay}(x)$ is zero. Otherwise, $\mathrm{delay}(x)$ is $1+$ the maximum unit delay of any fanin of x. Define $\mathrm{delay}(N)$ as the maximum, over all nodes x in N, of $\mathrm{delay}(x)$.

Definition 2.2: Define the *sequential level* of a node x. If x is a PI, then $\mathrm{level}(x)$ is zero. If x is a DFF, then $\mathrm{level}(x)$ is $1+$ the level of the D-input of x. Otherwise, $\mathrm{level}(x)$ is the minimum level over all fanins of x. Define $\mathrm{levels}(N)$ as $1+$ the maximum, over all nodes x in N, of $\mathrm{level}(x)$.

Definition 2.3: A netlist N is *combinational* if it contains no DFF nodes and *sequential* otherwise. If N is combinational it must have exactly one level, and all nodes x satisfy $\mathrm{level}(x)=0$. Otherwise, N has at least two levels and at least one node at each level.

Under the restrictions mentioned previously (no combinational cycles or bidirectional I/Os and a single global clock), both $\mathrm{level}(x)$ and $\mathrm{delay}(x)$ are well defined.

Definition 2.4: The *shape function* of a combinational netlist N is defined as an integer vector $\mathrm{shape}[d]$, $d=0\ldots\mathrm{dealy}(N)$, where $\mathrm{shape}[d]$ is the number of nodes in N which have delay d.

Definition 2.5: Given a directed edge $e=(x,y)$ in a netlist N, define $length(e)=\mathrm{delay}(y)-\mathrm{delay}(x)$. If $\mathrm{level}(y)<\mathrm{level}(x)$, then e is a *back edge*. If $\mathrm{level}(y)=\mathrm{level}(x)$ then $\mathrm{delay}(y)>\mathrm{delay}(x)$ and e is a *forward edge*. Otherwise, e is an *FF edge*, and we must have $\mathrm{delay}(y)=0$, $\mathrm{level}(y)=\mathrm{level}(x)+1$, and x is a DFF node. There are no other cases possible under the definitions of delay and level.

Definition 2.6: The edges function of a netlist N is defined as an integer vector $\mathrm{edges}[d]$, $d=0\ldots\mathrm{delay}(N)$, where $\mathrm{edges}[d]$ is the number of edges in N of length d.

We can now outline a mechanism to decompose or partition a netlist into two or more parts. Given N and a bipartition X and Y of the nodes of N, create two graphs N_x and N_y induced by the partition. For every edge $e=(x,y)$ where x is in N_x and y is in N_y, create a new primary input x_0 in N_y for x and a new primary output y_0 in N_x for y (and similarly for edges from Y to X). The netlist graphs X and Y are now disjoint, yet by identifying or gluing the appropriate nodes x, x_0 and y, y_0 together we can recreate N.

Definition 2.7: Under the decomposition of Definition 2.6, for an edge $e=(x,y)$, denote x_0 as a *ghost input* (GI) in N_y and y_0 as a *ghost output* (GO) in N_x. Define $\mathrm{delay}(x_0)$ to be that of $\mathrm{delay}(x)$ and $\mathrm{delay}(y_0)$ to be that of $\mathrm{delay}(y)$, supplementing the previous definitions with that of the new node-types GI and GO. Along with primary output nodes (PO), we can infer new

[1]For various implementation details we equate the number of nodes n with the number of nets, so primary output nodes (which have no fanout) are not counted in n. However, this choice is purely accounting.

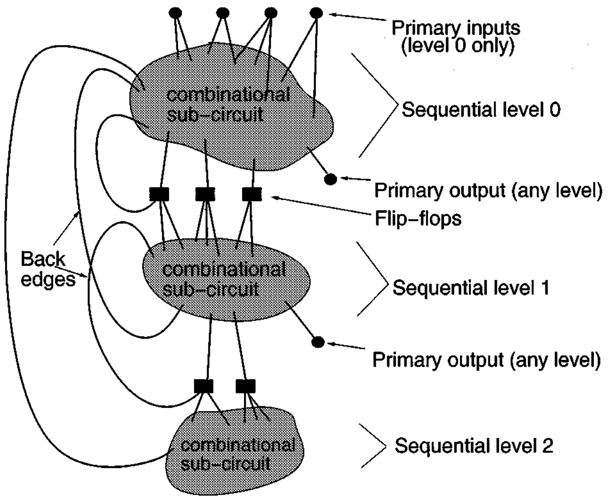

Primary inputs (level 0 only)
combinational sub-circuit
Sequential level 0
Primary output (any level)
Flip-flops
Back edges
combinational sub-circuit
Sequential level 1
Primary output (any level)
combinational sub-circuit
Sequential level 2

Fig. 2. Abstract model of a three-level sequential circuit.

shape functions $\mathrm{POshape}[d]$, $\mathrm{GOshape}[d]$ and $\mathrm{GIishape}[d]$ as we did for the delay-based shape function on the appropriate subset of nodes.

Definition 2.8: For netlist N and each level i, define the *level-netlist* N_i to be the subgraph induced by the set of nodes in N which are at level i and the edges between them. The partition of N into sequential netlists is its *sequential decomposition*. A set of netlists N_i is *compatible* if there exists an identification or gluing of ghost inputs and ghost outputs such that the sequential and combinational delay of all nodes in the combined circuit are maintained, and we define the resulting circuit the *sequential composition* of its subcircuits.

Fig. 2 illustrates sequential decomposition into three levels. It is important to point out that, though this model could appear to apply only to certain types of circuits which have a pipelined appearance, it does not actually preclude other views of sequential connections. Rather, we just *define* sequential levels in this way. The process of sequential decomposition and composition is fundamental to both the characterization and the later generation of sequential circuits.

A. Signature-Equivalence Classes of Clone Circuits

Definition 2.9: The *signature* of a level-netlist N_i is composed of i, n, n_{PI}, n_{PO}, n_{LOG}, n_{DFF}, n_{PO}, n_{GI}, n_{GO}, $\mathrm{delay}(N_i)$, $\mathrm{max\_fanout}(N_i)$, shape[ ], edges[ ], fanouts[ ], POshape[ ], GOshape[ ], and GIshape[ ]. (A purely combinational end-circuit would be the same with n_{GI} and n_{GO} set to zero.) The signature of a sequential netlist N is defined by the collective signatures of its sequential decomposition. For an exact specification the scalar parameters are redundant given the vector parameters in the signature but are part of the signature for clarity.

Given the concept of a signature of a netlist, we can now formally define equivalence classes of netlists.

Definition 2.10: Two netlists are *equivalent* if they have the same signature. Given the set of all netlists of any size, we can then induce a mathematical equivalence class to properly partition all netlists into equivalence classes under signatures.

Definition 2.11: Given a set of circuits generated to have the same signature as a given input circuit, we refer to the original circuit as the *seed* circuit and the other members of the equivalence class as *clone* circuits.

TABLE I
SEQUENTIAL CIRCUIT CHARACTERISTICS FOR SELECTED MCNC CIRCUITS: DESIGN NAME, NODES, IOs, DFFs, EDGES, BACK EDGES, AND SEQUENTIAL SHAPE

Name	N	IO	FF	E	BE	Seq. Shape
s838	167	37	32	556	256	169 65
s953	214	39	29	739	184	191 65 3
styr	238	19	5	814	219	207 45
planet	266	26	6	910	300	169 110
sbc	372	96	27	1273	300	388 51
mm30a	467	63	90	1697	235	500 90
dsip	1362	425	224	5440	896	1590 224
s298	1930	9	8	6944	2218	1636 305
bigkey	1699	425	224	6108	1344	1591 560
clma	8361	127	31	30114	5596	5810 2640 3

B. Circuit Characteristics and Empirical Distributions

As part of this work, we used the MCNC circuits [8] to empirially determine the distributions of parameters in the circuit signature (and some parameters not in the signature) [12]. The distributions of combinational shape and fanout were discussed in [1] and both combinational and sequential parameterization was covered in detail in [12]. This empirical description of the physical attributes is both interesting in its own right and also forms the basis of probability distributions used to complete a partial signature given as input to the circuit generator.

The division of a circuit into its combinational subcircuits introduces the concepts of *sequential shape*, the number of nodes in each successive sequential level. Table I shows sequential shape, along with scalar parameters of a selection of MCNC benchmark circuits.

We can make a number of observations with respect to the complete dataset.

The number of I/Os, which we modeled with a Rent-like parameterization for purely combinational circuits [1], [12] no longer holds for sequential circuits. Rather, there is no statistical correlation between n and n_{IO}. For the default profile we thus use a uniform distribution (between 2 and $n/4$) to select n_{IO} if it is unspecified.

The number of sequential levels is a small value (typically 1 to 3). Recall that a circuit with one sequential level is a combinational circuit. Of 78 sequential MCNC circuits, 69 have two sequential levels, 6 have three levels, and there is one circuit each of 4, 7, and 8 sequential levels. In all cases we saw, the majority of the combinational logic lies in the primary (zeroth) sequential level. We typically see successive sequential levels of logic having less than half the logic of the preceding level.

The number of flip–flops in a circuit also has low correlation to the amount of logic in the circuit. This is understandable, given the designer's flexibility in trading off logic and registers (e.g., 1-hot versus encoded state machines). We use a Gaussian distribution around a constant-deflated square root of the number of nodes as an approximation. Note that rather than an arbitrary function, this roughly models Rent's Rule for the number of flip–flops as the number of I/Os in a combinational circuit.

The number of back edges empirically varies between one and two times the number of nodes at the first sequential level, and we model it as such. The details of the empirical distributions, along with data for all circuits, is available in [12].

C. Extensions to the Sequential Model

With ghost input and output ports now defined, it is worth pointing out that the sequential model can be recursively generalized to describe arbitrary degrees of hierarchy, rather than just the flat interface between multiple levels in a simple sequential circuit.

For example, we can define also a purely combinational circuit as a hierarchy of combinational subcircuits simply by combinational specifications and a compatible GI and GO interface (without requiring that the circuit have flip–flop or back edges). In combination with a partitioner this would allow us to form a partition tree model of an input circuit (i.e., matching a given Rent parameter).

It would also be interesting to use this mechanism to describe an interface to other forms of circuits (e.g., memory [20]) or to deal with circuits at the block diagram level. To some degree these extensions have already been made by [19] since the publication of [2] and [9].

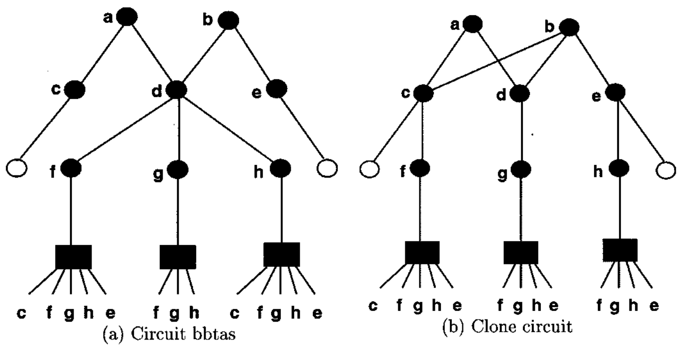
The ability to generalize the use of ghost inputs in generation and outputs would open the door to a more general hierarchical generation process. In this paper, however, we will restrict ourselves to simple sequential circuits.

III. The Algorithm for Sequential Generation

In [1], we gave a complete algorithm for generating purely combinational circuits. The input to the generation algorithm was the rough equivalent of a combinational circuit signature (no ghost inputs or outputs). In this section, we extend this algorithm to the more involved case of sequential circuits. The algorithm is presented in two parts: the generation of combinational subcircuits (with the additions of ghost inputs and outputs) and the gluing algorithm for sequential circuits.

Since one of our primary applications is to generate good circuits for FPGA research, our netlist will consist of four LUTs and DFFs.

As a preview to upcoming sections, Fig. 3 shows a small sequential circuit bbtas (left), its signature extracted by CIRC in the GEN langugage format, and a clone circuit (right) produced by GEN with that script. For readability, labels are used instead of back edges.



```
X = { name="bbtasclone";
  L0 = (@.comb_circ) { name="L0"; n=8; kin=4; nPI=2;
    nDFF=0; level=0; delay=2; nBot=3; shape=(2,3,3);
    nGI=13; GIshape=(4,9,0); nGO=3; GOshape=(0,0,3);
    nPO=2; POshape=(0,2,0); nEdges=7; edges=(0,7,0);
    outs=(5,0,2,1); max_out=3; nZeros=5;
  };
  L1 = (@.comb_circ) { name="L1"; n=3; kin=4; nPI=0;
    nDFF=3; level=1; delay=0; nBot=3; shape=(3);
    nGI=0; GIshape=(0); nGO=13; GOshape=(13); nPO=0;
    POshape=(0); nEdges=0; edges=(0); outs=(3);
    max_out=0; nZeros=3;
  };
  glue=(L0, L1);
};
output(circuit(X));
```

Fig. 3. The MCNC circuit bbtas, a clone produced by GEN script, and the GEN script used to generate the clone.

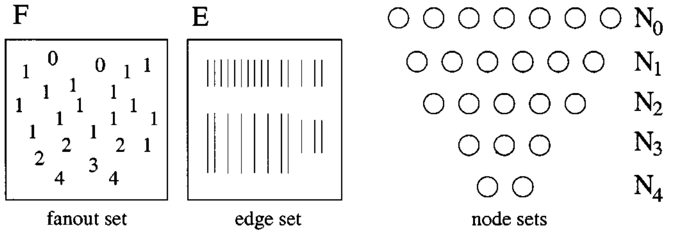


Fig. 4. Combinational circuit generation problem.

A. Generating Combinational Subcircuits

Given the combinational signature for a level netlist, we need to generate a graph (netlist) on n nodes and n_{edges} edges, such that each node x is assigned one fanout value from the set represented by the *fanouts*, that assigned value corresponds to the actual fanout of x in the graph, combinational delay is well defined for all nodes (i.e., $d(y) < d(x)$ for all fanins y of x, and at least one fanin y_0 has $d(y_0) = d(x) - 1$), fanin is bounded by k for all nodes, and all fanins to x are distinct (i.e., any signal enters a logic node at most once).

The parameterization defines a set of disconnected nodes at each combinational delay level and sets of unassigned edges and fanouts, as shown in Fig. 4. The goal of the algorithm is to complete the specific assignment of edges to nodes.

The progress of the algorithm after each step is illustrated in Fig. 5.

Step 1. Compute Boundaries on Level In- and Out-Degree: We initially consider all nodes on the same level as collapsed to a single *level node.* The goal of this step is to compute vectors $\text{min\_in}[i]$, $\text{max\_in}[i]$, $\text{min\_out}[i]$, and $\text{max\_out}[i]$ for bounds on the fanin (in-degree) and fanout (out-degree) of delay level i in the circuit.

Because GI and GO are special cases for fanout, we need their locations before fanout assignment. Experiments on industrial designs show that about 90% of the LUTs which feed a flip–flop in real circuits have no other outputs so we want to, wherever possible, assign fanout values of zero to nodes which will have a single ghost output destined for an FF edge. To accomplish this goal, we identify the delay location of the n_{latch} ghost outputs which will eventually feed a flip–flop in Step 1 of the original algorithm. This allows us to take them into account during the degree allocation phase. The result of this calculation is to make

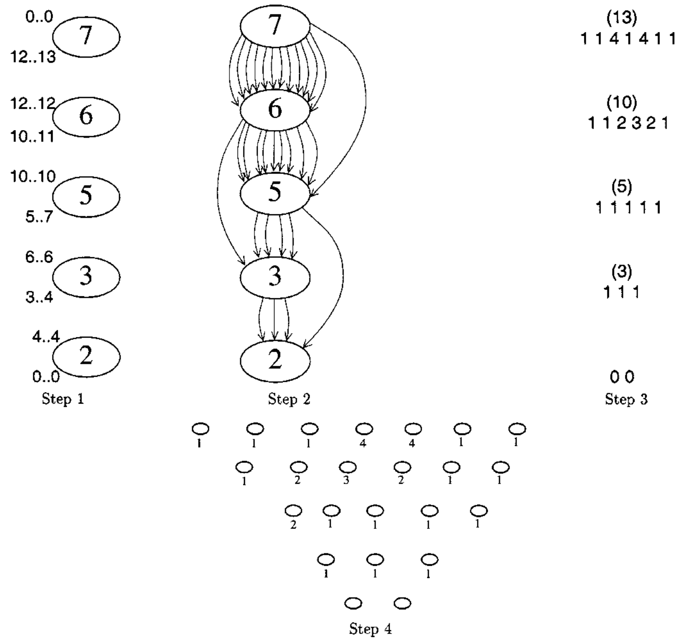


Fig. 5. Example at the conclusion of steps 1–4.

a new vector $\mathrm{latch\_shape}[i]$, $i = 0 \ldots d$ available to the degree calculations of Step 1.

The fanout assignment is iterative: we begin with rough bounds on the possible level fanin and fanout (i.e., each non-GI, non-PI final node will need between 2 and k fanins and at least one fanout if it is not a PO or GO) and refine these using a number of heuristic rules. For example, the out-degree at level j is bounded from above by the sum of the current maximum in-degrees at succeeding levels less the sum of the minimum in-degree at succeeding levels, and from below by the number of nodes at level j less the number of primary outputs at level j.

To accommodate ghost inputs and outputs, we must take the following points into account.

1) We assume that $\mathrm{latch\_shape}[i]$ nodes at level i will have a minimum fanout of zero, rather than one (as per the above discussion).
2) We allow (but do not require) $\mathrm{shape}[i]$–$\mathrm{GIshape}[i]$ nodes at level i to have minimum fanin one rather than two. Note that we must still allocate at least one nonghost fanin for each node or it would not (by definition) be in this subcircuit.
3) We subtract $\mathrm{GIshape}[i]$ nodes from the maximum fanin of level i to leave room for the incoming back edges.

Step 2. Assign Edges Between Levels: Given the boundaries from the previous step, we now assign (between all long levels) all long (nonunit) edges and enough unit edges to meet the minimum in and out degree boundary. We first dispose deterministically of the special cases of GI (no fanin) and GO (no fanout) nodes then similarly assign the required number of "critical" unit edges at the first and last delay level and enough unit edges between other levels to allow combinational delay to be well defined, e.g., each node at delay 5 needs at least one input at delay exactly 4. Then we probabilistically assign all long edges based on the available fanin and fanout at levels which are the appropriate length: draw one value from the distribution and assign it, then update the distribution accordingly and repeat. Finally, we deterministically assign enough of the remaining unit-edges to guarantee each level's minimum in-degree.

Step 3. Partition the Total Fanout at Each Level: Given the total out-degree of a level i, we now need to divide it into N_i node out-degrees taken from the fanout set F. To do this, we first calculate target (predicted) fanouts for each level, taking into account the remaining unassigned unit edges. At each level i, $\mathrm{latch\_shape}[i]$ and $+\mathrm{POShape}[i]$ are used to determine the number of fanout-0 nodes required. Fanout assignment for the remaining nodes after the removal of special cases is solved as an approximate integer-partitioning algorithm (solving the exact problem is NP-complete).

Step 4. Split Levels Into Nodes: We need to split each level-node N_i into its n_i individual nodes. This is a trivial process, except for the need to introduce *locality* into the final structure. To introduce the types of local structure which is empirically found in human-created netlists, we impose a list ordering on the nodes at each level $(1 \ldots n_i)$ and allocate the fanin values assigned to that level probabilistically so as to spread out the high-fanout nodes across the ordering. In the edge-connection stage to follow, we will use the ordering to determine the utility of connecting two nodes with an edge using their relative orderings as a metric of locality.

This step must now formalize the assignment of GI and GO designation to individual nodes. Previous steps have tried to "make room" for the ghost I/Os, so here the allocation is relatively straightforward: we allocate the $\mathrm{GIshape}[i]$ ghost inputs randomly and uniformly to the nodes at delay level i. Looking at the data for real circuits, we find that there is no statistical reason to do otherwise.

We also designate $\mathrm{latch\_shape}[i]$ nodes as *latched*. These nodes will eventually be candidates for gluing to a flip–flop. As much as possible, these will be fanout-0 nodes and will not be assigned additional GOs. If there are remaining fanout-0 nodes after this step, we assign additional GOs. All remaining GOs are kept for a new postprocessing step discussed next.

Step 5. Connect Edges Between Nodes: From the preceding steps we have a set of edge sources and destinations assigned to each level, and within the level we know the fanout of each individual node. The goal here is to assign each edge source and destination to specific nodes. The first pass connects unit edges to guarantee that the combinational delay of every node is well defined. Then the algorithm proceeds probabilistically. For each sampled node x with available fanin, we randomly choose L (the locality parameter) different possible fanout edges from the preceding levels which could attach to it and connect the one with the closest index to that of x. This process continues until all edges have been connected at both ends. The locality parameter L is an important tuning parameter of the generation algorithm. By forcing cells with similar indexes to have a higher connection probability we induce an element of structure to the design which would not otherwise be present.

Step 6. Remaining GO Assignment: Sequential subcircuits usually have fewer available edges than fully combinational circuits, so we use the ghost outputs, in part, to "repair" any extra zero-fanout nodes which may exist (usually some, but a small proportion) on the delay level they are assigned to. The

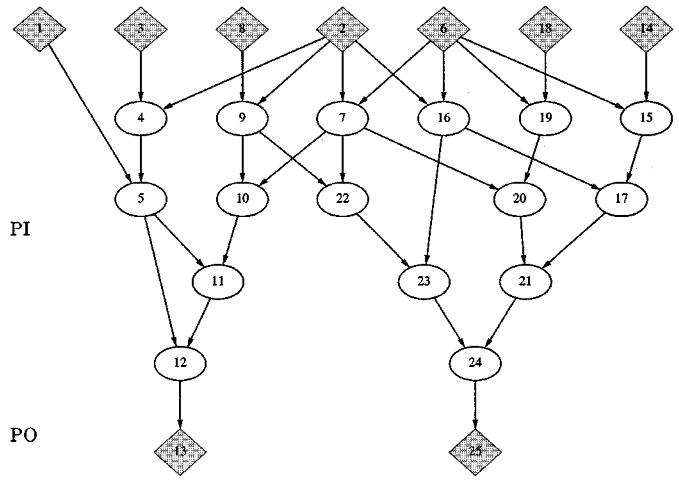


Fig. 6. Final result of the combinational algorithm.

remaining ghost outputs are not assigned uniformly. We want to generate more realistic circuits which tend to have a smaller number of high-fanout nodes to previous levels, rather than many nodes with a single ghost output. To do this, we choose a random subset of the nodes on each delay level requiring ghost outputs, smaller than the number of ghost outputs available, then assign the ghost outputs uniformly to nodes in the subset.

The overall algorithm yields a circuit as shown in Fig. 6—a combinational circuit with the correct number of GI and GO at the required combinational delay levels. In Section III-B, we will show the sequential composition or "gluing" process which operates on these subnetlists.

Note that the above algorithm assumes that an exact and complete signature is available. More typically, the user will specify only a few of the scalar parameters (either exactly or in relation to other parameters), and the front-end to GEN will create the remaining parameters from the *default scripts* also mentioned earlier. For example, the number of I/Os to a circuit can be defined as a random variable drawn from a gaussian distribution around the square-root of the number of lookup-tables, emulating Rent's Rule with "$r = 0.5$" and overriding the default distribution.

B. Gluing Subcircuits

The problem of joining subcircuits together into the final sequential circuit N is essentially one of appropriately matching the ghost ports between the subcircuits into back-edges and FF-edges.

When gluing begins, we have a list of subcircuits N_i, $i = 1 \ldots c$ to be connected, sorted by increasing sequential level. Each subcircuit contains a list *GI_list* of ghost inputs, a list *FF_outlist* of ghost outputs which have been labeled as targeting a flip–flop (from n_{latch} in the specification), a list *GI_list* of other ghost outputs intended for back edges, and a list *FF_inlist* of primary inputs in subcircuits at nonzero sequential levels which will become flip–flops. Each ghost input and output is attached to a node in the subcircuit and inherits the combinational delay of that node.

We have previously discussed the locality metric in making combinational connections between nodes in Step 5. For sequential gluing, define the *index* of a node as an integer proportional to the node's location in the node list for a given delay level in any subcircuit (the $0 \ldots n_i - 1$ ordering of the n_i nodes in delay level i, scaled to the maximum width over all combinational levels). When edges are connected in Step 5 of the base algorithm, we probabilistically favor connections between nodes which have closer indexes, in order to introduce clustering in the circuit. This form of geometric clustering is evident when viewing pictures of circuits generated by heuristic graph-drawing packages such as DOT [21].

In order to generate realistic circuits it is important to continue this process when connecting nodes to flip–flops and back edges, or we generate circuits with many crossing edges which are overly difficult to place and route. Thus we continue to use the node index for sequential gluing.

The matching is constrained by combinational delay and sequential levels. We cannot join a node x at sequential level l to a node y at level $l+1$, unless y is a PI (i.e., intended to become a flip–flop). We also cannot join a node x to *any* node y at a level beyond $l+1$ without violating the definition of sequential level on the nodes of N. Similarly, we cannot join a ghost output on a node x to a ghost input on a node y if $d(x) \geq d(y)$, without violating the combinational delay of y, and we cannot connect two ghost outputs attached to x with two ghost inputs to y, or we create a duplicate fanin to y.

This problem can be solved as a standard weighted bipartite matching problem (weights arise from locality). However, the $O(n^2 \log n)$ time [22] for weighted matching is too expensive for the size of netlists we need, so we use a heuristic greedy version instead. The most important aspect of the operation is to properly order the connections so as to increase the chances of finding a good solution. A solution which fails to connect all possible edges will result in GEN later having to diverge from its input specification by creating extra flip–flops or by moving ghost inputs or outputs to different nodes.

Because registered ghost outputs are labeled separately from the other ghost outputs, the problems of gluing back edges and gluing FF edges are independent. However, different subcircuits do "compete" for back edges. We give priority to earlier sequential levels by processing in the following order (justified in later detailed discussion):

```
for $i = 0 \ldots c$/* $c$ is the number of
     subcircuits */
  connect back-edges from $N_j$, $j \neq i$,
      to GI's of $N_i$.
  connect FF-edges from registered GO
      nodes in $N_i$ to PI's in $N_{i+1}$
end for
```

The greedy algorithm for gluing edges is described in Section III-B-I.

1) Gluing Back Edges: The greedy algorithm for gluing back edges to the ghost inputs of one circuit N_i from all other subcircuits is as follows.

First create a destination list of all ghost inputs in N_i and a source list of all ghost outputs in the other subcircuits which are at later sequential levels. Sort both lists by increasing *index* within decreasing *delay*. The purpose of this order is to use up the highest delay ghost outputs first (because they are less likely to find a matching ghost input and then require a flip–flop or movement later) and to match them to the highest delay ghost inputs with which they are compatible. Given that, we want to match indexes as well as possible.

Now proceed through the source list in order. Define the *match value* of a source node x with a destination node y as ∞ if (x, y) is an invalid edge (by the constraints above), and $d(y) - d(x)$, otherwise. We search the destination list for the first node with lowest match value, which also lines up a compatible index by the sorting. Note that we do not actually have to look at the entire destination list; this can be done in $O(d)$ time, using a few additional pointers indexed into the destination list. Combinational delay d is essentially a constant so the algorithm is fast.

The time required for this gluing phase is dominated by the sorting, so we need $O(n \log n)$ time[2] per subcircuit, of which there is a constant number. Note that "n" in this algorithmic complexity refers to the number of back edges in N, which is typically about 5%–10% of the size of the whole circuit.[3]

The reason that the main algorithm processes subcircuits in order of their sequential level is that the earlier levels typically have both many more nodes and greater combinational delay and also a more complex overall structure. (Later levels often reduce to a register-file with only a couple of logic nodes.)

2) Gluing Edges to Flip–Flops: The process for gluing nodes with ghost outputs labeled as latches to primary inputs at the next sequential level is more straightforward. For each adjacent pair of levels, create a source and destination list as before, sort the lists by index (independent of delay), and line up nodes directly (the lists are the same size, by the original specification of the subcircuits). This is an additive factor of $O(n \log n)$ time to the preceding steps, so the entire gluing algorithm remains $O(n \log n)$ time. (In this case, n refers to the number of flip–flops in the circuit which is, in practice, not the entire size of the circuit.)

Note that the order in which subcircuits are considered is unimportant, as the connections are independent.

C. Variance and Adherance to Specification

Because the algorithm for generation consists of a number of heuristics, we can never guarantee that we are able to completely match an input specification. In general, we find that we can miss a specification by a couple percent: In assigning 1000 edges to a given edge distribution, it is normal to have 10 to 20 edges be forced to the incorrect length from their exact specification. Similarly, it is normal to expect a small variation in delay-shape distribution. None of these is particularly significant given that our goal is to generate new and different benchmarks anyway.

A more significant issue is incompatibility between ghost input and output shape. This problem is harder to get right, so rather than forcing incorrect connections the basic algorithm is allowed to leave some ghost ports unconnected, and we add an additional postprocessing step to resolve the leftover ports. In this step, ghost inputs and outputs are moved to suitable candidates elsewhere in the subcircuits until matches are found. In extreme cases (flagged by warnings from GEN) up to 40% of back edges can be unresolved before postprocessing, but typically only 0%–5% of ghost inputs and outputs (which comprise less than 1% of all edges) remain after the main gluing algorithm. We note that since the underlying problem is NP-complete, to expect otherwise in polynomial time is unrealistic.

D. Software Tools: CIRC and GEN

The algorithms just described, and the source code for them is available under to the public domain [31]. To date, over 50 different academic and industrial users including Altera, Xilinx, Actel, and HP have obtained prototype versions of CIRC and GEN under academic license.

GEN is able to produce circuits in several different gate-level netlist formats, including Berkeley BLIF, Actel ADL, Altera AHDL, Xilinx XNF, and gate-level Verilog. Thus the tools are of immediate practical value to the community.

IV. Validating the Quality of GEN Circuits

As mentioned in Section I, we will test the viability of sequential GEN circuits in a number of different ways.

Our first evaluation relies primarily on generating clone circuits for known benchmarks and comparing their behavior to gen clones and to random graphs of the same size. The goals are to determine the relative viability of GEN circuits with respect to both random graphs and to existing benchmarks. We do this using two different tools, a high-quality academic placement tool VPR and a commercial FPGA place and route tool MAX+PLUS2 from Altera Corp. It is important to point out that the use of GEN circuits is not restricted to clone generation; this is simply a method of evaluation.

The second operation is to evaluate the ability of the tool to meet a given specification and to analyze the variance between clone circuits in the same equivalence class. For this operation, we chose a smaller subset of the design set comprising eight circuits. For each of these eight circuits we generate 100 clones using different random seeds and compare the resulting clones for their variation from the seed circuit and each other.

Before dicsussing these results we need to describe how we generate the random graphs used for comparison.

A. Generating Random Graphs

We generate a random directed graph on n nodes and n_e edges with n_{PI} primary inputs, n_{PO} primary outputs, with n_{DFF} available flip–flops (for breaking combinational cycles, as we want only synchronous designs) and $k_{\max}$-bounded fanin. The algorithm is as follows.

1) Determine the maximum k such that $2 \cdot k \cdot n$ is less than n_e. Create a random permutation σ of size $2 \cdot k \cdot n$, to represent

[2]Due to the fact that the node lists are already sorted, we can reduce this to an $O(n \cdot d)$ algorithm with appropriate data structures. However, given the tight constants which exist for sorting algorithms, we believe the constant for doing this would dominate $\log n$ for all reasonable n, so it is not of practical interest to do so. The same applies to most (but not all) sorts which occur in GEN.

[3]This does not change the abstract complexity, but the algorithm runs faster in practice.

$2 \cdot k \cdot n$ nodes, and join nodes σ_{2i} and σ_{2i+1} with an edge, $i = 0 \ldots (k \cdot n) - 1$. This creates a graph on $2 \cdot k \cdot n$ nodes with $k \cdot n$ edges, where each node is connected to exactly one other, i.e., a random matching.

2) Now collapse all nodes labeled $\sigma_{ki} \ldots \sigma_{(k+1)i-1}$ into a single node x_i. The result is an n node undirected graph where the degree of each node is exactly k (a k-regular graph[4]) and the distribution of graphs generated is guaranteed to be uniformly distributed over all k-regular graphs of size n.

3) Direct all edges from lower numbered nodes to higher, to get a directed graph. Randomly label n_{PI} fanin-0 nodes as PI (similarly n_{PO} fanout-0 nodes as PO). Randomly connect nonlabeled fanout-0 and fanout-0 nodes by new edges until they are exhausted, then continue randomly connecting random nodes to random nodes with fanin less than $k_{\max}$ until the graph contains n_e edges. When it is necessary to connect a node to a node of a lower number, separate the two by a flip–flop if one remains to allocate, otherwise search for an alternate connection that does not involve a back edge.

This process generates a graph with the specified number of each node type and the specified number of edges. A more standard definition random graph (i.e., $G(n, p)$ on n nodes with each edge existing with probability p), would not be an interesting comparison with GEN because it is much too hard to place and route (e.g., it contains more than $O(n)$ edges and a clique on $\log(n)$ nodes, almost always).

The graphs generated by the above process could be seen as a "first pass" version of GEN that takes fewer parameters into account. In fact, this algorithm alone would be an improvement over most naive approaches to generating random graphs for benchmarks. Comparing real circuits to clones and these random graphs is essentially measuring how far along the scale from "random" to "real" the current GEN approach has travelled. See Fig. 10 for an explicit visual of this scale.

B. Comparing Routing Results

We generated place and route data using 22 industrial benchmark circuits from Altera Corporation. For each circuit we generated both a clone circuit (by extracting the entire signature with CIRC) and a random netlist with the same number of nodes and edges. All 66 circuits were then run through each of VPR and Altera's MAX+PLUS2 commercial PLD software.

The benchmarks used are between 600 and 1100 logic blocks (four-input LUTs) and use up to 240 user I/O pins. By typical industry metrics this is roughly between 2000 and 5000 equivalent "ASIC gates."

The academic place and route tool VPR uses the model of a symmetric array of logic blocks, similar to a classic gate-array or the Xilinx 3000 FPGA architecture [23]. VPR allows the architecture to vary in channel width, and reports the total wirelength and the maximum channel width required for global routing.

[4]There are details to deal correctly and exactly with the double and and self-connections between nodes without sacrificing the uniform distribution, but these are beyond the current discussion.

TABLE II
ROUTABILITY COMPARISONS BETWEEN ORIGINAL BENCHMARK CIRCUITS, GEN CLONES, AND RANDOM GRAPHS ("·" IDICATES A NO-FIT)

	VPR wire			VPR tracks			10K20 tracks	
		clone	rand		clone	rand	clone	rand
	orig	%diff	%diff	orig	%diff	%diff	%diff	%diff
A	5102	21	144	6	16	83	14	132
B	7719	64	215	5	80	160	71	.
C	6344	27	160	6	16	116	30	.
D	6818	20	147	6	16	133	32	.
E	6609	53	266	5	60	160	35	.
F	4293	57	188	5	40	140	41	197
G	4147	2	158	5	0	140	16	208
H	5107	21	137	5	40	120	0	123
I	4692	19	155	5	40	160	23	132
J	6087	34	153	5	60	120	51	165
K	9313	42	202	6	33	133	38	.
L	6546	36	222	6	33	100	55	.
M	7748	86	248	5	100	220	85	.
N	10794	-43	52	10	-40	30	-41	.
O	8070	17	140	7	14	100	25	.
P	5562	88	268	5	80	180	90	.
Q	6460	71	167	5	80	160	.	.
R	6417	29	166	5	40	140	24	.
S	4662	28	170	6	0	83	16	108
T	8828	2	156	6	16	150	53	.
U	4876	81	201	4	75	175	63	174
V	4837	28	143	4	50	150	34	117
mean	6358	35%	175%	5.5	38%	134%	36%	151%

In MAX+PLUS2, we target an exact commercial programmable logic device (the Altera 10K20 [7]).

Our results are shown in Table II. The first three columns show the wirelength of the original circuit, and the percentage of extra wirelength required first by the clone circuit and second by the random netlist. The second group of columns show an alternative fitting quality measure, the maximum track-count required by VPR to achieve a route. In VPR there is no such thing as a "no-fit" because track-width are increased to the point where the circuit finally fits. The final two columns show the percentage increase in routing resources used by the clone circuit and the random circuit when implemented on the PLD. Since raw line-count information is proprietary, we show only percentage change for the commercial part; this is all that is needed to evaluate the algorithms in GEN.

For our metric of resource usage in the Altera part, we count the total number of full-horizontal (GH), half-horizontal (HH), and vertical (GV) lines used by the design in a 10K20, as reported by MAX+PLUS2. Because we are using an actual device, it is possible that a design does not fit (successfully complete place and route for the fixed resource counts of the part). Though all original circuits do fit in the commercial part, one of the clone circuits and 13 of the random graphs did not, and these are indicated by a "·" in the table. All of the original (nonsynthetic) circuits do fit in the device.

The last row of the table indicates the averages for each column. For the last two columns, the missing data is *not* included in the average, meaning that our summary statistics are conservative.

In [2] and [12] we also give the definition of a measure quantifying the degree of reconvergent fanout in a circuit between

0 and 1. By this measure, GEN circuits differ by about 0.19 on average, while random graphs differ by 0.28 on average.

We find that the clone circuits are harder to place and route than are the original circuits we took the specifications from, though a given clone is always dramatically closer to the original than the corresponding random graph which is much harder. On average the clone circuits used 35% more wirelength and 38% more tracks than the original circuit, whereas the random graphs, even ignoring those which failed entirely to fit in the device, used 175% more wirelength and 134% more tracks. This is further reflected in the implementation of the clone and random circuits on the commercial device where (when they did fit) the clone circuits used an average of 36% more routing resources and the random graphs used 151% more routing resources.

Put differently, the random graphs based only on size and edges are approximately four to five times the variation from the original benchmark (in terms of place and route metrics) as are the clone circuits generated using the entire specification. This provides overwhelming evidence both that the signature captures significant physical properties of the netlist which cannot be simulated randomly and that the GEN tool is able to generate circuits to resemble this signature.

The fact that the circuits do not exactly match the orignal means primarily that we have not fully captured all possible parameters with the current definition of circuit signature (or that to do so is not possible). We believe that a greater amount of local clustering is required (both in characterization and in generatioin) and continue to explore methods to provide this.

This experiment compares each of a group of clone circuits to the seed circuit which shares its signature. In Section V, we will analyze the behavior when we generate many clones of the same circuit.

C. Variation Within Equivalence Classes

It is interesting as well to discuss the variation between different circuit clones in the same equivalence class, because it gives us an idea of the completeness of the signature characterization as we have defined it.

In Section III, we measured how different a clone was from its seed circuit. We used many circuits but only one random data point for each circuit. Here we will generate many clones of some specific circuits and measure the distribution of clone circuits around the seed. In this context, we define *bias* to be the difference in wirelength between the seed and the mean of a large set of clone circuits and the *variance* to be the statistical variance around the mean for those clone circuits.

The best case for GEN is to achieve low bias and positive but controlled variance—i.e., the distribution of clones of a seed circuit is centred on the seed itself, but with enough variance that we are actually generating different circuits. A high variance would imply that we are generating close to random graphs (i.e., the circuits are not really in the same equivalence class). A "too low" variance would mean we have overspecified the problem—i.e., the signature actually captures all properties of the circuit and we basically echo the same circuit back as a clone. Bias most likely indicates a missing issue either in the parameterization or in the generation algorithm.

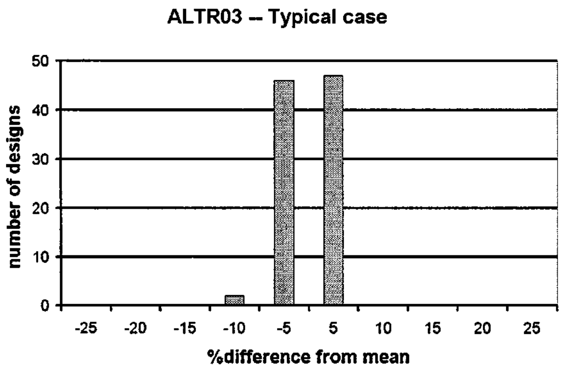


Fig. 7. Typical distribution of wirelength around the mean for 100 circuits in an equivalence class (ALTR03).

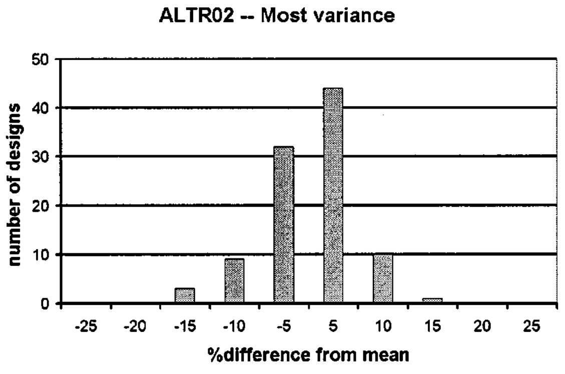


Fig. 8. Worst case distribution of wirelength around the mean for 100 circuits in an equivalence class (ALTR02).

For each of eight seed circuits we generated 100 clones as outlined in earlier sections. The typical case, as represented by 100 clone circuits generated from the seed circuit ALTR03 (Fig. 7), is that almost all circuits are within 5% from the average wirelength of the class. However, in line with the previously reported bias, the average wirelength of the class differed from the seed by 57%. Note that we are using larger circuits than in Table II, which contributes to greater variation.

The greatest variance case for the eight circuits studied is shown in Fig. 8. Approximately 20% of circuits were between 5% and 15% from the average.

To contrast the distribution of wirelength for circuits in the same equivalence class with a distribution of circuits not in the same equivalence class we generated 100 circuits using GEN. These circuits were forced to have 100 PI, 50 PO, and 1000 LUTs, but were otherwise unconstrained (except for the default distributions of the software as discussed in [2], [3]). Note these are not random graphs; they are GEN circuits whose signatures are drawn from default distributions rather than from a common seed circuit. The corresponding distribution of wirelength around the mean is shown in Fig. 9 and we see that the distribution is dramatically more varied. This is a positive result, providing further evidence that when the full signature is varied, we get significantly more variation than when the signature is held constant for a given set of circuits.

To illustrate the difference in bias and variance between GEN circuits and random circuits, we took one circuit (ALTR01) and generated 100 clone circuits and 100 random circuits (as defined in the preceding section). The results are shown in Fig. 10.

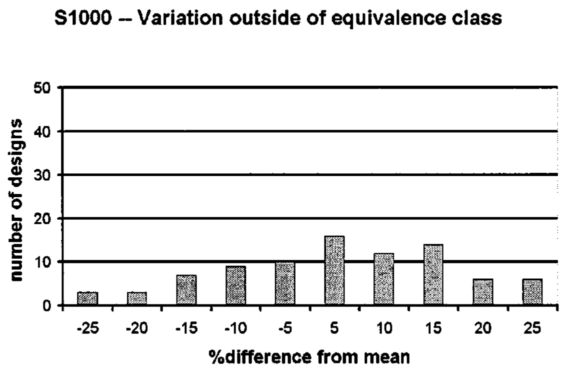


Fig. 9. Distribution of wirelength around the mean for 100 GEN circuits using default parameterization model.

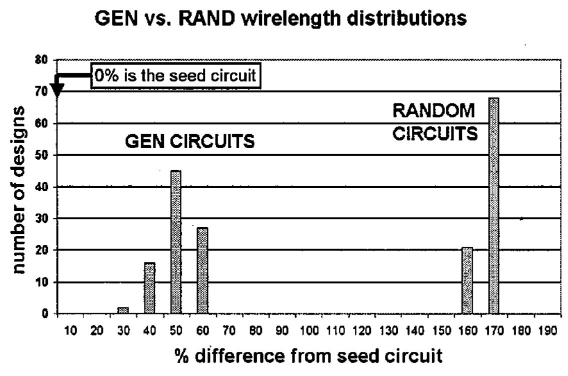


Fig. 10. Bias and variance for GEN versus random circuits.

TABLE III
WIRELENGTH AND TRACK STATISTICS FOR EIGHT ALTERA CIRCUITS AND THEIR 100 CORRESPONDING CLONES, AND FOR 100 1000-LUT RANDOM CIRCUITS

	W	W	W	W	T	T	T	T
		avg	sd	diff		avg	sd	diff
altr01	12219	21579	901	77	7	12.4	0.69	77
altr02	4989	7859	354	58	6	9.6	0.66	60
altr03	8560	16215	353	89	7	10.8	0.43	55
altr04	8197	12053	255	47	8	9.4	0.49	17
altr05	5711	7478	176	31	9	7.7	0.48	-15
altr06	6246	9093	253	46	6	9.5	0.50	58
altr07	8983	14716	515	64	6	11.4	0.61	91
altr08	7494	12670	222	69	7	11.0	0.36	57
			avg:	60			avg:	50
s1000		9611	2156			8.3	1.30	

Though the set of GEN clones show definite bias (57%) from the seed circuit, the randomly generated circuits show a pronounced 170% bias, roughly 3.5 times that of GEN. Fig. 10 in many results is the key point of this paper, illustrating the scale between "real" and "random" and the location of GEN circuits on this scale.

Table III shows the wirelength W reported for the original circuit, the average and standard deviations observed for the individual sets of 100 clones in the equivalence class of each seed circuit, and the percentage difference of the mean from the seed circuit. Also shown are the corresponding statistics for maximum track-count T as reported by VPR.

Results Summary: With respect to currently generated GEN circuits we can make several points. The "quality" of the circuits, as measured by bias, is significantly better than random circuits but not as close as we would like to the seed circuit. The seed circuit almost always requires less overall wirelength than the clones we generate. However, the variance is well in line with both expectations and our desire—we want varied circuits and ±5% is basically the amount of variance we would want to generate for reasonable experiments. For randomly generated benchmarks the variance is much more significant, larger than would be desired.

V. APPLICATIONS USING CLONE CIRCUITS

In this section, we address an important issue in CAD benchmarking: given that algorithm A has outperformed algorithm B by 10% on two different test circuits, what conclusion can we make? Is it reasonable to conclude that A is better, or is this simply noise because both the algorithms are heuristic? We have two fundamental problems. The first is that the two circuits may not be representative of the typical input to the program. For this we currently have no solution. The second problem is that we are observing noise in the behavior of the algorithms for these circuits because the algorithms are inherently heuristic: essentially we have a result which has no statistical significance. It is here that the use of clone circuits can play a role in our ability to benchmark.

The following simple methodology follows naturally from the definition of clone circuits and equivalence classes: Given a small set of initial benchmark circuits, use the process outlined in Section IV to generate a large number of clone circuits equivalent to each seed circuit. Apply each of the clone benchmarks to the problem under consideration and measure the appropriate statistical metric(s) to distinguish the multiple approaches. Then, in addition to the original circuits, consider the behavior of the class as a whole to the problem solution.

For example, if our goal is to analyze the effectiveness of two placement algorithms we could apply each to 100 clone circuits of each seed circuit and then compare the distribution of results between the two algorithms. If our goal is to determine whether an experimental programmable logic architecture requires 80 wires per row or if 60 is sufficient we perform place-and-route on the two different parts and analyze the number of fits and no-fits which result. In both cases we can gain more finely grained information from the large number of circuits than would be seen by looking only at the small number of initial benchmarks.

We point out that simply generating large numbers of circuits does not, in itself, allow us to make more accurate experiments. In order to apply this methodology, we are relying on the fact that the circuits being generated by GEN do have similar properties, as exhibited by their low variance, and that relative comparisons are thus justified.

In order to illustrate the use of GEN clones, we will apply the above methodology to distinguish two well-known partitioning algorithms.

For the first algorithm we obtained an implementation of the Fiduccia–Mattheyses partitioning algorithm [24] from Alpert's website [25]. This code was originally attributed to Dutt and

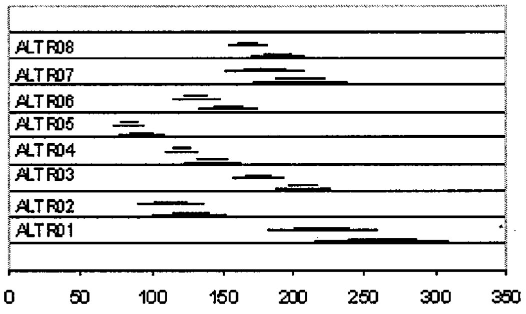


Fig. 11. 95% and 65% confidence intervals for cut-size as reported by hMetis (top) and FM (bottom), taken over equivalence classes for eight seed circuits.

TABLE IV
95% CONFIDENCE INTERVALS FOR THE DIFFERENCE IN MEAN CUT-SIZE BETWEEN HMETIS AND FM

	LOW	HIGH	LOW%	HIGH%
altr01	36.5	48.3	13.9	18.4
altr02	10.5	17.3	8.3	13.6
altr03	29.2	34.4	14.1	16.6
altr04	19.7	24.1	13.7	16.9
altr05	6.3	10.1	6.8	10.9
altr06	20.2	25.3	13.1	16.5
altr07	20.8	29.2	10.2	14.3
altr08	18.9	23.5	10.0	12.4
			11.3	15.0

Deng at the University of Minnesota Electrical Engineering Department and modified by Alpert for various netlist formats.

The second partitioning algorithm is the original implementation of hMetis by Karypis *et al.* [26], [27] from their University of Minnesota website. Based on the literature, this algorithm is expected *a priori* to have better results than the FM algorithm.

We performed bipartitioning on 100 clones each of the eight larger Altera benchmark circuits (800 circuits in total) and the original circuits and recorded the cut-size reported by each tool. We then calculated the mean and standard deviation and calculated the 68% (mean ± 1 standard deviation) and 95% confidence intervals (mean ± 2 standard deviation) for each of the two algorithms on each of the eight equivalence classes.

The results of the experiment are displayed pictorially in Fig. 11. For a given circuit, we have four lines: from top to bottom, the 68% and 95% confidence intervals for hMetis and then the 68% and 95% confidence intervals for FM (each calculated over the 100 circuits in the equivalence class). We observe (as expected) that the hMetis algorithm outperforms basic FM significantly: for half the circuit classes the 68% confidence intervals do not even overlap.

Given the large sample size, we can also get reasonable confidence intervals for the difference in mean cut-size between the two algorithms. Table IV shows this data. The average of the eight 95% confidence intervals for the difference in sample means is (11.3, 15.0), so we are 95% confident that hMetis will outperform FM by between 11% and 15%. Though not the point of this paper, it is clear that hMetis is a superior algorithm to the basic FM implementation.

A. Nonclone Uses of GEN

The preceding example used clone circuits in order to smooth variance and to improve statistical significance in the comparison of two algorithms. There are many other applications where we need to create circuits from scratch, without duplicating the physical properties of an existing benchmark, for example, if we need a circuit with 10 000 nets and do not have any seed circuits to clone. In these cases we utilize the specification language of GEN to specify several basic properties such as the number of nodes, edges, inputs and outputs, plus combinational delay, and the remaining portions of the incomplete signature are filled in from the GEN default scripts (the characterization parameters previously mentioned in Section III).

We and others [19], [28], [29] have successfully used such circuits in many applications. Currently the quality of the circuits is good enough that such experiments are reasonable, but we note that the variance does increase with the size of the circuit generated—as the size of the circuit increases away from the size of the benchmarks used for generating the default parameter distributions the quality does degrade. Future work to solve this problem involves primarily two issues: a better capturing of a *locality* parameter or distribution which allows us to build hierarchical circuits and a retuning of the default parameterization to larger benchmark circuits.

VI. CONCLUSION AND FURTHER WORK

In this paper, we have defined a new model for describing sequential circuits as a hierarchy of combinational subcircuits. The model includes the parameters of ghost inputs, ghost outputs, and their delay shapes. The model can also be used to describe more general forms of hierarchy than simply that between sequential levels. We have given an algorithm for generating realistic sequential benchmark netlists given the exact parameterization of a circuit in this model. This builds on previous research in which we gave a similar algorithm for the simpler problem of purely combinational circuits.

In addition, we have described a public-domain prototype software system which implements the sequential model with a characterization program (CIRC V3.1) and a generation program (GEN V3.1). These prototype tools have been installed at Altera, Xilinx, Actel, HP, and over 50 other academic and industrial sites and have were also used to contribute benchmarks to an informal partitioning competition at the 1996 Design Automation Conference. GEN has been used for benchmarking purposes in [28] and [29] and as a component of more recent automatic benchmarking efforts [19].

Using CIRC and GEN, we have "cloned" a number of industrial benchmark circuits and showed that GEN-circuits are significantly closer (three to five times) to real circuits for placement metrics than carefully generated random graphs. We analyzed the variance of circuits showing that the variation within an equivalence class was at acceptable and desirable levels and that true variation existed between different equivalence classes. Using GEN as a tool, we applied the generation of many circuits to a methodology to compare two partitioning algorithms with greater statistical significance that would otherwise be possible.

GEN is also capable of generating circuits “from scratch” using a set of default scripts based on analysis of benchmark circuits, which can be user modified.

We see a number of areas for future exploration. One is to modify the base generation algorithm to automatically impose a partition hierarchy on the circuit as it is being built, possibly similar to the use by Darnauer and Dai [3] of the Rent-exponent to introduce hierarchy in their partitioning benchmarks. Though GEN will currently output circuits of up to about 100 000 LUTs, we believe generating high-quality large benchmarks will require some degree of imposed symmetry and hierarchy within the netlist. A second area for future work would be to generate “system”-level hierarchy, by including datapath and other structured logic which can be synthesized or produced with LPM modules and random logic components from GEN. Other practical additions would be logical values (LUT contents) for simulation, addition of secondary signals, and multiple clock domains and embedded RAM.

Acknowledgment

The authors would like to thank Altera Corporation for providing the first author with a summer internship, during which parts of this research were performed, and for allowing access to their benchmark circuits during that time. The authors would also like to thank V. Betz for use of VPR.

References

[1] M. D. Hutton, J. P. Grossman, J. S. Rose, and D. G. Corneil, “Characterization and parameterized random generation of combinational benchmark circuits,” *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 955–996, Oct. 1998.

[2] ——, “Characterization and parameterized random generation of digital circuits,” in *Proc. 33rd ACM/SIGDA Design Automation Conf. (DAC)*, June 1996, pp. 94–99.

[3] J. Darnauer and W. Dai, “A method for generating random circuits and its application to routability measurement,” in *Proc. 4th ACM/SIGDA Int. Symp. FPGA’s*, Feb. 1996, pp. 66–72.

[4] W. E. Donath, “Placement and average interconnection lengths of computer logic,” *IEEE Trans. Comput.*, vol. CAS-26, pp. 272–277, 1979.

[5] K. Iwama and K. Hino, “Random generation of test instances for logic optimizers,” in *Proc. 31st Design Automation Conf. (DAC)*, 1994, pp. 430–434.

[6] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. presented at *Proc. 7th Int. Conf. Field-Programmable Logic*. [Online]. Available: http://www.eecg.toronto.edu/~jayar/.

[7] *1999 Data Book*, Altera Corp., San Jose, CA.

[8] S. Yang, “Logic Synthesis and Optimization Benchmarks, Version 3.0,” Microelectronics Centre of North Carolina, Research Triangle Park, NC, USA, 1991.

[9] M. D. Hutton, J. S. Rose, and D. G. Corneil, “Generation of synthetic sequential benchmark circuits,” in *Proc. 5th ACM/SIGDA Int. Symp. FPGA’s (FPGA’97)*, Feb. 1998, pp. 149–155.

[10] M. D. Hutton and J. S. Rose, “Equivalence classes of clone circuits for physical design benchmarking,” in *Proc. Int. Symp. Circuits Systems (ISCAS)*, 1999.

[11] ——, “Applications of clone circuits to issues in physical design,” in *Proc. Int. Symp. Circuits and Systems (ISCAS)*, 1999.

[12] M. D. Hutton, “Characterization and generation of digital benchmark circuits,” Ph.D., Univ. Toronto, Toronto, Canada, 1997.

[13] N. Kapur, D. Ghosh, and F. Brglez, “Toward a new benchmarking paradigm in EDA: Analysis of equivalence class mutant circuit distributions,” in *Proc. ACM Int. Symp. Physical Design (ISPD97)*, 1997.

[14] D. Ghosh, N. Kapur, J. Harlow, and F. Brglez, “Synthesis of wiring-signature-invariant equivalence class circuit mutants and applications to benchmarking,” in *Proc. Design Automation and Test in Europe (DATE)*, 1998, pp. 656–663.

[15] J. Harlow and F. Brglez, “Design of experiments for evaluation of BDD packages using controlled circuit mutations,” in *Proc. IEEE/ACM Int. Workshop Logic Synthesis (IWLS)*, June 1998.

[16] F. Brglez and R. Drechsler, “Design of experiments in CAD: Context and new data sets for ISCAS’99,” in *Proc. Int. Symp. Circuits Systems (ISCAS)*, 1999.

[17] D. Ghosh and F. Brglez, “Equivalence classes of circuit mutants for experimental design,” in *Proc. Int. Symp. Circuits and Systems (ISCAS)*, 1999.

[18] D. Stroobandt, P. Verplaetse, and J. Van Campenhout, “Generating synthetic benchmark circuits for evaluationg CAD tools,” *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1011–1022, Sept. 2000.

[19] J. Pistorius, E. Legai, and M. Minoux, “PartGen: A generator of very large circuits to benchmark the partitioning of FPGA’s,” *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1314–1321, Nov. 2000.

[20] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, “Memory-to-Memory connection structures in FPGA’s with embedded memory arrays,” in *Proc. 5th ACM/SIGDA Int. Symp. FPGA’s (FPGA97)*, 1997, pp. 10–16.

[21] E. R. Gasner, E. Koutsofios, S. C. North, and K.-P. Vo, “A technique for drawing directed graphs,” *IEEE. Trans. Software Eng.*, vol. 19, pp. 214–230, Mar. 1993.

[22] R. E. Tarjan, *Data Structures and Network Algorithms*: Soc. Industrial Applied Math, 1983.

[23] *The Programmable Gate Array DataBook*, Xilinx, Inc., San Jose, CA, 1999.

[24] C. M. Fidduccia and R. M. Mattheyses, “A linear time heuristic for improving network partitions,” in *Proc. 19th IEEE Design Automation Conf. (DAC)*, 1982, pp. 175–181.

[25] C. Alpert. The ISPD circuit benchmark suite. presented at *ACM Int. Symp. Physical Design (ISPD98)*. [Online]. Available: http://vlsicad.cs.ucla.edu/~cheese/.

[26] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: Application in the VLSI domain,” in *Proc. 34th IEEE Design Automation Conf. (DAC)*, 1997, pp. 526–529.

[27] G. Karypis and V. Kumar, “hMetis: A Hypergraph Partitioning Package, v1.5.3,” Univ. Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN, 1998.

[28] J. Swartz, V. Betz, and J. S. Rose, “A fast routability-driven router for FPGA’s,” in *Proc. 6th ACM/IEEE Int. Symp. FPGA’s (FPGA’98)*, 1998, pp. 140–149.

[29] Y. Sankar and J. S. Rose, “Trading quality for compile-time: Ultra-fast placement for FPGA’s,” in *Proc. 7th ACM/IEEE Int. Symp. FPGA’s (FPGA’99)*, 1999, pp. 157–166.

[30] M. R. Hartoog, “Analysis of placement procedures for VLSI standard cell layout,” in *Proc. 23rd Design Automation Conf. (DAC)*, 1986, pp. 314–319.

[31] . [Online]http://eecg.toronto.edu/~jayar

Michael D. Hutton (S’94–M’97) received the B.Math. and M.Math. degrees in computer science from Waterloo University, Toronto, ON, Canada, in 1989 and 1990, respectively, and the Ph.D. degree in computer science from the University of Toronto, in 1997.

He is a Senior Member of Technical Staff at Altera Corp., which he joined in 1997. While at Altera, he has been involved in the development of the APEX and Mercury programmable logic architectures, place&route algorithms, synthesis, and general research on programmable logic. He is author or co-author of eight patents or patent applications in the area of programmable logic. His research interests include graph theory and combinatorics and their application to architecture and design of programmable logic devices; wireability theory; and algorithms for CAD, including synthesis, partitioning, place&route, and timing analysis.

Jonathan S. Rose (S'79–M'80) received the Ph.D. degree in electrical engineering in 1986 from the University of Toronto, Toronto, ON, Canada.

He is a Professor of Electrical and Computer Engineering at the University of Toronto and a Senior Director of the Altera Toronto Technology Centre. From 1986 to 1989, he was a Research Associate in the Computer Systems Laboratory at Stanford University, Stanford, CA. In 1989, he joined the faculty of the University of Toronto. He spent the 1995–1996 year as a Senior Research Scientist at Xilinx, in San Jose, CA, working on the Virtex FPGA architecture. From 1989 until 1999, he was an NSERC University Research Fellow. In October 1998, he co-founded Right Track CAD Corp. which delivered architecture for FPGAs and PLDs as well as packing, placement and routing software for FPGAs and PLDs to PLD/FPGA and ASIC vendors. He was President and CEO of Right Track until May 1, 2000. Right Track was purchased by Altera and became part of the Altera Toronto Technology Centre, where he is currently one of two Directors. He has worked for Bell-Northern Research and a number of FPGA companies on a consulting basis. His research covers all aspects of FPGAs including architecture, CAD, field-programmable systems, and graphics and vision applications of rapid prototyping systems.

Dr. Rose is the cofounder of the ACM FPGA Symposium and remains part of that symposium on its steering committee. He coauthored a paper (with S. Brown) which won a distinguished paper award at the 1990 ICCAD Conference.

Derek G. Corneil received the Ph.D. degree from the Department of Computer Science, University of Toronto, Toronto, ON, Canada, in 1968.

He returned to the department after a Post Doctoral Fellowship in Eindhoven, The Netherlands. He served as the Chair of the department from 1985 to 1990. His research interests include the theoretical and algorithmic aspects of graph theory as well as computational geometry and other areas of combinatorics.

Dr. Corneil serves on the editorial boards of *Discrete Applied Mathematics*, *ARS Combinatoria*, and the *SIAM Monographs on Combinatorics and Applications.*