Characterization and Parameterized Generation of Synthetic Combinational Benchmark Circuits

Michael D. Hutton, Member, IEEE, Jonathan Rose, Member, IEEE,
J. P. Grossman and Derek G. Corneil

Abstract—The development of new Field-Programmed, Mask-Programmed and Laser-Programmed Gate Array architectures is hampered by the lack of realistic test circuits that exercise both the architectures and their automatic placement and routing algorithms. In this paper, we present a method and a tool for generating parameterized and realistic synthetic circuits. To obtain the realism, we propose a set of graph-theoretic characteristics that describe a physical netlist, and have built a tool that can measure these characteristics on existing circuits. The generation tool uses the characteristics as constraints in the synthetic circuit generation. To validate the quality of the generated netlists, parameters that are not specified in the generation are compared with those of real circuits, and with those of more "random" graphs.

I. INTRODUCTION

There is a need for benchmark netlists in order to compare and test the quality of new ASIC architectures and physical design algorithms. However, useful benchmarks are rare—they are usually too small to effectively test large future-generation products, and those large enough are often proprietary. Architectural research for FPGAs is even further constrained because large numbers of benchmarks are needed for specific sizes corresponding to the fixed capacity of the device.

Some attempts to alleviate this problem have been the efforts at MCNC to collect public benchmarks [24], the definition of a set of representative benchmarks by PREP [21], and the use of random graphs [15], [16], [18]. The use of random graphs is appealing because the supply is infinite, and the circuit size can be specified. However, only a small subset of random graphs can be considered reasonable with respect to electrical constraints such as gate fanin or fanout, topological properties such as maximum delay, and packaging constraints such as the number of pins. Compared to random graphs, circuits are inherently tame for implementation in gate arrays, and exhibit a hierarchical structure that leads to empirical observations such as Rent’s Rule1 [17].

Research supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Hewlett Packard. A preliminary version of this paper appeared at the 1996 Design Automation Conference [11].

Jonathan Rose is with the Department of Electrical and Computer Engineering, other authors are affiliated with the Department of Computer Science at the University of Toronto, Canada. M. Hutton's current affiliation is Altera Corporation, San Jose, CA. Email to Mike_Hutton@altera.com, jakay@eecg.toronto.edu, dgc@cs.toronto.edu or jrg@ai.mit.edu.

1Rent’s Rule: For a “reasonable” partition of a circuit into at least 5 modules, the relationship between the average number $P$ of terminals/pins on a module, and the average size $B$ of a module follows the relationship $P = k \times B^r$, where $k$ is a constant and $r$ is the Rent parameter which is characteristic of the circuit in question. Typical circuits have Rent parameters in the range 0.5 to 0.8.

In independent work, Darnauer and Dai [5] have proposed a method of generating random undirected graphs to meet a given ratio of I/O to logic and Rent parameter. Their work is primarily aimed at a study of routability and for creating partitioning benchmarks. They showed results for small circuits (from 77 to 128 lookup-tables) but it is not yet clear how successful the results are for evaluating new architectures and place and route software, or for larger circuits. Iwama et al. [13], [14] and also Kapur et al. [20] discuss the creation of benchmark circuits from existing circuits by function transformations, with applications to logic synthesis algorithms.

The key question for any work on benchmark generation is “How good are the circuits that are produced?” Thus, it is important both to have a strong experimental platform and to have objective measures of circuit quality with which to evaluate the output of the generation process.

As a measure of circuit quality we use other important characteristics that are not specified to the generation algorithm. In particular, one of the primary applications of automatic benchmark generation would be for testing physical-design CAD tools, so we place and global-route the circuits using VPR [2] and compare wirelength and channel width for the original circuits with circuits produced by GEN and with random graphs not produced by GEN. We call this step “validation” and illustrate it in Figure 1.

We define a set of graph-theoretic characteristics and parameters of circuits and measure these on real circuits up to 4500 LUTs (lookup tables) to form a profile of realistic circuits. This measurement is done with a new software tool called CIRC.

A second tool, GEN, generates a constrained synthetic circuit with values for the specified parameters either taken from the default profile or chosen by the user. In this way we can combine the advantages of parameterized random graph generation with the realism obtained by using actual circuits. This approach also allows for features not possible with standard benchmark sets. For example, one parameter can vary while others are fixed or scaled appropriately, to generate a “family” of circuits. The interaction between
the analysis and generation tools is of fundamental importance. CIRC can be used to analyze any private collection of circuits and determine alternative profiles for input to GEN.

The paper is organized as follows: Section 2 outlines the characterizations of circuits used for generation and validation of the synthetic circuits. In Section 3 we define the new algorithmic problem of synthetic combinational circuit generation with constraints. This problem is very difficult, and we present a heuristic algorithm to solve it exactly. The implementation of that algorithm is GEN. In Section 4 we describe the validation process and present results comparing GEN circuits with existing real benchmarks and random graphs. Some examples are presented in Section 5 and conclusions are drawn in Section 6.

II. CIRCUIT CHARACTERIZATION

This section describes some of the statistical and structural characteristics of circuits which we have identified. In this paper we focus on combinational circuits only, and have used the MCNC benchmark circuits [24] to form the basis for characterization and parameterization. Note that the users of our system could profile their own circuits with CIRC and specify the results as parameters to GEN (or modify the program default file) to customize the types of circuits generated.

A. Pre-processing of Analyzed Circuits

The MCNC benchmark circuits were converted from EDIF to BLIF, optimized with stis [23] (keeping the better result of script.rugged and script.algebraic) then technology mapped using FLOWMAP [4] into k-input lookup tables. Specifically, each circuit was mapped 7 times, into 2-input LUTs, 3-input LUTs up to 8-input LUTs. We chose to use lookup-tables because of their simplicity, functional completeness and the ease of changing to different LUT-sizes. We believe that the structural properties of circuits are sufficiently captured by the use of LUTs to determine valid characterizations without the added complexity of more technology-dependent libraries.

B. Characteristics and Parameters

There are two different types of characterizations: those needed to determine reasonable defaults for generation parameters which the user does not specify and those which characterize the fundamental structure of a circuit. In the remainder of this section we propose a set of characteristics. The complete default GEN-script for combinational circuits is available from our website [19].

B.1 Circuit Size and Number of I/Os.

The most basic characteristic of a circuit is the relationship between the size of the circuit (number of LUTs, \(n\)) and the number of primary inputs (\(n_{PI}\)) and outputs (\(n_{PO}\)). (Define \(n_{IO} = n_{PI} + n_{PO}\).) Using linear regression and experimentation, we have determined that a Rent-like functional relationship, \(\log(n_{IO}) = a + b \cdot \log(n)\) best captures the relationship between I/Os and circuit size. A simple linear relationship best describes the division of I/Os between inputs and outputs: \(n_{PI} = c + d \cdot n_{PO}\). Figure 2 shows a plot of \(\log(n)\) vs. \(\log(n_{IO})\), and a least-squares regression line for the Rent-like relationship. We note that simply determining values for the coefficients \(a, b, c,\) and \(d\) does not capture the increase in variance with \(n\) so we modeled these coefficients as Gaussian distributions around the best-fit line. The actual equations are given in the IOFrame section of comb.gen available from [19].

B.2 Combinational Delay.

Define \(d(x)\), the delay of node \(x\), as the maximum length over all directed paths beginning at a PI and terminating at \(x\), corresponding to the unit delay model. The delay, \(d(C)\) (or just \(d\)), of a circuit is the maximum delay over all nodes in \(C\). Using a similar empirical analysis to the above, we have determined a stochastic relationship between delay \(d\) and circuit size \(n\) in which \(d\) is roughly \(\log \log n\) on average (see the appendix).

B.3 Circuit Shape.

Combinational delay is very important in the characterization of circuits, precisely because it is so important in the design and synthesis process. Define the shape function, \(\text{shape}(C)\), of a circuit as the number of nodes at each combinational delay level. Figure 3 shows a small example circuit (cm151a), and its shape function (12, 4, 2, 2) displayed as a histogram. Note that even though the primary outputs are shown in circuit drawings we do not count them in determining delay or the shape function. Rather, we define "primary output" as a property of a node. While these examples are mapped to 4-LUTs, the basic form of the function remains similar for different LUT sizes.

The interesting thing about shape is that most circuits tend to have similar shapes. Figure 4 shows four shape functions. Of the 109 combinational multilevel circuits in the MCNC set, 36 have a shape which is strictly decreasing from the primary inputs (as "example2"), 53 have a conical shape, fanning out from the inputs to an extreme point, then strictly decreasing (as "alu2"), 12 have the con-
ical shape with a “bump” and only 8 did not fit into these categories. This is fundamentally different from degree-constrained random graphs (defined in Section 4 and discussed further in Section 5) which have much “flatter” shapes.

B.4 Edge Length Distribution.

Since nodes have a well-defined delay, we can define the length of a directed edge by length(x, y) = d(y) − d(x). Clearly, the edge length is always between 1 and delay(C), and we define a related edge length distribution. In the example of Figure 3 there are 24 edges of length 1, and 2 each of length 2 and 3, so the edge length distribution is (0,24,2,2,0). (For technical reasons there is a component for length-0 edges which always has the value 0.) We find that almost all circuits have an edge-length distribution with a similar shape: a large number of edges of length 1, and a quickly falling distribution over the combinatorial delay of the circuit. In the default files, we model this with a function based on the exponential distribution.

B.5 Fanout Distribution.

Define fanout(x) as the number of edges leaving a node x. A circuit’s fanout distribution (the number of nodes with fanout 0, 1, 2, etc.) is an important structural parameter. Note that fanin is less interesting for technology-mapped circuits because they have an a priori constraint on fanin. We have determined the fanout distributions of the MCNC circuits, and have developed a heuristic algorithm [10] which generates reasonable fanout distributions for specified size and shape parameters. This algorithm uses a greedy probabilistic sampling approach, parameterized by the number of nodes and edges, delay and the maximum fanout, whereby we take a truncated, exponential-based function and sample it for fanout values, occasionally re-building the function to avoid taking too many more high-fanout values than possible for the number of edges.

B.6 Reconvexity.

Reconvexity occurs when multiple fanouts from a single node x, after travelling through subsequent nodes in the circuit, branch back together at a later point y—we say the circuit is reconvexent at y. Many circuits exhibit reconvexent fanout, but in widely varied degree, so an appropriate characterization is to quantify this amount.

Define the out-cone of a node x (in a circuit with no directed cycles) to be the recursive fanout of x: all nodes reachable by a directed path from x. Figure 5 shows out-cone(a). Edges which are not in the out-cone, but are adjacent to nodes which are, are shown as dashed lines.

For circuits mapped to 2-LUTs, define the reconvexity number of node x, R(x), as the ratio of the number of fan-in-2 (i.e. “reconvexent”) nodes in out-cone(x) to the size of out-cone(x):

\[ R_n(x) = \frac{|\{y \in \text{outcone}(x) \mid y \text{ has fanin 2 in outcone}(x)\}|}{\text{outcone}(x)} \]  

(1)

This value arises from its combinatorial interpretation. By Kirchhoff’s theorem [9, pp. 49-54], the numerator counts the \( \log_2 t \) where t is the number of spanning out-trees\(^3\) rooted at x in the directed graph representation of the circuit. Essentially, each reconvergent node represents a choice of two alternatives in the construction of a spanning out-tree, which multiplies the number of trees by two (adds 1 to \( \log_2 t \)). Each non-reconvergent node represents a “required” in-edge, hence does not affect the number. The purpose of taking the logarithm is simply to obtain tractable numbers when dealing with large graphs. The denominator then scales that value with the size of the out-cone so that different graphs can be compared based on their relative amount of reconvergent, which otherwise would be dominated by the size of the circuit.

For circuits mapped to k-LUTs, k > 2, the reconvergent calculation generalizes, both algorithmically and combinatorially, if we set the numerator as the sum, over all nodes y in the out-cone of x, of \( \log_2(y) \). Thus \( 0 \leq R(x) \leq \log_2(k) \).

\[ R(x) = \frac{\sum_{y \in \text{outcone}(x)} \log_2(\text{fanin}(y))}{\text{outcone}(x)} \]  

(2)

Further generalizations yield various different quantifications of reconvergent in sequential circuits[10], but these are beyond the scope of this paper.

To identify the reconvergent \( R(C) \) present in an entire circuit C, we compute the weighted (by out-cone size) average of \( R(x) \) for all primary inputs x in C. Thus \( 0 \leq R(C) \leq \log_2(k) \) continues to hold for circuits. In this way, highly reconvergent small portions of a circuit will not unduly affect the overall quantification.

The observed reconvergent numbers for the 198 combinational and sequential 2-LUT-mapped MCNC circuits

\(^3\) A spanning out-tree rooted at r is a spanning tree such that each node, except the designated root node, has exactly one fanin. Hence each node lies on a unique directed path from the root.
vary between 0.0 and 0.92, with a relatively even distribution of circuits through the range 0.0 to 0.85. \( R \) is somewhat a measure of complexity of the logic—we find that intuitively simple, tree-like, logical functions have low \( R \) (e.g. parity: \( R = 0.00 \), decod: \( R = 0.00 \), max: \( R = 0.14 \)), and more complex functions have higher \( R \) (e.g. alu2: \( R = 0.53 \), sqrt8ml: \( R = 0.56 \)). Combinational logic and the combinational parts of sequential arithmetic logic fall mostly in the range 0.0 to 0.6, whereas the combinational parts of finite state machines are mostly in the range 0.5 to 0.85.

There is a high degree of correlation between \( R \) and the other characteristics of a circuit; in particular, the number of edges (when \( k > 2 \)), and the shape and out-degree functions. Using the examples of Figure 4, circuits which have an exaggerated conical shape, such as alu2 (\( R = 0.53 \)) and cordic (\( R = 0.45 \)) tend to have higher reconvergence values, whereas circuits like example2 (\( R = 0.17 \)) are lower. This also tends to explain the difference between combinational and sequential circuits because the first “sequential level” of most finite state machines tends to be very conical, due to a low I/O to logic ratio.

### III. Circuit Generation

Now that we have defined a number of parameters to describe circuits, we proceed to the second goal of the paper, an algorithm to generate parameterized synthetic circuits.

Figure 6 shows an example output from \( \text{GEN} \) for the parameterization: \( n=23 \), \( n_{\text{edges}} = 32 \), \( k=2 \), \( n_{\text{PI}} = 7 \), \( n_{\text{PO}} = 2 \), \( d=4 \), \( \text{shape} = (0.38, 0.31, 0.19, 0.12) \), \( \text{max}_\text{out} = 4 \), \( \text{fanouts} = (0.9, 0.65, 0.13, 0.04, 0.09) \), \( \text{edges} = (0.9, 0.1) \).

The \( \text{GEN} \) program consists of two functional stages. The first is to determine an exact and complete parameterization of the circuit to be generated, using partially-specified user parameters and default distributions. The exact parameterization shown to the right of Figure 6 is such an instantiation of the more general parameters just given. The second stage is to output a synthetic circuit with that exact parameterization, which we deal with first.

#### A. The Generation Algorithm

Here we give the details of the generation algorithm.

The inputs to \( \text{GEN} \) are \( n, n_{\text{edges}}, n_{\text{PI}}, n_{\text{PO}}, d \) (delay), \( k \) (LUT-size), \( \text{max}_\text{out} \) (maximum allowable fanout of any node), the shape function, the fanout and edge length distributions and the locality parameter \( L \) (not yet defined).

The output is a netlist of \( k \)-input lookup-tables. Note that we do not currently specify the contents of the LUTs, so the output is a physical netlist lookup-tables. Reconvergence is not a generation parameter but we use the reconvergence number of generated circuits in the validation process of Section IV.

Since parameter expansion (the first major step of \( \text{GEN} \)) has already taken place, we now the distributions are exact, meaning

\[
\sum_{i=0}^{n} \text{shape}[i] = \sum_{i=0}^{n_{\text{max}}} \text{fanouts}[i] = n, \text{ and} \]

\[
\sum_{i=0}^{n_{\text{out}}} \text{edgest}[i] = \sum_{i=0}^{n_{\text{out}}} \text{fanout}[i] = n_{\text{edges}}.
\]

Using the shape distribution, shape[1..d], we are able to immediately define the number of nodes at each combinational delay level. Fanouts[1..max_out] gives us the exact set of fanouts available (but not yet assigned to nodes). Edges[1..d] gives us the set of edges to be assigned between nodes. Our problem is then, as illustrated in Figure 7, to determine a one to one assignment of fanout values to nodes, and an assignment of edges between nodes such that the number of out-edges from a node equals its assigned fanout, and the number of edges in to a node is no more than the bound, \( k \), on fanin. We have a number of further constraints: the resulting graph must be acyclic (as the circuit is to be combinational); every node must have at least one fanin from the previous delay level, and no fanins from later delay levels (so that combinational delay of a node is as specified by the shape function); all nodes at delay-0 (i.e. the inputs) have no fanins, and all other nodes have at least 2 fanins; and all fanins to a node must come from distinct nodes (no duplicate inputs).

We need the following definitions:

(a) \( N_i, i=0..d \) is the set of nodes at delay level \( i \), where \( N = \bigcup_{i} N_i \).

(b) \( F = \{f_j, j = 1..n\} \), is the set of node-fanouts, and

(c) \( E = \{e_h, h = 1..n_{\text{edges}}\} \), is the set of edge-lengths (abstractly, the set of all edges).

We formally define the generation problem in Figure 7.

This assignment problem appears to be computationally difficult and we conjecture it is NP-hard. It is important, moreover, to have a nearly linear time algorithm in order to generate large circuits. Therefore we solve the problem heuristically, as described in detail in the sub-sections which follow.

The general line of approach is as follows: First we determine an assignment of edges and out-degree to levels \( N_i \), but not yet to individual nodes within each level. We call
the $N_i$ level-nodes and the graph at this point the level-
graph. We then split each level into nodes and assign first
fanouts and then edges, previously assigned only to levels,
to the individual nodes. A post-processing step designates
any additional primary outputs required.

There are 5 major steps in the algorithm for generating a
combinational circuit from an exact specification. We pro-
vide enough detail here to understand the important as-
pects of the algorithm. Readers who are interested in the
more detailed aspects of the software are referred to the
external documentation and the public-domain implemen-
tation and source-code [19]. Throughout the description
of the algorithm, we will follow through the small example
of Figure 6, from the exact parameterization to the final
circuit.

A.1 Boundaries on in/out-degree (pre_degree.c).

To assign edges between levels, we first determine the
maximum and minimum fanin (in-degree) and fanout (out-
degree) for each delay level: vectors $\min_{\text{in}}[i]$, $\max_{\text{in}}[i]$, 
$\min_{\text{out}}[i]$ and $\max_{\text{out}}[i]$. While the number of nodes at
each level is known, the total fanin is not known exactly
in general because a four input LUT may only have two or
three inputs in many cases. For 2-LUTs (as in our example)
the fanin bound is deterministic. The reason we need these
bounds is to more tightly constrain the problem before we
proceed with edge assignment.

We require each node at level $i$ to have between two
and $k$ fanins, one of which must come from the preceding
delay level to establish combinational delay. This gives
immediate rough bounds of $\min_{\text{in}}[i] = 2 \cdot n_i$ and $\max_{\text{in}}[i] = k \cdot n_i$. Similarly, each non-primary-output node
must have at least one fanout, providing an initial lower-
bound $\min_{\text{out}}[i] = n_i - (n_{\text{PO}} - n_d)$ (noting that level $d$ has
all POs, so level $i$ can have at most $S$ ($n_{\text{PO}} - n_d$) fanout-0
POs).

$\max_{\text{out}}[i]$ is calculated heuristically using the fanout
distribution and the previously calculated vectors for later
levels, based on a number of rules: For example, $\max_{\text{out}}[i]$
is bounded above by $\sum_{j=i+1}^{d} \max_{\text{in}}[j] - \sum_{j=i+1}^{d} \min_{\text{out}}[j]$
representing the remaining inputs in the LUTs at later levels
less the reserved output edges for later levels.

The initial bounds are improved iteratively: the bounds
on $\max_{\text{out}}$ just determined necessitate an updated calcula-
tion of $\min_{\text{in}}$ and $\min_{\text{out}}$ for later levels which in turn
affect $\max_{\text{out}}[i]$. We continue until no more tightening
of the boundaries is possible, typically only a few iterations,
and probably no more than $d^2$.

The result of this step is the determination of the bound-
ary vectors $\min_{\text{in}}[i]$, $\max_{\text{in}}[i]$, $\min_{\text{out}}[i]$ and $\max_{\text{out}}[i]$, $i=0..d$, as pictured in Figure 8 (Step A.1). Each level-node
$N_i$ is labeled with $n_i$ and its fanin boundaries (northwest
corner) and fanout boundaries (southwest corner).

A.2 Assign edges between levels (level.c).

There are three phases to edge assignment. As edges are
assigned, we calculate two new vectors, assigned$_{\text{in}}[i]$ and
assigned$_{\text{out}}[i]$ to represent the “used up” in and out-
degree for level $i$. The available in and out-degree to a level
is defined as the difference between the assigned and the
maximum, and the required in and out-degree is defined as
the difference between the assigned and the minimum (or
0 when assigned is larger than minimum).

Step A.2(a). We first consider the “critical” unit edges, edges
which connect to the first and last levels of the circuit or which are required to ensure that combinational delay constraints can be met. We assign $\max(\min_{\text{out}}[0], \min_{\text{in}}[i])$ edges between levels 0 and 1,
and $\max(\min_{\text{out}}[d-1], \min_{\text{in}}[d])$ edges between levels

Fig. 7. The generation/construction problem.
$d - 1$ and $d$. Then we establish the combinatorial delay for each other level $i$, $i = 2, \ldots, d - 1$, by assigning $n_i$ edges between levels $i - 1$ and $i$.

Step A.2(b). Secondly, we assign the long (length $> 1$) edges. This is a crucial step, because if these are assigned poorly it becomes difficult or impossible to complete the graph construction without violating the shape or edge-length distributions. Long edges are assigned probabilistically. We calculate the number of possible level to level starting and ending point combinations for edges of length $l$ at each level $i$. MIN$[\text{avail.in}[i], \text{avail.in}[i+l]]$, and sample the resulting discrete probability distribution to assign the edges, updating the distribution after each assignment. It is an important feature of GEN that we sample from this distribution rather than just choosing the “optimal” assignment, because we want to produce circuits with different features on each execution with the same parameterization.

Step A.2(c). We have only unit edges left. The last part of this step is to assign the remaining required edges—those necessary in order to meet the required min.in$[i]$ and min.out$[i]$ for each level $i$. This part is purely deterministic. Any remaining unit edges are held back for assignment later in A.3. Typically, these remaining edges are about 10-25% of the original unit edges (or 7-18% of all edges).

The output of A.2, shown in Figure 8 (A.2), is a modification to each level-node $N_i$ in the level-graph, this being a vector (though shown pictorially in the figure) indicating the number of assigned fanout edges of each length that have been assigned to the level. A.2 also guarantees that the assignment has met the minimum in and out degree requirements for each level.

A.3 Partition the total fanout at each level (degree.c).

We have the vectors assigned.in$[i]$, assigned.out$[i]$, max.in$[i]$ and max.out$[i]$. However, the assigned out-degree is a total for the level, not a list of individual node values from the fanout distribution.

In this step we partition the total out-degree (e.g. 10) of level $i$ into $n_i$ (e.g. 4) individual values taken from the fanout distribution (e.g. $4, 3, 2, 1$, summing to 10).

First calculate target fanouts, target$[i]$, $i = 0, \ldots, d - 1$, in the range assigned$[i]$ to max.out$[i]$, such that $\sum_{i=0}^{d} \text{target}[i] = n_{edges}$. Again, we sample a probability distribution calculated as in A.2(b), rather than performing a deterministic allocation. The goal is to assign the target out-degrees which are, on average, proportionate to the amount of slack between the minimum and maximum values for each level, but probabilistically rather than in exact proportion so that the resulting circuit is different with each execution of GEN with the same inputs.

We are left with the problem of partitioning each target$[i]$ into $n_i$ values taken from the fanout distribution. Even for a single level, this integer partitioning problem is NP-complete [7, page 223] to compute exactly, so we can only manage a heuristic solution. Fortunately, this is made easier because of the remaining unassigned unit-edges—target$[i]$ is flexible within the range min.out$[i]$ to max.out$[i]$, so we need only an approximate integer partition for each level, and can allocate the remaining unit edges as required to make the result exact.

Before entering the main operation of the degree-allocation step, we examine the low fanout levels, defined as levels which have a total fanout less than $2n_i$. Assigning a high-fanout value to such a level could result in later difficulties as we “run out” of edges for giving individual nodes at least one fanout. To dispose of these levels, assign fanouts of 0, 1, and 2 deterministically, based on the availability of fanout-0 values in the fanout set (some, but not all PO nodes will have fanout 0).

The main operation of this step is probabilistic and iterative. For each level, compute average.out$[i] = \text{target}[i]/n_i$, and the values min.possible.out$[i]$ and max.possible.out$[i]$ indicating the degrees which could feasibly be assigned to any node at level $i$ (using the rules of A.1 applied to individual nodes). Then iterate through the values in the fanout distribution $F$ from largest to smallest (the largest being usually the more restrictive, hence more difficult to place). Among the levels that can accept the current fanout $f_j$ (based on min.possible.out and max.possible.out) we sample average.out$[i]$ as a probability distribution (with the same goals as just mentioned for targets) to choose the level to which $f_j$ will be assigned. Each time we update the status vectors (assigned.out, available.out, average.out, min. fanout, maximum. fanout, min. possible.fanout and max. possible.fanout) for the chosen level.

Because of the probabilistic assignment, some levels will receive more than the target number of edges (based on the sum of their fanouts) and some will receive fewer. However, the details of the assignment do guarantee that all levels will receive between their minimum and maximum total fanout.

On the relatively rare occasion that a fanout cannot be accepted by any level, we decrement the fanout value by 1 and continue. This can lead to a minor modification of the input specification, as discussed further in Section III-C.

At the completion of A.3, all edges have been assigned to levels, and the level-node for each level $i$ contains a list of edges (and their length) which leave that level, and a list of $n_i$ fanout values $f_{ij}$, $j = 1, \ldots, n_i$, which sum to the total fanout of the level. Figure 8 illustrates this situation: the breakdown of total fanout into an (unordered) set of out-degrees is shown above A.3, and the edge-length distribution is as in A.2. (Unfortunately, to get an edge-length distribution which differs from steps A.2 to A.3, we would need to use $k > 2$ and a larger $n$, which would make the main operation of the algorithm more difficult to view.)

A.4 Split levels into nodes (nodes.c).

For this step, levels are treated independently. We need to split each level-node $N_i$ into $n_i$ individual nodes, and assign each of these a fanout from the list of available fanouts $f_{ij}$ now assigned to level $i$. This would be trivial, were it not for the necessity to introduce locality into the resulting circuit, and so we first discuss how we impose locality in the generation.

Because of the way that real circuits are designed,
whether a bottom-up or top-down methodology is used, an
inherent local structure develops in graph representation of
the circuit. Nodes tend to exhibit a clustered behaviour, whereby
nodes in a cluster tend to accept fanin from approxi-
mately the same set of nodes as other nodes in their
cluster. This local clustering is described by Rent’s Rule
[17] and theoretical models to explain it have been pro-
posed by Donath [6] and others. Without some method of
modeling local behaviour, our circuits would be “too ran-
don” and hence not realistic.

Our approach to introducing locality into the genera-
tion algorithm is to impose an ordering on the nodes, and
use proximity in this order as a metric of locality when
we later choose the edge-connections between nodes. This
can also be viewed as trying to generate graphs which will
“look good” when displayed as pictures such as Figure 6,
because minimization of edge lengths in a graph drawing
also has the effect of reducing crossings and of displaying
any inherent locality in the graph [8]—by creating a circuit
with one known good ordering/drawing we have simulated
this form of locality in the generation. The ordering we will
use is simply the sorted order within the linear list of nodes
within each level. Note that any ordering of the nodes is
arbitrary until we have associated distinguishing features
such as fanout or edge connections to the nodes. The mea-
ure of the goodness of an edge is then measured as the
distance between the source and destination nodes in their
levels node-lists, relative to competitors. As a result, the
order in which fanouts are assigned within the node list
becomes important, because placing high-fanout nodes in
an unbalanced way into the node list will skew the effects
of locality measurement in step A.5.

The locality index assigned to each of the \( n_i \) nodes in
the nodelist for level \( i \) is a scaled proportion of the maximum
size level. Thus if the maximum level contains 100 nodes,
and the current level 10, then its nodes will have locality
indices 5, 15, 25, ..., 95. Before fanout allocation the order
of nodes is arbitrary, so the nodes are now indistinguishable
other than for this index.

Our goal in assigning fanouts to nodes in the list is to
distribute the high fanout nodes well for maximum loca-
tility generation. To do this, we sample a binary tree
distribution to allocate fanouts, in order from the high-
est to lowest fanout. To calculate the distribution, la-
bel the nodes of a balanced binary tree on \( n_i \) nodes with
the number of leaves in its subtree. Then perform an in-
order traversal of the tree, and place the labels in pdf[i],
\( i = 1..n_i \). For example, the binary tree pdf of length 15 is
\([1, 2, 1, 4, 2, 1, 8, 1, 2, 1, 4, 1, 2, 1] \). In the most likely case, then,
the highest fanout node would be assigned in the middle,
the next two highest fanouts at the quartiles, and so on.

Another way to view this distribution is to take an ordered
list of \( n_i \) nodes, assign a value \( p \) to the middle node \( n_i/2 \),
a value \( p/2 \) to the nodes \( n_i/4 \) and \( 3 \times n_i/4, p/8 \) to the mid-
dle nodes in the resulting ranges and so on, then scale the
resulting distribution to integers. The point of this opera-
tion is to (on average) place the highest fanout node in
the middle of the ordering, the next two highest fanout nodes
at the quartile points, and so on. Again, probabilistic sam-
pling means we don’t get exactly the same result each time,
and just as importantly that we don’t generate artificially
symmetric circuits.

The purpose of assigning fanouts in this way is so that we
do not place high-fanout nodes at the edges of the ordering:
observe how placing the two higher fanout nodes towards
the centre of the drawing of Figure 6 serves to reduce the
wirelength of the drawing. We want to emulate this effect
in the generated circuits.

This algorithm assigns, to each node \( x_j \) in level \( i \), a value
fanout(\( x_j \)) from \( \{ f_{ij} \} \) and a value index(\( x_j \)) to each \( x_j, \)
\( j = 1..n_i \). A further calculation assigns \( p_j \), \( 0 \leq p_j \leq f_j \),
the long-edge fanout of node \( x_j \), defined as the number of
edges of length greater than one from \( x_j \). This is again
probabilistic, sampled uniformly over all out-edges in the
level.

At the conclusion of step A.4, each node \( x \) in the cir-
cuit has an assigned delay, fanout, long-fanout and index,
but no actual edges have been assigned between nodes at
different levels in the graph. The fanout values are shown
in Figure 8 (A.4). This information, plus the edge-length
assignments from A.2 in the figure comprise the input to
A.5 of the algorithm.

A.5 Assign edges to nodes (edges.c).

The major remaining step is to connect the fanout edges
on each node to a corresponding input port on a node on a
later delay level, as specified by the edge-length. We pro-
ceed from level 1 to level \( d \), connecting the edges to
each level \( i \).

To connect the in-edges to level \( i \), we first calculate the
source list, of unconnected edges preceding level \( i \) which are
of the correct length to connect to level \( i \). Nodes with mul-
tiple fanouts are inserted only once in the list, and nodes
are deleted as their fanout is exhausted. The destination
list consists of all nodes at level \( i \). Both these lists are
maintained in sorted order by node index (as defined in
A.4).

Step A.5(a). If the size (in edges) of the source list is
more than twice the number of available nodes in the des-
tination list, we preprocess the high-fanout nodes (those
with fanout more than \( 1/8 \) the number of nodes in the des-
tination list) separately. To process a single high-fanout
node \( x \), we randomly choose a range of nodes of size be-
tween fanout(\( x \)) and \( 3 \times \text{fanout}(x)/2 \), centered at the closest
index node \( y \) in the destination list to index(\( x \)). Choosing
a random set of fanout(\( x \)) nodes from this set, we make
the physical edge connections, and update all status vec-
tors. This process is repeated for all high-fanout nodes in
the source list. The purpose of this step is to avoid a situa-
tion where we have a large number of out-edges from the
same source node \( x \) later in the edge-assignment phase
which cannot be assigned without creating double connec-
tions from node \( x \) to some node \( y \).

Step A.5(b). Establish combinational delay by connect-
ing each node in the destination list which does not already
have a fanin edge from 5(a) to one node from the source
list which is at the previous combinational delay level. To choose the fanin for node $y$, we sample the unit-edges in the source list $L$ times, where $L$ is the locality parameter of generation (discussed below), choosing the result $x$ with the closest index to $\text{index}(y)$.

Step A.5(c). Perform a second sweep similar to (b) (including locality) to ensure that each node $y$ in the destination list receives a second incoming edge. There is no longer a restriction on the length of the edge, but we cannot choose the same fanin as is already attached to $y$ from step 5(b).

Step A.5(d). Now that the minimum requirements are met for each node in the destination list, iteratively choose a random node from the destination list, and choose an input from the source list as per 5(b) and (c), including locality generation. Continue until the source and destination lists are exhausted.

At the conclusion of A.5, the circuit is complete, except that we may have fewer out-degree zero nodes than the required number of primary outputs. We post-process the circuit to (randomly) label the required number of additional LUT nodes as primary outputs.

The final result of the generation algorithm (for one random seed) on the progression of Figure 8 from the original specification is the original example of Figure 6.

B. The Locality Parameter

The locality parameter $L$ has not been formally discussed at this point. As mentioned in Step 4, we find that a purely random connection of edges between levels does not model the type of clustering found in real circuits. At the same time, deterministically connecting the edges based on aligning index values yields a circuit which is overly local, and is actually too easy to place and route. We find that a reasonable approach is to define a locality parameter $L$, and use it to bias the above algorithm towards greater locality: when choosing an input for a given destination node, we sample $L$ times, and choose the source node which is closest in index value to the destination node under consideration. For higher values of $L$, the probability of directly lining up indices increase, for $L=1$, the algorithm is as originally described.

Though $L$ can be specified as a user-parameter to generation it does not currently tie to the characterization of a circuit. That is, we have no way to measure it for a specific given circuit. Through experimentation, we have found that there is no constant locality parameter which yields the correct results, but a value which scales logarithmically with the size of the circuit yields good results.

We find that the locality parameter can significantly affect the properties of the resulting circuit, an issue discussed further in Section 5. Though the empirical results from the algorithm for introducing locality are good, we feel that there is an underlying combinatorial structure which would give a better theoretical understanding of the connectivity in digital circuits. The ideal case would be to measure locality in the analysis of a circuit then parameterize and model it in the generation of a random circuit.

We are currently pursuing further work to this end.

C. Meeting the input specification

It is not always the case that $\text{GEN}$ determines a circuit which meets the input specification. As with any heuristic algorithm, there exist input possibilities for which the heuristics fail. In the case of $\text{GEN}$, we find that we are occasionally (1-2% of the time) unable to complete a valid circuit. In these cases, the tool reports a “failure to determine a circuit with this specification.” About 2-3% of the time, $\text{GEN}$ will complete a circuit, but will report that it was forced to significantly modify the input specification in order to finish (though this is necessarily minor enough to not warrant failure). We consider these to be minor problems, because the user can re-run the tool with a new random seed, and typically will get an acceptable output on the second try.

D. Parameterization and Default Scripts

The discussion to this point has involved the generation of a circuit with a completely specified exact specification. In practice, the user would choose only a small number of parameters (or possibly just $n$), and the remaining are chosen from default parameter distributions.

$\text{GEN}$ is augmented with a sophisticated C-like language, SYMPE, for parameter generation. The default distributions are written in this language, and the user can specify modifications in the control script for a circuit. SYMPE provides a great deal of control over parameters. For example, $n_{\text{ID}}$ is currently defined as a set of piecewise Rent-like equations, each of which has the Rent parameter drawn from a gaussian distribution.

The current default sets and parameters have been determined from experimentation with the MCNC benchmark circuits. It would be possible to perform the same experimentation with an alternate set of benchmarks, and generate a modified default script.

SYMPE allows parameters to be specified as constants, drawn from statistical distributions or chosen as functions of other parameters. Figure 9 shows a series of circuits generated with the varying $n$ but other parameters fixed, to generate a family of related circuits. SYMPE scales related parameters (e.g. depth and shape) yet retains the similarity of other properties. This ability to scale circuits while retaining fundamental similarities introduces an entirely new paradigm for evaluating the scalability of architectures and algorithms.

E. Input scripts and clone circuits

The input to $\text{GEN}$ takes basically two forms. The user can specify either a parameterization which they create themselves, or use $\text{CIRC}$ to extract a parameterization from an existing circuit and generate a clone of that circuit. The two approaches can be mixed by modifying a clone script.

Figure 10 shows the second case, in the form of a GEN-SCRIPT output from $\text{CIRC}$ given the MCNC circuit ah04. The object “comb.circ” referred in the script to is the default frame in the script $\text{comb.gen}$, and the specifications
In practice \texttt{gen} is very fast. Generation of a 2000 \texttt{LUT} circuit takes about 7 seconds on a Sparc-5, using 500K of memory. For perspective, the same circuit requires about 45 minutes and 2M of memory to place and route using even a fast and memory-efficient tool such as \texttt{vpr}. A circuit of 30,000 \texttt{LUTs} requires about 30 seconds and 1M to generate, versus a half-day or more to place and route.

We have successfully generated circuits of up to 200,000 \texttt{LUTs}, well beyond the level of current FPGAs. The \texttt{gen} implementation is currently limited to about that size, due simply to the use of 32 bit integers for counters and distributions. Larger circuits would require special-purpose arithmetic, at least for specific parts of the code, or a hierarchical approach to generation.

IV. \textbf{Validation}

In this section we deal with the question raised in the introduction: how realistic are the circuits produced by \texttt{gen}? We judge the quality of the generated circuits with respect to parameters not specified in generation: reconvergence, and post-placement and routing wirelength and track count. Since one of the primary applications of the circuits produced by \texttt{gen} is to test and evaluate physical design algorithms, the point of this exercise is largely to determine how reasonable the output is for this process. We note that a validation process for other characteristics such as node activity in simulation could also be performed; we leave this for future work.

We constructed the exact profile of 42 combinational \texttt{MCNC} circuits\footnote{There are actually 109 combinational circuits in the \texttt{LGSY} benchmark suite, but the majority are too small to be useful. We have restricted the experiments to circuits with 100 \texttt{LUTs} or more.} with \texttt{circ} (i.e. \texttt{n}, \texttt{nPI}, \texttt{nPO}, \texttt{d}, \texttt{shape}, \texttt{fanout} and edge length distributions), and generated corresponding circuits meeting those profiles with \texttt{gen}. Our method of validation is to compare unspecified characteristics of the \texttt{MCNC} circuits against those of the corresponding generated circuits and against “random graphs” of the same size.

Because the exact definition of a random graph varies, we now have to be precise: the most common usage of the term refers to a graph $G(n, p)$ on $n$ vertices with each possible edge existing with equal probability $p$. However, this is so drastically unlike a real circuit ($G(n, p)$ would usually be hopeless to route for even small $p$) that we have found it a more reasonable comparison to use a random $k$-regular graph—a random directed graph such that each node $x$ has $\text{fanin}(x)+\text{fanout}(x)=k$—as these graphs are more realistic in an electrical sense and are relatively easy to generate uniformly \cite{[10]}. We will compare against circuits mapped to 4-\texttt{LUTs}, and so we will use, for each circuit, the appropriate $k \in \{4, 5, 6, 7\}$ to generate approximately the same number of edges. Two drawbacks of this method are that random $k$-regular graphs have an inordinate number of $1/0$s (approximately 20\% of nodes) and no high fanout nodes, but they provide a convenient comparison to non-parameterized random generation. Earlier work using random graphs to test algorithms \cite{[15], [16]} used a similar

Fig. 9. A \texttt{gen} circuit family ($\{k=2; n=70, 80, 90, 100\}$ by 10).

Fig. 10. A \texttt{gen} clone script for \texttt{alu4}, output by \texttt{circ}.

inside the set brackets indicate modifications to parameters in \texttt{comb.circ} which override the defaults. Figure 11, in contrast, shows a user-defined \texttt{gen-script} to create a 1000 \texttt{LUT} circuit. Note that all unspecified parameters (shape, edges, etc.) are chosen from default distributions which use the specified circuit parameters such as delay and \texttt{nPI} as input parameters themselves.

F. \textbf{Time complexity of the \texttt{gen} algorithm}

The theoretical time complexity of the algorithm and its \texttt{gen} implementation is the larger of $O(d^2)$ from Step 1 and $O(n \log n)$ from other steps. In practice, we assume that $d \ll n$, so the complexity reduces to $O(n \log n)$. Each step in the algorithm addresses each element a constant number of times in processing for a linear factor, with possible constant number of pre-processing sorts or the creations of a random permutation, each of which takes $O(n \log n)$ time. The algorithm uses a constant amount of space per node, hence $O(n)$ for the algorithm.

Fig. 11. A simple user-generated \texttt{gen}-script for a 1000 node circuit.
generation process.

to the two factors mentioned earlier: the absence of high-fanout nodes and the large number of I/Os. Thus any generator which does not take these factors into account will fail to emulate crucial behaviour of real circuits.

B. Validating Routability

To test the “routability” of our output circuits, we used a locally available tool, VPR [2], to place and global route the sets of MCNC circuits, generated circuits, and random graphs described above. The circuits are compared on two different metrics: the maximum number of tracks per channel required to successfully route, and the total wavelength of the global routing.

VPR [2] chooses a minimal square grid to support the size of the circuit, and minimizes both maximum track-count per channel and total wavelength (by re-routing with successively fewer tracks per channel until failure occurs).

Table I also shows the routing statistics for the MCNC circuits, clones, and random graphs with summary percentages (percentage pairwise differences) on the last line. We see that the track count for the generated circuits differed by 14%, on average, from the corresponding MCNC circuit, whereas the random graphs differed by 123%. Wavelength differed for 17% for the generated circuits and 119% for random graphs.

For both track-count and wavelength, we note that the variation for GEN clones lies in both directions whereas random graphs were universally harder to place and route. Thus, the *signed* differences for the GEN clones were only 3% in track-count and 10% in wavelength, meaning that the difference speaks as much as the variance of GEN circuits as to an inherent specification bias. The random graphs, on the other hand, showed an obvious and consistent bias.

These results clearly show the circuits produced by GEN are very similar to the MCNC originals and significantly more realistic than random graphs as benchmark circuits.

C. Locality parameter revisited

It is important to point out that the locality parameter of generation is crucial in the above results. If the GEN circuits are created with a locality parameter of 1 (i.e. no locality), we find wavelength and track-count results which are about 70% above the original circuits on average. Similarly, a locality parameter that is too high for the given \( n \) can result in circuits which are all easier to place and route than the originals. Since the goal is to generate circuits which are as similar as possible to real circuits, the defaults are tuned to generate circuits which are similar on average to the original circuits. In these experiments a constant locality parameter, \( L = 6 \), was used.

This discussion further underscores the need for a characterization of locality which can tie the original circuit to its GEN clone, in order to reduce this variance.

V. EXAMPLES

For smaller circuits, we can observe the output of GEN pictorially.
A. Gen circuits from defaults

Figure 12 shows four different circuits produced by GEN using the default parameter distributions. We note that these circuits appear to be "normal" circuits, and include many features such as areas of high-fanout. The visual "quality" of the circuits is most striking when one observes the similarity to MCNC circuits, shown in Figure 13, and the contrast between MCNC circuits and the random graphs shown in Figure 14.

![Fig. 12. Varied circuits produced by GEN, using defaults.](image)

![Fig. 13. MCNC circuits sqrt8 and sa02.](image)

![Fig. 14. Random 4-regular digraphs](image)

B. Gen clone-circuits

Figures 15 and 16 show two MCNC circuits, each original circuit pictured with two different clones generated from its characterization by CIRC. Notice that the clones have a similar structure in terms of the parameters defined in this paper, but are different in the implementation of that structure, just as they are different from the original.

![Fig. 15. MCNC circuit sqrt5 and two clones from GEN.](image)

![Fig. 16. MCNC circuit sqrt8ml and two clones from GEN.](image)

VI. Concluding Remarks

In this paper we have introduced a new method for generating realistic parameterized combinational benchmark circuits. The circuit generation is derived from the measurement of a number of new graph-theoretic properties of digital circuits which we propose in this paper. As a result the circuits are much more realistic than random graphs. It has been shown that the quality of the circuits (as measured by reconvergence and routability) is comparable to an existing benchmark set and much better than that of random graphs that don't use these properties. Because of the close tie between characterization and generation, users are able to characterize their own circuits using CIRC and create defaults which more closely meet their own needs (rather than the MCNC defaults).

Using this method, we can generate a large set of circuits with the properties of the largest MCNC benchmark circuits. It remains to be seen if even larger circuits (which could easily be generated, just not as clones) have realistic circuit behaviour.

The GEN algorithm is fast, requiring less than 1 minute of SUN Sparc4 time to produce a circuit with 30,000 4-LUT nodes. The binary and source-code is freely available [19]. The output format for GEN and the input format for CIRC
is BLIF [23]. Circ can translate BLIF to a number of other netlist formats, such as Xilinx XNF, Altera AHD/L/TDF, Actel ADL, and a subset of Verilog.

In the future we will expand the GEN system to generate sequential circuits (with flip-flops, back-edges and cycles) [12] and to join sub-circuits together hierarchically. We also hope to add the ability to generate regular (datapath) structures and introduce LUT functionality so that we can apply our circuits to logic synthesis as well as physical-design problems. The most important area for further exploration is to determine justifiable models of locality in base level circuits which can be both measured and generated.

Acknowledgments. Thanks to Stephen North and AT&T Bell Labs for academic license to use DOT[8] and Vaughn Betz for the use of his place-and-route software VPR[2].

References


