

# A System-Level Stochastic Circuit Generator for FPGA Architecture Evaluation

Cindy Mark, Ava Shui, Steven J.E. Wilton  
Department of Electrical and Computer Engineering  
University of British Columbia  
Vancouver, B.C., Canada  
{cindym,steve}@ece.ubc.ca

**Abstract**—We describe a stochastic circuit generator that can be used to automatically create benchmark circuits for use in FPGA architecture studies. The circuits consist of a hierarchy of interconnected modules, reflecting the structure of circuits designed using a system-on-chip design flow. Within each level of hierarchy, modules can be connected in a bus, star, or dataflow configuration. Our circuit generator is calibrated based on a careful study of existing SoC circuits. We compare our circuits to those generated by previous circuit generators, and characterize our circuits with respect to the type of network used to connect modules.

## I. INTRODUCTION

Recent years have seen a tremendous increase in the density of Field-Programmable Gate Arrays (FPGAs). In the past twenty years, the density of FPGAs has increased by 200x [16]. This increase in density has led to new architectures; modern FPGAs look significantly different than their early predecessors. The routing architecture has evolved from a simple gridded network to include a variety of segment types, neighbour-to-neighbour connections, and multi-mode switches to reduce power. The logic architecture now consists of "fracturable" lookup-tables arranged in large tightly-connected clusters. This continued evolution of FPGA architectures is essential to support the increasing computation requirements of digital systems.

The development of these new architectures often employs an experimental methodology in which a potential architecture is modeled, benchmark circuits are mapped to the architecture, and detailed models are used to evaluate the density, speed, and power of the implementation [2]. A critical part of this experimental methodology is the set of benchmark circuits. These circuits must be representative of the circuits that will eventually be implemented on the FPGA. However, most benchmark suites used today are more representative of the glue logic circuits that were targets of early devices. Circuits from the Microelectronics Center of North Carolina (MCNC) have become almost ubiquitous in recent publications, yet even the largest of these circuits contains only 7694 logic blocks, which is approximately 2.3% of the largest available Altera Stratix III. Other benchmark circuits are available (eg. [3]), however, even these are significantly smaller than will be implemented in future devices. Commercial vendors have large databases of circuits, but also report that obtaining circuits

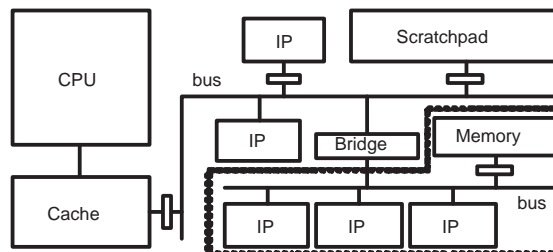


Fig. 1. Example system with two hierarchical levels

representative of those that will be implemented in next-generation parts is a challenge.

A potential solution is to use stochastically-generated benchmark circuits [6], [9], [10], [11], [15]. Typically, these circuits are created using a *circuit generator* which randomly creates netlists according to constraints that ensure the netlists share many of the structural characteristics of real circuits. Although these circuits are not "real", this approach has a number of advantages: an FPGA architect can generate as many circuits as desired, the circuits can be of any size, and often the generator can be further tuned to create only circuits with certain properties (eg. dataflow circuits in [11]). This latter advantage is critical during early architectural evaluation, when it is important to understand *what types* of circuits work well and what types do not work well. A generator that can create structures with a particular property can be an invaluable aid as new architectures are uncovered and evaluated.

These stochastically-generated circuits, however, must be realistic. Most existing generators build circuits using a "bottom-up" methodology using predetermined structural parameters and/or Rent's Rule. The resulting circuits realistically model glue-logic control or dataflow circuits. However, circuits implemented on today's FPGAs are typically entire systems consisting of processors and IP blocks connected using buses and on-chip networks. Often, the blocks contain several sub-blocks also connected using buses or networks; an example system is shown in Figure 1. It is unclear how well circuits generated using previous techniques reflect systems designed using such hierarchical system-on-chip (SoC) techniques.

In this paper, we present a *circuit generator which creates*

TABLE I  
BLOCK TYPE DISTRIBUTION

Type	Ratio
Processor	12%
Interface	39%
Controller	24%
Miscellaneous	25%

*circuits that better reflect the system-level circuits that are being implemented on today's FPGAs.* The generator creates circuits by combining blocks using on-chip buses or networks. The generation is done hierarchically; the blocks can themselves consist of subblocks connected by buses or networks. At the lowest hierarchical level, the blocks can be obtained from existing benchmark suites (such as MCNC circuits), or from previous circuit generators. Key parameters such as the number of hierarchical levels and the number of blocks on each level can be given as a constraint to the generator, allowing the user to generate circuits that reflect current and future system-on-chip circuits. Our generator is freely available to the research community.

To ensure our circuits are realistic, we started with a careful study of the block diagrams of industrial and academic circuits in recent publications to determine the composition and structure of modern circuits. Important parameters were extracted, and used to guide the development of our generator. To validate our generation process, we compare post placement results of our generated circuits with previous techniques and demonstrate that our circuits lead to more realistic architectural conclusions.

This paper is organized as follows. Section II describes previous circuit generators and Section III outlines the terminology we use to describe circuits. Section IV describes our analysis of existing circuits, and shows how we extracted key parameters from this analysis. Section V then describes how we used this information to create our circuit generator. Characterization of the resulting circuits and validation of our technique is described in Section VI, and the limitations of our generator are described in Section VII. Finally, Section VIII concludes and indicates how our generator can be downloaded.

## II. BACKGROUND

There have been several earlier attempts to generate synthetic circuits. Most follow a “bottom-up” approach in which logic structures with typical circuit characteristics are constructed. Hutton [10], [9] and later Kundarewich [11] describe how circuits can be generated using parameters such as the number of logic levels and average fanout. Hutton provides a program which extracts this information from an existing circuit; this information (or a subset of it) can then be used to generate a new “clone” circuit. Stroobandt describes a generator which can create circuits that match a target piecewise Rent parameter [15]. These generators were constructed and evaluated assuming a homogeneous circuit structure; as described earlier, this may not match the structure of very large system-level circuits.

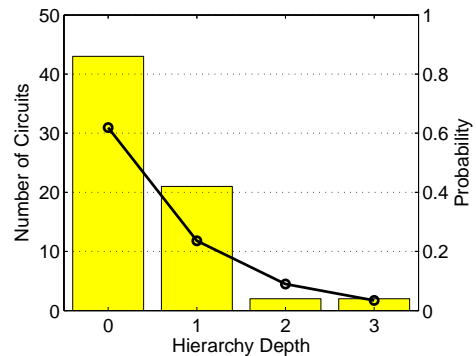


Fig. 2. Hierarchy depth distribution

Pistorius’s work in [14] is the most similar to our own. He intuited that circuits are composed of several different kinds of logic: regular and combinational, irregular and combinational, memory, controller, and interconnection. By varying the proportion of these various types of logic, this generator should be able to mimic different kinds of circuits in a realistic manner. Our work differs on the scale at which we address circuits. He attempts to join together sections of logic on a per bit basis into primary circuits, whereas we join blocks on a per word basis into systems.

Another method to generate circuits is the mutation approach. The most recent such work is [6], others include [7], and [5]. Portions of the logic are modified, but structural characteristics such as path length, I/O, and wirelength are kept the same. This method is effective at generating a family of circuits similar to an existing circuit, but they lack the ability to generate new circuits of different size or of different structure.

## III. TERMINOLOGY

To characterize a circuit, we follow the approach in [13]. A circuit is a *tree of modules*. Each module can contain a number of lower-level modules. Within a module, the lower-level modules are typically connected using a bus, a network-on-chip (NoC), or arranged in a dataflow configuration. We will refer to the interconnect structure within a module as that module’s *network*. The lowest-level modules will be referred to as *leaf modules*; leaf modules can be processors, memories, or other sequential or combinational logic circuitry. In the example of Figure 1, the top-level module consists of five lower-level modules: a CPU/cache, a scratchpad memory, two IP blocks, and one child module, all connected using an on-chip bus (all but the lower-level bus are leaf modules). The lower-level module contains three IP blocks and one memory module connected using a bus. The bridge module is considered part of the bus implementation and thus is not counted. Typically, such a circuit contains additional connections (such as interrupt request lines or reset signals) that do not follow the hierarchy.

TABLE II  
AVERAGE NUMBER OF NETWORKS PER LEVEL

Max Depth	Average
0	1
1	1.81
2	1.75
3	2.16

#### IV. PARAMETER EXTRACTION

##### A. Overview

To calibrate our circuit generator, we performed a detailed analysis of 66 different SoC circuits. Of these, 42 were academic designs gathered from conference proceedings from 2004 to 2008. The remaining 24 were from industrial sources. The chips spanned a wide range of applications including network communications, multicore processors, and multimedia. We did not have detailed designs for these chips, so our analysis was limited to what could be gleaned from block diagrams and the accompanying text. This section highlights some of the key results that were used to calibrate the generator.

##### B. Leaf Modules

For each circuit, the leaf modules were identified. Table I categorizes the leaf modules into four types. *Processor* circuits can be either CPUs or GPUs. *Interface* blocks, for example UART interfaces, are used for simple data transfer between the block. *Controllers*, such as memory controllers, are often built as a finite-state machine (FSM). Some blocks that manage more complex data transfers such as USB controllers also fall into this category. The remaining blocks are classified as *miscellaneous* and range from image processing circuits to custom purpose circuits.

The distribution in Table I shows that all four types occurred commonly in our circuits.

##### C. Circuit hierarchy

For each circuit, we identified the hierarchy shown in the block diagram. In some circuits there are two separate datapaths (in multimedia circuits, for example, there are often separate datapaths for audio and video). We broke these circuits into two designs and analyzed them separately. By this definition, our analysis includes 79 different designs. As described earlier, some connections do not follow the hierarchy of the circuit; those connections were ignored for this part of the analysis.

Figure 2 shows the depths of the hierarchies we observed. A depth of 0 means there is a single network connecting a collection of leaf modules. In our circuits, the maximum depth was 3, however, this does not include any hierarchy which may be present inside the leaf modules (and hence are not shown on the block diagrams we examined). Thus, the true hierarchy depth may be more than what is shown in Figure 2.

Note that the solid line in Figure 2 (and subsequent graphs) will be discussed in the next section.

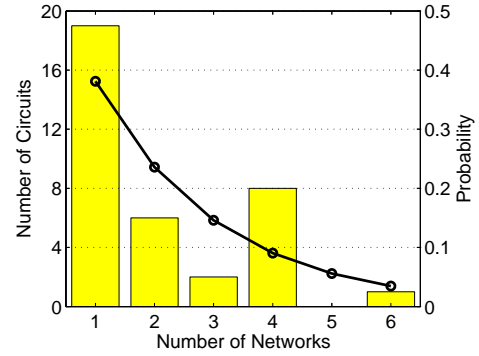


Fig. 3. Number of nets at level 1

TABLE III  
NETWORK TYPE DISTRIBUTION

Type	Ratio
Bus	53%
Dataflow	33%
Star	7%
Other	7%

We next examined the number of networks at each level of the hierarchy. By definition, Level 0 has one network that connects a collection of modules. Table II shows the average number of networks at each hierarchical level, and Figure 3 shows the distribution of the number of networks for Level 1. In a perfectly balanced binary tree, the number of networks per level would grow exponentially with the level number. The results in Table II show that this is not the case in our circuits; the number of networks for levels larger than one remain roughly constant. This is because many of our circuits contain only a few modules that contain sub-modules, making the hierarchy far from a complete tree.

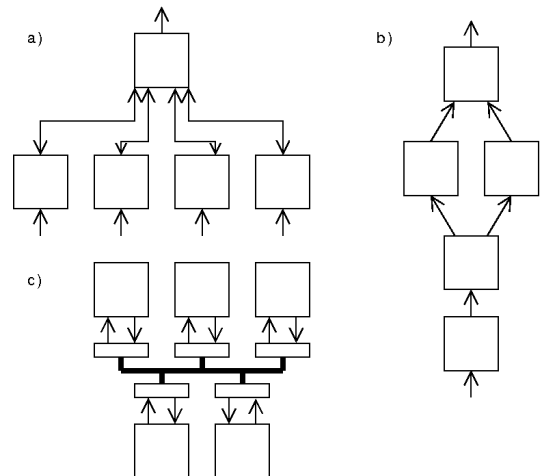


Fig. 4. Block diagrams of three network types a) Star b) Dataflow c) Bus

#### D. Network types

Within each module is a network that connects lower-level modules. We found three common connection patterns in our circuits. We will refer to these patterns as *bus*, *dataflow*, and *star*.

Figure 4(a) shows an example of a star connection pattern. This sort of pattern was commonly found in simple embedded controller chips, and is characterized by a bidirectional flow of data between the parent and the children blocks. Figure 4(b) shows a dataflow pattern, and is characterized by a unidirectional data flow between modules. The dataflow pattern was commonly seen in multimedia applications, and reflects the pipelined nature of many of these circuits. Figure 4(c) shows a bus interconnect structure, in which all sub-modules are connected to a common bus using an (often industry-standard) access protocol.

The distribution of these network types across all of our benchmark circuits is shown in Table III.

Finally, we examined the size of each network. Figure 5 shows the distribution of the number of leaf modules per network for each of our three network types. Buses are generally large since the complexity of implementing an access protocol is high, but remains relatively constant relative to the number of blocks. A dataflow typically has several blocks arranged in sequence; short pipelines are relatively rare. Star networks often have a limited number of blocks because of the communication limits of this topology.

Other network types (such as packet-based gridded networks) are possible, however, since they rarely occurred in the circuits we examined, we can not present data on them.

#### E. Correlation

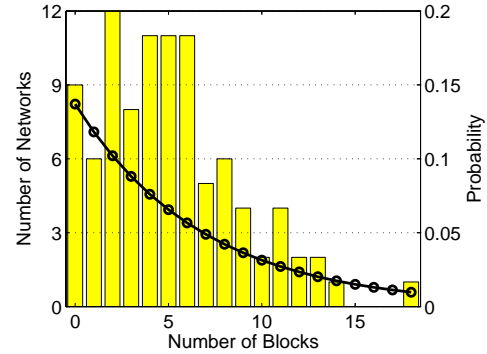
In the preceding subsections, we presented data regarding the hierarchy, module types, and network types. Although it would be possible to gather additional information on the correlation between these parameters, we do not feel we have enough circuits to draw meaningful conclusions from this sort of analysis.

### V. CIRCUIT GENERATION

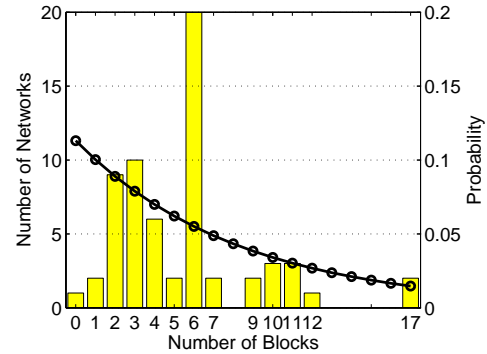
#### A. Overview

Our circuit generator creates random system-level circuits according to a set of constraints supplied by the user. While the shape of the distributions is coded into the algorithm, the slope or range for all non-uniform distributions is described in the constraints. This allows the researcher to easily modify the distribution parameters in order to tune the generation process for a particular application.

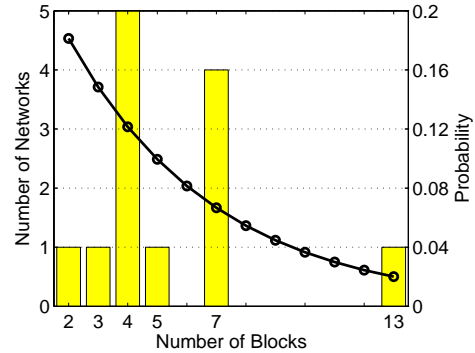
The probability distributions used in the generation are characterized from the information found during the analysis of current circuits. For simplicity, only exponential, weighted, and uniform distributions were used. The distributions were fitted to the data using Excel's regression tools. The resulting parameter values are listed in the following subsections and are used as defaults in the constraints file.



(a) Bus network



(b) Data network



(c) Star network

Fig. 5. Distribution of the number of leaf modules for each network type

As shown in Figure 6, inputs to the generator are a constraints file that controls the stochastic generation and a library of circuits that will be the content of the leaf modules. The generator outputs the netlist in the BLIF format (which can be directly read by the VPR place and route tool).

The library of circuits is divided into four parts: processor circuits, interface circuits, control circuits, and miscellaneous circuits; these types correspond to our observed categories shown in Table I. In our implementation, we use MCNC circuits as leaf modules, however, these circuits can also be generated stochastically using a generator such as those described in [6], [9], [10], [11], [15].

Generation of circuits takes place in three steps. First any missing primary parameters are generated automatically.

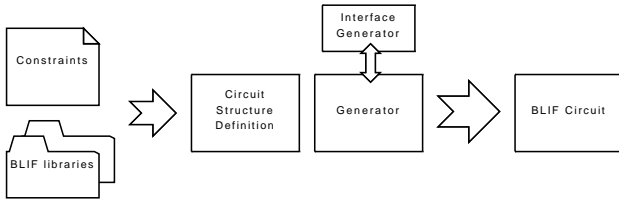


Fig. 6. Circuit Generation Process

These parameters broadly describe the circuit structure to be generated. Next, secondary parameters are calculated. At this stage, the hierarchy of the circuit is known. Finally, all of the bit level connections are made and the circuit is generated from the specifications.

### B. Primary Parameters

The primary parameters set the hierarchy depth, the total number of networks, and the total number of modules in the circuit. The user can set any combination of these values and any missing parameters will be supplied by the algorithm using the parameters described in the constraints file. This section explains the methods used to select the values for the missing parameters.

First, the hierarchy depth is selected randomly using an exponential distribution with a default lambda value of  $-0.97$  (the solid line in Figure 2). The number of networks is then generated by estimating the number of networks for each hierarchy level using an exponential distribution with default lambda  $-0.48$  as shown in Figure 3. The total number of modules depends on the the type of each network (as shown in Figure 5). Thus, before calculating the number of modules per network we need to select the network type. We first select the number of network *types* allowed in this circuit, then for each network, select one of the available types using the weighted distribution in Table 5. The reason for selecting the number of network types first is that if we did not, the set of network types in the circuit would usually approach the global average, reducing the diversity in the generated circuits.

The final primary parameter is the bus width (for circuits which contain at least one bus). If this parameter is not specified by the user, a power of two value is selected randomly between and including 16 and 256.

### C. Secondary Parameters

In this phase, the circuit elements described in the primary parameters are arranged into a tree structure in accordance to our model, and the final details of the circuit structure are determined. The parameters in this section cannot be controlled directly by the user, but they can be tuned by the probability distributions described in the constraints file. Unless otherwise specified, random selection implies a uniform probability distribution.

First, one network is assigned to the top level (level 0). The remaining networks are assigned randomly to the subsequent levels. Each network is the randomly assigned to a parent network on the previous level.

```

Generate_circuit {
  /* Generate all subcircuits */
  for each module {
    if this module is a leaf
      glue selected subcircuit from library
    else
      generate_circuit(module)
  }
  /* Generate connection between modules */
  case (network type) {
    BUS:
      randomly select one module to be master
      for each module {
        generate the bus interface and connect it to the module
      }
      wire address and data buses to all modules
    DATAFLOW:
      for each module
        Connect the inputs to the outputs in the immediately previous stage
      for each module
        If there are extra inputs, connect to blocks from any previous stage
      for each module
        If there are extra inputs, connect to blocks from any subsequent stage
    STAR:
      if the sources from the centre block exceed the total number of sinks
        distribute the pins among the child blocks
      else
        fanout each pin to all the child blocks while they have empty sinks
      if the sinks in the centre block exceed the total number of sources
        connect the child blocks to the centre
      else
        connect an equal number of inputs from each child to the centre block
  }
  connect any empty sources or sinks to external pins
}
connect the interrupt nets
connect the reset net
connect the clock net
  
```

Fig. 7. Pseudocode for construction of circuit

Each network is allocated a random proportion of the modules based on the network type and their level in the hierarchy. For dataflow networks, the maximum length and width of the pipeline is determined and then the modules are allocated to the dataflow stages such that they fit those constraints.

Next, a module type (processor, interface, controller, or miscellaneous) is selected for each module. The probability for the types is based on Table I, but it can be adjusted depending on the module's location in the hierarchy. CPUs are assumed to be found more likely at the top levels while interfaces and controllers are more likely at the lower levels. Similarly, CPUs are assumed to be found more likely in the middle of dataflow networks, and interfaces and controllers more likely at the ends of the dataflow networks. For each module, a block from the appropriate library directory is then selected randomly.

Finally, a module is chosen at random to be the reset source, and a module is chosen at random to be the interrupt sink. If necessarily, a module is chosen to be the bus master.

### D. Circuit Construction

Once parameter values are selected, the circuit is constructed recursively from the bottom of the hierarchy using the algorithm in Figure 7.

The heart of the algorithm is the manner in which the sub-modules are glued together. Each network type is constructed differently. A bus structure is constructed by first generating a custom bus interface for each module then gluing the pair together to form a node. Then the address and data buses are connected using a multiplexed configuration. Our interfaces are implemented using a simplified version of the AMBA AHB/APB single master specification [1] which was the most common type of bus represented in our survey. The other

TABLE IV  
COMPARISON BETWEEN GENERATORS

No	Number of LEs	Rent Parameter			Number of Nets			Avg. Net Length			Channel Width			Crit. Path Delay (ns)		
		GEN	GNL		GEN	GNL		GEN	GNL		GEN	GNL		GEN	GNL	
1	5687	0.71	0.81	0.80	3893	4128	5901	13	30	18	28	90	46	15.54	25.24	44.39
2	6436	0.69	*0.80	0.79	4399	*4885	6725	16	*28	18	40	*74	52	20.54	*27.64	46.40
3	7425	*0.71	*0.82	0.80	*6077	*6257	7855	*19	*27	19	*32	*110	54	*41.32	*24.34	41.43
4	8001	*0.71	0.80	0.80	*6171	5843	8426	*14	30	19	*36	82	56	*39.76	38.46	48.91
5	8468	0.71	0.80	0.81	6673	6553	9009	13	26	19	38	80	54	39.61	30.50	52.77
6	8967	0.72	0.84	0.81	6072	7678	9465	15	39	20	36	124	60	18.64	37.51	48.10
7	10937	0.71	0.67	0.82	7116	5660	11536	18	24	21	40	44	64	32.13	27.64	61.08
8	11643	0.73	0.82	0.82	8509	8917	12732	15	47	24	38	90	70	38.96	56.93	59.77
9	13012	0.71	0.81	0.82	10619	9003	14004	21	39	22	36	84	64	42.21	37.93	60.35
10	13790	0.73	0.83	0.82	10185	10621	14353	15	39	24	38	122	70	37.67	38.10	65.45
11	14099	0.74	*0.82	0.82	9893	*10430	15078	15	*46	24	38	*86	74	39.33	*49.05	66.08
12	14538	0.73	0.82	0.82	10306	12994	15512	14	38	23	38	126	70	39.99	54.21	64.88
13	15778	0.75	0.82	0.82	11474	11900	16176	18	51	24	50	106	74	37.21	43.56	67.97
14	18550	0.75	0.81	0.83	13127	14622	19507	14	34	25	38	128	76	38.73	48.70	71.57
15	19915	*0.74	0.81	0.83	*14877	13216	21196	*17	39	27	*40	86	80	*38.86	73.61	82.98
16	27577	*0.76	0.75	0.84	*20311	16701	28686	*19	36	29	*42	72	90	*40.29	80.77	100.53
17	37466	*0.78	0.65	0.85	*27902	17021	38452	*23	-	-	*44	-	-	*46.90	-	-
18	42261	*0.77	0.81	0.85	*30496	26900	51220	*18	-	-	*52	-	-	*45.77	-	-
19	46255	*0.77	0.81	0.84	*31913	26219	43898	*18	-	-	*42	-	-	*55.67	-	-
20	50275	0.77	0.78	0.85	37494	27600	49034	28	-	-	72	-	-	55.03	-	-
21	56296	*0.78	0.81	0.82	*41366	33108	57989	*20	-	-	*62	-	-	*42.62	-	-
Avg first 16		0.73	0.80	0.82	9356	9338	13510	16	36	23	38	94	66	35.05	43.39	61.42

Starred entries are pin-limited. Empty entries did not complete after 72 hours of place and route.

networks are based on patterns we observed during our circuit analysis.

Typically, an SoC contains at least one reset signal which is distributed to all modules, and interrupt lines from each module to a controlling processor. After constructing the entire circuit, we connect a single line from the reset source to all modules. We also construct an interrupt network within each module by ORing an interrupt signal from each sub-module. The top-level interrupt signal is connected to the selected interrupt sink module. Finally, a single clock is connected to all sequential elements in the circuit.

## VI. VALIDATION AND CHARACTERIZATION

In this section, we validate and characterize the synthetic circuits generated by our algorithm. To validate our circuit, we compare the post-routing results of our circuits versus the results from circuits generated by Gen and GNL, and demonstrate that our circuits lead to more realistic architectural conclusions. We then show the sensitivity of these statistics to the type of network used in the circuit.

### A. Comparison to Previous Circuit Generators

We first generated 21 circuits of varying sizes using our generator. Some of the default values in the constraints file were modified in order to quickly obtain circuits of the right size, and to match the characteristics of the MCNC circuits. We left the hierarchy and composition of the circuits unspecified. The leaf modules in the library included 61 MCNC circuits and 2 circuits from the OpenCores website; each leaf circuit contained between 28 to 7694 logic elements.

Table IV shows statistics regarding these circuits. The sizes of these circuits ranged from 5687 to 56296 logic elements (Column 2) and the Rent parameters (averaged over each

circuit) ranged from 0.69 to 0.75 (Column 3). The Rent parameters were computed using a recursive Fiduccia-Mattheyses partitioning algorithm.

We also generated comparably sized synthetic circuits using two previous generators, Gen [8] and GNL [17]. For each of our original synthetic circuits, we generated one circuit using Gen and one using GNL. The number of logic elements for each Gen and GNL circuit was constrained to be the same as the corresponding original synthetic circuit. The GEN circuits were strictly combinational while the GNL circuits were sequential. Based on experience, each GNL circuit was constructed using a Rent parameter of 0.7 for the first 3000 nodes, and 0.66 for the rest of the circuit. In addition, we specified a ratio of 2:1:1 for 2, 3, and 4 terminal logic elements respectively.

Statistics for each Gen and GNL circuit are shown in Columns 4 and 5 of Table IV. As shown in the table, the Rent parameter of the GEN circuits is considerably higher than most of our circuits. Experience has shown that Rent parameters of 0.6 to 0.7 are reasonable. The Rent parameters in the Gen circuits tend to be higher than this, primarily because Gen was designed and calibrated assuming smaller benchmark circuits than the ones we are generating. Though we specified a rent parameter of 0.66 for the GNL circuits, the average rent value is 0.82. Fixing the number of IO pins had little effect on the rent parameter. We believe this deviation is caused by the ratio of multiterminal logic elements in the GNL circuits.

We then mapped each benchmark circuit to a minimum-sized FPGA using T-VPACK and VPR 5.0 [12]. We assumed a clustered architecture in which each cluster contains four 4-LUTs and has 10 inputs and 4 outputs. A uni-directional routing architecture with single-length segments was assumed.

Columns 6 to 8 of Table IV show the number of nets in the

TABLE V  
COMPARISON BETWEEN NETWORK TYPES

	Average			Std. dev.		
	Bus	Dataf.	Star	Bus	Dataf.	Star
Number LEs	18019	14839	14839	14065	11873	11873
Number I/Os	263	631	765	197	420	761
Rent Param.	0.737	0.723	0.726	0.033	0.040	0.038
Num. Clusters	4559	3667	3656	3555	3058	3044
Number Nets.	12758	10712	10724	10198	9047	9094
FPGA size	63x63	71x71	76x76	26x26	55x55	62x62
Crit. Path(ns)	34.1	33.6	31.6	16.3	16.1	14.8
Wirelength	14.5	16.9	17.9	2.4	4.9	5.6
Chan. Wid.	38.5	40.0	40.0	9.6	10.0	8.7

packed circuits. As the table shows, the number of nets in the GNL circuit is somewhat higher than that in the other circuits. We have observed that, in the GNL circuits, T-VPACK is less able to share inputs between LEs in a cluster, and thus fewer nets can be absorbed into clusters during packing.

Columns 9 to 11 of Table IV show the average net length of each circuit. The net length is measured in terms of the number of CLBS the net spans. As the table shows, the average net length is significantly higher in both the GNL and the Gen circuits than in our circuits. The higher net length in the Gen circuits is primarily due to the larger Rent parameter of these circuits (which, as explained above, is primarily due to the fact that Gen was designed and calibrated assuming smaller benchmark circuits). The longer net length in the GNL circuits can be explained by observing that because T-VPACK is less effective at sharing inputs, the average fanout in the GNL circuits is somewhat larger than the other circuits. Our experience has shown that average net lengths for multi-terminal nets in the range of 11 to 15 is reasonable.

Columns 12 to 14 of Table IV show the minimum channel width required to route each circuit. As expected, the higher average wirelength in the GNL and Gen circuits leads to a higher average channel width (an equation describing this relation is presented in [4]).

Finally, Columns 15 to 18 of Table IV show the critical path delay of each mapped circuit. Since the Gen circuits are combinational, the longer critical path is expected. The GNL circuits also have a long critical path; The GNL result is surprising since it explicitly includes latches to control the logic depth yet it still have the longest critical delay. As described earlier, T-VPACK is less effective at sharing inputs when packing the GNL circuit into clusters. This means that a larger proportion of the critical path is made up of inter-cluster nets.

The post-routing results generated by our circuits lead to more sound architectural conclusions which suggests that these circuits are more realistic than those generated by Gen and GNL. Unfortunately, we can not compare these statistics to those obtained from real benchmark circuits, since we do not have circuits of the appropriate size.

### B. Network Characteristics

To investigate the sensitivity of these statistics to the type of network used in the circuit, we used our generator to create 11

sets of new circuits. Each set contains three circuits, all three circuits within a set use the same hierarchy and leaf modules; they differ only in the type of network used (bus, dataflow, or star).

Table V presents statistics on the circuits and the implementation of these circuits on a minimum-sized FPGA. For each statistic and each network type, the table presents the average and standard deviation across all 12 sets.

The results show that the bus-based circuits are 25% larger than the others. This is because the bus-based circuits require bus interface modules that coordinate transfers on the bus. The dataflow and star-based circuits contain direct connections which do not require such interface circuitry.

The results also show that the dataflow and star-based circuits have significantly more I/O than the bus-based circuits. This points to an important limitation of our generator. When connecting circuits in a dataflow pattern, pins are created for the inputs of those modules connected at the “head” of the dataflow at the top level, and the outputs of those modules connected at the “tail” of the dataflow at the top level. Depending on the sizes of these modules, this could result in a large number of pins. Similarly for the star-connected circuits, pins are created for nets not involved in the star pattern.

It is interesting to note that the average wirelength of the bus-based circuits is less than that of the other circuits. This may seem counter-intuitive, since bus wires are expected to be long. However, this is outweighed by the fact that the connections between the interface and the leaf modules are short, direct point-to-point connections. There are far more of these wires than the wires that make up the bus itself.

## VII. LIMITATIONS

One limitation of our generator is the manner in which it handles I/O pins. As described in the previous section, a large number of pins are created when the top level contains a dataflow or star connection pattern. Our circuit generator is meant to be used in FPGA architectural experiments, and in most of these experiments, an excess of I/O pins will not be a problem. However, such a circuit is more likely to become pad-limited, relaxing the importance of effective packing and making routing easier. Thus, when using our circuits in FPGA architectural experiments, it is important to take note of whether a circuit is pad-limited, and if so, interpret the results appropriately.

A second limitation is that we do not create circuits containing memory. Embedded memory blocks are an important part of modern FPGAs. For FPGA architecture experiments involving the logic fabric only, this will not be a problem; however, to fully exercise all parts of an FPGA, memory should be included. It would be straightforward to add memory blocks as leaf modules; a more careful extension would consider common memory connection patterns such as those described in [18] and integrate these patterns into the circuit generator. In fact, memory blocks would help control the number of excess pins, since some pins on large blocks such as

CPUs would normally be expected to connect to an exclusive memory cache.

Although we do not currently generate circuits with embedded hard blocks, it would only require a change to the circuit utility functions to allow leaf modules that contain hard blocks. This would not require modifications to the generation algorithm described here.

A third limitation is that future very large SoC's may contain more elaborate network-on-chip structures. Packet-based gridded networks are emerging, and these structures could be included in our generator. We anticipate that these structures may contain a large number of point-to-point connections; our bus results from the previous section suggest that this may influence the average wirelength and channel width of the resulting implementation.

### VIII. CONCLUSIONS

In this paper, we have described a stochastic circuit generator that can be used to automatically create benchmark circuits for use in FPGA architecture studies. The circuits consist of a hierarchy of interconnected modules, reflecting the structure of circuits designed using a system-on-chip design flow. Within each level of hierarchy, modules can be connected in a bus, star, or dataflow configuration. To ensure our circuits are realistic, our circuit generator was calibrated based on a careful study of existing SoC circuits.

These stochastic circuits are an important tool in the arsenal of any FPGA architect. Suitable "real" benchmarks that are large enough to investigate future FPGA architectures are difficult to obtain. Even if they could be obtained, there is value in being able to generate a "family" of benchmark circuits that have a specific size or that have a specific property. This allows FPGA architects to study exactly what types of circuit structures are well supported by a proposed device, and what sort of structures are not supported well. We do not propose that synthetic circuits replace "real" benchmark circuits entirely, but used correctly, these synthetic circuits will enable FPGA architecture research that would otherwise be very difficult to perform.

The source code for our generator can be downloaded from <http://www.ece.ubc.ca/~cindym>.

### REFERENCES

- [1] ARM. *AMBA Specification*, 2 edition, 2001.
- [2] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.
- [3] Altera Corp. Quartus II University Interface Program (QUIP). <http://university.altera.com/research/unv-quip.html>.
- [4] W. M. Fang and J. Rose. Modeling routing demand for early-stage fpga architecture development. In *International Symposium on Field Programmable Gate Arrays*, pages 139–148, February 2008.
- [5] D. Ghosh, N. Kapur, J. Harlow, and F. Brglez. Synthesis of wiring signature-invariant equivalence class circuit mutants and applications to benchmarking. In *Design Automation and Test in Europe*, pages 656–663, February 1998.
- [6] D. Grant and G. Lemieux. Perturber: Semi-synthetic circuit generation using ancestor control for testing incremental place and route. In *International Conference on Field-Programmable Technology (FPT)*, pages 189–195, December 2006.
- [7] J. Harlow and F. Brglez. Synthesis of esi equivalence class combinational circuit mutants. Technical Report 1997-TR@CBL-07-Harlow, North Carolina State University, October 1997. Also available at <http://www.cbl.ncsu.edu/publications>.
- [8] M. Hutton. The circuit characterization and generation project at the university of toronto. <http://www.eecg.toronto.edu/~mdhutton/gen/index.html>.
- [9] M. Hutton, J. Rose, and D. Corneil. Automatic generation of synthetic sequential benchmark circuits. *IEEE Transactions on Computer-Aided Design*, 21(8):928–940, 2002.
- [10] M. Hutton, J. Rose, J. Grossman, and D. Corneil. Characterization and parameterized generation of synthetic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):985–996, 1998.
- [11] P. Kundarewich and J. Rose. Synthetic circuit generation using clustering and iteration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):869–887, 2004.
- [12] I. Kuon and J. Rose. Area and delay trade-offs in the circuit and architecture design of fpgas. In *International Symposium on Field Programmable Gate Arrays*, pages 149–158, February 2008.
- [13] E. J. Marinissen, V. Iyengar, and K. Chakrabarty. Itc'02 soc test benchmarks. <http://www.hitech-projects.com/itc02socbenchm/>, October 2007.
- [14] J. Pistorius, E. Legai, and M. Minoux. Partgen: A generator of very large circuits to benchmark the partitioning of fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1314–1321, 2000.
- [15] D. Stroobandt, P. Verplaetse, and J. Van Campenhout. Generating synthetic benchmark circuits for evaluating cad tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1011–1022, 2000.
- [16] S. Trimmerger. Keynote talk: Redefining the FPGA. In *International Conference on Field-Programmable Logic and Applications*, August 2007.
- [17] P. Verplaetse and D. Stroobandt. Gnl: Generate netlist. <http://trappist.elis.ugent.be/~dstrooba/gnl/>.
- [18] S.J.E. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.