Random Benchmark Circuits with Controlled Attributes

Kazuo Iwama* Kensuke Hino† Hiroyuki Kurokawa‡ Sunao Sawada*

*Dept of Computer Science and Communication Engineering Kyushu University, Japan

†Industrial Instrumentation & Control Systems Department, TOSHIBA Corporation, Japan

‡Research & Development, JUSTSYSTEM Corporation, Japan

e-mail : [iwama, sawada]@csce.kyushu-u.ac.jp

Abstract

Two major improvements, controlled fan-in and automated initial-circuit production, were made over the random generator of benchmark circuits presented at DAC'94. This is an important progress towards our goal of random benchmarking: more general and secure testing, increasing the naturality of random circuits by controlling their attributes, and obtaining test results by which the difference of performances under evaluation can be made clear.

1 Introduction

No one disagrees that how to select benchmarks is a key issue in evaluating empirically the performance of CAD systems, including logic optimizers discussed in this paper. For example, the MCNC benchmark set [11] includes more than 70 combinatorial circuits (and also other types of circuits). Those circuits are carefully selected with a wide variety and appear to have been accepted in the logic-design community. However, it is an unforgettable fact that only a finite number of particular circuits are involved. Natural concern is its generality: There is no way of proving that algorithms being good for the benchmarks are good for every circuit. We cannot even deny the possibility of cheating, or unnatural tune of algorithms for the benchmarks, either.

To relief this concern, the best way seems to add random test-instances, or *random circuits* in the present case, into benchmarks. This approach is already common in some other areas like graph algorithms [10] and combinatorial optimization [3]. However, generating "reasonable" random circuits is not easy. Critics always say that random instances are too artificial and far from the real world. It also suffers from the difficulty of controlling several features and attributes of instances. For example, suppose that random circuits are generated just as random graphs, i.e., by placing connections between gates at random. Then it turns out that by a good chance the resulting circuits realize the constant (0 or 1) logic function.

In [2], Iwama and Hino introduced random generation *by random transformation* of multi-level logic circuits, for the purpose of testing the performance of logic optimizers such as MIS [1], SIS [8] and Transduction Method [7]. The algorithm starts with an initial circuit and then applies a sequence of random transformations. Each transformation is selected at random from the set of transformation rules. Each rule is an equivalent transformation, namely, it does not change the logic function. Its single application can be done quickly (in polynomial time). The set of rules is complete, which means there exists a sequence of rules that changes any circuit to any equivalent circuit. Thus, many test circuits of different sizes can be generated from a (small-sized) initial circuit and then can be fed to the optimizers to test them.

Unfortunately, this generator was rather experimental and included two major problems: One is that we can control virtually no attributes of circuits such as the maximum fan-in of logic gates. Note that this is a consequence of the completeness of our transformation rules; our rules can generate any equivalent circuits which obviously include those using large-fan-in gates. So this problem is rather intended but it is also true that circuits in practice are usually fan-in restricted such as four. The second problem is that we did not mention how to select the initial circuit. One reasonable way was to select it from the benchmark set. However, it is clearly a demerit for the primary goal of random benchmarks, i.e., the variety of instances, since the generated circuits realize only a limited small number of logic functions. MCNC The objective of this paper is now obvious: We shall first show a complete set of transformation rules for circuits of NAND gates whose fan-in is up to k (k is a fixed integer ≥ 2). Note that this new finding of the complete set and its proof are much more nontrivial than it might look: Recall that what we wish to do is to transform a circuit, C_1, of max-k fan-in into any circuit, C_2, of the same restriction. Only to this goal, the rule set in [2] does work since it can change any circuit to any equivalent one. However, one can see that there can be intermediate circuits that violate the fan-in restriction in the course from C_1 to C_2. Our new rules never violate the restriction, namely, any application (sequence) of the rules produces circuits of max-k fan-in if the original circuit is also of max-k fan-in. Such a complete set of rules was not known with the best knowledge of the authors.

We also present a new procedure for random generation of the initial circuits. Generation of the initial

ED&TC '97 on CD-ROM
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for fee or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. © 1997 ACM/0-89791-849-5/97/0003/$3.50

circuit is almost the same as generation of a logic function. Our approach is to generate prime implicants at random. It can control basic attributes including the size of the on-set of the function and the degree of its complexity, which also meets the first requirement of controllable attributes. Now our generator produces a variety of random circuits from less than ten simple parameters, by which we can get rid of rather troublesome procedures of selecting the initial circuit and typing it into the computer.

As for the circuits of limited fan-in, experiments are at the primary stage, but the experimental data for SIS and Transduction Method show clear differences of their performances for particular circuits. As for the generation of initial random circuits, a fairly large amount of experimental data has been obtained. We tested SIS and Transduction Method using two types of circuits, one is generated by initial circuits followed by random transformations (MCNC-RT instances) and the other by random initial circuits and random transformations (R-RT instances). Roughly speaking: (i) R-RT instances appear to be much harder than MCNC-RT ones for both SIS and Transduction Method. The degree of optimization is close to one (the output of the optimizers is almost of the same size as the best possible) for MCNC-RT instances but is as large as three (three times as bad as the best possible) for R-RT instances. (ii) SIS with the algebraic script occasionally showed significantly bad performance in terms of the degree of optimization.

Finally in this section it should be noted again that we do not claim that random circuits should replace conventional (natural) benchmark circuits but we do claim that random circuits should be *added* to compensate what lacks in the conventional benchmarks. One side effect is that we get a lot more chance of checking incompleteness of the system using random benchmarks. Our above experiment shows that there were a good number of circuits for which SIS or Transduction Method failed to run (some 8% for SIS and 10% for Transduction Method).

2 Random Transformation

2.1 Algorithm

The algorithm for transforming an initial circuit into a test circuit is exactly the same as [2]. A circuit is described by a set of equations like

$$\begin{aligned} g[0] &= ((x_1, g[2]), (((g[1], g[2]), (x_3, (x_4, x_2))))), \\ g[1] &= (x_1, x_2), \\ g[2] &= (x_2, x_3), \end{aligned}$$

which denotes the circuit, say, C_0, of Figure 1. Here, each equation is called *the definition of a subcircuit*, say, $g[0]$.

In this paper, we assume the circuit uses only NAND gates. A *k-NAND circuit* denotes a circuit of NAND gates with fan-in up to k. A *transformation rule*, or simply a *rule*, is denoted by $f \Longrightarrow g$. A k-NAND circuit C_1 is changed to an (equivalent) circuit C_2 by applying one of the rules in the set $\Re_k$,

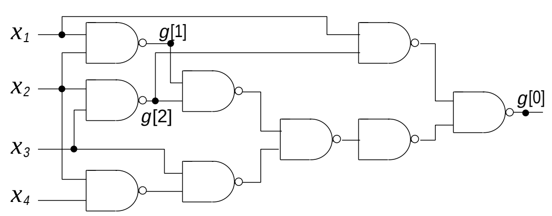

Figure 1: The circuit described by the equations

which will be given in Section 2.2. The following example would be beneficial to see how this application proceeds:

Example. Suppose that we wish to apply the rule

$$(x, (y, z)) \Longrightarrow (((x, (y)), (x, (z))))$$

to the subcircuit $g[0]$ of the above circuit C_0. Since we allow x, y and z to match any subcircuit of C_0, we can set

$$x = (g[1], g[2]), \quad y = x_3, \quad z = (x_4, x_2),$$

which transforms C_0 into C_0', namely

$$\begin{aligned} g[0] &= ((x_1, g[2]), ((x, (y, z)))) \\ &= ((x_1, g[2]), ((((x, (y)), (x, (z)))))) \cdots \text{rule } (9) \\ &= ((x_1, g[2]), ((x, (y)), (x, (z)))) \cdots\cdots \text{rule } (8) \\ &= ((x_1, g[2]), (((g[1], g[2]), (x_3)), ((g[1], g[2]), \\ &\qquad ((x_4, x_2))))). \end{aligned} \quad \square$$

The algorithm selects a transformation rule from $\Re_k$ at random and applies it to the initial circuit, and then repeats this operation:

Random Transformer RT.
Input: An initial circuit C_1
Output: A circuit C_2 that is equivalent to C_1 and is probably more complicated than C_1.

Step 1: $C \leftarrow C_1$

Step 2: Select a rule r at random from $\Re$.

Step 3: Apply r to C to get C'. If there are two or more possibilities, select one of them at random, where the probability of selecting each rule is not uniform (see Section 2.2).

Step 4: $C \leftarrow C'$ and repeat Step 2–Step 4 some specified times or until the size of C gets to some specified value.

Step 5: $C_2 \leftarrow C$

2.2 New Set of Rules

The set $\Re_k$ of transformation rules for k-NAND circuits consists of the following 12 rules, where $f \Longleftrightarrow g$ stands for $f \Longrightarrow g$ and $g \Longrightarrow f$, where S_x denotes a subcircuit as described above. It is a little surprising that $\Re_k$ is very similar to the rule set for unbounded fan-in circuits[2]; only rule (5) is different.

(1) $(\mathbf{1}) \Longleftrightarrow \mathbf{0}$	**(2)** $(\mathbf{0}) \Longleftrightarrow \mathbf{1}$
(3) $(S_x, S_x) \Longleftrightarrow (S_x)$	**(4)** $(S_x, (S_x)) \Longleftrightarrow \mathbf{1}$

(5) (a) $S_x, ((S_y, S_z)) \Longleftrightarrow ((S_x, S_y)), S_z$ if $k = 2$ or
(b) $(S_1, S_2, \cdots, S_i) \Longleftrightarrow (((S_1, S_2)), \cdots, S_i)$
for $i = 3, \cdots, k$ if k is 3 or more
(6) $S_x, S_y \Longleftrightarrow S_y, S_x$ **(7)** $(S_x, \mathbf{1}) \Longleftrightarrow (S_x)$
(8) $((S_x)) \Longleftrightarrow S_x$
(9) $(S_x, (S_y, S_z)) \Longleftrightarrow (((S_x, (S_y)), (S_x, (S_z))))$
(10) If $g[\ell] = f$ is the definition of subcircuit $g[\ell]$ then $g[\ell] \Longleftrightarrow f$.
(11) If $g[\ell]$ is neither an output of the circuit nor does not appear in the right-hand side of the definition of any k-NAND circuit, then the definition of $g[\ell]$ is removed. This rule is called a *deletion*.
(12) If the definition of label $g[\ell]$ does not exist in the right-hand side of any definition, then $g[\ell] = C$ is added, where C can be any k-NAND circuit whose length is bounded polynomially. This rule is called a *creation*.

Now we shall show that (i) k-NAND circuits are closed under any operation in $\Re_k$ (Theorem 1 below), and (ii) $\Re_k$ is complete (Theorem 2 below). Namely suppose that the algorithm RT takes a k-NAND circuit C_1. Then a circuit C_2 can be obtained by RT with positive probability if and only if C_2 is a k-NAND circuit and C_2 is equivalent to C_1.

Theorem 1. Let C_1 be any k-NAND circuit. Then the circuit C_2 obtained by applying any rule in $\Re_k$ is also a k-NAND circuit which is equivalent to C_1. (Proof is omitted.)

Theorem 2. Let C_1 and C_2 be any equivalent k-NAND circuits. Then there exists a sequence of transformation rules, each in $\Re_k$, which transforms C_1 into C_2.

Proof (Sketch). Note that rule (5-a) cannot be used directly if $k \geq 3$. However (5-a) can be simulated by using (5-b) and (6) several times (details are omitted). Also, any k-NAND circuit can be transformed to a 2-NAND circuit by applying rule (5-b) repeatedly. From these two facts, we can claim that it is enough to consider only the case for $k = 2$.

The idea of proof is the same as [2]: It is shown that any 2-NAND circuit can be transformed into a circuit of the canonical form that is unique for a particular logic function. If C_1 and C_2 are equivalent then both C_1 and C_2 can be transformed into the same canonical form, say, C, by sequences r_1 and r_2 of rules, respectively. Then, since each transformation rule is bidirectional, we can get a sequence of rules r'_2 that transforms C into C_2. Now the sequence r_1 followed by r'_2 transforms C_1 into C_2.

The canonical form can be associated with the well-known two-level NAND circuits as shown in Figure 2 (a), where each (unlimited fan-in) NAND gate corresponds to a minterm of DNF formulas. Then each NAND gate is expanded to a circuit of 2-NAND gates, as shown in Figure 2 (b) in the usual way. When a 2-NAND subcircuit takes the form of right-hand side of Figure 2 (b), we call it a *UNIT*. Now one can see that the canonical form is a tree such that its root part must be a *UNIT* and its leaf parts must be also *UNIT*s.

In Figure 2 (b), There is a 2-NAND gate whose output is connected to both inputs of another 2-NAND gate directly. Such a sequence of two 2-NAND gates can be considered as a single gate and will be called a *2-AND gate* below. Furthermore, if we say that a gate x is a 2-NAND gate, then we assume two inputs of x are connected to different two gates, i.e., it is not a 2-AND gate.

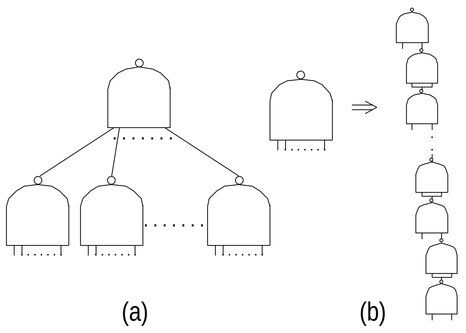

Figure 2: Canonical form of a circuit

Lemma 1. Suppose that at least one of subcircuits S_x and S_y has a 2-NAND gate as its top (output) gate. Then there exist a sequence of transformation rules, that realize the following transformations for some subcircuits S_a and S_b.

$$(S_x, S_y) \Longrightarrow ((S_a, S_b))$$
$$((S_x, S_y)) \Longrightarrow (S_a, S_b)$$

Proof. Let the top of S_y be a 2-NAND gate,

$$\begin{array}{lll} (S_x, S_y) & = & (S_x, (S_1, S_2)) \\ & = & (((S_x, (S_1)), (S_x, (S_2)))) \cdots \text{rule (9)} \\ & = & ((S_a, S_b)) \\ & & \text{(i.e., } S_a = (S_x, (S_1)) \text{ and } S_b = (S_x, (S_2))) \\ ((S_x, S_y)) & = & ((S_x, (S_1, S_2))) \\ & = & ((((S_x, (S_1)), (S_x, (S_2))))) \cdots \text{rule (9)} \\ & = & ((S_x, (S_1)), (S_x, (S_2))) \cdots \text{rule (8)} \\ & = & (S_a, S_b) \end{array}$$

Note that if the top of S_y is a 2-AND gate, then we cannot apply rule (9). □

Lemma 2. Suppose that at least one of S_x and S_y includes at least one 2-NAND gate (i.e., does not consist of only 2-AND gates). Then there exist a sequence of transformation rules, that realize the following transformations for some subcircuits S_a and S_b.

$$(S_x, S_y) \Longrightarrow ((S_a, S_b))$$
$$((S_x, S_y)) \Longrightarrow (S_a, S_b)$$

Proof. Find the 2-NAND gate $g[0]$ which is located at the most upper level. If it is at the top level, then we can use Lemma 1 directly. Otherwise, apply the transformation $((S_x, S_y)) \Longrightarrow (S_a, S_b)$ of Lemma 1 to the 2-AND gate to which the output of $g[0]$ is connected. Then we can move the 2-NAND gate "one level up". Repeat this procedure. □

Lemma 3. Suppose that S_x and S_y include no 2-NAND gate. Then the subcircuit (S_x, S_y) can be transformed into the form of a *UNIT*.

Proof. Apply rule (5-a) (from left to right) repeatedly. □

It is convenient to use a graph representation of circuits as illustrated in Figure 3, where a vertex denoted by • represents a 2-NAND gate and ∘ represents 2-AND gates. Note that Lemma 1 says that we can change • into ∘ if at least one of •'s two input vertices is •.

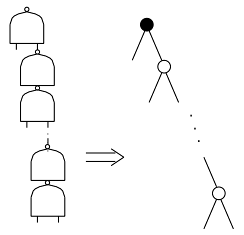

Figure 3: Graph representation of circuit

Now one can see that the canonical form is a tree such that its root must be • and there is exactly one other •, called an *intermediate • vertex*, in each path from the root to a leaf. Suppose that the tree for the circuit C_1 has a ∘ vertex as its root. Then what we have to do first is (i) to change this ∘ root to • root. Then we go down to level 2, level 3, and so on, where (ii) if we encounter a • vertex then we try to change it into a ∘ vertex. To do (i) above, i.e., to change ∘ into •, we can use Lemma 2. To do (ii), i.e., to change • into ∘, it turns out that we need another • vertex somewhere below the current • vertex. If there is no such • then the current • becomes the intermediate • vertex. After that we can transform each subcircuit into the *UNIT* form by Lemma 3. We also need several procedures such as changing the order of literals, adding missing literals and so on. For the complete proof, see [6]. □

It should be noted that Theorem 2 guarantees that there exists a path from a given circuit to any equivalent circuit, including simpler or even optimal ones. However, the theorem says nothing about how to get such an optimizing sequence or how long it is. Therefore, it is not realistic to try to use $\Re_k$ for the optimization purpose. On the other hand, it is quite likely that if we apply the rules at random, the circuit becomes more complicated. This is the basic idea of the algorithm RT.

In Step 2, we can set some biased probability for selecting each rule (for example, the one like that $x, ((y, z)) \Longrightarrow ((x, y)), z$ should be applied 10 times as often as the others). This probability setting is very important to determine the feature of the finally generated circuit. In other words, it is expected that we can control several features of the circuit by elaborating this probability and/or by introducing conditional probabilities (i.e., application of some rule being followed by some specific rule much often). For example, it would be desirable if we could generate those circuits which have relatively large depth or long paths by this improvement.

3 Random Generation of Initial Circuits

3.1 Basic Approaches

Recall that the generator RT can produce many (theoretically all) different circuits that realize the same logic function defined by the initial circuit. Therefore the role of the initial circuit can be considered as determining the logic function of generated circuits. (We will sometimes call initial circuits *initial functions.*) In this section we describe how to generate initial functions also at random. In doing so, we have to recall that random generation, in general, tends to lack the ability of controlling features and attributes of instances, which often results in too artificial or practically meaningless instances.

Then what kind of attributes should be considered for logic functions? Besides some trivial ones like the number, n, of logic variables, the following two attributes are obviously important:

(i) The size of the on-set, i.e., the number of truth assignments (out of 2^n ones) which make the function's value 1 (true).

(ii) The complexity of the function which can be measured by the size and the depth of the optimal circuit realizing the function. (Actually there is a trade-off between the size and the depth.)

An elemental way of specifying a logic function is by a truth table. A random truth table can easily be created by placing 0 or 1 at random in each of 2^n entries. Obviously it is easy to control the size of on-sets. However, it appears to be hard to control the above second attribute. As is well known [9], statically almost all logic functions need exponential circuit-size. Therefore, with a high probability, the size of circuits that realize such a random truth-table will be very large even if they are optimal. This is exactly a "too artificial nature" and must be a significant drawback. As mentioned in the first section, to draw a circuit as a random graph is not so good, either.

Our algorithm is based on random generation of a DNF formula like

$$x_1\overline{x_2}x_3 + x_2x_4\overline{x_7}x_8 + x_1x_5 + \cdots.$$

This method is clearly better than the previous one in the controllability of the function's complexity by the following reason: If we generate more (random) terms and if we assume that most of those terms are prime implicants, then it turns out that the complexity of the function tends to increase. (Clearly we need more gates if we try to realize the function with depth-two circuits. In the case of multi-level circuits we are now dealing with, there are some exceptions; e.g., the parity function needs great many (2^{n-1}) prime implicants but it has a simple log-depth circuit of linear size. However, this kind of exceptions do not seem serious.) Thus we can get more complicated functions by generating more terms in principle. Then how can we control the size of on-sets? We have to be a bit more careful about this problem.

3.2 Counting On-set

Several formal definitions are needed: a *literal* is a variable, say, x, or its negation $\overline{x}$. A term is a product of literals like $x_1\overline{x_2}x_3$. A (*DNF*) *formula* is a sum of terms. A (particular) truth assignment, say, $x_1=x_3=x_4=1$, $x_2=0$ in the case of four variables, is called a *cell*. A cell is said to be *covered* by a term C if the truth assignment denoted by the cell makes that term 1 (true). For example, term $x_1\overline{x_2}$ covers the cell of the above example. A term that covers many cells is said to be *large*. Obviously a large term consists of a few literals. A term including k literals is called a *k-term*.

Our goal is to generate terms $C_1, C_2, \cdots$ in this order and finally to get the following formula of n variables:

$$f = C_1 + C_2 + \cdots + C_m + C_{m+1} + \cdots + C_{m+t}.$$

Each C_i must cover at least one "new" cell, namely, the cell not covered by any of C_1 through C_{i-1}. C_1 through C_m are k_1-terms, namely, all those m terms are of the same size. Those terms are called *primary terms*. The size of the remaining t terms, C_{m+1} through C_{m+t}, called *secondary terms*, is determined at random between k_2 and k_3. Parameters k_1, k_2, k_3 and t are given as the input. Another parameter X determines the size of on-sets. X takes an integer between 1 and 99. The formula f must satisfy the condition that its subformula $f_{m-1} = C_1 + C_2 + \cdots + C_{m-1}$ covers less than $X\%$ of the whole (2^n) cells and subformula $f_m = C_1 + \cdots + C_m$ covers at least $X\%$ of the whole cells. Namely we stop the generation of k_1-terms when the size of the on-set becomes at least $X\%$ for the first time. Hence it does not make sense to give, for example, X=5 and k_1=3, for the first 3-term solely covers 12.5% of the whole cells.

We add the secondary terms C_{m+1} through C_{m+t}, to increase the complexity of the function. Not to change the size of the on-set too much, those terms should be small, namely k_2 and k_3 should be large. Suppose that we wish to generate a formula of 20 variables and X=50. Then suggested values for those parameters are 3 for k_1, 15 and 20 for k_2 and k_3, respectively. Since k_2 and k_3 are large (each term C_{m+i} is negligibly small), the value of t can be set without concern about the increasing size of the on-set. The standard value for t, however, is around 10. Now here is the algorithm:

Generator of Random Initial Circuit RICGEN

Input: n = the number of variables, k_1, k_2, k_3, t and X as described above.

Output: *DNF* formula $C_1 + \cdots + C_{m+t}$ as described above.

```
Step 1: $P \leftarrow 0$, $i \leftarrow 1$ and $j \leftarrow 1$.
Step 2: Generate a random $k_1$-term $C$ and compute the number $h$ of new cells covered by $C$,
    i.e., covered by $C$ but not by any of $C_1$ through $C_{i-1}$.
Step 3: If $h > 0$ then $C_i \leftarrow C$, $P \leftarrow P + h$ and $i \leftarrow i + 1$.
    Else (i.e., if $C$ covers no new cells), go back to Step 2.
Step 4: If $P < (X/100) \cdot 2^n$ then go to Step 2.
Step 5: Choose $k$ such that $k_2 \leq k \leq k_3$ at random.
    Generate a random $k$-term $C$.
Step 6: If $C$ covers at least one new cell then $C_i \leftarrow C$ and $i \leftarrow i + 1$.
    Else go back to Step 5.
Step 7: if $j = t$ then stop. Else $j \leftarrow j + 1$ and go back to Step 5.
```

Step 2 (Computing the number h) is harder than it looks (same for Step 6). For example, we can do so by memorizing all the cells covered so far. However, this method needs a lot of memory space to hold up to 2^n cells and also computation speed is slow. Our method is a combination of the inclusion-exclusion principle used in [4] and a backtrack tree search developed in [5]. Details are omitted.

4 Experiments

4.1 4-NAND Benchmarks

All experiments were conducted in SUN SPARCstation 10. For experiments on 4-NAND circuits, we selected an initial circuit that is similar to "con1" in MCNC benchmark set. Then it was transformed 30 (equivalent but) different circuits which are shown as "c-out01" through "c-out30" in Table 1. "con-in" is the initial circuit. The data of these circuits including "con-in", denoted by the number of gates/connections/depth, are shown in the second column. Those test circuits are given to logic optimizers, SIS_a, SIS_b and Trans. Here:

SIS_a: SIS with the algebraic script, 4-input NAND and NOR gates.

SIS_b: SIS with the boolean script, 4-input NAND and NOR gates.

Trans.: Optimization based on Transduction Method, developed in Kyoto Univ. 4-input NAND gates.

Table 1 shows, for example, SIS_a simplified input circuit c-out01 from 411 gates / 680 connections / 19 levels into 39 gates / 75 connections / 12 levels. The data is interesting: Generally speaking, Trans exhibits stable performance for each test circuit. SIS_b is better than Trans in many examples but sometimes very bad.

4.2 Unlimited Fan-in Benchmarks

About 200 R-RT (random initial circuit + random transformation) instances were generated, which are categorized by the following parameters:

(1) No. of variables: 10, 20, 30.

(2) Size of on-sets: 1, 5, 50, 95, 99(%). Also, we generated initial functions that consist of only one min-term, i.e., they become 1 for only one cell out of 2^n ones.

(3) No. of secondary terms: 0, 10.

For example, 10-.50-0 denotes the initial function of 10 variables, 50% on-set and no secondary terms. 10-.50-0(1) and 10-50-0(2) are different initial function with the same parameters. In the case of 50% on-set, we set k_1=4, namely, each primary term includes 4 literals, to make the number of primary terms reasonable.

name	init.	SIS_a	SIS_b	Trans.
con-in	16/27/5	12/21/5	10/19/6	13/24/5
c-out01	411/680/19	39/75/12	37/70/17	15/31/7
c-out02	645/1165/33	85/170/25	46/93/17	16/32/7
c-out03	632/1150/33	85/170/25	46/93/17	15/30/9
c-out04	258/449/30	27/59/9	11/21/6	13/26/7
c-out05	641/1098/28	41/81/13	11/21/6	15/28/9
c-out06	545/966/21	55/113/17	10/19/6	14/26/9
c-out07	621/1140/23	62/122/17	10/19/6	11/23/4
c-out08	416/727/17	11/20/5	10/20/5	13/25/8
c-out09	354/644/19	11/22/6	10/19/4	14/29/6
c-out10	295/514/19	47/89/17	11/20/5	18/30/9
c-out11	469/814/19	37/73/11	10/19/6	14/26/7
c-out12	1017/1720/23	89/182/17	10/19/6	18/32/9
c-out13	195/351/24	40/76/14	12/21/5	11/22/5
c-out14	384/692/21	46/92/19	25/45/13	13/25/8
c-out15	498/873/23	39/74/19	13/24/6	16/29/7
c-out16	301/528/26	16/31/8	10/19/6	13/24/5
c-out17	1516/2767/25	53/112/13	40/79/12	17/33/7
c-out18	437/774/25	56/112/19	10/19/6	12/24/5
c-out19	396/688/23	28/55/12	15/25/9	15/27/7
c-out20	209/366/16	14/26/6	10/19/6	13/25/6
c-out21	219/383/18	14/26/6	10/19/6	10/21/4
c-out22	709/1361/20	51/99/15	15/28/6	12/26/7
c-out23	303/515/25	12/21/5	10/19/6	14/27/5
c-out24	314/548/24	21/39/12	17/31/8	13/24/5
c-out25	882/1539/25	57/114/15	11/21/5	13/29/7
c-out26	753/1327/25	32/65/10	10/19/6	11/24/6
c-out27	261/484/25	66/131/15	11/20/6	14/28/7
c-out28	388/685/19	86/175/15	22/42/10	10/18/5
c-out29	355/671/17	60/117/21	18/34/8	14/28/7
c-out30	294/543/19	35/66/13	10/19/6	13/24/7

Table 1: Experimental result of 4-NAND circuits

(It was 10 or 11 in this setting.) For other on-set ratios, we set k_1=3, 3, 9 and 10 for 99%, 95%, 5% and 1%, respectively. The number of primary terms is approximately between 20 and 30. Secondary terms are added only to 50% on-set instances, which makes the number of the whole terms around 20. Computation time mostly depends on the number of terms generated, i.e., if the size of on-sets is large then it is slow. In the case of 99% on-set, it took about 10 sec (CPU time), 10^2 sec and 10^3 sec for functions of 10, 20 and 30 variables, respectively.

Those initial functions (circuits) are then fed to the random transformer. Again for each initial circuit, we generated three (different) circuits. They are denoted, for example, by 10-.50-0(1,1), 10-.50-0(1,2) and 10-.50-0(1,3), which are final test-circuits and were made sure (just in case of program errors) to be equivalent to the initial circuit (i.e. 10-.50-0(1)) using a program developed by the third party. We then ran the random transformer RT until the number of gates reached around 1000, for which it took about 10^3 sec.

We also generated MCNC-RT instances. As initial circuits, we picked 9symml, cm82a, z4ml, frg1 and cordic. From each initial circuit, about 10 final circuits were generated by the random transformer. In this case we executed 2000 times of rule applications.

4.3 Responses of SIS and Transduction Method

Table 2 to table 5 are part of experimental results. See Table 2, which is the responses of SIS_a and Trans. for 10-.50-0(2,1), 10-.50-0(2,2) and 10-.50-0(2,3). Note that SIS_a and Transduction Method take unlimited fan-in circuits and they output simplified circuits with fan-in up to 4. The table consists of four major rows: The first row shows the optimization results for the initial circuit 10-.50-0(2). The following three major rows are for the above three test circuits. For example, circuit 10-.50-0(2,1) is composed of 1000 gates, 2161 connections and 23 levels, which are reduced into 45, 91 and 11, respectively, by SIS_a and into 38, 87 and 9, respectively, by Trans. We assume that the data for the optimal circuit is close to 26 gates, 61 connections and 5 levels, namely, that is the minimization data for the initial circuits (we take the better one in the number of gates). Under this assumption, we calculate the degree of gate–optimization for 10-.50-0(2,1) as 45/26=1.73 for SIS_a and 38/26=1.46 for Trans. Similarly for connections and levels. The forth column shows CPU time in second.

Tables 3 to 6 are similar. The degree of optimization is sometimes large, for example, as large as 4.5 in Table 4. Note that in Table 3, Trans. is quite worse than SIS_a for the initial circuit. (Hence SIS_a data is taken as the assumed optimal.) It should be noted that we only used the algebraic script of area-priority for SIS. According to the previous Table 1, SIS with the boolean script would be better in terms of simplification. Table 5 is for MCNC-RT instances.

Table 6 is a grand summary of the experiments. Each row shows average values over three (sometimes two when SIS_a or Trans. failed) circuits for each initial function. The column denoted by "deg" is for the degree of optimization. The t/s column shows (deg for Trans. ÷ deg for SIS_a). The "diver" column shows the divergence, i.e., (max deg among three ÷ min deg among three).

1. The deg column for MCNC-RT holds mostly less than 2 but the value for R-RT is mainly between 2 and 4. Thus R-RT seems to be more difficult to be optimized than MCNC-RT.

2. See the t/s column. Many values are less than 1, namely, Trans. overperformed SIS_a. However, SIS_a is generally faster than Trans.

3. Degree of optimization does not differ so much due to the difference of the initial function. For example, 20-.50-10 should have larger complexity than 20-.50-0, but the deg values are similar. However, as for computation time, the former is obviously slower than the latter.

4. Finally, it should be remarked that there is a wide discrepancy in the computation time. Sometimes the divergence reaches almost 100 times even among the circuits from the same initial function.

Circuit		gate	conn.	level	time
Initial	input	20	59	3	–
	SIS_a	28	60	7	2.1
	Trans	26	61	5	1.5
No.1	input	1000	2161	23	–
	SIS_a	45	91	11	16.7
	Trans	38	87	9	134.4
No.2	input	1188	2236	22	–
	SIS_a	124	258	19	28.1
	Trans	42	98	11	139.9
No.3	input	1114	2405	15	–
	SIS_a	71	144	21	24.7
	Trans	44	106	11	100.1

Table 2: 10-.50-0(2)

Circuit		gate	conn.	level	time
Initial	input	38	271	4	–
	SIS_a	83	189	12	14.4
	Trans	148	320	10	5.1
No.1	input	1154	3235	19	–
	SIS_a	309	691	22	116.3
	Trans	202	451	22	169.4
No.2	input	1041	2883	16	–
	SIS_a	188	434	18	50.0
	Trans	179	421	14	133.2
No.3	input	1053	3586	14	–
	SIS_a	192	443	16	86.9
	Trans	207	475	16	190.7

Table 3: 10-.05-0(2)

Circuit		gate	conn.	level	time
Initial	input	27	70	3	–
	SIS_a	27	61	6	2.0
	Trans	32	69	5	2.5
No.1	input	1007	2378	24	–
	SIS_a	79	162	25	17.8
	Trans	55	127	13	169.1
No.2	input	1278	2439	21	–
	SIS_a	119	236	17	26.1
	Trans	56	124	11	106.6
No.3	input	1058	2181	19	–
	SIS_a	100	207	19	19.9
	Trans	40	90	9	170.7

Table 4: 20-.50-0(1)

Circuit		gate	conn.	level	time
Initial	input	167	401	13	–
	sis	177	385	13	13.0
	trans	153	384	13	8.2
No.1	input	1190	2818	19	–
	sis	219	479	19	39.6
	trans	207	509	15	92.9
No.2	input	1365	2950	26	–
	sis	404	845	25	90.7
	trans	199	499	19	103.8
No.3	input	1344	2682	22	–
	sis	305	642	25	55.7
	trans	189	462	21	98.3

Table 5: 9symml

5 Conclusion

The discussion using fixed benchmarks usually goes like "one system is better than theirs by 8.3% on average." The difference is this small. Our experiments show that if we use random benchmarks then this difference can get as large as 400%. Furthermore, the difference is significantly larger for R-RT instances than for MCNC-RT instances. Note that R-RT instances are "more random" than MCNC-RT instances. Thus it is quite reasonable to claim that random benchmarks are harder or more intractable than fixed benchmarks for logic optimizers.

References

[1] R.K.Brayton, R.Rudell, A.L.Sangiovanni-Vincentelli, and A.R.Wang, "Mis: A Multiple-level Logic Optimization System," *IEEE Trans.* CAD, 6, pp. 1062-1081, 1987.

[2] K. Iwama and K. Hino, "Random Generation of Test Instances for Logic Optimizers," 31st *ACM/IEEE* Design Automation Conference, pp. 430-434, 1994.

[3] B. Cha and K. Iwama, "Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas", *in* Proc. International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Aug 1995.

[4] K. Iwama, "CNF Satisfiability Test by Counting and Polynomial Average Time," SIAM J. Computing, 18,2, 385-391, 1989.

[5] O. Kanmoto and K. Iwama, "On Improvement of the Satisfiability Test by Counting using Backtracking," Record of Joint Conference of Electrical and Electronics Engineers in Kyushu, 1505, Kumamoto, 1994.

[6] H. Kurokawa, S. Sawada and K. Iwama, "A Complete Set of Transformation Rules for Fan-in Restricted NAND Circuits," Technical Report, 96C-06, CSCE Dept., Kyushu Univ., 1996.

[7] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method – Design of Logic Networks Based on Permissible Functions," *IEEE Trans. Comput.* 38, 10, 1989.

[8] E. M. Sentovich, K. J. Singh, *et al.*, "SIS: A System for Sequential Circuit Synthesis," Memorandum No. UCB/ERL M92/41, 1992.

[9] J. Savage, *The Complexity of Computing*, Wiley, New York, 1976.

[10] G. Tinhofer, "Generating Graphs Uniformly at Random," *in* Computational graph theory, pp. 235-255, Springer, 1990.

[11] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," *in* 1991 MCNC International Workshop on Logic Synthesis.

Initial Circuit		gate			conn.			level			time		
		deg.	t/s	diver.	deg.	t/s	diver.	deg.	t/s	diver.	av.	t/s	diver.
9symml	SIS_a	2.03		1.84	1.73		1.76	1.71		1.32	63.9		2.57
	Trans	1.26	0.62	1.11	1.26	0.73	1.10	1.35	0.79	1.53	89.9	1.41	2.92
cm82a	SIS_a	1.08		1.31	1.03		1.35	1.73		1.67	10.0		2.46
	Trans	0.95	0.88	1.33	0.98	0.95	1.23	1.27	0.73	1.67	71.4	7.13	23.92
z4ml	SIS_a	2.90		2.31	2.86		3.04	2.30		2.18	23.5		2.65
	Trans	1.02	0.35	1.32	1.02	0.36	1.21	1.18	0.51	1.25	96.5	4.10	18.01
frg1	SIS_a	1.71		1.70	1.72		1.65	1.24		1.35	37.0		2.88
	Trans	1.23	0.72	1.27	1.41	0.82	1.35	1.14	0.92	1.24	3201.0	86.63	2.55
cordic	SIS_a	1.88		1.47	1.88		1.47	1.72		1.93	26.1		2.51
	Trans	1.29	0.69	1.65	1.33	0.71	1.78	1.35	0.78	1.50	139.2	5.32	27.29
10-.01-0(1)	SIS_a	2.87		2.86	2.76		2.73	1.76		1.71	37.1		1.44
	Trans	2.10	0.73	1.31	1.94	0.70	1.26	1.33	0.76	1.50	360.5	9.71	4.13
10-.05-0(1)	SIS_a	3.83		2.84	3.64		2.70	1.97		1.58	660.3		26.96
	Trans	2.28	0.60	1.06	2.29	0.63	1.04	1.50	0.76	1.38	145.6	0.22	1.13
10-.05-0(2)	SIS_a	2.77		1.64	2.76		1.59	1.56		1.38	84.4		2.33
	Trans	2.36	0.85	1.13	2.38	0.86	1.13	1.44	0.92	1.57	164.4	1.95	1.43
10-.50-0(1)	SIS_a	2.26		2.05	2.01		2.11	3.00		2.38	17.3		1.61
	Trans	1.41	0.62	1.03	1.42	0.76	1.04	2.07	0.69	1.22	144.4	8.33	2.99
10-.50-0(2)	SIS_a	3.08		2.76	2.69		2.84	3.40		1.91	23.2		1.68
	Trans	1.59	0.52	1.16	1.59	0.59	1.22	2.07	0.61	1.22	124.8	5.39	1.40
10-.95-0(1)	SIS_a	3.25		2.47	3.40		2.40	2.56		1.90	20.9		2.32
	Trans	1.93	0.59	1.22	2.13	0.63	1.18	1.89	0.74	1.30	120.9	5.78	4.93
10-.95-0(2)	SIS_a	3.47		3.37	3.11		3.41	3.20		1.91	26.4		1.70
	Trans	1.74	0.50	1.15	1.67	0.54	1.14	2.80	0.88	1.15	102.0	3.87	1.31
10-.99-0(1)	SIS_a	7.33		1.89	8.00		1.92	2.60		1.89	20.7		1.39
	Trans	1.71	0.23	1.05	1.67	0.21	1.03	1.70	0.65	1.43	106.75	5.16	1.84
10-.99-0(2)	SIS_a	4.80		1.61	5.59		1.57	3.50		1.47	44.2		2.39
	Trans	1.88	0.39	1.14	2.10	0.38	1.12	1.58	0.45	1.11	301.1	6.81	6.72
20-.01-0(1)	SIS_a	1.09		1.09	1.13		1.07	1.80		1.25	74.8		1.71
	Trans	1.47	1.35	1.04	1.40	1.24	1.06	2.00	1.11	1.00	331.7	4.43	1.11
20-.50-0(1)	SIS_a	3.68		1.51	3.31		1.46	3.39		1.47	21.3		1.47
	Trans	1.86	0.51	1.40	1.86	0.56	1.41	1.83	0.54	1.44	148.8	7.07	1.60
20-.50-0(2)	SIS_a	2.58		1.44	2.34		1.44	3.53		1.77	23.5		2.33
	Trans	1.48	0.57	1.25	1.58	0.68	1.24	2.20	0.62	1.44	503.1	21.41	10.43
20-.95-0(1)	SIS_a	1.73		1.25	1.61		1.25	2.24		1.27	100.2		5.12
	Trans	1.59	0.92	1.11	1.58	0.98	1.16	2.14	0.96	1.00	397.8	3.97	1.59
20-.95-0(2)	SIS_a	2.04		1.72	1.83		1.75	2.33		1.46	31.4		1.53
	Trans	1.55	0.76	1.16	1.55	0.85	1.20	2.33	1.00	1.13	328.1	10.46	2.84
20-.99-0(1)	SIS_a	1.51		1.55	1.51		1.52	1.71		1.56	20.2		1.27
	Trans	1.68	1.12	1.11	1.89	1.25	1.16	1.95	1.14	1.15	264.7	13.1	1.31
20-.99-0(2)	SIS_a	1.36		1.66	1.39		1.63	1.86		2.33	18.0		1.52
	Trans	1.49	1.10	1.14	1.66	1.20	1.19	1.67	0.90	1.18	191.2	10.65	1.76
10-.50-10(1)	SIS_a	3.04		3.08	3.00		2.84	2.10		2.82	91.0		7.34
	Trans	2.14	0.71	1.04	2.29	0.76	1.04	1.80	0.86	1.40	5191.9	57.09	93.06
20-.50-10(1)	SIS_a	1.67		1.09	1.62		1.17	1.07		1.13	103.7		3.93
	Trans	2.08	1.25	1.14	1.98	1.22	1.16	1.07	1.00	1.20	614.0	5.92	2.57
10-s-0(1)	SIS_a	1.00		1.00	1.00		1.00	1.00		1.00	14.3		1.62
	Trans	2.56	2.56	1.14	1.69	1.69	1.22	2.44	2.44	1.33	676.1	47.28	3.78
SIS_a 20-s-0(1)	1.00		1.00	1.00		1.00	1.00		1.00	10.2		2.71	
	Trans	1.95	1.95	1.08	1.43	1.43	1.02	2.00	2.00	1.67	279.0	27.35	30.37

Table 6: Summary of the Experiments