# FRAIGs: A Unifying Representation for Logic Synthesis and Verification

**Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, Robert Brayton**

Department of EECS, University of California, Berkeley
{alanmi, satrajit, jiejiang, brayton}@eecs.berkeley.edu

## Abstract

*AND-INV graphs (AIGs) are Boolean networks composed of two-input AND-gates and inverters. In the known applications, such as equivalence checking and technology mapping, AIGs are used to represent and manipulate Boolean functions. AIGs powered by simulation and Boolean satisfiability lead to* functionally reduced AIGs (FRAIGs)*, which are "semi-canonical" in the sense that each FRAIG node has unique functionality among all the nodes currently present in the FRAIG. The paper shows that FRAIGs can be used to unify and enhance many phases of logic synthesis: from the representation of the original and the intermediate netlists derived by logic optimization, through technology mapping over multiple logic structures, to combinational equivalence checking. Experimental results on large public benchmarks confirm the practicality of using FRAIGs throughout the logic synthesis flow.*

## 1 Introduction

AND-INV graphs (AIGs) represent Boolean functions in combinational equivalence checking (CEC) [10][13], bounded model checking (BMC) [1] and technology mapping [12][15]. As a functional representation AIGs enjoy remarkable properties:

- AIGs are composed of two-input ANDs and inverters represented as flipped bits on the edges. This uniformity has considerable implementation advantages.
- The AIGs are a multi-level logic representation whose construction time and size are proportional to the size of the circuit. This is in contrast with two most commonly used representations: BDDs whose canonicity forces their size to be exponential for some practical circuits such as multipliers, and SOPs whose two-level form leads a non-robust manipulation of large logic nodes in SIS [21].
- AIGs enhanced with random simulation and Boolean satisfiability can efficiently solve a remarkable variety of computational problems in logic synthesis and verification. Simulation plays the role of directing synthesis and reducing the number of calls to the SAT solver.

AIGs are not canonical: a Boolean function has many AIG representations. For example, function $F = abc$ can be represented as follows: $((ab)c)$, $(a(bc))$, $((ac)(bc))$, etc. Figure 1 shows two AIGs of a four-variable function, which cannot be derived from each other by algebraic transformations. These AIGs are different Pareto points on the area/delay curve: one has fewer ANDs, while another has fewer levels of ANDs.

Since AIGs are not canonical, internal nodes of an AIG may have equivalent functionality. This increases the number of AIG nodes and makes reasoning on the graph structure inefficient and time consuming. Indeed, merging two equivalent nodes removes one variable from any underlying SAT problem and reduces the search space for feasible solutions by the factor of two.

Many applications, e.g. [17], require that the functionally equivalent nodes be detected and merged. In this paper, this process is called *functional reduction*. The known applications achieve functional reduction by applying BDD sweeping [13] or SAT sweeping [14] to the AIGs as a post-processing step.

The first contribution of the present paper is an algorithm that integrates functional reduction into the process of AIG construction (rather than a post processing step) leading to a functional representation which we call *Functionally Reduced AIGs* (FRAIGs). FRAIGs are "semi-canonical" because no two nodes in a FRAIG structure have the same function in terms of the primary inputs, but the same function may have different FRAIGs structures. The construction is efficient due to the synergy between simulation and SAT. Experimental results show that FRAIGs are much more robust than BDDs and can be constructed for a wide range of practical circuits in reasonable time.

The paper presents an implementation of a FRAIG package and discusses how functional reduction on-the-fly is similar to the efficient method [2] of building Reduced Ordered Binary Decision Diagrams (ROBDDs) in the present ROBDD packages. (Originally [5], the BDDs were constructed in a non-reduced form and the reduction process was applied as a post-processing step.)

The second contribution of the paper is in showing that, due to functional reduction, FRAIGs can efficiently accumulate *structural choices* (functionally equivalent FRAIG structures). The efficiency of FRAIG construction suggests a new synthesis methodology called *lossless logic synthesis*, which uses intermediate structures derived during logic optimization as well as the structure of the initial and the final networks. Therefore, in lossless synthesis, the role of logic optimization shifts from deriving a single final optimized network to performing guided permutations of multiple logic structures and allowing for the final implementation to be chosen among these by the technology mapping step. This idea also enables a different design style where-in a designer can optimize parts of the circuit manually. These are added as choices. If the designer's implementation is superior to the ones derived by the synthesis system, it will be automatically selected during technology mapping. This increases the applicability of synthesis tools in an industrial context.

The third contribution is in developing and experimentally evaluating a prototype of a lossless logic synthesis system, which uses FRAIGs at all stages of the flow, from the representation of the original and partially-optimized netlists, through technology mapping over multiple logic structures, to combinational equivalence checking.

The paper is organized as follows. Section 2 surveys traditional AIGs. Section 3 reviews the previous work. Section 4 outlines the new algorithm to construct FRAIGs. Section 5 discusses some implementation details. Section 6 presents lossless logic synthesis. Section 7 reports experimental results. Section 8 concludes and lists directions for future work.

## 2 Background

This paper assumes familiarity with the basics of Boolean functions, Boolean networks, and Binary Decision Diagrams [5].

### 2.1 Definitions

**Definition**. *AND-INV graph* (AIG) is a Boolean network composed of two-input AND-gates and inverters.

**Definition**. A representation of a Boolean function is *canonical* if, for any function, the representation is unique.

AIGs are not canonical, that is, the same function can be represented by two functionally equivalent AIGs with different structure. An example of such function is shown in Figure 1.

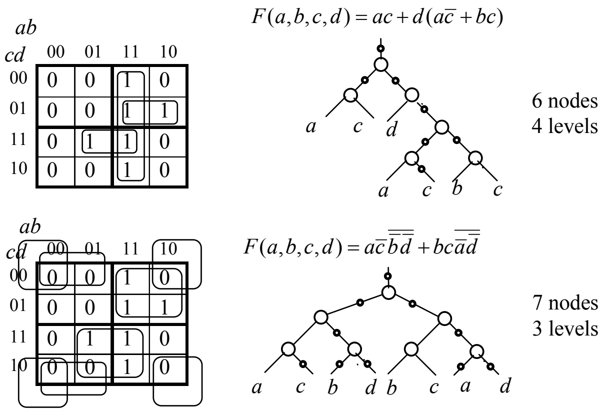
$F(a,b,c,d) = ac + d(a\bar{c} + bc)$
6 nodes
4 levels
$F(a,b,c,d) = a\bar{c}\overline{\bar{b}\bar{d}} + bc\overline{\bar{a}\bar{d}}$
7 nodes
3 levels
ab
cd
a c d a c b c
a c b d b c a d

**Figure 1**. Two different AIGs for a Boolean function.

Note that both graphs in Figure 1 are FRAIGs, since in each of them no pair of nodes represents the same function. Figure 2 shows the same function represented by a redundant graph with nodes *A* and *B* having the same function.

$$F(a,b,c,d) = \bar{d}((ab)\bar{c} + a(b\bar{c})) + d(a\bar{c} + bc)$$

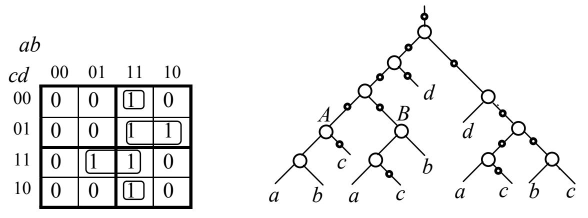

**Figure 2. An AIG with functionally redundant nodes.**

**Definition**. The *size* of an AIG is the number of AND nodes in it. The number of logic levels is the number of AND-gates on the longest path from a primary input to a primary output.

The inverters are ignored when counting nodes and logic levels. In the software implementation, inverters are represented as flipped node pointers [13]. This implementation is similar to that of BDDs with complemented edges [2].

**Definition**. The function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in node *n* and expressed in terms of the PI variables *x* assigned to the leaves of the AIG.

**Definition**. A *functionally reduced AIG* (FRAIG) is an AIG, in which, for any two $n_1$ and $n_2$, $f_{n_1}(x) \neq f_{n_2}(x)$ and $f_{n_1}(x) \neq \overline{f_{n_2}(x)}$.

### 2.2 AIG Construction

AIGs for Boolean functions can be constructed starting from different functional representation:

**SOP:** Given an SOP representation of a function, it can be factored [3] and the factored form can be converted into the AIGs. Each two-input OR-gate is converted into a two-input AND-gate using the DeMorgan rule.

**Circuit**: Given a circuit representation of a (multi-output) Boolean function, the (multi-output) AIG is constructed in a bottom-up fashion, by calling a recursive construction procedure for each PO of the circuit. When called for a PI node, the procedure returns the elementary AIG variable. Otherwise, it first calls itself for the fanins of a node and then builds the AIG for the node using the factored form of the node.

When an AIG is constructed from a circuit, the number of AIG nodes does not exceed the number of literals in the factored forms. When the AIG is constructed from a BDD, the number of AIG nodes does not exceed three times the BDD number since each MUX can be represented using three ANDs. It follows that the size of the AIG is proportional to the size of the circuit or BDD.

Quantifications performed on AIGs have an exponential complexity in the number of quantified variables because quantifying each variable is done by ORing the cofactors and can potentially duplicate the graph size. Except for quantification, Boolean computation is more robust with AIGs than with BDDs. This is because Boolean operations on AIGs lead to the resulting graphs whose size is bounded by the *sum* of the sizes of their arguments, while in the case of BDDs the worst case complexity of the result is equal to the *product* of the sizes of the arguments.

### 2.3 Structural Hashing

*Structural hashing* (*strashing*) of AIGs introduces partial canonicity into the AIG structure. When a new AND-gate is added to the graph, several logic levels of the fanin AND-gates are mapped into a canonical form. Although the resulting AIG is not canonical, it contains sub-graphs, which are canonical as long as they have less than the given number of logic levels.

**No strashing**: When an AIG is constructed without strashing, AND-gates are added one at a time without checking whether an AND-gate with the same fanins already exists in the graph.

**One-level strashing**: When a new AND-gate is added, checks is performed for a node with the same fanins (up to permutation).

**Two-level strashing**: In the pre-computation phase, all two-level AND-INV combinations are enumerated and, for each Boolean function realizable by a two-level AIG, one representation is selected as the representative one. In the AIG construction phase, when adding a new AND-gate, the canonical form of the two-level AIG rooted in this gate is constructed, which may require building new AND-gates for the fanins.

A detailed discussion of two-level structural hashing can be found in [9][13]. An efficient implementation runs in time linear in the number of AIG nodes. The resulting graphs may have 5-10% fewer nodes, compared to one-level strashing. A drawback of two-level strashing is that when multiple AIGs are constructed repeatedly, it leads to an increase in the number of unused nodes in the AIG manager, which in turn leads to the need to perform repeated garbage collection. This may slow down some AIG-based applications, such as image computation.

## 3 Previous Work

AIGs have been applied as a circuit representation in combinational equivalence checking (CEC) [13] and an object graph representation in technology mapping [15]. In both cases, AIGs are built initially using strashing, and later optionally post-processed to enforce functional reduction. If [18] AIGs are used for unbounded model checking, in which both the circuits and interpolants computed from the unsatisfiability proofs are represented by AIGs. This work recognizes the need for functional reduction ([18], Section 3.2, paragraph 1) noting that AIGs tend to have many redundancies not captured by strashing.

Two procedures have been proposed to perform functional reduction. *BDD sweeping* [13] constructs BDDs of the AIG nodes in terms of the PIs and intermediate "cut-point" variables. BDD construction is controlled by resource limits, such as a restriction on the BDD size. Any pair of AIG nodes with the same BDD is merged, and the fanout cones are rehashed. As long as all BDDs can be built within the resource limits, the result is a FRAIG. The second procedure, *SAT sweeping* [14][16], achieves the same merging and propagation by solving a sequence of incremental topologically-ordered SAT problems designed to prove or disprove the equivalence of cut-point pairs. The candidate pairs are detected using simulation. In both approaches, the initial graph is constructed in a redundant form followed by functional reduction applied as a post-processing step.

Another approach to CEC was developed using NAND graphs [7] but the authors do not discuss what methods are used to perform functional reduction or how they prove the equivalence of the output functions represented using NAND graphs.

## 4 Functional Reduction Algorithm

This section presents the main contribution of the paper, a new and efficient algorithm to build AIGs on-the-fly while ensuring that they are functionally reduced by construction.

Figure 3 shows the pseudo-code of the traditional AIG construction with one-level strashing. The first part checks trivial cases, such as when the nodes are equal up to complementation, or when one node is a constant. Next, the arguments are ordered to ensure that swapping of fanins does not create a new node. One-level strashing is performed by looking up in a hash table, which maps the pair of fanins into the AND gate with these fanins. If a node with these fanins exists, it is returned. Otherwise, a new node is created, added to the hash table, and returned.

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Nodee * n2 )
{
    Aig_Node * res, * cand, * temp;   Aig_NodeArray * class;
    /*** trivial cases ***/
    if ( n1 == n2 )                                  return n1;
    if ( n1 == NOT(n2) )                             return 0;
    if ( n1 == const )                               return 0  or  n2;
    if ( n2 == const )                               return 0  or  n1;
    if ( n1 < n2 )  { /*** swap the arguments ***/
        temp = n1; n1 = n2; n2 = temp;
    }
    /*** one level structural hashing ***/
    res = HashTableLookup( p->pTableStructure, n1, n2 );
    if ( res )                                       return res;
    res = CreateNode( p, n1, n2 );
    HashTableAdd( p->pTableStructure, res );         return res;
}
```

**Figure 3. AIGs construction with one-level strashing.**

Figure 4 contains the pseudo-code of FRAIG construction, which performs both one-level strashing followed by functional reduction. Functional equivalence test, the most time-consuming part of the algorithm, is performed by a call to the SAT solver in CheckFunctionalEquivalence(). To reduce the number of SAT solver calls, random simulation is employed. The simulation vector of each node is derived using bit-parallel simulation of the AIG starting from the PIs. In our implementation, simulation of a new node is done incrementally by the bit-wise ANDing (possibly complemented) simulation vectors of the fanins. An additional hash table is used, which maps each simulation vector into a set of AIG nodes that have this simulation vector (its *simulation class*).

```
Aig_Node * OperationAnd( Aig_Man * p, Aig_Node * n1, Aig_Nodee * n2 )
{
    Aig_Node * res, * cand, * temp;   Aig_NodeArray * class;
    /*** trivial cases ***/
    if ( n1 == n2 )                                  return n1;
    if ( n1 == NOT(n2) )                             return 0;
    if ( n1 == const )                               return 0  or  n2;
    if ( n2 == const )                               return 0  or  n1;
    if ( n1 < n2 )  { /*** swap the arguments ***/
        temp = n1; n1 = n2; n2 = temp;
    }
    /*** one level structural hashing ***/
    res = HashTableLookup( p->pTableStructure, n1, n2 );
    if ( res )                                       return res;
    res = CreateNode( p, n1, n2 );
    HashTableAdd( p->pTableStructure, res );
    if ( p->FlagUseOneLevelHashing )                 return res;
   /*** functional reduction ***/
    class = HashTableLookup( p->pTableSimulation, n1, n2 );
    if ( class == NULL ) {
        class = CreateNewSimulationClass( res );
        HashTableAdd( p-> pTableSimulation, class );   return res;
    }
    for each node cand in class
        if ( CheckFunctionalEquivalence( cand, res ) ) {
            AddNodeToEquivalenceClass( class, res );   return cand;
        }
    AddNodeToSimulationClass( class, res );          return res;
}
```

**Figure 4. FRAIG construction.**

In Figure 4, the simulation table lookup results in the simulation class of the new node. If the class is empty, a new class is created and initialized with the given node. In this case, there is no need for the equivalence check because the new node is proved to be functionally unique by simulation.

If the simulation class is not empty, then for each member *cand* of this class a SAT-based functional equivalence test is performed. If the equivalence checking procedure returns TRUE, that is, the new node *res* is equivalent to the old node *cand*, the old node is returned, which ensures functional reduction. Finally, if the new node is not equivalent to any node in its simulation class, it is added to the simulation class and returned.

## 5 Implementation Details

This section discusses the details of the FRAIG implementation.

### 5.1 Simulation

The performance of the proposed algorithm critically depends on the efficiency of simulation. The larger are simulation vectors, the better is their distinguishing power and the fewer SAT-based equivalence tests are needed. In the current implementation, approximately 4000 random bit-patterns are used for random simulation. The simulation runtime is typically about 10% of the SAT solver runtime. The memory overhead for storing simulation information is about 0.5K per node. This memory is allocated independently from the memory used for the AIG nodes. When the FRAIG is constructed, the simulation memory can be de-allocated and re-used by the application.

Another way of increasing the efficiency of simulation is using the counter-examples returned by the SAT solver during unsuccessful equivalence tests. As pointed out in [13], these counter-examples distinguish functions, which are hard to distinguish by random simulation. In the current implementation, random simulation is performed when a node is first constructed. To use the SAT solver feedback, we re-simulate the AIG periodically, each time 32 new counter-examples are accumulated.

### 5.2 SAT Solving

For efficiency, the algorithm requires tight integration of the circuit-based AIG data structure and a SAT solver. The solver used in the implementation is MiniSat [8], with modifications to restrict incremental SAT solving to a subset of variables and clauses.

The CNF for the AIG is loaded in the SAT solver incrementally, by adding three CNF clauses each time a new AIG node is created. Checking functional equivalence for AIG nodes $n_1$ and $n_2$ is performed as follows: (1) collect the AIG nodes in the union of the transitive fanin cones of $n_1$ and $n_2$; (2) set the "branchable" variables to be those corresponding to the above AIG nodes; (3) run the solver to prove or disprove equivalence.

Incremental runs of the SAT solver create learned clauses, which are stored in the global clause database. Because the logic cones of different equivalence checking problems often overlap, the learned clauses are shared and reused, which improves the performance of the SAT solver.

### 5.3 Handing Functionally Equivalent Nodes

In Figure 4, when a new node is found to be functionally equivalent to the old node, the new node can be garbage collected. However, in the current implementation, the new node is left in the graph as a node without fanouts. The node is stored in the list of equivalent nodes, and from the node, we have the pointer to the representative of the equivalence class. Keeping the equivalent nodes around works as a "structural record" of equivalences proved, similar to the computed table in the BDD package. If we hit the same structure again, we look at the pointer to the representative node, and return this representative immediately, without going through the potentially expensive equivalence test.

Saving the functionally equivalent nodes is also beneficial for some applications discussed in the following section.

## 6 Applications of FRAIGs

### 6.1 Traditional Logic Synthesis

A straight-forward use of FRAIGs in logic synthesis is to compact circuits by detecting and merging functionally equivalent nodes. To this end, the FRAIGs for all the network nodes are constructed in terms of the PI variables. Next, the network nodes are grouped into classes of equivalent functionality if they are represented by the same FRAIG node. One representative of each class is selected and substituted for other nodes of the same class.

Other potential applications of FRAIGs in synthesis include: (a) a uniform representation of algebraic factored forms and DAGs resulting from Boolean decomposition, (b) a robust representation of node functions, manipulated by a logic synthesis system when it performs operations, such as elimination, collapsing, and node immunization, (c) an alternative computation engine to solve Boolean problems, such as don't-care computation.

### 6.2 "Lossless" Logic Synthesis

In the classical synthesis, the network is subjected to a sequence of optimization steps. Each step leads to a new network while the previous networks are no longer considered; in other words, they are lost for optimization. Meanwhile, it is possible that an intermediate network was better, in whole or in part, compared to the final one. The idea of lossless synthesis is to accumulate all the intermediate logic representations and postpone the final selection until later in the design process.

FRAIG construction can be seen as an efficient way of detecting and accumulating alternative structural implementations of Boolean functions. In this scenario, several versions of the network derived by applying a sequence of optimization commands are FRAIGed into one AIG, which internally records the structural alternatives using classes of functionally equivalent AIG nodes. No matter where the structural differences occur in the networks, on the PI side or on the PO side, FRAIGs take care of identifying and storing these differences in terms of intermediate "cut-points". Technology mapping applied to this cumulative graph selects the best mapping over all available choices, which originate from different versions of the same network.

### 6.3 Technology Mapping

The traditional technology mapping [12] takes as input an object graph represented by a two-input gate network and a set of pattern graphs corresponding to the gates from a standard cell library. The goal of mapping is to find a covering of the object graph using the pattern graphs, which is optimal with respect to a cost function.

More recent work [15] represents the object graph using AIGs with choice nodes, which compactly encode multiple logic structures. In [15] the structures are derived by enumerating algebraic decompositions of the logic nodes. The more choices are present in the object graph, the better is the mapping quality.

In the present work, we developed a technology mapper that is similar to [15] in its capacity to handle choices and map over a number of encoded logic structures. A detailed discussion of our technology mapper is beyond the scope of this paper. Here we only make several remarks about the use of choices generated and used in the new technology mapper.

The choices used in our technology mapper include different logic structures derived by FRAIGing in the course of netlist optimization as well as different algebraic decompositions. The former are "sparse" and "deep" because there are relatively few of them but they penetrate across multiple logic levels. The latter are "dense" and "shallow" because there are many of them but they cover only a few logic levels. The combination of the two types of structural flexibility helps overcome structural bias, which favorably reflects on the quality of technology mapping.

## 6.4 Formal Verification

The traditional AIGs post-processed by BDDs sweeping or SAT sweeping to ensure functional reduction are widely used in CEC [13][10] and BMC [1][14]. In this paper, we use a simplified equivalence checker developed by performing functional reduction on the fly. Using FRAIGs for CEC in this case is similar to using BDDs. Once FRAIGs are constructed for the circuit outputs, the circuits are equivalent if and only if the corresponding pairs of outputs are represented by the same FRAIG nodes.

It should be noted that interleaving functional reduction on the PI side with SAT-based search for counter-examples on the PO side, as proposed in [13], leads to a more robust CEC for deeper circuits, compared to forcing functional reduction on the fly, as done in the current implementation. The non-robustness of FRAIGs shows in processing very deep circuits derived by unrolling sequential circuits in BMC, while relatively shallow circuits optimized in a delay-driven synthesis flow allow for fast construction of FRAIGs and for an efficient accumulation of structural choices. This observation is backed by our experiments.

## 7 Experimental Results

The proposed algorithm for constructing FRAIGs is implemented in C as a stand-alone AIG package "FRAIG" [19]. The package was tested in the MVSIS environment [20] and used in a number applications ranging from compression of logic functions (Experiment 1) to technology mapping (Experiment 3) and equivalence checking (Experiment 4).

## Benchmarks

The test cases are taken from the following sources:

- MCNC benchmarks [23] (the first four circuits in Table 1)
- ISCAS benchmarks [4] (*s15850.blif*)
- PicoJava benchmarks [22] (*pj1.blif*)
- ITC'99 benchmarks [11] (*b14.blif, b17.blif*)

Most of the test cases are included because of their relatively large size. Several smaller MCNC benchmarks were added to have circuits for which BDDs could be constructed. The above selection of circuits is available on the web [19].

The benchmark stats are given in Table 1: the number of inputs (column "ins"), outputs (column "outs"), and literals after sweeping (column "ff-lits"), which involves collapsing constants and single-input nodes as well as removing dangling nodes.

Unless specifically mentioned, runtimes are reported on a 1.6GHz laptop computer under Windows XP.

## 7.1 Experiment 1 (FRAIG Construction)

Table 1 compares the number of AIG nodes for three different methods of constructing the AIGs (the corresponding MVSIS command is given in parentheses):

- No strashing (*fraig -n*).
- One-level strashing (*fraig -r*).
- Strashing with functional reduction (*fraig*).

**Table 1. Node count after strashing and FRAIGing.**

| Name | ins | outs | ff-lits | *fraig -n* | *fraig -r* | *fraig* |
|---|---|---|---|---|---|---|
| des | 256 | 245 | 6084 | 5530 | 3895 | 3876 |
| c1355 | 41 | 32 | 992 | 550 | 537 | 537 |
| c6288 | 32 | 32 | 4675 | 2354 | 2339 | 2336 |
| i10 | 257 | 224 | 4355 | 2869 | 2436 | 2274 |
| s15850 | 611 | 684 | 7303 | 4344 | 4020 | 3884 |
| pj1 | 1769 | 1063 | 34533 | 18744 | 16834 | 16471 |
| b14 | 32 | 54 | 17388 | 9419 | 6300 | 5855 |
| b17 | 37 | 97 | 57311 | 32422 | 28916 | 27907 |
| Ratio | | | | 100.0 | 85.6 | 82.8 |

Table 1 shows that most of the reduction in the circuit size is achieved by one-level strashing (*fraig -r*) while FRAIGing (*fraig*) reduces the circuit size on average by 3% for the selected benchmarks. The reduction ratio due to FRAIGing is more significant (2-10 times) for highly redundant circuits, such as sets of reachable states and interpolants [18] represented as AIGs.

## 7.2 Experiment 2 (Runtime Tradeoffs)

Table 2 lists the runtime of FRAIGing with different options:

- No functional reduction (*fraig -r*).
- No SAT solver feedback (*fraig -f*).
- No equivalence test for sparse functions (*fraig -s*).
- Simulation with different number of bit patterns.

**Table 2. Runtime for FRAIGing with different options.**

| Name | *fraig-r* | *fraig -f* | *fraig-s* | *fraig* (2^8) | *fraig* (2^10) | *fraig* (2^12) | *fraig* (2^14) | *fraig* (2^16) |
|---|---|---|---|---|---|---|---|---|
| pj1 | 0.18 | 2.77 | 0.53 | 7.33 | 1.64 | 0.82 | 0.75 | 1.42 |
| b14 | 0.07 | 0.24 | 0.18 | 1.12 | 0.26 | 0.30 | 0.22 | 0.41 |
| b17 | 0.30 | 13.49 | 3.14 | 98.77 | 45.38 | 7.55 | 4.35 | 7.73 |

When no functional reduction is performed (*fraig -r*), FRAIGing becomes one-level structural hashing. Not using SAT solver feedback (*fraig -f*) slows down FRAIGing compared to other options when functional reduction is enabled. Not checking equivalence for sparse functions (*fraig –s*) speeds up FRAIGing but the resulting graph is not completely reduced since in some cases the equivalence test is skipped. (For space limitation, we omit the discussion of several possibilities for trading functional reduction for runtime.)

The remaining five columns compare FRAIGing with different number of simulation vectors: from 2^8 to 2^16. It is clear that too few simulation vectors perform badly because of the lack of distinguishing power while too many vectors lead to a slow-down as well because simulation is too time consuming. The default value set in the FRAIG package (~2^12 vectors) is a trade-off between the runtime and memory consumption.

## 7.3 Experiment 3 (Technology Mapping)

In Table 3 we compare the results of technology mapping with and without structural choices accumulated using the approach of "lossless synthesis" discussed in Section 6.1:

- Traditional mapping (*map*)
- Mapping with choices (*choice.script* + *map -rc*)

**Table 3. Technology mapping with different options.**

| Name | Traditional mapping | | | Mapping with choices | | |
|---|---|---|---|---|---|---|
| | Area | Delay | Runtime | Area | Delay | Runtime |
| des | 6565 | 13.60 | 0.50 | 7778 | 13.40 | 1.74 |
| c1355 | 1499 | 17.60 | 0.18 | 1149 | 14.10 | 0.48 |
| c6288 | 8394 | 82.60 | 1.13 | 8764 | 63.40 | 4.02 |
| I10 | 4332 | 36.10 | 0.50 | 4312 | 29.70 | 1.38 |
| s15850 | 6574 | 33.60 | 0.47 | 6447 | 28.70 | 1.48 |
| pj1 | 28726 | 44.80 | 2.19 | 29180 | 26.50 | 19.97 |
| b14 | 13458 | 76.60 | 1.55 | 10885 | 42.70 | 7.23 |
| b17 | 54568 | 69.50 | 4.88 | 53099 | 68.30 | 70.51 |
| Ratio | 1.00 | 1.00 | | 0.98 | 0.77 | |

We used a gain-based mapper with area recovery implemented in MVSIS environment. The standard cell library is *mcnc.genlib* from SIS distribution [21]. The first section of the table reports the results of mapping when the object graph is derived from the final netlist derived by running optimizing script *mvsis.rugged* applied to the original benchmark. The second section reports the results of technology mapping of five different versions of the same benchmark derived by *choice.script,* which is similar to *mvsis.rugged* with the additional capability of FRAIGing the intermediate networks into one object graph with choice nodes.

The runtime measurements in Table 3 refer to mapping alone. Network optimization by *mvsis.rugged* (the first part) and deriving choices by *choice.script* (the second part) took approximately the same time as technology mapping.

## 7.4 Experiment 4 (Equivalence Checking)

Table 4 reports the results of comparison of CEC using FRAIGs with the following equivalence checking options:

- o BDD: BDD-based CEC (*verify*).
- o SAT: Monolithic SAT-based CEC using "MiniSat" (deriving CNF from the miter of the two circuits and using SAT solver to prove it unsatisfiable).
- o SWEEP: SAT-based CEC with simulation-guided learning based on [16] (functional reduction is used as a post-processing step) (*sat_verify*).
- o FRAIG: FRAIGing (*fraig_verify*)
- o CSAT: The external checker based on CSAT [17].

**Table 4. Runtime comparison for CEC algorithms.**

| Name | BDD | SAT | SWEEP | FRAIG | CSAT | FRAIG |
|---|---|---|---|---|---|---|
| Des | 0.3 | 1.0 | 2.8 | 0.4 | 0.60 | 0.53 |
| c1355 | 10.0 | 0.2 | 0.1 | 0.1 | 0.07 | 0.06 |
| c6288 | - | - | 1.0 | 0.6 | 0.52 | 0.78 |
| i10 | 57.2 | 2.4 | 1.5 | 0.3 | 0.46 | 0.36 |
| s15850 | 6.3 | 1.5 | 3.3 | 0.5 | 0.76 | 0.41 |
| pj1 | - | - | 31.9 | 4.5 | 7.28 | 5.25 |
| b14 | - | - | 15.3 | 1.6 | 3.03 | 1.98 |
| b17 | - | - | 385.6 | 4.7 | 9.54 | 5.23 |

The two copies of the circuits used in this experiment are the original benchmark after sweeping and the same benchmark optimized by *mvsis.rugged*. The first part of the table reports the runtimes of the four equivalence checking options in MVSIS environment. The second part (comparison with CSAT) shows the runtimes on a Dell Precision 530 with 2.4 GHz Xeon CPU. The dash in the table indicates that the runtime exceeded 600 seconds.

## 8 Conclusions and Future Work

The paper shows that the AIGs powered by simulation and SAT lead to a representation called *functionally reduced AIGs* (FRAIGs), in which each AND-node is functionally unique by construction. An implementation of FRAIGs is proposed, which differs from other methods of functional reduction, such as BDD sweeping [13] and SAT sweeping [14][16], in that it detects and eliminates functionally equivalent AIG nodes on-the-fly.

FRAIGs can be used to unify all the stages of logic synthesis, from the functional representation of the original and optimized networks, through an improved technology mapping using multiple structural choices. At the same time FRAIGS enable transparent combinational equivalence checking to guarantee that the transformations made at each step during synthesis are correct.

The experiments indicate that the FRAIGs are more robust than BDDs in those applications that rely on partial canonicity and do not require quantification. A new application of FRAIGs is in lossless logic synthesis which structural information is accumulated during technology independent synthesis and used to provide choices during technology mapping.

The experiments also indicate that FRAIGs used for checking equivalence of large combinational circuits are on par with state-of-the-art academic equivalence checkers, such as CSAT [17].

The future work will include exploring other potential applications of FRAIGs in logic synthesis, technology mapping, and equivalence checking, as outlined in Section 6. An interesting application would be a tool to compare two networks for structural commonality using FRAIGs. Such a tool would be very useful for studying how a network changes structurally over the course of different transformations during logic synthesis.

## Acknowledgement

This research was supported in part by NSF contract, CCR-0312676, by the MARCO Focus Center for Circuit System Solution under contract 2003-CT-888, and by the California Micro program with our industrial sponsors, Fujitsu, Intel, Magma, and Synplicity.

The authors express their gratefulness to Ganapathy Parthasarathy of UC Santa Barbara for his help in running the experiments using the equivalence checker based on CSAT [17] reported in the second section of Table 4.

## References

[1] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking", In Advances in Computers, volume 58, 2003 ,Academic press.

[2] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient implementation of a BDD package", *Proc. DAC '90*, pp. 40-45.

[3] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[4] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. ISCAS '89*.

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. C-35(8), Aug 1986, pp. 677-691.

[6] J. Cortadella, "Bi-decomposition and tree-height reduction for timing optimization". *Proc. IWLS '02*, pp. 233-238.

[7] R. Drechsler, M. Thornton, "Fast and efficient equivalence checking based on NAND-BDDs", *Proc. VLSI '01*.

[8] N. Eén, N. Sörensson, "An extensible SAT-solver", *Proc. SAT '03.* http://www.cs.chalmers.se/~een/Satzoo/

[9] M. K. Ganai, A. Kuehlmann, “On-the-fly compression of logical circuits”. *Proc. IWLS ‘00*.

[10] E. Goldberg, M.Prasad, R.K.Brayton. “Using SAT for combinational equivalence checking”. *Proc. DATE ‘01*, pp. 114 -121.

[11] *ITC ’99 Benchmarks* http://www.cad.polito.it/tools/itc99.html

[12] K. Keutzer, “DAGON: Technology binding and local optimizations by DAG matching”, Proc. DAC ’87, pp. 617-623.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification”, *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[14] A. Kuehlmann, “Dynamic Transition Relation Simplification for Bounded Property Checking”. *Proc. ICCAD ‘04*.

[15] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, “Logic decomposition during technology mapping,” *IEEE Trans. CAD*, 16(8), 1997, pp. 813-833.

[16] F. Lu, L. Wang, K. Cheng, R. Huang. “A circuit SAT solver with signal correlation guided learning”. *Proc. DATE ‘03*, pp. 892-897.

[17] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases”, *Proc. DAC ’03*, pp. 668-673.

[18] K.L. McMillan, “Interpolation and SAT-based model checking”. *Proc. CAV ‘03*, pp. 1-13, LNCS 2725, Springer, 2003.

[19] A. Mishchenko. FRAIG source code and benchmarks. http://www.ee.pdx.edu/~alanmi/fraig.

[20] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[21] E. Sentovich, et al. “SIS: A system for sequential circuit synthesis”, *Tech. Rep. UCB/ERI*, M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[22] SUN Microelectronics. *PicoJava Microprocessor Cores.* http://www.sun.com/microelectronics/picoJava/

[23] S. Yang. *Logic synthesis and optimization benchmarks.* Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.