Lazy Man's Logic Synthesis

Wenlong Yang Lingli Wang
State Key Lab of ASIC and System
Fudan University, Shanghai, China
{allanwin@hotmail.com, llwang@fudan.edu.cn}

Alan Mishchenko
Department of EECS
University of California, Berkeley
alanmi@eecs.berkeley.edu

***Abstract*— Deriving a circuit for a Boolean function or improving an available circuit are typical tasks solved by logic synthesis. Numerous algorithms in this area have been proposed and implemented over the last 50 years. This paper presents a "lazy" approach to logic synthesis based on the following observations: (a) optimal or near-optimal circuits for many practical functions are already derived by the tools, making it unnecessary to implement new algorithms or even run the old ones repeatedly; (b) larger circuits are composed of smaller ones, which are often isomorphic up to a permutation/negation of inputs/outputs. Experiments confirm these observations. Moreover, a case-study shows that logic level minimization using lazy man's synthesis improves delay after LUT mapping into 4- and 6-input LUTs, compared to earlier work on high-effort delay optimization.**

I. INTRODUCTION

Multi-level logic synthesis [6][29] derives a multi-level logic circuit for a given Boolean function. Originally, the function is given as a truth table, Sum-of-Product (SOP), Binary Decision Diagram (BDD), etc, or a circuit of poor or unknown quality. The goal is to find a new circuit with the same functionality but smaller area, delay, power consumption etc., which can be efficiently mapped into standard cells or lookup tables, placed and routed, and realized as a high-quality hardware device.

For example, when logic synthesis is applied to a Boolean function represented as a BDD, we may first derive its SOP [15], next factor the SOP [5], and finally convert the factored form into a circuit. If the function is given as a circuit with a large number of inputs and outputs, the circuit can be restructured using circuit rewriting [4][17], resubstitution [19], logic sharing extraction [24], etc.

Different tools perform logic synthesis in different ways resulting in different circuit structures. These differences occur because implementations use heuristics and pursue different optimization goals. As a result, there is no "best" circuit structure for a given function. Sometimes even a suboptimal structure may be useful. For example, the result of technology mapping depends on the initial circuit structure [10]. In some cases, a suboptimal circuit before mapping leads to a better result after mapping.

This is why, in many applications, it is desirable to have multiple structural representations of each function. Hence, it is necessary to run different tools and different scripts and produce several structural snapshots of the same design. However, maintaining multiple tools or routinely applying multiple synthesis scripts is expensive and inefficient.

The proposed approach named "lazy man's logic synthesis" learns from the output of different tools applied to various designs and creates a library of AIG structures. When these structures are available and compactly recorded, there is no need to perform traditional logic synthesis. It is enough to check whether a function is already present in the library. The process of looking up is made fast by hashing functions using an affordable semi-canonical form described in this paper.

Consider trying to decompose a 6-input function into a network of 2-input gates. This may be done in many ways, especially when the function is non-trivial. Instead of implementing and running different decomposition methods [3][5][16], deriving several circuit structures and comparing them to choose the optimal one, we assume that this circuit structure is already present in some design before synthesis or is derived by some tool when it is applied to some design. It is not necessary to run that tool again or to try to find the design where the given function occurs. Instead, we rely on the fact that the "optimal" structure was available when the library was created and therefore it must be present in the library.

We analyzed several suites of public benchmarks and industrial designs and found that the number of semi-canonical classes of K-input functions ($K \leq 12$) appearing in these designs is large, but not prohibitively large. To perform this analysis, we computed K-input cuts [22] and collected semi-canonical classes of functions describing the output of each cut in terms of its inputs. We hashed the classes and counted how often they appear in the benchmarks before, during, and after synthesis using different logic synthesis scripts.

The reader may refer to the experimental results for a detailed description of this experiment. Here we mention only that, to cover 90% of the most frequently appearing 12-input functions, we need to maintain a library with 738,239 functional classes, whose truth tables take roughly 369 MBytes. This number is an underestimation, since for each functional class, several circuit structures need to be stored. However, it can be shown that for functions larger than 10 inputs, truth tables dominate memory requirements, which are realistic for modern workstations.

The contributions of this paper are:

- A new way of performing logic synthesis. Instead of deriving new circuit structures during runtime of a logic synthesis algorithm, the structure is retrieved from a precomputed library.
- A new way of accumulating circuit structures in a library and hashing them using a semi-canonical form, which trades speed of computation for reduced memory.
- Analysis of the logic synthesis benchmarks to show the number of frequently occurring semi-canonical classes of 12-input functions and how much memory is needed to represent a realistic library of these functions.

- Promising results of delay optimization after FPGA mapping into 4-LUTs and 6-LUTs, obtained using the proposed approach to logic synthesis.

The rest of the paper is organized as follows. Section 2 reviews previous work. Section 3 describes some background. Section 4 describes the algorithm. Section 5 represents the experimental results. Section 6 concludes the paper and outlines future work.

II. PREVIOUS WORK

Precomputing 4-LUT structures for FPGA mapping is proposed in [12]. A method for caching decomposable functions to up 16 inputs is used in [25] to speed up FPGA mapping. Both approaches precompute logic structures in terms of LUTs, rather than AIGs, as in the present work.

Technology-independent synthesis based on precomputed AIG structures is proposed in the following papers: [4] (4-input two-level structures), [17] (4-input cuts), [14] (5-input cuts). However, these methods do not use preexisting benchmarks or tools to generate useful structures, as in the present work. Moreover, they look only at 4- and 5-input functions, while the present work is geared to functions with 6-16 inputs.

Other methods of computing multiple logic structures useful for technology mapping were proposed, which led to the use of structural choices [13][10]. The difference here is that the number of choices used in [13][10] is relatively small (only one out of five nodes, on average, has structural choices in [10]) and the structures generated are of limited nature (only a small number of algebraic decompositions are considered in [13]).

On the application side, a closely related work is that on SOP balancing [21], which has the goal of reducing the AIG level count, as a preprocessing step before technology mapping into standard cells or LUTs. Both types of mapping have been tried in [21] with substantial delay improvements.

However, SOP balancing has several known limitations: for each cut, it considers only one logic structure derived by applying a delay-driven decomposition to the AND-gates and the OR-gates contained in the SOP. In contrast, the proposed approach is more general. It precomputes multiple circuit structures, including those not found by SOP balancing. Moreover, it scales to larger functions, compared to SOP balancing, which only works well for functions up to 8 inputs.

The following example illustrates the limitations of SOP-balancing. Consider function F = !c*!(b*!a) whose input arrival times are {0, 0, 1}. Figure 1a shows the structure found by SOP-balancing. Clearly, delay and area of the structure is 3 and 3, respectively. The proposed method will try different structures of the function and will chose the best one for the given input arrival times. If the precomputed library is good, the structure in Figure 1b will be found. The delay and area of the structure are 2, which are both better than SOP-balancing. Function F = !c*!(b*!a) is a small function, which appears often in the benchmarks. Similar situations may appear in other functions, resulting in logic structures with better delay than those found by SOP balancing.

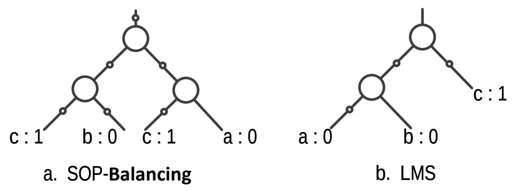

Figure 1: An AIG subgraph found in benchmark *s27.blif*, where SOP balancing loses to the proposed approach.

III. BACKGROUND

A Boolean network is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the gates. In this paper, we consider only combinational Boolean networks.

A node *n* has zero or more fanins, i.e. nodes driving *n*, and zero or more fanouts, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins in the current network. The primary outputs (POs) are a subset of nodes of the network, delivering the results to the environment. A fanin (fanout) cone of node *n* is a subset of all nodes of the network, reachable through the fanin (fanout) edges of the node.

A combinational And-Invertor Graph (AIG) is a Boolean network composed of two-input ANDs and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the AND and OR operations of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these two-input ANDs are added to the AIG manager in a topological order. The size (area) of an AIG is the number of its nodes; the depth (delay) is the number of nodes on the longest path from the PIs to the POs. The goal of AIG minimization is to reduce both area and delay.

Structural hashing of AIGs ensures that all constants are propagated and there is no two-input AND nodes with identical fanins (up to a permutation). Structural hashing is performed on-the-fly by hash-table lookups when AND nodes added to an AIG manager, which helps reduce the AIG size.

A cut *C* of a node *n* is a set of nodes of the network, called leaves of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the root of cut *C*. The cut size is the number of its leaves. A trivial cut of a node is the cut composed of the node itself. A cut is *K*-feasible if the number of leaves in the cut does not exceed *K*. A cut is dominated if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

Volume (area) of a cut is the number of unique AIG nodes found on the paths between the root and the leaves, including the root and excluding the leaves. Level (delay) of a cut is the number of AIG nodes on the longest path between the root of the cut and a primary input of the AIG. The concepts of delay, depth, and logic level are used interchangeably in this paper.

A *delay profile* of a cut is an ordered set of arrival times of each leaf of the cut. In this paper, delays are measured using the number of AIG logic levels from the primary inputs, and so delay profiles are arrays of non-negative integer numbers.

A local function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, *x*, of a cut of *n*.

Two functions are NPN-equivalent if one of them can be obtained from the other by negation and/or permutation of the inputs and outputs.

A positive minterm is a complete assignment of input variables, which makes the function evaluate to 1.

LMS is an abbreviation for "lazy man's logic synthesis".

IV. ALGORITHM

This section introduces the proposed algorithm.

A. Semi-canonical form

LMS is based on collecting, storing, and re-using circuit structures of Boolean functions with 6-12 input variables. Since the total number of completely-specified Boolean functions of N variables is 2^(2^N), storing all of them is infeasible for N > 4. For larger values of N, only the functions occurring frequently in the benchmarks should be considered. We call such functions *practical* and give frequency statistics in the experimental results section.

However, even the number of *practical* functions can be very large. To reduce this number and memory needed to store them in a library, they can be broken into equivalence classes using a canonical form. In this case, only representatives of equivalence classes are stored in the library.

In the proposed approach to logic synthesis, the canonical form should be computed in two situations: (1) during construction of the library when Boolean functions are canonicized and circuit structures for each class are recorded, and (2) during LMS based on the pre-computed library when the canonical form is computed for a Boolean function to find its circuit structures in the library.

Computing the complete NPN canonical form [9] may be too time-consuming, especially if this computation is used in the inner loop of an algorithm applied to a large number of functions of K-input cuts at each node during iterative optimization of logic networks. For this reason, in this paper, we propose to compute a semi-canonical form, which is more affordable than the NPN canonical form.

The idea is to order the input variables and the polarities of inputs/outputs using the number of positive minterms in the function and its cofactors with respect to each variable. A similar method is proposed in [1] and implemented using BDDs. In this work, all functional manipulation is performed using truth tables. The choice of truth tables is motivated by the fact that they require affordable amount of memory for functions up to 16 inputs and make the functional manipulation fast enough to be applied in the inner loop of logic synthesis algorithms.

```
truth table TruthSemiCanonicize(
        truth table F,
        unsigned  uCanonPhase,
        char * pCanonPerm )
{
   // canonize output
   count the number of 0s and 1s in the truth table of F;
   if ( number of 1s is more than number of 0s in F ) {
      complement F;
      record negation of the output in uCanonPhase;
   }
   // canonize variable phase
   count the number of 1s in the cofactors of F w.r.t. each variable;
   for each input variable of F
      if ( more 1s in negative cofactor than in positive cofactor ){
         change the variable's phase in F;
         record the change of varible's phase in uCanonPhase;
      }
   // canonicize variable permutation
   sort input variables by the number of 1s in their negative cofactors;
   permute inputs variables in F accordingly and
       record the resulting permutation in pCanonPerm;
   return F;
}
```

Figure 2: Generating semi-canonical form.

The pseudo-code is given in Figure 2. The procedure takes the truth table of a Boolean function and transforms it into a semi-canonical form. This form is the Boolean function of the representative of the equivalence class, to which the given function belongs. First, the output of the function is complimented based on the number of 1s in it. Then, the phase of each variable is decided by the number of the 1s in the negative and positive cofactors. Lastly, the variable ordering is determined by sorting variables using the number of 1s in their cofactors. Variables uCanonPhase and pCanonPerm record the changes in the truth table. They are returned to the calling procedure and later used to determine how to unpermute the subgraphs stored in the library, when this subgraph is used to synthesize a given function.

It was observed that, although the proposed semi-canonical form is less accurate than the complete NPN canonical form, it gives a reasonable trade-off between the speed of semi-canonical form computation and the conciseness of the resulting library. We estimated that computing the complete NPN canonical form for 12-variable functions would have increase the canonicization runtime by a factor of 10. Meanwhile, the memory footprint for the precomputed libraries would only be reduced by factor of 2.

B. Library representation

The pre-computed library of *practical* functions contains only non-redundant (often multiple) circuit structures collected for these functions. Although the number of practical functions can be large (say, one million), the number of precomputed structures can be an order of magnitude larger. This is why it is important to develop a compact memory model to store the precomputed structures.

All the optimized structures are stored in the same AIG manager. A library of N-input functions contains also functions with less than N-inputs, but they are remapped to strucutures which depend on the variables with the smallest variable indexes. Each circuit structure stored in the AIG is identified by a separate primary output.

When the library stored as an AIG, is loaded from file by the logic synthesis system, the following actions are performed:

- a hash table is created to hash the circuit structure of each primary output by its function, which serves as the semi-canonical form of the corresponding Boolean function;
- for each structure, the area and input-to-output delays are computed and stored with each primary output;
- optionally, areas and delays of structures belonging to the same equivalence class are compared and some of the structures are removed as dominated.

Since delay is the key aspect used in many logic synthesis applications, the arrival time of the output of each structure is required during synthesis for different structures and input arrival time profiles. To avoid multiple recursive traversals of the logic structure, input-to-output delays are pre-computed for each structure when the library is loaded. In this case, the resulting arrival time of the output is found by adding the arrival times of the inputs to the input-to-output delays of the structure, and finding the largest one over all inputs.

Figure 3 shows how input-to-output delays are computed when the library is loaded from file to accelerate the delay computation during synthesis.

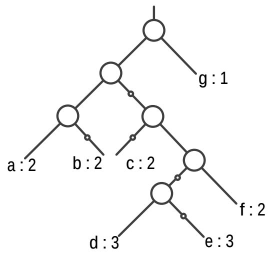

Figure 3: An illustration of the use of input-to-output delays.

The numbers in the figure show the delays from the corresponding input to the output of the structure, measured in terms of AIG nodes. Suppose the arrival time profile of the inputs listed from a to g, is {3, 2, 4, 5, 2, 3, 1}. Adding input-to-output delays to the input arrival times, the following arrival times of the output with respect to each input can be computed: {5, 4, 6, 8, 5, 5, 2}. Obviously, 8 is the largest number representing the output arrival time in this case.

C. Library construction

LMS does not derive new circuit structures during logic synthesis. Instead, it refers to a precomputed library and reuses the results generated by other synthesis tools. This is why building a high-quality library is important for LMS.

In this paper, we use the LUT mapper *if* in ABC as a structural cut browser to generate a fixed number of K-inputs cuts for each node in benchmark circuits. The circuits used for computing the library are the available benchmarks in their original form, as well as the same benchmarks after logic synthesis by a variety of tools and scripts. Using multiple snapshots of the same benchmark allows us to collect multiple circuit structures expressing the cut functions.

Figure 4 shows the pseudo-code of the procedure used by LMS to record the circuit structure of a cut in the library.

```
void **AddCut**(  cut C  )
{
   if ( cut C does not meet the requirements )
      return;
   compute Boolean function F of cut C as a truthtable;
   if ( F does not depend on some input variable of C )
      return;
   compute the semi-canonical form of F;
   find the corresponding functional class in the hash table;
   rebuild the structure of the cut in the library;
   if ( the structure already exists or is dominated )
      return;
   add a new primary output to store the structure in the hash table;
}
```

Figure 4: Adding structures to library.

Procedure AddCut() is executed on every cut generated by the cut browser. As a preprocessing step, the procedure removes cuts that do not fit the requirements, such as the restriction on the cut size and volume. If the structure of the cut has a redundancy or not every input variable appears in the support of the cut's function, the circuit structure of the cut is not considered.

When rebuilding the structure in the library, structurally isomorphic subgraphs are detected by structural hashing performed during AIG construction.

When library computation is performed in the "trim mode", the above procedure filters circuit structures using timing info. In this case, before the structure is inserted into the hash table, it is checked for dominance. A structure is dominated if its input-to-output delays are the same or larger, compared respectively to those of an existing structure. If the new structure is dominated, it is not added. On the other hand, if it dominates others, they are deleted from the hash table.

The data structure in the hash table keeps track of how often a semi-canonical form appeared during building libraries. If the user want to shrink the size of the library, a filtering can be applied, which deletes structures that appear infrequently.

Here are the commands for library construction implemented in ABC as part of this work:

- *rec_start*: Starts the LMS recorder.
- *rec_add*: Explores cuts of the current network using the cut browser and add useful AIG structures to the library.
- *rec_filter*: Removes those structures form the library, whose semi-canonical classes appear less frequently than the limit given by the user.
- *rec_merge*: Merges two previously computed libraries, by combining their circuit structures without the cut browser.
- *rec_ps*: Prints statistics for the currently loaded library.
- *rec_use*: Transforms the internal library to the current network in ABC, so it can be written into a file.
- *rec_stop*: Deletes the current library. Useful before the computation of a new library begins or when it is desirable to free the memory used by the currently stored library.

D. A case study of LMS: AIG level mininization

AIG level minimization is an important problem because AIGs are often used to represent logic networks during technology-independent logic synthesis.

Previous work on AIG level minimization [21] has shown that the reduction in AIG levels correlates well with the reduction of delay after technology mapping for both standard cells and LUT-based FPGAs.

In the same work [21] it was shown that AIG level can be optimized by applying SOP balancing to K-feasible cuts of each node. In this case, SOPs of the cut functions are computed and decomposed using the arrival times of cut leaves in order to minimize the arrival time of the cut root.

This case study demonstrates that the results produced by SOP balancing can be substantially improved when circuit restructuring is performed using a precomputed LMS library. The reasons for this are the following:

- SOP balancing uses only one AIG structure derived from an SOP of the function, while LMS has multiple structures to choose from, including those derived by SOP balancing, if these were harvested during library computation.
- LMS scales to larger functions (up to 12 inputs and more), while SOP balancing works in practice only when the number of variables does not exceed 8.

In our experiments, both LMS and SOP balancing have been performed using 6-input functions. It means that currently only the first reason account for the improved results.

```
aig PerformAIGLevelMinimizationUsingLMS (
   aig G,    // G is an And-Inverter Graph
   int K,    // K is the cut size
   int C )   // C is the number of cuts at each node
{
  for each node n in G, in a topological order {
    compute C structural K-input cuts of n;
      for each cut {
        compute Boolean function of the cut as a truth table;
        find the best structure in the library;
        if ( there is no structure for this function )
          mark the cut to ensure it is not selected as best cut for n;
        else if ( the cut leads to smaller AIG level than the best cut )
          save the cut as the best cut;
      }
  for each node n in G, in a topological order {
    if ( root node AIG level is reduced using the best cut )
      replace current AIG structure of node n by the delay-optimal
        AIG structure found for the best cut in the library;
  return G;
}
```

Figure 5: Perform LMS on subject graph.

The pseudo-code in Figure 5 shows the use of LMS for AIG level minimization. Structural K-feasible cuts are enumerated and restructuring is applied to each cut. If an AIG structure found in the library can improve the logic level of the root node, this structure is used to rewrite the AIG of the node.

V. EXPERIMENTAL RESULTS

The LMS algorithm is implemented in ABC [2][7]. The LUT mapper *if* [20] in ABC is used (a) as a cut browser for computing the libraries and (b) as a mapper in the case study on AIG level minimization.

To derive the libraries, benchmarks from MCNC [8], ISCAS [28], ITC [11], Altera [23], and other public sets are used.

The experimental results are divided into three categories:

- Library coverage for 12-input functions
- Library construction for 6-input functions
- Delay optimization for 4-LUT and 6-LUT mapping

A. Library coverage

This experiment was performed to show that LMS has practical memory requirements for functions whose support size is limited by 12 inputs. In this experiment, semi-canonical classes of all functions appearing in the cuts of the benchmark circuits *without synthesis*, were collected and the frequency of their appearance was recorded. It took about two days to collect these results on one computer.

The results which contain the truth tables and their frequencies in the benchmarks are publicly available at [31].

The results are shown in Table 1 and, schematically, in Figure 6. For example, to achieve 90% coverage for 12-input functions appearing in the benchmarks, the library should include 738,239 semi-canonical classes whose truth tables take 369MBytes of storage.

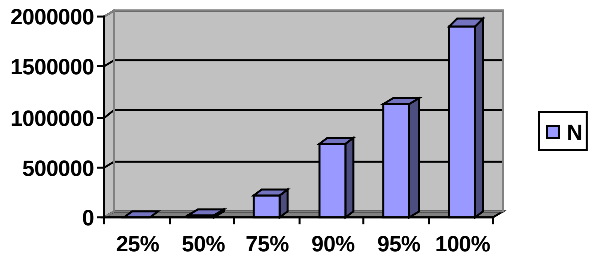

Figure 6: The number of equivalence classes needed to represent the given percentage of the most frequently occurring functions in the benchmark circuits.

In practice, having 90% coverage is enough for logic optimization because the remaining functions may be seen as composed of smaller functions included in the library.

In a separate experiment shown in Table 2, semi-canonical classes of functions with support size limited by 12, appearing in the cuts of the benchmark circuits *before and after synthesis*, were collected and their occurrences were counted.

The table shows that the number of classes has increased approximately 10x as a result of applying the ABC script shown in the next section. We conjecture that much of the observed increase is because AIG-based synthesis performs unnecessarily aggressive circuit restructuring. This motivates developing AIG-transformations based on a richer set of primitives, for example, multi-input ANDs, multi-input XORs, MUXes, and majority gates.

It can also be noted that the additional structures generated by synthesis, compared to those present in the original benchmarks circuits, are not required if LMS is applied before synthesis, as in the case-study of Section IV-D.

B. *Constructing library for 6-input functions*

The goal of this experiment is to derive a 6-input library used in the case study described in Section IVD with results reported in Section VC. Before the experiment, *rec_start* is called. After the experiment, *rec_use* is called, followed by writing the library into a file. The following script is used to collect circuit structures generated by logic synthesis in ABC:

- read file; st; rec_add; *// add the original AIG structures*
- dc2; rec_add; *// add structures derived by dc2*
- if -K 8; bidec; st; rec_add;
- if -K 8; mfs; st; rec_add;
- if -K 8; bidec; st; rec_add;
- if -g -K 6; st; rec_add;*// add structures created by SOP balancing*
- if -g -K 6; st; rec_add;

To keep the library compact, trim mode of LMS is turned on. As a result, circuit structures whose input-to-output delays are dominated by those of other structures are filtered out. In the end, classes appearing less than 3 times are filtered, too. The library is also publicly available at [31].

Table 3 shows statistics of the resulting library, containing 5,687,661 AIG structures classified into 1,249,229 semi-canonical equivalence classes. The AIGER file containing this library takes 77.85MBytes. The computation took several days needed to apply the above script to the available benchmarks.

It should be noted that the large number of structures included is the library is because our current implementation of the library construction does not handle multi-input gates. As a result, different tree decompositions of each such gate are recorded as separate subgraphs. This limitation will be addressed by future work.

To get an even wider spectrum of circuit structures, other synthesis tools, besides ABC, can also be used, for example SIS [26], BDS [27], FBDD [30], etc. The resulting circuits can be added to the library by reading them into ABC during library computation, followed by commands *st; rec_add*.

C. *Using LMS to optimize delay after LUT mapping*

The delay optimization experiments reported in this section were run on a workstation with Intel Xeon Quad Core CPU and 256 GBytes RAM. The resulting networks were verified by the SAT-based CEC engine [18] (command *cec* in ABC).

The algorithm for AIG level minimization from Section IVD is implemented as command *if –y –K <num> -C<num>*. Command line switches used in this experiment are:

- *if* is the priority-cut-based LUT mapper [20];
- *-g* enables delay optimization by SOP balancing [21];
- *-y* enables delay optimization by LMS (the present work);
- *-K <num>* specifies the cut size;
- *-C <num>* is the number of cuts used at each node.

The input of the command is the current AIG in ABC. The output is a delay-optimized mapped network.

Two sets of benchmarks are used in this paper: 20 large MCNC benchmarks [8] and 10 large Altera benchmarks [23].

LUT mapping was performed by the following scripts:

- Map: *st; resyn2; if -K 4 or 6*
- MapC: *st; resyn2; dch -f; if -K 4 or 6*
- SOPBC: *st; if -gm -K 6; st; resyn2; dch -f; if -K 4 or 6*
- LMSC: *st; if -ym -K 6; st; resyn2; dch -f; if -K 4 or 6*

The first script (Map) is a typical synthesis and mapping flow in ABC. It begins by deriving an AIG, applying delay-oriented script *resyn2*, which performs technology-independent synthesis without increasing the AIG logic level, followed by mapping into either 4-LUTs or 6-LUTs. MapC is similar to Map but it additionally computes structural choices using *dch*. SOPBC and LMSC use SOP balancing and LMS, respectively, to optimize the AIG before MapC is applied.

In the scripts, the cut size of both SOP balancing and LMS is set to 6. This way any difference in the results of mapping is due to different logic structures used by these algorithms.

The results for Altera benchmarks are reported in Tables 4-5 by measuring LUT level and LUT count. When compared against traditional mapping (Map), LMS reduced delay by 37% (26%) with the same area increase of 13% for 4-LUTs (6-LUTs). When compared against mapping with choices (MapC), LMS reduced delay by 9% (7%) with area increase of 9% (6%) for 4-LUTs (6-LUTs). When compared against SOP balancing, in both cases the delay gain is close to 17% with an area increase of 2% (5%).

Tables 6-7 show the results of applying LMS to MCNC benchmarks. The delay improvements in this case are similar to those for industrial designs, but the area penalty was higher.

The runtime of *if –y* is similar to that of *if –g* and exceeds the runtime of regular LUT mapping (*if*) roughly by a factor of 3. The runtime of the whole script, such as MapC, is not substantially different when *if –y* and *if –g* are used as a preprocessing step because the runtime in this case is dominated by computing structural choices (command *dch*).

VI. CONCLUSIONS

In the existing logic synthesis tools, useful circuit structures are derived by applying dedicated algorithms at runtime. In contrast, we harvest and re-use circuit structures produced by different tools working on a variety of benchmarks.

The "lazy" approach to logic synthesis is made practical by

- mapping functions into semi-canonical equivalence classes with isomorphic sets of optimal circuit structures;
- relying on AIGs with structural hashing to compactly store precomputed libraries in memory and on disk;
- using truth tables to manipulate Boolean functions during both library construction and circuit restructuring.

As a case-study, the proposed approach was applied to AIG level minimization as a way to improve delay after FPGA mapping. For industrial benchmarks, the delay after LUT mapping was reduced by 17% (18%) for LUT4 (LUT6) with the area penalty of 2% (5%), compared to the recent work on SOP balancing [21]. This is a remarkable result, given the fact that SOP balancing is a high-effort delay-optimization algorithm. It clearly shows that the “lazy” approach often finds better logic structures than SOP balancing.

Future work may include:

- finding better tradeoffs between delay and area (currently, delay improvement comes at a cost of an increase in area, which is not acceptable in some design flows);
- extending the approach to work for multi-input gates (currently, only two-input AND gates are used);
- performing AIG level optimization with larger cuts (currently, only 6-input cuts are used while the proposed approach is shown to be practical up to 12 inputs at least).

ACKNOWLEDGMENTS

This work has been supported by

- National Natural Science Foundation of China, under grants 61131001 and 61171011.
- National High Technology Research and Development Program 863, China, under grant 2009AA012201.
- Fudan University Key Laboratory Senior Visiting Scholarship program.
- Contacts from Semiconductor Research Corporation and industrial sponsors: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Microsemi, Oasys, Real Intent, Synopsys, Tabula, and Verific.

REFERENCES

[1] A. Abdollahi and M. Pedram, “A new canonical form for fast boolean matching in logic synthesis and verification”. *Proc. DAC’05*, 379-384.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] V. Bertacco and M. Damiani. "The disjunctive decomposition of logic functions". *Proc. ICCAD '97*, pp. 78-82.

[4] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD ’04*, pp. 42-49.

[5] R. Brayton and C. McMullen, “The decomposition and factorization of Boolean expressions,” *Proc. ISCAS ‘82*, pp. 29-54.

[6] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, “Multilevel logic synthesis”, *Proc. IEEE*, Vol. 78, Feb.1990.

[7] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[8] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” *Proc. ISCAS ’89*, pp. 1929-1934.

[9] D. Chai and A. Kuehlmann, “Building a better Boolean matcher and symmetry detector”. *Proc. DATE 2006*, pp. 1079-1084.

[10] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *IEEE TCAD’06*, Vol. 25(12), pp. 2894-2903.

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

[12] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu, "Generating efficient libraries for use in FPGA resynthesis algorithms". *Proc. IWLS'10*, pp. 147-154.

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

[14] N. Li and E. Dubrova, “AIG rewriting using 5-input cuts”, *Proc. IWLS’11*.

[15] S. Minato: "Fast generation of prime-irredundant covers from binary decision diagrams," *IEICE Trans. Fundamentals*, Vol. E76-A, No. 6, pp. 967-973, June 1993.

[16] A. Mishchenko, B. Steinbach, and M. A. Perkowski, "An algorithm for bi-decomposition of logic functions", *Proc. DAC '01*, 103-108.

[17] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[18] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", Proc. ICCAD '06, pp. 836-843.

[19] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*, pp. 15-22.

[20] A. Mishchenko, S. Cho, S. Chatterjee, R. Brayton, “Combinational and sequential mapping with priority cuts”, *Proc. ICCAD ’07*, pp. 354-361.

[21] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing", *Proc. ICCAD'11*, pp. 375 - 382.

[22] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA’98*, pp. 35-42.

[23] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton. "Benchmarking method and designs targeting logic synthesis for FPGAs", *Proc. IWLS '07*, pp. 230-237.

[24] J. Rajski and J. Vasudevamurthy, “The testability-preserving concurrent decomposition and factorization of Boolean expressions”. *IEEE TCAD’92*, vol. 11(6), pp. 778-793.

[25] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, “Mapping into LUT structures”, *Proc. DATE’12*.

[26] 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.

[27] C. Yang and M. Ciesielski. ”BDS: a BDD-based logic optimization system”, *IEEE TCAD’02*, vol. 21(7), pp. 866–876.

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

[29] L. Wang, and A. E. A. Almaini, “Multilevel logic simplification based on containment recursive paradigm”, *IEE Proceedings Computers and Digital Techniques*, 2003, Vol.150, No.4, pp, 218-226.

[30] D. Wu and J. Zhu, “FBDD: a folded logic synthesis system”, *Proc. DAC’05*, pp. 746-751.

[31] https://skydrive.live.com/redir.aspx?cid=76d4b8991df82cf3&resid=76D4B8991DF82CF3!152&parid=root

Table 1: The number of equivalence classes of N-input functions needed to represent the given percentage of the most-frequently appearing functions in the benchmark circuits without synthesis.

Inputs \ Percentage	20%	25%	50%	75%	90%	95%	100%
2	1	1	1	1	2	2	3
3	1	1	2	4	6	8	21
4	2	2	5	19	65	114	1478
5	6	8	38	177	659	1481	27523
6	14	23	177	1172	5713	14203	125725
7	46	93	885	6859	36383	84165	329630
8	76	157	1686	14551	73739	155383	506497
9	46	130	3602	34697	158466	302800	778698
10	156	365	5807	48748	218515	409054	897730
11	353	861	14231	119713	434679	717726	1332731
12	793	1786	27104	223943	738239	1128006	1907484

Table 2: The same as Table 1, except the benchmarks were synthesized by an ABC script.

Inputs \ Percentage	20%	25%	50%	75%	90%	95%	100%
2	1	1	1	1	1	2	3
3	1	1	2	3	6	8	21
4	2	2	4	17	46	86	1703
5	6	8	36	160	920	2593	248581
6	19	28	209	2112	19475	75428	2755032
7	77	150	2254	31453	337943	1293513	8677854
8	187	431	9660	188108	2092479	5890862	17485043
9	453	1192	41216	859287	6887725	14086676	28687179
10	747	2054	120198	3218612	16669810	29203415	44814184
11	2471	7332	425701	8607987	32104100	50026675	67949249
12	7379	22789	1358918	19991157	60917017	83584641	106252265

Table 3: Statistics of the precomputed 6-input library of semi-canonical classes.

Inputs	Semi-canonical Forms	Structures	Ratio
2	3	3	1.00
3	32	88	2.75
4	2430	12673	5.22
5	98208	471973	4.81
6	1148556	5202924	4.53
Total	1249229	5687661	4.55

Table 4: The results of delay optimization after LUT mapping for Altera benchmarks (4-LUTs).

Design	4-LUT levels				4-LUT count			
	Map	*MapC*	*SOPBC*	*LMSC*	*Map*	*MapC*	*SOPBC*	*LMSC*
carpat.blif	68	68	53	40	38856	39842	42092	42371
fp_operators.blif	119	116	88	76	17902	17401	18538	18800
oc_video_compression_systems_dct_opt.blif	19	19	19	14	8995	9114	12221	11158
oc_video_compression_systems_jpeg_opt.blif	20	19	17	13	10967	10940	14590	14321
radar20_opt.blif	39	38	23	16	16834	17216	17717	20663
screen_saver_cyclone.blif	18	18	16	17	35627	35183	35614	35900
sudoku_check.blif	11	11	10	10	20998	20774	21094	21286
top_rs_decode.blif	43	43	31	24	31381	30729	30798	30926
umass_weather.blif	38	38	25	17	15821	15734	18250	18292
uoft_raytracer.blif	70	69	58	30	33294	33852	37118	40147
Ratio	1.00	0.99	0.80	0.63	1.00	1.00	1.11	1.13

Table 5: The results of delay optimization after LUT mapping for Altera benchmarks (6-LUTs).

Design	6-LUT levels				6-LUT count			
	Map	*MapC*	*SOPBC*	*LMSC*	*Map*	*MapC*	*SOPBC*	*LMSC*
carpat.blif	35	35	35	27	29826	31098	32243	33321
fp_operators.blif	67	66	57	50	10541	11118	12005	11982
oc_video_compression_systems_dct_opt.blif	10	10	12	9	7349	7566	8816	8606
oc_video_compression_systems_jpeg_opt.blif	10	10	12	9	7796	7822	8365	9537
radar20_opt.blif	20	20	13	10	12351	12705	12871	14964
screen_saver_cyclone.blif	13	12	12	12	27129	27113	27503	27373
sudoku_check.blif	7	7	7	7	14542	14355	14707	15501
top_rs_decode.blif	24	24	20	16	21271	21324	21668	21615
umass_weather.blif	24	24	16	10	12196	11990	13287	14123
uoft_raytracer.blif	36	35	31	19	26128	26666	29802	31356
Ratio	1.00	0.99	0.92	0.74	1.00	1.02	1.08	1.13

Table 6: The results of delay optimization after LUT mapping for MCNC benchmarks (4-LUTs).

Design	4-LUT level				4-LUT count			
	Map	*MapC*	*SOPBC*	*LMSC*	*Map*	*MapC*	*SOPBC*	*LMSC*
alu4.blif	7	7	7	7	1137	1117	1149	1145
apex2.blif	8	7	7	7	1295	1325	1319	1373
apex4.blif	6	6	6	6	1028	1016	1041	1030
bigkey.blif	3	3	3	4	1259	1256	1592	1443
clma.blif	15	14	13	12	4447	3995	4498	4989
des.blif	6	6	6	6	1215	1211	1265	1240
diffeq.blif	14	14	12	9	841	908	911	945
dsip.blif	3	3	4	3	925	1109	929	1132
elliptic.blif	18	18	15	10	2020	2150	2125	2327
ex1010.blif	7	7	7	7	3481	3317	3427	3384
ex5p.blif	6	6	6	6	827	795	905	880
frisc.blif	23	23	18	14	2234	2182	2293	2349
misex3.blif	6	6	6	6	1093	1053	1060	1041
pdc.blif	9	9	8	8	2953	2839	2839	2836
s298.blif	12	11	11	11	904	897	896	881
s38417.blif	10	9	9	9	3450	3402	3512	3398
s38584.1.blif	9	9	8	7	3829	3686	3701	3590
seq.blif	6	6	6	6	1219	1198	1199	1219
spla.blif	8	8	8	8	2628	2529	2535	2390
tseng.blif	13	13	10	10	756	800	801	826
Raito	1.00	0.98	0.95	0.90	1.00	1.00	1.03	1.04

Table 7: The results of delay optimization after LUT mapping for MCNC benchmarks (6-LUTs).

Design	6-LUT level				6-LUT count			
	Map	*MapC*	*SOPBC*	*LMSC*	*Map*	*MapC*	*SOPBC*	*LMSC*
alu4.blif	6	5	5	5	802	853	892	901
apex2.blif	6	6	6	6	1023	1008	1007	1091
apex4.blif	5	5	5	5	784	771	801	800
bigkey.blif	3	3	3	3	579	579	689	692
clma.blif	10	10	9	9	3363	3049	3575	3722
des.blif	5	4	5	4	855	888	880	946
diffeq.blif	8	8	8	6	637	667	648	725
dsip.blif	3	3	3	3	689	689	689	901
elliptic.blif	10	10	9	7	1796	1941	1886	2094
ex1010.blif	6	6	6	6	2555	2520	2608	2625
ex5p.blif	5	5	5	5	560	543	685	685
frisc.blif	13	13	12	9	1743	1723	1797	1834
misex3.blif	5	5	5	5	810	786	793	777
pdc.blif	7	7	7	7	2175	2086	2081	2028
s298.blif	9	8	8	8	648	655	645	651
s38417.blif	7	7	7	6	2629	2622	2621	2668
s38584.1.blif	6	6	6	6	2371	2428	2394	2414
seq.blif	5	5	5	5	888	876	893	935
spla.blif	6	6	6	6	1910	1853	1860	1794
tseng.blif	8	8	6	6	648	694	689	748
Raito	1.00	0.98	0.96	0.91	1.00	1.00	1.04	1.08