

A Hash-based Approach for Functional Regularity Extraction During Logic Synthesis

Angelo P. E. Rosiello¹ Fabrizio Ferrandi¹ Davide Pandini² Donatella Sciuto¹

¹ DEI – Politecnico di Milano

² STMicroelectronics

{rosiello,ferrandi,sciuto}@elet.polimi.it davide.pandini@st.com

Abstract

Performance, power, and functionality, yield and manufacturability are rapidly becoming additional critical factors that must be considered at higher levels of abstraction. A possible solution to improve yield and manufacturability is based on the detection of regularity at logic level. This paper focuses its attention on regularity extraction, after technology independent logic synthesis, to detect recurring functionalities during logic synthesis and thus constraining the physical design phase to exploit the regular netlist produced. A fast heuristic to the template identification is proposed and analyzed on a standard set of benchmarks both sequential and combinational.

I. INTRODUCTION

Modern semiconductor fabrication processes for nanometer technologies allow designing high-density and high-speed circuits. High-performance ASICs and general-purpose microprocessors are normally datapath-intensive. In datapath-dominated circuits, a portion of logic (i.e., a *bit-slice*) is repeated multiple times. Not only are these bit-slices regular with respect to each other, but interconnections between these replicated bit-slices are also typically very regular [1].

It is now well accepted that the next big EDA challenge is Design for Manufacturing (DFM), which can be defined as a set of techniques adopted to estimate, control, and improve the yield and robustness of a circuit prior to fabrication [2]. One of the most effective DFM approaches is to generate regular layouts [3]. A regular layout increases manufacturability because regular patterns drastically reduce the number of design rules that must be considered during layout generation, and Resolution Enhancement Techniques (RET) favor the use of a limited number of regular shapes.

The concept of regularity extraction during logic synthesis is not new in EDA, and several authors have proposed techniques for regularity extraction from the behavioral and structural description of datapath-dominated circuits. Those approaches can be distinguished into two main categories: the first one addresses the problem of covering a circuit given a set of pre-defined library templates [5][6][7][8][10][11], while the second one is based on the automatic generation of templates from a given circuit, followed by the covering phase [12].

An early work on regularity extraction focused on datapath circuits was presented by Rao and Kurdahi [6], where the idea of regularity-based clustering is applied to datapath synthesis, and the core of the synthesis process is a cluster-identification procedure. Regularity extraction is thus based on identification of a few distinct logic components (i.e., templates), which can be replicated to cover the entire circuit. The same authors proposed a string-matching algorithm to identify templates in a circuit, given an initial library of predefined templates [7]. In this approach, the solution is quite dependent on the user-specified template library, and does not consider the systematic template generation.

Similar techniques are used by graph mining approaches like Subdue [8]. The aim of [8] is not the detection and the improvement of regularity but the minimization of the graph representation, as multi-level synthesis area minimization approaches similarly do.

Corazao et al. [9], starting from the behavioral-level circuit description, proposed to identify not only the complete, but also partial template instances from a given template library. A different approach to regular datapath function identification was presented by Arikati and Varadarajan [10], where a signature-based approach is used to discover regularity given some initial datapath functions. All datapath functions larger than a predefined min-size are identified as regular functions. Similarly, Nijseen et al. [11] suggested to use small components (i.e., latches) as initial templates, and then to expand these seeds to obtain larger templates. Obviously, these approaches depend on the initial choice of components to be used as templates. For these methods, the availability of a good template library is important for an effective exploration of the trade-offs amongst multiple optimization criteria, such as area, timing, and power.

Chowdhary et al. proposed a different approach based on extracting the regularity inherent in the high-level circuit description [12], where two algorithms that generate a sufficiently large set of templates from the circuit HDL (Hardware Description Language) representation were presented. These algorithms create templates with a tree structure, and a special class of multi-output templates, with the limitation that every template output belongs to the transitive fanin cone of a particular output (the principal output of the template). The main limitation of this approach stems from the template generation algorithms, where two particular classes of templates are extracted: Trees (TREEs) and Single Principal-Output Subgraphs (SPOGs).

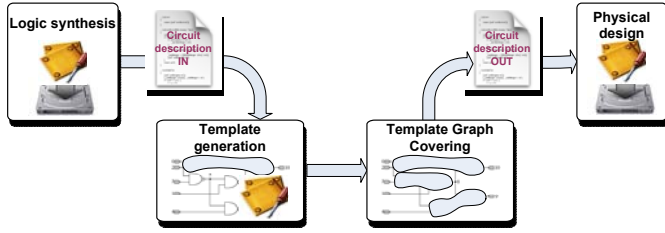


Figure 1. Regularity extraction in a standard design flow

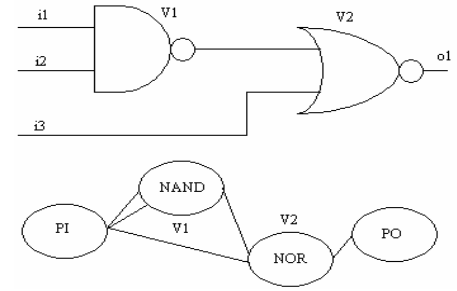


Figure 2. Graph representation of a logic circuit

Therefore, multi-output templates are either partially considered, or are too restricted for an effective regularity extraction.

Despite these approaches to regularity extraction, regularity preservation throughout the design flow has been rarely considered [13]. More recently, Kutzchebauch and Stok [14] addressed this problem. Their technique models regularity by means of global and local regularity metrics. *Global regularity* refers to regular structures that improve the cost functions during the physical design process, whereas *local regularity* is based on regularity signatures, which help to compare and control the changes in the logic network performed by any transformation applied during logic synthesis. Different signatures are proposed in [14], i.e., logic, timing, area, and power signatures, to satisfy different optimization objectives. This paper introduces a more general algorithm that allows an efficient management of large multi-output functions, as necessary in industrial design environments.

Regularity extraction can be applied at different design problems. In this paper we focus our attention on **functional regularity extraction** since by preserving the regularity extracted during logic synthesis we are able also to control the regularity at the structural and geometrical level throughout the design flow.

Therefore, we exploit regularity to improve the results of logic synthesis, thus constraining the physical design phase to work on a hierarchical netlist where ungrouping or flattening is not allowed. Hence, the standard design flow is modified by introducing the Template Generation and Template Graph Covering functions, as shown in Figure 1.

Our approach for functional regularity extraction tries to identify which is the recurring functionality present in the design and it aims at extracting a hierarchical structure that will be preserved during the following physical design phases. It is based on the assumption that only a good automatic generation of templates may guarantee good results in terms of design regularity. The main contribution of this paper is the definition of a fast heuristic for automatic identification of a large set of templates, but whose complexity is still bounded by $O(V^2)$, where V is the number of vertices of the considered circuit. The templates obtained with our algorithm also include all those generated with TREES and SPOGs in [12], thus allowing a wider range of possible circuit covers. Moreover, we also present a general technique to identify subgraphs by cryptographic hash functions, thus reducing the problem of matching two equivalent subgraphs to the detection of hash collision occurrences.

The paper is organized as follows: Section II introduces the regularity extraction problem at logic level, while Section III

discusses the proposed hash-based approach for the regularity extraction problem. Section IV reports the experimental results obtained by the proposed approach demonstrating the effectiveness of the method with respect to previously published results. Finally, some conclusions are drawn.

II. FORMULATION OF THE REGULARITY EXTRACTION PROBLEM

The input for the regularity extraction problem is a structural description C modeled by a labeled directed graph $G(V, E)$ [15], where the set of Boolean functions corresponds to the set of vertices V , while the interconnections among the circuit components are modeled by the set of edges $E \subset V \times V$. All the primary inputs and outputs are connected to two special vertices, PI and PO respectively. Since each graph vertex represents a Boolean function of C , each vertex is labeled by the function $l: V \rightarrow l_1, \dots, l_n$, which maps the circuit logic functions onto the graph vertices. Similarly to [12] or [7], each fanin edge of every vertex $v \in V$ is assigned a unique index i , with $1 \leq i \leq f_v$, where f_v is the in-degree for v , and the indices are derived from the function $k: E \rightarrow 1, \dots, k_f$. Figure 2 shows the corresponding graph representation of a given logic circuit. A subgraph $G_i(V_i, E_i)$ of G is *consistent* if and only if (iff) $V_i \subset V$, $E_i \subset E$, and G_i does not include any disconnected subgraph. Hence, a consistent subgraph of G is a subcircuit of C . Two subgraphs G_i and G_j are equivalent iff:

- They are isomorphic;
- The functionalities of corresponding vertices are the same;
- The indices of the corresponding edges are the same.

The *template generation problem* can be defined as follows:

Template Generation Problem: *Given a circuit C expressed by a graph $G(V, E)$, find the consistent subgraphs of G , which are not completely included in any other subgraph and with at least another equivalent subgraph in G .*

In most cases, maximizing the area of a template T_i extracted from a circuit C implies decreasing the number of its instances and vice versa, thus the problem requires finding the best trade-off between these conflicting objectives. Actually, the best solution strongly depends on the context, so there is no optimal solution defined beforehand. Area maximization impacts on circuit optimization, since larger templates would be better refined during technology mapping and physical design. On the other hand, maximizing the number of template instances would reduce the overall design

effort. Our method identifies the recurring functions embedded into the circuit description implying more regular structures at physical level.

So, given a set of templates, the *graph covering problem* can be expressed as:

Template Graph Covering Problem: *Given a circuit C represented by a graph $G(V, E)$ and a set of templates T, find a cover of G such that $\forall T_i \in T, \text{area}[T_i] \wedge |T_i|$ are maximized.*

In this definition, $\text{area}[T_i]$ represents the area of template T_i and $|T_i|$ is the number of instances of template T_i in circuit C. Since the problem of finding a graph optimum cover is NP-complete [15], we heuristically solve the problem by choosing at each iteration one template, and removing (covering) all the corresponding subgraph instances in the graph.

III. HASH-BASED REGULARITY EXTRACTION ALGORITHM

The next two subsections detail how we propose to solve the Template Generation problem and the Template Graph Covering Problem while the third one discusses the computational complexity of the Template Generation problem.

A. Template Generation

The Template Generation algorithm needs an efficient technique to perform the subgraph matching and in particular we have developed a technique to easily detect when two subgraphs are different. The proposed approach verifies that two graphs are different if the corresponding hash-based signatures representing the graphs are different. In contrast, when the signatures coincide, in order to guarantee that a bad collision does not occur, we verify the isomorphism of the extracted templates with standard equivalence checking techniques.

During regularity extraction having a fast function able to identify differences is valuable in terms of computational performance. Similarly to [7] we describe a graph using a string starting from a lead vertex, but with a different structure. In particular, the backward string representation for a lead vertex of a subgraph is described by a comma-separated string with the type of lead vertex followed by the number of fanins, fanouts, and by the string associated with the fanin vertices, considered as if they are backward lead vertices. For example, the string obtained for the whole subgraph V1,V2 of Figure 2, where V2 is the lead vertex, is NOR,2,1, Φ ,NAND,2,1, Φ , Φ , Φ , where Φ is the empty string. We can define a forward string representation for a lead vertex of a subgraph as a comma-separated string with the type of lead vertex followed by the number of fanins, fanouts, and by the string associated with the fanout vertices considered as if they are forward lead vertices. For example, the string obtained for the subgraph V1,V2 of Figure 2, where V1 is the lead vertex, is NAND,2,1, Φ ,NOR,2,1, Φ , Φ .

However, this representation is exponential with the number of vertices, therefore, we introduce a cryptographic hash-based approach. We consider cryptographic hash functions because of their strong collision resistance property and computational efficiency. Moreover, a cryptographic hash function returns a constant output string (the signature) whose length requires minimal waste of memory. For example, MD5 [16] produces strings of 128 bits, while SHA-1 [16] requires strings of 160 bits.

Obviously, there is a correlation between the output string length of the hash function and the probability of having a collision when two subgraphs are different (bad collisions). For MD5 the probability of a bad collision is $1/2^{64}$, while it is $1/2^{80}$ for SHA-1. To guarantee that a bad collision does not occur during the coverage step, we verify the isomorphism of the extracted templates with standard functions. The backward signature for a lead vertex u of a subgraph is:

$$s_B(u) = H(u.f, u.\#in, u.\#out, \Phi | s_B(u.v_in_1) | \dots | s_B(u.v_in_{u.\#in})),$$

where H is the string-based hash function adopted, $u.f$ is the type of the logic function associated with the leading vertex u , $u.\#in$ is the number of inputs of u , $u.\#out$ is the number of fanouts of u , Φ is the empty string, and $u.v_in_i$ is the i -th fanin vertex to be considered iff it belongs to the subgraph. Similarly, a forward signature for a lead vertex u of a subgraph can be defined as:

$$s_F(u) = H(u.f, u.\#in, u.\#out, \Phi | s_F(u.v_out_1) | \dots | s_F(u.v_out_{u.\#out})),$$

where H is the string-based hash function adopted, $u.f$ is the type of the logic function associated with the leading vertex u , $u.\#out$ is the number of fanouts of u , Φ is the empty string and $u.v_out_i$ is the i -th fanout vertex. Even when referring to the same subgraph, the forward and backward signatures are different. In fact, for the template-generation algorithm, different data structures are used to store backward and forward results, and to be merged at the end of their execution. Consider the example in Figure 2, where a simple NAND-NOR circuit must be signed. To obtain the signature of the NAND and NOR gates we use the hash function previously described:

$$x = H(\text{NAND}, 2, 1, \Phi), \quad (1)$$

$$y = H(\text{NOR}, 2, 1, \Phi). \quad (2)$$

To compute the backward signature of the whole subgraph it is sufficient to concatenate (1) to the last argument in (2) as:

$$z = H(\text{NOR}, 2, 1, \Phi | x),$$

where z is the backward signature of the circuit subgraph shown in Figure 2. Similarly, the forward signature w of the subgraph can be obtained as follows:

$$y = H(\text{NOR}, 2, 1, \Phi),$$

$$w = H(\text{NAND}(2, 1, \Phi | y)).$$

Similarly to the approach presented in [7] we use signature based template identification, but we have a lower computational complexity. The K-formula used in [7] maintains the references to the vertices (the K-formula of subgraph V1,V2 where V1 is the lead vertex is *V1V2) and needs to be manipulated to verify the equivalence of two subgraphs through the equivalence of two K-formula. Therefore, besides the time needed to compute the template, [7] needs also some time to perform graph matching. On the other hand, our approach greatly simplifies this second phase exploiting hash matching to quickly detect if two subgraphs are different. AIG [18] based equivalence checking approach suffers from the same disadvantages of K-formulas, since they require a preprocessing step to take into account the relations between the inputs of the two subgraphs. The proposed approach, besides extending the templates identified, has been designed to simplify the most common case of subgraph mismatching.

The template generation procedure requires four main steps:

1. Initialization
2. Backward expansion;
3. Forward expansion;
4. Template merging.

Let's analyze each step in detail.

1) Initialization Function

An efficient method to simplify the overall extraction procedure is to build a list of compatible vertices that are reliable seeds for the template generation function. Two vertices are inserted into the compatibility list L iff:

- The vertices have the same functionality;
- The vertices have the same number of fanins.

The initialization procedure can extract at most $V(V-1)/2$ compatible vertices.

2) Backward Expansion Function

The compatibility list, after the initialization step, contains the pair of compatible vertices, which are inputs of the backward expansion function. For each vertex pair, representing the subgraph roots, their children (i.e., the fanins) are considered for the signatures of the whole subgraphs. Consider two children x and y . If the signatures of the new subgraphs are equivalent (i.e., $x = y$), then also the descendants are added, until the signatures of the subgraphs are different. However, if the signatures containing the children are different, then one child at a time is considered for the computation of the new signatures. In this way, it is possible to extract also equivalent strips of the subgraphs. If no more equivalent subgraphs can be extracted, then the function is terminated. For each pair of vertices in the compatibility list, the function will extract the largest equivalent subgraphs and the corresponding template signature. The signatures and subgraphs are stored in a table (*backwardTemplate_map*) having as primary key the subgraph signatures, which identify the templates, and as argument the subgraph vertices (or their identifiers in the graph). When considering a couple of equivalent subgraphs, if their signature already exists in the template table, then the template was previously found. Hence, only new templates are added to the table. A relation between roots and their corresponding subgraphs is defined by storing in the table *roots_BackwardMap* the roots and their subgraph signatures. To keep track of the couple of roots, associated with their equivalent subgraphs, we connect in another table (*roots_BackwardMap*) the subgraph signatures with the roots, so that it will be possible to perform a merge between the results of the backward and forward expansion.

To illustrate the execution of the algorithm, consider the example in Figure 3. After the initialization function the compatibility list contains the following pairs of compatible vertices: (V1,V4); (V1,V6); (V1,V9); (V3,V8); (V4,V6); (V4,V9); (V5,V10); (V6,V9).

As a significant case we consider only the pair (V3,V8). The first iteration of the while() loop yields: firstSubG=(V3,V1,V2) and secondSubG=(V8,V6,V7). Since the function *are_equivalent()* finds the two subgraphs not equivalent, the algorithm considers the children of V3 and V8 individually. The subgraphs (V3,V1) and (V8,V6) have the same following signature, i.e.

Backward Expansion Function

Input: vertex u , vertex v

firstSubG = empty

secondSubG = empty

while continue_while subgraphs_are_equivalent **do**

firstSubG = getBackSubG(u , firstSubG);

secondSubG = getBackSubG(v , secondSubG);

if are_equivalent(firstSubG, secondSubG) **then**

store_backward_signature (u , v , firstSubG, secondSubG);

end if

end while

$$z = \underbrace{H[NOR,2,2,\Phi]}_{V3 \wedge V1} \left[\underbrace{H[AND,2,1,\Phi]}_{V1} \right] = \underbrace{H[NOR,2,2,\Phi]}_{V8 \wedge V6} \left[\underbrace{H[AND,2,1,\Phi]}_{V6} \right]$$

and node PI is reached from every incoming edge. Hence, (V3,V8) with the signature of the identified equivalent subgraphs (i.e.: z) are stored in the *roots_backwardMap*, while firstSubG and secondSubG in *backwardTemplate_map*. It is worth noticing that there is at most one extracted template for every couple of compatible vertices, and since there are at most $V(V+1)/2$ pairs of vertices in the compatibility list, the number of the overall extracted templates is bound by V^2 .

3) Forward Expansion Function

After the backward expansion function is completed, the forward expansion is called. Algorithmically the forward expansion function proceeds as the backward expansion for each couple of compatible vertices, but in the opposite direction. For example, for the pair of vertices (V3, V8) the first iteration of the while() loop yields: firstSubG=(V3,V4,V5) and secondSubG=(V8,V9,V10). Since the subgraphs are equivalent the function *are_equivalent()* returns TRUE. At the next iteration node PO is reached from every outgoing edge, therefore (V3,V8) with the signature of the identified equivalent subgraphs are stored in *roots_forwardMap*, while firstSubG and secondSubG in *forwardTemplate_map*. Similarly to the backward expansion, the number of extracted templates is bounded by V^2 , where V is the number of graph vertices.

4) Merge Function

After the backward and forward expansion, the subgraphs thus obtained are merged. This step allows the identification of

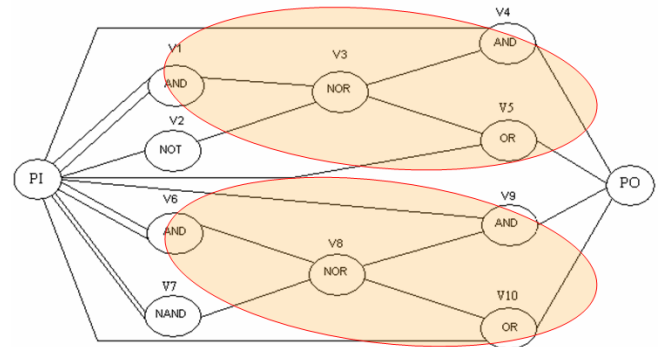


Figure 3. Illustrative example circuit

particular structures, such as stars. In this case the signature of the whole template is computed as the hash of the concatenation of the backward and forward signatures. For example, the signature for the template having as seeds the couple of vertices (V3,V8) in Figure 3 is computed as:

$$\begin{array}{c}
 \text{Backward Signature: } V3 \wedge V1 \\
 \overbrace{H\{H[NOR,2,2,\Phi | H(AND,2,1,\Phi)]\}}^{V1} \\
 H[NOR,2,2,\Phi | \underbrace{H(AND,2,1,\Phi)}_{V4} | \underbrace{H(OR,2,1,\Phi)}_{V5}] \\
 \text{Forward Signature: } V3 \wedge V4 \wedge V5
 \end{array}$$

The signature obtained by the merge function is then considered for the next iteration. Note that the type of template shown in the example, i.e. (V3,V1,V4,V5) as shown in Figure 3, is not detectable by the extraction algorithms presented in [12] since it is not a TREE or a SPOG.

B. Template Graph Covering

Given the set of templates generated during the Template Generation phase the graph covering will be solved heuristically by choosing a template from the set of templates, and then remove all instances of the chosen template from the graph. In case the identified templates or the remaining vertices are under a given threshold, measured in terms of area or number of vertices, the template graph covering ends, otherwise the template generation phase restarts on the reduced graph. This iterative procedure identifies a set of templates that experimentally give good results [7][12]. The key point of this procedure is the template selection. We use two different heuristics, the first one proposed in [12] while the second one proposed in this paper:

- *LFF* (Largest Fit First) – given a set of templates T , LFF selects the template T_i with the largest area.
- *KF* (Koala Fast) – the set of templates T_i is computed starting from a set of compatible pairs of seeds (defined in the initialization phase) with at most $l \in \mathbb{N}$ pairs. As LFF, *KF* selects the template T_{i_l} with the largest area but since it considers a pre-fixed and constant number of compatible pairs of seeds it returns a suboptimal solution with respect to *LFF*. However, *KF* is able to analyze larger circuits than LFF since it has a linear time complexity as shown next.

C. Template Generation Computational Complexity Analysis

The Template Generation phase is performed by generating a possible template for each pair of vertices of the graph associated with the circuit. Therefore, the complexity of the template generation is:

$$O(V^2) \cdot \text{Complexity}(B+F) + \text{Complexity}(M),$$

where $\text{Complexity}(B+F)$ is the complexity of the Backward and Forward expansion, while $\text{Complexity}(M)$ relates to the Merge function. Backward and Forward complexity is $O(V)$ since at most each vertex of the graph is considered once. On the other hand, the complexity of the Merge function is $O(V^2)$, since it is equivalent to the intersection of the forward and backward map that can have at most $O(V^2)$ elements. Therefore, the complexity of the template generation phase is $O(V^3)$. Under the same

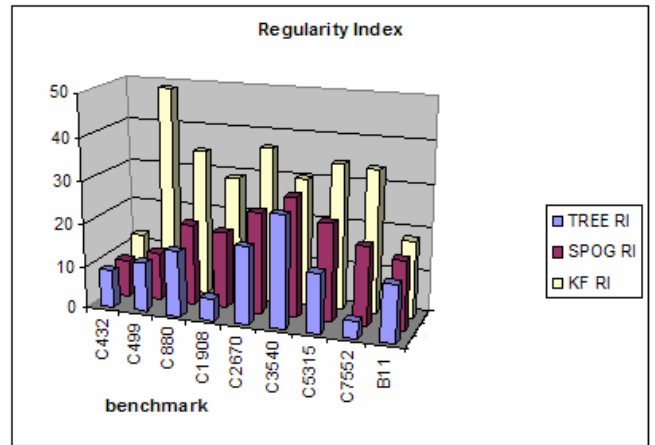


Figure 4. Regularity Index

assumptions of [12], the time complexity is $O(V^2 \cdot \log V)$ for TREES and $O(V^3)$ for SPOGs with respect to $O(V^3)$ for our proposed approach. On the other hand, the *KF* heuristic has a complexity of $O(V)$, since the number of pairs is constant and the Merge function works on constant size maps.

IV. EXPERIMENTAL RESULTS

The approach presented in this paper and the algorithms in [12] have been implemented in our synthesis framework called *Panda*. The framework takes as input a circuit description in *bench* or in *EDIF* format. We extracted the regularity from several ISCAS85 benchmarks and from two ITC99 circuits. Table 1 reports the results of our proposed regularity extraction algorithm, compared to TREES and SPOGs. We computed the area of each extracted template using the standard-cell technology mapping functionality in ABC [19] using the library *mcnc.genlib* from the SIS distribution. To quantify the reduction of the design effort we computed the *regularity index* measure (RI), as suggested in [12]. The regularity index is a measure defined by the area of all templates in the cover, given by $\sum_i area[S_i]$, as a percentage of the total area of the graph G , given by $\sum_i |S_i| \cdot area[S_i]$ post-mapping. A small regularity index implies that a low effort is needed during synthesis and layout stages, while a large template implies that a better optimization can be achieved during synthesis and layout [13]. Table 1 reports also the covering index (CI), i.e. the percentage of the gates of the circuits covered by the selected templates, the number of gates of the largest template found (Largest), the average number of gates (ANG), the number of templates (#t) and subgraphs (#s) identified, and the size of the set of compatible seeds analyzed by the *KF* heuristic (WS). Since very small templates do not represent interesting solutions, we constrain the analysis to use template at least of size (MS). For every benchmark we obtain a better regularity and covering indices with respect to both TREES and SPOGs. It is important to notice that for the largest circuits, both TREES and SPOGs could not terminate because of the unmanageable size given their computational complexity (e.g., B14).

Finally, we used equivalence checking to verify all the final benchmark implementations using Synopsys *Formality*TM.

Circuit (#gates)	TREE				SPOG				KOALA FAST					
	CI	RI	Largest (ANG)	#t (#s)	CI	RI	Largest (ANG)	#t (#s)	CI	RI	Largest (ANG)	#t (#s)	WS	MS
C432 (160)	43,1	9	5 (4,6)	4 (15)	40,6	9	5 (4,6)	4 (15)	78,1	12,7	11 (8,3)	4 (15)	200	3
C499 (202)	100	11,7	17 (5,3)	5 (58)	100	11,4	17 (5,3)	5 (58)	100	49,1	93 (93)	1 (2)	200	3
C880 (383)	68,4	15	17 (6,7)	9 (39)	65	19,2	17 (7,8)	10 (33)	68,7	34,8	37 (17,5)	7 (15)	200	3
C1908 (880)	56,7	5	8 (4,6)	7 (108)	61,8	18,1	23(7,8)	16 (72)	63,1	28,8	38 (12,6)	15 (44)	200	3
C2670 (1193)	64	18	32 (5,1)	24(151)	63,3	23,8	40 (6,1)	28 (128)	68,4	36,8	127 (19,4)	18 (42)	200	3
C3540 (1669)	59,7	26,2	63 (9)	32(111)	58,6	28	64 (9,8)	35 (103)	61,6	30	114 (25,7)	27 (56)	800	3
C5315 (2307)	71,9	13,8	14 (5,1)	48(322)	73,6	22,9	163(7,4)	54 (237)	88	34,3	39 (11,1)	65 (183)	300	3
C7552 (3512)	11	4,1	21 (12)	7 (32)	31	18,3	49(16,1)	27 (72)	54,7	33,6	79 (21,6)	43 (89)	200	8
B11 (757)	23,4	13,3	25 (21,2)	4 (8)	22,5	16,3	25(19,8)	5 (9)	34,1	18	39 (18,4)	7 (14)	250	8
B14 (9767)	-	-	-	-	-	-	-	-	23,8	13,2	178 (37,1)	29 (59)	250	5

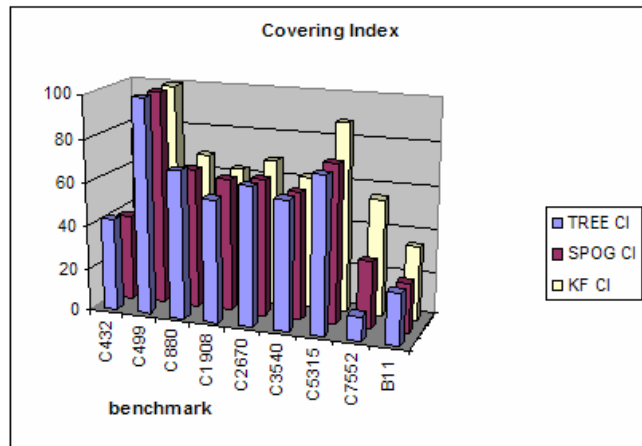


Figure 5. Covering Index

V. CONCLUSIONS

In this work, we proposed a new approach for regularity extraction during logic synthesis that can be seamlessly integrated into a standard top-down design flow. This includes a new algorithm for regular logic structure identification based on hash functions. We believe that regular design will be mandatory at 45nm and below, and regularity-aware logic synthesis will be a critical enabler of design methodologies based on regular fabrics for high yield and manufacturability. Regularity must be preserved at different levels of abstractions, and our approach extracts and preserves logic regularity at logic level. Future work will include physical design based on the regular macro-blocks derived from the templates generated with the approach described in this paper.

ACKNOWLEDGEMENTS

This work was supported by Central CAD and Design Solutions of STMicroelectronics.

REFERENCES

- [1] S. Das and S. P. Khatri, "An Efficient and Regular Routing Methodology for Datapath Designs Using Net Regularity Extraction," *IEEE Trans. on Computer-Aided Design*, vol. 21, pp. 93-101, Jan. 2001.
- [2] A. Nardi and A. L. Sangiovanni-Vincentelli, "Logic Synthesis for Manufacturability," *IEEE Design & Test of Computers*, vol. 21, pp. 192-199, May-Jun. 2004.
- [3] V. Kheterpal, V. Rovner, T. G. Hersan, D. Motiani, Y. Takegawa, A. J. Strojwas, and L. T. Pileggi, "Design Methodology for IC

Manufacturability Based on Regular Logic Bricks," in *Proc. Design Automation Conf.*, Jun. 2005, pp. 353-358.

- [4] T. Jhaveri, L. T. Pileggi, V. Rovner, and A. J. Strojwas, "Maximization of Layout Printability/Manufacturability by Extreme Layout Regularity," in *Proc. of SPIE Microlithography*, Feb. 2006, vol. 6156.
- [5] T. Kutzschebauch, "Efficient Logic Optimization Using Regularity Extraction," in *Proc. Intl. Conf. on Computer Design*, Oct. 2000, pp. 487-493.
- [6] D. S. Rao and F. J. Kurdahi, "Partitioning by Regularity Extraction," in *Proc. Design Automation Conf.*, Jun. 1992, pp. 235-238.
- [7] D. S. Rao and F. J. Kurdahi, "On Clustering for Maximal Regularity Extraction," *IEEE Trans. on Computer-Aided Design*, vol. 12, pp. 1198-1208, Aug. 1993.
- [8] L.B. Holder, D.J. Cook, S. Djoko, Substructure discovery in the subdue system, in *Proc. Intl. of Knowledge Discovery in Databases Workshop (KDD-94)*, 1994, pp. 169-180.
- [9] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey, "Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis," *IEEE Trans. on Computer-Aided Design*, vol. 15, pp. 877-887, Aug. 1996.
- [10] S. R. Arikati and R. Varadarajan, "A Signature Based Approach to Regularity Extraction," in *Proc. Intl. Conf. on Computer-Aided Design*, Nov. 1997, pp. 542-545.
- [11] R. X. T. Nijssen and C. A. J. Van Eijk, "Regular layout generation of logically optimized datapaths," in *Proc. Intl. Symp. on Physical Design*, Apr. 1997, pp. 42-47.
- [12] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "A General Approach for Regularity Extraction in Datapath Circuits," in *Proc. Intl. Conf. on Computer-Aided Design*, Nov. 1998, pp. 332-338.
- [13] V. N. Kravets and K. A. Sakallah, "M32: A Constructive Multilevel Logic Synthesis System," in *Proc. Design Automation Conf.*, Jun. 1998, pp. 336-341.
- [14] T. Kutzschebauch and L. Stok, "Regularity Driven Logic Synthesis," in *Proc. Intl. Conf. on Computer-Aided Design*, Nov. 2000, pp. 439-446.
- [15] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, McGraw-Hill, 1994.
- [16] R. Rivest, "The MD5 Message-Digest Algorithm," *RFC 1321*, MIT LCS and RSA Data Security, Inc., April 1992.
- [17] National Institute of Standards and Technology. Secure Hash Standard, 1995.
- [18] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. Design Automation Conf.*, Jul. 2006, pp. 532-536.
- [19] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. December 2005 Release. <http://www-cad.eecs.berkeley.edu/~alanmi/abc>