A Synthesis Approach for Coarse-grained, Antifuse-based FPGAs

Chang Woo Kang, Ali Iranli, and Massoud Pedram

University of Southern California
Electrical Engineering Department – System
3740 McClintock Ave. EEB-314
Los Angeles, CA, 90089

***Abstract* — In this paper, we present a synthesis technique targeted toward coarse-grained, antifuse-based FPGAs. A macro logic cell, in this class of FPGAs, has multiple inputs and multiple outputs. A library of small logic cells can be generated from this macro cell, and used to map the target netlist. First, we calculate the minimum number of macro logic cells required to map a given circuit by using either a dynamic programming or a linear programming technique. Given this minimum number of macro logic cells, we introduce an interconnect-aware clustering algorithm that assigns logic cells to individual macro cells so as to minimize the routing costs. Alternatively, a timing slack-driven clustering algorithm is presented where timing criticalities of nodes in a network are calculated and used to determine the final packing into the macro cells so as to minimize the number of the macro cells on the critical paths. When compared to results from a commercial tool, our two synthesis techniques reduce the number of macro logic cells by 12%, and the maximum depth by 35%, respectively.**

***Index Terms*—Antifuse, Clustering, Coarse-grained, FPGA**

I. Introduction

Field programmable gate arrays (FPGAs) can provide many advantages over standard cells, in terms of satisfying market demand while assuring configurability. Fast time-to-market satisfies industry designers to keep up with newly created standards, and configurability provides flexible hardware on demand of both new standards without fabricating a new chip.

FPGAs usually consist of small, configurable basic elements, connected by rich programmable interconnects [1]. Since routing resources grow faster than on-chip logic resources, routing resources account for the major portion of the device's overall area and delay [2]. In addition, speed and area-efficiency of an FPGA are directly related to the granularity of its logic block [3]. While coarse-grained blocks have long internal logic delays, they can reduce the placement and routing stress by having fast local routing and significantly reduce external routing. Typically, synthesis tools prefer "gate array-like" fine-grained architectures; however, fine-grained

FPGA architectures generally yield a very poor performance due to the long delays resulting from building functions, with multiple levels of gates and slow interconnect elements. Coarse-grained architecture gives the tools the needed degrees of freedom for the high logic utilization benefits of a fine-grained architecture, without sacrificing the high performance benefits of coarse-grained, high fan-in architecture. Recently, FPGA manufacturers have introduced coarse-grained architectures. Examples of such devices are the pASIC3 [6], the Xilinx Virtex [7], and the Apex and Flex from Altera [8].

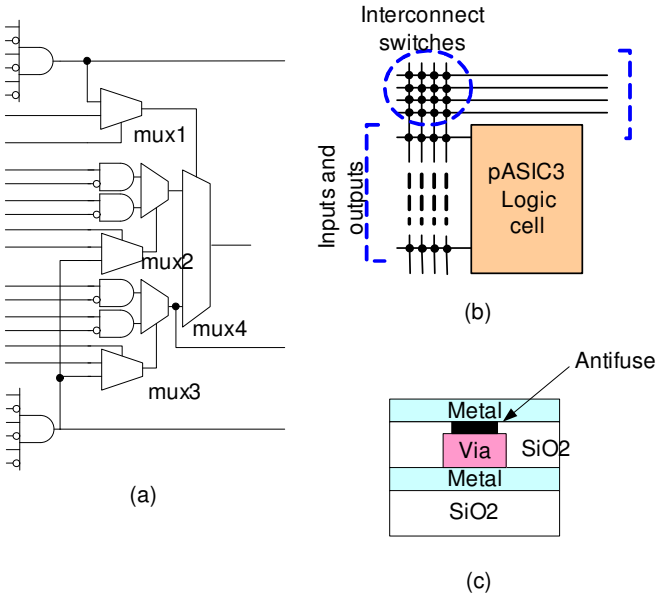

Fig. 1. Coarse-grained, antifuse-based FPGA: (a) pASIC3 logic cell, (b) architecture, and (c) antifuse switch.

Coarse-grained, antifuse-based FPGAs have emerged as a promising technology for limited space, high speed, and low power. The architecture consists of interconnects, antifuse switches, and programmable logic cells as shown in Fig. 1(b). Fig. 1(a) shows a coarse-grained, antifuse-based pASIC3 logic cell, which has 26 inputs and four outputs. The function of the logic cell is determined by the logic levels applied to the inputs of the AND gates and multiplexers. The high logic capacity and wide fanin of the logic cell accommodate many user functions with a single level of logic delay. Because the architecture provides tremendous flexibility, with small hardware overhead, coarse-grained, antifuse-based FPGAs demands

highly intelligent CAD algorithms.

Antifuse-based FPGAs are one time programmable logic devices. The antifuse is initially in a high impedance state and is transformed into a low impedance metal-to-metal link when programmed. Fig. 1(c) illustrates the cross-sectional view of the antifuse programming technology. The antifuse element is formed by depositing a high resistance layer (> 1GΩ) of amorphous silicon above a tungsten via a plug that would otherwise bridge the insulation between the two metal layers. When a programming voltage is applied to a selected via, a direct metal-to-metal link is formed by permanently transforming the silicon to a low resistance state. A typical resistance for a programmed connection is about ~30 Ω. The size of the programmed link is physically smaller than that of a via. The small size of the interconnect coupled with the high dielectric constant of the via material ensures that unprogrammed links exhibit capacitive loading < 1fF [6].

Clustering refers to the task of grouping logic gates in the circuit netlist and assigning each group to a configurable logic block in the FPGA array (in the case of our target architecture, this means packing gates into pASIC3 logic cells). Circuit clustering is an important technique for coarse-grained FPGAs. First, clustering can reduce the complexity of large circuit designs by a significant factor. Second, clustering can improve the quality of the results of other operations such as placement and routing

In this paper, we present area-driven clustering techniques with considerations given to the routing complexity of inter-cluster interconnects and the delays in terms of the number of pASIC3 logic cells on critical paths. Although we target a specific logic cell architecture, e.g., pASIC3 FPGA, our method can be applied to similar type of coarse-grained, antifuse FPGAs with slight modification. For example, QuickLogic has recently launched new coarse-grained antifuse FPGA devices, named Eclipse II and PolarPro, which have a complex logic description and architecture. Our techniques can be easily modified to target these coarse-grained antifuse FPGA devices.

We extract library cells from the pASIC3 logic cell and then mapped a network. After technology mapping, we calculate the minimum number of macro logic cells, which is a lower-bound, to cover the network by either dynamic programming or linear programming. Then, we cluster network by using the minimum number of available pASIC3 logic cells with considerations of interconnect and delay. The goal of the interconnect-aware clustering is to minimize the number of inter-cluster interconnects, while, for slack timing-driven clustering, we minimize the number of logic cells on the critical paths.

A preliminary version of this work appeared in [4][5]. These papers presented different ways of computing the minimum number of macro logic cells and an algorithm for doing performance-driven clustering with node replication. The present manuscript unifies the findings of [4][5] and extends them by describing clustering algorithms targeting the minimum number of inter-cluster interconnections and the minimum number of macro logic cells on the critical paths without any logic replication. New results and discussions are added to support these extensions.

This paper is organized as follows. In Section II, a brief background on clustering techniques for FPGAs is provided. We present the proposed CAD tool flow and the procedure for creating a library set in section III. The lower-bound calculation algorithm for the minimum number of logic cells is presented in section IV. The area-driven clustering algorithms with interconnect awareness and delay optimization, are presented in section V. In section VI, the experimental results are provided. Finally, we conclude in section VII.

II. BACKGROUND

FPGAs have clusters with basic logic elements (BLEs) and those BLEs are ready to be programmed to implement specific functions. Therefore, the technology mapping locates a feasible portion of circuits and implements the functions of that portion into those BLEs. For two different types of FPGAs, various mapping techniques have been developed. Cong and Ding developed FlowMap [9] that guarantees to produce depth-optimal mapping solutions. An extensive survey of existing SRAM-based FPGA mapping techniques is provided in [10]. For antifuse-based FPGAs, Boolean matching techniques have been used for technology mapping and those research results on technology mapping for antifuse logic cells have been reported in [11]. Boolean matching is therefore a key enabler for antifuse-based FPGA mapping. Lai et al. in [12] proposed a Boolean matching algorithm and introduced matching filters for speedup. A more comprehensive review is provided in [13].

Clustering techniques for two different technologies, SRAM and antifuse, are also somewhat different. SRAM-based FPGAs have logic clusters, each of which consists of multiple BLEs. Clustering BLEs have typically two constraints: the number of BLEs in a logic cluster and the number of inputs of a logic cluster. On the other hand, clustering gates for an antifuse-based logic cell means that all gates in a cluster must be able to realize functions completely within a logic cell, which is pASIC3 in this research. Since it is too difficult to map a network with multiple output logic cells, the macro logic cell must be divided into small base gates and library cells are generated from those base gates. After technology mapping, the library cells must be packed to fit the macro logic cell. Therefore, the constraint for packing is more stringent.

As far as we know, there is not any prior work on clustering techniques targeting coarse-grained anti-fuse FPGAs and that is why our discussion of the prior work will

focus on clustering algorithms for SRAM-based FPGAs, from which we have borrowed some concepts and ideas.

Clustering techniques for SRAM-based FPGAs have been evolving [13]-[18]. The algorithms have relied on good seed selection and smart gain functions to evaluate the gain of absorbing a neighbor node according to their objectives. The RASP [13] is a general synthesis and mapping system for SRAM-based FPGA. The clustering algorithm is based on a sequence of maximum weighted matching operations on a compatibility graph, which yields the proper grouping of LUTs into programmable logic blocks (PLBs). For each step, a compatibility graph is formed in which vertices represent the partial PLBs (initially LUTs) that will be considered for grouping at this step. An edge is formed between two vertices, if the two corresponding partial PLBs can be grouped into one. Then, weights are assigned to edges to guide the matching algorithm to select the best merging of partial PLBs. VPACK [15] is a clustering algorithm to minimize both the number of logic clusters and the number of used inputs to each cluster. Minimizing used inputs for each cluster is important to develop a routable design. The algorithm constructs each cluster sequentially. First, a seed BLE is chosen, which has the most used inputs among the currently unclustered BLEs. The inputs are a scarce resource. Thus, VPACK greedily selects the BLE that shares the most inputs and outputs with the cluster being constructed.

T-VPACK [16] is based on VPACK algorithm [15]. Its optimization goal is minimizing the number of external connections (connections between clusters) on the critical path. Since the external cluster routing delay is much larger than the local routing inside a cluster, minimizing the number of routing on critical paths can improve delay significantly. The algorithm consists of two steps: static timing analysis and clustering. During the static timing analysis step, criticalities of interconnects are computed. During the clustering phase, selecting a seed BLE and attracting BLEs take place. The seed BLE is unclustered but has the most critical connection in the circuit. RPACK [17] is a routability-driven packing algorithm, which first identifies routability factors, prioritizes these factors into an improved clustering cost function. The beauty of iRAC is that it [18] packs closely connected components together, achieves spatial uniformity in the clustered design using Rent's Rule [23], and reduces the external routing requirement in clustered FPGAs. It alleviates routing congestion for clustered FPGAs by absorbing as many small nets into clusters as possible, and depopulating clusters according to Rent's rule in order to achieve spatial uniformity in the clustered netlist.

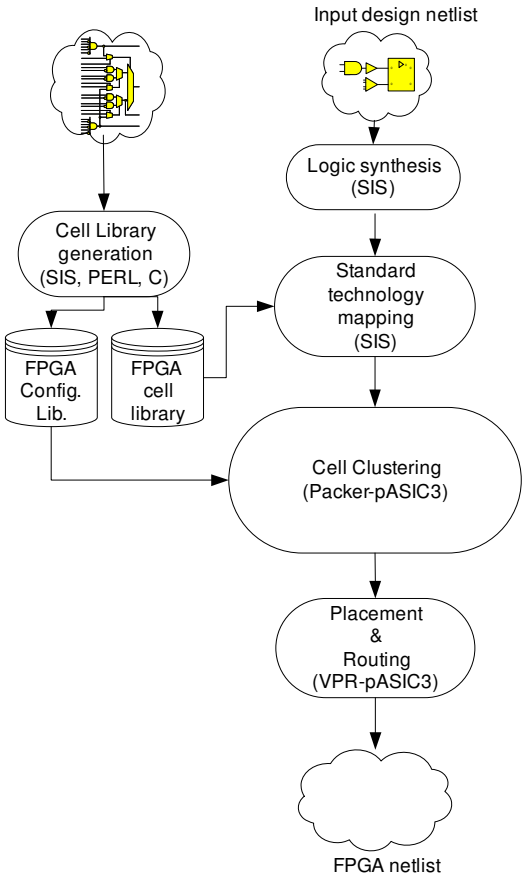

Fig. 2. Proposed synthesis flow for pASIC3 family FPGA.

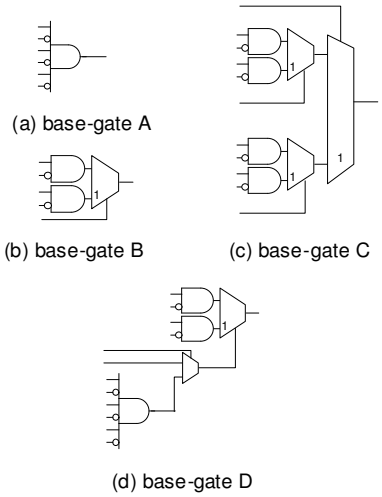

Fig. 3. Base gates extracted from pASIC3 logic cell.

A. Notation

To improve readability of this paper, we summarize the notation used throughout the paper in the following table.

S_{AD}	set of library cells that can be

	realized by personalization of either base-gate A or base-gate B
S	collection of sets of primitive cell types i.e., $\{S_{AD}, \ldots, S_{ABCD}\}$
M	number of distinct pASIC3 logic cell embeddings (configurations for filling a pASIC3 cell by primitive cell types)
q_i	number of embeddings of type i utilized in a mapped network
$C_{i,Sj}$	number of type S_j primitive cells in the i^{th} embedding
m_{Sj}	number of the primitive cells of type S_j utilized in a mapped network
n_A	number of base-gates A in a mapped network
N_{pASIC3}	number of pASIC3 logic cells needed to cover a network
2A+2B, 2A+C, A+B+D	cluster types according to the number of base gates in a cluster
n_{2A+2B}	number of type 2A+2B clusters
$c(u,b)$	local connectivity factor of node u for a base gate realization of type b
$P_d(u)$	number of free nodes within topological distance d from node u
$N_d(u,b)$	number of absorbable nodes within distance d from node u for a base gate realization of type b
$G(L,u,b)$	gain value for merging node u that is realized with base gate b into cluster L
$r(x)$	number of pins on net x
$\alpha_L(x)$	number of pins of net x that are already inside cluster L
Nets(L)	Nets connected to nodes within cluster L
crit(u)	timing criticality of node u
slack(u)	timing slack of node u
$E(y_j,L)$	set of neighboring nodes of node y_j in cluster L
$maxNC(y_j,L)$	the maximum criticalities of any node in $N(y_j,L)$
$minNC(y_j,L)$	the minimum criticalities of any node in $N(y_j,L)$

III. TOOL FLOW AND CELL LIBRARY CONSTRUCTION

Fig. 2 shows our CAD tool flow for pASIC3 family FPGAs. We generate a cell library and configuration information from the pASIC3 logic cell. A target circuit is synthesized by SIS [20] and then the circuit is mapped by cells in the cell library. Our clustering tool called Packer-pASIC3 packs nodes mapped by library cells into clusters. A cluster is assigned to a pASIC3 logic cell. VPR [25] places and routes the clustered network with the architecture description of pASIC3 family FPGAs.

Mapping multiple output logic with large fanin inputs is very expensive in terms of the memory requirements and computational complexity, and the number of gates that may be generated from the pASIC3 logic cell by assigning 0 or 1 to its inputs (i.e., connecting inputs to either VDD or GND levels) is quite large. Therefore, we break the pASIC3 logic cell into manageable sub-blocks at the expense of not exploiting the full flexibility/programmability of the larger block. By appropriately connecting the control inputs of the four multiplexers (cf. **mux1** through **mux4** in Fig. 1) to zero or one logic levels, four *base gates* (A, B, C, and D) can be obtained as shown in Fig. 3.

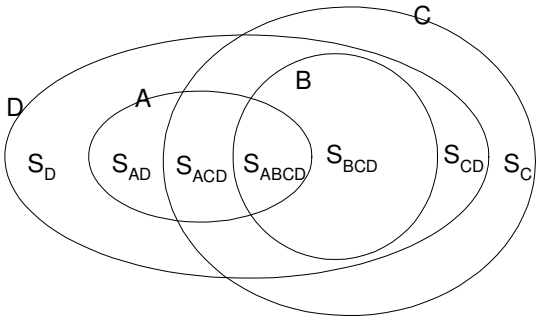

Fig. 4. Venn's diagram of for the set of logic cells that can be personalized from base gates.

After deriving the base gates, cell generation is performed for each base gate. Cell personalization is done either by *assigning constant 1 or 0* to some of the inputs or by *connecting some of the inputs together* (bridging.) By applying all possible combinations of these two operations to a base gate, a large number of library cells can be generated. We call these personalized cells "*primitive cells*". However, some of the primitive cells generated from different base gates will have identical Boolean function. In fact, we can draw a Venn's diagram to depict the set relationship among the primitive cells that are generated from different base gates, as depicted in Fig. 4. There are seven different primitive cell types, S_{AD}, …, S_{ABCD}. S_{AD} denotes the set of library cells that can be realized by personalization of either base-gate A or base-gate D. Other primitive cell types are similarly defined.

Table I Cell type distribution.

Type set	S_{AD}	S_{ACD}	S_{BCD}	S_C	S_D	S_{CD}	S_{ABCD}
Cells	3	5	20	714	110	28	6

Note that the total number of primitive cells is more than 5,000. Using all of these primitive cells results in high cpu time and memory usage during the FPGA mapping process, and should thus be avoided. To limit the number of primitive cells that tend to be useful in practice, we performed the following experiment. We selected thirteen circuits from the MCNC91 benchmark suite and mapped each circuit to the full set of

primitive cells. Next we counted the number of times that each primitive cell was utilized as a match for an intermediate node of the technology decomposed Boolean network during the mapping process. We noticed that 886 primitive cells were matched at some node at least once i.e., more than 4000 primitive cells were never used during the mapping process for these benchmark circuits. These primitive cells tended to be complex (multiple inputs and large number of literals in their factored forms.) So for the sake of improving the computational efficiency and reducing the memory usage of the mapper, we opted to keep only this set of 886 primitive cells. Note that the selected primitive cells include almost all of the standard cells in a typical ASIC library. Table I shows the cell distribution of those selected primitive cells.

IV. LOWER-BOUND CALCULATION

In this section, we provide an algorithm to find the minimum number of pASIC3 logic cells to cover a mapped network.

A. Problem Statement and Dynamic Programming

Once a mapped netlist is generated after technology mapping, we must solve the problem of clustering the primitive cells used in the mapped netlist into the pASIC3 logic cells. Since the mapping is performed before placement and routing, detailed physical information is not available. In addition, antifuse-based FPGAs have relatively rich routing resources since routing switches are abundant and many layers of metal wires can cross over the pASIC3 logic cells [1]. Thus, we have opted to minimize the total area taken by the pASIC3 logic cells during the initial clustering step.

Problem 1: Given a mapped netlist comprised of primitive cells, find the minimum number of *pASIC3* logic cells that can realize the network.

There are seven different primitive cell types, S_{AD}, …, $S_{ABCD,}$ as defined in Fig. 4. Looking at Fig. 1(a), it is easy to see that base gates may conflict with one another in the sense that they cannot be packed together in the same *pASIC3* logic cell. For example, base-gates B and C cannot be packed together in one pASIC3 cell. There is thus a fixed number of ways to embed (pack) a number of these primitive cells into one *pASIC3* logic cell. For example, two type-AD primitive cells and two type-BCD primitive cells can be packed in a single logic cell by using two base-gate A's and two base gate B's. Alternatively, two type-AD primitive cells and one type-C primitive cell can be packed in one *pASIC3* logic cell by using two base-gate A's and one base-gate C. Any such *embedding* (out of M possible embeddings) gives rise to the following equation:

$$LC_i = \sum_{j \in S} C_{i,S_j} \qquad i = 1,...,M \tag{1}$$

where S is the collection of sets of primitive cell types, {S_{AD}, …, S_{ABCD}}, and $C_{i,Sj}$ is the number of type S_j primitive cells in the i^{th} embedding.

The packing problem can be restated as follows. Given M configurations of filling a pASIC3 logic cell by primitive cells derived from the base gate types and a netlist of cells generated by the mapper, find the minimum number of logic cells that cover all cells in the netlist.

Let m_{Sj} denote the number of the primitive cells of type S_j in a mapped network. For example, m_{AD} is the number of type-AD primitive cells. The problem can be restated as:

$$\begin{aligned} &Minimize \quad \sum_{i=1}^{M} q_i \\ &s.t. \quad \forall S_j \in S: \sum_{i=1}^{M} q_i \cdot C_{i,S_j} \geq m_{S_j} \end{aligned} \tag{2}$$

where q_i is the number of embeddings of type i in the mapped network. This is the same problem as the well-known coin change problem as defined next.

Coin Change Problem: Let $c_1, c_2, \ldots c_q$ be the coin types of a currency. Let C_i denote the value of coin c_i in cents and K be some integer. We assume $C_1=1$. The problem is to produce K cents of change by using a minimum number of coins. The recursive expression for the solution can be

$$cnt[K] = \begin{cases} 0 & if \ K = 0 \\ \min\limits_{i: C_i \leq K} cnt[K - C_i] + 1 & if \ K > 0 \end{cases} \tag{3}$$

where *cnt[K]* is the minimum number of coins for K cents. We can compute the optimal solution to the coin change problem by using a bottom-up approach. By solving the optimal solution for values smaller than K, we can find the optimal solution for the exact amount of K by referring to the optimal solutions of the previously solved sub-problems. The running time complexity is *O(qK).*

To formulate the cell-packing problem, we must extend the coin change problem. First, instead of a single amount K, there will be seven different amounts, each of which is the number of primitive cells in the mapping solution that are in each of the seven base sets, S_{AD} thru S_{ABCD}. The recurrence equation for this problem is written as follows:

$$cnt(m_{AD},...,m_{ABCD}) = \begin{cases} 0 & if \ \forall S_j \in S: m_{S_j} \leq 0 \\ \min\limits_{\forall i} \left(cnt\left(m_{AD} - C_{i,S_{AD}}, \ ..., \ m_{ABCD} - C_{i,S_{ABCD}}\right) + 1 \right) & otherwise \end{cases} \tag{4}$$

The complexity of the corresponding dynamic programming algorithm is:

$$O\left(M \cdot \prod_{j \in U} m_{S_j} \right).$$

B. Set containment relations

Base gates can be put into two classes: simple and complex base gates. A complex base gate is one that consists of multiple base gates and internal multiplexers, while a simple base gate cannot be composed by other base gates. Base-gates C and D

are complex, whereas base-gates A and B are simple. The inclusion relationship between these base-gates is expressed as follows:

$$\begin{aligned} &\text{basegate } B \subset \text{basegate } C \\ &\text{basegate } B \subset \text{basegate } D \\ &\text{basegate } A \subset \text{basegate } D \end{aligned} \tag{5}$$

Notice that when both a simple base gate and a complex base gate can implement a primitive cell, the simple base gate will be selected for realizing the function of the primitive cell. Realizing the function by the complex base gate not only wastes area of the pASIC3 logic cell but also needlessly increases the circuit delay. Therefore, we can safely state that base-gates C and D are inferior to base-gates A and B when they implement the same logic function.

C. Minimum number of pASIC3 logic cells

Given the number of base gate types needed for mapping a circuit, the key question is how many pASIC3 logic cells are required to contain all of the base gates. There are three types of pASIC3 embeddings (clusters) i.e., 2A+2B, 2A+C, and A+B+D. A type 2A+2B pASIC3 logic cell is defined as the pASIC3 logic cell that has two base-gate A's and two base-gate B's in it. Other types can be defined similarly.

Theorem 1: Let n_A denote the number of base-gates A in a mapped netlist. n_B, n_C, and n_D are similarly defined. The minimum number of pASIC3 logic cells N_{pASIC3} needed to implement a mapped netlist containing n_A, n_B, n_C, and n_D base-gates can analytically be calculated as follows:

$$N_{pASIC3} = MAX(N_{2A}, N_{2B}) + n_C + n_D$$

$$N_{2A} = \begin{cases} \dfrac{n_A - 2n_C - n_D}{2} & n_A + 2n_C \geq n_D \\ 0 & otherwise \end{cases} \tag{6}$$

$$N_{2B} = \begin{cases} \dfrac{n_B - n_D}{2} & n_B \geq n_D \\ 0 & otherwise \end{cases}$$

Proof: Base-gates C and D cannot be packed together while base-gates B and C cannot be packed together. Therefore, the number of type 2A+C pASIC3 logic cells is equal to the number of base-gate C's. Similarly, the number of type A+B+D pASIC3 logic cells is equal to the number of base-gate D's. The number of type 2A+2B pASIC3 logic cells is determined by dividing the maximum number between the number of base-gate A's and base-gate B's. However, a type 2A+C pASIC3 logic cell can pack two base-gate A's as well as one base-gate C. Thus, the actual number of base-gate A's for type 2A+2B pASIC3 logic cells must be calculated by subtracting the number of base-gate A's, which have been packed by type 2A+C and A+B+D pASIC3 logic cells, from the original number of base-gate A's. Similarly, the actual number of base-gate B's for type 2A+2B pASIC3 logic cells can be calculated. Finally, the total number of pASIC3 logic cells becomes the sum of all three type pASIC3 logic cells. ■

D. Type distribution table

Theorem 1 can be used to significantly simplify the problem. After technology mapping, we count the number of primitive cells of specific types. The problem can be restated follows.

Problem 2: Given a primitive cell library generated from the pASIC3 logic cell structure and a mapped network comprising the primitive cells, we want to find the best choices of base gates A, B, C and D for realizing all of the primitive cells in the network so as to minimize the number of required pASIC3 logic cells.

Note that after the base gate counts are known, the minimum number of logic cells can be computed straightforwardly based on Theorem 1.

Table II
The type distribution table for primitive cell to base-gate mapping

# of primitive cell types	# of Base-gate types			
	A	B	C	D
m_{AD}	m_{AD}	0	0	0
m_{ACD}	x	0	$m_{ACD} - x$	0
m_{BCD}	0	m_{BCD}	0	0
m_D	0	0	0	m_D
m_C	0	0	m_C	0
m_{CD}	0	0	$m_{CD} - y$	y
m_{ABCD}	z	$m_{ABCD} - z$	0	0

Table II shows how a primitive cell of type Γ in the mapped network is realized with a base gate of type A, B, C, or D. Notice that many of the primitive cell types have a unique realization in a single base-gate type. Examples include types BCD of primitive cells. Note that a type BCD primitive cell should be realized only using type B base gates because of the inclusion relationship of (5) and the fact that complex base-gates are always more costly than the corresponding simple base gates. Three of the primitive cell types, however, can be realized by using either of two base gates. For example type ACD primitive cell can be realized as either type A or type C base gate. This table shows that, to solve problem 2, all we have to do is to determine variables *x, y* and *z* where *x* denotes the number of primitive cells of type ACD that are realized as a type A base gate, *y* denotes the number of primitive cells of type CD that are realized as a type D base gate, and *z* denotes the number of primitive cells of type ABCD that are realized as a type A base gate.

Problem 3: Given the occurrence count of different primitive cell types in a mapped network, find the values of variables x, y and z so as to minimize the number of pASIC3 logic cells required to cover the network.

E. Linear programming formulation and solution

We formulate Problem 3 as a linear programming problem and then obtain the optimal solution by finding either the minimum point of an intersected plane of two equations [22] or the minimum point of an equation that is always above the other within certain ranges of variables. Equation (6) can be restated as in (7).

$$N_{pASIC3} = MIN\{\max(N_{ACD}(x,y,z), N_{BCD}(x,y,z))\}$$
$$0 \le x \le m_{ACD}; 0 \le y \le m_{CD}; 0 \le z \le m_{ABCD}$$

$$N_{ACD} = \begin{cases} \frac{1}{2}(m_{AD} + x + z + m_D + y), \\ \quad if\ m_{AD} - 2(m_{ACD} + m_C + m_{CD}) - m_D + 3x + y + z \ge 0 \\ m_D + m_{ACD} - x + m_C + m_{CD}, \quad otherwise \end{cases} \tag{7}$$

$$N_{BCD} = \begin{cases} \frac{1}{2}(m_{BCD} + m_{ABCD} - z + m_D - y) + (m_{ACD} - x + m_C + m_{CD}), \\ \quad if\ m_{BCD} + m_{ABCD} - m_D - y - z \ge 0 \\ m_D + m_{ACD} - x + m_C + m_{CD}, \quad otherwise \end{cases}$$

The brute-force algorithm is to search for the optimal solution by trying out every possible combinations of x, y, and z within their allowed ranges ($0 \le x \le m_{ACD}$, $0 \le y \le m_{CD}$, $0 \le z \le m_{ABCD}$). The computational complexity, however, is $O(m_{ACD} \times m_{CD} \times m_{ABCD})$, which can be quite high. Fortunately, equation (7) has an important property that allows us to speed up the search: As *x, y*, and *z* increase, N_{ACD} increases but N_{BCD} decreases. Therefore, within allowed ranges of *x, y,* and *z*, equations for N_{ACD} and N_{BCD} may intersect in a plane or one equation is above the other all the time. We explain the solution for the two cases as follows.

Case 1: When N_{ACD} and N_{BCD} intersect in a plane, at the intersected plane, N_{ACD} and N_{BCD} become equal:

$$F(x,y,z) = N_{ACD}(x,y,z) - N_{BCD}(x,y,z) = ax + by + cz + d = 0 \tag{8}$$

where a, b, c, and d are coefficients of an equation of a plane after the subtraction. All points in this plane guarantee that logic cells are full because N_{ACD} and N_{BCD} are equal but choosing one arbitrary point on the plane may not give the optimal solution. Therefore, we need to find the point that gives the optimal solution in this plane. Notice that we should consider only points on the plane within the specified ranges for x, y, and z. Further more, we need to check only corners of the plane because of the property of N_{ACD} and N_{BCD} mentioned above.

Case 2: N_{ACD} and N_{BCD} may not intersect at all, resulting in one equation lying above the other in the ranges of *x, y*, and *z*. In this case, simply, two points are evaluated: ($x=0, y=0, z=0$) and ($x = m_{ACD}, y = m_{CD}, z = m_{ABCD}$). If N_{ACD} is larger than N_{BCD} at $x=0, y=0,$ and $z=0$, $N_{ACD}(x=0, y=0, z=0)$ is the minimum solution. Otherwise, N_{BCD} ($x = m_{ACD}, y = m_{CD}, z = m_{ABCD}$) is the minimum solution.

The worst case of the above algorithm is when it requires checking all of the candidate points. Those candidate points can be enumerated by setting minimum or maximum values to variables except one variable. Therefore, the complexity is $O(k \cdot 2^{k-1})$ where *k* is the number of variables. In this case, *k=3*. Notice that the computational complexity of this algorithm is independent of the network size.

From the optimal distribution of primitive cells, we can easily find out what kind of logic cells and how many of those logic cells are required. Notice that there are only three types of clusters (embeddings): 2A + 2B, 2A+C, and A+B+D. n_A indicates the number of base-gate A required from the optimal distribution of primitive cells. Likewise, n_B, n_C, and n_D can be computed by adding numbers for each column in Table II as followed:

$$\begin{aligned} n_A &= m_{AD} + x + z \\ n_B &= m_{BCD} + m_{ABCD} - z \\ n_C &= m_{ACD} - x + m_C + m_{CD} - y \\ n_D &= m_D + y \end{aligned} \tag{9}$$

The numbers of logic cells for different cluster types (n_{A+B+D}, n_{2A+C}, n_{2A+2B}), can be computed by the following equations:

$$\begin{aligned} n_{A+B+D} &= m_D + y \\ n_A &= (m_{AD} + x + z) - n_{A+B+D} \\ n_B &= (m_{BCD} + m_{ABCD} - z) - n_{A+B+D} \\ n_{2A+C} &= m_{ACD} - x + m_C + m_{CD} - y \\ n_A &= n_A - 2n_{2A+C} \\ n_{2A+2B} &= \max\left(\left\lceil \frac{n_A}{2} \right\rceil, \left\lceil \frac{n_B}{2} \right\rceil\right) \end{aligned} \tag{10}$$

Knowing the numbers of logic cells for each cluster type can be used to guide algorithms to achieve small area. We use this information as area constraints to build a clustering solution.

V. THE CLUSTERING TECHNIQUE

In this section, we present a cell clustering technique that considers both interconnect connectivity and circuit delay. The algorithm improves routability and delay under the constraints of the minimum number of pASIC3 logic cells for a given circuit.

Knowing the minimum number of pASIC3 logic cells is not sufficient information to enable us to assign mapped nodes into the pASIC3 logic cells. In other words, we can calculate the minimum number of clusters of different types required for a circuit by using the algorithm described in the previous section. However, that algorithm does not produce a complete clustering solution because it does not take into account the connectivity between nodes in the circuit. There are two facts worth mentioning again. First, nodes that are mapped to a

certain primitive cell type can only be realized by a limited number of different base gates; second, there are upper-bounds on the number of different types of clusters, i.e., n_{A+B+D}, n_{2A+C}, and n_{2A+2B}. We refer to these two conditions as *resource constraints* for a circuit. Therefore, when we create a new cluster, we have to ensure that the resource constraints are not violated.

A. Interconnect-aware Clustering

Since the routing area is one of the primary goals, we propose a heuristic algorithm of interconnect-aware clustering algorithm. The problem can be stated as follows:

Problem 4: Given a network mapped to primitive cells and the number of different cluster types specified for the packing solution with the minimum number of pASIC3 logic cells, find a clustering solution that has the minimum number of inter-cluster interconnects.

A wire connecting two un-clustered (*free*) nodes that can be packed together is called an *absorbable wire*. When an absorbable wire connects node u with some free node x, then we say that node x is an *absorbable node* with respect to node u. Considering conflicts between base gates in a pASIC3 logic cell, and motivated by [25], we define a *local connectivity factor* of node u for a base gate realization of type b as follows:

$$c(u,b) = \frac{N_d(u,b)}{P_d(u)} \qquad (11)$$

where $N_d(u,b)$ is the number of absorbable nodes within distance d of node u for a base gate realization of type b whereas $P_d(u)$ is the total number of free nodes within topological distance d from node u. Higher local connectivity factor, $c(u,b)\leq 1$, for a node u of base gate type b signifies that more absorbable nodes are located in the node's neighborhood and/or that the number of nodes in its neighborhood is small.

Fig. 5(a) and (b) show examples of calculating local connectivity factors for n_3 assuming no cluster has been yet formed and for $d = 1$. For example, in Fig. 5(a), $P_1(n_3)=4$ while $N_1(n_3,B)=4$ since all nodes connected to n_3 are compatible with base gate type B. Consequently, $c=1$.

In Fig. 5(c), node n_5 is non-absorbable with respect to n_3 because its base gate conflicts with base gate type B assigned to n_3. Therefore, $P_1(n_3)=4$ while $N_1(n_3,B)=3$ and $c=0.75$. Fig. 5(c) depicts a case in which node n3 has two possible base gate realizations e.g., A or B. As a result, two connectivity factors must be calculated for each base gate as were done for Fig. 5(a) and (b). From this example, it becomes clear that each node may have different connectivity factors corresponding to different base gate realizations.

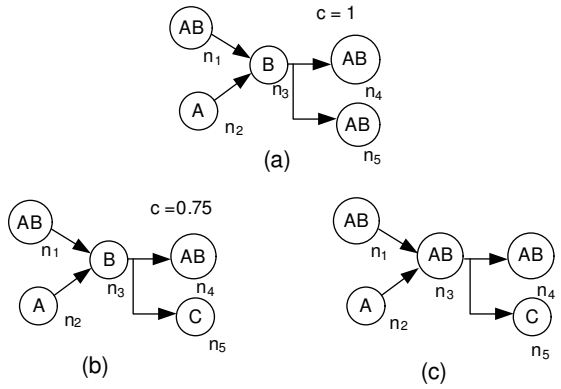

Fig. 5. Examples of local connectivity factor computation.

We propose a heuristic, interconnect-aware algorithm. Clustering is done in two steps. In the first step, the local neighborhood connectivity factor, c, for each base gate realization of each free node in the network is computed and a free node that has the highest local connectivity factor is chosen as a seed for a new cluster. Next, the type of the new cluster (and if the seed node admits different base gate realizations, the base gate type of the seed node) is determined based on the availability of compatible cluster types for the seed node i.e., a cluster type that is compatible with the base gate type of the seed node and has the highest availability is chosen. For example, with base gate type B for n_3, a cluster type of 2A+2B may be chosen in Fig. 5(a) and (b) whereas a cluster type of 2A+C and a base gate type of A for n_3 may be chosen in Fig. 5(c). Once the type of the cluster is determined, the number of available clusters for the type is reduced by one and an absorbable node which is compatible with the selected cluster type and has the highest *affinity* toward the cluster is packed into the cluster (see below.) The process is continued until no further clustering can be performed or no free node remains in the network.

Notice that the above heuristic for seed selection may be improved by considering the composition of the base gate types of the neighbors of the seed node.

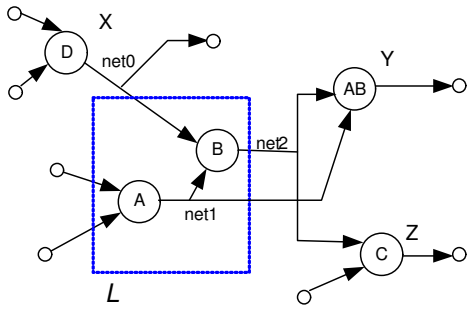

Fig. 6. Example of packing a new node into a partially formed cluster.

The degree of a node's affinity toward a cluster may be quantified by a *gain function*. Motivated by [18], given a

partially formed cluster L and a merging candidate node u with base gate realization b, the gain of packing u into L is calculated as follows:

$$G(L,u,b) \equiv \begin{cases} 0 & \text{if } u \text{ cannot be merged into } L \\ \sum_{x \in Nets(L) \cap Nets(u)} \frac{1+\alpha_L(x)}{r(x)} & \text{otherwise} \end{cases} \quad (12)$$

where $Nets(L)$ and $Nets(u)$ refer to the set of nets connected to nodes inside L and to u, respectively. $\alpha_L(x)$ denotes the number of pins of net x that are already inside cluster L and $r(x)$ is the number of pins of net x.

Fig. 6 depicts an example of gain calculation. Notice that net2 cannot be absorbed into cluster L because the cluster type of L is either 2A+2B or A+B+D, and node Z of base gate type C is not compatible with either cluster type. Therefore, either node X or node Y will be merged into L. If we consider only the gain, node Y must be packed into cluster L because it has higher gain than node X. However, we also have to ensure that the resource constraint is satisfied. For example, when we choose node Y with base gate A or B, the cluster type of the new cluster becomes 2A+2B. If there is no available pASIC3 logic cell of type 2A+2B, node Y cannot be packed into cluster L. Instead, assuming that there is an available cluster of type A+B+D, node X will be placed into cluster L.

At the end of the process described above, we may be left with a situation in which all available pASIC3 logic cells have been partially utilized, yet there are still un-clustered nodes. This is possible, for example, when an unfilled cluster cannot find a new node to bring in because all of its neighboring nodes have already been assigned to some other cluster, or all of its free neighbor nodes have resource conflicts with nodes that are already in that cluster. From our experiments, on average 20% of the nodes in a circuit are not clustered at the end of the clustering procedure described above.

To address this issue, we pack the remaining free nodes into unfilled clusters by using a linear assignment procedure as follows. We place the complete netlist composed of the (filled and unfilled) clusters and the free nodes by using a VPR high-temperature simulated annealing placer [25]. We can thus calculate the Euclidean distances between the unfilled clusters and the free nodes based on the placement result. Finally, we set up a linear assignment (bipartite graph matching) problem where on one side are the free nodes and on the other side are the unfilled clusters. An edge exists between a free node and an unfilled cluster if the node is absorbable into that cluster (e.g., it has compatible base gate type with respect to the unfilled portion of the cluster type.) The edge weight is set to the Euclidean distance between its two end points. This problem is solved optimally and in polynomial time using linear assignment solvers.

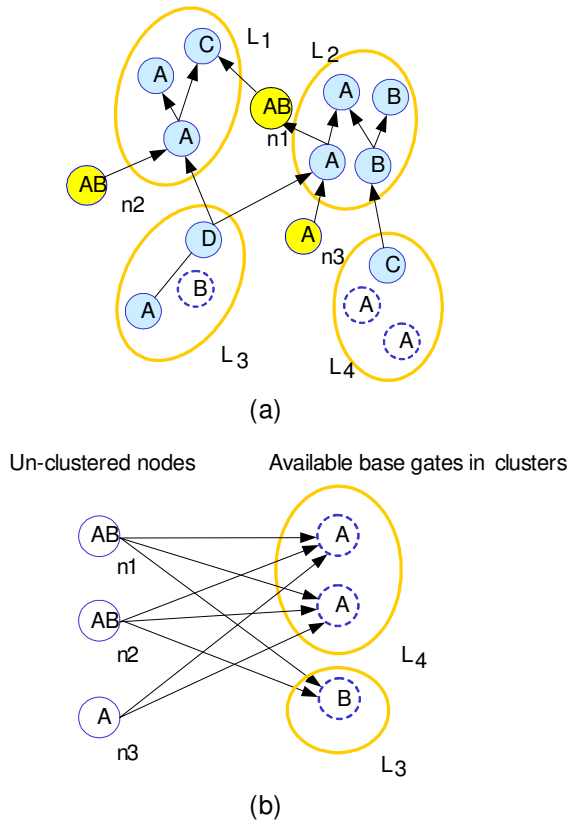

Fig. 7. Packing un-clustered nodes by using linear assignment: (a) partially clustered network; (b) bipartite graph for linear assignment.

Fig. 7 shows an example of transforming the partially clustered network into a bipartite graph for linear assignment. Notice that node n_3 cannot be clustered into L_3 because n_3 needs a type-A base gate mapping, which is not available in L_3. On the other hand, since a type-AB primitive cell can be implemented either by a base-gate A or by a base-gate B, nodes n_1 and n_2 can be clustered into either L_3 or L_4. We point out that because we guarantee that only available pASIC3 logic cells are used for the minimum area clustering, the number of empty spaces for base gates in clusters is equal to or larger than the number of free nodes.

B. Timing-driven Clustering

Delay caused by inter-cluster interconnect, which connects pASIC3 logic cells through interconnect wires and antifuses, tends to be much larger than the delay caused by intra-cluster interconnect. Therefore, we can assume that inter-cluster delay has a unit delay while the intra-cluster delay is negligible. This assumption is reasonable because no placement and routing information is known and the inter-cluster interconnect delay is much longer than the intra-cluster interconnect delay. The timing-driven clustering problem can be stated as follows.

Problem 5: Given a mapped network comprised of primitive cells and the numbers of different cluster types for the packing

solution that uses the minimum number of pASIC3 logic cells, find a clustering solution that has the minimum number of pASIC3 logic cells on the timing-critical paths of the circuit.

We use the notion of criticality of a node in a network as described in [16]. The timing criticality of a node u is redefined to have the range from 0 to 1 as follows:

$$crit(u) = 1 - \frac{slack(u) - MinSlack}{MaxSlack - MinSlack} \qquad (13)$$

where $slack(u)$ is the slack time of node u, MinSlack and MaxSlack are the minimum slack and the maximum slack in the network, respectively. When the criticality of a node is higher than that of the other nodes, then the node will be on a more critical timing path compared to the other nodes.

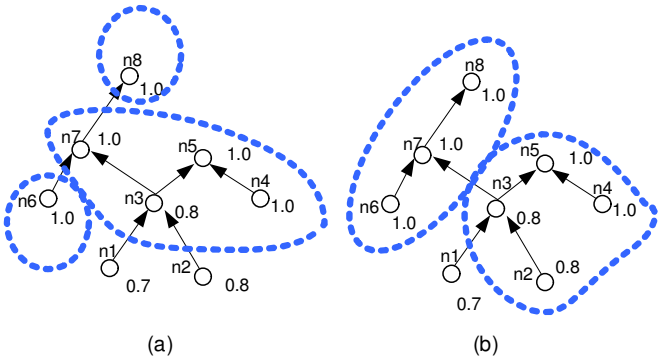

Fig. 8. Selecting the best node for clustering: (a) greedy selection and (b) critical-path aware selection.

In [16], a node with the highest criticality is absorbed into a cluster in a manner that possibly reduces the number of clusters on the critical path. However, we observed that packing nodes in such a greedy manner could increase the number of clusters on the critical path. Fig. 8 depicts a situation where greedily selecting a node with the largest criticality can cause a worse clustering solution. Suppose that node n_5 is a seed node. After packing node n_4 and n_3, in Fig. 8(a), the greedy algorithm will choose node n_7 because its criticality is greater than the criticalities of node n_1 and n_2. Note that this will prevent another cluster from absorbing node n_6, n_7, and node n_8, which are on the critical paths. On the other hand, by selecting node n_2 in Fig. 8(b), node n_6, n_7, and n_8 on the critical paths can be packed into a cluster. From this example, we notice that clustering a node with higher criticality than the connected nodes in a cluster can lower the chance of reducing the number of pASIC3 logic cells on the critical path.

Consider a partially-formed cluster L comprising of nodes $x_1,\ldots,x_m$. Let the absorbable neighboring nodes of L be denoted by $y_1,\ldots,y_p$. Let $E(y_j, L)$ denote the set of immediate neighbors of y_j in L. Furthermore, let $maxNC(y_j,L)$ and $minNC(y_j,L)$ denote the maximum and the minimum criticalities of any node in $E(y_j,L)$, respectively. Our approach selects the best node for packing with an order of the following priority: 1) a neighbor node, y_{j*}, such that $crit(y_{j*})$ is maximum among all y_j's and $crit(y_{j*}) = maxNC(y_{j*},L)$; 2) a neighbor node, y_{j*}, such that $crit(y_{j*})$ is maximum among all y_j's and $crit(y_{j*}) < maxNC(y_{j*},L)$; and 3) a neighbor node, y_{j*}, such that $crit(y_{j*})$ is minimum among all y_j's and $crit(y_{j*}) > minNC(y_{j*},L)$.

We use the same flow for the timing-driven clustering under the minimum area constraint as the interconnect-aware clustering. For seed selection, we give higher chance of being a seed for those nodes on critical paths. Since nodes on critical paths have the same criticality, we select the node with the lowest connectivity value among those nodes. In the next step, to select the best node for clustering, we use the algorithm based on the priority.

VI. Experiment Results

We have selected 18 large combinational circuits from the MCNC91 benchmark. SIS [20] reads the circuits in blif format. To evaluate our library generation and area-driven clustering, we compare our results to those from a commercial tool, called QuickWorks 4.1 from QuickLogic [19]. For QuickWorks 4.1, the following options were selected to minimize area:

- Logic optimization: level – technology map, mode-overnight, type-area, and no buffer insertion
- Placement and Route: overnight

QuickWorks uses the term *cell fragment* to indicate a library cell generated from a pASIC3 logic cell. The results were taken after placement. For our simulation set-up, the library was read and script.rugged was used to optimize a circuit. SIS was used for technology mapping with the library. We estimated the minimum number of logic cells by using our algorithms, Packer-area. Table III reports the results of the area-driven clustering. In most of the cases, Packer-area used fewer primitive cells than QuickWorks. Packer-area reduced the number of pASIC3 logic cells by 12.29% on an average compared to QuickWorks.

For timing-driven clustering experiments, because clustering quality can be reflected in placement results, it would be ideal if we could use QuickWorks to read, place, and route the clustered circuits and then measure delays of clustered circuits. Unfortunately, Quickworks does not read clustered circuits from external inputs. Therefore, max-depth which is a first-order measure of circuit delay in FPGA, is used to quantify the circuit delay in the pre-layout phase.

Table IV shows the results of clustering with different objectives such as the minimum number of external wires and the minimum pASIC3 logic cells on the critical path. Compared to the Packer-pASIC3-area-interconnect, the Packer-pASIC3-area-timing generated more inter-cluster interconnects by 7%. In order to measure the total wirelength after placement and routing, we used the VPR [25]. Packer-area-interconnect provided a reduction on the total wirelength by 4%, compared to the Packer-area-timing. In terms of the number of clusters on the critical path, the Packer-area-timing provided a shorter critical path than

Table III
Results of lower-bound calculation

	QuickWorks	Packer-pASIC3	
Circuits	PASIC3 logic cells	PASIC3 logic cells	pASIC3 logic cells
i9	95	96	-1.05
rot	104	88	15.38
i8	184	142	22.83
pair	243	213	12.35
vda	131	79	39.69
x1	45	42	6.67
C6288	476	448	5.88
C5315	264	196	25.76
alu4	125	113	9.60
apex6	124	84	32.26
C880	57	54	5.26
C3540	181	175	3.31
alu2	66	57	13.64
C1355	57	53	7.02
C1908	56	55	1.79
C432	31	31	0.00
C499	58	53	8.62
Average Improvement (%)			12.29

QuickWorks, and the Packer-area-interconnect by 35%, and 14%, respectively.

VII. CONCLUSION

In this paper, we presented clustering algorithms for coarse-grained, antifuse-based FPGAs. We generated a library set from the pASIC3 logic cell and mapped a network with the library set. For the mapped network, we presented a dynamic programming solution as a general solution to find the minimum number of pASIC3 logic cells. By considering the architectural characteristic of the pASIC3 logic cell, we set up a pair of linear equations and found the optimal solution. With this minimum area requirement, we proposed an interconnect-aware clustering algorithm and a timing-driven clustering algorithm. The interconnect-aware clustering algorithm used connectivity information among nodes under the constraint for the minimum area. The timing-driven clustering algorithm intelligently packs nodes into clusters to minimize the number of clusters on the critical path, by avoiding false selection of critical nodes. For the minimum number of pASIC3 logic cells, our low-bound calculating algorithm provided approximately a 12% reduction, when compared to QuickWorks from QuickLogic. The interconnect-aware clustering also required a 21% reduction on the number of inter-cluster interconnects, when compared to a simple clustering algorithm based on placement and proximity among nodes. The timing-driven clustering algorithm reduced the number of pASIC3 logic cells on the critical path by 35%, compared to QuickWorks.

REFERENCES

[1] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," *IEEE Design and Test of Computers*, Vol. 13, No. 2, pp. 42-57, 1996.

[2] V. Betz, J. Rose, A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, 1999.

[3] Elias Ahmed, "The effect of logic block granularity on deep-submicron FPGA performance and density," M.A.Sc thesis, University of Toronto, Canada, 2001.

[4] C.W. Kang, A. Iranli, and M. Pedram, "Technology mapping and packing for coarse-grained, anti-Fuse based FPGAs," in Proc. *Asia-South Pacific Design Automation Conference*, pp. 209-211, 2004.

[5] C.W. Kang and M. Pedram, "Clustering techniques for coarse-grained, antifuse FPGAs," in Proc. *Asia-South Pacific Design Automation Conference*, pp. 785-790, 2005.

[6] *pASIC3* FPGA Family Datasheet, QuickLogic Corporation (http://www.quicklogic.com).

[7] Virtex FPGA Datasheet, Xilinx Corporation (http://www.xilinx.com).

[8] APEX FPGA Datasheet, Altera Corporation (http://www.altera.com).

[9] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactiosn on Computer-Aided Design,* Feb. 1994,vol. 13, no. 1, pp. 1- 12.

[10] J. Cong and Y. Ding, "Combinational logic synthesis for SRAM-based field-programmable gate arrays," *ACM Transactions on Des. Automat. Electron. System*, April, pp. 145 – 204, 1996.

[11] S. Ercolani and G. De Micheli, "Technology mapping for electrically programmable gate arrays," in *Proc. 28th ACM/IEEE Design Automation Conference*, 1991, pp. 234- 239.

[12] Yung-Te Lai, Sarma Sastry, and Massoud Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," in *Proc. IEEE International Conference on Computer Design*, 1992, pp. 42 – 458

[13] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. on Design Automation of Electronic Systems*, vol. 2, no. 3, pp. 193 – 226, July 1997.

[14] J. Cong, J. Peck, and Y. Ding, "RASP: a general logic synthesis system for SRAM-based FPGAs," in *Proc. FPGA*, pp. 137 – 143, 1996.

[15] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size," in Proc Custom Integrated Circuits Conference, 1997, pp. 551 – 554.

[16] Alexander Marquardt, Vaughn Betz, and Jonathan Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in Proc. FPGA, pp. 37- 46, 1999.

[17] E. Bozogzadeh, S. Ogrenci-Memik, M. Sarrafzadeh, "Rpack: routability-driven packing for cluster-based FPGA," *in Proc. Asia-South Pacific Design Automation Conference*, pp. 629 – 634, 2001.

[18] A. Singh and M. Marek-Sadowska, "Efficient circuit clustering for area and power reduction in FPGAs," in Proc. FPGA, pp. 59-66, 2002.

[19] QuickLogic.com, QuickWorks User Manual

[20] Sentovich, E.M., et al., SIS: A system for sequential circuit synthesis, 1992, Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

[21] M. Wang, X. Yang, and M. Sarrafzadeh, "Dragon2000: standard-cell placement tool for large industry circuits," in *Proc. International Conferecence on Computer Aided Design*, 2000, pp. 260-263.

[22] http://mathworld.wolfram.com

[23] W. E. Donath, "Placement and average interconnect requirements of computer logic," IEEE Trans. Circuits and Systems, CAS-26:272-277, 1974.

[24] M. Tom and G. Lemieux, "Logic block clustering of large designs for channel-width constrained FPGA," in Proc. Design Automation Conference, pp. 726-731, 2005.

[25] Vaughn Betz and Jonathan Rose, "VPR: a new packing, placement, and routing tool for FPGA research," *Int'l Workshop on FPL*, 1997, pp. 213- 222.

[26] M. Pedram and B. Preas, " Interconnection length analysis for standard cell layouts," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 10, Oct. 1999, pp. 1512 – 151.

Table IV
Results of different clustering objectives with the minimum area solution

		QuickWorks4.1	Packer-pASIC3-area-interconnect			Packer-pASIC3-area-timing		
Circuits	Total Wires	Max Depth	Max. Depth	Inter-cluster Wires	Wire Length	Max. Depth	Inter-cluster Wires	Wire Length
i9	462	9	8	285	4750	6	287	4649
rot	485	15	11	338	4881	9	379	5794
i8	701	9	8	475	9159	8	478	9599
pair	975	15	18	675	12080	14	761	11272
vda	329	10	11	247	3026	8	257	3123
x1	216	6	5	140	1890	5	152	1987
C6288	1545	91	69	1167	12049	67	1323	13659
C5315	894	16	18	661	14175	15	718	14471
alu4	433	25	23	284	3540	19	320	3843
apex6	464	9	8	338	6285	7	357	5899
C880	237	20	17	183	2060	17	197	1922
C3540	722	23	26	440	6743	23	547	6887
alu2	226	32	17	134	1650	16	164	1779
C1355	251	17	11	185	1350	12	186	1382
C1908	248	25	16	174	1506	14	186	1578
C432	156	25	22	101	802	16	113	804
C499	251	13	11	186	1352	11	182	1529
Average Change		1.35	1.14	1	1	1	1.07	1.04