2010 International Conference on Field Programmable Logic and Applications

Efficient FPGA Resynthesis Using Precomputed LUT Structures

Andrew Kennings
Dept. Elec. and Comp. Eng.
University of Waterloo
Email: akenning@ece.uwaterloo.ca

Alan Mishchenko
Dept. of EECS
University of California, Berkeley
Email: alanmi@eecs.berkeley.edu

Kristofer Vorwerk, Val Pevzner, Arun Kundu
Actel Corporation
Email: {vorwerkk, pevznerv,kundua}@actel.com

***Abstract*—The ability to efficiently match logic functions to structures of K-input look-up tables (K-LUTs) is a central problem in FPGA resynthesis algorithms. This paper addresses the problem of matching logic functions of $\sim$ 9 to 12 inputs to K-LUT structures. Our method is based on the off-line generation of libraries of LUT structures. During resynthesis, matching is accomplished efficiently using NPN encoding and hash table look-ups. Generating an effective library of LUT structures may seem prohibitive due to the overwhelming number of logic functions which must be considered and represented in the library. We show that, by careful consideration of which logic functions and LUT structures to keep, it is possible to generate useful, compact libraries. We present numerical results demonstrating the effectiveness of our ideas when used during area-oriented resynthesis after FPGA technology mapping.**

I. Introduction

The most common architecture for an FPGA is the LUT-based architecture in which the basic programmable logic element for combinational logic is the K-input look-up table (K-LUT). A K-LUT is capable of implementing any logic function up to K-inputs. Modern FPGAs typically have $3 \leq K \leq 6$.

We consider the problem of matching completely-specified logic functions to *structures* of LUTs—that is, networks consisting of a small number of K-LUTs. Such networks are useful for implementing logic functions with $\sim$ 9 to 12 inputs. Figure 1 illustrates structures which consists of three 4-LUTs suitable for implementing logic functions of up to 10 inputs (not all logic functions with ≤ 10 inputs can be implemented). As stated in [8], the matching problem for LUT structures comes in two forms. In the first form, a specified LUT structure is provided and the task is to determine if a given logic function can be implemented. In the second form, a logic function is provided and the goal is to generate a LUT structure to implement the logic function. We consider aspects of both of these forms of the matching problem.

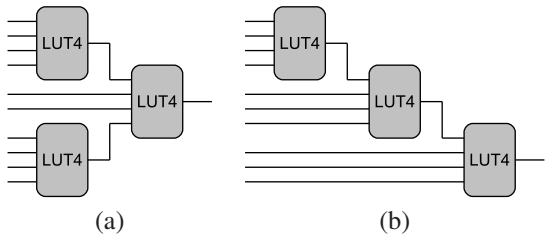

Figure 1. LUT structures with three 4-LUTs suitable for implementing many logic functions with ≤ 10 inputs.

The matching problem has several applications within the FPGA design flow. For example, matching can be employed in the resynthesis algorithms applied to both post-technology-mapped [8] and post-placement [10] networks for optimization of objectives such as delay, area, and power. In such algorithms, a subset of cells is selected from the network based on desirable criteria (e.g., the cells are on the critical path). For each cell in the selected subset, several different cones of logic rooted at the cell are computed. Subsequently, the cones of logic are resynthesized in an attempt to improve one or more of the aforementioned objectives; it is precisely in the resynthesis step that matching proves useful to obtain alternative logic structures. Performing resynthesis can require significant computational effort, including the effort required to perform matching. Hence, efficient matching is essential to the overall efficiency and effectiveness of a resynthesis algorithm.

The contributions of this paper are several-fold:

- For a set of industrial FPGA circuits, we show that the number of k-input logic functions which occur in practice is much smaller than the entire set of possible logic functions. We show that the number of NPN equivalence classes required to cover a high percentage of k-input logic functions is reasonable. These statistics are presented for $5 \leq k \leq 9$.
- We propose a matching algorithm (used during FPGA resynthesis) which relies on matching against a library of precomputed and stored LUT structures (as described in this paper).
- We present a means by which the off-line generation of an effective library of LUT structures can be performed. The approach is based on decomposition algorithms and the idea of dominant LUT structures. We quantify the memory necessary to store the resultant library.
- We provide numerical results from using the proposed matching technique inside an area-oriented resynthesis algorithm applied after technology mapping to demon-

1946-1488/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.105

strate the efficacy of matching based on precomputed libraries.

This paper is organized as follows. Section II provides background information on matching. In Section III, we present an analysis of the logic functions found in a large set of industrial designs. In Section IV, library generation is described. Section V analyzes the memory usage of the computed libraries. Section VI illustrates the use of our library-based matching algorithm during resynthesis. Numerical results are presented in Section VII. Conclusions are presented in Section VIII.

II. Background

Historical approaches to matching include: (1) structural-based, (2) SAT-based, (3) class-based, and (4) decomposition-based techniques. Each method exhibits advantages and disadvantages which can be expressed in terms of their efficiency and effectiveness at matching to LUT structures.

Structural-based techniques [4] are rooted in the idea of remapping cones of logic. These approaches use the same concepts as those employed in structural FPGA technology mappers [6], [9]. Although efficient, structural techniques are not effective at matching to particular LUT structures; they are subject to the logic function being matched and are structurally biased by the logic function's representation as a subject graph.

SAT-based techniques [3], [5] are effective at matching logic functions to particular LUT structures and are guaranteed to find a match if one exists. However, these techniques are not efficient for matching logic functions with a large number of inputs (≥ 10). Structures are typically limited in size to ≤ 3 LUTs. Larger functions and structures increase the size and difficulty of the SAT problems and lead to increased run-time.

Class-based techniques [10] are efficient due to available equivalence class encoding algorithms [1], [2]. However, class-based methods require libraries against which matching can be performed. Their effectiveness depends greatly on the library. For LUT structures, the use of libraries implies finding all logic functions implemented by a particular LUT structure. It is not feasible to generate complete libraries for larger LUT structures (and large K) due to the flexibility (programmability) of a LUT.

Decomposition-based techniques [8] attempt to apply logic decomposition to "split" a logic function into individual pieces such that each piece can be implemented by a K-LUT. These techniques vary greatly in terms of efficiency depending on the logic function being matched and the particular decomposition strategy employed. Decomposition can be effective but can miss certain matches due to various heuristics employed to speed-up the decomposition.

We view *class-based* and *decomposition-based* techniques as the more promising techniques for matching to LUT structures. As mentioned, the main disadvantage of class-based techniques is the need to generate a library. If the task of library generation could be overcome, class-based techniques would be well-suited for logic functions of $\sim$ 9 to 12 inputs due to the efficient encoding algorithms. Decomposition-based techniques, on the other hand, offer the flexibility of matching a logic function to a variety of LUT structures. The combination of these two techniques leads to the application of decomposition-based methods to generate a library which, in turn, can be employed by a class-based technique to perform matching.

III. Function Coverage

In practice, it is not necessary to fully enumerate all logic functions implementable by a particular LUT structure. It has been observed that, for a given value of n, only a small subset of n-input functions occur in practice on real networks [7]. Hence, rather than first defining LUT structures and computing their implementable functions, it is more reasonable to determine which functions need to be considered and then to determine how such functions can be implemented via structures.

To determine which logic functions to consider, we have performed an experiment using a set of 100 industrial designs which have been synthesized, subjected to technology-independent optimizations, and converted into subject graphs. All logic functions with ≤ 9 inputs have been computed and NPN-encoded (along with their frequency of occurance) over the entire design suite using enumerative cut generation on the subject graphs. The results are presented in Figure 2. The number of equivalence classes that occur in practice is small which implies that full enumeration of LUT structures is not required to achieve an effective library of LUT structures.

Many equivalence classes occur only a few (≤ 2) times and it is reasonable to *exclude* these less common equivalence classes to reduce the amount of information stored in the library. To judge the usefulness of removing less common equivalence classes (logic functions), an additional experiment was performed. For n-input logic functions, less frequent equivalence classes were discarded while still maintaining a *target coverage*; i.e., we removed equivalence classes until the number of logic functions encoded to the

#Inputs	#Functions	#NPN Classes	$\frac{\#\text{NPN Classes}}{\#\text{Functions}}$
5	7768377	3269	4.21e-4
6	18814641	34225	1.82e-3
7	50239975	271646	5.41e-3
8	146678254	2317679	1.58e-2
9	165876500	7145748	4.31e-2

Figure 2. Number of logic functions and equivalence classes for a large set of industrial designs.

#Inputs	#NPN classes to cover % of logic functions							
	99%	98%	97%	96%	95%	90%	85%	80%
5	268(8.2%)	158(4.8%)	120(3.7%)	101(3.1%)	87(2.7%)	55(1.7%)	42(1.3%)	34(1.0%)
6	3573(10.2%)	1822(5.3%)	1168(3.4%)	831(2.4%)	638(1.9%)	311(0.9%)	207(0.6%)	148(0.4%)
7	54266(20.0%)	25364(9.3%)	15093(5.6%)	10139(3.7%)	7322(2.7%)	2437(0.9%)	1313(0.5%)	865(0.3%)
8	920963(39.7%)	410955(17.7%)	232892(10.0%)	148041(6.4%)	101368(4.4%)	26956(1.2%)	11399(0.5%)	6155(0.3%)
9	5486984(76.8%)	3828219(53.6%)	2276353(31.9%)	1460943(20.4%)	991073(13.9%)	215899(3.0%)	70242(1.0%)	29554(0.4%)

Figure 3. Number of NPN classes required to maintain a certain coverage (percentage) of logic functions. Percentages in each column are the percent of remaining NPN classes vs. the baseline of keeping all NPN classes.

remaining classes dropped below a specified percentage. The results are shown in Figure 3 and are striking. If one is willing to sacrifice a *small* percentage of function coverage, then the actual number of equivalence classes required to cover the remaining functions is *significantly* reduced.

IV. Library Generation

A library for matching consists of several things: (1) functions which are covered by the library (i.e., functions for which a LUT structure exists in the library); (2) different LUT structures; and (3) the mapping of functions to different LUT structures (i.e., details of how functions are implemented by the structures). The functions covered by the library and structures in the library relate to the ultimate effectiveness of the library. The memory overhead required to store the library relates to its ultimate efficiency.

A. Decompositions and LUT Structures

For each equivalence class, we compute LUT structures using Roth-Karp decomposition [11] as shown in Figure 4. For a given n-input logic function F (NPN class), we consider the division of its inputs into the *bound set (BS)*, the *shared set (SS)* and the *free set (FS)*. Decomposition is applied with certain restrictions:

1) The sum $|\mathrm{BS}|+|\mathrm{SS}| \leq K$, where K is the number of LUT inputs for the given technology.
2) The value $|\mathrm{BS}| \geq 1$, implying that the bound set is not empty.
3) The sum $|\mathrm{FS}|+|\mathrm{SS}|+1 < n$ such that the support set of a given logic function, H, is reduced (that is, made smaller than n). If the size of the support set of H is larger that K, H is decomposed recursively in the same fashion as F.

With these requirements, the logic function G is always implementable in a single K-LUT. The number of inputs to H is always smaller than F which means that we will eventually either find a decomposition via recursive decomposition calls or find that no decomposition exists.

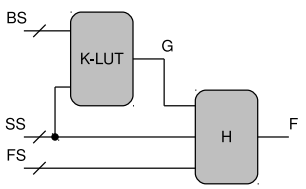

Figure 4. Roth-Karp decomposition used to create LUT structures.

The decomposition for a particular logic function is performed enumeratively; i.e., all possible bound and free sets are considered. At first glance, this might seem unnecessary. However, during resynthesis circuit performance must be considered. When using libraries and matching, all combinations of input sets must be considered to enable the "pivoting" of late-arriving signals closer to the output of the structure. Although we consider Roth-Karp decomposition, other decomposition techniques, such as DSD decompositions [8] or LUT cascades [12], could be used during library generation.

Once a given logic function is decomposed, we write out the logic function along with its decompositions into a file which forms the library of LUT structures.

B. Decomposition Pruning

It is possible that many LUT structures (for a given logic function) will be duplicated or redundant. Maintaining redundant LUT structures for a function only serves to increase the memory footprint of the library.

To prune redundant structures, we use the notion of "dominating structures". For a *given* logic function, consider two decompositions A and B. We can decide if A "dominates" B by considering the timing profile. For a structure, we define input delay for the i-th input to be the largest number of LUTs on a path from the i-th input to the output of the structure. The timing profile is defined as the ordered set of input delays. We can say that A *dominates* B if the delay of every input in the timing profile of A is less than or equal to the delay of every input in the timing profile of B **and** the area of A is less than or equal to the area of B. Note that dominance of one structure over another is done *per logic function*.

Dominating structures are illustrated in Figure 5 for a 7-input logic function. Each LUT structure requires 4 LUTs. If the LUT structure in Figure 5 (a) is computed first, then neither of the structures shown in Figure 5 (b) or (c) offer any benefit in delay or area and can be rejected.

Statistics regarding the number of LUT structures determined using Roth-Karp decomposition and pruning via timing profiles are presented in Figure 6 for a 4-LUT

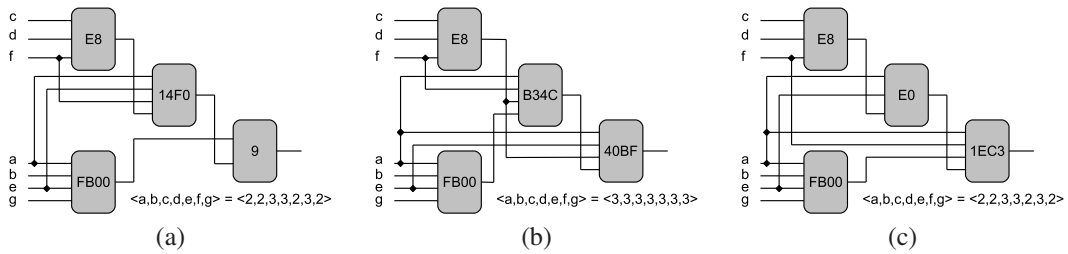

Figure 5. Pruning LUT structures using timing profiles. Given structure (a), both structures (b) and (c) are rejected because of timing profiles. Neither structure (b) nor (c) offer any benefit in performance or area over structure (a) for the given logic function (config bits are shown inside each LUT).

architecture. The average number of decompositions for each equivalence class is small.

V. Library Storage

It is important to determine the approximate amount of memory which will be required to store the library. Since the library consists of LUT structures, the storage requirements can be analyzed based on the amount of storage required to save a single LUT. A summary of the memory requirements for different sized *K*-LUTs is given in Figure 7 and is explained as follows.

A LUT requires certain information to be stored: (1) the configuration for the LUT and (2) the identifiers for the LUT inputs. For a *K*-LUT, the number of configuration bits is 2^K. The number of bits to store input identifiers for a *K*-LUT is $K \times r$ where r is the number of bits reserved per input. We consider difference sized LUTs as follows.

3**-LUT:** Configuration requires 8 bits. We can reserve 5 bits for each input where values 0...15 represent logic function inputs (i.e., up to 16 input functions). The values 16...31 identify other LUTs in the structure. Since practical structures require only 5 or 6 LUTs, all these values are not required. A total of $8+15=23$ bits are required and a single 3-LUT can be stored in one unsigned word.

4**-LUT:** Configuration requires 16 bits. Depending on the number of logic function inputs and/or the number of LUTs allowed in the structure, we can pack 4-LUTs in different ways. If we limit ourselves to 12 input functions and LUT structures with ≤ 4 LUTs, we can use only 4 bits per input. This implies that 16 bits are required for inputs for a total of $16+16=32$ bits (one unsigned word) to store a single 4-LUT. To represent larger functions and/or LUT structures with ≥ 4 LUTs, we require 5 bits per input and two unsigned words are required.

5**- and** 6**-LUT:** Configuration require 32 and 64 bits for 5- and 6- LUTs, respectively. We use 5 bits per LUT input to handle 16 input functions. A similar analysis to that given above implies that 2 and 3 unsigned words are required to 5- and 6-LUTs, respectively.

The storage requirements of the library can be computed based on the desired amount of coverage. For 4-LUTs (asuming structures with ≤ 4 LUTs, but two unsigned words per LUT), the library memory requirements are shown in Figure 8 (computing using information from Figure 6 and Figure 7). The memory requirements for functions with ≤ 7 inputs is reasonable. For 8- and 9-input functions, reasonable memory requirements are possible if a reduction in coverage is allowed. The memory requirements to store the equivalence classes must also be taken into account. Logic functions of 5, 6, 7, 8 and 9 inputs require 1, 2, 4, 8 and 16 unsigned words to store, respectively. Regardless, the practicality of storing a comprehensive library still holds once this additional memory is taken into account.

VI. Library Usage And Resynthesis

Once the library is loaded (prior to resynthesis), it is necessary to match the logic function to LUT structures stored in the library. This operation is straightforward. For any *k*-input cone of logic for which an alternative LUT structure is required, the cone's logic function is computed, NPN-encoded, and matched to the library of LUT structures via hash-lookup. If a match is found, the list of LUT structures is returned. When no match is found, the alternative response is to apply decomposition directly which is the approach we take.

It is also possible to match *k*-input functions in which *k*

#Inputs	#NPN classes	#Structures/NPN class
5	3269	2.77
6	34225	3.56
7	271646	5.87
8	2317679	5.52
9	7145748	4.91

Figure 6. Number of LUT structures in the library per equivalence class (4-LUT architecture).

#LUT Inputs	3	4	5	6
#Bits	23	32/36	57	94
#Unsigned words ($\lceil$(#Bits)$/32\rceil$)	1	1/2	2	3

Figure 7. Number of bits required to store a *K*-LUT in memory while maintaining all necessary information.

#Function Inputs	#Avg Decomp	Percentage coverage of logic functions 100% # Bytes	99% # Bytes	98% # Bytes	97% # Bytes	96% # Bytes	95% # Bytes	90% # Bytes	85% # Bytes	80% # Bytes
5	2.77	289.8K	23.8K	14.0K	10.6K	9.0K	7.7K	4.9K	3.7K	3.0K
6	3.56	3.9M	407.0K	207.6K	133.1K	94.7K	72.7K	35.4K	23.6K	16.9K
7	5.87	51.0M	10.2M	4.8M	2.8M	1.9M	1.4M	457.8K	246.6K	162.5K
8	5.52	409.4M	162.7M	72.6M	41.1M	26.2M	17.9M	4.8M	3.0M	1.1M
9	4.91	1122.7M	862.1M	601.5M	357.7M	229.5M	155.7M	33.9M	11.0M	4.6M

Figure 8. Estimated memory usage for libraries for LUT structures using 4-LUTs with a maximum of 4 LUTs per structure.

is larger than those structures stored in the library (e.g., we store structures with $k \leq 9$ inputs, but we could compute logic functions with > 9 inputs). We can use decomposition to extract a bound set and use library-based matching for the remainder of the decomposition. To be more specific (c.f. Figure 4), we use decomposition to generate different bound sets (function G) and use library matching to implement function H if the number of inputs is small enough. This hybrid scheme combines decomposition and matching for improved efficiency while facilitating the matching of functions with a slightly larger number of inputs.

VII. Numerical Results

We have implemented a depth-optimal technology mapper similar to [6], [9] and an area-oriented resynthesis algorithm. All software was implemented in C++. Our target architecture consists of 4-LUTs. In each pass of resynthesis, we examine the LUTs in the mapped network in topological order from inputs (primary inputs and flip-flop outputs) to outputs (primary outputs and flip-flop inputs). For each LUT we find a set of logic cones with ≤ 9 inputs. and attempt for find structures for each cone of logic. If there exists a structure such that (1) the worst-case logic depth of the network is not worsened and (2) the area of the network is reduced, then the structure is used to replace the cone of logic; i.e., we reduce area without harming depth.

We use a set of 100 industrial designs. All designs are subjected to commercial high-level synthesis and technology-independent logic optimizations prior to technology mapping. Designs range in size from 100 to 25K 4-LUTs and have varying logic depths after technology mapping.

Our first experiment determined the effectiveness of the resynthesis algorithm itself. Figure 9 shows the area saved per design (sorted by area reduction) with two passes of resynthesis. The area saved is 3.2% on average with a maximum savings of 20.5%. The is reasonable given that our designs have been pre-processed using commerical tools. The run-time of resynthesis was on the order of minutes for the larger designs.

Our second experiment was aimed at determining the benefit of using libraries in terms of run-time, as well as to form an idea of the penalties to be paid for using "less complete" libraries. Two passes of area-oriented resynthesis were performed but with different logic function coverage (c.f. Figure 3). In this experiment, the area saved is not affected compared to the previous experiment; the only difference is whether or not the structures for a function are found in the library or produced via decomposition. Reduced coverage will require more functions to be decomposed on-the-fly which, in turn, will increase the run-time needed to match a function. Figure 10 shows the increase in run-time as the library coverage is reduced relative to using a full library. The run-time increase can be as much as 11.5$\times$ if the coverage is reduced to 90%. Figure 10 also shows the hit ratios for different coverages. By a "hit" we mean that a logic function is matched to a structure stored in the library. When a hit is not found, the logic function is matched by performing on-the-fly decomposition. As expected, with a full library, the hit ratio is close to the ideal scenario of 1.0, but decreases as the library coverage decreases. Note that a hit ratio lower than the coverage implies that resynthesis discovered logic functions that were not discovered during library generation. With a coverage as low as 90%, the hit ratio remains reasonable at 0.82 and lends credence to the notion of using a small but well-conceived library of structures.

Our last experiment involved determining if run-time can be improved when a logic function is not matched to the library. Heuristics can be used to reduce the run-time of decomposition; e.g., shared set sizes can be restricted and bound sets can be selected intelligently to include only late-arriving signals. Heuristics used during decompostion to improve run-time imply that some decompositions will not be discovered which may impact the final quality of result.

Figure 11 illustrates the consequences of speeding up decomposition by restricting the shared set size to be ≤ 1.

NPN Coverage	CPU Ratio	Hit Ratio
100%	1.00	0.98
99%	1.71	0.96
98%	3.44	0.94
97%	3.94	0.92
96%	4.66	0.90
95%	5.33	0.89
90%	11.5	0.82

Figure 10. Consequences to CPU time and hash-lookup "hits" to the library as library coverage decreases.

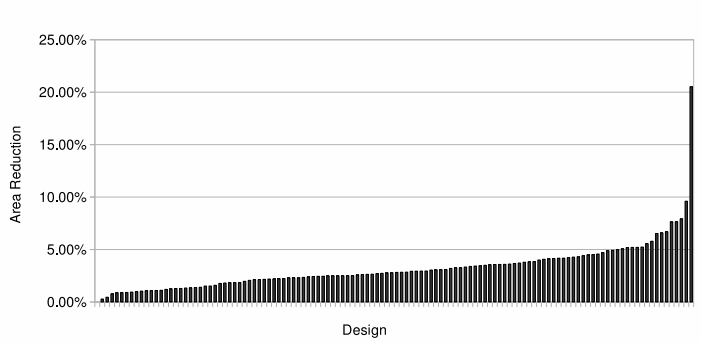

Figure 9. Area savings obtained from resynthesis. Average and maximum savings are 3.21% and 20.51%, respectively.

This reduction in the flexibility of the decomposition algorithm has little impact of the final quality of result. The hit ratio is not affected (as expected). However, run-time is significantly improved. Our analysis showed that the quality of result is not affected because of the large number of hits to the library—restricted decomposition is only applied to a small number of functions and therefore its overall impact is negligible. Libraries are still beneficial—they serve to maintain the quality of result and still yields a run-time benefit of as much as 2.87× in our experiments.

VIII. Conclusions

We have proposed a method for Boolean matching based on the use of precomputed libraries. Libraries of LUT structures are possible due to: (1) careful consideration of the stored logic functions; (2) the use of equivalence classes; and (3) the use of timing profiles to eliminate redundant structures. We have empirically confirmed our results for a 4-LUT architecture. Presented numerical results demonstrate the effectiveness of the library in practice; the use of our library-based technique can serve to speed-up an area-oriented resynthesis algorithm when compared to one based on restrictive decomposition.

Future work would include library generation for $10-12$ inputs logic functions and the consideration of 5- and 6-LUT architectures. We would also like to test our approach within a physical resynthesis algorithm that is coupled to an incremental placer such as that described in [10].

NPN Coverage	%Area Reduction Avg	Max	CPU Ratio	Hit Ratio
100%	3.20	20.51	1.00	0.98
99%	3.17	20.51	1.47	0.96
98%	3.17	20.51	1.76	0.94
97%	3.16	20.51	1.89	0.92
96%	3.16	20.51	1.99	0.90
95%	3.16	20.51	2.02	0.89
90%	3.16	20.51	2.87	0.82

Figure 11. Consequences to CPU time and hits to the library as library coverage decreases and on-the-fly decomposition is restricted.

References

[1] A. Abdollanhi and M. Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *Proc. DAC*, pages 379–384, 2005.

[2] D. Chai and A. Kuelmann. Building a better Boolean matcher and symmetry detector. In *Proc. DATE*, pages 1079–1084, 2006.

[3] Y. Hu, V. Shih, R. Majumdar, and L. He. Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping. In *Proc. IWLS*, 2007.

[4] H. Kim and J. Lillis. A framework for logic-level logic restructuring. In *Proc. ISPD*, pages 87–94, 2008.

[5] A. Ling, D. Singh, and S. Brown. FPGA technology mapping: a study in optimality. In *Proc. DAC*, pages 427–432, 2005.

[6] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *TCAD*, 25(11):2331–2340, November 2006.

[7] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proc. DAC*, pages 532–536, 2006.

[8] A. Mishchenko, S. Chatterjee, and R. Brayton. Fast Boolean matching for LUT structures. Technical report, ERL technical report, UC Berkeley, 2007.

[9] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. *TCAD*, 26(2):250–253, February 2007.

[10] V. Pevzner, A. Kennings, and A. Fox. Physical optimization for FPGAs using post-placement topology rewriting. In *Proc. ISPD*, pages 91–98, 2009.

[11] J. P. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2):227—238, 1962.

[12] T. Sasao and A. Mishchenko. LUTMIN: FPGA logic synthesis with MUX-based and cascade realizations. In *Proc. IWLS*, pages 310–316, 2009.