Generating Efficient Libraries for use in FPGA Resynthesis Algorithms

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

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

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

ABSTRACT

The ability to efficiently match completely-specified 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 completely-specified logic functions of 9 to 12 inputs in K-LUT structures. Our proposed method is based on the off-line generation of libraries of LUT structures. During FPGA resynthesis, matching is performed efficiently using NPN encoding and hash table look-ups to find alternative LUT structures for the implementation of completely-specified logic functions. Generating an effective library of LUT structures may seem prohibitive due to the overwhelming number of potential logic functions which must be considered and represented in the library; however, we show that, by careful consideration of which logic functions and LUT structures to keep in the library, it is, indeed, possible to generate useful, compact libraries. In addition to describing our library generation procedure, we present numerical results demonstrating its effectiveness when employed during area-oriented resynthesis after FPGA technology mapping.

1. INTRODUCTION

It is well understood that recent generations of FPGAs have improved significantly in speed, capacity and functionality. Consequently, FPGAs are displacing ASICs in many application domains. The most common architecture for an FPGA is the so-called 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 $3 \leq K \leq 6$.

In this paper 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 the implementation of logic functions with between 9 and 12 inputs. For example, Figure 1 illustrates a structure which consists of three 4-LUTs suitable for implementing logic functions of up to 10 inputs. Due to the particular wiring of the structure, not all logic functions with ≤ 10 inputs can be implemented. As stated in [13], 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 whether or not 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. In this paper, we consider aspects of both of these forms of the matching problem.

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 [13] and post-placement networks [15] 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 (such as whether the cell is on the critical path). For a given 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 for the cones of logic. 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.

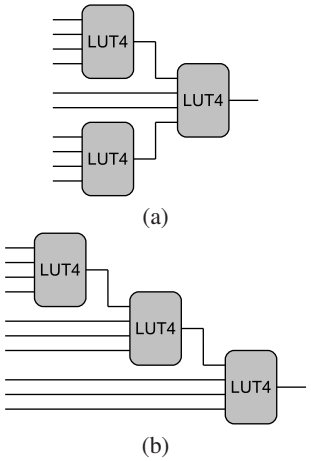

Figure 1: Examples of LUT structures consisting of three 4-LUTs which are suitable for implementing many logic functions of up to 10 inputs.

The contributions of this paper are several-fold:

- We present statistics for a set of industrial FPGA circuits and show that the number of k-input logic functions which occur in practice is typically much smaller than the entire set of possible logic functions. Furthermore, we show that the number of NPN equivalence classes required to cover a high percentage of k-input logic functions is very 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 a the off-line generation of an

effective library of LUT structures can be performed. The approach to library generation described in this paper is based on decomposition algorithms and the idea of dominant LUT structures. We quantify the necessary memory usage to store the resultant library.

- We provide numerical results to demonstrate the compactness of the library approach. Furthermore, we use the proposed matching algorithm inside an area-oriented resynthesis flow applied after technology mapping to demonstrate the efficacy of our proposed matching approach based on precomputed libraries.

The remainder of this paper is organized as follows. Section 2 provides background information on the matching problem and reviews existing approaches. In Section 3, we present an analysis of the logic functions found in a large set of industrial designs. This analysis serves to form the basis of our library-based matching procedure. In Section 4, our approach to library generation and LUT structure pruning is described. Section 5 presents an analysis of the amount of memory required to store the generated libraries. Section 6 illustrates the use of our library-based matching algorithm during resynthesis. Numerical results are presented in Section 7 to demonstrate the effectiveness of our proposed ideas on a set of actual designs. Finally, Section 8 summarizes this work and describes additional work which can be performed.

2. BACKGROUND

Historical approaches to matching include: (1) structural-based techniques, (2) SAT-based techniques, (3) class-based techniques, 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 particular LUT structures.

Structural-based techniques [9] are rooted in the idea of remapping cones of logic. These approaches use the same concepts as those employed in structural FPGA technology mappers [3–5, 11, 14]. Structural techniques are very efficient but are not necessarily effective at matching to particular LUT structures. These techniques 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 [6–8, 10, 17] are effective at matching logic functions to particular LUT structures—they are guaranteed to find a match if one exists. Unfortunately, SAT-based techniques are not efficient for matching logic functions with a large number of inputs (≥ 10). Similarly, the LUT structures considered in previous literature are typically limited in size to $\leq$ 3-LUTs which restricts the ability to successfully match larger functions. Larger functions and additional LUTs can significantly increase the size and difficulty of the resulting satisfiability problem and can lead to increased run-time when solving the matching problem.

Class-based techniques [15] are efficient due to the availability of equivalence class encoding algorithms (such as [1, 2] for finding NPN equivalence classes). However, class-based methods require libraries against which matching can be performed. Consequently, their effectiveness depends greatly on the library. For LUT structures, the use of libraries implies determining all of the logic functions implemented within a particular LUT structure. While feasible for LUT structures with only a few LUTs (and small K), it is not likely feasible to generate complete libraries for larger LUT structures (and large K) due to the flexibility implied by having more LUTs with increased LUT flexibility.

Decomposition-based techniques [13] attempt to apply logic decomposition to "split" a logic function into individual pieces such that each piece can be implemented within a K-LUT. These techniques vary greatly in terms of efficiency depending on the logic function being matched and the particular decomposition strategy employed. Typically, decomposition can be effective but can possibly miss certain matches due to the various heuristics employed to speed-up the decomposition, such as restrictions in bound set selection.

Our point of view is that *class-based* and *decomposition-based* techniques are the more promising techniques for matching to LUT structures. As previously mentioned, the main disadvantage of class-based techniques is the need to generate a library against which matching can be performed. If the task of library generation could be overcome, class-based techniques would be well-suited for logic functions of ≤ 12 inputs due to the efficiency of available encoding algorithms. Decomposition-based techniques, on the other hand, offer the flexibility of matching a logic function to a LUT structure when afforded the flexibility to use different techniques to explore many different bound sets. In other words, 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 while performing matching.

3. FUNCTION COVERAGE

As mentioned, class-based techniques are complete for a given LUT structure provided that all possible functions implementable by the structure are enumerated. For larger K-LUTs (and larger LUT structures), the full enumeration of implementable functions is not possible. However, it has often been observed for a given value of n, only a small subset of n-input functions occur in practice on real networks [12]. It can be concluded that, in practice, it is *not* required to fully enumerate LUT structures to determine all implementable functions. Rather than first defining LUT structures and then computing the implementable functions, it is possble to determine which functions need to be considered and then to determine how such functions can be implemented via structures.

To determine an *appropriate* set of logic functions to store in a library, we have performed the following experiment. We have taken a set of 100 industrial designs targeted toward implementation in an FPGA. These designs have been synthesized, subjected to technology-independent optimizations, and converted into subject graphs. Enumerative cut generation has been performed on each of these subject graphs to determine all of the logic functions present within the design suite. The logic functions have been NPN-encoded along with the frequency of occurrence of each NPN equivalence class. Appendix A describes the format of the data file employed for gathering these NPN-encoded functions.

The results of the analysis of the NPN classes are presented in Figure 2 for n-input logic functions for $5 \leq n \leq 9$. From Figure 2, we see that the number of n-input NPN equivalence classes that occur in practice (to cover all the logic functions over a large set of designs) is remarkably small compared to the total number of equivalence classes possible. This observation implies that full enumeration of LUT structures is not required to generate an effective library of LUT structures for matching.

This analysis can be carried further. Many NPN equivalence classes only occur 1 or 2 times over all designs. It is therefore possible (and reasonable) to *exclude* these less common NPN equivalence classes to reduce the amount of information to be stored in the library. To judge the usefulness of removing less common equivalence classes (logic functions), we performed the following experiment.

#Inputs	#NPN classes to cover % of logic functions								
	100%	99%	98%	97%	96%	95%	90%	85%	80%
5	3269	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	34225	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	271646	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	2317679	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	7145748	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 for n-input logic functions required to maintain a certain coverage (percentage) of logic functions over a large set of industrial designs. Percentages shown in each column are the percent of remaining NPN classes vs. the baseline of keeping all NPN classes.

#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: Statistics about the functions and equivalence classes found over a large set of industrial designs. The number of equivalence classes is a small fraction of the possible number of equivalence classes for a given number of inputs.

For n-input functions, we sorted the equivalence classes according to the number of times they occur in the set of designs from smallest to largest number of occurrences. We then removed as many equivalence classes as possible while maintaining a *target coverage*; i.e., we removed equivalence classes until the number of logic functions encoded to the remaining NPN classes dropped below a specified percentage. The results of this experiment are illustrated in Figure 3 for n-input logic functions where $5 \leq n \leq 9$. The results of this experiment are striking: they indicate that if one is willing to sacrifice a *small* percentage of function coverage, the actual number of equivalence classes required to cover the remaining functions is *significantly* reduced.

4. LIBRARY GENERATION

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

4.1 Decompositions and LUT Structures

For each NPN class discovered, it is necessary to compute LUT structures which can implement the NPN class. To accomplish this task, we employ Roth-Karp decomposition [16], as illustrated in Figure 4. For a given n-input logic function F (NPN class), we consider the division of its inputs into three sets, namely the bound set (BS), the shared set (SS) and the free set (FS). In performing the Roth-Karp decomposition, we observe certain restrictions:

1. The sum $|\text{BS}| + |\text{SS}| \leq K$, where K is the number of LUT inputs for the given technology.

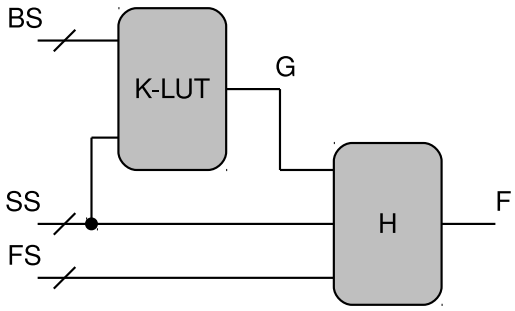

Figure 4: Illustration of the Roth-Karp decomposition technique used to create LUT structures for logic functions.

2. The value $|\text{BS}| \geq 1$, implying that the bound set is not empty.
3. The sum $|\text{FS}| + |\text{SS}| + 1 \leq n$ 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, we find that the logic function G of the decomposition is always implementable in a single K-LUT. Further, the number of inputs to H is always smaller than F. This result means that we will eventually either find a decomposition (via recursive calls to the decomposition code) or find that no decomposition exists.

The decomposition for a particular logic function is performed enumeratively. (We consider all possible combinations of bound, shared and free sets.) At first glance, it might not seem that this is required and only serves to add additional LUT structures to the library which are unnecessary; however, during resynthesis, it is important to consider circuit performance and the use of libraries with matching requires that all combinations of the various input sets are enumerated. This enables the resynthesis algorithm to "pivot" or "permute" late-arriving signals closer to the output of the LUT structure. (Conversely, if the matching were done strictly with decomposition, the resynthesis could "suggest" or "require" that certain inputs are placed into the different sets of inputs; however, this luxury is not available when using a library of predefined LUT structures.) Finally, we note that, although we consider only Roth-Karp decomposition techniques in this paper, other decomposition techniques, such as DSD decompositions [13] or LUT cascades [18], 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. Details of this file format are provided in Appendix B.

4.2 Decomposition Pruning

Because the decomposition is performed enumeratively (to allow for the permuting of inputs for purposes of timing improvement during resynthesis), it is possible that many LUT structures will be generated for a given n-input logic function. It is entirely possible that LUT structures will be duplicated and will actually be redundant. Maintaining all enumerated LUT structures will only serve to increase the size of the library without offering any benefit during resynthesis.

To address this issue, we introduce the notion of "dominating structures". For a given n-input logic function, consider two decompositions A and B. We can decide if A "dominates" B by considering the timing profile. For a LUT 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 LUT structure. The timing profile is defined as the ordered set of input delays. Using their timing profiles, we can say that *A dominates B* if the delay of every input in the profile of A is less than or equal to the delay of every input in the profile of B **and** the area of A is less than or equal to the area of B.

We illustrate the concept of dominating structures in Figure 5, which shows several different structures for the same 7-input logic function (assuming 4-LUTs). From Figure 5, we see that each of the LUT structures requires 4 LUTs. If we assume that the LUT structure in Figure 5 (a) is computed first, then it is clear from the timing profiles that neither of the other two LUT structures shown in Figure 5 (b) or (c) offer any benefit in delay or area and, hence, can be rejected from the library.

With the LUT structures determined from Roth-Karp decomposition and pruned using timing profiles, statistics about the generated library files were gathered and are presented in Figure 6. These results assume a target architecture comprising 4-input LUTs which are commonly-employed in the academic literature (and in several commerical FPGA architectures) and are therefore a reasonable basis for comparison. We observe that that the the average number of LUT structures required for each NPN class is reasonably small.

5. LIBRARY STORAGE

It is important to determine the approximate amount of memory which will be required to store the library as the library must be loaded and stored in memory prior to its use in an sort of resynthesis algorithm. 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 in a structure; the amount of storage for a structure can then be computed by counting the number of LUTs in the structure.

A single LUT in a structure requires a certain information to be maintained: (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 which is practical for $3 \leq K \leq 6$. 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. We reserve the bits 0...15 to represent logic function inputs (e.g., we can handle LUT structures for up to 16 input functions). This leaves the remaining indices of 16...31 to identify individual LUTs in the LUT structure. In fact, since practical LUT structures consist of fewer than 5 or 6 LUTs, all remaining indices are not required but, nevertheless, 5 bits are needed. Hence, a total of $8+15=23$ bits are required and the single 3-LUT can be stored in a single 32-bit 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. For instance, if we limit ourselves to 12 input logic functions and LUT structures with ≤ 4 LUTs, we can use only 4 bits per input. This implies that 16 bits are required for inputs to a total of $16+16=32$ bits to store a single 4-LUT. This requires only a single 32-bit word. To represent larger logic functions and/or LUT structures with ≥ 4 LUTs, we require 5 bits per input and, consequently, two 32-bit words are required.

5**- and** 6**-LUT:** Configuration bits require 32 and 64 bits for 5- and 6− LUTs, respectively. Since it is likely that logic functions with up to 16 inputs are required and that larger LUT structures are desireable, we must use 5 bits per LUT input. A similar analysis to that given above implies that 2 and 3 32-bit words are required to 5- and 6-LUTs, respectively.

A summary of the memory requirements for different sized K-LUTs is given in Figure 7 for different values of K. Finally, assuming that a LUT structure can consist of a maximum of M LUTs where is typically $1 \leq M \leq 6$, the amount of storage per LUT structure can be easily calculated.

Given the storage requirements for a LUT structure, we can estimate the storage requirements for the library based on the amount of coverage desired within the library. Again, we consider the case of 4-LUTs, which are common in the academic literature. We restricted ourselves to structures with at most four 4-LUTs, but assume that 2, 32-bit words are required for each LUT in a LUT structure. Using the results presented in Figure 6 and Figure 7, we present the storage requirements for n-input logic functions in Figure 8. From Figure 8, we conclude that the memory overhead to store the generated libraries is reasonable for up to and including 7-input functions. For 8- and 9-inputs, reasonable memory requirements are possible if a reduction in coverage is allowed. (We note that memory requirements to store the equivalence classes also needs to be taken into account. Logic functions of 5, 6, 7, 8 and 9 inputs require 1, 2, 4, 8 and 16, 32-bit words to store, respectively. Hence, the appropriate number of words needs to be multiplied by

#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: Average number of LUT structures in the library per NPN class (found in a large set of industrial designs) for logic functions with different numbers of inputs using 4-LUTs for decomposition.

#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 memory bits required to store a K-LUT in memory while maintaining all necessary information. Requirements for 4-LUTs vary depending on the size of the LUT structures and the number of logic function inputs.

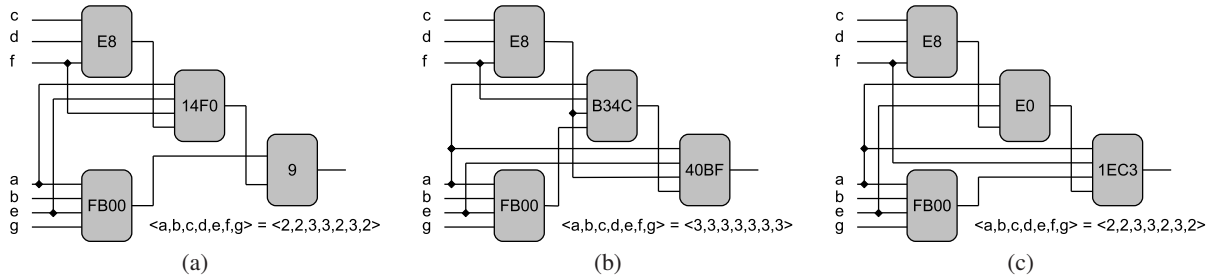

Figure 5: Example of pruning LUT structures using timing profiles. Assume that structure (a) has already been computed. Structure (b) is rejected because its timing profile is worse than that of (a). Structure (c) is rejected as well because its timing profile is equal to that of (a) and its area is not better than (a). Neither structure (b) nor (c) offer any benefit in performance or area over (a).

		Percentage coverage of logic functions								
#Function Inputs	**#Avg Decomp**	**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 The number of 32-bit words per LUT structure is 8 assuming at most four 4-LUTs per structure and the worst case of every structure requiring four 4-LUTs.

the number of equivalence classes and added to the values shown in Figure 8 for the particular percentage of function coverage. However, our conclusions regarding the practicality of storing a fairly comprehensive library still hold once this additional memory usage is taken into account.)

To further reduce memory usage, it is possible to skip the less common NPN classes, as mentioned in Section 3. This can remove a significant number of NPN classes and LUT structures from the libraries without necessarily compromising the quality of the libraries. In the event that a particular logic function cannot be found in the library, it is alternatively possible to call the decomposition code directly for the infrequent logic functions.

```
    Input: k-input cone of logic c, library of LUT structures lib.
    Output: list of LUT structures.
1   begin
2     // determine the k-input logic function f implemented
3     f ← simulate_logic_cone(c);
4     // determine the equivalence class for f
5     g ← npn_encode(f, ip_perm, ip_phase, op_phase);
6     // determine if a cell implementing g exists.
7     return hash_lookup(lib, g);
8   end
```

Figure 9: Algorithm for k-input logic functions to precomputed LUT structures stored in the libraries.

6. LIBRARY USAGE DURING RESYNTHESIS

Once the library is generated and loaded (prior to performing resynthesis), it is necessary to match the logic function to the LUT structures stored in the library. This operation is straightforward; the relevant pseudocode is provided in Figure 9. Essentially, the resynthesis algorithm computes a k-input cone of logic for which an alternative LUT structure is required. The logic function implemented by the cone of logic 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. If no match is found, an empty 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 consider computing k-input cones of logic in which k is larger than those LUT structures stored in the library (i.e., in our case, we store libraries for $k \leq 9$, but we could compute cones with > 9 inputs). In this case, we can again turn to the direct application of decomposition. However, we have another alternative as well. Specifically, we can use a *hybrid* scheme in which we apply decomposition to extract an appropriate bound set and use library-based matching for the remainder of the decomposition. To be more specific, we can refer to Figure 4. In the hybrid scheme, we use decomposition to generate different bound sets (the logic function G) and subsequently use the library-based matching to find LUT structures for logic function H since the number of inputs to H could be small enough to be matched to the library. This hybrid scheme effectively combines decomposition and matching for improved efficiency and allows for the fast generation of LUT structures for slightly larger logic functions than those for which libraries have been prepared.

7. NUMERICAL RESULTS

As we require a resynthesis algorithm to test our proposed ideas, we have implemented, within an academic framework, an area-oriented resynthesis algorithm which can be applied after technology mapping. Specifically, after mapping, we process the mapped LUT network via several passes of area-oriented resynthesis. In each pass, we examine the cells in the LUT network in topological order from inputs (primary inputs and flip-flop outputs) to outputs

(primary outputs and flip-flop inputs). For any given cell n, we apply enumerative cut generation up to some maximum k to determine a set of logic cones which implement the function of cell n. We then select the logic cones and attempt to determine an alternative LUT structure which can implement each cone of logic. If there exists an alternative LUT structure such that (1) the worst-case logic depth of the circuit is not worsened and (2) the resulting area (measured by the number of LUTs) of the network is reduced, then the LUT structure is used to replace the cone of logic. In other words, the objective of our area-oriented resynthesis algorithm is to reduce the area of the LUT network without harming the logic depth of the network.

Our target FPGA architecture consists of 4-LUTs, as these are a common basis for academic comparison. Since our algorithm requires technology-mapped LUT networks as input, we have implemented a mapping algorithm along the lines of [11, 14]. All of our software was implemented in C++ and tested on a set of roughly 100 industrial designs. All designs are subjected to a commercial high-level synthesis tool, with commercial technology-independent logic optimizations applied prior to technology-mapping. In terms of size, our designs range from as few as 100 4-LUTs to as many as 25K 4-LUTs after technology mapping. We consider cones of logic with up to 9-inputs to be consistent with the LUT structures described throughout the paper. We conducted several different experiments to judge our proposed ideas, as detailed below.

7.1 Effectiveness of area-driven resynthesis

Our first experiment was aimed at determining the effectiveness of the area-oriented resynthesis algorithm itself. Figure 10 shows the area saved per design (sorted by area reduction). These results were obtained using two passes of resynthesis. From Figure 10, it is observed that area-oriented resynthesis saved an average of 3.2% LUTs and as much as 20.5% LUTs on our design suite. Initially, the average reduction in area appears small, but it is important to consider that our designs have been pre-processed using commercial FPGA tools.

7.2 Run-time impact and incomplete libraries

Our next 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. To accomplish these objectives, 2 passes of area-oriented resynthesis were performed but with different coverages of logic functions as illustrated previously in Figure 3. Recall that with reduced library sizes, there will be additional logic functions for which alternative LUT structures must be found using decomposition directly (rather than hash table look-ups). It is expected that run time will increase with less coverage, but with the benefit of a reduction in the library sizes and required memory usage. The results of these experiments are shown in Figure 11. Figure 11 shows the increase in CPU time taken, as the library coverage is reduced, relative to using a full library. As can be seen, the run-time increase for area-oriented resynthesis can increase by as much as 11.5× if the coverage is reduced to 90%. This figure also shows the resulting hit ratios for different coverages. By a "hit" we mean that a logic function is matched to a LUT structure stored in the library. When a hit is not found, the logic function is matched to alternative LUT structures by performing decomposition on-the-fly. As expected, with a full library, the hit ratio is 0.98 which is close to the ideal scenario of 1.0. As library coverage decreases, so does the hit ratio. Note that the hit ratio is lower than the coverage, which implies that additional logic functions are encountered which were not discovered during the library generation—this is an expected result. However,

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

even with a coverage as low as $90\% - 95\%$, the hit ratio remains very reasonable at 0.82, lending credence to the notion of using a small but well-conceived library of pre-computed LUT structures.

7.3 Run-time impact of heuristic decomposition

Our last experiment involved determining if improvements to the run-time efficiency can be obtained by considering alternative options when a logic function is not matched to the library. Since the decomposition techniques used to generate libraries are reasonably enumerative, effective heuristics can be employed to reduce the run-time required to perform decomposition on-the-fly. For instance, the shared set sizes can be restricted—this can serve to reduce the number of LUT structures obtained. Further, since decomposition is done on-the-fly, it is possible to be more intelligent about what inputs to a logic function can be in the bound and shared sets while maintaining timing. (For example, a timing critical signal should be pivoted to the output of a LUT structure to avoid harming timing, which will reduce the number of decomposition options that need to be considered.) Of course, additional heuristics imposed on the decomposition techniques to improve run-time come at the expense of missing certain decompositions, which can, in turn, impact the quality of the final result.

Figure 12 presents the results from this experiment. In particular, to speed up the on-the-fly decomposition, we imposed the restriction that the maximum number of shared variables during decomposition can be at most 1; this significantly reduced the sorts of LUT structures which can be found by the decomposition algorithm. As can be seen from Figure 12, reducing the flexibility of the on-the-fly decomposition has little to no impact on the overall quality of result (the average and maximum amount of area reduc-

NPN Coverage	%Area Reduction		CPU Ratio	Hit Ratio
	Avg	Max		
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 12: Consequences to CPU time and hits to the library as library coverage decreases and on-the-fly decomposition is restricted to reduce the number of potential LUT structures that can be found.

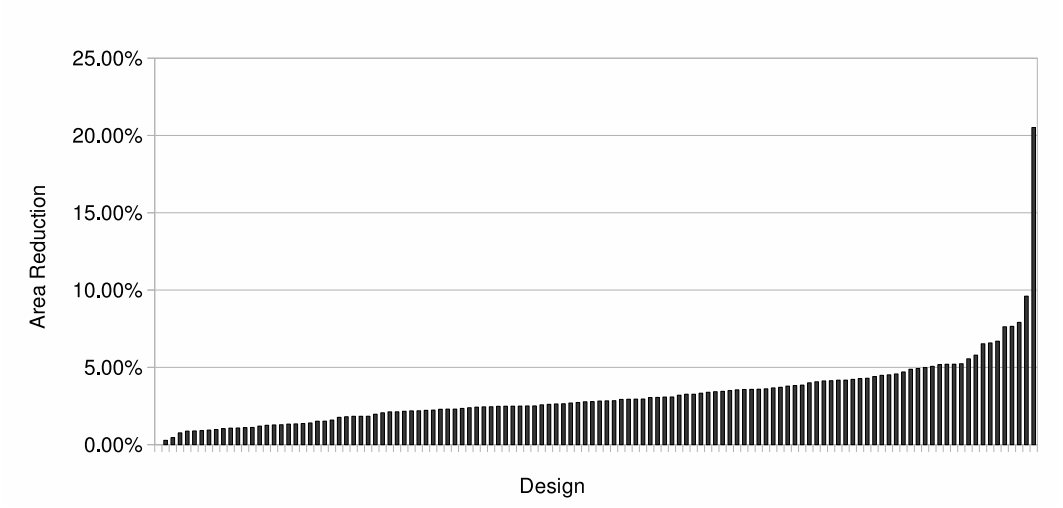

Figure 10: Percentage area reduction obtained from area-oriented resynthesis; an average reduction of 3.21% is obtained with a maximum reduction of 20.51%.

tion achieved). Further, the hit ratio is not affected, as expected; however, using a faster decomposition routine can significantly reduce the total amount of run-time required to perform resynthesis. Our analysis show that the quality of result is not affected because of the large number of hits to the library—the simpler decomposition is only applied to a small number of functions and therefore its overall impact on the quality of result is negligible. Finally, we note that despite the run-time improvements offered by simpler decomposition, the use of a library is still beneficial: it serves to maintain the quality of result and offers a run-time improvement of as much as 2.87× in our experiments.

8. CONCLUSIONS

FPGA resynthesis algorithms typically require Boolean matching to find alternative LUT structures for small k-input cones of logic where $9 \leq k \leq 12$. While much emphasis has been directed at SAT and decomposition-based techniques, we have proposed a method based on the use of precomputed libraries.

We have demonstrated that construction of a library of LUT structures is possible, despite the impending size of the library (due to the use of LUTs) through (1) careful consideration of the stored logic functions; (2) the use of equivalence classes, and (3) the use of timing profiles to eliminate redundant LUT structures. Our analyses indicate that it is practical to generate libraries of LUT structures, and have confirmed our results using architectures of 4-LUTs. We also note that compared to SAT-based approaches, our approach can take advantage of larger LUT structures (e.g., most SAT-based approaches targeting 4-LUT architectures limit themselves to structures with at most three 4-LUTs).

Finally, we have presented numerical results demonstrating the effectiveness of the library in practice. Our results show that 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. Further, the use of the library with more enumerative decompositions can contribute to a (small) improvement in the final quality of result.

Going forward, we would like to continue to generate libraries for functions of between $10-12$ inputs. Also, we would like to consider 5-LUT and 6-LUT architectures. Finally, we would like to test our algorithm within a physical resynthesis algorithm coupled with incremental placement such as that described in [15].

APPENDIX

A. NPN RESULT GATHERING

The results of the experiment in Section 3 are written to a single file for a particular number of logic function inputs. This concept is illustrated in Figure 13 for 7-input functions. The format of the file is a sequence of lines in which each line lists one of the discovered NPN classes (expressed in hexadecimal notation) followed by its number of occurrences. It is this file of NPN classes which is

```
0660600660060660F00F06600660F00F 24
00000000000000007E000000000F81F 1
001100441A114F4400AA00FFFFAAFFFF 2
00000000001100440001010B00151FAB 4
8181817F7E7E7E807E7E7E808181817F 22
01FF01000100FE00000000FF00FFFFFF 1
00000000010137370 0FF00FF0F0FFFF 9
000F555A00F055A5FF03AA56FFFCAAA9 2
...
```

Figure 13: Format of the output file for gathering NPN equivalence classes over a set of designs. Each line lists and NPN equivalence class along with its number of occurrences.

processed to analyze the discovered classes. One of the benefits of using such a format is that additional NPN classes can simply be appended to the end of the file (if, for instance, additional benchmarks are used and new NPN classes are discovered).

B. LIBRARY FORMAT

Our chosen file format for logic functions and their decompositions is illustrated in Figure 14. Figure 14 shows a small selection of the library generated for 6-input logic functions. The format of the file is as follows. Each line lists the LUT structures for a given logic function. The logic function is provided as the first token (in HEX) followed by a space-separated list of its LUT structures. The format of the decomposition is:

```
<truth_table>'('<arg1>,<arg2>,...,<argK>')'
```

```
0000000700077B77 0107(e,f,8488(a,b,c,d),E(c,d)) 0B07(a,b,FEE0(c,d,e,f),4(c,d))
500F00000000722F 1909(e,f,BC(a,c,d),2(a,b)) 002F(a,b,BC(a,c,d),E6(e,f,BC(a,c,d))) 1C03(b,f,BC(a,c,d),2C(a,e,BC(a,c,d)))
00000F0F65FB65FB 101F(c,e,f,9A04(a,b,c,d))
1818181818F81FFF 18FF(a,b,c,FFD8(c,d,e,f))
000000000A5027FF 1(f,EC70(a,d,e,F0D8(a,b,c,e))) 0853(d,e,F0D8(a,b,c,e),3FA0(a,d,e,f)) 1(f,DEB0(a,c,e,F0D0(a,b,d,e)))
444422226FF6F66F 462F(a,b,f,F096(c,d,e,f))
0F0F1510F0F0EAEF 5A69(c,e,f,E5E0(a,b,c,d)) 3C4B(b,c,f,FF98(a,c,d,e)) 1EC3(a,e,f,CC74(b,c,d,e))
00000088000F77FF 1107(d,e,f,F088(a,b,c,e))
...
```

Figure 14: Format of the output file for gathering the discovered NPN classes over a set of designs. The file lists, per line, an NPN class found along with its number of occurrences.

where each argument is either an input of the logic function or a formula (another LUT). For instance, the decomposition described by `1107(d,e,f,F088(a,b,c,e))` is for a 6-input logic function and consists of two 4-LUTs with truth tables equal to `1107` and `F088` (expressed in hexadecimal). Further, the 4-LUT with truth table `F088` drives the 4-th input of the 4-LUT with truth table `1107`. The lower-case variables (a,b,c, etc.) are reserved for representing the inputs of the logic functions. The upper-case variables (A through F) are reserved for representing hexadecimal truth tables. In the parenthesis, the arguments are listed from the least significant input to the most significant input. No spaces are allowed in the decomposition format. We chose this format carefully in order to allow for easy updating and exchange of library files (e.g., due to obtaining additional LUT structures from either different decomposition techniques or from the analysis of additional benchmarks that identify additional, useful, NPN classes that should be represented within the library).

C. 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] D. Chen and J. Cong. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *Proc. ICCAD*, pages 752–759, 2004.

[4] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *TCAD*, 13(1):1–12, January 1994.

[5] J. Cong and Y. Ding. Combinational logic synthesis for LUT based field programmable gate arrays. *TODAES*, 1(2):145–204, April 1996.

[6] J. Cong and K. Minkovich. Improved SAT-based Boolean matching using implicants for LUT-based FPGAs. In *Proc. FPGA*, pages 139–147, 2007.

[7] Y. Hu, V. Shih, R. Majumdar, and L. He. Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping. In *Proc. ICCAD*, pages 350–353, 2007.

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

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

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

[17] S. Safarpour, A. Veneris, G. Baecklet, and R. Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. In *Proc. DAC*, pages 466–471, 2006.

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