A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping

Gai Liu and Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{gl387, zhiruz}@cornell.edu

Abstract

Modern FPGA synthesis tools typically apply a predetermined sequence of logic optimizations on the input logic network before carrying out technology mapping. While the "known recipes" of logic transformations often lead to improved mapping results, there remains a nontrivial gap between the quality metrics driving the pre-mapping logic optimizations and those targeted by the actual technology mapping. Needless to mention, such miscorrelations would eventually result in suboptimal quality of results.

In this paper we propose PIMap, which couples logic transformations and technology mapping under an iterative improvement framework to minimize the circuit area for LUT-based FPGAs. In each iteration, PIMap randomly proposes a transformation on the given logic network from an ensemble of candidate optimizations; it then invokes technology mapping and makes use of the mapping result to determine the likelihood of accepting the proposed transformation. To mitigate the runtime overhead, we further introduce parallelization techniques to decompose a large design into multiple smaller sub-netlists that can be optimized simultaneously. Experimental results show that our approach achieves promising area improvement over a set of commonly used benchmarks. Notably, PIMap reduces the LUT usage by up to 14% and 7% on average over the best-known records for the EPFL arithmetic benchmark suite.

1. Introduction

Modern FPGA designs rely on sophisticated CAD algorithms and tools to achieve high-quality solutions [4]. A very important step in this toolflow is called technology mapping, which transforms a gate-level Boolean logic network[1] into a functionally equivalent netlist composed of look-up tables (LUTs). Minimizing the depth and the total LUT count of the mapped netlist are two of the typical optimization goals for an FPGA-targeted technology mapper.

A key challenge to technology mapping is that the quality of the mapping solution depends heavily on the structure of the input logic network. It is well known that the problem of restructuring the network for depth- or area-optimal technology mapping is NP-hard [5]. Modern FPGA synthesis tools usually apply a series of structural optimizations to transform the input logic network to be more amicable for technology mapping and other downstream optimizations [8, 11]. Examples of the commonly used logic optimizations include balancing the levels of different paths in a logic network (i.e., balancing), and replacing a subnetwork with a smaller one that realizes the same function (i.e., rewriting). In practice, such logic optimizations are usually interleaved with each other and repeatedly applied to better optimize the logic network. While such transformations can effectively reduce the complexity of the logic network in terms of the gate count and/or the number of logic levels, we argue that there still exists considerable room in improving the FPGA mapping quality based on two important observations:

- The mainstream FPGA synthesis frameworks use a fixed predetermined sequence of pre-mapping logic transformations that may not always generate high-quality logic structures. For example, the popular academic tool ABC provides synthesis scripts with more than 20 different optimization sequences [3]. Since the efficacy of these sequences varies across different designs, it is very challenging for a user to quickly identify the best sequence to employ given a new design.
- Miscorrelations exist between the quality metrics driving the pre-mapping logic optimizations and those targeted by the actual technology mapping. Specifically, minimizing the gate count or the number of logic levels may not necessarily translate to reduced LUT count or depth in the final mapped netlist, thereby creating a gap between the optimality at the logic stage and the technology mapping stage.

We use Figure 1 to concretely illustrate the drawbacks of existing techniques. Consider the problem of using 3-input LUTs to map the logic network shown in Figure 1(a), which has four inputs (a-d) and four outputs (o_1-o_4). The original circuit can be implemented using four 3-input LUTs high-

[1] In the rest of paper, we use the term *logic network* to denote a pre-mapping gate-level Boolean logic network.

FPGA '17, February 22-24, 2017, Monterey, CA, USA
 ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021735

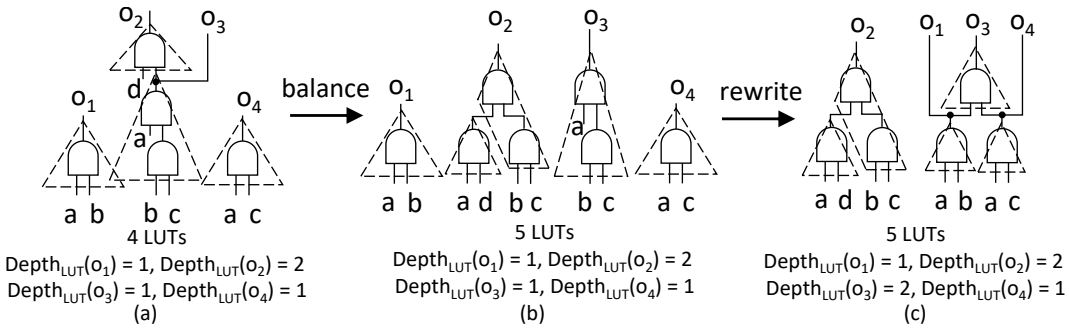

Figure 1. Logic optimizations and mapping on a simple example assuming 3-input LUTs — (a) The original circuit. (b) The circuit after applying `balance`. (c) The circuit after applying `rewrite` to the circuit from (b). Mapping results are indicated with dashed regions.

lighted in Figure 1(a). Suppose we apply two transformations step by step to the network. The first step performs balancing which manages to decrease the depth of the logic network by one as shown in Figure 1(b). The second step uses rewriting to reduce the gate count by one while maintaining the same depth as illustrated in Figure 1(c). While these transformations can successfully simplify the original network, the eventual mapping results are unfortunately worsened in terms of both LUT count and depth if we compare the mapped netlists shown in Figure 1(a) and Figure 1(c). Specifically, the netlist after balancing and rewriting requires one more LUT to map, and the depth of output o_3 also increases by one in the mapped netlist.

Clearly, reducing the depth and area of logic network does not necessarily translate to performance improvements or area savings after mapping. To address this challenge, we propose PIMap — a parallelized iterative improvement approach to area-driven LUT mapping. Unlike existing methods that decouple the logic transformations from technology mapping, PIMap makes use of the actual mapping results to guide a series of randomly proposed structural optimizations. Proposing logic transformations in a probabilistic way allows PIMap to explore a larger design space that cannot be uncovered by fixed optimization sequences. According to our experimental results, PIMap consistently outperforms the state-of-the-art LUT mapping solutions for unconstrained area optimization as well as delay-constrained area minimization.

Since iterative improvement usually comes with nontrivial runtime overhead, we further propose techniques to decompose a large netlist into multiple smaller sub-netlists, and optimize these sub-netlists in parallel across multiple machines. This parallelization framework enables PIMap to handle large circuits with more than 40 thousand LUTs, with a synthesis time in the range of tens to hundreds of seconds. In addition, PIMap also allows the users to easily explore the trade-offs between the design quality and the synthesis effort in runtime.

Our primary technical contributions are as follows:

- We provide a quantitative study on the (mis)correlation between the gate count reduction in the pre-mapping logic network and the LUT count savings after technology mapping.
- We propose a stochastic iterative improvement algorithm and associated parallelization techniques to enable efficient mapping-in-the-loop area optimization for LUT-based FPGAs.
- We demonstrate promising improvements in area reduction for a set of common benchmarks, including breaking many best-known records for the EPFL arithmetic benchmark suite.

The rest of the paper is organized as follows: Section 2 provides an overview of technology mapping and common logic transformations; Section 3 studies the correlation between the gate count in the logic network and the LUT count after mapping; Section 4 describes the key techniques in PIMap; Section 5 presents the experimental results; Section 6 reviews the related work, followed by conclusions in Section 7.

2. Preliminaries

In this section, we discuss the basics of technology mapping and common logic transformations used in PIMap.

2.1 Overview of Technology Mapping

Generally speaking, technology mappers are divided into structural mappers and functional mappers [12]. Structural mappers consider the input logic network as fixed, and attempt to cover the circuit with K-input LUTs. Functional mappers are allowed to modify the structure of the logic network before mapping to LUTs. In this work we focus on functional mappers for generating higher-quality mapping solutions.

Before covering the logic network with LUTs, functional mappers usually apply a sequence of logic transformations to the network, which we call *moves*. The goal of these moves is to prepare the network for technology mapping so that the subsequent LUT covering step can generate high-quality results in terms of LUT depth or LUT count. We defer the discussion on the details of logic transformation to Section 2.2, and first describe the mechanism of covering a logic network with LUTs.

During the LUT covering step in technology mapping, we view the logic network as a directed acyclic graph, where the nodes represent logic gates and the edges capture the connections between the gates. We define a cone C_v at node v as the sub-netlist of v and some of its predecessors so that any path from a node in C_v to v is entirely contained in C_v. A cone is said to be K-feasible if there are no more than K nodes outside C_v that have edges pointing to the nodes in C_v. A cut of C_v is defined to be the set of input nodes of C_v.

In LUT-based FPGAs, we can implement any K-feasible cones using a K-input LUT. Consequently, the mapping problem reduces to the problem of optimally covering the input graph with K-feasible cones [13]. A LUT covering framework generally consists of cut enumeration, cut ranking, cut selection, and final mapping generation. Cut enumeration explores all K-feasible cuts at each node, while cut ranking evaluates the quality of the cuts based on the optimization objective. Cut selection determines the optimal cut for each node based on the ranking information to generate the final covering solution.

2.2 Common Logic Transformations

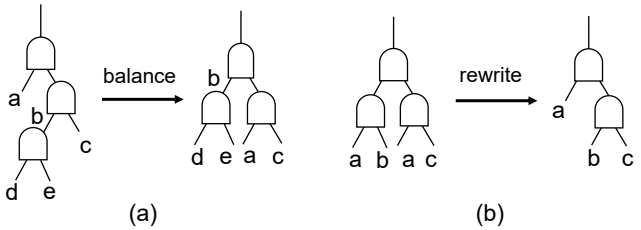

Figure 2. Illustration of two common logic transformations — (a) `balance`: balance the depth of the netlist using associative transform $a(bc) = (ab)c = (ac)b$. (b) `rewrite`: replace a sub-netlist with an equivalent but smaller one.

A logic transformation (or a move) applies optimization on the logic network in order to reduce the size or the number of levels of the network. Figure 2 shows two common logic transformations. The balancing transformation [10] tries to balance the depth of different paths in the netlist using associative transformations in the form as $a(bc) = (ab)c = (ac)b$. An associative transform at a given node is accepted if it reduces the depth of the corresponding node. In Figure 2(a), the balancing move swaps the left child of the output node with the branch that generates node b. As a result, the network is more balanced and the level of the output node is reduced by one.

A rewriting transformation [11] visits each node in the network in a topological order, and enumerates all K-feasible cuts of the subject node. The Boolean function of each cut is then computed and matched against all the equivalence classes of K-variable functions. After trying all the available circuit representations for the given node, the rewriting move picks the one with the largest improvement. Figure 2(b) provides an example of the rewriting transform, where a 3-input function is rewritten to a smaller structure shown on the right side. Refactoring is a variation of the rewriting move [10]. It uses a heuristic algorithm to compute a large cut for each node, and then tries to replace the cut with a factored form of the cut function. The transform is accepted if the replacement does not increase the size of the network.

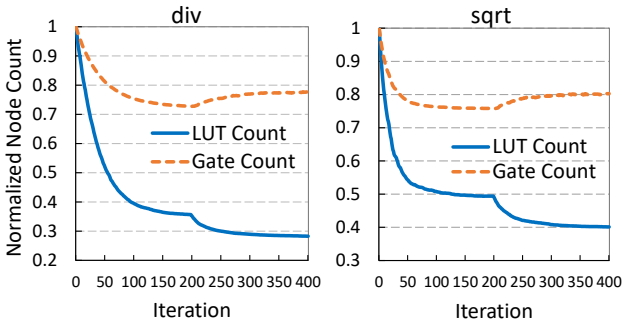

Figure 3. Correlation between gate count in the logic network and post-mapping LUT count — For the first 200 iterations, we perturb the logic network with the objective of reducing gate count. After 200 iterations, we change the objective to reducing LUT count.

3. Quantitative Study of Correlation between LUT Count and Gate Count

In this section we study the impact of commonly used logic transformations on the gate count in the logic network as well as the corresponding LUT count after technology mapping. Our experimental methodology is to iteratively perturb a given logic network (or a sub-network) to generate a sequence of equivalent design points with varying sizes in terms of gate count and LUT count. More specifically, we use two different strategies to perturb the logic network. The *gate-centric perturbation* enumerates a set of logic transformations to the input network, then greedily accepts the resulting logic networks that reduce the gate count. This way we iteratively generate a sequence of design points with decreasing number of gates, at the same time, the LUT count of each design point is also recorded. With the second strategy called LUT-centric perturbation, we also iteratively apply a set of candidate transformations to the logic network and measure the LUT count after each transformation. However, we only accept the transformations that reduce the LUT count of the resulting mapped netlist. We record both gate count and LUT count upon the acceptance of each transformation.

Here we evaluate two representative designs from the EPFL arithmetic benchmark suite [1], and use and-inverter graph (AIG) as the gate-level representation of the logic network. We use the aforementioned method to apply three transformations in the ABC logic synthesis framework [3] (balance, refactor, rewrite) to generate 400 intermediate design points for each benchmark. Notably, we employ the gate-centric perturbation for the first 200 iterations, and

switch to LUT-centric perturbation mode afterwards. Figure 3 shows the normalized LUT count and gate count during the 400 iterations of perturbations. During the initial phase of gate-centric perturbation, the decrease of LUT count coincides with the gate count reduction. Eventually, both descending curves level off, which seems to suggest that little room is left for improving area. Interestingly, when switching to LUT-centric perturbation after 200 iterations, we observe further reduction in LUT count with an increasing gate count. While we are only presenting two benchmarks here due to space limitation, we observe from our experiments very similar trends (to Figure 3) across a broad range of designs, which motivates us to propose PIMap that will be detailed in the next section.

4. PIMap Techniques

PIMap decomposes a large circuit netlist into smaller sub-netlists, and uses an iterative routine to minimize the area of these sub-netlists in parallel. The area minimization routine integrates commonly used logic transformations and technology mappers to progressively improve the design quality. In this section, we describe the PIMap techniques in detail and mainly focus on the unconstrained area optimization. We also show that PIMap can easily be extended to handle depth-constrained area minimization.

4.1 Iterative Area Minimization

The very core of PIMap is an iterative area minimization framework that repeats three major steps: (1) proposing logic transformation moves, (2) evaluating the quality of the move through technology mapping, and (3) determining whether to accept the proposed move. Figure 4 sketches the high-level design flow of this iterative procedure.

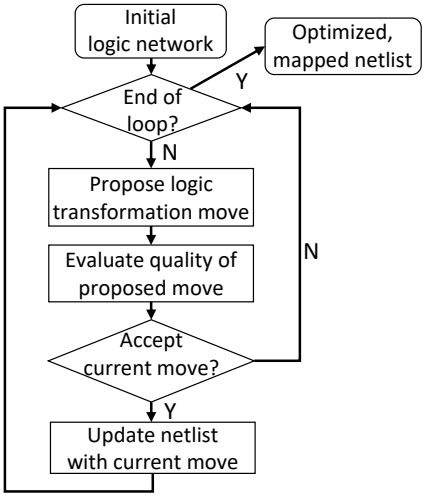

Figure 4. High-level flow chart of the iterative area minimization routine.

Proposing a Transformation Move PIMap makes use of a collection of logic transformation moves, denoted as set T. Each move in T is capable of optimizing a given logic network for a certain target, such as reducing the number of nodes in the circuit and balancing the node levels of different paths. We further associate T with a discrete probability distribution named P, where the probability of selecting the i^{th} at any iteration is denoted as p_i. At the beginning of each iteration, PIMap randomly chooses one logic transformation from T based on P. The transformed network is then evaluated by invoking an existing area-minimizing technology mapping algorithm.

Evaluating a Move In this step, the transformed netlist is first mapped to K-input LUTs using an existing area-oriented technology mapper. With unconstrained area optimization, we directly tie the quality metric Q of a proposed move to the number of LUTs in the mapped circuit netlist (denoted as N_{LUT}). We note that Q can be extended to include other user-specified factors such as the number of gates in the pre-mapping logic network.

Accepting a Move After obtaining the quality metric of the currently proposed move Q_{curr} and that of the previous iteration, denoted as Q_{prev}, we use the Markov Chain Monte Carlo (MCMC) method to probabilistically determine whether to accept the proposed move [6]. In particular, we employ the Metropolis-Hastings algorithm [7] for calculating the acceptance probability.

This process is detailed in Algorithm 1, which dictates that if the quality of the current move is better than the previous one, we accept the current move unconditionally. Otherwise, we accept the move with a small probability that decreases exponentially as Q_{curr} increases. Probabilistically accepting a move with inferior quality helps PIMap avoid quickly getting stuck in local minima during the search process.[2] Once a move is accepted, we update Q_{prev} to be Q_{curr}, save the updated network, and continue with a new proposal. On the other hand, if the current move is rejected, we do not update Q_{prev} and directly proceed to the next iteration. During the search procedure, we also keep track of the best mapping result and the corresponding circuit netlist. We return the best result at the end of the iterative area minimization routine.

In contrast to the previous methods that apply a fixed sequence of logic transformations, our randomized approach can effectively explore and search a large design space. Moreover, this search is guided by the actual mapping results instead of logic-level design metrics. This combination of a large number (tens or hundreds of iterations) of randomly proposed moves and the mapping-guided search is the key to achieving the superior mapping quality with PIMap.

[2] It is worth noting that MCMC and simulated annealing are closely related [9]. Compared to MCMC sampling, simulated annealing has one additional temperature term that decreases over time to control the likelihood of accepting an inferior move. In our experiments, we observe that the temperature term has almost no impact on the convergence rate, thus we decide to directly use the Metropolis-Hastings algorithm to compute the acceptance probability.

Algorithm 1 Calculating acceptance probability

if $Q_{curr} < Q_{prev}$ **then**
  Accept the current move
else
  // $rand()$: random number between 0 and 1
  if $rand() < e^{-\gamma(Q_{curr}/Q_{prev})}$ **then**
    Accept the current move
  else
    Reject the current move

4.2 Netlist Extraction and Parallel Optimization

To enable parallel optimization of multiple sub-netlists, PIMap automatically extracts a user-configurable number of non-overlapping sub-netlists from a mapped netlist, and optimize them in parallel through multithreading.

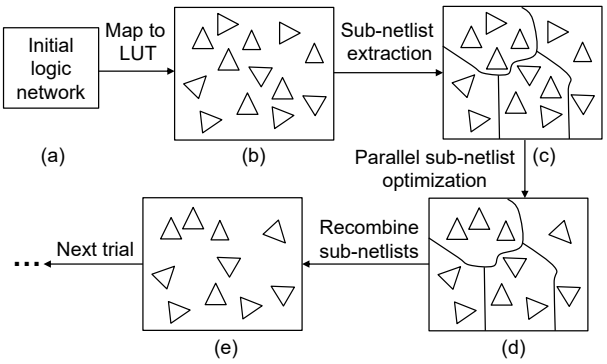

Figure 5. Illustration of netlist decomposition and parallel optimization — (a) Original logic network. (b) Netlist after LUT mapping, where each triangle represents a LUT, the connections between LUTs are omitted. (c) The four sub-netlists after sub-netlist extraction. (d) The four sub-netlists after optimization. (e) The netlist after recombining the four optimized sub-netlists.

Figure 5 conceptually illustrates the netlist extraction and parallel optimization steps. Given an input logic network, we first map it into a circuit netlist composed of LUTs shown as the triangles in Figure 5. We then partition the netlist into multiple sub-netlists, and apply the area minimization technique in Section 4.1 to optimize the sub-netlists in parallel. After optimizing the sub-netlists, we recombine them into a single netlist, and start the next trial of the sub-netlist extraction and optimization. We discuss these two steps in detail below.

Partitioning Mapped Netlists Algorithm 2 describes the steps required to partition a mapped netlist to enable effective parallelization. More specifically, the inputs to our partitioning algorithm include (1) a netlist that has already been mapped to LUTs, (2) a parallelization factor n, and (3) a size constraint M for each sub-netlist, the goal is to extract n non-overlapping sub-netlists with each of which containing no more than M LUTs. It is worth noting that partitioning the mapped netlist allows us to easily merge the optimized sub-netlists to regenerate the complete LUT netlist. More importantly, any improvement to a sub-netlist will directly contribute to the overall LUT savings in the recombined netlist.

Algorithm 2 Extracting sub-netlists

Input: A mapped netlist G_0, a parallelization factor n, and a sub-netlist size constraint M.
Output: n sub-netlists $\{G_1, G_2, ..., G_n\}$, each of which contains M LUTs.

// G_{res} is the residual graph of G_0
Initialize $G_{res} = G_0$, and $G_1 = G_2 = ... = G_n = \varnothing$
// extract the i^{th} sub-netlist
for i *from* 1 *to* n **do**
  Randomly pick a node j in G_{res}
  Start from j, visit G_{res} in breadth-first order:
  for *each node* k *during traversal* **do**
    Add node k to G_i
    Remove node k from G_{res}
    if *size of* G_i *reaches* M **then**
      break
  // Determine primary inputs (PI) and primary outputs (PO) of G_i
  for *each node* k *in* G_i **do**
    for *each fan-in* l *of* k **do**
      if l *is not assigned inside* G_i **then**
        Add l to the PI set of G_i
    if *(*k *is a PO of* G_{res}*) or (*k *is used in* G_{res}*)* **then**
      Add k to the PO set of G_i
return $\{G_1, G_2, ..., G_n\}$

When generating a sub-netlist, our algorithm first randomly picks a *seed*, and expands the sub-netlist using breadth-first search (BFS) from the seed until the number of LUTs in the sub-netlist reaches M. When constructing the sub-netlists, we also maintain a residual graph that contains the nodes not yet added to any sub-netlists. The residual graph is initialized to be the same as the original netlist, and will gradually decrease in size as more sub-netlists are extracted. After generating the first sub-netlist, the algorithm will pick another random seed, and extract the next sub-netlist from the residual graph until all the n sub-netlists have been generated. In case BFS cannot find a cluster of size M, the algorithm extracts another cluster and append it to the sub-netlist until the sub-netlist reaches a size of M LUTs. After the partitioning step, our algorithm assigns the primary inputs (PI) and primary outputs (PO) of each sub-netlist by identifying the nodes that have external fan-ins as well as those that fanout to external nodes.

Optimizing Sub-Netlists After obtaining the sub-netlists from the previous step, PIMap distributes them to available computing resources for independent optimization. We create one thread for each sub-netlist, and assign threads to machines to balance the load. Optionally, PIMap allows the user

to use multiple threads to optimize different copies of the same sub-netlist in parallel to increase the likelihood of generating a high-quality solution. After all threads finish execution, a master thread collects the optimized sub-netlists, and combine them to reform the entire design. This combining process involves concatenating all the sub-netlists into a single netlist, and remove the PIs and POs of each individual sub-netlists. Since all sub-netlists are of equal or very similar size, the runtime of different threads are similar to each other. Consequently, the workloads of different threads are highly balanced.

4.3 Overall Flow

We summarize the overall flow of PIMap using the techniques in Sections 4.1 and 4.2. Figure 6 shows the overall flow of PIMap.

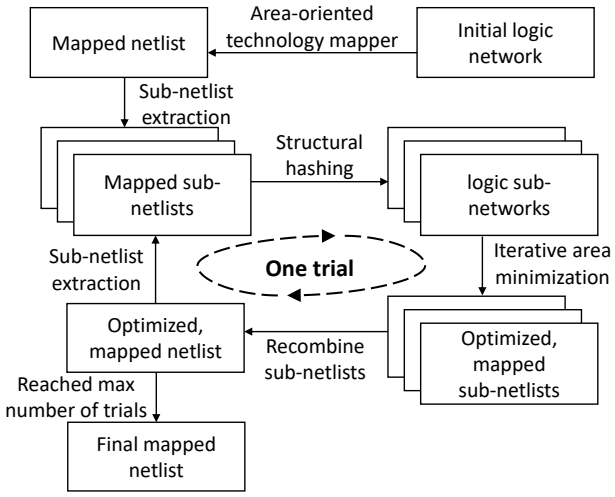

Figure 6. Overall synthesis flow of PIMap.

PIMap takes the initial logic network as the input, and first uses an area-oriented technology mapper to transform the logic network into a mapped netlist. PIMap then uses the sub-netlist extraction technique detailed in Section 4.2 to extract a number of sub-netlists. Since the iterative area minimization requires a gate-level logic network, we apply a netlist decomposition technique, such as structural hashing, to convert the mapped sub-netlist back to the corresponding logic sub-networks. These logic sub-networks are subsequently optimized using the iterative area minimization technique detailed in Section 4.1, which generates the optimized version of the mapped sub-netlists. PIMap then recombines these optimized netlists into a single netlist that is equivalent to the original design.

We define a trial as the four steps including sub-netlist extraction, converting a mapped netlist back to logic network, iterative area minimization, and sub-netlists recombination. At each trial, PIMap repartitions the mapped netlist into a different set of sub-netlists using the technique in Section 4.2 with randomly-selected seed node for each partition. In the rare case where the repartitioned sub-netlists are identical to the ones from the previous trial, we discard the current partition and repartition the netlist again. This repartition scheme is more effective than a static pre-partitioning, as it allows global cross-boundary optimization that uncovers more area saving opportunities.

In PIMap, the overall optimization flow contains a user-specified number of trials, and the overall flow terminates when it reaches the maximum number of trials or the runtime limit. A highly-optimized and mapped netlist is generated as the final result.

4.4 Extension to Depth-Constrained Area Minimization

PIMap can be extended to support depth-constrained area minimization with only a few modifications to the optimization flow. In depth-constrained area minimization, we assume that the input logic networks are already optimized for depth, and PIMap will try to reduce the area of such networks given that the depth does not increase. To handle depth-constrained area minimization, we modify the PIMap flow in the following aspects:

- We replace the area-oriented technology mapper with a depth-oriented technology mapper so that the depth constraint is likely to be met.
- During each trial, after recombining the optimized sub-netlists, we add a global depth-optimization step that re-maps the logic network using the depth-oriented technology mapper.
- After the global depth-optimization step, we reject the result of the current trial if the design exceeds the depth constraint or the area of the design increases compared to the previous trial. If a current trial is rejected, we reuse the netlist from the previous trial.

The above extensions ensure that the depth constraint is satisfied throughout the optimization flow, while the PIMap techniques are able to minimize the design area under the depth constraint.

5. Experimental Results

We implement the PIMap techniques in C, including the iterative search procedure, the netlist decomposition algorithm, and the routine to manage the parallel optimization across different machines. We integrate the logic optimization moves and the technology mapper in ABC [3] as a static library into PIMap. Throughout the experiment, we use ABC's native AIG as the gate-level representation. In our experiment, we use PIMap to refine designs that are already optimized by existing technology mappers (e.g., optimization scripts from ABC, or the best known results from the EPFL benchmark suite [1]). PIMap can also handle unoptimized designs, and generate final designs without any quality loss, but at the cost of slightly longer runtime.

Table 1. Area reduction using PIMap on the 10 largest MCNC combinational benchmarks — `Base` = the baseline designs synthesized using ABC's `compress2rs` script followed by an area-oriented technology mapper (command `if -a -K 6`); `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; `Time` = runtime in seconds; `Improv` = improvement in size between PIMap and the baseline designs.

	Base		5 Trials				10 Trials				40 Trials			
Designs	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
alu4	455	9	425	13	22.3	6.6%	405	15	42.9	11.0%	393	13	168.8	13.6%
apex2	526	12	493	15	22.2	6.3%	488	15	43.1	7.2%	439	17	177.4	16.5%
apex4	568	9	555	13	18.1	2.3%	541	13	38.3	4.8%	526	13	162.4	7.4%
des	631	9	544	8	31.9	13.8%	509	8	62.2	19.3%	477	8	253.0	24.4%
ex1010	606	9	589	11	18.8	2.8%	584	13	39.4	3.6%	556	15	158.5	8.3%
ex5p	332	10	324	11	16.3	2.4%	319	12	34.0	3.9%	304	12	136.9	8.4%
misex3	382	9	352	9	18.6	7.9%	333	10	36.3	12.8%	298	9	153.0	22.0%
pdc	1251	14	1219	19	31.8	2.6%	1200	22	66.6	4.1%	1150	19	266.5	8.1%
seq	627	10	606	12	22.1	3.3%	596	11	43.2	4.9%	567	12	177.0	9.6%
spla	1251	14	1222	18	32.5	2.3%	1191	18	63.8	4.8%	1133	25	250.8	9.4%
geomean						4.8%				7.4%				12.4%

Table 2. Area reduction using PIMap on the EPFL arithmetic benchmarks — `Base` = the best known results on EPFL benchmarks [1]; `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; `Time` = runtime in seconds; `Improv` = improvement in size between PIMap and the baseline designs.

	Base		5 Trials				10 Trials				40 Trials			
Designs	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
adder	201	73	196	68	19.2	2.5%	196	68	37.7	2.5%	194	66	150.5	3.5%
shifter	512	4	512	4	21.1	0.0%	512	4	41.1	0.0%	512	4	164.5	0.0%
divisor	3813	1542	3636	1490	53.1	4.6%	3527	1431	104.3	7.5%	3331	1277	418.1	12.6%
hyp	44635	4194	44095	4341	195.5	1.2%	43677	4431	394.9	2.1%	42164	4542	1604.3	5.5%
log2	7344	142	7036	133	60.9	4.2%	6904	129	119.8	6.0%	6749	119	491.5	8.1%
max	532	192	525	190	28.1	1.3%	525	190	57.6	1.3%	522	190	222.3	1.9%
mult	5681	120	5184	97	64.6	8.7%	5069	90	133.7	10.8%	4986	86	544.9	12.2%
sine	1347	62	1273	57	40.3	5.5%	1261	57	81.2	6.4%	1235	56	332.7	8.3%
sqrt	3286	1180	3246	1198	52.1	1.2%	3200	1188	103.8	2.6%	3127	1154	412.1	4.8%
square	3800	116	3380	77	94.1	11.1%	3346	77	184.8	11.9%	3281	74	730.3	13.7%
geomean						4.1%				5.2%				7.2%

In our experiment, the set of logic transformation techniques T consists of three elements: `balance`, `rewrite`, and `refactor`, with a uniform probability distribution $P = \{1/3, 1/3, 1/3\}$. We set $\gamma = 1$ in Algorithm 1. Throughout the experiment, we target mapping to 6-input LUTs. Of course, PIMap also supports other LUT architectures. For each design, we execute 40 trials, and each trial contains 100 iterations of mapping-guided logic optimization. For parallelization, we partition the original design to up to 16 sub-netlists, where each sub-netlist contains up to 100 LUTs. We run PIMap on up to eight machines, and each machine has a quad-core Xeon CPU operating at 2.66GHz.

We use two well-known benchmark suites to evaluate the effectiveness of PIMap: the 10 largest combinational benchmarks in the MCNC benchmark suite [15], as well as the entire EPFL arithmetic benchmark suite [1]. This collection of benchmarks contains a diverse set of designs ranging from common arithmetic units to realistic industrial designs. These designs also greatly differ in size.

5.1 Unconstrained Area Minimization

Table 1 shows the results of unconstrained area minimization for the 10 largest MCNC combinational benchmarks. For this set of benchmarks, we first apply ABC's `compress2rs` logic optimization script targeting area reduction. Based on our experiments with the available ABC synthesis scripts, `compress2rs` achieves the best area results for the majority of the designs. The optimized logic network is then mapped into 6-input LUTs using ABC's area-optimized mapper with command `if -a -K 6`. For PIMap, we record the size, depth, and runtime after 5, 10 and 40 trials. We also report the improvement of LUT counts in the PIMap-optimized designs over the baseline designs.

PIMap is able to reduce the LUT count by 4.8% on average after five trials, and 12.4% after 40 trials. For `des` and `misex3`, PIMap is able to reduce the size by more than 20%, showing the effectiveness of PIMap compared to ABC. The runtime of the 10 benchmarks are similar due to the similar sizes of the designs, averaging around 20 seconds for five trials, and 160 seconds for 40 trials. Although the runtime of PIMap is noticeably higher than existing mappers, which usually take less than a second for designs of similar sizes, we argue that PIMap is still valuable and viable in a high-effort FPGA implementation mode where technology mapping is unlikely the performance bottleneck.

Table 3. Area reduction under depth constraint using PIMap on the 10 largest MCNC combinational benchmarks — We use the depth of the baseline designs as the depth constraint. `Base` = the baseline designs synthesized using ABC's `resyn2` script followed by a depth-oriented technology mapper (command `if -K 6`); `n Trials` = result after `n` number of trials using PIMap; `Size` = size of the design in terms of number of 6-input LUTs; `Dpt` = depth of the design defined as the highest LUT level; `Time` = runtime in seconds; `Improv` = improvement in size between PIMap and the baseline designs.

	Base		5 Trials				10 Trials				40 Trials			
Designs	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
alu4	511	5	438	5	32.4	14.3%	438	5	68.0	14.3%	437	5	254.3	14.5%
apex2	674	6	511	6	31.3	24.2%	489	6	60.9	27.4%	469	6	250.2	30.4%
apex4	588	5	588	5	33.4	0.0%	588	5	63.9	0.0%	588	5	251.8	0.0%
des	818	5	651	5	50.3	20.4%	632	5	97.9	22.7%	584	5	395.4	28.6%
ex1010	655	5	654	5	30.5	0.2%	654	5	63.9	0.2%	652	5	258.2	0.5%
ex5p	351	5	351	5	25.4	0.0%	351	5	51.0	0.0%	351	5	202.7	0.0%
misex3	443	5	318	5	32.8	28.2%	314	5	65.0	29.1%	306	5	239.4	30.9%
pdc	1431	7	1430	7	55.0	0.1%	1430	7	107.4	0.1%	1427	7	441.0	0.3%
seq	693	5	590	5	33.3	14.9%	588	5	66.9	15.2%	588	5	282.9	15.2%
spla	1392	7	1387	7	61.9	0.4%	1361	7	128.0	2.2%	1361	7	479.7	2.2%
geomean						8.7%				9.4%				10.7%

We further apply PIMap to the EPFL arithmetic benchmark suite, and compare our results with the best known mapping records from the EPFL database that are publicly available [1]. Table 2 shows the comparison between PIMap and the existing best known results (used as baseline). PIMap is able to improve nine out of the 10 best known mapping results, with an average improvement of 7.2%. Notably, PIMap reduces the LUT count for `divisor`, `mult`, and `square` by more than 12%. In addition, PIMap improves the depth in eight out of the 10 designs even though it is not intended for depth optimization in this particular use case. We conjecture that existing area-oriented mappers that generate the best known min-area solutions have to make an unnecessary compromise in depth to gain additional area savings. We also note that even for the largest design `hyp` that has more than 44 thousand LUTs, the PIMap runtime remains reasonable, owing to the fact that we optimize multiple small sub-netlists in parallel instead of directly optimizing the entire design.

5.2 Depth-Constrained Area Minimization

Table 3 shows the result of depth-constrained area minimization on MCNC benchmarks. For the baseline designs, we first apply ABC's depth-minimizing `resyn2` script, then map the optimized logic network into 6-input LUTs using ABC's depth-oriented mapper with command `if -K 6`. We use the depth of the mapped baseline designs as the depth constraint for PIMap, and invoke PIMap to reduce the area of the baseline designs. PIMap is able to reduce the area for eight out of the 10 designs, with an average improvement of 10.7%, while preserving the depth from the baseline designs. It is noteworthy that PIMap achieves 30% area reduction for `apex2`, `des`, and `misex3`, showing that PIMap is still highly effective under a hard constraint on depth.

5.3 Scalability of Parallel Optimization

Figure 7 shows the scalability of PIMap. In our experiment, we partition an input netlist to up to 16 sub-netlists, each of which contains up to 100 LUTs. We select four large benchmarks in the EPFL benchmark suite, and study the runtime required to achieve a fixed area target. The area target of each benchmark is set to be the area of the PIMap-optimized design using one sub-netlist and 100 trials. In this experiment, we use four parallel threads to optimize one sub-netlist, which requires up to 64 threads in total for the 16 sub-netlists.

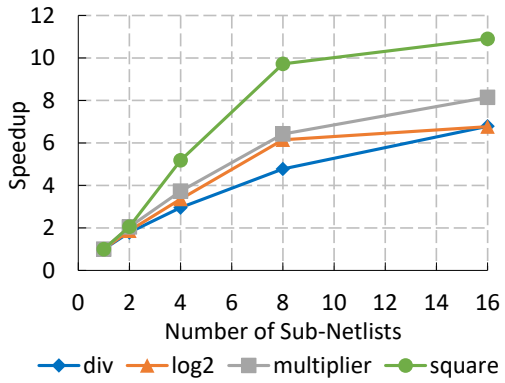

Figure 7. Scalability of the parallel optimization technique — We use PIMap for area reduction to test the scalability of parallelization. We measure the runtime to achieve a specific area target and plot the speedup in runtime versus number of sub-netlist partitions.

As shown Figure 7, PIMap scales reasonably well up to 16 sub-netlist partitions across multiple designs. In particular, PIMap scales near-linearly up to eight sub-netlists. With more sub-netlists, the overhead of netlist decomposition and reassembly becomes nontrivial and prevents PIMap from achieving the ideal speedup.

5.4 Runtime Breakdown of PIMap

Figure 8 shows the relative runtime of the four main steps in PIMap. Sub-netlist generation refers to the step of decomposing the original netlist into sub-netlists. The logic transformation step first proposes a transformation move, then applies the selected move to the network. The LUT mapping

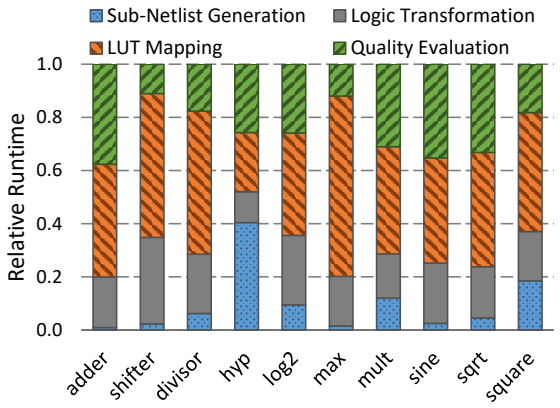

Figure 8. Runtime breakdown of PIMap.

step converts the logic networks into LUTs using ABC's built-in technology mapper named `if`. The quality evaluation step calculates the quality of the proposed transformation and decides whether to accept the proposed move.

Not surprisingly, the LUT mapping consumes the largest portion of the runtime, followed by quality evaluation and logic transformation. These three steps together dominate the runtime since they need to be iteratively invoked for many times in each trial (100 in our experiment). The runtime of the sub-netlist generation step is negligible for most of the benchmarks since the BFS-based extraction algorithm scales linearly as the size of the netlist. For `hyp`, the runtime of sub-netlist generation is noticeably higher than the other designs since it is significantly larger than other designs. Nevertheless, the runtime of sub-netlist generation for `hyp` is still on the same order of the other steps.

5.5 Impact of Sub-Netlist Size on PIMap Runtime

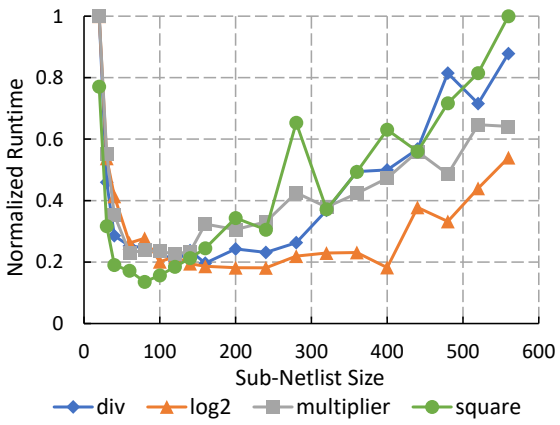

Figure 9. Impact of sub-netlist size on PIMap runtime.

Figure 9 shows the impact of the sub-netlist size on the PIMap runtime to achieve a fixed area target, defined as the area after 100 trials using a sub-netlist size of 20 LUTs. We partition the designs up to 16 sub-netlists. For smaller designs that do not admit 16 sub-netlists, we partition them into as many sub-netlists as feasible. The runtime in Figure 9 is normalized to the longest runtime of the corresponding design.

We observe that across the four benchmarks, the runtime inflection point is around the size of 100 LUTs. With smaller sub-netlists, each PIMap optimization thread runs faster, but overall progress may be slow since each sub-netlist only covers a small fraction of the entire design.

5.6 Area Reduction under a Tight Runtime Limit

Table 4. Area reduction using PIMap with 10 second runtime limit — `Base` = the best known results on EPFL benchmarks [1]; `PIMap` = solution of PIMap after 10 seconds. We highlight the designs that are improved by PIMap.

	Base		PIMap	
Designs	Size	Dpt	Size	Dpt
adder	201	73	**197**	69
shifter	512	4	512	4
divisor	3813	1542	**3787**	1536
hyp	44635	4194	44635	4194
log2	7344	142	**7305**	144
max	532	192	**526**	190
mult	5681	120	**5594**	118
sine	1347	62	**1309**	62
sqrt	3286	1180	**3279**	1181
square	3800	116	**3675**	102

Table 4 shows the performance of PIMap under a tight runtime limit, which is set to be 10 seconds. In this case, PIMap achieves less area savings but still manages to improve the best-known mapping results in eight out of the 10 EPFL benchmarks.

5.7 LUT Count vs. Gate Count Reduction

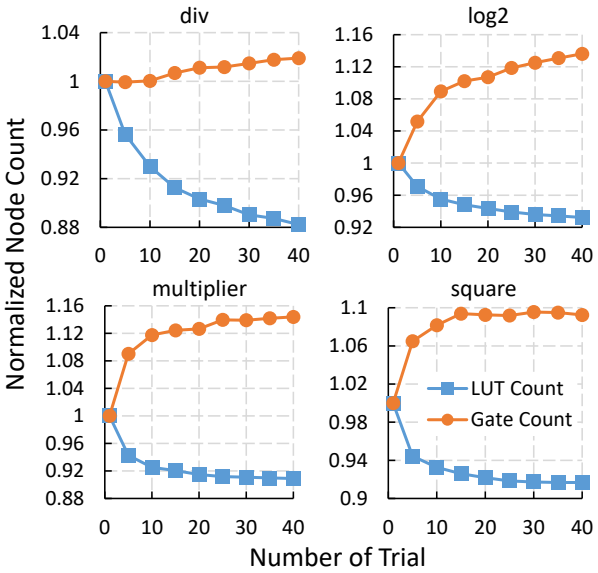

Figure 10. Relation between LUT count and AIG gate count at various design points of the same design.

Figure 10 shows the LUT count and the corresponding gate count in the AIG of the same design during the opti-

mization process in PIMap, normalized to their initial values. For the four benchmarks, the LUT count decreases as the number of trials increases. However, we observe an opposite trend in gate count during the optimization, which agrees with our correlation study in Section 3.

6. Related Work

Mishchenko, et al. [11] describe a number of efficient rewriting techniques on AIGs, which serve as the basis for the logic transformations used in this work. The majority-inverter graph (MIG) proposed by Amarú, et al. [2] provides an alternative logic representation using three-input majority nodes and regular/complemented edges. MIG is shown to be beneficial for improving mapping quality in a number of cases. This is complementary to PIMap, since our iterative improvement framework is agnostic to logic representations.

Yang, et al. [16] propose a new way of logic synthesis by maintaining a precomputed library of optimal or near-optimal circuits for small practical functions. Their logic synthesis flow matches and replaces small circuit components in a new design to the elements in the precomputed library. However, this approach can only find optimal or near-optimal solution for small functions with no more than 12 inputs, and become sub-optimal for functions with more inputs. PIMap is orthogonal to [16] and it is not limited by the input size of the sub-netlist. It is also possible to incorporate Boolean matching techniques as new transformation moves in our iterative improvement framework.

STOKE [14] uses stochastic search to optimize x86 programs by randomly rewriting the x86 assembly instructions. Both STOKE and our approach randomly propose transformations using MCMC sampling to explore a large design space. Besides the different application domains, our work differs from STOKE in two major aspects: (1) STOKE focuses on using local moves that modify a single instruction at a time, while we make use of the logic rewriting techniques applied to multiple nodes in the network; (2) STOKE can only handle small programs with around one hundred instructions. In contrast, PIMap makes use of parallel optimization to effectively handle much larger circuits with tens of thousands LUTs.

7. Conclusions

We propose PIMap, a parallelized iterative improvement framework for area-oriented FPGA technology mapping. PIMap iteratively proposes logic transformation moves to optimize an input logic network for LUT mapping, and uses the actual mapping result to evaluate the quality of a proposed move. To improve the runtime, PIMap decomposes a large circuit netlist into multiple smaller sub-netlists, and optimizes them in parallel across different machines. Experimental results demonstrate significantly improvement in mapping quality for both unconstrained area optimization and depth-constrained area optimization compared to the state-of-the-art technology mappers. As a future direction, we plan to investigate global restructuring techniques on the logic network to further improve the quality of PIMap.

8. Acknowledgements

This work was supported in part by NSF Awards #1337240, #1453378, #1512937, a DARPA Young Faculty Award D15AP00096, and a research gift from Xilinx, Inc.

References

[1] L. Amarú, P.-E. Gaillardon, and G. De Micheli. The EPFL Combinational Benchmark Suite. *International Workshop on Logic & Synthesis (IWLS)*, 2015.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(5):806–819, 2016.

[3] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 60413. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[4] D. Chen, J. Cong, and P. Pan. FPGA Design Automation: A Survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[5] A. H. Farrahi and M. Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11), 1994.

[6] C. J. Geyer. Practical Markov Chain Monte Carlo. *Statistical Science*, pages 473–483, 1992.

[7] W. K. Hastings. Monte Carlo Sampling Methods using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[8] Y. Hu, V. Shih, R. Majumdar, and L. He. FPGA Area Reduction by Multi-Output Function Based Sequential Resynthesis. *Design Automation Conf. (DAC)*, 2008.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simmulated Annealing. *Science*, 220:671–680, 1983.

[10] A. Mishchenko and R. Brayton. Scalable Logic Synthesis using a Simple Circuit Structure. *International Workshop on Logic & Synthesis (IWLS)*, pages 15–22, 2006.

[11] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. *Design Automation Conf. (DAC)*, 2006.

[12] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):240–253, 2007.

[13] P. Pan, A. K. Karandikar, and C. Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 17(6):489–498, 1998.

[14] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[15] S. Yang. *Logic Synthesis and Optimization Benchmarks*. Microelectronics Center of North Carolina (MCNC), 1991.

[16] W. Yang, L. Wang, and A. Mishchenko. Lazy Man's Logic Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2012.