PIMap: A Flexible Framework for Improving LUT-Based Technology Mapping via Parallelized Iterative Optimization

GAI LIU and ZHIRU ZHANG, School of Electrical and Computer Engineering, Cornell University, USA

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 article, we propose PIMap, which couples logic transformations and technology mapping under an iterative improvement framework 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. By adjusting the optimization objective and incorporating required time constraints during the iterative process, PIMap can flexibly optimize for different objectives including area minimization, delay optimization, and delay-constrained area reduction. 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 PIMap achieves promising quality improvement over a set of commonly used benchmarks, including improving the majority of the best-known area and delay records for the EPFL benchmark suite.

CCS Concepts: • **Hardware** → **Logic synthesis**; **Software tools for EDA**;

Additional Key Words and Phrases: Technology mapping, lookup table

ACM Reference format:

Gai Liu and Zhiru Zhang. 2019. PIMap: A Flexible Framework for Improving LUT-Based Technology Mapping via Parallelized Iterative Optimization. *ACM Trans. Reconfigurable Technol. Syst.* 11, 4, Article 23 (January 2019), 23 pages.

<https://doi.org/10.1145/3268344>

1 INTRODUCTION

Modern FPGA designs rely on sophisticated CAD algorithms and tools to achieve high-quality solutions [7]. A very important step in this toolflow is called technology mapping, which transforms a gate-level Boolean logic network¹ 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.

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

Authors' addresses: G. Liu, 471 Rhodes Hall, Ithaca, NY 14853; Z. Zhang, 320 Rhodes Hall, Ithaca, NY 14853.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1936-7406/2019/01-ART23 \$15.00

<https://doi.org/10.1145/3268344>

Fig. 1. Logic optimizations and mapping on a simple example assuming three-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.

A key challenge to technology mapping is that the quality of the mapping solution depends heavily on the structure of the input logic network. However, it is well known that the problem of restructuring the network for depth- or area-optimal technology mapping is NP-hard [11]. Modern FPGA synthesis tools usually apply a series of structural optimizations as a heuristic to transform the input logic network to be more amicable for technology mapping and other downstream optimizations [17, 24]. Examples of the commonly used logic optimizations include balancing the levels of different paths in a logic network (i.e., balancing), and replacing a sub-network 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 [5]. 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.
- Quality improvements from logic-level optimizations do not always lead to better quality of results after 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 three-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 three-input LUTs highlighted in Figure 1(a). Suppose we apply two transformations step by step to the network. The first step performs balancing that 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 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 area and delay optimizations.

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 40,000 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 tradeoffs between the design quality and the synthesis effort in runtime.

A preliminary version of this work was presented in Reference [21], which focused on area optimization. This work extends the previous version by introducing new techniques to minimize delay during the iterative logic transformation routine and extract delay-critical sub-netlists for effective delay minimization. In addition, we propose to adjust the area recovery effort dynamically during the optimization process that achieves delay-optimized designs with reduced area. These extensions enable PIMap to flexibly optimize for various objectives such as area minimization, delay optimization, and delay-constrained area reduction. For sequential circuits, we integrate PIMap with the retiming technique to achieve even better solution quality for delay optimization.

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, demonstrating the drawbacks of the mainstream logic synthesis techniques.
- 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 devise a network repartitioning scheme that enables logic optimization across sub-netlist boundaries, which leads to significant improvements in the solution quality.
- We extend the area minimization flow to delay optimization by proposing techniques to extract delay-critical sub-netlists and minimize the output arrival times.
- We show promising improvements in area reduction for a set of common benchmarks, including breaking many best-known records for the EPFL arithmetic benchmark suite.
- We demonstrate the effectiveness of the delay optimization flow for a set of common combinational and sequential benchmark circuits, including improving 13 of the 20 records in the EPFL benchmark suite.

The rest of the the article is organized as follows: Section 2 reviews the related work; Section 3 provides an overview of technology mapping and common logic transformations; Section 4 studies the correlation between the gate count in the logic network and the LUT count after mapping; Section 5 describes the key techniques in PIMap for area optimization; Section 6 extends the PIMap framework for delay optimization; Section 7 presents the experimental results, followed by conclusions in Section 8.

2 RELATED WORK

Heuristic Logic Rewriting. Rewriting uses greedy algorithms to minimize the size or depth of a Boolean logic by iteratively selecting subgraphs and replacing them with their equivalent implementations. Examples of rewriting techniques include refactoring for size reduction [23], and algebraic balancing for tree height reduction [9]. Mishchenko et al. describe improvements to a number of efficient rewriting techniques on and-inverter graphs, which serve as the basis for the logic transformations used in this work [24].

Exact Synthesis. Exact synthesis is a new way of improving the quality of logic synthesis by maintaining a precomputed library of optimal or near-optimal circuits for small-size functions. The exact synthesis flow matches and replaces small circuit components in a new design to the elements in the precomputed library, which has been shown to be effective for both area and delay optimization [4, 31, 32, 36]. Alternatively, the optimal circuits can be computed during runtime if their optimal implementations are not found in the library. Due to the scalability limit of SAT solvers and the exponentially growing library size, exact synthesis is limited to relatively small subgraphs with no more than 12 inputs for online computation, or four to six inputs for precomputed libraries. Schmitt et al. propose a similar approach for area recovery in structural technology mapping, which optimizes a sequence of relatively small overlapping regions in a mapped network using a SAT solver [28]. PIMap is orthogonal to the exact synthesis approach 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.

Alternative Logic Representation. The majority-inverter graph (MIG) proposed by Amarú et al. [1, 3] 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.

Randomized Synthesis Optimization. Optimization using randomized algorithms has been explored in different domains. The simulated annealing method, which improves the optimization objective using repeated local changes, has been successfully demonstrated in problems such as VLSI packing [33], placement [6], and routing [29]. Genetic programming has been applied in logic synthesis by repeatedly mutating the circuit structure, where a SAT solver is used to verify the equivalence between the reference circuit and the circuit after mutations [14, 34]. Fišer and Schmidt propose to apply iterative resynthesis on parts of the circuit [12, 13]. At each iteration, the proposed method extracts a sub-netlist from the original circuit, and applies a fixed sequence of logic transformations to optimize the sub-netlist. Compared to this method, PIMap stochastically accepts randomly proposed transformations, and uses parallelization to speed up the optimization process. STOKE [27] 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.

Parallelized Technology Mapping. There have been extensive research efforts on parallelizing FPGA technology mapping using multicore processors and graphics processing units [8, 18, 20, 30], where a key technique is to partition a large netlist into multiple sub-netlists and assign them to different threads for parallel mapping. Up to 3× speedup has been shown by parallelizing a sequential version of the technology mapping with negligible overheads in solution quality. A major difference in PIMap is that we couple parallel technology mapping with iterative logic transformations to improve the solution quality. In addition, PIMap repartitions the overall netlist repeatedly during the optimization process, which is crucial for achieving superior mapping quality, since it

Fig. 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.

eliminates the drawback of fixed partition boundaries and uncovers different optimization opportunities in each repartition of the netlist.

3 PRELIMINARIES

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

3.1 Overview of Technology Mapping

Generally speaking, technology mappers are divided into structural mappers and functional mappers [25]. 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 3.2 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 [26]. 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.

3.2 Common Logic Transformations

A logic transformation (or a move) applies optimization on the logic network to reduce the size or the number of levels of the network. Figure 2 shows two common logic transformations. The balancing transformation [22] 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 [24] 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 three-input function is rewritten to a smaller structure shown on the right side. Refactoring is a variation of the rewriting move [22]. 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.

4 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 (e.g., the number of AND gates in an and-inverter graph) in the logic network as well as the corresponding LUT count (i.e., the number of LUTs) 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, where each design point is a gate-level logic network that is functionally equivalent to the initial design but with different network structures. At the same time, the LUT count of each design point after mapping to six-input LUTs 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 [2], 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 [5] (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 for area comparisons 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.

5 PIMAP FRAMEWORK FOR AREA OPTIMIZATION

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

Fig. 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.

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

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.

5.1 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.

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 [15]. In particular, we employ the Metropolis-Hastings algorithm [16] 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. However, 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.

ALGORITHM 1: Calculating acceptance probability

```

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

```

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.

5.2 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.

---

²It is worth noting that MCMC and simulated annealing are closely related [19]. 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.

Fig. 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 5.1 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.

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 combines them to reform the entire design.

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 $\{G_1, G_2, \dots, G_n\}$, each of which contains M LUTs.

// G_{res} is the residual graph of G_0

Initialize $G_{res} = G_0$, and $G_1 = G_2 = \dots = G_n = \emptyset$

// 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

end

end

    // 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

end

end

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

end

end

end

return $\{G_1, G_2, \dots, G_n\}$

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.

5.3 Overall Flow

We summarize the overall flow of PIMap using the techniques in Sections 5.1 and 5.2. Figure 6 shows the overall 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 5.2 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 5.1, 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,

Fig. 6. Overall synthesis flow of PIMap.

PIMap repartitions the mapped netlist into a different set of sub-netlists using the technique in Section 5.2 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.

6 EXTENSION TO DELAY OPTIMIZATION

We extend the PIMap framework detailed in Section 5 to delay optimization with the goals of (1) minimizing the delay of the circuit after technology mapping and (2) reducing the area of the mapped circuit when possible without increasing the delay. We discuss two techniques to achieve these goals. First, we augment the iterative area minimization routine (Section 5.1) by considering the input arrival time information and output arrival time constraints to enable delay optimization of the sub-netlists. Second, we enhance the netlist extraction algorithm (Section 5.2) to extract delay-critical regions of the overall design for effective delay optimization.

6.1 Iterative Logic Transformation under Required Time Constraint

Given a sub-netlist with arrival times for each of the inputs and the required times for each of the outputs, the iterative logic transformation routine aims to find an optimized sub-netlist implementation that (1) satisfies the output required times, (2) minimizes the arrival time of the outputs, and (3) reduces the circuit area after technology mapping when possible. Figure 7 illustrates the flow for iterative logic transformation for delay optimization, where we highlight the steps that differ from the flow in Figure 4.

Computing Input Arrival Time and Output Required Time. Given an extracted sub-netlist in LUTs, we first annotate the inputs of the sub-netlist with their arrival times by examining the original circuit from which the sub-netlist was extracted. This is carried out by topologically traversing the original circuit from the primary inputs and computing the accumulated delay of the input

Fig. 7. High-level flow chart of the iterative logic transformation routine for depth optimization. We highlight the steps that differ from the iterative area minimization flow.

Fig. 8. Illustrative example of calculating the arrival and required times of the nodes in a mapped netlist. AT = arrival time; RT = required time. Each triangle represents a LUT. Here we assume that each LUT has a delay of 1, and the arrival times of the PIs and the required times of the POs are given.

nodes to the sub-netlist. Similarly, to compute the required time for the outputs of the sub-netlist, we traverse the original circuit in reverse topological order. During the traversal, we annotate the output nodes of the sub-netlist with their required times. The required time of a node n , defined as the maximum delay that does not affect the overall delay of the original circuit, can be computed during the reverse topological traversal by examining the set of fanout nodes of node n , denoted as $\text{fanout}(n)$.

The required time $RT(n)$ of the current node n is then computed as $RT(n) = \min_{i \in \text{fanout}(n)} RT(i) - \text{delay}(\text{LUT})$. Figure 8 shows an example of calculating the arrival and required times of the nodes in a netlist, where we highlight the calculation for the required time of node c .

Optimizing Sub-Netlists under Required Time Constraint. After annotating the sub-netlist with the arrival time and required time information, we iteratively transform the sub-netlist by

extending the flow in Section 5.1. At each iteration, the proposed transformation heuristically optimizes the sub-netlist. The size and the output levels of the transformed circuit are then measured after technology mapping. If any of the outputs in the transformed sub-netlist has an arrival time exceeding its required time, then the proposed move is immediately rejected. Otherwise, we compute the sum of the output arrival times in the transformed sub-netlist, which will be used to determine whether to accept the transformation. The quality of the proposed transformation move Q_{curr} is computed as $Q_{curr} = \alpha \Sigma_{ArrivalTime} + \beta N_{LUT}$, which is a weighted sum of total output arrival times ($\Sigma_{ArrivalTime}$) and the circuit size after technology mapping (N_{LUT}). The weights α and β are hyperparameters that trade off delay optimization effort and area recovery effort. We use Algorithm 1 for deciding whether to accept the proposed transformation using the quality of the proposed transformation Q_{curr} and that of the previous iteration (i.e., Q_{prev}). We note that our approach can potentially be extended to enable multi-objective design space exploration, where we construct a Pareto-optimal frontier in delay and area. However in this work, we focus on delay minimization and only recover area after an optimized delay is achieved. We use the technique in Section 6.2 to dynamically adjust the area recovery effort, which ensures that the final solution maintains an optimized delay during the optimization process.

6.2 Dynamic Adjustment of Area Recovery Effort

The values of the hyperparameters α and β in the iterative logic transformation routine controls the tradeoff between optimization efforts for delay minimization and area reduction. In one extreme case where $\alpha = 0$, PIMap reduces the circuit area while satisfying the required time constraint, which is equivalent to solving the delay-constrained area recovery problem. For the other extreme scenario where $\beta = 0$, PIMap tries to minimize the circuit delay without any consideration on the size of the generated circuits.

In PIMap, we adjust the values of α and β at runtime to achieve an optimized strategy for the design at hand. We implement a simple mechanism to automatically adjust the efforts for delay optimization versus area recovery by tuning the values of α and β . Specifically, we start with $\beta = 0$ to minimize circuit delay in the first trial. Every time when the current trial does not improve the circuit delay, we gradually increase the value of β so that PIMap will emphasize on reducing the size of the current circuit. We observe that this strategy achieves the goal of prioritizing delay minimization while reducing the circuit area after reaching an optimized delay value.

6.3 Delay-Critical Netlist Extraction

For delay optimization, PIMap extends the netlist extraction technique in Section 5.2 to focus the optimization efforts on promising sub-netlists that are more likely for delay improvements. To extract delay-critical sub-netlists, PIMap first computes the timing slack for every node in the initial mapped circuit. Starting from the primary output with the smallest timing slack, where ties are broken randomly, PIMap traverses the netlist towards the primary inputs while adding the visited nodes only when its timing slack is no greater than a critical value. In practice, a critical value of three LUT levels works well for generating delay-critical sub-netlists. After extracting the first sub-netlist, PIMap starts over and extracts the next sub-netlist from the remaining nodes in the original netlist. Figure 9 uses an example to depict the process of extracting delay-critical netlists with a critical value of one.

We note that the delay-critical netlist extraction algorithm will unlikely extract the same set of sub-netlists in different trials mainly due to the following reasons: (1) the randomized tie-break rule for selecting the starting primary output nodes tends to choose different subsets of primary outputs, and (2) the netlist structure changes non-trivially after each trial, so the extracted sub-netlists will differ even when starting from the same primary output.

Fig. 9. Illustrative example of the delay-critical netlist extraction process. We extract two sub-netlists with a critical value of one. Each triangle represents a LUT, the number next to each LUT indicates its slack value. The extracted sub-netlists are highlighted with colors.

7 EXPERIMENTAL RESULTS

We implement the PIMap techniques in C, including the iterative search procedure, the netlist extraction algorithm, and the routine to manage the parallel optimization across different machines. We integrate the logic optimization moves and the technology mapper in ABC [5] 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 [2]). PIMap can also handle unoptimized designs, but at the cost of slightly longer runtime.

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 six-input LUTs. Of course, PIMap also supports other LUT architectures. For each design, unless otherwise stated, 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 designs in the MCNC benchmark suite [35], as well as the EPFL benchmark suite [2]. 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.

7.1 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 six-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 20s for five trials, and 160s for 40 trials. Although

Table 1. Area Reduction Using PIMap on the 10 Largest MCNC Combinational Benchmarks

Designs	Base		5 Trials				40 Trials			
	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
alu4	455	9	425	13	22.3	6.6%	393	13	168.8	13.6%
apex2	526	12	493	15	22.2	6.3%	439	17	177.4	16.5%
apex4	568	9	555	13	18.1	2.3%	526	13	162.4	7.4%
des	631	9	544	8	31.9	13.8%	477	8	253.0	24.4%
ex1010	606	9	589	11	18.8	2.8%	556	15	158.5	8.3%
ex5p	332	10	324	11	16.3	2.4%	304	12	136.9	8.4%
misex3	382	9	352	9	18.6	7.9%	298	9	153.0	22.0%
pdcc	1251	14	1219	19	31.8	2.6%	1150	19	266.5	8.1%
seq	627	10	606	12	22.1	3.3%	567	12	177.0	9.6%
spla	1251	14	1222	18	32.5	2.3%	1133	25	250.8	9.4%
geomean						4.8%				12.4%

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 six-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.

Table 2. Area Reduction Using PIMap on the EPFL Arithmetic Benchmarks

Designs	Base		5 Trials				40 Trials			
	Size	Dpt	Size	Dpt	Time	Improv	Size	Dpt	Time	Improv
adder	201	73	196	68	19.2	2.5%	194	66	150.5	3.5%
shifter	512	4	512	4	21.1	0.0%	512	4	164.5	0.0%
divisor	3813	1542	3636	1490	53.1	4.6%	3331	1277	418.1	12.6%
hyp	44635	4194	44095	4341	195.5	1.2%	42164	4542	1604.3	5.5%
log2	7344	142	7036	133	60.9	4.2%	6749	119	491.5	8.1%
max	532	192	525	190	28.1	1.3%	522	190	222.3	1.9%
mult	5681	120	5184	97	64.6	8.7%	4986	86	544.9	12.2%
sine	1347	62	1273	57	40.3	5.5%	1235	56	332.7	8.3%
sqrt	3286	1180	3246	1198	52.1	1.2%	3127	1154	412.1	4.8%
square	3800	116	3380	77	94.1	11.1%	3281	74	730.3	13.7%
geomean						4.1%				7.2%

Base = the best-known results on EPFL benchmarks [2]; n Trials = result after n number of trials using PIMap; Size = size of the design in terms of number of six-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.

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.

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 [2]. Table 2 shows the comparison between PIMap and the existing best-known results (used as baseline). PIMap is able to improve 9 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 8 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

Fig. 10. 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.

depth to gain additional area savings. We also note that even for the largest design hyp that has more than 44,000 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.

7.2 Scalability of Parallel Optimization

Figure 10 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.

As shown Figure 10, 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.

7.3 Runtime Breakdown of PIMap

Figure 11 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 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.

Fig. 11. Runtime breakdown of PIMap.

Fig. 12. Impact of sub-netlist size on PIMap runtime.

7.4 Impact of Sub-Netlist Size on PIMap Runtime and Quality

Figure 12 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 12 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.

The size of the sub-netlists also influences the quality of the PIMap optimization. We run PIMap targeting area minimization on two benchmarks (div and sqrt) using sub-netlist sizes ranging from 5 to 3000 LUTs. We partition the design 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. Figure 13 shows the reduction of LUTs as a function of the number of trials. We observe that a sub-netlist size of 100 to 200 LUTs leads to the best results. For small sub-netlists with only 5 to 20 LUTs, the results are suboptimal due to the limited optimization opportunity constrained by the small sub-netlists. However, an overly large sub-netlist results in very similar network structures across trials, which also reduces the optimization opportunity between trials.

Fig. 13. Impact of sub-netlist size on PIMap quality. We run PIMap on the same design with varying sub-netlist sizes ranging from 5 to 3000 LUTs. The legend indicates different sub-netlist sizes.

Fig. 14. PIMap convergence experiments over independent runs. For each design, we conduct 10 runs of identical experiments to investigate the effects of randomness on the solution quality.

7.5 Convergence Experiment over Independent Runs

Randomized selection of logic transformations and the randomized repartitioning technique are the two sources of randomness during the PIMap optimization process. In Figure 14, we study the effects of the randomness on the solution quality. For each design, we conduct 10 identical runs of experiment and measure the reduction in LUT count. For each run, we partition the netlist into eight sub-netlists, where each sub-netlist contains 100 LUTs. We observe that the independent experiments achieve similar quality of results despite the randomness in the optimization process. For each design, the maximum difference in the solution quality is within 1% across the 10 runs. We also observe that the final netlists after the optimization are similar in structure, showing that PIMap empirically converges to an optimized final solution regardless of the randomness during the optimization process.

7.6 Area Reduction under a Tight Runtime Limit

Table 3 shows the performance of PIMap under a tight runtime limit, which is set to be 10s. In this case, PIMap achieves less area savings but still manages to improve the best-known mapping results in 8 of the 10 EPFL benchmarks.

Table 3. Area Reduction Using PIMap with 10 Second Runtime Limit

Designs	Base		PIMap	
	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

Base = the best-known results on EPFL benchmarks [2]; PIMap = solution of PIMap after 10s; Size = size of the design in terms of number of six-input LUTs; Dpt = depth of the design defined as the highest LUT level. We highlight the designs that are improved by PIMap.

Fig. 15. Relation between LUT count and AIG gate count at various design points of the same design.

7.7 LUT Count vs. Gate Count Reduction

Figure 15 shows the LUT count and the corresponding gate count in the AIG of the same design during the optimization 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 4.

7.8 Delay Optimization for Combinational Circuits

Table 4 shows the results of applying PIMap delay optimization to the 10 largest MCNC combinational benchmarks [35]. The baseline designs are generated using ABC's delay-optimizing script resyn2 and mapped to 6-LUTs using delay-oriented technology mapper (if -K 6). PIMap is able to reduce the maximum LUT depth by one in three designs (ex5p, pdc, and spla). Out of the remaining seven designs where PIMap does not improve the maximum depth, we observe that

Table 4. Delay Optimization Using PIMap on the 10 Largest MCNC Combinational Benchmarks

Designs	PI/PO	Base			5 Trials				40 Trials			
		Size	Max Dpt	Avg Dpt	Size	Max Dpt	Avg Dpt	Time	Size	Max Dpt	Avg Dpt	Time
alu4	14/8	511	5	4.50	492	5	4.38	10.2	461	5	4.38	79.9
apex2	39/3	674	6	6.00	647	6	6.00	10.4	575	6	6.00	78.6
apex4	9/19	588	5	4.68	575	5	4.68	11.0	575	5	4.68	84.1
des	256/245	818	5	3.23	818	5	3.23	12.1	818	5	3.23	98.8
ex1010	10/10	655	5	5.00	655	5	5.00	11.2	655	5	5.00	89.9
ex5p	8/63	351	5	3.67	404	4	3.03	11.0	404	4	2.98	81.5
misex3	14/14	443	5	5.00	413	5	4.93	10.2	387	5	4.93	77.9
pdc	16/40	1431	7	5.93	1713	6	5.05	11.6	1713	6	5.05	89.2
seq	41/35	693	5	4.49	669	5	4.40	10.0	669	5	4.34	80.8
spla	16/46	1392	7	5.63	1599	6	4.83	11.6	1599	6	4.78	88.9

PI/PO = number of primary inputs and primary outputs in the design; Base = the baseline designs synthesized using ABC's delay-optimizing resyn2 script followed by an delay-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 six-input LUTs; Max Dpt = depth of the design defined as the highest LUT level; Avg Dpt = average depth of the design defined as the average LUT level over all primary outputs of the design; Time = runtime in seconds. We highlight the designs whose maximum or average depth is improved by PIMap.

PIMap can improve the average depth of the designs in three cases (alu4, misex3, and seq), where the average depth is measured as the number of LUT levels averaged over all the primary outputs in the design. Reducing the average depth of a design can potentially benefit the downstream CAD flow for generating faster circuit after physical design. As mentioned in Section 6.1, while the delay algorithm prioritizes depth reduction, the impact of logic transformations on area is also considered during the optimization flow. Consequently, for the seven designs that PIMap does not improve depth, PIMap reduces the size of the circuits in five designs. The size of the remaining two designs (des and ex1010) stay unchanged.

We further apply PIMap to the latest depth records (version 2017.1) of the EPFL benchmark suite [2] to further improve the quality of the existing circuits. These depth records are used as the starting point for PIMap optimization. For each design, we execute PIMap for 100 trials. The runtime per trial for each design is almost identical to the per-trial runtime shown in Table 2, since the additional steps for depth optimization require negligible runtime. The results are shown in Table 5. According to the rule from the EPFL benchmark competition³, PIMap improves the existing best records in 13 of the 20 designs by reducing the circuit area while maintaining the same depth of the previous records. Noticeably, PIMap achieves significant area reduction while maintaining the best-known depths (e.g., 7.8%, 7.0%, and 6.3% area reductions for cavlc, router, and sine, respectively). In addition, two designs in Table 5 (mem_ctr1 and decoder) also achieve the best-known area while maintaining the best-known depth. The design router optimized by PIMap is larger than its best-area counterpart by only one LUT (53 LUTs vs. 52 LUTs), but improves the depth of the best-area version by two (four levels vs. six levels).

7.9 Delay Optimization for Sequential Circuit with Retiming

The optimization techniques in PIMap can be naturally extended to handle sequential circuits by treating the inputs and outputs of the registers as primary outputs and primary inputs of the

³The EPFL delay records rank designs by their depth after mapping to six-input LUTs. Designs with the same depth are further ranked by their sizes (measured as the number of six-input LUTs).

Table 5. Comparison with Depth Records on EPFL Benchmarks

Designs	Best Known Record		PIMap	
	Size	Dpt	Size	Dpt
adder	511	5	511	5
shifter	512	4	512	4
divisor	47964	230	47964	230
hyp	146302	573	145236	573
log2	9218	55	9218	55
max	882	10	882	10
mult	8215	28	8105	28
sine	1801	30	1690	30
sqrt	11680	254	11391	254
square	4038	11	3992	11
arbiter	2884	5	2871	5
alu ctrl	29	2	27	2
cavlc	115	4	106	4
decoder	272	2	270	2
i2c controller	244	3	244	3
int2float	41	3	40	3
mem ctrl	2490	7	2486	7
priority	157	4	157	4
router	57	4	53	4
voter	1469	12	1423	12

Best Known Record = the best-known delay results on EPFL benchmarks [2]; PIMap = solution of PIMap after 100 trials; Size = size of the design in terms of number of six-input LUTs; Dpt = depth of the design defined as the highest LUT level. We highlight the designs that are improved by PIMap.

combinational blocks, respectively. Table 6 shows the PIMap delay optimization results on 10 large MCNC sequential circuits that combine the retiming technique to further improve the delay. Retiming is a common sequential circuit optimization technique that adjusts the locations of the registers to optimize the overall circuit delay [10]. We observe that the logic transformations in PIMap generally not only optimize the combinational blocks between register boundaries, but also present optimization opportunities for retiming to achieve better quality. In Table 6, the baseline designs are generated using ABC's resyn2 script combined with delay-optimizing technology mapper if -K 6. For comparison, we also measure the quality of the circuits after applying PIMap without retiming, and after applying retiming on the baseline designs. Both versions of the PIMap optimized designs are generated with 10 trials. For the case of PIMap with retiming, we apply ABC's retime command at the end of each trial. Without applying retiming, PIMap is able to improve the depths of three designs when compared to the baseline circuits. With retiming enabled, PIMap improves the depth of the designs in 5 of the 10 designs when compared to the baseline with retiming, demonstrating the effectiveness of PIMap delay optimization on sequential circuits.

8 CONCLUSIONS

We propose PIMap, a parallelized iterative improvement framework for FPGA technology mapping. PIMap iteratively proposes logic transformation moves to optimize an input logic network

Table 6. Depth Reduction for Sequential Circuits with Retiming on 10 Large MCNC Sequential Benchmarks

Designs	Base			PIMap			Base + Retime			PIMap + Retime		
	Reg	Size	Dpt	Reg	Size	Dpt	Reg	Size	Dpt	Reg	Size	Dpt
b14	245	1272	13	245	1295	11	448	1461	8	502	1496	8
bigkey	224	579	3	224	575	3	672	1094	2	672	1094	2
clma	33	2965	9	33	2879	9	41	3011	9	42	2887	8
frisc	886	1776	14	886	1774	9	1276	2199	7	1288	2257	6
fsm8_16_13	8	386	5	8	386	5	8	390	5	8	477	4
mm30a	90	266	25	90	273	25	90	266	25	90	340	18
s38417	1462	2535	7	1462	2535	7	1462	2547	7	1462	2497	7
s38584.1	1260	2322	6	1260	2279	6	1267	2394	5	1267	2361	5
s9234.1	135	289	5	135	293	5	189	331	4	186	326	4
sort	136	387	14	136	387	13	146	410	13	146	448	12

Base = the baseline designs synthesized using ABC's resyn2 script followed by a depth-oriented technology mapper (command `if -K 6`); PIMap = solution of PIMap after 10 trials without applying retiming; Base + Retime = results after applying retiming on Base; PIMap + Retime = solution of PIMap after 10 trials, where retiming is applied after each trial; Reg = number of registers in the design; Size = size of the design in terms of number of six-input LUTs; Dpt = depth of the design defined as the highest LUT level; We highlight the designs whose depth is improved by PIMap.

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 significant improvements in mapping quality for both area and delay optimizations compared to the state-of-the-art technology mappers.

REFERENCES

- [1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2014. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *Proceedings of the Design Automation Conference (DAC'14)*.
- [2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *Proceedings of the International Workshop on Logic & Synthesis (IWLS'15)*.
- [3] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2016. Majority-inverter graph: A new paradigm for logic optimization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 5 (2016), 806–819.
- [4] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Pierre-Emmanuel Gaillardon, Janet Olson, Robert Brayton, and Giovanni De Micheli. 2017. Enabling exact delay synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'17)*.
- [5] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, Release 60413. 2017. Retrieved from <http://www.eecs.berkeley.edu/~alanmi/abc/>.
- [6] Andrea Casotto, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. 1987. A parallel simulated annealing algorithm for the placement of macro-cells. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 6, 5 (1987), 838–847.
- [7] Deming Chen, Jason Cong, and Peichen Pan. 2006. FPGA design automation: A survey. *Found. Trends Electron. Des. Autom.* 1, 3 (2006), 139–169.
- [8] Doris Chen and Deshanand Singh. 2010. Parallelizing FPGA technology mapping using graphics processing units (GPUs). In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*.
- [9] Jordi Cortadella. 2003. Timing-driven logic bi-decomposition. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 22, 6 (2003), 675–685.
- [10] Giovanni De Micheli. 1991. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 10, 1 (1991), 63–73.
- [11] Amir H. Farrahi and Majid Sarrafzadeh. 1994. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 13, 11 (1994).

- [12] Petr Fišer and Jan Schmidt. 2010. It is better to run iterative resynthesis on parts of the circuit. In *Proceedings of the International Workshop on Logic & Synthesis (IWLS'10)*.
- [13] Petr Fišer and Jan Schmidt. 2012. Improving the iterative power of resynthesis. In *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'12)*.
- [14] Petr Fišer, Jan Schmidt, Zdeněk Vašíč, Lukáš Sekanina, et al. 2010. On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In *Proceedings of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 346–351.
- [15] Charles J. Geyer. 1992. Practical Markov chain Monte Carlo. *Statist. Sci.* 7, 4 (1992), 473–483.
- [16] W. Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (1970), 97–109.
- [17] Yu Hu, Victor Shih, Rupak Majumdar, and Lei He. 2008. FPGA area reduction by multi-output function based sequential resynthesis. In *Proceedings of the Design Automation Conference (DAC'08)*.
- [18] Andrew Kennings and Chirag Ravishankar. 2011. Parallel FPGA technology mapping using multi-core architectures. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'11)*.
- [19] Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.
- [20] Laurent Lemarchand. 1999. Parallel performance directed technology mapping for FPGA. In *Proceedings of the Southwest Symposium on Mixed-Signal Design*.
- [21] Gai Liu and Zhiru Zhang. 2017. A parallelized iterative improvement approach to area optimization for LUT-based technology mapping. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.
- [22] A. Mishchenko and R. K. Brayton. 2006. Scalable logic synthesis using a simple circuit structure. In *Proceedings of the International Workshop on Logic & Synthesis (IWLS'06)*. 15–22.
- [23] Alan Mishchenko, Robert Brayton, and Satrajit Chatterjee. 2008. Boolean factoring and decomposition of logic networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'08)*.
- [24] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proceedings of the Design Automation Conference (DAC'06)*.
- [25] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2007. Improvements to technology mapping for LUT-based FPGAs. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 26, 2 (2007), 240–253.
- [26] Peichen Pan, Arvind K. Karandikar, and C. L. Liu. 1998. Optimal clock period clustering for sequential circuits with retiming. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 17, 6 (1998), 489–498.
- [27] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.
- [28] Bruno Schmitt, Alan Mishchenko, and Robert Brayton. 2018. SAT-based area recovery in structural technology mapping. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'18)*.
- [29] Carl Sechen. 1988. Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing. In *Proceedings of the Design Automation Conference (DAC'88)*.
- [30] Chuyu Shen, Zili Lin, Ping Fan, Xianglong Meng, and Weikang Qian. 2016. Parallelizing FPGA technology mapping through partitioning. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'16)*.
- [31] Mathias Soeken, Luca Gaetano Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. Exact synthesis of majority-inverter graphs and its applications. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 36, 11 (2017), 1842–1855.
- [32] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. 2017. Busy man's synthesis: Combinational delay optimization with SAT. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'17)*.
- [33] Simon Szykman and Jonathan Cagan. 1995. A simulated annealing-based approach to three-dimensional component packing. *J. Mech. Des.* 117, 2A (1995), 308–314.
- [34] Zdenek Vasicek and Lukas Sekanina. 2011. A global postsynthesis optimization method for combinational circuits. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'11)*.
- [35] Saeyang Yang. 1991. *Logic Synthesis and Optimization Benchmarks*. Microelectronics Center of North Carolina (MCNC).
- [36] Wenlong Yang, Lingli Wang, and Alan Mishchenko. 2012. Lazy man's logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'13)*.

Received January 2018; revised July 2018; accepted August 2018