DACPara: A Divide-and-Conquer Parallel Approach for High-Quality Logic Rewriting in Large-Scale Circuits

Nanjiang Qu
ICTT and ISN Laboratory, Xidian University, Xi 'an, China
njqu@stu.xidian.edu.cn

Cong Tian
ICTT and ISN Laboratory, Xidian University, Xi 'an, China
ctian@mail.xidian.edu.cn

Zhenhua Duan
ICTT and ISN Laboratory, Xidian University, Xi 'an, China
zhhduan@mail.xidian.edu.cn

ABSTRACT

Logic rewriting is a critical and time-consuming task in logic synthesis, which determines the area and delay of the synthesized circuit. However, existing parallel solutions for this task suffer from limitations in terms of runtime or quality in large-scale complex circuits. In this paper, we propose a divide-and-conquer parallel approach namely DACPara for high-quality logic rewriting in large-scale circuits. Specifically, after nodes in AIG are divided in each level, dynamic global information is considered to divide and conquer rewriting into three stages for parallel processing. Experiments show that DACPara using 40 CPU physical cores can be 34.36x and 1.96x faster than logic rewriting in ABC and the state-of-the-art CPU parallel method on large benchmarks, respectively, with extremely comparable quality of result. Also, for large-scale complex benchmarks, compared with state-of-the-art GPU accelerated method ours can achieve 1.1% quality improvement.

CCS CONCEPTS

• **Hardware → Circuit optimization**.

KEYWORDS

Logic synthesis, logic rewriting, AIG rewriting, parallel computing, large-scale circuits

ACM Reference Format:
Nanjiang Qu, Cong Tian, and Zhenhua Duan. 2024. DACPara: A Divide-and-Conquer Parallel Approach for High-Quality Logic Rewriting in Large-Scale Circuits. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655678

1 INTRODUCTION

As a key technology for multi-level logic optimization, logic rewriting plays a critical role in synthesis, with the goal of minimizing circuit area under certain delay constraints through equivalent logic reorganization [1]. According to Moore's Law, future generations of integrated circuit will contain trillions of logic gates as the level of integration continues to increase. Currently, even if fast serial logic rewriting methods are applied, the time required to synthesize a large-scale AIG with more than 10 million nodes has become no longer acceptable. In addition, logic rewriting techniques are often applied many times for optimization due to its local optimality. Therefore, it is essential to utilize multi-core CPU or GPU to study parallel solutions for synthesis of large-scale circuits.

In recent years, researchers have proposed many logic rewriting methods applied to different technology-independent logic representations, such as the AND-Inverter Graph (AIG) [2, 3], Majority-Inverter Graph (MIG) [4, 5], and Xor-Majority Graph (XMG) [6]. Among them, the method based on AIG is widely studied because of its flexible logic structure. The earliest work on it was proposed in [2] and has been integrated into ABC [7], a famous logic synthesis tool in the academic community.

As VLSI designs gradually increases in size and complexity, the runtime of existing serial rewriting algorithms is no longer acceptable. The work in [8] proposed a state-of-the-art CPU parallel rewriting algorithm, which achieves inter-node parallelism by using exclusive locks on all related nodes involved in the rewriting process of AIG node. However, this method suffered from terrible speedup for large-scale complex circuits due to the large number of conflicts. This is because it retains exclusive locks after cut enumeration until the entire process of rewriting is completed. In fact, the evaluation stage, which accounts for more than 90% of the rewriting runtime, does not conduct actual modifications. Therefore, maintaining these exclusive locks throughout the entire rewriting process will significantly limit the parallelism of this stage, thus greatly affecting the speedup. Furthermore, a state-of-the-art GPU-accelerated parallel logic rewriting method was proposed in [9]. It explores inter-node parallelism and intra-node one to the greatest extent, resulting in a high speedup. However, in order to completely eliminate exclusive locks for parallelism, this method replaces all subgraphs based on static global information without considering logical sharing, and then merges logical equivalent nodes to obtain gains. Inevitably, the replacement of some subgraphs may no longer have a gain, or even a negative gain, due to the modifications of the AIG graph structure when other subgraphs are replaced. In addition, a large amount of information needs to be written to GPU memory during the enumeration and evaluation stages. It is worth mentioning that with the continuous improvement in design complexity, there are as many as 616,126 NPN equivalence classes for 5-input cuts, and the astonishing $2 * 10^{14}$ ones for 6-input cuts. The limitations in GPU memory and single-core computing power will cause this method to face new challenges.

To solve the above problems, we propose a divide-and-conquer CPU parallel approach (namely DACPara) for high-quality logic rewriting in large-scale circuits. DACPara is based on the AIG rewriting method proposed by Mishchenko et al. in [2] and the parallel one that relies on operator formulas and Galois system proposed by Possani et al. in [8]. First, we divide and conquer the nodes in AIG, specifically partitioning and processing them one by one according to the level, i.e. the depth of the node. Afterwords, the steps of logic rewriting also adopt the divide-and-conquer strategy. The algorithms in the three stages of enumeration, evaluation and replacement are respectively reformulated and redesigned by

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655678

applying dynamic global information. The main contributions of this work are summarized as follows:

- We propose a new divide-and-conquer parallel rewriting approach called DACPara, which releases parallelism in rewriting based on dynamic global information.
- We implemented DACPara using the Galois system that is suitable for parallel computing of irregular algorithms. It can be an average of 34x faster than the AIG rewriting method in ABC at basically the same quality.
- Experimental results on the EPFL benchmarks show that, our approach achieves an average speedup of 1.96x with comparable quality compared to the state-of-the-art CPU parallel method. Specially, for all large-scale complex benchmarks in the MtM set, our approach can achieve 1.1% quality improvement compared to state-of-the-art GPU parallel method.

2 PRELIMINARIES

2.1 Related Definitions

And-Inverter Graph (AIG) is a directed acyclic graph (DAG) used to represent circuits. As a widely used data structure, AIG usually contains four types of nodes: primary input (PI), primary output (PO), constant, and two-input AND (AND2) gate. Inverter is represented by the complemented edges in the graph.

A cut c of node n is a set of nodes of the network, called $leaves$, such that each path from PIs to n passes through at least one leaf [2]. The $cut\ function$ is the boolean function of node n in terms of cut $leaves$, which is defined as the logic cone of the node n. The $cover$ of a cut c is the set of all nodes that appear on a path from any leaf to n including n, but excluding the $leaves$.

A maximum fanout free cone (MFFC) of a given AIG node n is a subset S of the $cover$ of the cut c for n such that every path from any node in S to a PO passes through n [8]. If node n is removed, each node in its MFFC can also be removed without affecting the function of the remaining circuit structure.

A Boolean function f_1 is NPN-equivalent to another f_2 if one of them can be obtained from the other by applying negations/permutations of its inputs/output [10]. In the same NPN class, logic cones represented by different Boolean functions are logically equivalent but structurally different.

The fanins (fanouts) of AIG node n is the set of nodes connected to its inputs (outputs). The transitive fanin (fanout) of AIG node n is the set of nodes n_i such that there is a path connecting $n_i(n)$ to $n(n_i)$ [9].

In the context of AIGs, structural hashing is a technique adopted to ensure that there are no AND2 gates with the same pair of fanins [10]. Before a new node is created, it must be determined that there is no existing node having the same fanins with the corresponding inverters in the hash table.

2.2 Galois System

Galois is a system that provides a data-centric programming model to exploit amorphous data parallelism in irregular graph algorithms [11]. The logic rewriting of each AIG node will continuously modify a small portion of the overall graph, which is a typical irregular algorithm. Therefore, the problem can be efficiently processed in parallel using the Galois system. Among them, the graph elements ensure mutual exclusion through exclusive locks that have been acquired while the thread is modifying the graph. In the overlapping area, if an activity processed by one thread needs to acquire a lock that has been acquired by an activity processed by another thread, the latter will abort due to conflict and wait for rescheduling. At this point, any locks that the aborted activity has acquired are released and all computations it has performed are lost. Therefore, it is desirable to design cautious operator to avoid waste of computing resources.

3 RELATED WORK

The original DAG-aware AIG rewriting method [2] was proposed by Mishchenko et al. to replace the logic cone in AIG with a logically equivalent but better structure. As far as we know, almost all of the recent logic rewriting studies are based on the this research. The overall procedure of this method is as follows. For each node executed in topological order, the algorithm computes 4-input cuts and the associated Boolean functions. Note that all the 65536 4-input Boolean functions are divided into 222 NPN equivalence classes, each of which contains equivalent structures stored in the NPN-structural hash table (NST) determined by this class. Subsequently the corresponding precomputed structures are retrieved from the NST to evaluate the current cut. Specifically, the evaluated gain is calculated in terms of the number of nodes that will be removed and added to the AIG while taking into account logical sharing. Thereafter, the structure that results in the best improvement is selected for replacement.

After the above work, [5] proposed a rewriting method of Majority-Inverter Graph (MIG) consisting of three-input majority gates and inverters. By exploring the relationship between majority and threshold logic functions, Augusto et al. [12] proposed a novel majority gate logic synthesis process that extends three inputs to have an arbitrary number of inputs. Compared with AIG, MIG can effectively improve the quality of arithmetic-intensive circuits by implementing the ternary majority function as a logical operation. In addition, Winston et al. [6] introduced the XOR Majority Graph (XMG) by adding XOR operators to the MIG proposed in [5]. In this structure, exact synthesis is performed only on the actual occurring functions to avoid enumerating the Boolean space. Compared with the first two logic representations, XMG is more compact due to its expressiveness, which also allows to find the best replacement faster in exact synthesis of small subnetworks. Afterwards, Riener et al. [13] proposed a resynthesis method for general logic representations and showed that various logic synthesis algorithms can work across logic representations. Therefore, the parallel rewriting algorithm we studied is scalable and can be continuously explored for novel solutions by combining the best characteristics of existing methods.

Several works on parallelizing logic synthesis algorithms using GPU or multi-core CPU have been proposed. Elbayoumi et al. [14] separated the cut enumeration stage from synthesis and improved some performance by completely parallelizing this stage. Liu [15] et al. achieved parallelism by decomposing a large design into multiple smaller subnets that can be optimized simultaneously. Since then, to further enhance parallelism, Possani et al. [8] proposed fine-grained parallel AIG rewriting that relies on operator formulas and Galois system, which is particularly suitable for exploring parallelism in irregular algorithms such as logic rewriting. However, for large-scale complex benchmarks, this method suffers from poor acceleration due to a large number of conflicts. [16] proposed a node-level parallel rewriting method based on GPU acceleration, which applies the original AIG to enumerate and evaluate all nodes only once in parallel and then perform serial conditional replacement on

the CPU. Among them, the application of static global information may lead to a decrease in rewriting quality. Also, applying the same GPU platform, Li et al. [9] proposed a logic rewriting framework that exploited inter-node and intra-node parallelism and achieved a high speedup. Actually, the essence of replacement that without considering logical sharing in this method is still the replacement based on static global information, which inevitably affects the quality of rewriting.

4 DACPARA

4.1 Overview

The workflow of the proposed parallel rewriting method DACPara is shown in Fig. 1. Before rewriting, the AIG nodes are divided according to their level, i.e. depth. Note that the levels of nodes increase sequentially based on their fanout relationships starting from the PI (level 0). In this process, all nodes in the AIG are pushed into the list array corresponding to the level of the node. On this basis, we perform parallel rewriting on the nodes in each list in order from small to large subscripts of the list array.

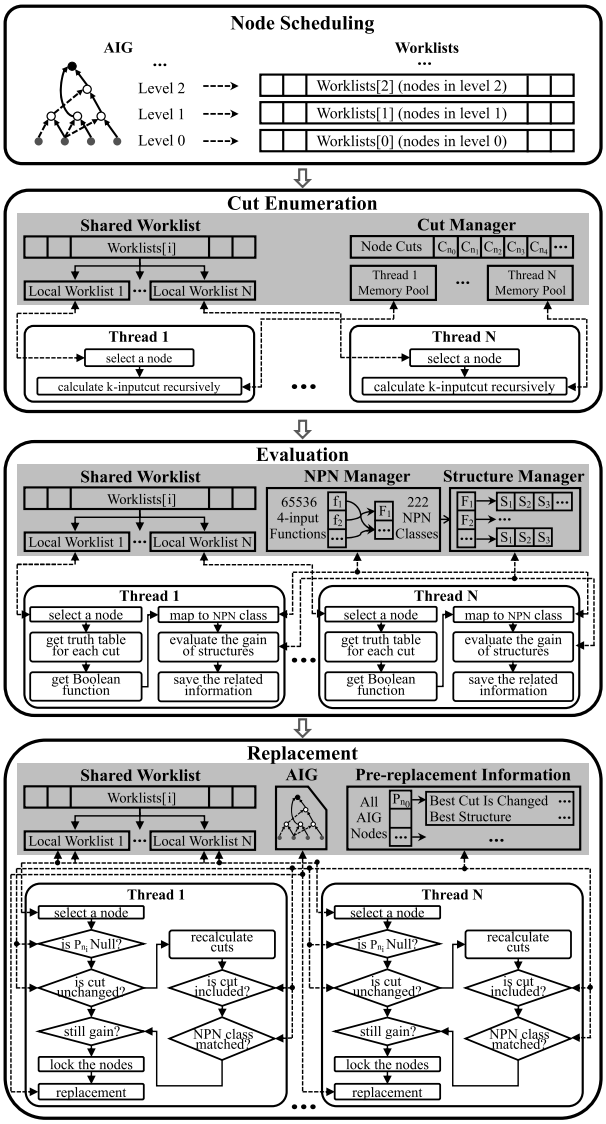

Figure 1: Workflow of DACPara.

The *Cut Manager*, *NPN Manager* and *Structure Manager*, which are necessary for the DAG-Aware AIG rewriting mentioned in Section 3, and the related data structures required by the new operators are initialized in line 2 of Algorithm 1. Among them, the container *prepInfo* with the same capacity as AIG is used to store and retrieve the pre-replacement information of all nodes by matching the subscript with the *ID* of the node. Subsequently, as shown in lines 4-8 of Algorithm 1, *Worklists*[*i*] in *Worklists* are processed in parallel in three stages, the specific details of which will be introduced in Sections 4.2, 4.3 and 4.4 respectively. The most critical point is that the parallel rewriting based on dynamic global information in this paper requires that each replacement must obtain a positive gain on the latest AIG. This issue will be discussed in Section 4.4.

Algorithm 1 : Divide-and-Conquer Parallel Logic Rewriting

Input: AIG
Output: rewritten AIG
1: Function DACPara()
2: Initialize *Worklists*[*AIG.level*], *prepInfo*[*AIG.area*], *cutMan* *npnMan*, *strMan*
3: *Worklists* ← nodeDividing(AIG)
4: **foreach** *Worklists*[*i*] **in** *Worklists* **do** ▷ Galois Parallel
5: paraCutEnuOperator(*Worklists*[*i*], *cutMan*)
6: paraEvaOperator(*Worklists*[*i*], AIG, *npnMan*, *strMan*)
7: paraRepOperator(*Worklists*[*i*], AIG, *prepInfo*)
8: **end foreach**

4.2 Parallel Cut Enumeration

Since the cut of any node originates from the PI, the nodes in the first list *Worklists*[*0*] to be processed must be all PIs with level 0 and the cut themselves. The cuts for the remaining nodes can be enumerated recursively according to their fanin.

When a node in the list *Worklists*[*i*] is rewritten, its level may change due to optimization, which causes the level of its fanout in the next list *Worklists*[*i+1*] to be updated. Therefore, the updated level of the nodes in *Worklists*[*i+1*] is no longer the same as the initial division, and all nodes in this list may correspond to multiple levels. In addition, some transitive fanin (transitive fanout) relationships may also be formed between nodes belonging to different levels in this list due to logical sharing during the rewriting process of previous nodes. Because of the recursive nature of cut enumeration, the above relationship may cause the cuts of a node to be operated by multiple threads at the same time in parallel, resulting in thread dependency error. To this end, the proposed method recursively acquires exclusive locks for the current node and all its relevant nodes before the computation of cut enumeration. Note that the conflicts caused by exclusive locks in this stage are almost negligible, since the number of nodes with transitive fanin (transitive fanout) relationships accounts for only a small portion and the runtime of this stage is very short.

As shown in Fig. 2, different from the approach in this paper, the three stages of rewriting in [8] are completed in the same operator. Therefore, in the parallel process, it is necessary to acquire the exclusive locks for all related nodes of the current node in advance to ensure that subsequent replacements are successfully executed. If there is a conflict with other threads when acquiring the exclusive locks, the execution of the operator will be interrupted and all computations performed will be lost. In order to avoid such a waste of computing resources, DACPara implements the rewriting as three operators: enumeration, evaluation and replacement, and acquires the exclusive locks of the relevant nodes in the last operator.

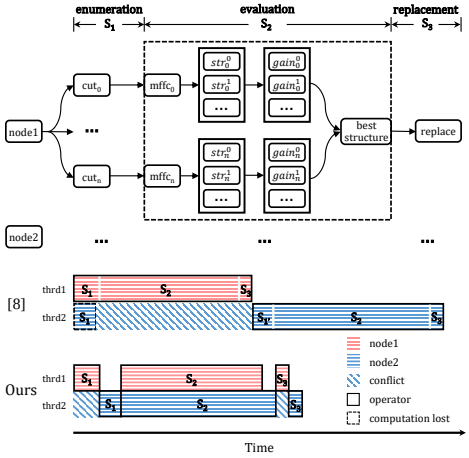

Figure 2: Comparison of operator parallelism in [8] and our approach when conflicts occur.

However, with the continuous replacement, the correctness of the cut has become a new issue that must be considered and addressed, which will be discussed in Section 4.4.

4.3 Parallel Evaluation

As the most time-consuming stage in the rewriting, evaluation takes up more than 90% of the runtime and does not actually modify the graph structure. However, the parallelism of this stage is severely constrained when an operator consisting of enumeration, evaluation, and replacement suffers conflicts, as illustrated in Fig. 2. In particular, for large-scale complex circuits, acquiring exclusive locks during parallel rewriting would cause more conflicts due to the large number of high-fanout nodes. To this end, DACPara eliminates all exclusive locks with a separate operator to completely release the parallelism in this stage. The result that can achieve the best gain for each node through the evaluation, including cut, its NPN class, equivalent structure, etc., is stored in the corresponding position in *prepInfo*.

After all exclusive locks are eliminated, an important point of parallel evaluation is to ensure the uniqueness of the data used for evaluation by all nodes in the current list. Otherwise, the evaluation results may be inaccurate. To solve this problem, DACPara creates copies of MFFC and NPN equivalent structures through the local data structure of thread to prevent these pieces of information from being temporarily modified. In addition, another key point is the structure hash used to search for logical shared nodes. This paper adopts the decentralized structural hashing method in [8] to create a local hash table for each AIG node for fast search so as to avoid using the global hash table.

4.4 Parallel Replacement

In DACPara, replacement is based on dynamic global information, which requires ensuring that the positive gain is correctly obtained on the latest AIG each time. Therefore, it is necessary to ensure the correctness of the enumeration results and the effectiveness of the evaluation results before changing the graph structure. During the initial partitioning, all nodes in the same list belong to the same level, meaning they have no fanin (fanout) relationships with each other. As rewriting continues, the nodes in the list no longer correspond to only one level, as explained in Section 4.2. At this time, the relationship between all nodes in the list can be expressed with or without transitive fanin (transitive fanout).

Theorem 1. *In AIG, if there is no transitive fanin (or transitive fanout) relationship between two nodes $n1$ and $n2$, then the rewriting of $n1$ will not affect the cut enumeration result of $n2$, that is, its leaves will always exist and still be the cut of $n2$.*

Proof. During the rewriting process of node $n1$, all nodes that can be deleted are included in the MFFC calculated recursively with $n1$ as the root. According to the definition of MFFC, it can be concluded that every path from any node in MFFC to PO passes through $n1$. This means that all transitive fanouts of all nodes in MFFC must have a transitive fanin (transitive fanout) relationship with $n1$. Since there is no transitive fanin (transitive fanout) relationship between nodes $n1$ and $n2$, then the nodes that need to be deleted in the rewriting of $n1$ do not have such a relationship with $n2$ either. All nodes that determine the logical function of the cut of node $n2$, including its *leaves* and nodes in the *cover*, are all transitive fanins of $n2$. Therefore, these nodes must not be the ones deleted during the rewriting process of $n1$. It can further be concluded that the logical function represented by the cut of $n2$ remains unchanged, that is, the *leaves* of the cut will always exist and are still the cut of $n2$. □

According to Theorem 1, the replacement of a node will not affect the logical functions represented by the cuts of all nodes that do not have a transitive fanin (transitive fanout) relationship with it. In this case, the correctness of the results of cut enumeration can be guaranteed. On the contrary, the existence of such relationships may also cause the result of the cut enumeration to be outdated, i.e. the result is no longer correct. In Fig. 3(a), {1, 2, 3, 4} and {4, 5, 6, 9} are the cuts of nodes 10 and 11 respectively, and node 10 is the transitive fanin of node 11. Fig. 3(b) shows the AIG after logically equivalent replacement of node 10. Note that the nodes 6 and 8, marked in red font here, reuse the IDs of the deleted nodes {6, 8, 10}. It can be seen that the logic expressed by node 6 has changed after being deleted and reused, so {4, 5, 6, 9} is no longer a cut of node 11 in terms of structural form and logical function. However, the set {3, 4, 5, 6} is still the cut of node 11 in structural form after rewriting, and whether the cut can eventually be used for replacement will be discussed later.

If the *leaves* of the stored cut always exist, i.e. they are not deleted, according to Theorem 1 and Theorem 2 in [16], it can be concluded that the cut is still a logical cut of the current node and can be correctly replaced. On the contrary, as long as any of the *leaves* does not always exist, then the stored cut must be judged. This is because the stored one may or may not be the cut of the current node on the latest AIG, as shown in the example in Fig. 3. In this case, DACPara re-enumerates the cuts on the current node and determines the correctness by matching the stored cut with the latest enumerated cut set. To ensure the timeliness and accuracy of the re-enumeration results, the previous enumeration results (if not empty) of all transitive fanouts for each deleted node will be recursively cleared.

In addition to ensuring the correctness of the stored cut, it is also necessary to ensure the effectiveness of the evaluated result. If the *leaves* in the stored cut of a node always exist, then the stored NPN structure must be used as an equivalent logical representation of the current node, because their truth tables are of the same class. Therefore, if there is still gain after re-evaluating the stored equivalent structure on the latest AIG, it can be used for equivalent replacement. For the other case, where some *leaves* in the cut after correctness determination are deleted and reused, it is necessary to

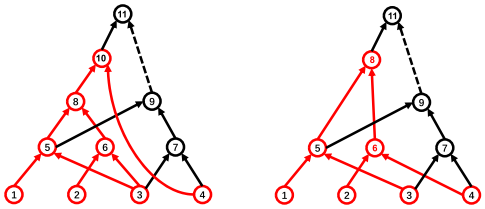

Figure 3: AIG (a)before and (b)after logic rewriting. The logic cone of node 10 is marked in red.

check whether the NPN class corresponding to the truth table of this cut matches the NPN class of the stored equivalent structure. A re-evaluation is also performed after a successful match. Finally, after locking all relevant nodes involved in the replacement stage of the current node, the graph structure is updated. It is undeniable that there will also be conflicts in this stage, but they are ignorable compared with the ones in the entire rewriting process for the similar reason mentioned in Section 4.2.

5 EXPERIMENT

5.1 Environment and Benchmarks

The approach in this paper was implemented in C++ using Galois system. It was compiled using gcc/g++ 7.5.0 and was conducted on a server with 256G RAM and a 2.90GHz AMD 3990X CPU, which has 64 physical cores.

DACPara adopted the same Arithmetic and Random/Control circuit sets derived from the EPFL benchmark suite as in [8], [16] and [9]. Specifically, all circuits with AIG nodes greater than 5000 in the above two sets are selected to apply the ABC command *double* to obtain larger benchmarks. Note that the command *double* only performs a simple copy of the original design, so its complexity remains unchanged. In order to verify the effectiveness for large-scale complex circuits, this paper also selected the same MtM set without using the command *double* as in [16] and [9] from the EPFL benchmark suite. The detail of the benchmarks is shown in Table 1.

Table 1: Benchmark Detail

Benchmark	PIs	POs	Area	Delay	Sources
sin_10xd	25476	25600	5545984	225	
voter_10xd	1025024	1024	14088192	70	
square_10xd	65536	131072	18927616	250	Arithmetic
sqrt_10xd	131072	65536	25208832	5058	+
mult_10xd	131072	131072	27711488	274	Random/
log2_10xd	32768	32768	32829440	444	Control
mem_10xd	1232896	1260544	47960064	114	[17]
hyp_8xd	65536	32768	58469760	24801	
div_10xd	131072	131072	58620928	4372	
sixteen	117	50	16216836	140	
twenty	137	60	20732893	162	MtM [17]
twentythree	153	68	23339737	176	

5.2 Results and Comparison

DACPara is compared with logic rewriting in ABC [7], state-of-the-art CPU parallel rewriting [8] (referred to as ICCAD'18), and two advanced GPU-accelerated rewriting methods [16], [9] (referred to as DAC'22 and TCAD'23, respectively). The experimental data reported in this paper are the average total time of executing the program 5 times, and the rewritten circuits all passed the equivalence check. The following is an introduction from the comparison between DACPara and the two parallel methods of CPU and GPU.

Serial logic rewriting in ABC, the state-of-the-art CPU parallel rewriting method ICCAD'18, and DACPara were all compiled and executed in the same software and hardware environment. Among them, ABC was executed using the command *rewrite*, because the other two methods are based on it. The experimental results are shown in Table 2.

Overall, DACPara is on average 34.36x and 1.96x faster than ABC and ICCAD'18 respectively, with extremely comparable quality. Specifically, DACPara suffered an average decrease of 0.18% and 0.56% in terms of area reduction, respectively. This is because the enumeration and evaluation results of each node in ABC and ICCAD'18 are calculated on the latest AIG, allowing each node to achieve local optimal results. However, these results for all the nodes in a list in DACPara are all calculated based on the AIG updated by the latest list. Therefore, subsequent changes in AIG as the replacement proceeds may cause these results of a small number of nodes in the current list to be outdated and miss optimization opportunities. It is worth mentioning that a very small proportion of nodes that miss optimization opportunities enable large-scale complex benchmarks to jump out of local optima to achieve better quality during rewriting, which we will explain later. In terms of delay, the three methods are basically the same. In addition, DACPara demonstrated a significant runtime advantage on all large-scale benchmarks in the MtM set. Compared to ICCAD'18, DACPara achieved an average speedup of 4.37x while showing improvements in both area reduction and delay, as shown in Table 3. This indicates that the approach proposed in this paper can effectively release parallelism and jump out of local optimal solutions in terms of quality for large-scale complex benchmarks. The remaining circuits with simple logic will only produce few conflicts in parallel, so the runtime of DACPara and ICCAD'18 is basically comparable except for the three circiuts "sqrt_10xd", "hyp_8xd" and "div_10xd". Such slightly longer runtime is due to the large number of levels, resulting in a larger number of lists that need to be processed.

At present, the works on GPU-accelerated rewriting have not been open sourced, and these works are based on another rewriting operator *drw* in ABC. This operator selects all 222 NPN equivalence classes for evaluation while the operator *rewrite* only selects the common 134 classes. Furthermore, *drw* only selects 8 cuts of the node and 5 equivalent structures of the NPN class during the evaluation process. The latest research of DAC'22 and TCAD'23 both utilized a GPU with 9216 cores for acceleration and execute the program twice for the experiments. Therefore, direct comparisons in terms of runtime and quality are unfair.

To this end, DACPara applied two parameters for comparison, and the experimental results are shown in Table 3. Among them, the parameter *P1* means that DACPara only selects 8 cuts and 5 equivalent structures of NPN classes for evaluation for each node, and the program is executed twice. The difference is that due to the applied operators, *P*1 can only use 134 NPN equivalence classes for evaluation, while both DAC'22 and TCAD'23 use all 222 classes. Parameter *P2* adopts the same configuration as ICCAD'18, which uses 134 NPN classes to execute the program only once without limiting the number of cuts and equivalent structures. The experimental data of DAC'22 and TCAD'23 are all from those reported in their papers. It can be observed that DACPara-P1 is almost the same as the two GPU accelerated methods in terms of area reduction, despite the fact that our approach only utilizes a smaller set of 134 NPN classes for evaluation. This indicates that the replacement based on dynamic global information in parallel is effective for improving the quality. In terms of delay, DACPara-P1 deteriorates by about 4%, caused by the reduced number of NPN classes used for evaluation. Due to the significant difference in the number of cores,

Table 2: Results of DACPara (Ours) and Two Other Works

Benchmark	ABC (1 Thread)			ICCAD'18 (40 Thread)			DACPara (40 Thread)		
	Time (s)	Area Reduction	Delay	Time (s)	Area Reduction	Delay	Time (s)	Area Reduction	Delay
sin_10xd	78.9	80896	223	1.7	84992	223	2.0	80896	223
voter_10xd	218.3	2534400	63	3.9	2700474	67	4.5	2584582	64
square_10xd	251.2	123904	250	4.9	152485	250	5.1	121856	250
sqrt_10xd	392.7	6285312	6048	7.3	6285312	6048	11.7	6285312	6048
mult_10xd	396.9	159744	274	7.3	191127	274	8.2	159744	274
log2_10xd	562.7	525312	443	10.5	527416	443	12.6	524288	443
mem_10xd	351.8	74752	114	6.5	75576	114	8.8	65536	114
hyp_8xd	747.6	15616	24801	17.2	10496	24801	27.3	10496	24801
div_10xd	760.5	16482304	4406	15.6	16465773	4407	23.1	16425681	4405
sixteen	393.5	4039197	109	58.3	4023449	109	14.3	4023046	109
twenty	550.3	5221299	111	79.3	5203296	114	18.1	5205173	114
twentythree	665.9	5966375	129	94.2	5945892	130	20.6	5948166	129
Normalized Mean	34.3589	1.0018	**0.9999**	1.9623	**1.0056**	1.0002	**1**	1	1

Table 3: Results of DACPara (Ours) and Three Other Works

Benchmark	ICCAD' 18 (40 Thread)			DAC' 22 (1GPU 9216 cores)			TCAD' 23 (1GPU 9216 cores)			DACPara-P1 (40 Thread)			DACPara-P2 (40 Thread)		
	T (s)	Area Re	D	T (s)	Area Re	D	T (s)	Area Re	D	T (s)	Area Re	D	T (s)	Area Re	D
sixteen	58.3	4023449	109	7.8	3971553	105	2.1	3979618	108	14.9	3968746	109	14.3	4023046	109
twenty	79.3	5203296	114	9.9	5150895	110	2.8	5144381	108	18.9	5147274	113	18.1	5205173	114
twentythree	94.2	5945892	130	11.2	5861343	124	3.2	5878184	123	21.5	5862782	128	20.6	5948166	129
Norm Mean	4.3735	0.9997	1.0028	0.5452	0.9873	**0.9630**	**0.1528**	0.9885	0.9630	1.0433	0.9869	0.9943	1	**1**	1

the proposed method does not have any advantages in runtime. However, as the design complexity and scale increase, GPU accelerated methods may face new challenges in terms of quality due to limitations in memory and single-core computing power, which this method can easily deal with. Specially, DACPara-P2 achieved an improvement of around 1.1% in area reduction compared to DAC'22 and TCAD'23. This demonstrates the significant potential of the proposed method in processing large-scale complex benchmarks.

6 CONCLUSION

In this paper, we propose DACPara, a divide-and-conquer parallel approach for high-quality logic rewriting in large-scale circuits. In order to solve the problems of limited parallelism and poor quality caused by static information, the operators in the three stages of cut enumeration, evaluation and replacement are separated and redesigned based on dynamic global information. Experimental results indicate that the proposed method achieved an average speedup of 1.96x compared to state-of-the-art CPU parallel method with essentially the same quality. In addition, compared with state-of-the-art GPU accelerated method, the proposed approach demonstrates significant potential in terms of scalability and quality.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China under Grant Nos. 62172322, 62272359 and 62192734, the Fundamental Research Funds for the Central Universities and the Innovation Fund of Xidian University with Grant No. YJSJ24015. Cong Tian is the corresponding author.

REFERENCES

[1] Robert K Brayton, Gary D Hachtel, and Alberto L Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[2] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-Aware AIG Rewriting A Fresh Look at Combinational Logic Synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535. ACM, 2006.

[3] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1907–1911, 2017.

[4] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[5] Mathias Soeken, Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Optimizing Majority-Inverter Graphs With Functional Hashing. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1030–1035. IEEE, 2016.

[6] Winston Haaswijk, Mathias Soeken, Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. A Novel Basis for Logic Rewriting. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 151–156. IEEE, 2017.

[7] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 24–40. Springer, 2010.

[8] Vinicius Possani, Yi-Shan Lu, Alan Mishchenko, Keshav Pingali, Renato Ribas, and Andre Reis. Unlocking Fine-Grain Parallelism for AIG Rewriting. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[9] Lin Li, Rui Li, and Yajun Ha. A Recursion and Lock Free GPU-based Logic Rewriting Framework Exploiting Both Intra-node and Inter-node Parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[10] Alan Mishchenko and Robert Brayton. Scalable Logic Synthesis using a Simple Circuit Structure. In *Proc. IWLS*, volume 6, pages 15–22, 2006.

[11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25. ACM, 2011.

[12] Augusto Neutzling, Felipe S Marranghello, Jody Maick Matos, Andre Reis, and Renato P Ribas. maj-*n* Logic Synthesis for Emerging Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(3):747–751, 2019.

[13] Heinz Riener, Eleonora Testa, Winston Haaswijk, Alan Mishchenko, Luca Amarù, Giovanni De Micheli, and Mathias Soeken. Scalable Generic Logic Synthesis: One Approach to Rule Them All. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6. ACM, 2019.

[14] Mahmoud Elbayoumi, Mihir Choudhury, Victor Kravets, Andrew Sullivan, Michael Hsiao, and Mustafa Elnainay. TACUE: A Timing-Aware Cuts Enumeration Algorithm for Parallel Synthesis. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[15] Gai Liu and Zhiru Zhang. A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 147–156. ACM, 2017.

[16] Shiju Lin, Jinwei Liu, Tianji Liu, Martin DF Wong, and Evangeline FY Young. NovelRewrite: Node-Level Parallel AIG Rewriting. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 427–432. ACM, 2022.

[17] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL Combinational Benchmark Suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.