Unlocking Fine-Grain Parallelism for AIG Rewriting

Vinicius Possani[1], Yi-Shan Lu[2], Alan Mishchenko[3], Keshav Pingali[2],
Renato Ribas[1] and Andre Reis[1]
[1]Institute of Informatics, Universidade Federal do Rio Grande do Sul, Brazil
[2]Department of Computer Science, The University of Texas at Austin, USA
[3]Department of EECS, University of California, Berkeley, USA

ABSTRACT

Parallel computing is a trend to enhance scalability of electronic design automation (EDA) tools using widely available multicore platforms. In order to benefit from parallelism, well-known EDA algorithms have to be reformulated and optimized for multicore implementation. This paper introduces a set of principles to enable a fine-grain parallel AND-inverter graph (AIG) rewriting. It presents a novel method to discover and rewrite in parallel parts of the AIG, without the need for graph partitioning. Experiments show that, when synthesizing large designs composed of millions of AIG nodes, the parallel rewriting on 40 physical cores is up to 36x and 68x faster than ABC commands *rewrite -l* and *drw*, respectively, with comparable quality of results in terms of AIG size and depth.

Keywords

AND-Inverter Graph; K-Cuts; Logic Rewriting; Parallel Computing; Operator Formulation; Galois System.

1. INTRODUCTION

Fast algorithms for electronic design automation (EDA) are crucial in the design flow of current and future generations of integrated circuits (IC). The high complexity of system-on-chips (SoCs) and the increasing demand for hardware accelerators impose new challenges on the EDA field. Moreover, short design cycles are essential for improving productivity and reducing project cost. This has led to the concept of *EDA 3.0*, which emphasizes the need for a new generation of parallel and distributed computer-aided design (CAD) tools able to quickly handle large designs [21, 22].

In this scenario, some EDA algorithms must be rethought to work in parallel environments. In this paper, we revisit the *multi-level optimization*, which aims to prepare a logic network for technology mapping into standard cells or programmable devices [6]. Multi-level optimization is an important and time-consuming task during logic synthesis because the underlying logic representation has direct impact on the quality and speed of technology mapping, also known as *structural bias* [7, 14].

Local transformations, such as *logic rewriting*, play an important role in logic synthesis. Typically, the rewriting process uses a hash table of precomputed structures to incrementally update the underlying logic representation to improve a cost function. The common costs used to guide the rewriting techniques are the network *size* and *depth* (that is, the number of nodes and levels in a direct acyclic graph (DAG)). In recent years, many approaches for logic rewriting have been proposed, which consider different sets of precomputed structures and are based on logic representations, such as AND-inverter graph (AIG) and majority-inverter graph (MIG) [17, 13, 23, 20, 10]. Even though Boolean function decomposition and factorization are general, logic rewriting is a fast alternative that can be applied incrementally, allowing different trade-offs between runtime and quality-of-results (QoR). As a result, rewriting techniques are often more suitable than Boolean decomposition for multi-level optimization of large IC designs.

It is predicted that future generations of integrated circuits will contain trillions of logic gates [21, 22]. Currently, the time needed to synthesize large AIGs with more than ten million nodes has become considerable, even when fast rewriting methods are applied. Consider, for instance, command *rewrite* [17] in the logic synthesis tool ABC [1]. This command takes approximately 18 minutes on a modern processor to perform a single iteration of rewriting for a 35-million-node AIG, as shown in our experiments. The command *rewrite* is one of the main algorithms used in the well-known ABC scripts *resyn2* and *dc2* for delay- and area-oriented synthesis, respectively. Moreover, rewriting techniques are often applied many times to compensate for the local nature of the transformations. Therefore, the runtime may become a significant part of logic synthesis. These observations and the wide availability of multi-core processors have motivated the investigation of parallel solutions for logic synthesis problems.

This paper introduces a fine-grain parallel AIG rewriting that relies on the *operator formulation* and the Galois system [19, 11], in a multicore environment with shared memory. The proposed approach is based on the AIG rewriting proposed by Mishchenko *et al.* in [17]. Fine-grain parallelism allows us to rewrite multiple nodes of the graph at the same time, rather than applying graph partitioning and rewriting each partition sequentially. The *operator formulation* is a data-centric abstraction of algorithms, which are described in terms of a collection of atomic actions on data structures [19]. These actions are determined using an *operator*, which specifies an atomic update to the data structure, and a *schedule* for applying the operator to certain sites in the data structure. The Galois system provides a programming model where a specified *operator* is speculatively executed to exploit parallelism in irregular algorithms such as graph applications [11]. AIG rewriting is an irregular algorithm that fits well in the Galois approach in which non-overlapping subgraphs are discovered and rewritten in

ICCAD '18, November 5–8, 2018, San Diego, CA, USA
 ISBN 978-1-4503-5950-4/18/11…$15.00
DOI: https://doi.org/10.1145/3240765.3240861

parallel.

The main contribution of this paper is a set of principles that describe how to unlock the parallelism during AIG rewriting. Our experiments show that the proposed approach is able to speed up the multi-level optimization of huge designs without penalty in the QoR when compared to the reference method.

The rest of this paper is organized as follows. Section 2 presents preliminary definitions. Section 3 presents an overview of related works on AIG and MIG rewriting. A more detailed view of the Galois system is presented in Section 4. Section 5 presents the proposed approach for parallel AIG rewriting. Experimental results are summarized in Section 6, and an analysis of speedup and QoR is presented. Finally, Section 7 discusses conclusions and future work.

2. PRELIMINARIES

An *AND-inverter graph* (AIG) is a directed acyclic graph adopted as the data structure in logic synthesis. AIG is a homogeneous circuit representation containing four type of nodes: the constant, primary inputs (PI), primary outputs (PO), and two-input AND (AND2) gates, possibly with complemented inputs. Sequential elements such as latches can be viewed as special nodes or pseudo-PIs/POs. The graph edges play the role of directed or complemented interconnections. A *majority-inverter graph* (MIG) is defined similarly [2]. However, instead of two-input AND gates, a MIG comprises three-input majority (MAJ3) gates.

The set of nodes connected to the inputs (outputs) of a given AIG node is called its *fanins* (*fanouts*). In the context of AIGs, *structural hashing* is a technique adopted to ensure that there is no AND2 gates with the same pair of fanins, even up to permutation [15]. Conventionally, the AIG manager stores each AND2 node in a hash table by building a *key* in terms of its fanin edges and nodes. Before creating a new node, the AIG manager performs a lookup in the hash table and reuses the node with the same fanins if such node exists.

A *maximum fanout free cone* (MFFC) of a given AIG node n is a subset S of the predecessors of n such that every path from any node in S to a PO passes through n. In other words, the MFFC of a given node n contains the nodes that are exclusively used to define the logic function of n. Therefore, if the node n is removed, all nodes in its MFFC can also be removed [15].

A *cut* C rooted in a given AIG node n is a set of nodes, called *leaves*, such that every path from the PIs to the root n contains at least one *leaf* $\in C$. A cut C_1 is dominated by another cut C_2 if $C_2 \subseteq C_1$, i.e., C_1 is a redundant cut. A cut is *k-feasible* if it contains k nodes or less; such a cut is known as a *k-cut*. A *k-cut* is associated with a local Boolean function, which is defined by the logic cone of the root node n and expressed in terms of the cut *leaves*. Boolean functions of cuts up to 16 inputs can be represented by truth tables, implemented using bit-strings, and manipulated using bitwise operations.

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 [15]. In the context of logic rewriting, NPN classification is commonly used to find logically equivalent but structurally different implementations used to rewrite a cut in the subject graph.

3. RELATED WORKS ON REWRITING

In this section, we briefly describe the rewriting methods, highlighting the main innovation that each one introduced. First, we show that the recent rewriting methods are based on the DAG-aware AIG rewriting method [17]. Therefore, if we understand how to unlock the parallelism in this rewriting approach, we can exploit parallelism in other state-of-art logic synthesis methods. In other words, we can extend the proposed parallel rewriting to incorporate the best characteristics of previous methods as well as investigate novel solutions in this direction. Moreover, parallel logic synthesis enables intensive and iterative optimizations while controlling computation time [9, 14].

3.1 DAG-Aware AIG Rewriting

Mishchenko *et al.* in [17] proposed a *rewrite* algorithm that can be interleaved with two other techniques: *refactor* [6] and *balance* [8]. The *rewrite* algorithm extends the prior work [5] and comprises three main components:

A hash table of precomputed structures serves as a database to represent 4-input AIGs used for replacement. The set of all 4-input Boolean functions can be grouped in 222 NPN classes. Many of these classes appear rarely in practical designs and therefore only about one hundred of the NPN classes are used in logic optimization. A hash table containing optimized AIG implementations for this useful subset is precomputed in advance and loaded into the rewriting manager. During the rewriting, the hash table is used to retrieve a subset of the subgraphs.

The enumeration of 4-feasible cuts is performed to select parts of the AIG structure to be rewritten. Thus, all cuts that implement a 4-input Boolean function are candidates to be replaced, if there exist a better and equivalent subgraph in the hash table.

Overall procedure to find improved implementation for AIG cuts works as follows. For each node n, in the topological order, the algorithm computes 4-input cuts and their respective truth tables. The Boolean functions of the 4-input cuts of n are mapped into their NPN classes, which are used to find a new cut implementation in the hash table of precomputed structures. The gain obtained by rewriting a given cut, rooted in the node n, is calculated in terms of the number of nodes that will be deleted and added into the AIG. Thus, the cut/implementation that leads to the best local improvement is greedily selected to replace the old structure of the cut. The number of deleted nodes is related to the MFFC of n. The number of added nodes is related to the amount of nodes in the window of n that can be reused (shared) to express the new implementation of the cut. The structural hashing technique, mentioned in the preliminaries of this paper, is commonly used to figure out such logic sharing.

3.2 Recent Rewriting Methods

The local scope of 4-input cuts and the reduced set of 4-input Boolean functions, used in the previous method [17], motivated Li *et al.* in [13] to extend this approach to 5-input cuts. The authors introduced a technique to build a library of precomputed subgraphs that can implement 1,185 NPN classes of 5-input Boolean functions. The 5-input cut enumeration and subgraph replacement is analogous to the previous work.

Another work presents an alternative to increase the size of k-cuts and explore larger databases with more complex Boolean functions, as introduced by Yang *et al.* in [23]. The paper presents an approach to extract several optimized subnetworks from designs already synthesized with different techniques. Similar to the previous methods, the extracted subneworks are stored and viewed as a library of structures to be used for logic rewriting.

Soeken *et al.* in [20] proposed an approach for rewriting MIG in three different graph traversals, top-down, bottom-up, and based on fanout-free regions. Moreover, the authors proposed an exact MIG synthesis method, which was used to build a hash table of precomputed structures to be used during rewriting.

Recently, a new method for *XOR Majority Graphs* (XMG) rewriting was proposed by Haaswijk *et al.* in [10]. This approach uses a LUT-based technology mapper to mine useful Boolean functions to be considered during the rewriting process. Then, the exact MIG synthesis proposed in [20] is used to compute implementations for the collected functions. The implementations are stored in a database used for logic rewriting, similarly to the previous work.

In the last year, two other techniques were proposed to perform technology independent synthesis together with technology mapping [14, 4]. Liu *et al.* in [14] proposed a parallel iterative approach that applies logic transformations, *e.g., rewrite*, *balance*, and *refactor*, as stochastic moves to change the logic network during technology mapping. This approach leads to improvements due to the ability to escape local minima and overcome structural bias.

The most recent approach, proposed by Amarú *et al.* in [4], is based on DAG-aware AIG rewriting. However, instead of replacing k-cuts by improved AIG structures, the method replaces k-cuts by combinations of standard cells from a precomputed database. The paper introduces the concept of *equioptimizable arrival times*, which retrieves the combination that minimizes the delay at the cut output.

4. GALOIS SYSTEM

Galois is a system that provides a data-centric programming model to exploit *amorphous data parallelism* in irregular graph algorithms [19]. It is based on an abstraction of algorithms called the *operator formulation*. In this abstraction, there is a local view and a global view of algorithms.

- The local view is described by an *operator*, which specifies an action that modifies the state of the graph atomically. Each application of the operator is called an *activity*, and the region of the graph modified by an activity is called its *neighborhood*.
- In general, there may be many places in a graph where an operator can be applied. If there is an order in which these operator applications must be performed, that is specified by the *schedule*, which provides a global view of the algorithm. For the algorithms of interest in this paper, operator applications may be performed in any order, so these are called *unordered* algorithms.
- Usually, each activity modifies only a small portion of the overall graph. Therefore, for unordered algorithms, activities that modify non-overlapping regions of the graph can be performed in parallel without changing the semantics of the program. A pair of activities with overlapping neighborhoods can be performed in either order but not concurrently.

From the programmer's point of view, Galois provides C++ thread-safe data structures such as graphs and sets, and parallel executors such as *for_each, do_all* [11]. Thread-safe sets are used to implement worklists to store active nodes. The parallel executor consumes nodes from the worklist and dynamically assigns them to threads. Operator execution is speculative and optimistic in the sense that the activities are assumed to be non-conflicting but Galois dynamically treats and reschedules activities if conflicts happen. Galois scheduling is non-deterministic although it can be modified to work deterministically with some runtime cost. The dynamic management of thread conflicts is needed in irregular algorithms, unlike in regular algorithms in which non-conflicting activities can be found and scheduled statically.

Optimistic scheduling of activities is implemented as follows. Graph elements have exclusive locks to ensure mutual exclusion when threads are changing the graph. Therefore, Galois manages thread conflicts in the owners of locks from graph elements. Threads hold the abstract locks until the end of an activity or until a conflict is detected. For instance, consider two threads t_1 and t_2 that are processing the active nodes n_1 and n_2, respectively, and n_3 is a shared neighbor of n_1 and n_2. If t_1 acquires the lock of n_3, then t_2 will not be able to proceed the execution of its operator until the lock of n_3 is released. In these cases, Galois detects the conflict and aborts the execution of the operator at the active node n_2 by releasing its already acquired locks. Then, the active node n_2 is rescheduled to be processed later.

Intuitively, activities are processed in an all-or-nothing approach in terms of the acquisition of the necessary locks. When the execution of the operator in a given active node is aborted due to conflicts, all computation performed at this point is lost. In this sense, it is desirable to design *cautious operators*, which first try to acquire all necessary locks in the neighborhood of the active node and only then perform the graph modifications. This way, after acquiring necessary locks, it is possible to ensure that the operator will be successfully executed without wasting time in the complex computation before all locks are available.

Although Galois offers a high level of abstraction for programmers, recent studies have compared Galois to a native thread implementation such as *pthreads* and have shown that the abstraction penalty is small. An efficient parallel FPGA router was designed using Galois [18]. In [18], Moctar and Brisk state that they believe that the Galois model is the right solution for parallel CAD. This statement is based on the wide number of irregular graph-based algorithms used to solve problems in EDA.

5. PARALLEL AIG REWRITING

This section presents our reformulation of the rewriting method presented in [17] to unlock the parallelism using the Galois programming model. Algorithm 1 describes the top-level routine that aims to initialize the necessary components and call the Galois parallel *for_each*. This routine receives as input the AIG and other parameters such as the number of cuts stored per AIG node, the maximum number of precomputed structures tried per cut and the flag indicating if the zero-cost replacement is enabled.

The proposed approach is based on 4-input cuts, NPN-equivalence, and the same set of precomputed structures used in the command *rewrite* from ABC [1]. The rewriting manager and all the other necessary modules are instantiated in line 3 of Algorithm 1. The initial set of active nodes is defined by all AIG primary inputs, as shown in lines 6-7 of Algorithm 1. When the AIG under optimization comprises sequential elements, all latches can also start as active nodes, similarly to the primary inputs. Active nodes are stored in a Galois thread-safe worklist, where its items are distributed in local worklists along the cores. In line 9 of Algorithm 1, the parallel *for_each* dynamically assigns active nodes to available threads as the computation proceeds. The activities are performed by executing the operator implemented in the rewrite manager.

5.1 Rewriting Manager

Algorithm 2 presents an overview of the rewriting manager, which implements the operator executed at each active node. Essentially, the rewriting operator works just like the reference method in the sequential code. However, its main internal routines were rethought to support concurrent operations. Notice that the rewriting manager is constructed based on references to other shared-memory objects e.g., AIG, $cutMan$, $npnMan$ and $strMan$. These auxiliary managers are discussed in the next subsection.

The operator receives as argument an *active node* and the

Algorithm 1: Parallel AIG Rewriting

```
1  Function parallelAIGRewriting()
       Input  : AIG, nCuts, nStr, useZero
       Output: a rewritten AIG or the original one
2      // Instantiate all necessary components
3      RewritingManager rwMan( AIG, nStr, useZero,
         cutMan(4, nCuts), npnMan(), strMan() );
4      GaloisWorklist = ∅;
5      // All primary inputs and latches are active nodes
6      for each pi from AIG do
7          GaloisWorklist.push( pi);
8      // Parallel For
9      GaloisForEach( GaloisWorklist, rwMan);
```

Algorithm 2: Rewriting Manager Operator

```
1  RewritingManager begin
2      constructor( AIG, nStr, useZero, cutMan,
         npnMan, strMan)
3      Function operator( node, GaloisCtx)
4          C = cutMan.computeKCuts( node);
5          lockFanoutNodes( node);
6          for each 4-input cut c in C do
7              lockFaninCone( node, c);
8          bestS = null;
9          bestG = -1;
10         for each 4-input cut c in C do
11             nSaved = computeMFFC( node, c);
12             f = cutMan.getCutFunction( node, c);
13             f_npn = npnMan.getRepresentative( f);
14             S = strMan.lookupStructures( f_npn);
15             for each structure s in S up to nStr do
16                 nAdded = countAddedNodes( s, c);
17                 gain = nSaved − nAdded;
18                 if (gain > 0 || gain == 0 && useZero)
                     then
19                     if (bestS == null || bestG < gain)
                         then
20                         bestS = s;
21                         bestG = gain;
22         if (bestS != null) then
23             updateAIG( node, bestS);
24         pushFanoutNodes( node, GaloisCtx);
```

GaloisCtx, which provides access to the Galois context such as worklists and customized memory allocators. In order to minimize the computation lost due to thread conflicts, all logical locks in the neighborhood of an active node must be acquired in advance. In the proposed approach, such a neighborhood is defined by a window containing the fanouts of the active node and the logic cones rooted into the active node and expressed in terms of its k-cut leaves. The operations performed in lines 4-7 of Algorithm 2 make the operator cautious. In case of conflict, the operator is aborted as soon as possible and does not lose any logic optimizations already done. This way, only the k-cuts of the active node are discarded (lost) because another thread can change the graph structure and make such cuts inconsistent.

In the sequence, all 4-input cuts are evaluated in order to figure out the best cut/structure to be used for rewriting. The number of precomputed structures tried per cut can be limited by using the parameter *nStr*, shown in line 15 of Algorithm 2. When the operator reaches line 22, it means that all necessary locks were acquired and the subgraph replacement can be committed safely. In other words, the operator cannot make the AIG inconsistent by aborting during the subgraph replacement.

Finally, the last task of the operator is to evaluate the fanouts of the active node and determine the ones that may become active. We adopted other two auxiliary labels, beyond the label *active*, to define the state of each AIG node. Therefore, a node can be unprocessed (*inactive*), under processing (*active*) or already processed (*done*). As the k-cut enumeration is based on the cuts of previous levels, an AND node can become active and pushed into the worklist only when its two fanin nodes were already processed (*done*). Such constraint is required to avoid inconsistencies in k-cuts, which may be introduced when the operator aborts its execution due to thread conflicts. The routine *pushFanoutNodes* called in line 24 of Algorithm 2, is responsible for evaluating and pushing the fanout nodes that are ready to become active.

5.2 Access to Shared Data Structures

This subsection presents the design of auxiliary data structures that work in a multi-threaded environment. Figure 1 shows how the data structures are organized and how threads communicate.

Cut Manager is responsible for providing all necessary routines and data structures for k-cut computation and storage. When it comes to high-performance computing, it is important to handle memory allocation efficiently. As the k-cut enumeration creates a large set of cuts, it is wise to use pre-allocated memory for cut storage. In this sense, we designed the cut manager so that each thread has its own memory pool. Thus, when a thread needs to compute k-cuts, a thread-local pointer is used to get access to its own memory pool, as shown in Fig. 1. This way, we unlock the parallelism by avoiding dependencies among threads and avoiding call to the operating system for memory allocation. The initial size of each memory pool is defined according to the initial AIG size and each memory pool is resized as necessary. It is worth mentioning that, even though we are using 4-input cuts for rewriting, the proposed parallel k-cuts were designed to compute cuts for any *k*. Moreover, the parallel k-cut enumeration can be easily adapted to work during technology mapping with priority cuts [16]. An evaluation of parallel k-cuts is presented in the section of experimental results showing scalability for large designs.

NPN Manager provides a fast NPN classification tailored to work with 4-input Boolean functions. It provides NPN classification in constant time as all of the 65,536 4-input Boolean functions and their respective 222 classes are stored in a hash table, including the inputs/output phase assignments and permutations of each function. This manager is strongly based on the ABC code [1] and does not require any significant modification to work in a parallel environment. The only restriction is that the hash table must be read-only, designed in such way that all threads can access the data without any kind of locks or synchronization.

Structure Manager stores a set of precomputed structures containing efficient implementations for a subset of 134 useful NPN classes selected out of all 222 NPN classes. In [17],

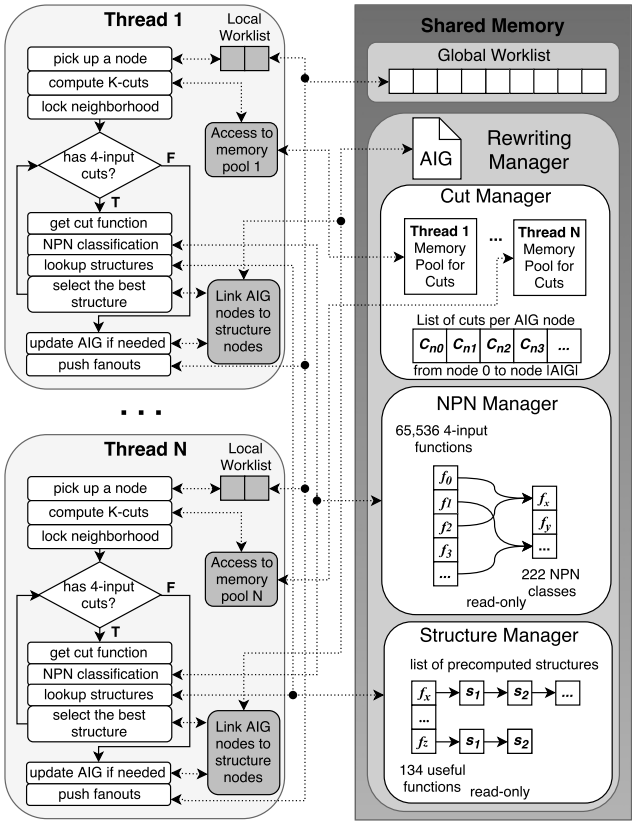

Figure 1: Relationship among rewriting operator, thread-local, and shared-memory data structures.

the authors defined as useful all 4-input Boolean functions appearing as functions of 4-input cuts selected during AIG rewriting on a set of benchmarks. We consider the same set of structures as used in the ABC command *rewrite* but change the method used to try a structure for a given cut.

Generally speaking, the precomputed structures act as templates to guide the construction of improved and logically equivalent arrangements of AIG nodes. In this sense, when a given structure is tried as a candidate for replacement, it is needed to keep a temporary one-to-one assignment between each node of the precomputed structure ("template nodes") and its corresponding AIG node ("real nodes"). This one-to-one assignment is used to figure out how many AIG nodes can be reused or must to be created for implementing the new subgraph. In the reference method, such one-to-one assignment is done using pointers from the nodes of precomputed structures to their correspondent nodes in the AIG. In other words, when a precomputed structure is tried, it is necessary to write information in such a structure. Therefore, the main challenge in this step is that many threads can try to use the same set of precomputed structures simultaneously, requiring an strategy to ensure mutual exclusion.

We solve this issue by making the table of precomputed structures read-only and using a thread-local data structure for tracking the one-to-one assignment of nodes. The IDs of nodes from precomputed structures are used to index a thread-local map, which contains pointers to temporary copies of the corresponding AIG nodes. This approach enables parallelism in AIG rewriting, since many threads can use the same precomputed structures simultaneously without using locks.

Parallel-aware structural hashing is a reformulation of the

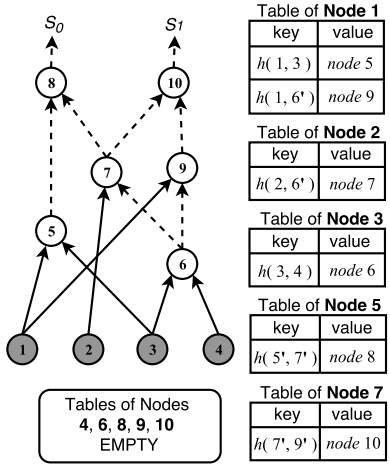

Figure 2: Proposed approach for decentralized structural hashing. Assuming h as a hash function applied on the pair of edges/nodes, where complemented edges are represented as apostrophes (') added to node ids.

conventional AIG structural hashing, making it more suitable to work in a parallel environment. Conventionally, structural hashing is performed using a global hash table. Each AIG node is added to this table using its fanins as a hash key. This hash table is built when the AIG is parsed and used during the operations that change the AIG structure. For instance, when a synthesis method is about to add an AND node to the AIG, the hash table is used to ensure that there is no replicated AND nodes with exactly the same fanins. Therefore, if an equivalent node is found in the table, it can be reused (shared). Otherwise, a new AND is created and added to the table. However, notice that handling a global hash table in a multi-threaded environment may lead to a bottleneck.

Structural hashing plays an important role in AIG rewriting, allowing the identification and sharing of equivalent AND nodes during subgraphs replacement. In line 14 of Algorithm 2, the structural hashing *lookup* is intensively called as an internal routine to determine how many nodes can be reused if the current structure is used for rewriting the AIG. In this context, we propose to use a decentralized scheme of hash tables, which works efficiently for performing parallel *lookup*, *insertion* and *deletion* of AIG nodes. The proposed approach is based on the observation that structural hashing relies on: *given a pair of nodes (n_1, n_2) and a pair of edge polarities (e_1, e_2), check if there exists a two-input AND node n_3, which is connected to the **fanout** of n_1 and n_2 through the polarities e_1 and e_2, respectively.* This task can be solved by searching for node n_3 directly in the fanout of nodes n_1 and n_2, avoiding the use of a global hash table.

However, a linear search in the fanout of n_1 and n_2 may be a time consuming task due to high fanout nodes. In this sense, we are using a local hash table at each AIG node in order to lookup its fanout nodes as fast as in conventional structural hashing. Concerning the memory usage, we have adopted a rule to ensure that each two-input AND node is stored exactly once, only in the hash table of its fanin node with the *smallest identifier* (*id*). For instance, assume that node n_3 has fanins n_1 and n_2 and assume that $n_1.id < n_2.id$. In this case, n_3 is stored only in the hash table of node n_1. This rule saves memory without losing the property of conventional structural hashing.

In the context of the algorithm, it is only needed to perform a simple comparison of node ids to access the right hash

Table 1: Subset of EPFL arithmetic and random control circuits after applying ABC *double* 10x (8x).

Benchmark	PIs	POs	ANDs	Levels
sin_10xd	24,576	25,600	5,545,984	225
arbiter_10xd	262,144	132,096	12,123,136	87
voter_10xd	1,025,024	1,024	14,088,192	70
square_10xd	65,536	131,072	18,927,616	250
sqrt_10xd	131,072	65,536	25,208,832	5,058
mult_10xd	131,072	131,072	27,711,488	274
log2_10xd	32,768	32,768	32,829,440	444
mem_10xd	1,232,896	1,260,544	47,960,064	114
hyp_8xd	65,536	32,768	54,869,760	24,801
div_10xd	131,072	131,072	58,620,928	4,372

table and, then, apply the conventional operations for node *lookup*, *insertion* and *deletion*. Moreover, if a given thread owns the locks for the pair of nodes n_1 and n_2, it means that such thread has exclusive access to the hash tables of these nodes, ensuring mutual exclusion. In other words, this approach fits well with the Galois strategy to handle logical locks.

Figure 2 illustrates an example of the decentralized approach for structural hashing. In this case, the structural hashing for the AIG presented in the left side of Fig. 2 leads to the set of non-empty hash tables of nodes 1, 2, 3, 5 and 7. In this case, nodes 4, 6, 8, 9 and 10 have empty tables because their respective fanouts are already registered in the hash table of some node with smaller identifier. Notice that, such AIG has six AND nodes and the hash tables, altogether, also store only six nodes. This way we efficiently manage memory usage, compared to the conventional structural hashing. The benefits of the proposed approach become more evident because logic sharing increases when rewriting is applied to large AIGs.

6. EXPERIMENTAL RESULTS

The parallel AIG rewriting was implemented in C++ 11 using Galois system [19, 12]. Both the proposed method and ABC [1] were compiled using GNU g++ version 6.1.0 and executed in a 64-bit Linux distribution. The results were collected on a server with 128GB of shared RAM and 4 processors Intel®Xeon®CPU E7- 4860 at 2.27GHz, where each processor has 10 physical cores. We ran command *&cec* in ABC to check the correctness of the parallel rewriting on a large set of MCNC, ISCAS, and EPFL benchmarks.

In all of the experiments, we ran ABC and the parallel rewriting five times for each design and for each number of threads, in order to minimize the effects of external noise on the runtime. Therefore, the results under comparison were obtained by computing the average among five executions. Since the Galois scheduling for handling thread conflicts is non-deterministic, we compute the average *size* and *depth* of the AIGs optimized by the parallel rewriting.

6.1 Benchmarks

The parallel rewriting is designed to target large AIGs. However, the largest test cases available in public benchmarks are three AIGs with *more than ten million* (MtM) nodes from EPFL benchmark suite. These synthetic circuits contain structures that are quite different from those found in practical designs. Therefore, we selected ten more realistic designs: seven largest AIGs among the arithmetic and three largest control circuits from EPFL benchmark [3].

In order to derive larger AIGs from these test cases, the ABC command *double* was applied 10 times to each design, except for the large "hyp" design, to which command *double* was applied 8 times. This command doubles the size of a given AIG by creating a copy of the original design. The test cases generated using command *double* are still synthetic but they are arguably more realistic than the MtM designs. In the real designs, synthesis tools are applied to a combinational logic cloud located between flip-flops in multiple design blocks. Since design blocks are often unrelated to each other, the resulting logic cloud may look somewhat similar to a set of copies of the same design used in the adopted benchmarks. Table 1 presents test cases derived by applying the command *double*.

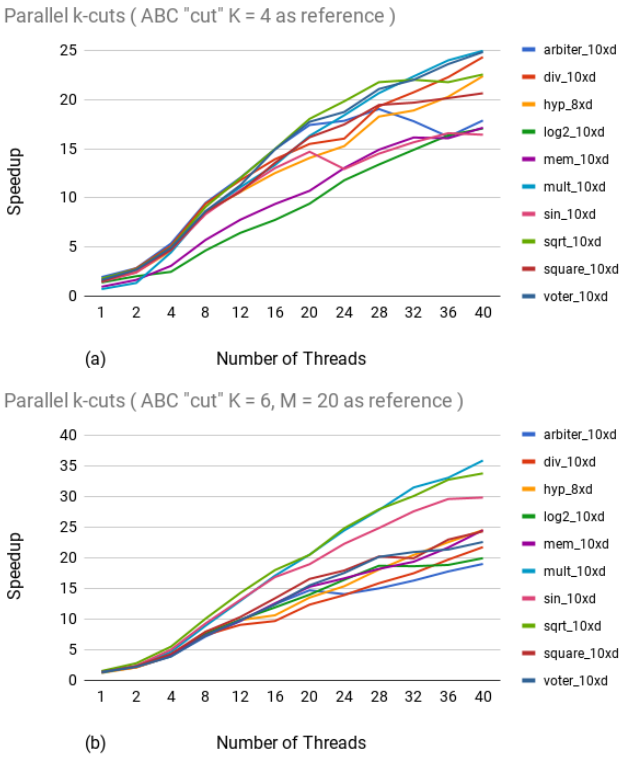

Figure 3: Scalability of parallel k-cut compared to ABC commands: (a) *cut, K=4* and (b) *cut, K=6*.

6.2 Parallel K-Cuts Scalability

First, we present an evaluation of the standalone parallel k-cut computation, since the cut enumeration is commonly used in other logic synthesis methods. In this experiment, the performance of the parallel k-cut enumeration is compared to command *cut* in ABC [1], using 4-input and 6-input cuts. The number of cuts per AIG node is limited to 20 for 6-input cuts and unbounded for 4-input cuts, *i.e., ABC commands: "cut -K 6 -M 20 -a -t -x" and "cut -K 4 -a -t -x"* . The remaining flags are used to disable some extra computations in order to perform a fair comparison between the methods.

The parallel k-cut computation is executed using a given number of threads, going from 1 to 40. Fig. 3(a) and Fig. 3(b) present the scalability of the 4-input and 6-input cut enumeration, respectively. The parallel k-cut enumeration, is up to 25x and 36x faster than the sequential version when using $k = 4$ and $k = 6$ respectively. It can be observed that the parallel k-cut enumeration scales reasonably well, which could also motivate its use in a parallel technology mapping.

6.3 Parallel AIG Rewriting Scalability

In the following experiments, we compare the proposed parallel AIG rewriting with commands *rewrite* and *drw* in

Table 2: Runtimes, in seconds, of the rewriting methods under comparison.

Benchmark	ABC *rewrite -l*	40 thr. *unbounded*	ABC *drw*	40 thr. *bounded*
sin_10xd	155.96	6.54	99.13	2.86
arbiter_10xd	227.73	8.17	177.90	6.89
voter_10xd	425.49	13.86	238.05	10.29
square_10xd	492.96	14.09	302.75	6.49
sqrt_10xd	784.64	32.82	1,368.16	22.18
mult_10xd	770.70	21.38	460.28	11.39
log2_10xd	1,077.12	34.37	602.86	17.31
mem_10xd	623.16	23.41	611.56	19.91
hyp_8xd	1,451.22	40.93	988.28	19.74
div_10xd	1,575.80	81.15	1,223.83	58.39

ABC. In most of the practical applications, rewriting is applied to the same AIG several times, but since we focus on comparing the parallel rewriting against the sequential one, a single pass is used in each run.

Command *rewrite* by default does not limit the number of cuts per node and the number of precomputed structures tried per cut. Therefore, we ran the parallel rewriting with these two parameters *unbounded*. Moreover, preserving logic level during rewriting was disabled using switch "-l", *i.e., rewrite -l*. To measure the parallel rewriting scalability as the number of threads increases, we ran the parallel method with 1 to 40 threads. Fig. 4(a) shows the parallel rewriting scalability. Rewriting with one thread is approximately 2x faster than command *rewrite* while rewriting with 40 threads is up to 36x faster. The proposed method running with one thread is faster than command *rewrite -l* mainly due to different graph representation and structural hashing organization. Table 2 presents the runtimes for ABC *rewrite -l* and for parallel rewriting with 40 threads *unbounded*.

Command *drw* is an improved version of command *rewrite*. One of the main differences is the larger set of precomputed structures used during rewriting. Moreover, *drw* accepts some parameters to trade off QoR and runtime, making it possible to limit the number of cuts per node and the number of precomputed structures per cut. In this experiment, both methods under comparison were limited (*bounded*) to eight cuts per node and five precomputed structures per cut, which are the default values in *drw*. We also ran both methods without preserving logic level and increasing the number of threads from 1 to 40. Fig. 4(b) shows that the parallel rewriting had super-linear speedups in some cases.

Overall, the rewriting *bounded* running with one thread is approximately 2.5x faster than the sequential command *drw* while rewriting with 40 threads is up to 50x faster. Since *drw* uses a more complete table of precomputed structures containing 222 Boolean functions, it is expected to have variations in both runtime and QoR when compared to the parallel method. This difference between algorithms justifies the super-linear speedups and the particular behavior in the scaling curve for the "sqrt_10xd" design, shown in Fig. 4(b). When processing "sqrt_10xd", *drw* algorithm performed many more attempts for replacing subgraphs, leading to a higher execution time. In this test case, the proposed method with one thread is approximately 6x faster than *drw*, reaching a speedup peak of 68x with 28 threads. The runtimes of command *drw* and parallel method with 40 threads *bounded* are presented in Table 2.

6.4 Parallel AIG Rewriting QoR

We also report the number of AIG nodes (*size*) and levels (*depth*) obtained at each execution, allowing for comparison the parallel rewriting with commands *rewrite* and *drw*

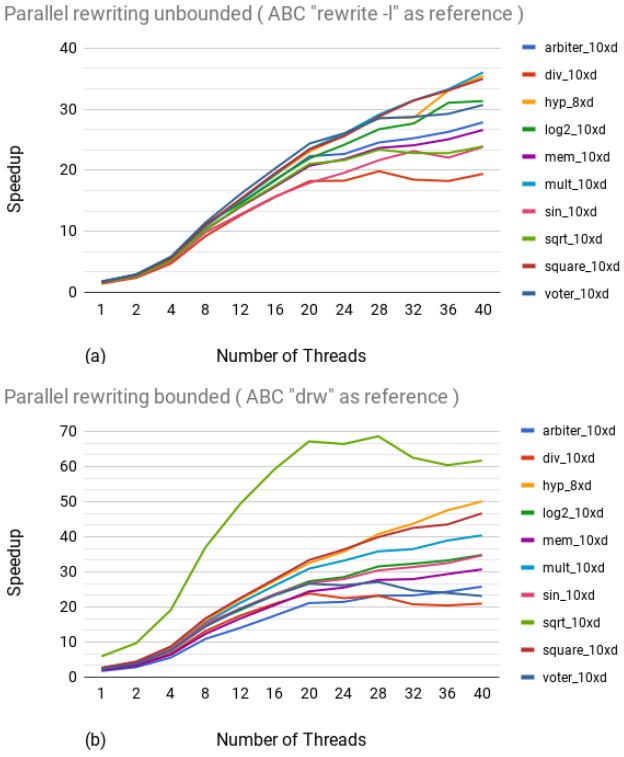

Figure 4: Scalability of parallel rewriting compared to ABC commands: (a) *rewrite -l* and (b) *drw*.

in terms of AIG *size* and *depth*. We consider the *size* and *depth* of the rewriting with 40 threads as the baseline because it produced the largest speedup. Table 3 shows that the parallel rewriting presents a small variation in the AIG *size* and *depth* when compared to command *rewrite* (reference method). As shown in Table 4, the command *drw* was able to reduce the AIG *size* and *depth* due to its more complete set of precomputed structures. It is expected that, using the same set of precomputed structures, the parallel rewriting and the command *drw* present the same quality in both the AIG *size* and *depth*.

6.5 QoR Variation

We performed previous experiments considering 10 different designs, 12 different thread counts and 5 executions of the same design for each thread count. This way, we produced 600 samples of runtime, AIG *size* and *depth*, i.e., 60 samples per design. Table 5 presents a comparison of QoR variation for many executions of parallel rewriting. For each design, we consider as the baseline the minimum *size* and *depth* observed among the 60 samples. We calculate the maximum percentage of variation from the minimum values. Table 5 shows that, the variations in AIG *size* have been negligible. Besides, variations in terms of AIG *depth* were zero for all the 600 executions. In summary, even when different thread counts are used, the proposed approach produces similar solutions in terms of AIG *size* and *depth*.

7. CONCLUSIONS

This paper introduces a set of principles that allow for the use of parallel computation in DAG-aware AIG rewriting. We redesign the data structures and the implementation of the traditional rewriting to match the Galois system. The experiments demonstrate that the proposed approach speeds

Table 3: QoR of rewriting with 40 threads *unbounded* over ABC command *rewrite -l*.

Benchmark	ABC *rewrite -l* size	ABC *rewrite -l* depth	40 thr. *unbounded* size	40 thr. *unbounded* depth
sin_10xd	5,465,088	223	1.00	1.00
arbiter_10xd	12,123,136	87	1.00	1.00
voter_10xd	11,553,792	63	0.99	1.06
square_10xd	18,803,712	250	1.00	1.00
sqrt_10xd	18,923,520	6,048	1.00	1.00
mult_10xd	27,551,744	274	1.00	1.00
log2_10xd	32,304,128	443	1.00	1.00
mem_10xd	47,885,312	114	1.00	1.00
hyp_8xd	54,854,144	24,801	1.00	1.00
div_10xd	42,138,624	4,406	1.00	1.00

Table 4: QoR of rewriting with 40 threads *bounded* over ABC command *drw*.

Benchmark	ABC *drw* size	ABC *drw* depth	40 thr. *bounded* size	40 thr. *bounded* depth
sin_10xd	5,313,053	223	1.04	1.01
arbiter_10xd	12,123,136	87	1.00	1.00
voter_10xd	10,640,603	68	1.12	1.07
square_10xd	18,184,830	250	1.04	1.00
sqrt_10xd	18,924,544	6,048	1.00	1.00
mult_10xd	25,355,760	273	1.08	1.00
log2_10xd	30,475,035	423	1.05	1.05
mem_10xd	47,401,984	115	1.01	0.99
hyp_8xd	54,566,159	24,801	1.01	1.00
div_10xd	42,186,609	4,406	1.00	1.00

up logic synthesis without sacrificing QoR, when compared to the traditional rewriting. The proposed approach scales to many cores when applied to AIGs composed of millions of nodes. Our implementation extends to other state-of-the-art techniques, including technology mapping. Moreover, our approach can be used in the context of stochastic synthesis and technology mapping similar to Liu *et al.* [14].

ACKNOWLEDGEMENT

This research was partially supported by the Brazilian Funding Agencies CNPq and CAPES-PDSE and by NSF grants 1337217, 1337281, 1406355, 1618425, 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

References

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification.* http://www-cad.eecs.berkeley.edu/~alanmi/abc.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli. "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization". In *Proc. DAC'14*.

[3] L. Amarú, P.-E. Gaillardon, and G. De Micheli. "The EPFL Combinational Benchmark Suite". In *Proc. IWLS'15*.

[4] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P. E. Gaillardon, J. Olson, R. Brayton, and G. D. Micheli. "Enabling exact delay synthesis". In *Proc. ICCAD'17*, pp. 352-359.

[5] P. Bjesse and A. Boralv. "DAG-aware circuit compression for formal verification". In *Proc. ICCAD'04*.

[6] R. Brayton and C. McMullen. "The Decomposition and Factorization of Boolean Expressions". In *Proc. ISCAS'82*, pp. 29-54.

Table 5: QoR variation, considering 600 executions of the parallel AIG rewriting *unbounded*.

Benchmark	*size* min	*size* max var.	*depth* min	*depth* max var.
sin_10xd	5,460,989	0.0001%	223	0%
arbiter_10xd	12,123,136	0.0000%	87	0%
voter_10xd	11,387,187	0.0124%	67	0%
square_10xd	18,775,062	0.0005%	250	0%
sqrt_10xd	18,923,520	0.0000%	6,048	0%
mult_10xd	27,520,256	0.0011%	274	0%
log2_10xd	32,302,022	0.0001%	443	0%
mem_10xd	47,884,313	0.0005%	114	0%
hyp_8xd	54,859,255	0.0000%	24,801	0%
div_10xd	42,154,768	0.0009%	4,407	0%

[7] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam. "Reducing structural bias in technology mapping". *IEEE Trans. on Comput.-Aided Design of Integr. Circuits and Syst.*, 25(12), 2006.

[8] J. Cortadella. "Timing-driven logic bi-decomposition". *IEEE Trans. on Comp.-Aided Design of Integr. Circuits and Syst.*, 22(6):675–685, 2003.

[9] M. Elbayoumi, M. Choudhury, V. Kravets, A. Sullivan, M. Hsiao, and M. Elnainay. "TACUE: A timing-aware cuts enumeration algorithm for parallel synthesis". In *Proc. DAC'14*.

[10] W. Haaswijk, M. Soeken, L. Amarù, P. E. Gaillardon, and G. D. Micheli. "A novel basis for logic rewriting". In *Proc. ASP-DAC'17*, pp. 151-156.

[11] A. Lenharth and K. Pingali. "Scaling Runtimes for Irregular Algorithms to Large-Scale NUMA Systems". *Computer*, 48(8):35–44, 2015.

[12] A. Lenharth, D. Nguyen, and K. Pingali. "Parallel graph analytics". *Comm. of the ACM*, 59(5), 2016.

[13] N. Li and E. Dubrova. "AIG rewriting using 5-input cuts". In *Proc. ICCD'11*, pp. 429-430.

[14] G. Liu and Z. Zhang. "A Parallelized Iterative Improvement Approach to Area Optimization for LUT-Based Technology Mapping". In *Proc. FPGA'17*.

[15] A. Mishchenko and R. K. Brayton. "Scalable logic synthesis using a simple circuit structure". In *Proc. IWLS'06*.

[16] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. "Combinational and Sequential Mapping with Priority Cuts". In *Proc. ICCAD'07*, pp. 354-361.

[17] A. Mishchenko, S. Chatterjee, and R. Brayton. "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis". In *Proc. DAC'06*, pp. 532-535.

[18] Y. O. M. Moctar and P. Brisk. "Parallel FPGA routing based on the operator formulation". In *Proc. DAC'14*.

[19] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. "The tao of parallelism in algorithms". In *ACM Sigplan Notices*, volume 46 of number 6, 2011.

[20] M. Soeken, L. G. Amarù, P. E. Gaillardon, and G. D. Micheli. "Optimizing Majority-Inverter Graphs with functional hashing". In *Proc. DATE'16*, pp. 1030-1035.

[21] L. Stok. "Developing Parallel EDA Tools [The Last Byte]". *IEEE Design Test*, 30(1):65–66, 2013.

[22] L. Stok. "The Next 25 Years in EDA: A Cloudy Future?" *IEEE Design & Test*, 31(2), 2014.

[23] W. Yang, L. Wang, and A. Mishchenko. "Lazy man's logic synthesis". In *Proc. ICCAD'12*, pp. 597-604.