Parallel Combinational Equivalence Checking

Vinicius Possani[1], Alan Mishchenko[2], Renato Ribas[1], Andre Reis[1],
[1]Institute of Informatics, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil.
[2]Department of EECS, University of California, Berkeley, USA.

***Abstract*—Combinational equivalence checking (CEC) is widely used to ensure design correctness after logic synthesis and technology-dependent optimization in digital IC design. The runtime of CEC is often critical for large designs, even when advanced techniques are employed. Three complementary ways for enabling parallelism in CEC are proposed, addressing different design and verification scenarios. Experimental results have demonstrated speedups up to 63x when compared to a single-threaded implementation of similar CEC engine. The practical impact of such a speedup represents a runtime reduction from 19 hours to only 18 minutes. Therefore, the proposed solution presents great potential for improving current EDA environments.**

***Index Terms*—Digital IC, Verification, Combinational Equivalence Checking, Graph Partitioning, Parallel Computing.**

I. Introduction

Fast and scalable techniques for combinational equivalence checking (CEC) are essential in modern electronic design automation (EDA) environments. In a typical scenario, the logic function implemented by an optimized digital integrated circuit (IC) is checked for equivalence against the original specification after multi-level logic synthesis [1]. Moreover, scalable CEC techniques are quite useful in several key logic synthesis process, which depend on efficient computation of equivalence classes of internal circuit nodes. A non-exhaustive list of these tasks is the following:

- Removal of functionally equivalent logic in the design during logic synthesis and optimization;
- Computation of structural choices, enabling circuit area and signal delay improvement after technology mapping step [2];
- Sequential equivalence checking based on register and signal correspondence [3], [4];
- Bridging circuits for the implementation and the specification in engineering change orders (ECOs) [5];
- A number of utility packages, *e.g.* node name transfer across netlists before and after synthesis.

In recent years, equivalence checking has become more critical due to the increasing in complexity of current and upcoming system-on-chip (SoC) and VLSI designs. To demonstrate the complexity of CEC when dealing with large designs, consider one instance of the problem for checking equivalence between the original specification comprising 14 million nodes and a synthesized version comprising 9 million nodes when both are represented using AND-inverter graphs (AIGs). The circuits were checked using command $\&cec$ in ABC, which is an industrial-strength academic tool for logic synthesis and formal verification [6]. ABC took more than 24 hours to prove equivalence, making it clear that verification becomes harder as the design size increases, reinforcing the need for improved CEC to make computer-aided design (CAD) tools more scalable.

In order to enable the next rounds of technology innovation, there is a demand for massively parallel EDA tools running on the cloud [7], [8]. Considering this scenario, traditional EDA algorithms and data structures, in particular, those dealing with CEC, need to be rethought to benefit from parallel computing platforms. In typical EDA environment, algorithms work on a sparse graph representing the circuit. It is often challenging to exploit parallelism of graph-based algorithms due to the irregular nature of graphs implemented using pointer-based data structures. Besides, several CEC approaches are based on BDDs [9]–[11] and hybrid BDD/SAT-based engines [12], which are less scalable when running into single thread and so harder to parallelize than those based on simulation and Boolean satisfiability (SAT) [13], [14].

In [15], a different method called EQUIPE was propposed for parallelizing CEC based on a hybrid solution that combines CPU and GPU. The authors exploit parallelism in GPU to perform signature-based analysis and structural matching in order to minimize the number of SAT calls. The method evaluates the circuits under verification trying to prove speculatively the equivalence between internal nodes. When the equivalence cannot be proved by the GPU-based solution, a SAT solver is then executed in the host processor (CPU) to check equivalence of individual nodes. However, even when using 14 GPU-cores and 4 CPU-cores, the speedup provided by EQUIPE is limited to a factor of three compared to the non-parallel CEC engine in ABC tool.

In this paper, we are unlocking massive parallelism for CEC by applying graph (miter) partitioning to balance data sharing and data independence during SAT solving. The paper proposes two models of parallelism that can be applied separately or combined, leading to a third hybrid model that allows to fully exploit the power of parallel environments. Our parallel CEC is based on the state-of-the-art CEC engine available in ABC, which exploits the synergy between logic simulation and SAT [14]. The main contributions of this work are the following:

- Three novel models are introduced to enable massive parallelism for speeding up two crucial time-consuming CEC tasks, mitering and SAT sweeping.
- The proposed parallel CEC engine handles large designs comprising millions of AIG nodes, and scales to many threads unlocking the potential of parallel environments available through cloud computing.
- Experimental results showed significant runtime improvement when compared to both single-threaded ABC and parallel commercial CEC engines. In some cases, the

proposed solution reduces the CEC runtime from more than one day to only a few minutes/hours.

- Proposed models are based on simple principles making them easy to reproduce and deploy in a standard EDA flow. Therefore, several other important tasks that depend on CEC, can be benefited from the speedup enabled by the proposed scalable solution.

The rest of this paper is organized as follows. Section II presents a brief review of techniques used in modern CEC engines. Section III describes the proposed approach for accelerating CEC. Experimental results are discussed in Section IV. The conclusions are outlined in Section V.

II. Preliminaries

A. AND-Inverter Graph

AND-inverter graph is a directed acyclic graph used as data structure in logic synthesis. AIG is a homogeneous circuit representation comprising four types of nodes: constants, primary inputs (PI), primary outputs (PO) and two-input AND (AND2) operators. Sequential elements such as latches and flip-flops can be viewed as special nodes or pseudo-PI/PO. A graph edge can present an optional complemented attribute depending on whether the corresponding signal has an inverter.

The set of nodes connected to the inputs of a given AIG node n is called the *fanin* of n. Analogously, the set of nodes connected to the outputs of n is called the *fanout* of n. If there is a path from node n to n', then n is in the *transitive fanin (TFI)* of n' and n' is in the *transitive fanout (TFO)* of n. The *TFI cone* of a given node n includes n and all those nodes in the transitive fanin of n towards the primary inputs of the AIG. The *TFO cone* of n is analogous, including n and all those nodes in the transitive fanout of n towards the primary outputs of the AIG [16].

B. Boolean Satisfiability

Boolean satisfiability is the decision problem to determine whether there exist an assignment to the input variables that makes the output of a given Boolean formula evaluates to $true$. If such an assignment exists, then the formula is $satisfiable$ (sat). Otherwise, the formula is $unsatisfiable$ ($unsat$). Conventionally, SAT problem instances are represented by a formula in the conjunctive normal form (CNF). SAT solvers are tools implementing advanced techniques for deciding whether a given SAT instance is sat or $unsat$ [17]. Several techniques are used for speeding up SAT solving such as the cube-and-conquer method [18], the dual SAT solving [19] and others. Many practical problems can be modeled using SAT and efficiently solved by modern SAT solvers.

C. Mitering

A typical CEC engine transforms two circuits under verification into a single circuit called *miter* [20]. The miter is constructed by pair-wise connecting the inputs with the same name and by pair-wise comparing the outputs with the same name using Exclusive-OR (EXOR) operator. At the top of the miter, an OR operator connects all the outputs of EXORs, representing the output of the comparator for the primary outputs.

In practice, a miter can be represented by a single AIG, where each output is a separate decision problem. The AIG is viewed as an instance of a multi-output SAT problem that can be easily converter to CNF form by using the Tseitin transformation [21], and then given to the SAT solver. If the SAT solver returns $unsat$, all pairs of outputs under comparison are equivalent. Otherwise, if the solver returns sat, it means that at least one pair of outputs is different. This process for proving equivalence using miter and SAT solving is also referred as *mitering*.

D. BDD and SAT Sweeping

The techniques known as *BDD sweeping* [12] and *SAT sweeping* [22] are used to detect functionally equivalent nodes internally in AIG or miter. In this approach, pairs of internal nodes are checked for equivalence in a topological order. If the equivalence is proved, the corresponding nodes can be merged to simplify the miter. BDD and SAT sweeping are useful since to prove equivalence by directly constructing a BDD of a miter for the entire circuits under verification is often not practical [23]. In many cases, BDD construction and SAT solving for all the outputs require a lot of memory and computation resources.

E. Logic Simulation

Logic simulation is a common technique used to quickly detect non-equivalent nodes, helping to reduce the number of SAT calls during SAT sweeping. Typically, random vectors and counter-examples returned by SAT solvers are used as input patterns for simulation [13], [22]. A *counter-example* is an assignment of input variables of the Boolean formula representing the SAT problem instance, proving that a pair of nodes is not equivalent. Simulation is used to group potential-equivalent nodes in classes and SAT solving is used for checking equivalences among nodes in the same class.

III. Proposed Parallel CEC

We propose three different strategies to speedup CEC by enabling massive parallelism in the ABC $\&cec$ engine. As discussed before, the conventional CEC solves many SAT problems represented as a miter. The proposed approach relies on graph partitioning to exploit data independence during miter simplification and solving.

First of all, we dissect the CEC engine in ABC tool, presenting its main components and investigating the most promising points to exploit parallelism. In the sequence, we define the proposed strategy for graph partitioning. In the next, we propose two models for enabling parallel CEC by partitioning the main miter and the temporary miters constructed during SAT sweeping. Finally, we present how both strategies can be combined in a hybrid model that allows to fully exploit the power of parallel computing.

A. CEC Engine in ABC Tool

Typically, modern CEC engines alternate between miter solving to prove equivalences for outputs and miter simplification to prove equivalence for internal nodes. Simulation and SAT sweeping are used for gradually simplifying the miter complexity. Therefore, for the sake of simplicity, we

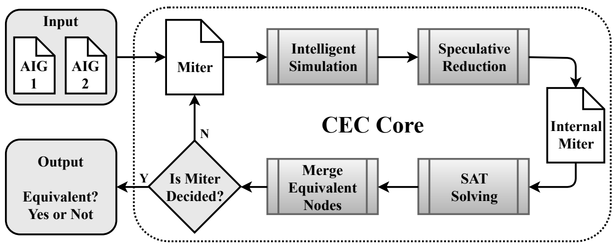

Fig. 1. CEC engine of ABC command $\&cec$.

are adopting the terms *main miter* and *internal miter* when referring to the data being processed in different steps of the CEC engine core.

- The *main miter* refers to the miter created once at the beginning of the verification process. This miter comprises all the logic of the two circuits under comparison. During the CEC, the main miter is gradually simplified by merging internal equivalent nodes from both circuits.
- The *internal miter* refers to the miter temporarily created at each integration of SAT sweeping in the CEC core. This miter comprises subgraphs from the main miter, which represent a set of CEC subproblems for determining those internal equivalent nodes.

Fig. 1 presents a scheme of the CEC core implemented in the ABC command $\&cec$ [6]. This command implements an improved version of the method proposed in [14]. This engine is based on the SAT solver *MiniSat* [17] and employs several techniques introduced in previous work such as *intelligent simulation* and *functionally reduced AIGs (FRAIGs)* [24] .

Initially, intelligent simulation is used to group nodes that appear to be equivalent into classes. The input patterns used for simulation are computed based on SAT counter-examples that contribute to distinguish non-equivalent nodes belonging to candidate equivalence classes. An extended set of *distance-1* simulation vectors are produced by flipping one bit at a time from the counter-example [14].

While simplifying the miter in $\&cec$ command, pairs of nodes from each class are represented in the temporary internal miter and checked by the SAT solver. All those nodes proved equivalent are merged in order to simplify the main miter under verification and the counter-examples are used to disprove equivalences in further iterations. This iterative process of miter refinement and checking is executed until: (i) the miter is fully solved (proved $unsat$); (ii) a counter-example is detected (proved sat); or (iii) a resource limit is reached. We have observed that the most advantageous strategy to accelerate the ABC command $\&cec$ is to apply main and internal miter partitioning for enabling parallel SAT solving.

B. Graph Partitioning

The SAT problem arising in the proof of equivalence between pairs of POs or internal nodes are intrinsically independent of each other. These problems are encoded together in the same graph due to the natural logic sharing introduced during logic synthesis and optimization. We can exploit such an intrinsic independence of the problems to solve them in separate batches. Since each AIG output represents a SAT problem, we have three main motivations for performing the graph partitioning based on POs:

- The SAT problems for checking pairs of POs or internal nodes are independent of each other.
- We have empirically observed that adjacent POs tend to present more shared logic than randomly selected groups of POs because RTL elaborators that translate word-level design description into bit-level circuit, place bit-level flops next to each other.
- The set of SAT problems formulated for checking equivalent classes during SAT sweeping are encoded as subsequent POs in the internal miter.

Miter partitioning provides a trade-off between data sharing and data independence during equivalence checking. On the one extreme side, we have several SAT problems encoded into a single miter, which can be incrementally solved in the same SAT solver by exploiting shared clauses (all together). On the other extreme side, we can apply graph partitioning to extract the TFI cones related to the pairs of outputs under comparison and check each pair using independent SAT calls (all separated). In order to achieve an equilibrium between data sharing and data independence, we are proposing to create relatively large partitions comprising a subset of SAT problems. The partitioning enables to solve the subset of SAT problems in parallel by independent SAT solver instances. Algorithm 1 presents a top-level view of the graph partitioning procedure.

Initially, Algorithm 1 checks whether it is feasible to decompose the given miter (AIG) into the desired number of partitions N, as shown in lines 6-12. This checking ensures that the graph partitioning will be consistent by assigning feasible values to the number of partitions N and partition size S. When the division of $miter.nPO$ by N produces a remainder R, the algorithm distributes the set of remaining POs by placing one more PO to the first R partitions. This distribution contributes to produce a workload balance among partitions.

In line 15 of Algorithm 1, all partitions start empty and the transitive fanin cones of subsequent POs are gradually appended to the current partition until the partition size is reached, as shown in the loop of lines 17-25. The routine *appendTFICone* collects all the logic that a given PO depends on towards the PIs, recursively. The proposed graph partitioning uses structural hashing and preserves logic sharing inside the partitions with some logic duplication among the partitions. However, since the CEC is a decision problem, logic duplication does not affect the solution quality. In other words, the CEC engine is not sensitive to logic duplication unlike other optimization problems, such as multi-level logic optimization and technology mapping.

The proposed graph partitioning guarantees the soundness and completeness by ensuring that, for each output pair to be checked for equivalence in a given partition, the partition also contains all the logic needed to prove or disprove equivalence (soundness) and each output pair belongs to some partition (completeness). The same graph partition can be applied for both main miter and internal miter partitioning. The soundness and completeness properties holds for both cases, since internal miter comprises smaller instances of the same problem represented in the main miter.

Algorithm 1: Top-level view of graph partitioning

```
Function GraphPartitioning()
    input: $N$ (number of partitions), $miter$ ( AIG )
    output: $P$ ( partitions )
    $int\ S$; // partition size
    $int\ R$; // division reminder
    if ( $miter.nPO \geq N$ ) then
        $S = miter.nPO$ / $N$;
        $R = miter.nPO$ % $N$;
    else
        $S = 1$;
        $R = 0$;
        $N = miter.nPO$ ;
    if ( $R > 0$ ) then
        $S++$; // to treat remaining POs
    $AIG * P$ [ $N$ ]; // partitions as an array of AIGs
    $int\ i = 0, j = 0$;
    for each po in miter do
        appendTFICone( po, miter, P[ i ] );
        $j++$;
        if ( j == S) then
            $i++$; // move to the next partition
            $j = 0$; // reset the counter
            // if all remaining POs were processed
            if ( $--R == 0$ ) then
                $S--$; // go back to the original size
    return $P$;
```

C. Main Miter Partitioning

We are introducing the *Model P*, depicted by the scheme of Fig. 2, to decompose the main miter into a set of P independent sub-problems (partitions) and to verify them in parallel. This model receives the pair of circuits under verification and pre-process them by applying mitering construction and the graph partitioning introduced in Algorithm 1. The graph partitioning delivers a list of AIGs representing the miter partitions, which are mapped to threads by using the POSIX Threads standard (*Pthreads*). Each thread receives a miter partition as argument and checks this miter by executing the CEC core illustrated in Fig. 1. By applying the proposed graph partitioning, the *Model P* is able to explore massive parallelism where each thread solves part of the problem without any dependencies and conflicts to other threads.

Finally, all threads are joined to the main thread and the intermediate solutions are combined to deliver the final solution. If all the threads return the answer *equivalent*, then it means that the whole circuits under verification are equivalent. However, when at least one pair of outputs is proved *non-equivalent*, the *Model P* enables earlier termination to the verification process by exploring the solution space quickly. For instance, if a given thread proves that at least one pair of POs is non-equivalent, then, that thread can report the input/output patterns and broadcast a stop signal to other threads.

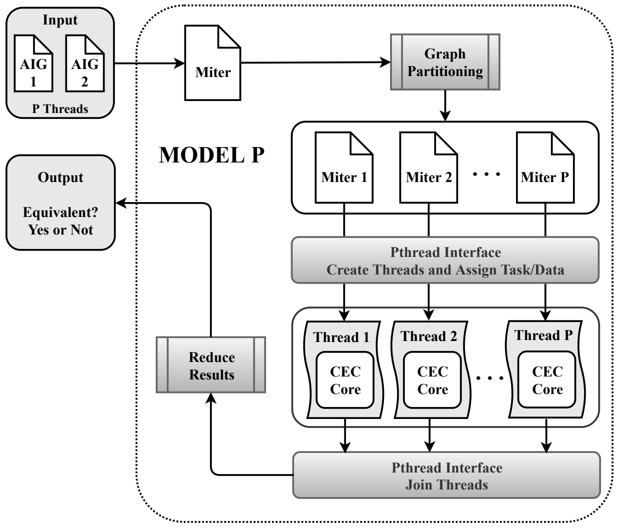

Fig. 2. Scheme of Parallel CEC by main miter partitioning.

The proposed approach lies closer to the middle of the spectrum between data sharing and data independence, enabling faster equivalence checking. This solution works like a divide-and-conquer approach to decompose a problem into smaller ones, enabling efficient parallel processing on multi-core platforms with shared memory. Moreover, since miters are completely independent to each other, the proposed approach can be easily extended to exploit the advantages of distributed computing in cloud.

D. Internal Miter Partitioning

We are introducing the *Model S*, depicted by the scheme of Fig. 3, to speed up SAT sweeping while checking equivalences of internal nodes in the CEC core. The same principle of graph partitioning presented in Algorithm 1 can be applied to decompose the internal miter constructed during SAT sweeping. After applying internal miter partitioning, the *Pthreads* interface is used to launch S SAT solver calls for checking equivalences of internal nodes in parallel, as illustrated in Fig. 3. In this approach, each thread receives a partition of the internal miter and executes an independent instance of the Minisat [17] for checking the miter.

Notice that the proposed approach interleaves serial and parallel sections of the code at each iteration, as shown in Fig. 3. Therefore, before starting the next iteration of the CEC core loop, all threads must finish and return partial results. For each iteration, a set of SAT counter-examples is collected from several SAT calls to produce simulation vectors for the next iteration. Moreover, the pairs of nodes proved to be equivalent are collected to refine the main miter. Therefore, auxiliary routines are used for reducing back all these partial results representing counter-examples and equivalent nodes. This way, the runtime of the latest thread defines the delay before moving to the next iteration of CEC core.

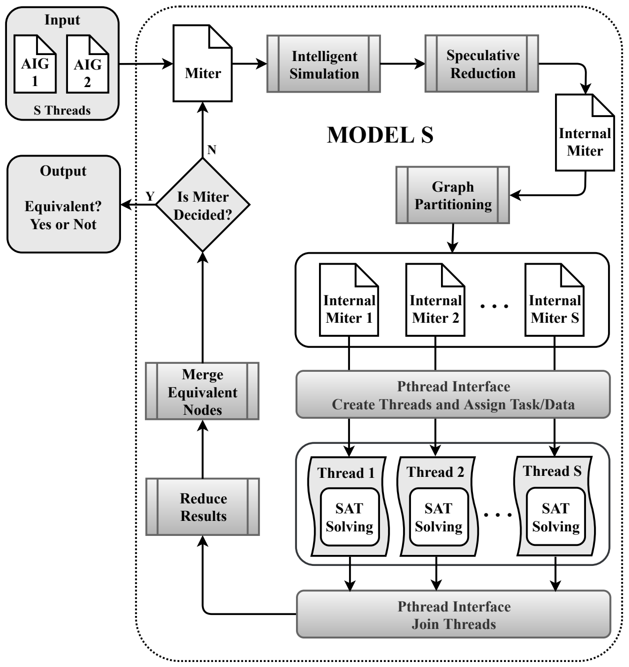

Fig. 3. Scheme of Parallel CEC by internal miter partitioning.

The amount of SAT problems encoded in the internal miter and the complexity of these problems are strongly dependent on the design structure. Typically, the main miter comprises several complex SAT problems whereas the internal miter comprises a lot of small and easy SAT subproblems. Considering this scenario, it is a challenge to find a good trade-off between data sharing and data independence of internal miter to perform a partitioning that leads to a good workload balance among the threads.

It is worth to mention that several other applications of SAT sweeping can be benefited from the proposed *Model S*. In optimization problems based on SAT sweeping, such as redundancy removal and signal correspondence [25], it is not wise to partition the input graph because optimization opportunities tend to be lost due to the negative bias of partition boundaries. Therefore, in such optimization problems, the proposed *Model P* is less useful. On the other hand, the proposed *Model S* can enable parallelism to solve internal problems encoded in the temporary miter, without adversely impacting the quality of results of such optimization problems.

E. Combined Miter Partitioning

We introduced the models P and S for speeding up equivalence checking by exploiting parallelism on different levels of the CEC engine. A strong feature of our approach is that both models can work together, cooperating to improve the CEC runtime. It means that each thread created in *Model P* can create a set of subthreads in *Model S* to speedup the computation. The amount of threads working at each model can be specified by the parameters P and S. This way, based on the design characteristics, one can fine tune the engine to get the best thread workload balance.

Since the amount of parallelism available in the main miter and internal miter partitioning is strongly related to the design structure, the ability to run *Model P* and *Model S* together helps to deal with a wide range of design characteristics. It has been confirmed in our experiments where the combined execution of models P and S achieves higher speedups, leading to the best results for some designs. Moreover, when considering massively parallel environments used in cloud computing, both models allow for a synergistic integration to fully exploit the power of distributed and shared memory computing. For instance, *Model P* can be used to partition the initial problem and distribute tasks to many nodes in the cloud whereas *Model S* can exploit the potential of multi-core architectures at each computing node.

IV. Experimental Results

The proposed methods are implemented in C programming language using *Pthreads* and incorporated in the ABC command $\&cec$ [6]. The three ways of parallelizing CEC proposed in Section III can be enabled in the $\&cec$ using the switches $-P$ and $-S$, together or separately, followed by the desired number of threads applied in each model. In the following experiment, the proposed parallel CEC is compared to a commercial verification tool. In the sequence, we present a scaling analysis of the proposed models in a massive parallel environment when compared to the reference single-threaded method implemented in ABC command $\&cec$.

A. Benchmarks

Since this works are focusing on speeding up CEC for large designs, we selected the three circuits comprising more than ten million (MtM) AIG nodes from EPFL benchmark suite [26]. These circuits comprise AIGs with sixteen, twenty and twenty three million nodes, representing random Boolean functions with complex implementation cost. Additionally, we applied the ABC command *double* 10 times for six other circuits from this suite to generate large AIGs. This command doubles the AIG size by creating two copies placed side by side, each one with its own primary inputs and primary outputs. The circuits generated using the *double* command are synthetic but they are very similar to the combinational logic cloud extracted from a heavily pipelined industrial design, in which pipeline stages are represented as different copies of the same design. All the circuits considered in our experiments are large enough to present challenges for CEC engines. Table I presents the MtM AIGs from EPFL benchmark and the circuits derived by applying the *double* command resulting in AIGs with millions of nodes.

In a typical scenario, CEC is used to check equivalence between original and optimized versions of the same design after logic synthesis and/or technology-dependent optimization. To reproduce this scenario, we first applied script *dc2* in ABC to the designs in Table I. The script performs area-driven delay-constrained multi-level optimization using *rewriting*, *balancing* and *refactoring* algorithms. In the following experiments, we check the equivalence between the original designs and the *dc2*-optimized ones using a commercial verification tool, the original single-threaded version of ABC command $\&cec$ and the proposed models for parallel CEC.

B. Comparing to Commercial Verification Tool

In this experiment we are comparing the proposed models for parallel CEC to a commercial verification tool. The results

TABLE I
THE SET OF SIX CIRCUITS OBTAINED BY APPLYING THE ABC COMMAND *double* 10X AND THE THREE MTM AIG NODES CIRCUITS FROM THE EPFL BENCHMARK SUITE [26].

Design	# PI	# PO	# AND2	# 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
sixteen	117	50	16,216,836	140
twenty	137	60	20,732,893	162
twentythree	153	68	23339737	176

TABLE II
RUNTIME COMPARISON AMONG THE COMMERCIAL TOOL AND THE PROPOSED MODELS RUNNING AT FOUR THREADS, IN (H:M:S).

Design	Commercial (4 threads)	-P 4	-S 4	-P 2 -S 2	Speedup
sin_10xd	1:02:09	**0:07:09**	0:11:15	0:08:29	8.68x
arbiter_10xd	0:51:53	**0:00:51**	0:01:36	0:01:06	60.56x
voter_10xd	**1:19:37**	4:19:08	4:54:53	4:09:05	0.32x
square_10xd	2:57:02	**0:03:28**	0:07:45	0:04:52	51.13x
sqrt_10xd	69:03:25	**3:31:50**	6:47:06	6:38:47	19.56x
mult_10xd	5:20:41	**0:09:04**	0:20:15	0:12:46	35.34x
sixteen	21:21:26	4:22:54	3:34:31	**1:45:06**	12.19x
twenty	38:08:40	3:35:22	6:27:38	**2:19:40**	16.39x
twentythree	49:46:06	3:56:46	8:11:20	**2:52:52**	17.27x

were collected in a server with 64GB of shared RAM and a processor Intel®Core®i7-7700K CPU operating at 4.20GHz, where the processor has four physical cores. The tools under comparison were executed in a 64-bit Linux distribution and the runtimes were measured using the Linux bash command $time$ (real).

Both the commercial tool and the proposed models were executed using four threads to exploit the potential of the four physical cores available on the server. We execute Model P and Model S separately and combined. Table II presents the absolute runtimes for each method in the format ($hours : minutes : seconds$). The last column of the Table II presents the best speedups provided by the proposed approach when compared to the commercial tool. The experimental results have demonstrated that the proposed models for parallel CEC present significant smaller runtimes than the commercial verification tool, when running at the same number of threads. Notice that, for several cases, the commercial tool took more than a day for verifying the designs, whereas the proposed approach took only few minutes/hours.

C. Scaling Analysis Comparing to ABC command &cec

In order to asses the scalability of the proposed parallel models to many threads, a set of experiments has been carried out using a server with 128GB of shared RAM and four processors Intel®Xeon®CPU E7- 4860 operating at 2.27GHz, where each processor has 10 physical cores, corresponding the total of 40 cores. ABC tool was compiled using GNU g++ version 6.1.0 and executed in a 64-bit Linux distribution. The runtimes were measured using the Linux bash command $time$ (real). Unfortunately, we do not have a commercial tool available on this server with greater computing power. Therefore, in the following experiments, we are comparing the proposed parallel CEC approach to the state-of-the-art serial CEC from ABC.

In the first experiment, we compare the proposed models P and S separately. The goal of this experiment is to measure the speedup and the scalability that each model can bring to the verification process as the thread count increases. Table III presents the runtime for the original single-thread CEC and for parallel models P and S running at 4 threads up to 40 threads. The speedup introduced by each model according to the thread count is shown in Fig. 4 and in Fig. 5.

Regarding the benefits of the proposed approaches, *Model P* leads to more significant improvement than *Model S*. For most circuits, *Model P* works 30x faster than the single-thread CEC. Moreover, for the circuits ”twenty” and ”twentythree”, we have observed super-linear speedups of 60.73x and 63.08x, respectively. In this cases, graph partitioning enabled a good equilibrium between data sharing and data independence. Besides that, the CEC engine is based on incremental SAT solving by sharing clauses among successive SAT calls [14], [17]. Therefore, miter partitioning can lead to a better incremental behavior in the SAT solving heuristics, since each thread is applied to a smaller set of problems using separate SAT solvers.

It is harder to get high speedups in *Model S* because many smaller SAT problems are created in the interleaved sections of sequential and parallel code in the CEC core. Actually, the SAT problem size and complexity depends on the characteristics of the designs under verification. Therefore, we should not discard *Model S* since it can be advantageous for certain designs, and it can also be combined to work together with *Model P*.

By combining models P and S, we enable extra opportunities to trade-off data sharing and data independence in the CEC engine. In this experiment, we consider three different thread configurations for combining both models using the total of 40 threads. In the first configuration ($-P$ 4 $-S$ 10), we consider fewer partitions of the main miter and more partitions of the internal miter. In the second configuration ($-P$ 10 $-S$ 4), we swapped the thread count between main miter and internal miter partitioning. In the last configuration ($-P$ 20 $-S$ 2), we increased the thread count for the main miter partitioning since it has presented the best results in previous experiments. The absolute runtime values of each configuration are presented in Table IV and the respective speedups are shown in Fig. 6.

To demonstrate the advantages of combining both models, consider the results for designs ”voter_10xd” and ”arbiter_10xd” shown in Fig. 6. The configuration ($-P$ 10 $-S$ 4) resulted in extra speedups for both designs when compared to the models P and S running independently. For ”arbiter_10xd”, we observed the speedup of 6.7x whereas in the previous experiments the speedups were 4x for *Model P* and 1.5x for *Model S*. Moreover, for ”voter_10xd”, we observed the speedup of 39x which is practically linear (optimal) in the number of threads. In the previous experiments, the speedups for ”voter_10xd” were 37x and 23x for models P and S, respectively.

TABLE III
RUNTIME COMPARISON AMONG THE ORIGINAL ABC COMMAND &*cec* AND THE PROPOSED MODELS *P* AND *S* RUNNING SEPARATELY, IN (H:M:S).

Design	ABC &*cec*	-P 4	-S 4	-P 10	-S 10	-P 20	-S 20	-P 30	-S 30	-P 40	-S 40
sin_10xd	1:04:52	0:15:33	0:23:56	0:06:28	0:16:27	0:03:09	0:12:55	0:02:12	0:11:48	**0:02:05**	0:12:14
arbiter_10xd	0:05:17	0:01:35	0:03:36	**0:01:18**	0:03:18	0:01:38	0:03:16	0:02:08	0:03:22	0:02:41	0:03:28
voter_10xd	24:13:00	5:56:16	7:34:12	2:51:05	3:25:12	1:17:48	1:46:06	0:50:16	1:12:14	**0:39:12**	1:02:42
square_10xd	0:33:53	0:07:58	0:17:28	0:03:23	0:14:23	0:01:56	0:13:02	0:01:28	0:12:41	**0:01:16**	0:12:56
sqrt_10xd	21:34:04	7:45:45	14:17:51	4:05:54	12:57:21	2:18:04	12:19:09	1:34:57	12:09:34	**1:21:22**	12:10:35
mult_10xd	1:21:10	0:19:37	0:43:36	0:08:27	0:36:30	0:04:35	0:33:28	0:03:25	0:32:42	**0:03:03**	0:33:26
sixteen	8:27:33	7:07:47	7:17:48	1:43:48	7:49:08	0:29:03	7:08:11	0:15:21	7:10:25	**0:13:28**	7:24:37
twenty	14:35:59	4:14:36	14:53:13	3:48:59	13:39:38	0:27:43	14:32:43	0:15:27	13:55:20	**0:14:25**	13:50:52
twentythree	18:51:30	5:08:32	17:50:37	2:01:59	17:09:33	0:50:28	18:41:32	0:33:01	18:46:43	**0:17:56**	17:03:48

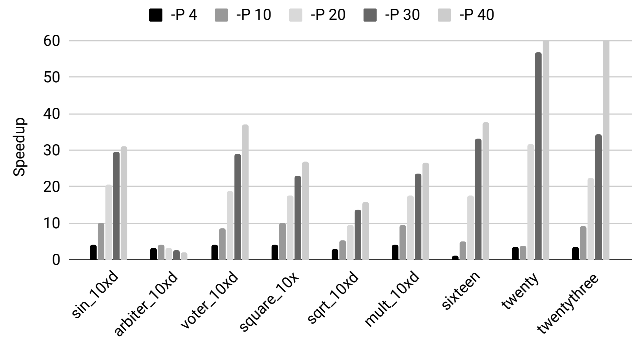

Fig. 4. Speedups by main miter partitioning (*Model P*).

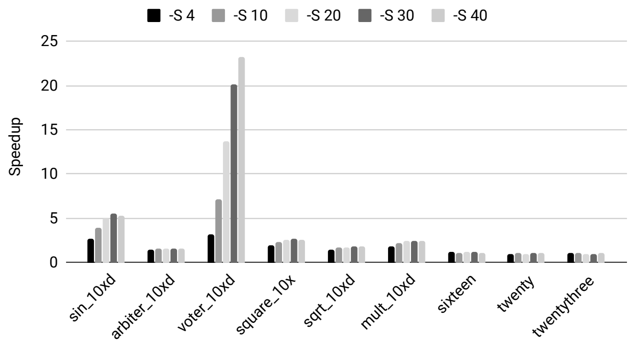

Fig. 5. Speedups by internal miter partitioning (*Model S*).

D. Discussion

Table V presents a summary of the best results and the respective thread configuration. In general, the configurations $-P$ 40 and $-P$ 10 $-S$ 4 lead to the best results. Notice that the proposed approaches can significantly reduce the runtime of CEC, compared to the single-thread method. One great improvement took place when the CEC runtime went down from 24h13min to only 37min and from 18h51min to only 18min. Overall, the proposed solutions have the potential to improve existing EDA environments.

Given that CEC is used as a building block in many computations, the parallel CEC engine proposed in this work can speed up other important tasks which depends on proving logic equivalence. Moreover, with additional customization, which performs proper handling of fanout nodes during equivalence checking, the scalable CEC techniques are also applicable in: (i) SAT sweeping under observability *don't-cares* [27]; (ii) node minimization with satisfiability, observability and external *don't-cares* [28]; (iii) high-effort resynthesis for circuit delay, area, power and congestion reduction using Boolean resubstitution [25]; (iv) various traversal-based computations comparing functions of the nodes in terms of primary inputs under observability conditions (ATPG, redundancy removal, false path detection and removal), and others.

Regarding the previous parallel CEC method EQUIPE [15], it is hard to perform a direct comparison to such a method. At first, the methods are running on different platforms with the technology gap of almost a decade. Besides that, it is not clear whether the authors are comparing against *cec* or &*cec* command, which is significantly faster than *cec*. Moreover, it is hard to ensure that we are using exactly the same pairs of original and optimized designs as input to CEC. Therefore, in order to avoid an unfair comparison, we are presenting an analysis based on our solution and on the general information presented in [15].

The results produced by EQUIPE method were collected on a platform containing a CUDA-enabled 8800GT GPU with 14 multiprocessors operating at 600 MHz and a CPU Intel Core 2 Quad operating at 2.4 GHz. The authors reported average speedups of one order of magnitude when compared to a commercial tool and only up to 3.22x speedup when compared to ABC tool. In the latter case, even using the 4 CPU cores and the 14 auxiliary GPU cores, the method was not able to speedup CEC beyond 3.22x. It is worth to notice that the CPU and GPU cores work at different frequencies and there is an additional cost related to the communication between CPU and GPU. Also, authors mention that in some cases internal miters need to be reconstructed, leading to runtime degradation, as shown in [15]. On the other hand, the parallel models P and S proposed in this work are able to verify designs with millions of AIG nodes. It scales to many cores and achieves significant improvement when compared to &*cec*, the latest CEC engine in ABC. Therefore, the proposed approach is more promising than EQUIPE method when it comes for improving CEC for large designs.

V. CONCLUSIONS

The paper proposes a novel approach for speeding up CEC, comprising three different models to enable parallelism in a modern CEC engine. The models trade off data sharing and data independence by applying graph partitioning during mitering and SAT sweeping. Experiments lead to promising results in speeding up the verification task for large designs. In several cases, where the ABC or the commercial verification tool took more than one day for checking designs, the proposed approach finished this task in few minutes/hours. The proposed solution has additional practical benefits, in par-

TABLE IV
RUNTIME COMPARISON BETWEEN ORIGINAL ABC COMMAND *&cec* AND THE COMBINED MODELS *P* AND *S* RUNNING AT 40 THREADS, IN (H:M:S).

Design	ABC *&cec*	-P 4 -S 10	-P 10 -S 4	-P 20 -S 2
sin_10xd	1:04:52	0:03:55	0:02:32	**0:02:04**
arbiter_10xd	0:05:17	0:01:03	**0:00:47**	0:01:20
voter_10xd	24:13:00	0:39:45	**0:37:05**	0:37:26
square_10xd	0:33:53	0:03:21	0:01:54	**0:01:27**
sqrt_10xd	21:34:04	5:33:28	2:38:45	**1:44:48**
mult_10xd	1:21:10	0:09:09	0:04:51	**0:03:36**
sixteen	8:27:33	1:54:34	1:14:02	**0:28:07**
twenty	14:35:59	2:17:06	1:42:43	**0:27:01**
twentythree	18:51:30	2:21:32	2:22:09	**1:06:56**

TABLE V
SUMMARY OF THE BEST RESULTS AND CONFIGURATIONS FOR EACH DESIGN, IN (H:M:S).

Design	ABC *&cec*	Parallel	Speedup	Config.
sin_10xd	1:04:52	0:02:04	31.33x	-P 20 -S 2
arbiter_10xd	0:05:17	0:00:47	6.76x	-P 10 -S 4
voter_10xd	24:13:00	0:37:05	39.17x	-P 10 -S 4
square_10xd	0:33:53	0:01:16	26.72x	-P 40
sqrt_10xd	21:34:04	1:21:22	15.90x	-P 40
mult_10xd	1:21:10	0:03:03	26.59x	-P 40
sixteen	8:27:33	0:13:28	37.67x	-P 40
twenty	14:35:59	0:14:25	60.73x	-P 40
twentythree	18:51:30	0:17:56	63.08x	-P 40

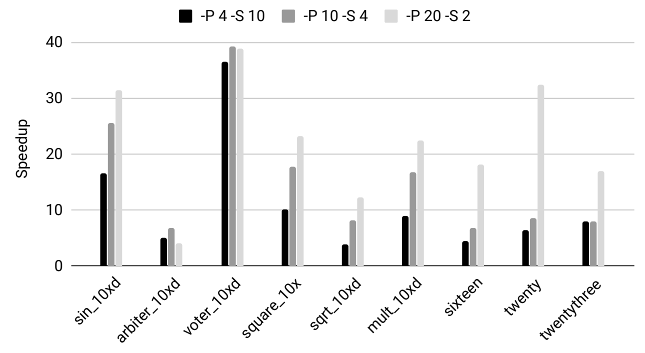

Fig. 6. Speedups by combining models *P* and *S* at 40 threads.

ticular, the potential to improve the runtime and scalability of other applications in current EDA environments. Moreover, the proposed models can exploit massively parallel environments of cloud computing.

ACKNOWLEDGMENT

This work was supported in part by the Brazilian Funding Agencies CNPq, CAPES and by SRC Contracts 2710.001 and 2867.001. The authors would like to thank Prof. Keshav Pingali from the UT Austin for the courtesy to collect experimental results in his multi-processor servers.

REFERENCES

[1] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, Feb 1990.

[2] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, Dec 2006.

[3] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proc. of Formal Methods in Computer Aided Design - FMCAD*, 2000, pp. 372–389.

[4] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Proc. of International Conference on Computer Aided Design - ICCAD*, 2008, pp. 234–241.

[5] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," in *Proc. of International Conference on Computer Aided Design - ICCAD*, 2009, pp. 789–796.

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

[7] L. Stok, "Developing parallel EDA tools [the last byte]," *IEEE Design and Test*, vol. 30, no. 1, pp. 65–66, 2013.

[8] ——, "The next 25 years in EDA: A cloudy future?" *IEEE Design and Test*, vol. 31, no. 2, 2014.

[9] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Proc. of Design Automation Conference - DAC*, 1996, pp. 629–634.

[10] J. Moondanos, C. Seger, Z. Hanna, and D. Kaiss, "CLEVER: Divide and conquer combinational logic equivalence verification with false negative elimination," in *Proc. of International Conference on Computer Aided Verificatio - CAV*, 2001, pp. 131–143.

[11] I.-H. Moon and C. Pixley, "Non-miter-based combinational equivalence checking by comparing BDDs with different variable orders," in *Proc. of Formal Methods in Computer Aided Design - FMCAD*, 2004.

[12] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, Dec 2002.

[13] F. Lu, L. . Wang, K.-T. Cheng, and R. C. . Huang, "A circuit SAT solver with signal correlation guided learning," in *Proc. of Design and Test in Europe - DATE*, 2003, pp. 892–897.

[14] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. of International Conference on Computer Aided Design - ICCAD*, 2006, pp. 836–843.

[15] D. Chatterjee and V. Bertacco, "EQUIPE: Parallel equivalence checking with GP-GPUs," in *Proc. of International Conference on Computer Design - ICCD*, 2010, pp. 486–493.

[16] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. of International Workshop on Logic and Synthesis - IWLS*, 2006.

[17] N. En and N. Srensson, "An Extensible SAT-solver," in *Proc. of International Conference on Theory and Applications of Satisfiability Testing*. Springer, Berlin, Heidelberg, 2003, pp. 502–518.

[18] M. J. H. Heule, O. Kullmann, e. Y. Biere, Armin", and L. Sais, *Cube-and-Conquer for Satisfiability*. Cham: Springer International Publishing, 2018, pp. 31–59.

[19] L. Amarú, P. Gaillardon, A. Mishchenko, M. Ciesielski, and G. D. Micheli, "Exploiting Circuit Duality to Speed up SAT," in *2015 IEEE Computer Society Annual Symposium on VLSI*, July 2015, pp. 101–106.

[20] D. Brand, "Verification of large synthesized designs," in *Proc. of International Conference on Computer Aided Design - ICCAD*, 1993, pp. 534–537.

[21] G. S. Tseitin, *On the complexity of derivation in propositional calculus, Automation of reasoning*. Springer Berlin Heidelberg, 1983.

[22] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proc. of International Conference on Computer Aided Design - ICCAD*, 2004, pp. 50–57.

[23] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug 1986.

[24] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," Technical Report, Tech. Rep., 2005.

[25] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 34:1–34:23, Dec. 2011.

[26] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. of International Workshop on Logic and Synthesis - IWLS*, 2015.

[27] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. L. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *Proc. of Design Automation Conference - DAC*, 2006, pp. 229–234.

[28] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proc. of Design and Test in Europe - DATE*, 2005, pp. 412–417.