Improvements to Combinational Equivalence Checking

Alan Mishchenko Satrajit Chatterjee Robert Brayton
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
{alanmi, satrajit, brayton}@eecs.berkeley.edu

Niklas Een
Cadence Berkeley Labs
1995 University Ave, Suite 460
Berkeley, CA 94704
niklas@cadence.com

Abstract

The paper explores several ways to improve the speed and capacity of combinational equivalence checking based on Boolean satisfiability (SAT). State-of-the-art methods use simulation and BDD/SAT sweeping on the input side (i.e. proving equivalence of some internal nodes in a topological order), interleaved with attempts to run SAT on the output (i.e. proving equivalence of the output to constant 0). This paper improves on this method by (a) using more intelligent simulation, (b) using CNF-based SAT with circuit-based decision heuristics, and (c) interleaving SAT with low-effort logic synthesis. Experimental results on public and industrial benchmarks demonstrate substantial reductions in runtime, compared to the current methods. In several cases, the new solver succeeded in solving previously unsolved problems.

1 Introduction

Combinational equivalence checking (CEC) plays an important role in EDA. Its immediate application is verifying functional equivalence of combinational circuits after multi-level logic synthesis [6]. In a typical scenario, there are two structurally different implementations of the same design, and the problem is to prove their functional equivalence. This problem was addressed in numerous research publications, in particular [26][21][23][32][31].

In a modern CEC flow, the two circuits to be verified are transformed into a single circuit called a *miter* [4] derived by combining the pairs of inputs with the same names and feeding the pairs of outputs with the same names into EXOR gates, which are ORed to produce the single output of the miter. The miter is a combinational circuit with the same inputs as the original circuit and the is constant 0 if and only if the two original circuits produce identical output values under all possible input assignments.

Sequential equivalence checking (SEC) benefits directly from improved CEC in three ways: (1) if a subset of unreachable states of the sequential circuit is known, CEC can often prove sequential equivalence [16], (2) verifying CEC of sequential circuits unrolled for a fixed number of timeframes gives a practical method for bounded SEC when other methods fail, and (3) as a preliminary step in unbounded SEC [25]. Numerous methods use CEC in bounded model checking, for example [22].

ICCAD '06, November 5-9, 2006, San Jose, CA.

CEC has applications in logic synthesis. For example, it can be used to compute node flexibilities, such as don't-cares and resubstitutions [28], or canonical function properties, such as classical symmetries [42].

Because of its importance in many practical applications, fast and scalable CEC methods are needed. Contrary to common belief, CEC is not a "solved problem". New applications and the increase in the sizes of problem instances cause existing CEC methods to break down, as to be expected of an co-NP-hard problem.

The current state-of-the-art in CEC [21] uses an *integrated approach*, starting from an And-Inverter Graph (AIG) representation of the logic. Hashing is applied to detect structural similarity that may be present in the miter derived from the two circuits. Next, simulation is performed to detect possible classes of AIG nodes with equal global functions (up to complementation). The equivalence of these nodes is checked by constructing BDDs or solving SAT under user-controlled resource limits. Intermittently, attempts are made to solve SAT for the output of the miter under increasing resource limits. The main idea is that the more equivalent nodes detected inside the miter, the easier it is to prove (solve SAT on) the output. However, proving all equivalent internal nodes is unnecessary if the output is easy for SAT.

In this paper, we improve the above integrated approach to CEC with the following contributions:

1. **Use of fast logic synthesis**. Preprocessing the miter by AIG rewriting was introduced in [3] for verification and developed in [29] for logic synthesis. Logic synthesis results in fewer AIG nodes, which correlates with faster SAT solving; logic sharing discovered during synthesis proves some equivalences between the nodes very quickly. In one extreme case, an industrial CEC example could not be solved in hours, yet was solved by one quick pass of rewriting-based logic synthesis.
2. **Development of "intelligent simulation".** This substantially reduces the number of satisfiable SAT calls (thereby improving runtime) needed to disprove equivalences of internal AIG nodes not distinguished by known simulation techniques.
3. **Use of CNF-based SAT for circuits.** Previous work in CEC [21] used circuit-based SAT solvers. We modify a CNF-based solver [33][12] to use circuit information [37][38] and efficient circuit-to-CNF conversion [41] resulting in faster CEC.

The result is a new CEC package that is substantially faster than other similar tools. This claim is supported by extensive experiments with diverse applications from academia and industry.

The paper is organized as follows. Section 2 surveys traditional AIGs. Section 3 details the proposed improvements and reviews related previous work. Section 4 outlines the use of logic synthesis in a modified integrated CEC. Section 5 reports experimental results. Section 6 concludes and lists directions for future work.

2 Background

Familiarity with the basics of Boolean functions, Boolean networks, and Boolean satisfiability is assumed.

Definition. *And-Inv Graph* (AIG) is a Boolean network composed of two-input AND-gates and inverters.

To derive an AIG, the SOPs of the nodes in the network are factored [5], the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these nodes are added to the AIG manager.

Definition. The *size* of an AIG is the number of its AND nodes. The *number of logic levels* is the number of AND nodes on a longest path from any primary input to any primary output.

The inverters are ignored when counting nodes and logic levels. In the software implementation, inverters are represented as flipped node pointers, similar to the complemented edges in a BDD.

Figure 1 shows a Boolean function and two of its structurally-different AIGs. The nodes in the graphs denote AND-gates, the bubbles label complemented edges.

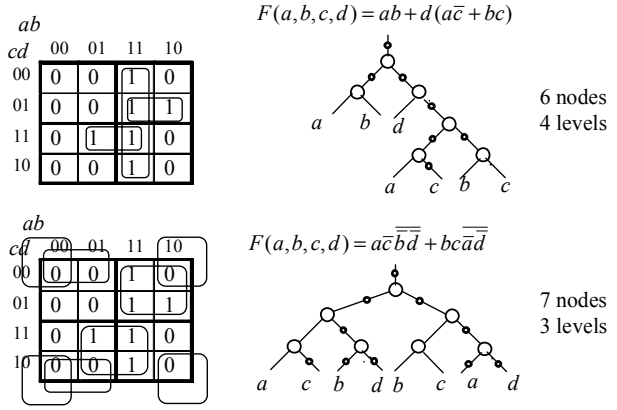

Figure 1. Two different AIGs for a Boolean function.

Definition. *Structural hashing* (*strashing*) of an AIG is a transformation intended to reduce the AIG size by partially canonicizing the AIG structure. When a new AND-gate is added, a hash-table is checked for a node with the same fanins (up to permutation). Although the resulting AIG is not functionally canonical, it does not contain isomorphic subgraphs.

Structural hashing was originally introduced for netlists of arbitrary gates in some early IBM CAD tools [11][37] and was extensively used for AIGs in previous work on CEC [20][21].

Definition. The function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in node n and expressed in terms of the leaves, x, of the AIG.

Definition. A *functionally reduced AIG* (FRAIG) is one in which, for any two n_1 and n_2, $f_{n_1}(x) \neq f_{n_2}(x)$ and $f_{n_1}(x) \neq \overline{f_{n_2}(x)}$.

Proving the output of the miter to be constant 0 is called *mitering* in this paper. Transforming an AIG into a functionally reduced AIG is called *fraiging*. In general, fraiging can be performed with a fixed resource limits, for example, fraiging may be forced to quit proving equivalence of a pair of intermediate nodes when a user-specified limit on the number of conflicts is reached. In this case, the functional reduction of the AIG is not complete. The same process of resource-aware detection and merging of functionally equivalent nodes in an AIG (up to complementation) using BDDs (SAT) is known as *BDD sweeping* [21] (*SAT sweeping* [22]).

Unless stated otherwise, mitering is done by asserting the output of the miter to be constant 1 and running a SAT solver on the resulting problem. If the solver returns "unsatisfiable", the miter is proved constant 0 and the original circuits are equivalent. If the solver returns "satisfiable", a *counter-example* is returned. A counter-example is an assignment of the primary input variables leading to 1 at the output of the miter. It can be used for debugging the original circuits verified by running SAT on the miter.

SAT solvers can be circuit-based or CNF-based. The former represent the SAT problem as a circuit composed of simple gates, while the latter use conjunctive-normal-form (CNF). The second type of solvers is more general and can also be applied to circuits, by converting them into CNF form.

A naïve circuit-to-CNF conversion uses as many variables as there are primary inputs and internal nodes. It derives clauses for each gate and adds them to the resulting CNF. For example, a two-input AND, $c = ab$, leads to $(c \Rightarrow ab) \wedge (ab \Rightarrow c)$, or equivalently, to the three CNF clauses:

$$(\bar{c} \vee a) \wedge (\bar{c} \vee b) \wedge (\bar{a} \vee \bar{b} \vee c).$$

A more efficient circuit-to-CNF conversion [41], when applied to an AIG, groups AIG nodes into larger gates as follows: (a) whenever possible two-input ANDs are expanded into multi-input ANDs without area duplication, (b) EXORs, MUXes, and trees of MUXes are detected and converted into clauses as single nodes. This reduces the number of clauses and variables and noticeably improves the performance of the SAT solver.

The speed of Boolean constraint propagation can be further improved by adding certain redundant clauses, as shown in [15]. For example, when MUX, $m = ca + \bar{c}b$, is translated into CNF, in addition to the four required clauses (two for each output phase):

$$c \wedge a \Rightarrow m,\ \bar{c} \wedge b \Rightarrow m,\ c \wedge \bar{a} \Rightarrow \bar{m},\ \bar{c} \wedge \bar{b} \Rightarrow \bar{m},$$

two redundant clauses are added:

$$a \wedge b \Rightarrow m,\ \bar{a} \wedge \bar{b} \Rightarrow \bar{m}.$$

As a result, when the input variables of a MUX have equal values, the SAT solver uses the additional clauses to imply the value of the output directly, without other decisions. For example, if the inputs have values $a = 1$ and $b = 1$, it will be implied directly that $m = 1$.

3 Improvements to integrated CEC

This section details the improved integrated CEC proposed in this paper. Each subsection is preceded by a summary of the relevant previous work.

3.1 Intelligent simulation

Previous integrated approaches to CEC perform *fraiging* (identifying and merging functionally equivalent nodes) as a way to simplify the miter. For large miters, the runtime of fraiging can be prohibitive. Therefore, it is important to reduce to a bare minimum the number of SAT calls needed to test the equivalence of the intermediate nodes. Typically, simulation is used to detect nodes that are not functionally equivalent.

Previous work [23][22] used random vectors and counter-examples returned by the SAT solver as simulation vectors. A counter-example in this case is an assignment of the primary input variables that distinguishes the two nodes under investigation. Although counter-examples improve the performance of random simulation, there are still a substantial number of satisfiable SAT calls needed during fraiging to resolve the remaining node pairs.

We propose an improvement to simulation, which systematically generates useful additional simulation patterns. These are derived by focusing on distinguishing nodes that cannot be distinguished easily using random patterns and counter-examples alone.

Typically, random simulation is done until "saturation". This identifies classes of nodes that appear equivalent under simulation. Next, these candidate equivalent nodes are checked by SAT, giving preference to node pairs appearing earlier in a topological order. If the SAT problem is "unsatisfiable", the nodes are equivalent and can be merged; otherwise, a new counter-example is generated.

Our new simulation patterns are derived as follows. Given the simulation vector representing the counter-example, all *distance-1 patterns* from it are obtained by flipping one bit in this vector. Only the bits in the structural support of the logic cones of the current node pair are flipped. Other bits can be selected at random. Next, the distance-1 patterns are simulated to check if they resolve other pairs not resolved by the counter-example. We collect the distance-1 patterns which resolved additional pairs, generate a new set of distance-1 patterns, and simulate them. This is continued as long as new candidate pairs are resolved. When the intelligent simulation saturates, a new SAT call is made. The process goes on until the candidate node pairs are exhausted or a resource limit is reached. The pseudo-code is shown in Figure 2.

```
fraigingWithIntelligentSimulation( miter, backtrack_limit, time_limit )
{
    // Start with a fixed amount of random simulation
    runRandomSimulation( miter );
    classes = detectNodeEquivalenceClassesUnderSimulation( miter );

    // Continue random simulation till saturation
    do {
        classes_old = classes;
        classes = runRandomSimulation( miter, classes_old );
    } while ( classes != classes_old );   // classes are refined

    // Alternate SAT solving and intelligent simulation
    while ( classes != empty && time < time_limit ) {
        pair = popFirstPairUsingTopologicalOrder( classes );
        miter_temp = generateTemporaryMiter( miter, pair );
        if ( applySat( miter_temp, backtrack_limit ) == UNSAT ) {
            miter = mergePair( miter, pair ); // merge equal nodes
            continue;
        }
        if ( !checkIsMiterSolved( miter_temp ) )  // limit reached
            continue;

        // Nodes are different and a counter-example is generated
        pattern = getCounterExample( miter_temp );
        dist1_patterns = generateDistance1Patterns( pattern );

        // Peform simulation using distance-1 patterns till saturation
        do {
            classesOld = classes;
            classes = runSimulation( miter, classes_old, dist1_patterns );
            patterns = getSuccessfulPatterns( classes, dist1_patterns );
            dist1_patterns = generateDistance1Patterns( patterns );
        } while ( classes != classes_old ); // classes are refined
    }
}
```

Figure 2. Detecting and merging candidate equivalence classes using intelligent simulation and SAT.

Intelligent simulation has been shown to substantially reduce runtime for those benchmarks, whose fraiging is dominated by satisfiable SAT runs. In such cases, the number of satisfiable runs decreases roughly two times, compared to using only random simulation and counter-examples.

The reason why distance-1 simulation works on practical benchmarks is not known, but we speculate as follows. Consider a class of n potentially equivalent nodes. Then there are $n(n\text{-}1)/2$ pairs to be resolved. Suppose the counterexample splits the class into two parts of size $n/2$. There still remains to resolve $n(n\text{-}2)/4$ pairs. We speculate that the counterexample is "close" to resolving the other pairs in the class since all functions in any potential equivalence class have significant overlap of their onsets and offsets. For example, many realistic functions would have a small number of large cubes covering their onsets or offsets. Therefore, two functions that have a significant overlap of onsets and offsets would tend to have cubes that are distance-1 apart, since larger distance cubes would be easier to resolve by random simulation.

Obviously, this heuristic does not work for random functions, for which the probability of finding a distinguishing pattern is the same for any minterm.

3.2 Interleaving input and output SAT runs

It is important to balance fraiging (solving intermediate node equivalence) with mitering (solving output node equivalence) by using interdependent resource limits [21]. If a proper balance is not achieved, the SAT solver may spend too much time mitering, which would be easier after some fraiging. At the other extreme, the solver may spend too much time fraiging while the mitering problem may be easy to solve without it. Some degree of integration of fraiging and mitering was achieved in [21] by using the same SAT solver for both procedures. This saves the runtime for setting up the solver and shares learned-clause information across all the SAT calls.

Synchronization of fraiging and mitering can be taken one step further. For this, during mitering we can count the number of conflicts, in which each variable has been involved. This information will be used as follows. In the next round of fraiging, we can first try to resolve potential equivalence classes involving variables, which participated in many conflicts during mitering. New equivalences proved while fraiging will simplify the miter in the critical areas, which may speed up mitering.

This improvement is currently not implemented. We plan to investigate it as part of the future work.

3.3 Using CNF-based solvers for circuits

Traditionally, circuit-based solvers have been preferred when working with verification problems for circuits [4][21][23] because of their fast constraint propagation and good circuit-based variable ordering heuristics, perfected in ATPG research [1].

In recent years, CNF-based solvers [31][12] have improved significantly in performance by using new methods for clause-database management (two-literal watching), improving conflict analysis (non-chronological backtracking), and finding better variable ordering heuristics (activity-based decisions).

However, circuit-based solvers still have the advantage of making better decisions because of circuit structure information, which is not available to the CNF-based solvers. On the other hand, circuit-based solvers cannot efficiently record information obtained by conflict-based learning. This motivated hybrid solvers, such as [17][19], which rely on circuits to represent the SAT problem and on CNF to store learned clauses.

In this work, we follow the approach proposed in [37][38] and enhance the CNF-based solver with circuit based information. The resulting solver has the following strengths:

- It uses the most recent advances in efficient CNF-based SAT solving, such as conflict-clause minimization [14].
- It uses efficient circuit-to-CNF translation [41], which decreases the number of variables and clauses (compared to the brute-force translation of each AND2 into three clauses), improves *implicativity* [35] (measured as the average number

of implications between subsequent decisions), and reduces memory footprint.
- Circuit-based decision heuristics are added to a CNF-based solver by augmenting it with the APIs described below.

The first API allows a CNF-based solver to work on a subset of variables. This is particularly important if the miter is large while a cone to be solved during fraiging is small. An unconstrained CNF-based solver may waste some time making decisions and generating implications outside of the relevant cone. A simple implementation of this idea traverses the relevant cone and marks all the SAT variables in it. The CNF-based solver can be modified to decide and propagate implications only for the marked variables.

The second API augments the CNF-based solver with variable ordering procedures based on the notion of justification frontier, or *J-frontier* [1][37][38]. At any time during the solving process, the *J*-frontier is the set of currently unassigned variables, each of which has at least one fanin/fanout variable that is currently assigned. In the context of a CNF-based solver, default decisions based on variable activity [12], are modified to choose the most active variable among those currently on the J-frontier.

Implementation of this heuristic requires providing the CNF-based solver with structural information about the circuit (the set of fanin/fanout variables for each variable) and updating the internal representation of the *J*-frontier each time a new variable is assigned and unassigned during solving. Our current implementation (which is not optimized) spends approximately 40% of SAT solver runtime for supporting the *J*-frontier. Nevertheless, the capability of the solver to make circuit-based decisions typically leads to runtime improvements, compared to using the same solver with only activity-based decisions. The improvements are benchmark dependent and can be either negligible or an order of magnitude.

It is possible that *J*-frontier heuristic is important for large unsatisfiable benchmarks because the complete Boolean space exploration for them is hard to efficiently do using activity-based decisions alone. Such decisions tend to make decisions in distant parts of the search space, leading to larger conflict clauses. In contrast, incorporating circuit-based decisions tends to make the SAT search more focused, resulting in smaller conflict clauses and hence faster coverage of the search space.

A more efficient implementation of the *J*-frontier in a CNF-based solver can use a variable stack for storing currently unassigned variables. Variables are removed from the stack or added to it, as they are assigned/unassigned during SAT solving. To make a decision, the first *k* variables in the stack can be traversed to choose the one with the highest activity. Another way of simulating the *J*-frontier in a CNF-based solver uses multiplicative activity weights, which are higher for SAT variables that are closer to the root of the miter to be checked [12].

3.4 Using logic synthesis in CEC

We have observed that preprocessing of the miter with logic synthesis often leads to a reduction in the runtime of CEC. Logic synthesis makes the AIG representation of the miter more compact by removing redundancies and detecting shared logic. This reduces the number of CNF variable and clauses, which in turn makes SAT solving more efficient.

For application to CEC, we use a fast AIG rewriting method, which was proposed for quick circuit compression in [3]. We developed it further in [29] as an alternative logic synthesis flow, and showed it to be comparable to or better than traditional logic synthesis.

AIG rewriting

Rewriting is a sequence of fast local transformations of the AIG representation of combinational logic. Rewriting alternates DAG-aware Boolean restructuring and algebraic tree-balancing, which reduces the number of AIG nodes by sharing common logic. The following are the advantages of the AIG rewriting, compared to the traditional logic synthesis:

- While still being heuristic and suboptimal, AIG rewriting requires no hand-tuning or trial-and-error of scripts because it monotonically reduces the number of AIG nodes.
- Improvements in the logic complexity, measured by the number of AIG nodes and levels, are in good correspondence with the complexity of resulting SAT problems generated from the AIG using efficient circuit-to-CNF conversion.
- AIG rewriting is much simpler than traditional synthesis. A robust implementation used in this paper took a few weeks of one person's time to implement.
- It is orders of magnitude faster than the traditional flow in its most rugged and robust versions, while the quality is comparable or better when measured by the delay and area of the network after technology mapping.

It should be noted that AIG rewriting is local; however, rewriting is very fast and can be applied to the miter many times. For example, performing 10 rewriting passes over a typical AIG is still at least an order of magnitude faster than running the resource-aware implementation of the traditional flow in MVSIS [34] and 2-3 orders of magnitude faster than the traditional flow in SIS [35]. By applying rewriting many times, the scope of changes is no longer local. The result is that the cumulative effect of several rewriting passes is often superior to traditional synthesis.

Rewriting works by exhaustively enumerating all four-input logic cones of each AIG node in the topological order. A Boolean function of each such cone is computed and checked against a number of pre-computed non-redundant AIGs for this function, while accounting for possible logic sharing with other logic cones. If the current representation can be replaced by the stored one without increasing the total number of AIG nodes, the transformation is accepted and processing moves to the next node.

For a detailed description of the algorithm and its experimental comparison against the traditional logic synthesis, the reader is referred to [3] and [29].

4 Modified integrated approach

The following describes the CEC framework based on rewriting, mitering, and fraiging – all performed with balanced resource limits, controlled by the user. The original integrated approach [21] performed only fraiging and mitering. The pseudo-code of the resulting algorithm is shown in Figure 3.

Our CEC procedure takes a problem in the form of a miter and a set of resource limits indicating how many iterations of integrated solving to perform (*iter_limit*), what are the limits for the intermediate steps (*mitering_limit, rewriting_limit, fraiging_limit*), and how these limits change over time (*mitering_increase, rewriting_increase, fraigng_increase*).

An iteration of CEC begins by mitering with a fixed resource limit (typically, 1000 conflicts). The reason for beginning the first iteration with mitering rather than other steps, is that some seemingly difficult problems can be solved quickly by SAT alone.

Mitering involves converting the current AIG structure into CNF, as described in Section 2, and running a CNF-based solver [12]. In our current implementation, circuit-based decisions are only used

in fraiging but not in mitering. Moreover, CNF is re-created from scratch in each iteration of the loop. These limitations will be addressed in the future.

```
integratedCEC( miter, resource_limits )
{
    status = undecided;
    for ( iter = 1;  iter <= iter_limit; iter++ ) {
        // Try mitering
        status = doMitering( miter, mitering_limit + iter*mitering_increase );
        if ( status != undecided )   break;
        // Try rewriting
        status = doRewriting(miter, rewriting_limit+iter*rewriting_increase );
        if ( status != undecided )   break;
        // Try fraiging
        status = doFraiging( miter, fraiging_limit + iter*fraiging_increase );
        if ( status != undecided )   break;
    }
    if ( status != undecided )
        status = doMitering( miter, final_mitering_limit );
    if ( status == satisfiable )
        miter->counter_example = generateCounterExample( miter );
    return status;
}
```

Figure 3. Pseudo-code of the integrated CEC.

If the miter is not solved after the application of brute-force SAT, then rewriting and fraiging are attempted. Each of these transformations may actually solve the miter: rewriting may have simplified the miter to a constant, or fraiging with the given resource limit may have succeeded in proving that the output of the miter is equivalent to a constant. If the miter is not solved after these steps, the next iteration is performed with the increased resource limits.

In the end, if the miter is not solved after a user-specified number of iterations, one last attempt is made to solve it with a larger resource limit. Also, whenever the miter is found satisfiable, a counter-example is generated and returned.

Experiments using a large selection of benchmarks from different applications domains, including verification [8], logic synthesis [28] and software synthesis [40], have shown that the proposed integrated CEC is faster and more rugged than the integration of any two methods, such as only fraiging and mitering [21]. This can be explained as follows.

Rewriting efficiently detects logic sharing and thereby quickly finds and merges numerous pairs of functionally equivalent nodes, which would take longer to be proved by fraiging. For example, two different tree-decompositions of a three-input AND, (*ab*)*c* and *a*(*bc*), are trivial to merge by rewriting but would require an expensive, even if relatively fast, SAT call during fraiging. On the other hand, if fraiging is not performed, local rewriting will quickly saturate and fail to substantially reduce the miter. Finally, rewriting and fraiging, used with matching resource limits, provide both local and global views of node equivalences, which facilitate quicker reduction of the miter, leading to proving it or finding a disproving counter-example.

5 Experimental results

The proposed modifications to CEC were implemented in ABC [2], a public-domain synthesis and verification system. The new equivalence checker was tested on numerous benchmarks from industrial as well as academic sources [8][40]. The correctness of the results reported on unsatisfiable miters was verified by running external public-domain SAT solvers, whenever possible. The counter-examples for satisfiable miters were verified by simulating them through the miters and ensuring that the output becomes 1.

5.1 Performance on IWLS benchmarks

The first experiment reports the performance of the proposed checker on bounded sequential synthesis problems generated from IWLS benchmarks. The original benchmarks were compared against ones synthesized using fast AIG-based logic synthesis [29] followed by an integrated mapping/retiming for FPGAs [30]. The miters were produced by unrolling the product machine for a given number of timeframes and comparing the POs at all time frames. The number of time frames was selected individually for each benchmark to create a miter taking about one minute to solve.

Table 1. IWLS benchmark statistics and runtime comparison.

Benchmark	AIG statistics				Runtime, sec			
	Fr	PIs	AND2	Lev	prove	prove-r	prove -j	sat
ac97_ctrl	20	1680	66681	120	20.52	48.08	25.86	-
aes_core	5	1295	85100	64	42.30	45.88	64.58	-
des_area	8	1920	79242	194	52.37	71.31	290.39	-
des_perf	15	3510	113322	130	66.53	155.21	89.12	36.19
ethernet	15	1470	66098	239	36.23	70.12	41.57	-
i2c	20	380	36144	213	26.34	41.24	61.75	-
mem_ctrl	20	2300	81271	159	48.87	110.73	80.71	-
sasc	50	800	66096	370	21.40	22.77	81.76	-
simple_spi	50	800	90790	479	16.46	19.45	13.13	981.31
spi	10	470	58788	201	50.91	80.11	92.28	-
ss_pcm	80	1520	55739	453	28.24	50.67	150.86	-
systemcaes	15	3900	223480	434	58.15	35.08	54.85	6.37
systemcdes	15	1980	79641	270	67.90	96.48	474.54	-
usb_funct	10	1280	106051	118	93.04	150.85	131.92	-
usb_phy	50	750	33543	364	25.67	38.13	59.57	-
vga_lcd	3	267	88784	35	10.59	123.49	10.68	15.16
wb_conmax	4	4520	163548	60	49.37	60.77	74.05	-
wb_dma	20	4340	91034	128	39.46	63.80	44.49	-
Ratio					1.00	2.12	2.34	> 42.00

The miters were generated in ABC using the following script: **read <input_file>; resyn2; sfpga; miter -c; frames -i -F <num>; orpos; write_blif <output_file.blif>**, where **resyn2** is the logic synthesis script performing 10 iterations of AIG rewriting, **sfpga** runs integrated mapping/retiming for FPGAs using 5-input LUTs, **miter -c** derives the product machine from the original and synthesized benchmark, **frames -i -F <num>** creates initialized unrolled timeframes of the product machine, **orpos** derives Boolean OR of the POs of all frames, resulting in a single-output combinational miter. Finally, the miter is written into the output file. The resulting miters are publicly available in BLIF and BENCH formats [43].

Table 1 lists the statistics of the miters and the results of running our equivalence checker with the proposed improvements. Table 1 helps document the relative contributions of the separate improvements. Column "Benchmark" lists the IWLS benchmarks used to derive the miters. Columns "Fr", "PIs", "AND2", and "Lev" show the number of timeframes used to unroll the product machine, as well as the number of PIs, AIG nodes, and AIG levels after unrolling.

The remaining columns show the runtime of different CEC options reported on a 1.6GHz laptop with 1Gb of RAM. Column "prove" shows the runtime of the new checker with the proposed improvements. Dash means that a solver timed out after 1800 seconds. Column "prove –r" shows the runtime when AIG rewriting is not performed as an intermediate step in the integrated

procedure of Figure 3. Column "prove -j" shows the runtime when fraiging (which internally runs a modified version of MiniSat-1.12 [12]) is performed with default activity-based variable ordering heuristics, instead of the proposed *J*-frontier heuristic of Section 3.3. Finally, column "sat" shows the runtime of MiniSat-1.14 [14] on the CNF derived from the miter using efficient circuit-to-CNF conversion [41]. This uses the default activity-based variable ordering heuristics. The runtime listed as 1800.00 indicates that the status of the miter was undecided after 30 minutes.

Table 2 provides additional details on solving the combinational miters. Columns "Orig" and "IterN" show the sizes of the original miters as well as the sizes of the miters after *N* successive iterations of solving. All miters used could be solved in four iterations. Dash in the column indicating AIG size means that the miter was already solved in the previous iteration. The following resource limits were used for iteration *N*:

- **mitering** - at most $1000*2^N$ conflicts
- **AIG rewriting** - three iterations
- **fraiging** - at most $2*8^N$ conflicts at a node.

The second section of Table 2 shows the runtimes of the three main components of CEC summed over the iterations. Columns "Miter", "Synth", and "Fraig" show the runtime for mitering, AIG rewriting, and fraiging, respectively. The ratios of runtimes are with respect to the total runtime (column "prove" in Table 1).

Table 2. AIG size reduction details and breakdown of runtime.

Benchmark	AIG size (AND2)				Runtime, sec		
	Orig	Iter1	Iter2	Iter3	Miter	Synth	Fraig
ac97_ctrl	66681	14487	-	-	0.87	6.30	13.20
aes_core	85100	57472	4286	-	2.23	19.30	20.58
des_area	79242	49295	32250	-	1.69	14.39	36.10
des_perf	113322	35896	18008	-	20.71	19.74	25.80
ethernet	66098	20458	-	-	0.98	6.63	28.44
i2c	36144	21873	10058	7586	1.20	4.24	20.83
mem_ctrl	81271	37874	4336	-	2.04	11.89	34.75
sasc	66096	49663	2676	-	0.97	9.84	10.41
simple_spi	90790	3370	-	-	0.85	9.43	5.98
spi	58788	38812	-	-	2.18	8.16	40.47
ss_pcm	55739	41997	-	-	1.25	8.02	18.76
systemcaes	223480	54250	-	-	5.59	42.32	9.65
systemcdes	79641	59216	29355	-	1.56	18.02	47.09
usb_funct	106051	60228	14587	-	6.68	16.02	71.10
usb_phy	33543	25179	8921	3389	1.45	4.98	19.16
Vga_lcd	88784	11673	6821	-	1.04	7.19	2.23
wb_conmax	163548	52858	-	-	2.21	25.24	11.60
wb_dma	91034	91034	34858	-	1.34	10.42	27.40
Ratio	1.00	0.51	0.12	0.02	0.06	0.36	0.58

The conclusion from Table 1 is that the integrated CEC flow works better than a comparable flow without AIG rewriting or with activity-based variable ordering. Table 2 shows that the AIG size is quickly reduced during solving taking at most 4 iterations in the given examples. It also shows that fraiging is the slowest part of the flow, which justifies intelligent simulation and *J*-frontier variable ordering, and motivates further research to reduce its runtime. In other benchmark types, we observed even larger ratios of fraiging and smaller ratios of rewriting.

5.2 Comparison with CSAT

In this experiment, the proposed checker is compared with the circuit-based SAT solver and equivalence checker CSAT [23][24][10]. Since CSAT does not use logic synthesis, for fairness of comparison, the miters were preprocessed with three iterations of AIG rewriting (script *rwsat* in ABC).

Table 3 lists benchmark names followed by the miter statistics after preprocessing. Columns "AND2" and "Lev" show AIG nodes and levels, respectively. The next section lists the runtime of preprocessing (column "Prepro"), the runtime of the proposed checker (column "Prove -r"), and that of CSAT. The runtime of CSAT is reported on a 2GHz CPU under Linux while all other runtimes are on the 1.6GHz Windows laptop.

Table 3. Runtimes of equivalence checking with different options.

Benchmark	AIG after prepro		Runtime, sec		
	AND2	Lev	Prepro	Prove -r	CSAT
ac97_ctrl	27147	99	23.56	12.30	13
aes_core	71074	54	17.83	23.24	21
des_area	55213	157	12.87	42.10	44
des_perf	38875	86	20.28	44.08	37
ethernet	36566	187	7.90	25.51	19
i2c	25358	281	3.77	19.55	17
mem_ctrl	57407	120	13.75	42.44	33
sasc	51776	316	9.97	9.92	19
simple_spi	71229	555	15.52	5.32	14
spi	46142	179	9.40	40.72	64
ss_pcm	50619	446	9.17	22.86	38
systemcaes	161470	293	70.98	2.34	136
systemcdes	63955	216	15.23	48.91	87
usb_funct	73882	114	18.02	75.99	47
usb_phy	26626	340	4.05	20.63	15
vga_lcd	63345	39	13.96	146.30	1004
wb_conmax	58860	52	27.27	24.29	11
wb_dma	53104	88	13.33	30.13	17
Ratio				1.00	4.62

In summary, the proposed solver is comparable to CSAT on most benchmarks but more rugged on the larger ones. On average it is 4.62 times faster. When the two outlier cases (*systemcaes, vga_lcd*) are not considered, an average improvement over CSAT is 1.14.

5.3 Role of intelligent simulation

The runtime of the above unsatisfiable examples is dominated by unsatisfiable SAT calls during fraiging. This is in general true about miters constructed from two copies of a circuit. To show the contribution of intelligent simulation we need benchmarks, for which fraiging is dominated by satisfiable SAT calls. Such test-cases occur when a single copy of the netlist has to be fraiged, for example, during lossless logic synthesis [9].

Table 4. Speedup in fraiging due to intelligent simulation.

File	AIG statistics			UN SAT calls	Without dist-1 patterns		With dist-1 patterns	
	Fra-mes	AND2	Lev		SAT calls	SAT time	SAT calls	SAT time
pj1	1	16285	156	237	200	0.12	118	0.10
s444	100	15500	614	978	910	6.93	429	3.86
b14	10	60690	371	1462	600	17.70	270	9.97
b17	5	130492	317	2632	3782	162.63	2550	106.81
ratio					1.00	1.00	0.55	0.65

Table 4 shows the results of fraiging of four networks: PicoJava circuit *pj1,* the unrolled ISCAS circuit *s444*, and two unrolled ITC circuits, *b14* and *b17*. The first section shows the AIG statistics: the number of frames (column "Frames"), the number of AIG nodes (column "AND2") and the number of AIG levels (column "Lev"). The next column shows the number of unsatisfiable SAT calls. The next column shows the number of satisfiable SAT calls and the total runtime used for SAT in two cases: without and with

distance-1 patterns used for simulation. In both cases, random simulation and counter-examples are used.

In summary, Table 4 shows several non-miter circuits, for which the number of satisfiable SAT calls is comparable to the number of unsatisfiable SAT calls during fraiging. Intelligent simulation makes some of the satisfiable SAT calls unnecessary, which tends to reduce the SAT solver runtime. The additional runtime for intelligent simulation is relatively small.

5.4 Performance on industrial benchmarks

Table 5 reports running the equivalence checker on a set of unsatisfiable industrial benchmarks. Column "File name" lists the names of the benchmarks. Column "AND2" lists the number of AIG nodes in the miter after structural hashing. The remaining columns show the runtimes of different CEC options reported on a 1.6GHz laptop with 1Gb of RAM. Dash means that a solver timed out after 1800 seconds. The notation used is the same as in Table 1.

Table 5. Runtimes of equivalence checking with different options.

File name	AND2	Equivalence checking runtime, sec			
		prove	prove -r	prove -j	sat
Ex01	20631	19.80	58.94	79.72	-
Ex02	27574	36.59	333.94	142.98	-
Ex03	29745	46.16	149.18	178.25	-
Ex04	30178	36.26	152.25	942.53	-
Ex05	31869	49.99	189.98	-	-
Ex06	35141	103.03	324.29	1663.56	-
Ex07	10619	39.62	39.29	40.97	7.54
Ex08	36234	80.70	355.30	-	-
Ex09	9731	11.41	26.14	32.67	0.92
Ex00	36145	36.43	106.97	31.82	1711.21
Ex11	35978	95.20	2112.96	340.45	-
Ex12	40989	108.45	2123.06	416.39	-
Ex13	38093	105.61	2024.00	1287.46	-
Ex14	44098	53.85	185.54	64.03	0.68
Ex15	43302	3.65	177.21	3.70	0.31
Ex16	47078	56.90	355.40	288.60	5.40
Ratio		**1.00**	**9.77**	**> 9.00**	**> 25.28**

We also experimented with these benchmarks using other CNF-based solvers: ZChaff [30] and CSAT [23][24]. The former on average performed worse, compared to MiniSat (column "sat"). The latter on average performed better than MiniSat but still about an order of magnitude worse then the proposed equivalence checker (column "prove"). The improvement of CSAT over MiniSat-1.14 is probably due to using circuit-based variable decision heuristics.

Table 5 shows that the proposed improvements (the use of *J*-frontier and interleaving SAT solving with fast logic synthesis) are important to achieve robustness on a variety of benchmarks. The fact that some easy test cases (such as *ex*07, *ex*09, etc) can be solved faster using a SAT solver directly motivates trying for an even tighter integration between fraiging and mitering outlined in Section 3.2.

We observed improvements similar to those reported in Tables 1 and 5 on several other classes of benchmarks, including hard satisfiable benchmarks generated in software synthesis [40].

6 Conclusions and future work

We described several enhancements to CEC, which substantially reduce its runtime on hard industrial problems. The enhancements are based on:

- better simulation, which substantially reduces the number of satisfiable SAT calls;
- leveraging the advances in CNF-based SAT for circuit-based CEC problems through an efficient circuit-to-CNF conversion [41] and circuit decision heuristics (such as *J*-frontier [1]); these heuristics are particularly efficient for hard unsatisfiable problems and may lead to abandoning the need for circuit-based SAT solving in the future;
- performing logic synthesis on circuits before or during CEC tends to dramatically reduce subsequent SAT solver runtimes.

A public-domain equivalence checker based on these ideas was implemented in ABC [2] and tested on a wide variety of academic and industrial benchmarks.

As part of future work, we plan to implement interleaving of SAT runs suggested in Section 3.2 and experiment with different ways of realizing *J*-frontier, as described in Section 3.3. Other aspects of future work include developing CEC methods for miters with few or no internal equivalences and extending the proposed integrated approach to work for word-level verification problems.

Acknowledgement

This research was supported in part by SRC contract 1361.001, NSF contract CCR-0312676, and by the California Micro program with our industrial sponsors, Altera, Intel, Magma, and Synplicity.

The authors acknowledge Tim Cheng and Feng Lu from UC Santa Barbara for their help in running the experiments using CSAT reported in Table 3, and Ganapathy Parthasarathy from Real Intent for helpful discussions.

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems testing and Testable Design*, Computer Science Press, 1990.

[2] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*, *Release 51205*. http://www.eecs.berkeley.edu/~alanmi/abc/

[3] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[4] D. Brand. "Verification of large synthesized designs." *Proc. ICCAD '93*, pp. 534 -537.

[5] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[6] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, Vol. C-35(8), Aug 1986, pp. 677-691.

[8] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions", *Proc. CAV '02*, LNCS, Vol. 2404, pp. 78-92.

[9] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*, pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[10] CSAT: UCSB circuit SAT solver: http://cadlab.ece.ucsb.edu/downloads/CSAT.htm

[11] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan, "Logic synthesis through local transformations," *IBM J. of Research and Development*, Vol. 25(4), 1981, pp 272-280.

[12] N. Een, "Description of Cadence MiniSat", SAT-Race 2006. http://fmv.jku.at/sat-race-2006/

[13] N. Een and N. Sörensson, "An extensible SAT-solver", *Proc. SAT '03,* pp. 502-518. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[14] N. Een and N. Sörensson, "MiniSat: A SAT solver with conflict-clause minimization ". *Proc. SAT '05.*

[15] N. Een and N. Sörensson, “Translating pseudo-Boolean constraints into SAT”. Vol. 2 (2006), pp. 1-26. http://jsat.ewi.tudelft.nl/content/volume2/JSAT2_1_Een.pdf

[16] C. A. J. van Eijk. ”Sequential equivalence checking based on structural similarities”, *IEEE TCAD*, Vol. 19(7), 2000, pp. 814-819.

[17] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik, “Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver”, *Proc. DAC ’02*, pp. 747-750.

[18] L. Hellerman, "A catalog of three-variable Or-Inverter and And-Inverter logical circuits", *IEEE Trans. Electron. Comput.,* Vol. EC-12, June 1963, pp. 198-223.

[19] H. S. Jin, M. Awedh, and F. Somenzi, “CirCUs: A Satisfiability Solver Geared towards Bounded Model Checking”. *Proc. CAV ’04*, pp. 519-522.

[20] A. Kuehlmann, A. Srinivasan, D. P. LaPotin, “Verity - A formal verification program for custom CMOS circuits”, *IBM J. of Research and Development*, 1995, Vol. 39(1/2), pp 149-165. http://www.research.ibm.com/journal/rd/391/kuehlmann.pdf

[21] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification”, *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[22] A. Kuehlmann, “Dynamic transition relation simplification for bounded property checking”. *Proc. ICCAD ’04*, pp. 50-57.

[23] F. Lu, L. Wang, K. Cheng, R. Huang. “A circuit SAT solver with signal correlation guided learning”. *Proc. DATE ‘03*, pp. 892-897.

[24] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases”, *Proc. DAC ’03*, pp. 668-673.

[25] F. Lu and K. T. Cheng, “Sequential equivalence checking based on *K*-th invariants and circuit SAT solving”, *Proc. HLDVT Workshop ’05*, pp. 45-52.

[26] Y. Matsunaga, “An efficient equivalence checker for combinational circuits”, *Proc. DAC ’96*, pp. 629-634.

[27] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification". *ERL Technical Report*, EECS Dept., UC Berkeley, March 2005. http:// www.eecs.berkeley.edu/~alanmi/publications/2005/tech05_fraigs.pdf

[28] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE Trans. CAD*, May 2006*,* Vol. 25(5), pp. 743-755.

[29] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf

[30] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *ERL Technical report,* April 2006. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tech06_int.pdf

[31] I.-H. Moon and C. Pixley, “Non-miter-based combinational equivalence checking by comparing BDDs with different variable orders”. *Proc. FMCAD ‘04*, pp. 144-158.

[32] J. Moondanos, C.-J. H. Seger, Z. Hanna, and D. Kaiss, “CLEVER: Divide and conquer Combinational Logic Equivalence VERification with false negative elimination”, *Proc. CAV ’01*, pp. 131-143.

[33] M. Moskewicz, C. Madigan, Y. Zhao, L.Zhang, S. Malik. “Chaff: Engineering an efficient SAT solver”. *Proc. DAC ’01*, pp. 530-535.

[34] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System. UC Berkeley*. http://www cad.eecs.berkeley.edu/mvsis/

[35] Ya. Novikov and R. Brinkmann, “Foundations of hierarchical SAT-solving”. *ZIB -Report 05-38,* August 2005. Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany. http://www.zib.de/Publications/Reports/ZR-05-38.pdf

[36] E. Sentovich et al. “SIS: A system for sequential circuit synthesis.” *Technical Report, UCB/ERI, M92/41, ERL*, Dept. of EECS, UC Berkeley, 1992.

[37] L. G. e Silva, L. M. Silveira, and J. P. Marques-Silva, “Algorithms for solving Boolean satisfiability in combinational circuits”. *Proc. DATE ’99,* pp. 526-530.

[38] J. P. Marques-Silva and L. G. e Silva, “Solving satisfiability in combinational circuits”. *IEEE Design and Test of Computers 2003*, Vol. 20(4), pp. 16-21.

[39] G. L. Smith, R. J. Bahnsen, H. Halliwell, "Boolean comparison of hardware and flowcharts". *IBM J. of Research and Development*, Vol. 26(1), 1982, pp. 106-116.

[40] A. Solar-Lezama, R. Rabbah, R. Bodik, K. Ebcioglu, ”Programming by sketching for bit-streaming programs”, *ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '05*), Chicago, IL, June 2005, pp 281-294.

[41] M. N. Velev, ”Efficient translation of Boolean formulas to CNF in formal verification of microprocessors”, *Proc. ASP-DAC '04*, January 2004, pp. 310-315.

[42] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability”, *Proc. DAC '06*, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_sym.pdf

[43] CEC benchmarks used: http://www.eecs.berkeley.edu/~alanmi/cec