Certificate-Based Combinational Equivalence Checking

Rémi Germe[1,2]

April 2025 - August 2025

Supervisor	Advisor	Contact at École polytechnique
Flavien Solt[2]	Christopher W. Fletcher[2]	Jean-Christophe Filliâtre[1]

[1] École polytechnique [2] University of California, Berkeley

Abstract

Combinational equivalence checking (CEC) ensures that an optimized electronic design indeed behaves like its unoptimized counterpart. However, the complexity of this task and its lack of scalability has so far limited its application to small design portions. We argue that we can leverage knowledge about what the expected effect of optimizations is to drastically reduce the complexity of the equivalence check, especially for large designs. To this end, we introduce optimization certificates, which are emitted by the design optimizer to log circuit modifications and provide hints for the equivalence proof. We present CheCker, our new certificate-guided CEC engine. Our preliminary evaluation shows an average 10x speedup over ABC's state-of-the-art CEC engine.

1 Introduction

Designing modern chips, either application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs), is an increasingly complicated task, notably due to their increasing scale and complexity. Electronic design automation (EDA) software is developed to address this issue and help hardware designers do their work. EDA tools take many different forms, as many distinct operations are performed during the design of a chip.

Overview of EDA pipeline. Generally, the reference specification of a design is human-written using a hardware description language (HDL), such as Verilog. At this point, behavior of the design is expressed at register-transfer level (RTL). Then, before actually being manufactured on silicon, this original design goes through many different passes, such as:

- HDL preprocessing [1], which processes high-level constructions provided by the HDL to a synthesizable subset of the language
- logic synthesis [2], [3], [4], [5], which converts the HDL description to a network of logic gates and memory elements
- logic optimization [6], [7], [8], [9], [10], [11], [12], which attempts at minimizing metrics such as the number of gates, while preserving the logic functionality of the circuit
- technology mapping [13], which maps a network with arbitrary gates to a network using only gates provided by the hardware designer, such as a standard cells library
- place-and-route [14], which determines the geometrical layout of gates and memory elements, as well as the placement of wires connecting them
- parasitic extraction [15], which checks that the computed physical layout does not induce any physical parasitic effects, if two transistors are too close for example.

This description of an EDA pipeline is far from being exhaustive, but gives an overview of some challenges addressed by EDA tools. Note that a design will go through many different representations at different stages of the EDA workflow. We will present the specific representation we are working with later on.

Cost of hardware bugs. The cost of bugs is particularly high in hardware design [16], [17]. A bug discovered after a chip has been manufactured on silicon is generally impossible to fix, requiring to delve back into the design process and the fabrication process, which can lead to millions of dollars in costs and months of delays. This high cost of post-manufacturing bug fixing creates a strong incentive for rigorous verification throughout the design flow.

Formal verification. Obtaining strong guarantees to ensure the absence of bugs is the challenge of formal verification. Many properties can be checked for at different stages of the pipeline, leading formal verification to come in many flavors:

- equivalence checking [18], [19], [20] consists in verifying that the behavior of the design is preserved throughout the EDA pipeline
- some power constraints [21], or timing constraints, can be required for the physical circuit to behave properly
- some critical industries such as aerospace or automotive industries might even be interested in high-reliability designs, resistant to faults such as bit flips [22].

EDA tool reliability challenges. Unfortunately, the EDA tools themselves are not immune to bugs [23], [24], [25], [26]. Recent studies have uncovered numerous critical bugs in commercial and open-source EDA software using techniques like fuzzing. Note that, while effective at finding bugs, fuzzing cannot prove their absence, highlighting the need for formal verification methods to complement testing.

Combinational equivalence checking. This paper focuses on combinational equivalence checking (CEC) [18], [19], [20], [27], [28], [29], [30], [31], [32], which is the process of checking that two given combinational circuits are functionally equivalent. Combinational circuits have no memory elements, only logic gates, and thus do not depend on timing or clock cycles. They are implementations of plain boolean functions. Intuitively, CEC is performed to ensure that circuits which should be equivalent are indeed equivalent. Such circuits might be a reference circuit, and the result of this reference circuit after it has been processed by EDA software. We will later explain how our approach leverage this intuition.

Scalability challenges in formal verification. However, formal verification methods generally come with a high runtime cost, which can prevent wide adoption. This is especially true for large industrial circuits, which can reach millions of gates. CEC is unfortunately no exception. We aim to provide a new, more scalable approach to perform CEC, thus reducing the runtime and leading to a higher adoption rate.

Focus on logic optimizations. We specifically target logic optimizations [6], [7], [8], [9], [10], [11], [12]: we want to verify that circuits before and after optimizations are functionally equivalent. We rely on the idea that a logic optimizer applies transformations precisely because it has already established the functional equivalence between the reference circuit and its optimized version. This intuition will be detailed in the next sections. Then, instead of performing CEC on what would otherwise be considered two arbitrary circuits, we leverage the knowledge of the specific operations which were applied to guide the proof of equivalence.

ChECker. We apply this principle by modifying the logic optimizer to emit certificates, detailing the operations performed and providing some hints to facilitate the future proof of equivalence. These certificates are then used by an independent program we developed, ChECker, which implements a novel certificate-guided proof strategy to efficiently verify the equivalence.

We focused on the logic optimizer ABC [2], [3], which is used as technology mapper (including logic optimizations) in the open source synthesizer Yosys [4], [5]. Both ABC and Yosys are state-of-the-art open source EDA tools. ABC performs optimizations [7], [8], [10], [11] on designs represented as and-inverter graphs (AIGs) [10], [33], [34], [35]. ABC is also the most efficient open source CEC engine, and will serve as the primary benchmark for evaluating our new engine. AIG is the only design representation considered in this paper, as they are also central to CEC.

By focusing the verification effort on the specific changes made, we expect to achieve better scalability on large designs.

Contributions. Our contributions are as follows:

- We design generic certificates, that is certificates which can be used to describe any arbitrary logic optimization pass, and implement them in the ABC logic optimizer.
- We design and implement, in a program named ChECker, a new combinational equivalence checking workflow using these certificates.
- We achieved a x10 speed up as preliminary results. In the upcoming weeks, we plan to improve our use of certificates in ChECker, and benchmark our new workflow on large designs, planning for strong improvements for large circuits when comparing results to the state-of-the-art `&cec` engine in ABC.

Open sourcing. The source code of our underlying AIG library, implementing mutations described by certificates, is available on https://github.com/remigerme/mutaig. The source code of ChECker will be open-sourced later.

2 Background

In this section, we provide background on and-inverter graphs, logic optimization passes, and finally on combinational equivalence checking.

2.1 And-inverter graphs

RTL circuits can be seen as sequences of combinational circuits. And-Inverter Graph (AIG) is a data structure representing such combinational circuits [33]. An AIG is a directed acyclic graph (DAG), where nodes are either primary inputs or outputs or `AND` gates, and where edges can be complemented using a `NOT` gate. These two gates are sufficient to express arbitrary boolean logic. To support sequential circuits, AIGs may also contain latches, used to represent memory elements. Figure 1 shows an example of an AIG for a half adder.

Incoming (resp. outgoing) edges of a node are called fanins (resp. fanouts). The transitive fanin (resp. fanout) of a node includes all nodes reachable by following fanins (resp. fanouts) through the DAG.

AIGs can be stored efficiently in binary files using the AIGER format [34], [35].

2.2 Logic optimization passes

The AIG directly extracted from an RTL design might not be optimal in two major metrics: the number of logic gates and the depth of the graph. These metrics are particularly important as the number of logic gates correlates with the physical gate count, and thus the chip size, while the depth correlates with the delay between transistors, affecting circuit timing. Although there is no direct equivalence between these AIG metrics and their physical counterparts, strong correlations have been observed experimentally [10].

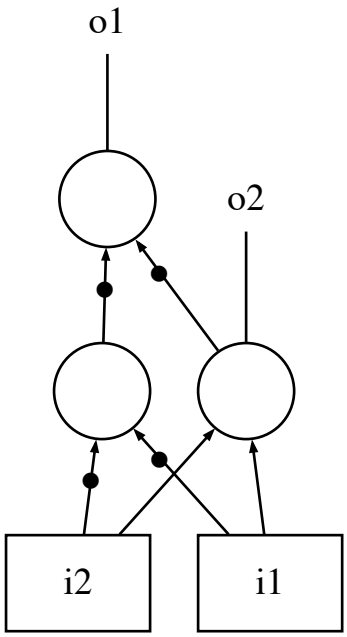

Figure 1: The AIG of a half-adder. Complemented edges are indicated with black dots. Here, o_1 is the sum bit, and o_2 is the carry bit.

Logic optimizers such as ABC [2], [3] perform sequences of optimization passes to improve these metrics while preserving the logic function represented by the AIG. These passes include rewriting, refactoring, and balancing [7], [8], [10], [11]. They preserve or improve the overall number of nodes and the maximal depth of the AIG. Rewriting and refactoring also guarantee that the depth of individual nodes will not increase, whereas balancing can increase the depth of some individual nodes.

Rewriting. The rewriting pass aims to locally optimize the AIG by replacing small subgraphs with more efficient, functionally equivalent ones. This process relies on the concept of cuts. A cut of a node is a set of nodes, called leaves, that separates the node from the primary inputs. Any path from a primary input to the node must pass through at least one leaf of the cut. Intuitively, a cut defines a subgraph whose inputs are the leaves. This subgraph is a candidate for replacement by a better, equivalent one.

To find equivalent subgraphs, the notion of NPN equivalence is used (NPN stands for Negation-Permutation-Negation). Two AIGs are NPN-equivalent if one can be transformed into the other by negating inputs, permuting inputs, and negating the output. There are only 222 NPN classes of circuits having four inputs [6], way fewer than the 2^{16} boolean functions of four inputs. For each of these 222 classes, different equivalent AIG representations have been precomputed and stored.

The rewrite of a single node goes as follows:

- different cuts with up to four leaves are computed, then for each cut:
  - the NPN class of the subgraph defined by the cut is determined
  - all precomputed AIG representations for this NPN class are considered
- the best subgraph in the current context is chosen to rewrite the node

The choice of the optimal subgraph is determined by its cost, which is computed as the number of new nodes that must be added to the AIG if the subgraph is used. For example, if the nodes for the left AIG in Figure 2 already exist and can be reused, while the right variant

requires creating new nodes, the left one is preferred as it minimizes the number of additional nodes.

The rewriting pass attempts to rewrite all nodes of the AIG in topological order.

Refactoring and balancing. In the refactoring pass, each node is resynthesized by looking at its cuts and trying to factorize the boolean function of the node. Cuts with more than four leaves are considered. Balancing aims at reducing the maximal depth of the AIG using heuristics to determine the new structure of the AIG.

ABC provides a high-level interface `dc2` command, which is a sequence of these three commands and is used, for example, by the Yosys synthesizer [4], [5].

2.3 Combinational equivalence checking

Handling of sequential circuits. Sequential circuits are sometimes checked for combinational equivalence checking. In that case, latches, representing sequential elements, are treated as pseudo-inputs for their fanouts, and they also are considered as pseudo-outputs. As primary outputs, pseudo-outputs are checked for equivalence.

Miter circuit. Two circuits are equivalent if for any inputs, they provide the same outputs. Both primary inputs (resp. outputs) and pseudo-inputs (resp. outputs) are considered in the same way. Figure 2 depicts a miter construct, where two circuits checked for equivalence are supplied with the same inputs, and their outputs are `XORed` together [20], [30].

The two circuits are equivalent if and only if the `XOR` output Z cannot be different from zero for any inputs. Hence, to verify the equivalence between two circuits, the value of Z is checked for any possible inputs.

A SAT formula in conjunctive normal form (CNF) describing the circuit assuming $Z = 1$ is extracted from the miter circuit using the Tseitin transformation described below. This formula is handed to a SAT solver. If the CNF is `UNSAT`, then the assumed output $Z = 1$ cannot occur for any inputs and the two circuits are equivalent. However, if CNF is `SAT`, then the boolean valuation satisfying the CNF is mapped to a valuation of inputs where the two circuits differ.

When circuits have multiple outputs, their supposedly equivalent outputs are `XORed` pairwise. Each pair of outputs can then be checked independently, or an additional `OR` gate is can be added as a fanout of every `XOR` gate. In this case, circuits are equivalent if the output of the final `OR` gate is zero, which is equivalent to all outputs of `XOR` gates being zero.

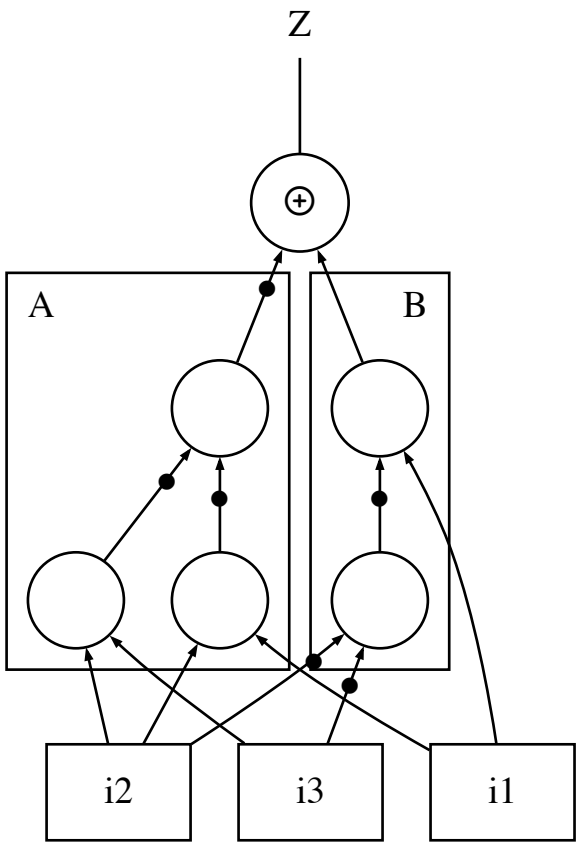

Figure 2: A miter between two circuits. On the left: $(i_1 \wedge i_2) \vee (i_1 \wedge i_3)$, on the right: $i_1 \wedge (i_2 \vee i_3)$.

Tseitin transformation. The constraints imposed by logic gates can be expressed as SAT clauses using the Tseitin transformation [36]. Each node in the miter is mapped to a SAT literal. To generate the SAT clauses, the transformation creates clauses that enforce the gate's logical behavior by forbidding any assignment where the inputs and output do not match the gate's truth table. Table 1 illustrates the Tseitin transformation for an `AND` gate.

The overall SAT formula generated for the `AND` gate is:

$$(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$$

a	b	$c = a \wedge b$	SAT clause
0	0	0	$a \vee b \vee \neg c$
0	1	0	$a \vee \neg b \vee \neg c$
1	0	0	$\neg a \vee b \vee \neg c$
1	1	1	$\neg a \vee \neg b \vee c$

Table 1: SAT clauses generated from `AND` truth table. a, b and c are SAT literals mapped to nodes in the miter.

which can be simplified into

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c).$$

When an edge in the AIG is complemented, the corresponding SAT literal is simply negated in the generated clauses. A similar procedure can be applied to generate clauses for `XOR` and `OR` gates within the miter. At most four clauses are generated for each gate. Using this approach, a SAT formula in conjunctive normal form (CNF) of linear size in the number of nodes in the AIG can be extracted.

3 Motivation and Challenges

In this section, we present the limitations of current CEC that motivate our approach. Then, we describe challenges which will guide the design of the certificates and the new CEC engine.

3.1 Limitations of current CEC

SAT limitation. Although the generated SAT formulas are in linear size in the number of gates, modern circuits can easily reach millions of gates. Despite significant improvements to SAT solvers in recent years [37], [38], [39], solvers are still not able to handle these giant monolithic SAT queries. Indeed, these queries are describing the whole transitive fanin of a pair of outputs at once.

Node merging. Combinational equivalence checkers, like ABC, adopt a divide and conquer approach to address this limitation. Identifying internal nodes of the two circuits that are equivalent, and merging them, allow to simplify the future formulas that will be extracted from their fanouts. Finding candidate nodes to be merged is a complicated task, relying on intelligent simulations to determine equivalence classes of nodes [19], [30], [31]. To prove that two internal nodes are equivalent, they are simply `XORed` together and a SAT formula is extracted. If the nodes are not equivalent, the SAT solver is returning a valuation of inputs such as the two nodes differ. This valuation can be reused in subsequent simulations. In practice, such simulations rely on heuristics in order to refine as much as possible supposed equivalence classes. Simulations and SAT queries are alternated until the equivalence of the two circuits can be either proven or disproven.

Table 2 presents two metrics computed on two benchmarks: the EPFL combinational benchmark [40] (excluding the "More than ten Million gates" circuits for now for runtime reasons), and also the BEnchmark for Explicit Model checkers (BEEM) [41]. Circuits of the first benchmark are presented in Table 3. The reference circuits were compared to their optimized versions, obtained using the `&dc2` pass from ABC. The first metric is the fail rate of SAT queries when trying to merge two supposedly equivalent nodes, that is, the proportion of nodes wrongly classified as equivalent. The second is the proportion of the time spent doing simulations when trying to merge nodes. Note that, the rest of the time was dedicated to SAT queries which can themselves be unsuccessful.

Benchmark	Average SAT fail rate	Average proportion of time spent on simulations
EPFL	0.10	0.23
BEEM	0.38	0.10

Table 2: Metrics on simulation-based `&cec` engine of ABC. The table shows the average rate of incorrect equivalence candidates found by simulation (SAT fail rate) and the proportion of runtime dedicated to these simulations. Detailed per-circuit data for the EPFL benchmark is available in Appendix A.

Opportunity to leverage optimization knowledge. In practice, the two circuits checked for combinational equivalence are not unrelated. One is the reference design, whereas the other is a revision of this design, or the result of transformations performed by EDA software. Logic optimization passes, in particular, are applied with the explicit goal of preserving functional equivalence. This implies that the optimizer already possesses knowledge about the equivalence between specific subcircuits. Current CEC workflows generally ignore this information, instead treating the two circuits as arbitrary. This idea has not been explored yet, and constitutes the core of our approach. By incorporating knowledge from the optimization process, it may be possible to guide equivalence checking more efficiently, especially for large designs. Ideally, simulations would be unnecessary, and it might even be possible to determine some nodes to merge to reduce the problem just enough to handle it directly using a SAT solver.

3.2 Overview of the challenges

Our analysis suggests that we might significantly improve CEC efficiency by leveraging the optimizer's knowledge. We provide an overview of the challenges based on our previous observations and how we address them in the rest of this paper.

First challenge: designing a certificate format. The first challenge is to express the knowledge known by the optimizer, in a useful yet compact way. The core challenge is to formalize this intuition of knowledge into a concrete data format. Also, note that the specific information contained in the certificates will shape the pipeline of the proof of equivalence. Section 4 describes the requirements we listed for such certificates, and the format we adopted.

Second challenge: design an efficient CEC pipeline using certificates. The second challenge is to design a CEC pipeline that effectively leverages these certificates. Having access to optimization knowledge is only the first step, another difficulty lies in creating a proof strategy that uses this information to guide the verification process. This new pipeline should avoid the heuristics of traditional methods, such as simulation, by using the certificate data to intelligently simplify the problem. The ultimate goal is to perform the minimal number of node merges required to make a final reasonable monolithic SAT query, and accelerate the overall equivalence check. Section 5 details the certificate-guided workflow we have developed to address this challenge.

Third challenge: implementation. Finally, our goal is to develop an efficient CEC engine. To achieve better results than the currently optimized CEC engine of ABC, providing efficient implementations of our ideas is required, especially as we are targeting large designs. This implementation should be as fast as possible, but also make sure to use a reasonable amount of memory, as memory use can become a concern when handling AIGs with millions of nodes. The first goal is of course to have a working pipeline, but efficiency will be a central concern in the implementation. Section 6 provides details on implementations, both within ABC and our independent program ChECker.

4 Optimization Certificates

In this section, we present the requirements we listed to create concrete certificates, and we describe the adopted format.

4.1 Certificate requirements

We present different considerations on certificates, which will lead to the upcoming specification.

Certificates are not trustworthy. Certificates themselves should not be trusted. They are emitted by the logic optimizer, which we are trying to verify and is not part of the trusted computing base (TCB). They can suggest a proof, but this proof should be replicable independently from the validity of the certificate. If a certificate suggests an invalid proof, we want our certificate-guided CEC engine to realize that the "proof" suggested by the certificate is invalid. This is key to ensure soundness of our CEC engine.

Providing valuable information. The reason for being of certificates is to provide valuable information when trying to prove equivalence. The scalability limitations of CEC arise from a vast search space, which leads to monolithic SAT queries that are too large for modern solvers. Moreover, identifying relevant mergeable internal nodes to simplify the problem is a complex task.

Ideally, certificates would provide these equivalent internal nodes, thus removing the need for costly simulations and speculative SAT queries. However, a single rewriting pass for example does not represent only one transformation, but rather a sequence of many "atomic" rewrites (that is, one node being rewritten). This raises a new concern: how to store these transformations efficiently.

Size constraints. A naive approach would be to dump the entire AIG state after each atomic rewrite. However, this is impractical for large designs where millions of atomic rewrites can occur. A single AIG can already occupy hundreds of megabytes, making it infeasible to store millions of such dumps. Therefore, the certificates must be compact to avoid becoming a storage bottleneck themselves.

Retrieving intermediate states with generic certificates. Because it is not possible to dump every intermediate AIG state, we need to be able to retrieve these intermediate states from the original input design and the certificates.

Moreover, certificates need to be generic, ie they should not be bound to a specific optimization pass. This requirement ensures that certificates can be emitted by any logic optimization pass deemed relevant.

4.2 Certificate specification

We now present the concrete specification of our certificates. Certificates are split in two distinct components:

- mutations, which describe operations performed on the circuit
- hints, which provide information to be used when checking the equivalence.

Mutations. Mutations allow us to reconstruct intermediate states. The operations applied by the optimizer are replicated. The key challenge is to be able to refer to one specific node in an unambiguous way, from the optimizer but also from our CEC engine. To do that, we attach permanent `id`s to the AIG nodes. Mutations are one of the following:

- `CREATE(id, f0, f1)` when a node (an `AND` gate) identified by `id` is created with fanins `f0` and `f1`
- `REPLACE(old_id, new_id, compl)` when node `old_id` is replaced by node `new_id`. Note that, due to NPN equivalence, a node can be replaced by a new node that is its functional complement. The `compl` flag indicates whether such a complementation occurred. If this is the case, the fanouts of the replaced node must be complemented too to maintain logical consistency.

Originally, a mutation `SET_FANIN(id, old_fanin, new_fanin)` was also considered, but it was dismissed because it was not required to model the targeted passes (rewrite, refactor, balance). Note that to support arbitrary modifications on AIG, this mutation (or another one) would be required.

Hints. Hints are just pairs of nodes that are known to be equivalent when optimizing the circuit. Some hints are redundant with `REPLACE` mutations, but there is no a priori reason for hints to be limited to nodes being replaced. They provide a set of equivalent internal nodes to be merged.

5 Prove Equivalence Using Certificates

In this section, we present our certificate-guided proof-of-equivalence strategy implemented within ChECker. We begin by presenting a naive strategy, which we then refine into a more efficient approach.

5.1 Naive approach

This section presents a working approach, which faces limitations that motivate the more efficient strategy.

Naive pipeline. Figure 3 depicts the naive CEC pipeline. A full proof of equivalence between the current candidate representation and the next one is performed for each optimization pass. For the `&dc2` script which contains 9 optimization passes, this leads to 9 full proofs of equivalence.

Proof of equivalence using hints. Once that the next candidate AIG has been reconstructed using mutations, the goal is to prove the full equivalence between

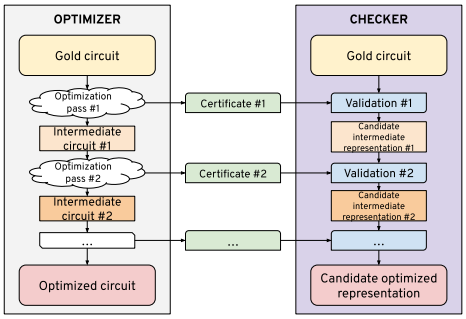

Figure 3: The naive proof-of-equivalence strategy. Each optimization pass generates a certificate, allowing to reconstruct the next candidate intermediate representation using mutations. Then, using hints, a full proof of equivalence between the circuit before the optimization pass and the next candidate intermediate representation is established.

the original AIG (denoted C_i) and the next candidate (denoted C_{i+1}). The algorithm consists in merging progressively each pair of nodes in $C_i \times C_{i+1}$ that can be merged:

- create a miter M from C_i and C_{i+1}
- for each node n in C_i
  - if $n \in$ hints and hints$[n] \in C_{i+1}$
    - extract a cnf from the miter from $n(C_i)$ and hints$[n]$ (C_{i+1})
    - if the cnf is `UNSAT`, nodes can be merged
  - else if n is obviously mergeable (i.e. a node n' exists in C_{i+1} with the same `id` and fanins as n, and all corresponding fanin pairs have already been merged), nodes can be merged
- return true iff all outputs of the miter have been merged

ChECker output. Our approach differs from traditional equivalence checkers. Traditional checkers, when they do terminate, either prove or disprove equivalence. ChECker is not able to disprove equivalence, only to prove equivalence or fail in the process. As circuits are usually checked and not tested for equivalence (note the terminology), we believe this is not an important limitation.

Missing certificate. Note that the pipeline presented above is not resilient against missing certificates: if the design goes through an optimization pass which does not emit a certificate, then we cannot reconstruct the intermediate target. A fallback solution could be

to provide a dump of the full target AIG, and use traditional CEC, hoping that the two circuits share similarities making the CEC easier. This fallback mechanism was not implemented in ChECker, as we believe that a missing certificate would severely degrade performance.

Temporary nodes. An issue emerges from the fact that the proof of equivalence is performed once the next candidate has been fully reconstructed. Indeed, for a rewrite pass, which may contain millions of atomic rewrites, some nodes might only exist temporarily. For example, consider a node `X` that is replaced by a new node `A`. Later in the same rewrite pass, if the only fanout of `A`, a node `B`, is itself replaced by another node `C`, then `A` becomes disconnected from the circuit's outputs. As a result, `A` is redundant and will be removed (dropped). This means `A` existed only temporarily during the rewrite pass and is not present in the final, fully reconstructed AIG. This first strategy will not make any use of `A`.

Trade-off between number of nodes merged and difficulty of SAT queries. This approach is trying to merge as many internal nodes as possible. In practice, this can lead to generate too many SAT queries. This is unnecessary as we are simplifying again and again a problem which was already solvable by a SAT solver. The bottleneck is then the number of SAT queries rather than their complexity. We believe there is a sweet spot to find between submitting one giant monolithic SAT query and too many tiny SAT queries. The refined strategy described in the next section addresses this trade-off problem.

5.2 More efficient strategy

The aim of this strategy is to prove the equivalence of as few nodes as possible, while still allowing to submit a reasonable SAT query describing the outputs of the miter. This section is more a draft as designing in details the efficient strategy will be upcoming work.

Intuition motivating the strategy. Figure 4 depicts the tradeoff previously mentioned between one monolithic SAT query (unsolvable) and too many (easy) SAT queries. The idea is that the complexity of a SAT formula extracted from two nodes can be estimated by looking at the size of the transitive fanins of these two nodes. Indeed, each node in the transitive fanins yield 3 SAT clauses.

Evaluating useful merges. Rather than merging each possible node from the inputs to the outputs, we identify relevant nodes to merge. To do that, we start from the outputs. If their transitive fanins are sufficiently small, we can directly extract a SAT formula to prove their equivalence. Else, we search some nodes in their fanins that can be merged, applying the procedure recursively if their transitive fanins are too large.

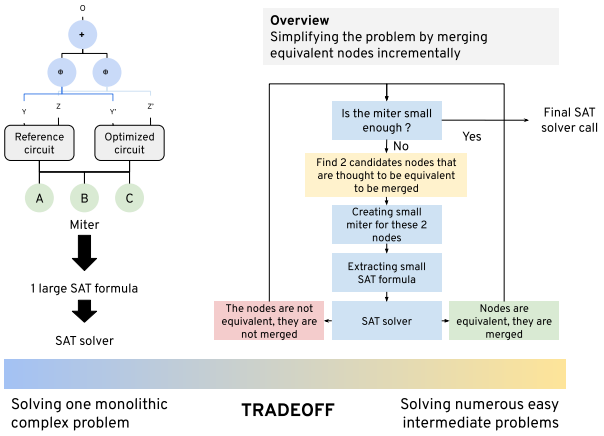

Figure 4: The tradeoff motivating our approach: one (too) complex SAT query vs (too) many easy SAT queries.

Impact of don't cares. We mention a failed approach. Instead of building a SAT formula that spans from a node back to the primary inputs, we attempted to treat already-merged nodes as pseudo-inputs. This would mean the SAT formula for a node would only consider its transitive fan-in up to these merged pseudo-inputs, not the actual primary inputs. Unfortunately, this approach is incomplete. By treating internal nodes as unconstrained inputs, we allow value combinations that would never occur in the real circuit (known as don't cares). A SAT solver could find such a counterexample, making the equivalence proof fail for nodes that are, in fact, equivalent.

6 Implementation

The implementation consists of two parallel projects:

- some modifications to ABC to emit the certificates.
- our CEC engine ChECker.

6.1 Modifications to ABC

Certificates for rewrite have been implemented. Certificates for refactor have not been implemented yet, however, we anticipate a straightforward implementation due to the structural similarities between the two passes. Certificates for balance have not been implemented yet.

Practical difficulties. Although these modifications are not conceptually difficult, we encountered notable practical difficulties. Becoming familiar with the large undocumented C codebase (2000 files, 700,000 lines) was quite a challenge.

Finding the particular relevant implementation for our use case was also non-trivial, as different commands have both an old and a newer implementation. For example `rw` (for rewrite) should absolutely not be used, the command is now `drw`. Sometimes newer commands are prefixed with a `&` (`&cec` performs far better than `cec`), but this prefix does not always indicate a newer command (`&dc2` just calls `dc2`).

It was also the opportunity for some C discoveries, such as the use of tagged pointers. The internal AIG struct uses the least significant bit of pointers to AIG nodes to store the flag indicating whether the edge to this node is complemented. Accessing a field of such a pointer requires caution to avoid errors.

6.2 CheCker

On the CheCker side, the work was divided into two separate projects:

- a Rust AIG library, mutaig, to provide the required trackable and mutable AIGs
- CheCker itself, heavily relying on mutaig as a dependency.

This allows to separate the logic between the AIG operations and the proof methodology. The initial development cost of the library was very high, and the pipeline has only recently become (almost fully) operational.

SAT solver integration. The currently used SAT solver is Kissat [37], [39] but any SAT solver could be plugged in easily instead.

Missing features. Unfortunately, some features are still missing (but are currently under active development):

- constant signal propagation: an `AND` gate cannot be replaced by a primary input yet (easy)
- smart construction of the miter using structural hashing (technique known as strashing) to identify duplicate nodes to allow for faster verification (especially if circuits are equal) (medium)
- debug a serialization issue: AIG nodes are identified uniquely using `ids`, but the very last check of the pipeline compares a candidate circuit (with `ids` set) to a freshly parsed AIG file (containing the optimized circuit, with `ids` not set). We designed and implemented a normalization procedure to address this issue, but a bug remains for now (unknown difficulty).

7 Preliminary Evaluation

In this section, we provide a preliminary evaluation of our current implementation. First, we demonstrate that certificates have a reasonable size in practice, then, we compare CheCker to ABC `&cec` engine.

Methodology of the preliminary evaluation. The results presented in this section were obtained on the circuits of the EPFL benchmark. Table 3 presents these circuits. For now, we exclude the three More than ten Million gates (MtM) circuits: `sixteen`, `twenty`, `twentythree`. The optimized version of each circuit was obtained by applying a rewrite pass using the `drw` command in ABC.

All experiments were run on a a server equipped with an Intel(R) Xeon(R) Gold 6354 CPU running at 3.00GHz.

7.1 Size of the certificates

A key requirement for the practical adoption of our certificate-based approach is that the certificates themselves remain storable. This section provides a preliminary analysis of their size overhead. For each circuit in the EPFL benchmark, Figure 5 plots the certificate-to-AIG size ratio for a single rewrite pass (`drw`), as well as the number of nodes of the original AIG.

Analysis. Ultimately, the key metric will be the overall certificate size for the entire `&dc2` script. This script

Name	Inputs	Outputs	`AND` gates	Depth
adder	256	129	1020	255
bar	135	128	3336	12
div	128	128	57247	4372
hyp	256	128	214335	24801
log2	32	32	32060	444
max	512	130	2865	287
multiplier	128	128	27062	274
sin	24	25	5416	225
sqrt	128	64	24618	5058
square	64	128	18484	250
arbiter	256	129	11839	87
cavlc	10	11	693	16
ctrl	7	26	174	10
dec	8	256	304	3
i2c	147	142	1342	20
int2float	11	7	260	16
mem_ctrl	1204	1231	46836	114
priority	128	8	978	250
router	60	30	257	54
voter	1001	1	13758	70

Table 3: Statistics of the circuits from the EPFL combinational benchmark (excluding More than ten Million gates (MtM) circuits).

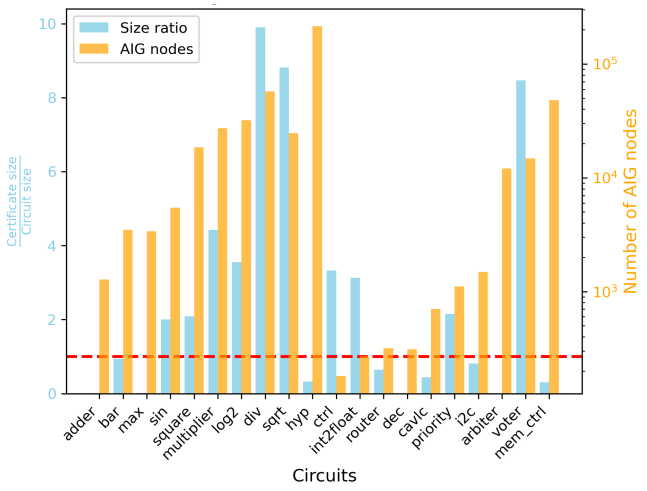

Figure 5: Size ratio (disk size of the certificate over disk size of the AIG circuit) and the number of AIG nodes have been plotted for each circuit of the EPFL benchmark.

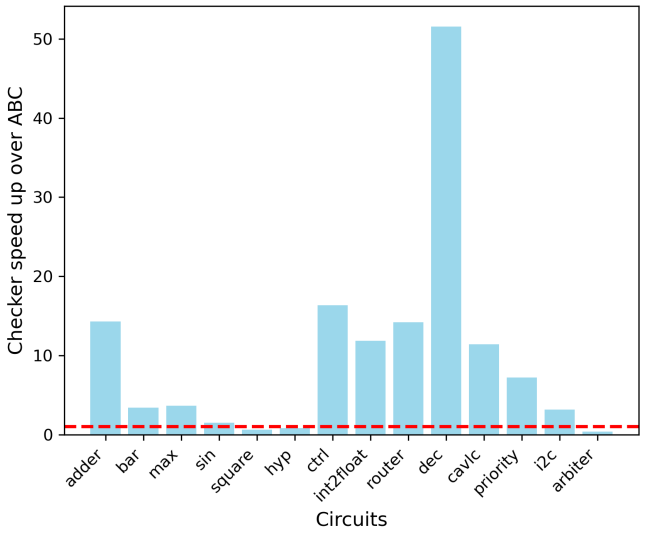

Figure 6: Comparing CEC runtime: ABC `&cec` vs CHECKER. Red line represents a speed up of 1. CHECKER shows a significant performance improvement over ABC on most benchmark circuits.

performs 9 optimization passes: 4 rewrites, 3 balances, 2 refactors.

For the rewrite pass, the observed overhead of at most 10x the original AIG size is acceptable.

We expect a similar overhead from refactor, as both passes perform similar local transformations.

Balancing, however, is a global pass, which could lead to larger certificates. Upcoming work will have to take that into account.

7.2 Comparing CEC runtime

Figure 6 presents CHECKER speedup over ABC `&cec` engine. We already achieved a 10x speedup on average over the circuits of the EPFL combinational benchmark. Without considering the `dec` circuit, we got an average speedup of 6.8x. This is a very promising start towards efficient CEC for large designs, acknowledging that we built a solid infrastructure.

Missing circuits. Circuits `multiplier`, `log2`, `div`, `sqrt`, `voter`, and `mem_ctrl` do not appear in Figure 6 because they require the constant signal propagation feature mentioned in Section 6, which was not implemented yet.

Analysing outliers. This paragraph investigates the presence of performance outliers. Notably, `dec` runs 50x faster on CHECKER, while `square`, `hyp`, and `arbiter` run faster on ABC (by 1.7x, 1.2x, and 2.6x respectively).

To better understand the role of certificates, we plotted the speedup as a function of the number of AIG nodes in Figure 7. The speedup is decreasing with the number of nodes. We hypothesize that optimizing these circuits generates few certificates, as few atomic rewrites are performed (`dec`, `adder`, `arbiter` are not rewritten at all). In such cases, ABC performance is likely due to supporting miter strashing described in Section 6. While the absence of this optimization in CHECKER is not an issue for small circuits like `dec` or `adder`, it becomes a critical factor for large unchanged circuits. Observing results on larger circuits for which more certificates are emitted, such as `div` and `sqrt`, will be crucial for evaluating CHECKER.

7.3 Future evaluation

Missing results. The first thing will obviously be to redo the previous evaluations with the six missing circuits, and using the whole `&dc2` script instead of a rewrite `dwr` pass only.

Methodology for large designs. Then, our ultimate goal is to benchmark CHECKER on large designs. We are

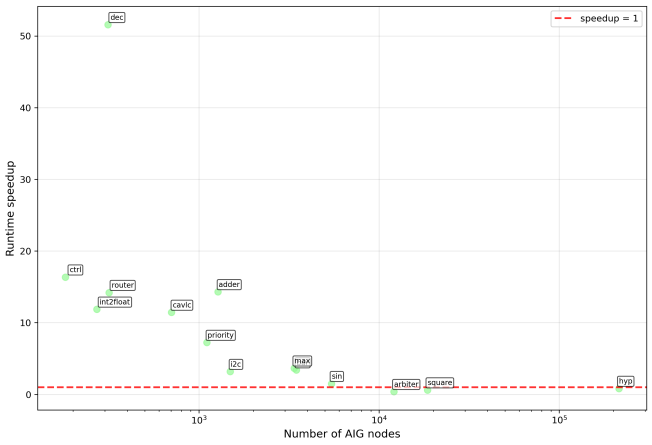

Figure 7: Runtime speedup is a decreasing function of the number of nodes.

planning to proceed as follows. First, include the three More than ten Million gates (MtM) circuits of the EPFL benchmark (circuits `sixteen`, `twenty`, `twentythree`). Second, create arbitrarily large combinational circuits by repeatedly doubling the existing circuits of the EPFL benchmark, using the methodology described in [31]. Third, we plan to also consider open source CPUs such as picorv32 [42], kronos [43], CVA6 [44], rocket [45], OpenC910 [46], Berkeley Out-of-Order Machine (BOOM) [47]. The practice of using sequential circuits to benchmark CEC engines is well-established, as seen in [30] and [31].

Expected results. By implementing our smart proof of equivalence strategy, we expect significant improvements for large designs, with speedups on the order of 100x. This would reduce CEC runtime from hours to minutes, drastically cutting the cost of this verification technique. In some cases, we anticipate even larger improvements, enabling CEC for circuits previously considered not verifiable (with runtimes exceeding a month). This represents a fundamental breakthrough in CEC capabilities.

8 Discussion

We first address a remark on our current pipeline, and discuss two limitations that provide room for extensions.

On integrating the equivalence checks at optimization time. We first want to address a legitimate question about the possibility of integrating the equivalence checks directly within the logic optimizer. The current pipeline of CheCker mimics the optimization passes flow so closely it is natural to consider internalizing it within ABC and alternating optimization passes and intermediate equivalence checks.

We believe there are two arguments against this. The first is that we are in favor of a smaller trusted computing base (TCB), which seems particularly relevant in the context of formal verification. ABC is a huge software which is difficult to delve into. We believe a smaller, independent program is easier to review and thus is less bug-prone. The second argument is that, by externalizing the equivalence checks of these optimizations, we can replicate the proof of equivalence any time we want, as long as the certificates are stored.

On optimizations. The optimization passes considered in our work are complex operations (rewriting, refactoring, and balancing) performed on AIGs. While AIGs enable for sophisticated logic optimizations, it seems that many optimization passes are simpler. For example, the widely used open source synthesizer Yosys performs simpler optimizations on networks directly. An example of such a pass is a looking at chains of multiplexers and try to simplify them. The `dc2` script from ABC is used by Yosys when doing technology mapping. These network passes have not been investigated.

On sequential equivalence checking. While having efficient CEC is relevant, it is not sufficient for real-world formal verification of designs. Purely combinational blocks can be limited in size, and such blocks can be intertwined with memory elements representing registers. Sequential equivalence checking (SEC) aims at verifying that two sequential circuits, featuring memory elements, have the same temporal behavior [48]. It is not clear how much the ideas presented in this report can be extended to SEC. As they often rely on different proof methodologies than CEC, using finite-state automatons for example. We believe there is at least one reasonable use case, which is unrolling. Unrolling one time consists in replacing a memory element by its input combinational block. It is possible to apply this procedure an arbitrary number k of times, thus leading to a purely combinational circuit describing a sequential circuit over k clock cycles [49]. However, this only provides a bounded proof of equivalence, that is, sequential equivalence guaranteed over a bounded number of clock cycles.

9 Related Work

We first present some orthogonal work that could be integrated within our pipeline, then we acknowledge different formal verification approaches.

Parallel CEC. Some work was conducted to parallelize the traditional CEC workflow [31], [32]. Let's recall that this workflow consists in alternating simulations identifying classes of nodes that might be equivalent, and running SAT queries to merge these nodes or check if the `SAT` or `UNSAT` state of the miter can be determined. Authors of [31] identified two main opportunities for parallelization, which lead to three parallelization techniques : two focused on each opportunity individually, and one hybrid.

The first is to parallelize the handling of each pair of outputs in the miter (and their transitive fanins). The second consists in running SAT queries in parallel to prove (or disprove) the equivalence of internal nodes that were flagged as potentially equivalent. Using these two techniques, and combining them, lead to signifi-

cant improvements by a factor of up to 60 compared to the single-threaded existing workflow.

Although we didn't implement any parallelization, we believe those two techniques can easily be replicated in our situation, and integrated in our pipeline. Details such as nodes appearing in multiple transitive fanins need to be taken into account, but they were already addressed by the previous paper implementation. Parallelizing our current implementation would be a natural extension to improve performance.

Circuit-aware SAT solvers. Some specialized circuit-aware SAT solvers have emerged [50], [51], [52]. These solvers leverage the structural properties of circuits to guide the search process, aiming at improving performance on equivalence checking problems. Such SAT solvers could reasonably replace generic SAT solvers currently used in ChECker.

Non-miter based combinational equivalence checking. Historically, combinational equivalence checking was mainly done using binary decision diagrams (BDDs) [18], [19], [27], [29]. BDDs provide a canonical representation of boolean functions, meaning that two functions are equivalent if and only if their BDDs, for a given variable ordering, are isomorphic. This property makes the equivalence check trivial once the BDDs are constructed. However, building BDDs suffer from memory size explosion, as the size of a BDD can be exponential in the number of input variables. Because of this limitation, SAT-based approaches using miters, which proved to be more scalable, have become the standard.

Certifying compilers. Our certificate-based approach can be compared to proof-carrying code [53] and certifying compilers [54]. Certifying compilers generate both an executable binary and some proof artifacts, bundled within the binary, which can be later used by a verifier program to check properties on the binary. There is an obvious parallel with our situation, where the logic optimizer emits certificates, that are then used to prove equivalence of the two circuits.

Yet, the situations are different because equivalence checking involves comparing the semantics of two circuits, whereas proof-carrying code is more about checking properties on one given program. This approach is closer to model checking [49], [55], where properties (for example, timing properties) are proven on a given circuit.

Formally verified logic synthesizers. Rather than checking equivalence of reference and optimized circuits for each logic synthesis, it might be possible to prove the correctness of the logic synthesizer itself once and for all. This approach completely eliminates the need of performing equivalence checking on particular circuits that were provided to the logic synthesizer, as it would have been proven correct (*i.e.* preserves the semantics of the circuit). The most mature project using this approach is the formally verified C compiler CompCert [56] from the software world. Similar work is being done in the hardware world to formally specify the semantics of hardware description languages (HDLs) such as Verilog, as well as building semantics-preserving formally-proven design compilers [57], [58], [59], [60].

10 Conclusion and Remaining Work

Remaining work. This project is still in active development, and significant improvements to ChECker are planned. We plan to proceed as follows:

- implement missing features in mutaig, such as constant signal propagation, serialization (which also concerns ABC) needed for the very last check, and strashing for miters
- emit certificates for the refactor and balance optimization passes within ABC
- design and implement the smart proof-of-equivalence strategy in ChECker.

Conclusion. We formalized the intuition of leveraging an optimizer's knowledge by having it produce certificates. We introduced a new and efficient CEC pipeline based on certificates. We presented ChECker, the first certificate-guided CEC engine, and achieved a preliminary 10x speedup compared to the state-of-the-art `&cec` engine in ABC. Despite its current limitations, our certificate-based approach is a promising step towards scalable equivalence checking.

Bibliography

[1] J. M. Williams, "Digital VLSI design with verilog," *Silicon Valley Technical Institute*, 2008.

[2] A. Mishchenko, "ABC: System for Sequential Logic Synthesis and Formal Verification." Accessed: Aug. 08, 2025. [Online]. Available: https://github.com/berkeley-abc/abc

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International*

Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22, 2010, pp. 24–40.

[4] Y. Headquarters, " Yosys Open SYnthesis Suite ." Accessed: Aug. 08, 2025. [Online]. Available: https://github.com/YosysHQ/yosys

[5] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[6] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 310–313.

[7] R. K. Brayton, "The decomposition and factorization of Boolean expressions," *ISCA-82*, pp. 49–54, 1982.

[8] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[9] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," 2005.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 532–535.

[11] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 38–44.

[12] F.-X. Reichl, F. Slivovsky, and S. Szeider, "eSLIM: Circuit minimization with SAT based local improvement," in *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, 2024, pp. 23–21.

[13] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[14] K. Tiri and I. Verbauwhede, "Place and route for secure standard cell design," in *Smart Card Research and Advanced Applications VI: IFIP 18th World Computer Congress TC8/WG8. 8 & TC11/WG11. 2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS) 22–27 August 2004 Toulouse, France*, 2004, pp. 143–158.

[15] W. H. Kao, C.-Y. Lo, M. Basel, and R. Singh, "Parasitic extraction: Current state of the art and future trends," *Proceedings of the IEEE*, vol. 89, no. 5, pp. 729–739, 2001.

[16] K.-h. Chang, I. L. Markov, and V. Bertacco, "Reap what you sow: Spare cells for post-silicon metal fix," in *Proceedings of the 2008 international symposium on Physical design*, 2008, pp. 103–110.

[17] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 12–17.

[18] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Proceedings of the 33rd Annual Design Automation Conference*, 1996, pp. 629–634.

[19] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli, "Formal verification of combinational circuits," in *Proceedings Tenth International Conference on VLSI Design*, 1997, pp. 218–225.

[20] 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, 2002.

[21] P. Khondkar, *Low-Power Design and Power-Aware Verification*. Springer, 2018.

[22] E. Goudet *et al.*, "Approximate Analytical Model Evaluating Digital Systems Compliance with Automotive Standards," in *2025 IEEE 26th Latin American Test Symposium (LATS)*, 2025, pp. 1–6.

[23] F. Solt and K. Razavi, "Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz," *USENIX Security*, 2025.

[24] R. Saravanan, S. Paria, A. Dasgupta, V. N. Patnala, S. Bhunia, and others, "SynFuzz: Leveraging Fuzzing of Netlist to Detect Synthesis Bugs," *arXiv preprint arXiv:2504.18812*, 2025.

[25] J. V. A. Vieira *et al.*, "Bottom-Up Generation of Verilog Designs for Testing EDA Tools," *arXiv preprint arXiv:2504.06295*, 2025.

[26] J. Feldtkeller, P. Sasdrich, and T. Güneysu, "Challenges and opportunities of security-aware EDA," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, pp. 1–34, 2023.

[27] J. Moondanos, C. H. Seger, Z. Hanna, and D. Kaiss, "CLEVER: Divide and conquer combinational logic equivalence verification with false negative elimination," in *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*, 2001, pp. 131–143.

[28] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001, pp. 114–121.

[29] I.-H. Moon and C. Pixley, "Non-miter-based combinational equivalence checking by comparing BDDs with different variable orders," in *International Conference on Formal Methods in Computer-Aided Design*, 2004, pp. 144–158.

[30] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 836–843.

[31] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis, "Parallel combinational equivalence checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3081–3092, 2019.

[32] Z. Chen, X. Zhang, Y. Qian, and S. Cai, "Datapath Combinational Equivalence Checking With Hybrid Sweeping Engines and Parallelization," *arXiv preprint arXiv:2501.14740*, 2024.

[33] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, 2004, pp. 42–49.

[34] A. Biere, "The AIGER and-inverter graph (AIG) format version 20071012," 2007.

[35] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," 2011.

[36] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[37] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024," in *Proc.~of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., in Department of Computer Science Report Series B. University of Helsinki, 2024, pp. 8–10.

[38] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, A. Gurfinkel and V. Ganesh, Eds., in Lecture Notes in Computer Science, vol. 14681. Springer, 2024, pp. 133–152. doi: 10.1007/978-3-031-65627-9_7.

[39] C. Jabs, "RustSAT: A Library For SAT Solving in Rust," in *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, J. Berg and J. Nordström, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 341. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2025, pp. 1–13. doi: 10.4230/LIPIcs.SAT.2025.24.

[40] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[41] R. Pelánek, "BEEM: Benchmarks for explicit model checkers," in *International SPIN Workshop on Model Checking of Software*, 2007, pp. 263–267.

[42] C. X. Wolf, " PicoRV32 - A Size-Optimized RISC-V CPU ." Accessed: Aug. 12, 2025. [Online]. Available: https://github.com/YosysHQ/picorv32

[43] SonalPinto, "Kronos is a 3-stage in-order RISC-V RV32I_Zicsr_Zifencei core geared towards FPGA implementations." Accessed: Aug. 12, 2025. [Online]. Available: https://github.com/SonalPinto/kronos

[44] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Trans-*

actions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 11, pp. 2629–2640, Nov. 2019, doi: 10.1109/TVLSI.2019.2926114.

[45] K. Asanovic *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[46] L. T-Head Semiconductor Co., " OpenXuantie - OpenC910 Core ." Accessed: Aug. 12, 2025. [Online]. Available: https://github.com/XUANTIE-RV/openc910

[47] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," May 2020.

[48] M. N. Mneimneh and K. A. Sakallah, "Principles of sequential-equivalence verification," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 248–257, 2005.

[49] A. Biere, "Bounded model checking," *Handbook of satisfiability*. IOS press, pp. 739–764, 2021.

[50] 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," in *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 747–750.

[51] H. Jin, M. Awedh, and F. Somenzi, "CirCUs: A satisfiability solver geared towards bounded model checking," in *International Conference on Computer Aided Verification*, 2004, pp. 519–522.

[52] H.-T. Zhang, J.-H. R. Jiang, and A. Mishchenko, "A circuit-based SAT solver for logic synthesis," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–6.

[53] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 106–119.

[54] G. C. Necula and P. Lee, "The design and implementation of a certifying compiler," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 333–344, 1998.

[55] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *International conference on computer aided verification*, 1996, pp. 419–422.

[56] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[57] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: a core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 243–257.

[58] A. Lööw, "The simulation semantics of synthesisable Verilog," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 1295–1320, 2025.

[59] A. Lööw, "Lutsig: A verified Verilog compiler for verified circuit development," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2021, pp. 46–60.

[60] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal Verification of High-Level Synthesis," New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3485494.

A Detailed Motivation Results for the EPFL Benchmark

Circuit name	SAT failure rate	Proportion of time spent on optimizations
int2float	0.0	0.21
cavlc	0.0	0.17
ctrl	0.0	0.21
i2c	0.25	0.37
dec	0	1.0
voter	0.04	0.02
priority	0.45	0.1
arbiter	0.0	0.75
mem_ctrl	0.26	0.14
router	0.89	0.1
multiplier	0.0	0.07
log2	0.01	0.01
square	0.0	0.1
div	0.07	0.25
sin	0.08	0.01
max	0.0	0.19
sqrt	0.0	0.38
adder	0.0	0.21
bar	0.0	0.26
hyp	0.01	0.06
average EPFL	0.1	0.23
average BEEM	0.38	0.1