2025 Design, Automation & Test in Europe Conference (DATE 2025)

# Efficient SAT-based Bounded Model Checking of Evolving Systems

Sophie Andrews
*Stanford University*
sophie1@cs.stanford.edu

Matthew Sotoudeh
*Stanford University*
sotoudeh@stanford.edu

Clark Barrett
*Stanford University*
barrettc@stanford.edu

***Abstract*—SAT-based verification is a common technique used by industry practitioners to find bugs in computer systems. However, these systems are rarely designed in a single step: instead, designers repeatedly make small modifications, reverifying after each change. With current tools, this reverification step takes as long as a full, from-scratch verification, even if the design has only been modified slightly. We propose a novel SAT-based verification technique that performs significantly better than the naïve approach in the setting of evolving systems. The key idea is to reuse information learned during the verification of earlier versions of the system to speed up the verification of later versions. We instantiate our technique in a bounded model checking tool for SystemVerilog code and apply it to a new benchmark set based on real edit history for a set of open source RISC-V cores. This new benchmark set is now publicly available for further research on verification of evolving systems. Our tool, PrediCore, significantly improves the time required to verify properties on later versions of the cores compared to the current state-of-the-art, verify-from-scratch approach.**

***Index Terms*—evolving systems, bounded model checking**

## I. Introduction

Hardware and software systems continue to grow larger, more complex, and more widely used in the modern world. Ensuring the correctness of such systems is essential for the safety and security of these applications. Formal verification is one powerful tool for addressing this challenge.

Formal techniques provide strong guarantees, but they are also computationally expensive. One understudied way to reduce the cost of formal verification is to reuse previous computational effort when solving a similar problem.

Large systems are rarely written and verified only once. Instead, systems *evolve*, adding new features, refactoring code, and fixing bugs over time. Existing formal workflows are not designed to leverage this ‘evolving systems’ reality. Verifying a slightly modified version of the system takes just as long as verifying the system from scratch, even if very little of the design has changed since the last verification run.

This paper introduces a methodology for reusing verification effort. Assuming we have two designs, where the second (today’s design) has evolved from the first (yesterday’s design) via a small set of changes, we propose to verify today’s design by leveraging what was learned from verifying yesterday’s design. We focus on verification queries encoded as propositional satisfiability (SAT) problems. After verifying yesterday’s design, we ask the SAT solver for an *unsat core*, a subset of the original problem sufficient for proving the property. The *anticore* is the complement of the unsat core, i.e., the set of clauses that are irrelevant for proving the property. Our key idea is to remove clauses that were part of yesterday’s anticore when verifying today’s design. However, due to changes in the design, the variable names in the formula representing today’s design may be very different from those in yesterday’s design. Hence, we also introduce an algorithm to recover a mapping between variables in the two formulas.

The authors would like to thank funding via NSF DGE-1656518 and Stanford IOG Research Hub 281101-1-UDCPQ 298911, in addition to helpful thoughts from Aina Niemetz, Mathias Preiner, and Dawson Engler.

We apply our method to SAT-based bounded model checking (BMC) problems. To test our approach, we created a new benchmark set based on open source RISC-V cores [1]. Our benchmarks are based on the real evolution of these cores using their edit history. We show that our technique significantly accelerates design reverification on these benchmarks.

To summarize, the paper makes the following contributions:

- a new technique for finding shared structure between Boolean formulas;
- a new BMC algorithm designed for evolving systems;
- a new benchmark set based on the evolution of real hardware designs; and
- an evaluation of our technique on these benchmarks, demonstrating its effectiveness.

The rest of this paper contains relevant background information (Section II), a motivating example (Section III), our algorithm for faster reverification (Section IV), an instantiation of that algorithm in a bounded model checking tool (Section V), evaluation of our tool (Section VII), related work (Section VIII), and a conclusion (Section IX).

## II. Background

**SAT Solvers.** Many important problems can be reduced to determining whether there exists an assignment to each variable in a Boolean formula (either true or false) under which the entire formula evaluates to true. This is known as the Boolean satisfiability, or SAT, problem, and solvers for this problem are called *SAT solvers*. Since the turn of the century, there has been steady progress on improving their efficiency, and modern SAT solvers can solve large problems with millions of clauses and variables [2]. SAT solvers take as input a formula in conjunctive normal form (CNF), i.e., a conjunction of disjunctions (clauses) of variables or their negations, and either return a satisfying assignment, report that none exists, or do not terminate if the problem is too difficult. If no solution exists, the problem is *unsatisfiable*, and many solvers can produce an *unsat core*, a subset of clauses in the CNF formula that are together unsatisfiable. The unsat core captures which constraints are essential to the proof of unsatisfiability found by the solver.

**Bounded Model Checking.** One major application for SAT solvers is *bounded model checking* (BMC) [3]. Given a hardware or software system and a property, a BMC tool creates a Boolean formula representing whether the property holds after running the system on a *symbolic* input for a specified number of steps $k$, known as the *unrolling bound*. Each bit of state in the original system is represented by a set of Boolean variables, one for each time step up to $k$. Satisfying assignments correspond to inputs that violate the property (bugs). If a SAT solver reports the Boolean formula is unsatisfiable, this means that the system cannot violate the property in $k$ or fewer steps. Because a SAT solver is used, we can extract unsat cores from BMC instances.

**And-Inverter Graphs.** And-inverter graphs (AIGs) [4]–[6] are frequently used to represent Boolean formulas derived from hardware or software systems. AIGs are useful because they have more structure than formulas in CNF, while being lowerable into CNF without introducing new variables. Formally, an AIG is a rooted, directed, and acyclic graph whose leaves represent variables and whose interior nodes (called *gates*) represent the conjunction of their children. Each gate has exactly two children. Any edge in the AIG can be *inverted*, meaning that the parent node operates on the logical negation of the child node. Section V-A will use the AIG to identify structure shared between similar verification problems.

## III. Motivating Example

This section describes our technique at a high level.

### A. Solving the Initial Problem

Imagine a complex circuit, buried within which are two signals, $a_1$ and $a_2$, each based on a different encoding of a ternary logical 'and' operation. A Boolean formula checking that $a_1$ and $a_2$ must be equivalent might look like this:[1]

$$\begin{aligned}&(a_1 \neq a_2) \wedge (a_1 = (x \wedge b_1)) \wedge (b_1 = (y \wedge z))\\ &\wedge (a_2 = (b_2 \wedge z)) \wedge (b_2 = (x \wedge y))\\ &\wedge \phi_1(x,y,z) \wedge \phi_2(x,y,z) \wedge \phi_3(x,y,z) \wedge \cdots\end{aligned}$$

The extra $\phi$ formulas are constraints produced during the translation of the circuit that are not semantically relevant to the property being verified: they might represent control or data circuits that are not related to the property.

Importantly, although the clauses $\phi_1, \phi_2, \ldots$ are not used when proving the formula unsatisfiable, neither the tool that encodes the circuit into a formula nor the SAT solver that receives the formula can typically determine which clauses are irrelevant up front. It is only after a complete proof of unsatisfiability has been found that this becomes clear.

Suppose a SAT solver reports that this formula is unsatisfiable, i.e., the signals are indeed equivalent as expected. We can then ask the solver for an unsat core; in this example, it might return the first five constraints:

$$\begin{aligned}&(a_1 \neq a_2) \wedge (a_1 = (x \wedge b_1)) \wedge (b_1 = (y \wedge z))\\ &\wedge (a_2 = (b_2 \wedge z)) \wedge (b_2 = (x \wedge y)),\end{aligned}$$

as they alone are already unsatisfiable.

[1]For brevity, and without affecting the validity of the approach, we use $=$ and $\neq$ instead of representing everything in CNF.

### B. Predicting an Unsat Core

Now, suppose the developer makes a small change to some irrelevant part of the design, e.g., they add an extra register to the register file. They may then want to reverify the equivalence of the two ternary 'and' operations to check that the modification has not introduced a bug. Constructing a new SAT instance from the modified circuit might produce a Boolean formula of the following form:

$$\begin{aligned}&(a_1 \neq a_2) \wedge (a_1 = (x \wedge b_1)) \wedge (b_1 = (y \wedge z))\\ &\wedge (a_2 = (b_2 \wedge z)) \wedge (b_2 = (x \wedge y))\\ &\wedge \phi_1'(x,y,z) \wedge \phi_2(x,y,z) \wedge \phi_3(x,y,z) \wedge \cdots\end{aligned}$$

Here, $\phi_1'$ differs from the original $\phi_1$ because details of the register file implementation have been modified.

To reverify the property, the standard approach would be simply to call a SAT solver on the new formula. The goal of this paper is to speed up this second check of the *modified* circuit by leveraging information from the first check made on the *original* version of the circuit.

More specifically, we use the unsat core from the original formula to predict a subset of clauses in the modified formula that are sufficient to prove unsatisfiability (the *predicted core*) and then run the SAT solver on *only that subset* of clauses. If the solver reports that the predicted core is unsatisfiable, then clearly, the full instance must be as well. If the prediction fails (the predicted core is satisfiable), we fall back to running the solver on the full instance.

To form the predicted core, we first remove the clauses in the unsat core from the original formula to form the *anticore*, i.e., a set of clauses that were not needed to prove unsatisfiability. In our example, the anticore consists of $\phi_1$, $\phi_2$, and $\phi_3$. We then remove from the modified formula any clauses that are identical to those in the anticore of the original formula; in the example, we would remove $\phi_2$ and $\phi_3$. Note that we cannot remove $\phi_1'$ because it is not identical to $\phi_1$. Thus, for our example, the predicted core would be:

$$\begin{aligned}&(a_1 \neq a_2) \wedge (a_1 = (x \wedge b_1)) \wedge (b_1 = (y \wedge z))\\ &\wedge (a_2 = (b_2 \wedge z)) \wedge (b_2 = (x \wedge y))\\ &\wedge \phi_1'(x,y,z).\end{aligned}$$

Note that in this simple example, the entire unsat core is still present; hence, it would be more efficient to only use those clauses, i.e., to exclude $\phi_1'$ as well. However, in real designs, we have found that it is much less likely for the unsat core to remain unchanged. This is why we remove clauses predicted to be unnecessary instead of keeping clauses that are necessary; it is safer to keep a clause if we are unsure about whether it will be useful or not.

## C. Mapping Between Formulas

In reality, the situation is a bit more complicated. Tools that translate verification problems into SAT instances often perturb the assignment of variable names in unpredictable ways. Hence, the modified Boolean formula might look more like:

$$\begin{aligned}&(a_6 = (m \wedge b_1)) \wedge (b_1 = (n \wedge q))\\ &\wedge (a_3 = (b_4 \wedge q)) \wedge (b_4 = (m \wedge n)) \wedge (a_6 \neq a_3)\\ &\wedge \phi_1'(m,n,q) \wedge \phi_2(m,n,q) \wedge \phi_3(m,n,q) \wedge \cdots .\end{aligned}$$

Notice that this formula has a syntactically similar structure but uses different variable names. Hence, we first rename variables to maximize syntactic similarity with the original formula, then construct the predicted core as before.

## IV. Approach

Our general approach for SAT-based checking of evolving systems has two stages: the *initial stage* (Algorithm 1) and the *post-modification stage* (Algorithm 2). For simplicity, we only focus on a fixed unrolling bound for the problem instance.

**Algorithm 1:** Initial stage

```
Input: Problem instance prob; CNF encoder ENCODE;
       SAT solver SOLVE.
1 $\texttt{cnf} \leftarrow \textsc{Encode}(\texttt{prob})$;
2 $\texttt{res}, \texttt{core} \leftarrow \textsc{Solve}(\texttt{cnf})$;
3 if res is sat then Return ;
4 $\texttt{anticore} \leftarrow \texttt{cnf} \setminus \texttt{core}$;
5 Save prob, cnf, anticore for later use;
```

### A. Initial Stage

In the initial run (Algorithm 1), we solve the problem and, if the result is unsatisfiable, store some information to help speed up subsequent runs. Specifically, in addition to the problem and its CNF encoding, we store the *anticore*, the set of clauses not in the unsat core. Intuitively, these clauses are extraneous: they are not needed to prove unsatisfiability.

### B. Post-Modification Stage

Now, suppose we modify the problem and redo the verification. Algorithm 2 uses the anticore stored during the initial stage to speed up solving the modified problem. We predict that the core of the modified problem $\texttt{prob}'$ consists of all clauses that do *not* appear in the initial stage's anticore. If the prediction fails, we fall back to solving the full formula.

Importantly, before constructing the predicted core, Algorithm 2 renames the variables in the anticore to increase the number of syntactically identical clauses in the two formulas (i.e., in $\texttt{cnf}$ and $\texttt{cnf}'$). Two clauses are considered syntactically identical if they contain exactly the same variables (even if they are in a different order). This step makes use of the MapCNFVars function, which takes as input the two problems and produces a mapping between the encoded CNF variables. We explain an efficient mapping algorithm for an instantiation of this technique for BMC of hardware designs in Section V-A. Soundness is straightforward.

**Algorithm 2:** Post-modification stage

```
Input: Problem instance prob'; CNF encoder
       ENCODE; SAT solver SOLVE. prob, cnf, and
       anticore from a previous run of Algorithm 1.
1 $\texttt{cnf}' \leftarrow \textsc{Encode}(\texttt{prob}')$;
2 $\texttt{map} \leftarrow \textsc{MapCNFVars}(\texttt{prob}, \texttt{prob}', \texttt{cnf}, \texttt{cnf}')$;
3 $\texttt{anticore}'_{pred} \leftarrow \textsc{RenameVars}(\texttt{anticore}, \texttt{map})$;
4 $\texttt{core}'_{pred} \leftarrow \texttt{cnf}' \setminus \texttt{anticore}'_{pred}$ ;
5 if $\textsc{Solve}(\texttt{core}'_{pred})$ is unsat then
6     Return Verified/unsat;
7 Return $\textsc{Solve}(\texttt{cnf}')$;
```

**Theorem 1.** *Regardless of the mapping found by* MapCNFVars, *if Algorithm 2 returns 'unsat', then* $\texttt{cnf}'$ *(i.e.,* $\textsc{Encode}(\texttt{prob}')$*) is indeed unsatisfiable.*

*Proof.* Suppose the algorithm reaches line 6, i.e., $\texttt{core}'_{pred}$ is unsatisfiable. Line 4 defines $\texttt{core}'_{pred}$ to be a subset of the clauses in $\texttt{cnf}'$; hence, if $\texttt{core}'_{pred}$ has no solution, we know $\texttt{cnf}'$ has no solution as well. □

## V. Instantiation in a SystemVerilog BMC Tool

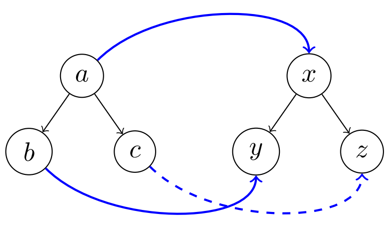
This section shows how to instantiate the algorithms in Section IV in a working BMC pipeline for SystemVerilog code. We implement Encode using Yosys [7] and Solve using CaDiCaL [8]. MapCNFVars is explained in Section V-A below. Section V-B then describes a number of optimizations that further boost performance. Our implementation is provided as a publicly available tool, PrediCore [9].

### A. Implementation of MapCNFVars

MapCNFVars is given two problem instances and must find a mapping from the variables in the initial problem to those in the modified problem that maximizes the number of syntactically identical clauses in the two formulas. BMC encoders produce a large number of variables, so a brute-force approach does not scale. Instead, we use a two-phase greedy approach, in which a partial mapping is gradually extended according to a small number of rules. The two phases are: (i) selecting an initial set of mapped variables to seed the mapping; and (ii) incrementally extending an existing (partial) mapping.

To construct the initial variable mapping, we take advantage of the symbol table produced by Yosys during encoding, which matches variables in the high-level SystemVerilog code with variables in the low-level Boolean formula representation. (Our tool uses Yosys, but any encoder that produces a similar symbol table would suffice.) If the symbol table assigns the same symbol to both CNF variable $a$ from the initial problem and $b$ from the modified problem, then we include a mapping from $a$ to $b$ in our initial mapping. Unfortunately, only a few variables (empirically, about 10%) can be mapped in this way.

Thus, we need a second approach to expand the mapping. We use two rules which are applied repeatedly until a fixed point is reached. These rules operate on an AIG representation provided by Yosys as an intermediate output in AIGER format [4].



(a) Rule 1 (b) Rule 2

Fig. 1: Illustrations of the two variable mapping rules. Variables in the initial AIG are labeled $a$, $b$, $c$, while variables in the post-modification AIG are labeled $x$, $y$, $z$. A blue edge $a \rightarrow x$, indicates a mapping from variable $a$ to $x$. Mappings already present before executing the rule are shown with solid lines, while those added by the rule are shown as dashed lines.

The first rule (Figure 1a) finds gates in the initial AIG where two out of the three variables making up the gate already have mappings to corresponding variables in a gate in the post-modification AIG. In this case, we extend the mapping to also include the last remaining variable.

The second rule (Figure 1b) finds mapped variables with a unique parent in the initial AIG graph. If the corresponding variable in the post-modification AIG also has a unique parent, those parents are mapped to each other.

We apply rule 1 wherever possible, then rule 2 whenever possible, repeating these two steps until a fixed point is reached. In our evaluation we usually observed close to 90% of the clauses in the final CNF were identical after applying these rules.

### B. Other Optimizations

We implemented several additional optimizations to speed up the general framework described above. After the initial stage, we store the anticore extracted from the initial problem in a binary hashmap to avoid deserializing during the post-modification stage. Also, we do the mapping on the original circuit to minimize the number of variables that the mapping algorithm must consider, then duplicate for every unroll level.

## VI. Benchmark Set

The lack of a realistic benchmark set has been an obstacle for evaluating techniques for model checking of evolving systems. Typically, hardware designs (not to mention their edit histories) are proprietary, and companies are extremely selective about who can access them. The Hardware Model Checking Competition (HWMCC) [10] contains an open library of problems, but unfortunately, no edit history is available for them.[2]

This paper introduces what is, to our knowledge, the first large-scale set of benchmarks based on actual edit histories of real-world hardware designs. Our benchmarks are extracted from the RISC-V Formal project [1], which contains formal verification wrappers for several different open-source CPU cores. Each CPU has a set of formal checks, which ensure that the design properly implements the RISC-V instruction set architecture and satisfies other design-specific guarantees.

Out of five CPUs supported by the RISC-V Formal project, we extracted benchmarks based on the three (`nerv`, `picorv`, and `serv`) that are represented directly in SystemVerilog (the other two [12], [13] require a Scala preprocessor). The `nerv`, `picorv`, and `serv` CPUs have, respectively, 151, 87, and 43 associated formal properties, and 55, 148, and 229 relevant commits in their git histories. `nerv` [14] is a naïve educational CPU, `picorv` [15] is a size-optimized CPU, and `serv` is described as "the world's smallest RISC-V CPU" [16]. While we provide the `serv` benchmark results in our artifact, the checks for this benchmark are all trivial (under 10s), so we do not discuss it further in this paper.

For each CPU–commit–formal check pair, we used Yosys 0.38 to extract a bounded model checking instance with 50 unrolls and CaDiCaL 1.9.4 to test the satisfiability of the instance. The results are shown in Figure 2. Each column corresponds to a single commit point in the CPU core's source repository. The different fills indicate how many of the formal verification checks pass (unsatisfiable), fail (satisfiable), time out, or are ill-formed. Some are ill-formed because the core is developed independently of the formal checks, so there are commit points where the formal checks reference signals that do not exist or have the wrong type (especially for commits made before the formal checks were introduced). Both cores show a clear evolution in their edit histories, with more formal checks passing over time as functionality is added and bugs are fixed. [3]

Each benchmark is a formal property paired with two consecutive commit points, but only if the property passes (the formula is unsatisfiable) at both commit points. We also ignore pairs of commit points for which the AIG files are identical (as simple caching can solve the reverification effort problem in such cases). This leaves 1248 and 1029 pairs for `nerv` and `picorv`, respectively. Our benchmark set is publicly available [9].

## VII. Evaluation

This section reports on an evaluation of our approach on the benchmark set described in Section VI. We ran our evaluation on a large compute cluster, with each benchmark instance running on an AMD 7502 CPU with 32 GB of memory.

Table I shows the amount of time spent in each phase of the tool, separated by whether the core is correctly predicted or not. 'Ours Total' is the average time required in the post-modification phase, comparable with the 'Baseline Time' of directly solving the problem using CaDiCaL. When the core is correctly predicted (`n-correct`, `p-correct`), the average time is significantly reduced (2.3× for `nerv` and 3.6× for `picorv`). When the predicted core is satisfiable, our technique

[2]Analogous work on *unbounded* model checking [11] has resorted to using random corruptions of HWMCC benchmarks, which runs the risk of producing unrealistic hardware design edits and potentially misleading results.

[3]For `nerv`, a few checks fail even on its latest commit. This is due to a mismatch in port naming between the latest version on the `nerv` repository and the expected names used by the RISC-V formal repository's checks. This does not occur on the main RISC-V formal repository's checks of `nerv` because they use a fork of `nerv` that is maintained separately and does not have this naming mismatch issue.

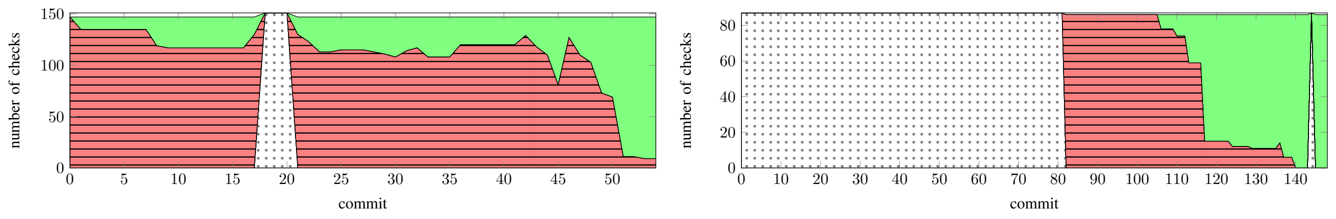


Fig. 2: Summary of passing formal checks over time for the `nerv` (left) and `picorv` (right) CPU designs. Dotted (white) corresponds to ill-formed; striped (red) corresponds to not passing (satisfiable); solid (green) corresponds to passing (unsatisfiable).

TABLE I: Experimental results. Times are averages in seconds. Rows with 'correct' and 'incorrect' include only instances whose predicted core is unsatisfiable and satisfiable, respectively. Initial 'Solve' is the time to solve the initial CNF. 'Anticore' is the time to store the anticore. 'Map' is the time to compute the mapping. 'Pred' is the time to predict the unsat core. Post-Modification 'Solve' is the time to solve the predicted core. 'Re-Solve' is the time to solve the full post-modification problem, incurred if the predicted core is satisfiable. 'Ours Total' is the total time needed for our technique to check the post-modification core. 'Baseline Time' is the time for CaDiCaL to solve the full post-modification problem (without core generation).

| | | Initial | | Post-Modification | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Core | Which | Count | Solve | Anticore | Map | Pred | Solve | Re-Solve | Ours Total | Baseline Time | Ours Speedup |
| nerv | n-all | 1248 | 29.1 | 4.6 | 1.9 | 3.7 | 7.0 | 11.1 | 23.8 | 28.5 | 1.2× |
| | n-correct | 779 | 30.4 | 4.4 | 1.8 | 3.6 | 6.6 | 0.0 | 12.1 | 28.1 | 2.3× |
| | n-incorrect | 469 | 26.8 | 4.8 | 2.0 | 3.9 | 7.5 | 29.6 | 43.3 | 29.1 | 0.7× |
| picorv | p-all | 1029 | 125.9 | 3.5 | 2.2 | 2.4 | 16.7 | 36.7 | 58.2 | 90.7 | 1.6× |
| | p-correct | 734 | 121.6 | 3.4 | 2.1 | 2.5 | 20.0 | 0.0 | 24.8 | 88.6 | 3.6× |
| | p-incorrect | 295 | 136.7 | 3.6 | 2.2 | 2.4 | 8.2 | 127.9 | 141.3 | 95.9 | 0.7× |

must re-solve the entire problem on top of the work to map, predict the core, and solve the predicted core. Hence, the `n-incorrect` and `p-incorrect` instances always take longer than the baseline.[4] Importantly, in the rows that take both into account (`n-total` and `p-total`), there is still an overall speedup (1.2× for `nerv` and 1.6× for `picorv`), indicating that despite the incorrect predictions, the approach still leads to improvement overall.

Note that these speedups do not include the time taken to process the anticore, because this can be done in an overnight (i.e., "offline") period, where we only require information about the initial problem instance we already have, not the new instance we are trying to do a quick verification of. Thus, processing the anticore does not contribute to verification time *latency* (minimizing latency is our primary goal). Importantly, even when including that time (1.0× for nerv and 1.5× for picorv), we see the improved verification latency does not come at the expense of additional end-to-end compute.

Figure 3 visualizes the results as a scatter plot. Each point corresponds to a single property and commit. The horizontal axis is the baseline time. The vertical axis is the time it takes using our approach. Points below the $y = x$ line represent speedup; points above represent slowdown. The two main clusters of points on these scatter plots correspond to whether or not the prediction is correct. The cluster with a large speedup corresponds to instances where the predicted core was unsatisfiable. The rest of the points have a slight slowdown

[4]Note that if additional compute resources are available, the cost of an incorrect prediction can be eliminated by running our approach in parallel with the baseline. We report the more conservative sequential performance here.

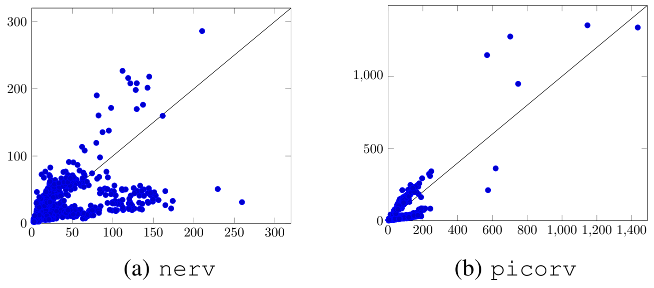


Fig. 3: Scatterplot: each point compares the time to verify a single property at a single commit, either using our approach (vertical axis, seconds) or not (horizontal axis, seconds).

(right above the $y = x$ line). These are instances where the core was incorrectly predicted, thus requiring re-solving the full problem. The slowdown comes from the overhead of computing and solving the predicted core.

It is worth pointing out that our evaluation suggests that the benefit of our technique *grows* as the problems become more difficult. We can see this in the scatter plot—the distance between the clusters grows as the problems get more difficult—and in the table, where the harder `picorv` benchmarks are sped up on the whole much more from our approach.

These results suggest that overall, our approach is effective at reducing the latency of post-modification verification. They also suggest that future work on more accurately predicting the unsat core could yield significant further performance improvements, even if it requires slightly longer to compute the prediction.

## VIII. Related Work

Green [17] caches unsatisfiable subformulas to immediately determine that later queries are unsatisfiable if the later constraints are a superset of any of those known-unsatisfiable subformulas. They also have a canonicalization pass that serves a similar purpose to our mapping stage. However, their canonicalization pass seems specific to high-level linear arithmetic SMT formulas, whereas our hardware model checking setting involves low-level Boolean problems. They use a syntactic slicing operation on the SMT formula rather than our proposal to use the unsat core produced by the SAT solver. We ran some preliminary experiments with their slicing technique and found that our CNF files have already been sliced by the Yosys encoder. Furthermore, our approach of removing clauses from the anticore allows us to speed up the subsequent solving time even if the new formula does not contain the entirety of the previous unsatisfiable core.

Many techniques focus on calling the solver fewer times, whereas we try to call the solver on an easier version of the problem, so our work is, largely, complementary to these approaches. Sorbot [18] operates on high-level relational logic specifications (vs. our low-level AIG and CNF), assumes variable names do not change, and requires the user of the tool to provide, ahead of time, a list of ways that the system might change in the future. FLAIR [19] speeds up BMC of evolving systems in a specific domain (Android application interactions) and does not seem immediately generalizable. Platinum [20] speeds up model checking problems written in the Alloy language [21] by splitting each problem into independent subproblems that do not share variables; this approach does not directly apply to our setting, as our CNF formulas usually are not decomposable into independent subproblems. iAlloy [22] relies on complicated a source-level analysis.

eVolCheck [23] proposes an efficient BMC technique for evolving software. It relies on source-level information to break up the program into functions and compute summaries of each function; functions that are unchanged (along with their dependencies) need not update their summaries. It is not immediately clear what the analog for function summaries would be in the context of large hardware circuits. Also, our approach requires significantly less insight into the front-end/source level.

Our approach is complementary to the well-studied area of *incremental* bounded model checking. In incremental BMC, the system being checked stays unmodified but the unroll bound is gradually increased [24]–[27]. In system evolution (our setting), the system being checked is modified between checks.

Beyer et al. [28] propose a technique to speed up *unbounded* model checking of device drivers in the program evolution setting. They primarily reuse information about abstraction precision, which is not relevant in our bounded model checking scenario. They introduce a benchmark set for software evolution, while we focus on hardware evolution.

Chockler et al. [11] and FuseIC3 [29] speed up *unbounded* model checking of hardware by attempting to reuse IC3 invariants learned on earlier versions of the system. The bounded model checking setting does not have explicit invariants in the IC3 sense, so it is not clear that their approach can be directly adapted. The only realistic, non-proprietary benchmarks that they evaluate on involve the same hardware designs with different parameters, rather than our benchmark set which features a sequence of real modifications to a single design. Both tools aim to reduce the number of SAT queries, while we aim to reduce the solve time of a single query.

The fact that solvers can be slowed down by the introduction of unnecessary clauses is well-known [30]. NeuroCore [31] also works by predicting the unsat core. Because they are focused on general-purpose SAT solving, rather than the system evolution setting we consider here, their technique requires both modifications to state-of-the-art solvers and 20 GPUs for training. In the system evolution setting, our method is much simpler and treats solvers as a black box.

Unsatisfiable cores have also been shown to be useful in other problems, e.g., speeding up MaxSAT solving on a sequence of related problems [32]–[36]. Our approach for renaming AIG nodes is related to symmetry-finding and canonicalization techniques [37], [38]. Our approach prioritizes speed, while those works can provide higher precision.

Große and Drechsler [39] consider the complementary setting where properties change over time rather than the design.

## IX. Conclusion

We proposed a way to reuse information learned while verifying an earlier version of a system to speed up verification of later, modified versions. We instantiated our technique in a tool for bounded model checking of hardware designs written in SystemVerilog and evaluated it on a new benchmark set based on real edit histories of multiple open source RISC-V CPUs. Our technique achieves significant speedups, especially for the hardest-to-verify instances in our benchmark set.

A natural direction for future work is to develop additional heuristics for predicting what clauses will be in the core, thereby hopefully reducing how often we must fall back to solving the full CNF formula. In addition, we could try to use information obtained while solving the predicted core to speed up solving the full CNF (we have experimented, e.g., with adding clauses learned while solving the predicted core, but this naïve idea does not seem to help).

Our benchmark set could be used to evaluate a variety of different tools that operate on hardware designs during their evolution, such as existing work on *unbounded* model checking of hardware. Typical work in this area evaluates on proprietary designs (unreproducible outside that particular company) or random corruptions of HWMCC benchmarks [11], [29]; our benchmarks provide a more realistic sequence of edits.

We would also like to extend our technique to support word-level model checking tools targeting satisfiability modulo theories (SMT) [40] rather than CNF. Our general approach of predicting the unsatisfiable core could still be applied to SMT-based bounded model checking, but a specific mapping algorithm would need to be designed for SMT formulas.

Finally, we could apply our approach to other settings such as BMC of software, which would require a mapping algorithm designed for the structure of software programs.

## References

[1] C. Wolf, "Riscv-formal," https://github.com/YosysHQ/riscv-formal, 2024.

[2] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 133–182. [Online]. Available: https://doi.org/10.3233/FAIA200987

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of TACAS*, 1999, pp. 193–207.

[4] A. Biere, K. Heljanko, and S. Wieringa, "Aiger 1.9 and beyond," 2011. [Online]. Available: https://fmv.jku.at/papers/BiereHeljankoWieringa-FMV-TR-11-2.pdf

[5] A. Kuehlmann, V. Paruthi, F. Krohm, and M. 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.

[6] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based boolean reasoning," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 232–237. [Online]. Available: https://doi.org/10.1145/378239.378470

[7] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[8] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[9] S. Andrews, M. Sotoudeh, and C. Barrett, "Efficient sat-based bounded model checking of evolving systems," DATE, 2025. [Online]. Available: https://github.com/sophiejandrews/PrediCore

[10] "Hwmcc'20," 2020. [Online]. Available: https://fmv.jku.at/hwmcc20/

[11] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," *2011 Formal Methods in Computer-Aided Design, FMCAD 2011*, 01 2011.

[12] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[13] S. H. contributors, "Vexriscv," https://github.com/SpinalHDL/VexRiscv.

[14] C. Wolf and N. Engelhardt, "Nerv," https://github.com/YosysHQ/nerv, 2023.

[15] C. Wolf, "Picorv32," https://github.com/YosysHQ/picorv32, 2024.

[16] O. Kindgren, "Serv," https://github.com/olofk/serv, 2024.

[17] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, W. Tracz, M. P. Robillard, and T. Bultan, Eds. ACM, 2012, p. 58. [Online]. Available: https://doi.org/10.1145/2393596.2393665

[18] C. Stevens and H. Bagheri, "Combining solution reuse and bound tightening for efficient analysis of evolving systems," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 89–100. [Online]. Available: https://doi.org/10.1145/3533767.3534399

[19] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 357–368. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00044

[20] G. Zheng, H. Bagheri, G. Rothermel, and J. Wang, "Platinum: Reusing constraint solutions in bounded analysis of relational logic," in *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, ser. Lecture Notes in Computer Science, H. Wehrheim and J. Cabot, Eds., vol. 12076. Springer, 2020, pp. 29–52. [Online]. Available: https://doi.org/10.1007/978-3-030-45234-6_2

[21] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[22] W. Wang, K. Wang, M. Gligoric, and S. Khurshid, "Incremental analysis of evolving alloy models," in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, T. Vojnar and L. Zhang, Eds., vol. 11427. Springer, 2019, pp. 174–191. [Online]. Available: https://doi.org/10.1007/978-3-030-17462-0_10

[23] O. Sery, G. Fedyukovich, and N. Sharygina, "Incremental upgrade checking," in *Validation of Evolving Software*, H. Chockler, D. Kroening, L. Mariani, and N. Sharygina, Eds. Springer, 2015, pp. 55–72. [Online]. Available: https://doi.org/10.1007/978-3-319-10623-6_6

[24] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental bounded model checking for embedded software (extended version)," *CoRR*, vol. abs/1409.5872, 2014. [Online]. Available: http://arxiv.org/abs/1409.5872

[25] J. Kim, J. Whittemore, and K. Sakallah, "On solving stack-based incremental satisfiability problems," in *Proceedings 2000 International Conference on Computer Design*, 2000, pp. 379–382.

[26] J. Whittemore, J. Kim, and K. Sakallah, "Satire: a new incremental satisfiability engine," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 542–545. [Online]. Available: https://doi.org/10.1145/378239.379019

[27] S. Wieringa, "On incremental satisfiability and bounded model checking," in *First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS11)*, 2011, vK: dc.

[28] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision reuse for efficient regression verification," 08 2013.

[29] R. Dureja and K. Y. Rozier, "Fuseic3: An algorithm for checking large design spaces," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 164–171.

[30] M. Vinyals, "Hard examples for common variable decision heuristics," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 02, pp. 1652–1659, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5527

[31] D. Selsam and N. Bjørner, "Guiding high-performance sat solvers with unsat-core predictions," in *Theory and Applications of Satisfiability Testing – SAT 2019*, M. Janota and I. Lynce, Eds. Cham: Springer International Publishing, 2019, pp. 336–353.

[32] X. Si, X. Zhang, V. Manquinho, M. Janota, A. Ignatiev, and M. Naik, "On incremental core-guided maxsat solving," in *Principles and Practice of Constraint Programming*, M. Rueher, Ed. Cham: Springer International Publishing, 2016, pp. 473–482.

[33] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[34] J. Marques-Silva and J. Planes, "On using unsatisfiability for solving maximum satisfiability," *CoRR*, vol. abs/0712.1097, 2007. [Online]. Available: http://arxiv.org/abs/0712.1097

[35] M. H. Liffiton and K. A. Sakallah, "Generalizing core-guided max-sat," in *Theory and Applications of Satisfiability Testing - SAT 2009*, O. Kullmann, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[36] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, 2008, pp. 77–82.

[37] H. Katebi, K. A. Sakallah, and I. L. Markov, "Graph symmetry detection and canonical labeling: Differences and synergies," *CoRR*, vol. abs/1208.6271, 2012. [Online]. Available: http://arxiv.org/abs/1208.6271

[38] T. Junttila and P. Kaski, *Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs*, pp. 135–149. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972870.13

[39] D. Große and R. Drechsler, "Acceleration of sat-based iterative property checking," in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, D. Borrione and W. J. Paul, Eds., vol. 3725. Springer, 2005, pp. 349–353. [Online]. Available: https://doi.org/10.1007/11560548_29

[40] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability, Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2021.