SUPPLEMENT to Certified Knowledge Compilation with Application to Verified Model Counting

Randal E. Bryant
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA

Wojciech Nawrocki
Department of Philosophy, Carnegie Mellon University

Jeremy Avigad
Department of Philosophy, Carnegie Mellon University

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University

This document is a supplement to the paper "Certified Knowledge Compilation with Applications to Verified Model Counting," published at the 2023 Conference on Theory and Applications of SAT Testing (SAT).

1 CPOG Example

Figure 1 illustrates an example formula and shows how the CPOG file declares its POG representation. The input formula (A) consists of five clauses over variables x_1, x_2, x_3, and x_4. The generated POG (B) has six nonterminal nodes representing four products and two sums. We name these by the node type (product $\mathbf{p}$ or sum $\mathbf{s}$), subscripted by the ID of the extension variable. The first part of the CPOG file (C) declares these nodes using clause IDs that increment by three or four, depending on whether the node has two children or three. The last two nonzero values in the sum declarations are the hint providing the required mutual exclusion proof.

1.1 Node Declarations

We step through portions of the file to provide a better understanding of the CPOG proof framework. Figure 1(D) shows the defining clauses that are implicitly defined by the POG operation declarations. These do not appear in the CPOG file. Referring back to the declarations of the sum nodes in Figure 1(C), we can see that the declaration of node $\mathbf{s}_7$ had clause IDs 7 and 10 as the hint. We can see in Figure 1(A) that these two clauses form a RUP proof for the clause $\overline{p}_5 \vee \overline{p}_6$, showing that the two children of $\mathbf{s}_7$ have disjoint models. Similarly, node $\mathbf{s}_{10}$ is declared as having clause IDs 16 and 19 as the hint. These form a RUP proof for the clause $\overline{p}_8 \vee \overline{p}_9$, showing that the two children of $\mathbf{s}_{10}$ have disjoint models.

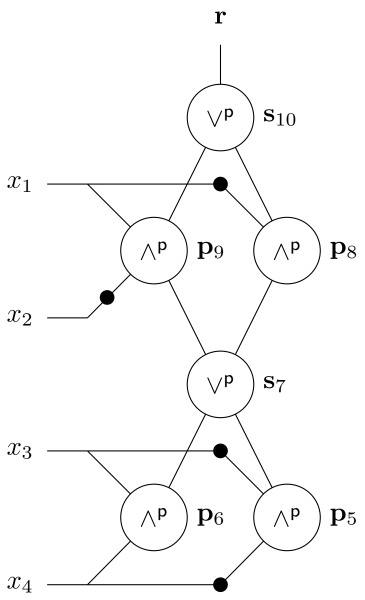
1.2 Forward Implication Proof

Figure 1(E) provides the sequence of assertions leading to unit clause 36, consisting of the literal s_{10}. This clause indicates that $\mathbf{s}_{10}$ is implied by the input clauses, i.e., any total assignment α satisfying the input clauses must have $\alpha(s_{10}) = 1$. Working backward, we can see that clause 29 indicates that variable p_8 will be implied by the input clauses when $\alpha(x_1) = 0$, while clause 34 indicates that node p_9 will be implied by the input clauses when $\alpha(x_1) = 1$. These serve as the hint for clause 36.

(A) Input Formula

ID	Clauses	
1	`-1 3 -4`	`0`
2	`-1 -3 4`	`0`
3	`3 -4`	`0`
4	`1 -3 4`	`0`
5	`-1 -2`	`0`

(B) POG Representation



(C) POG Declaration

ID	CPOG line		Explanation
6	`p 5 -3 -4`	`0`	$p_5 = \overline{x}_3 \wedge^{\mathsf{p}} \overline{x}_4$
9	`p 6 3 4`	`0`	$p_6 = x_3 \wedge^{\mathsf{p}} x_4$
12	`s 7 5 6 7 10`	`0`	$s_7 = p_5 \vee^{\mathsf{p}} p_6$
15	`p 8 -1 7`	`0`	$p_8 = \overline{x}_1 \wedge^{\mathsf{p}} s_7$
18	`p 9 1 -2 7`	`0`	$p_9 = x_1 \wedge^{\mathsf{p}} \overline{x}_2 \wedge^{\mathsf{p}} s_7$
22	`s 10 8 9 16 19`	`0`	$s_{10} = p_8 \vee^{\mathsf{p}} p_9$
	`r 10`		Root $r = s_{10}$

(D) Defining Clauses

ID	Clauses		Explanation
6	`5 3 4`	`0`	Define p_5
7	`-5 -3`	`0`	
8	`-5 -4`	`0`	
9	`6 -3 -4`	`0`	Define p_6
10	`-6 3`	`0`	
11	`-6 4`	`0`	
12	`-7 5 6`	`0`	Define s_7
13	`7 -5`	`0`	
14	`7 -6`	`0`	
15	`8 1 -7`	`0`	Define p_8
16	`-8 -1`	`0`	
17	`-8 7`	`0`	
18	`9 -1 2 -7`	`0`	Define p_9
19	`-9 1`	`0`	
20	`-9 -2`	`0`	
21	`-9 7`	`0`	
22	`-10 8 9`	`0`	Define s_{10}
23	`10 -8`	`0`	
24	`10 -9`	`0`	

(E) CPOG Assertions

ID	Clause		Hint		Explanation
25	`a 5 1 3`	`0`	`3 6`	`0`	$\overline{x}_1 \wedge \overline{x}_3 \Rightarrow p_5$
26	`a 6 1 -3`	`0`	`4 9`	`0`	$\overline{x}_1 \wedge x_3 \Rightarrow p_6$
27	`a 3 7 1`	`0`	`13 25`	`0`	$\overline{x}_3 \wedge \overline{x}_1 \Rightarrow s_7$
28	`a 7 1`	`0`	`27 14 26`	`0`	$\overline{x}_1 \Rightarrow s_7$
29	`a 8 1`	`0`	`28 15`	`0`	$\overline{x}_1 \Rightarrow p_8$
30	`a 5 -1 3`	`0`	`1 6`	`0`	$x_1 \wedge \overline{x}_3 \Rightarrow p_5$
31	`a 6 -1 -3`	`0`	`2 9`	`0`	$x_1 \wedge x_3 \Rightarrow p_6$
32	`a 3 7 -1`	`0`	`13 30`	`0`	$\overline{x}_3 \wedge x_1 \Rightarrow s_7$
33	`a 7 -1`	`0`	`32 14 31`	`0`	$x_1 \Rightarrow s_7$
34	`a 9 -1`	`0`	`5 33 18`	`0`	$x_1 \Rightarrow p_9$
35	`a 1 10`	`0`	`23 29`	`0`	$\overline{x}_1 \Rightarrow s_{10}$
36	`a 10`	`0`	`35 24 34`	`0`	s_{10}

(F) Input Clause Deletions

CPOG line			Explanation
`d 1`	`36 8 10 12 16 21 22`	`0`	Delete clause 1
`d 2`	`36 7 11 12 16 21 22`	`0`	Delete clause 2
`d 3`	`36 8 10 12 17 19 22`	`0`	Delete clause 3
`d 4`	`36 7 11 12 17 19 22`	`0`	Delete clause 4
`d 5`	`36 16 20 22`	`0`	Delete clause 5

Figure 1 Example formula (A), its POG representation (B), and its CPOG proof (C), (E), and (F)

1.3 Reverse Implication Proof

Figure 1(F) shows the RUP proof steps required to delete the input clauses. Consider the first of these, deleting input clause $\overline{x}_1 \vee x_3 \vee \overline{x}_4$. The requirement is to show that there is no total assignment α that falsifies this clause but assigns $\alpha(s_{10}) = 1$. The proof proceeds by first assuming that the clause is false, requiring $\alpha(x_1) = 1$, $\alpha(x_3) = 0$, and $\alpha(x_4) = 1$. The hint then consist of unit clauses (e.g., clause 36 asserting that $\alpha(p_{10}) = 1$) or clauses that cause unit propagation. Hint clauses 8 and 10 force the assignments $\alpha(p_5) = \alpha(p_6) = 0$. These, plus hint clause 12 force $\alpha(s_7) = 0$. This, plus hint clauses 16 and 21 force $\alpha(p_8) = \alpha(p_9) = 0$, leading, via clause 22, to $\alpha(s_{10}) = 0$. But this contradicts clause 36, completing the RUP proof. The deletion hints for the other input clauses follow similar patterns—they work from the bottom nodes of the POG upward, showing that any total assignment that falsifies the clause must assign $\alpha(s_{10}) = 0$.

Deleting the asserted clauses is so simple that we do not show it. It involves simply deleting the clauses from clause number 35 down to clause number 25, with each deletion using the same hint as were used to add that clause. In the end, therefore, only the defining clauses for the POG nodes and the unit clause asserting s_{10} remain, completing a proof that the POG is logically equivalent to the input formula.

Observe that the forward implication proof shown in Figure 1 must "visit" nodes $\mathbf{p}_5$ and $\mathbf{p}_6$ twice, separately considering assignments where $\alpha(x_1) = 1$ (clauses 25 and 26) and where $\alpha(x_1) = 1$ (clauses 30 and 31). Section 2.3 shows how to use a lemma such that these nodes are each only visited once.

2 Optimizations for Forward Implication Proofs

The two optimizations we have implemented exploit the power of our general resolution framework to define new structures within the proof. They create new product nodes that are not part of the POG representation.

2.1 Literal Grouping

A single recursive step of validate can encounter product nodes having many literals as children. The naive formulation of validate considers each literal $\ell \in \lambda$ separately. Literal grouping allows all literals to be validated with a single call to a SAT solver. It collects those literals $\ell_1, \ell_2, \ldots, \ell_m$ that cannot be validated by BCP and defines a product node $\mathbf{v}$ having these literals as children. The goal then becomes to prove that any total assignment must yield 1 for extension variable v. A single call to the solver can generate this proof by invoking it on the formula $\psi|_\rho \cup \theta_v|_{\{\overline{v}\}}$, which should be unsatisfiable. The proof steps can be mapped back into clause addition steps in the CPOG file, incorporating the input clauses and the defining clauses for $\mathbf{v}$ into the hints.

2.2 Lemmas

As we have noted, the recursive calling of validate starting at root $\mathbf{r}$ effectively expands the POG into a tree, and this can lead to an exponential number of calls. These shared subgraphs arise when the knowledge compiler employs *clause caching* to detect that the simplified set of clauses arising from one partial assignment to the literals matches that of a previous partial assignment [1]. When this dec-DNNF node is translated into POG node $\mathbf{u}$, the proof generator can assume (and also check), that there is a simplified set of clauses $\gamma_{\mathbf{u}}$ for which the subgraph with root $\mathbf{u}$ is its POG representation.

The proof generator can exploit the sharing of subgraphs by constructing and proving a *lemma* for each node $\mathbf{u}$ having indegree($\mathbf{u}$) > 1. This proof shows that any total assignment α that satisfies formula $\gamma_{\mathbf{u}}$ and the defining clauses for the POG must yield $\alpha(u) = 1$. This lemma is then invoked for every node having $\mathbf{u}$ as a child. As a result, the generator will make recursive calls during a call to validate only once for each node in the POG.

The challenge for implementing this strategy is to find a way to represent the clauses for the simplified formula $\gamma_{\mathbf{u}}$ in the CPOG file. Some may be unaltered input clauses, and these can be used directly. Others, however will be clauses that do not appear in the input formula. We implement these by adding POG product nodes to the CPOG file to create the appropriate clauses. Consider an *argument* clause $C \in \gamma_{\mathbf{u}}$ with $C = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$. If we define a product node $\mathbf{v}$ with arguments $\overline{\ell}_1, \overline{\ell}_2, \ldots, \overline{\ell}_k$, we will introduce a defining clause $v \vee \ell_1 \vee \ell_2 \vee \cdots \ell_k$. We call this a *synthetic* clause having $\overline{v}$ as the *guard literal*. That is, a partial assignment ρ such that $\rho(v) = 0$ will *activate* the clause, causing it to represent argument clause C. On the other hand, a partial assignment with $\rho(v) = 1$ will cause the clause to become a tautology and therefore have no effect.

Suppose for every clause $C_j \in \gamma_{\mathbf{u}}$ that does not correspond to an input clause, we generate a synthetic clause C'_j with guard literal $\overline{v}_j$, for $1 \leq j \leq m$. Let $\gamma'_{\mathbf{u}}$ be the formula where each clause C_j is replaced by synthetic clause C'_j, while input clauses in $\gamma_{\mathbf{u}}$ are left unchanged. Let $\beta = \{\overline{v}_1, \overline{v}_2, \ldots, \overline{v}_m\}$. Invoking validate($\mathbf{u}, \beta, \gamma'_{\mathbf{u}}$) will then prove a lemma, given by the target clause $u \vee v_1 \vee v_2 \vee \cdots \vee v_m$, showing that any total assignment α that activates the synthetic clauses will have $\alpha(u) = 1$.

Later, when node $\mathbf{u}$ is encountered by a call to validate($\mathbf{u}, \rho, \psi$), we invoke the lemma by showing that each synthetic clause C_j matches some simplified clause in $\psi|_\rho$. More precisely, for $1 \leq j \leq m$, we use clause addition to assert the clause $\overline{v}_j \vee \bigvee_{\ell \in \rho} \overline{\ell}$, showing that synthetic clause C_j will be activated. Combining the lemma with these activations provides a derivation of the target clause for the call to validate.

Observe that the lemma structure can be hierarchical, since a shared subgraph may contain nodes that are themselves roots of shared subgraphs. Nonetheless, the principles described allow the definition, proof, and applications of a lemma for each shared node in the graph. For any node $\mathbf{u}$, the first call to validate($\mathbf{u}, \rho, \psi$) may require further recursion, but any subsequent call can simply reuse the lemma proved by the first call.

2.3 Lemma Example

Figure 2 shows an alternate forward implication proof for the example of Figure 1 using a lemma to represent the shared node $\mathbf{s}_7$. We can see that the POG with this node as root encodes the Boolean formula $x_3 \leftrightarrow x_4$, having a CNF representation consisting of the clauses $\{x_3, \overline{x}_4\}$ and $\{\overline{x}_3, x_4\}$. The product node declarations shown in Figure 2(A) create synthetic clauses 25 and 28 to encode these arguments with activating literals $\overline{v}_{11}$ and $\overline{v}_{12}$, respectively. Clauses 31–34 then provide a proof of the lemma, stating that any assignment α that activates these clauses will assign 1 to s_7. Clauses 35 and 36 state that an assignment with $\alpha(x_1) = 0$ will cause the first synthetic clause to activate due to input clause 3, and it will cause the second synthetic clause to activate due to input clause 4. From this, clause 37 can use the lemma to state that assigning 0 to x_1 will cause s_7 to evaluate to 1. Similarly, clauses 39 and 40 serve to activate the synthetic clauses when $\alpha(x_1) = 1$, due to input clauses 1 and 2, and clause 41 then uses the lemma to state that assigning 1 to x_1 will cause s_7 to evaluate to 1.

In this example, adding the lemma increases the proof length, but that is only because it is such a simple formula.

(A) Additional nodes

ID	CPOG line	Explanation
25	`p 11 -3 4 0`	$v_{11} = \overline{x}_3 \wedge^{\mathsf{p}} x_4$
28	`p 12 3 -4 0`	$v_{12} = x_3 \wedge^{\mathsf{p}} \overline{x}_4$

(B) Implicit Clauses

ID	Clauses	Explanation
25	`11 3 -4 0`	Argument clause $\{x_3, \overline{x}_4\}$, activated by $\overline{v}_{11}$
26	`-11 -3 0`	
27	`-11 4 0`	
28	`12 -3 4 0`	Argument clause $\{\overline{x}_3, x_4\}$, activated by $\overline{v}_{12}$
29	`-12 3 0`	
30	`-12 -4 0`	

(C) CPOG Assertions

ID	Clause	Hint	Explanation
Lemma Proof			
31	`a 5 11 12 3 0`	`25 6 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge \overline{x}_3 \Rightarrow p_5$
32	`a 6 11 12 -3 0`	`28 9 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge x_3 \Rightarrow p_6$
33	`a 3 7 11 12 0`	`13 31 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge \overline{x}_3 \Rightarrow s_7$
34	`a 7 11 12 0`	`33 14 32 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \Rightarrow s_7$
Lemma Application #1			
35	`a -11 1 0`	`26 27 3 0`	$\overline{x}_1 \Rightarrow \overline{v}_{11}$
36	`a -12 1 0`	`29 30 4 0`	$\overline{x}_1 \Rightarrow \overline{v}_{12}$
37	`a 7 1 0`	`35 36 34 0`	$\overline{x}_1 \Rightarrow s_7$
38	`a 8 1 0`	`37 15 0`	$\overline{x}_1 \Rightarrow p_8$
Lemma Application #2			
39	`a -11 -1 0`	`26 27 1 0`	$x_1 \Rightarrow \overline{v}_{11}$
40	`a -12 -1 0`	`29 30 2 0`	$x_1 \Rightarrow \overline{v}_{12}$
41	`a 7 -1 0`	`39 40 34 0`	$x_1 \Rightarrow s_7$
42	`a 9 -1 0`	`5 41 18 0`	$x_1 \Rightarrow p_9$
43	`a 1 10 0`	`23 38 0`	$\overline{x}_1 \Rightarrow s_{10}$
44	`a 10 0`	`43 24 42 0`	s_{10}

Figure 2 Example of lemma definition, proof, and application

3 More Information about the Formally Verified Checker

Our verified toolchain has been implemented in Lean 4, which is based on a formal logical foundation in which expressions have a computational interpretation. As with other proof assistants like Isabelle and Coq, a function defined within the formal system can be compiled to efficient code. At the same time, we can state and prove claims about the function within the system, thereby verifying that the functions compute the intended results.

Our code is contained in the directory `ProofChecker`. The main checker loop is implemented in `Checker/CheckerCore.lean`. That file defines a structure `PreState` that contains all the relevant data structures, which include the input CNF formula, the clause database, the POG under construction, and the root literal. We define a `State` of the checker to be a `PreState` that satisfies all the invariants that we need to establish the final result, and the bulk of our work involved showing that these are indeed preserved by each step of the proof, which modifies the clause database and the POG. The ring evaluator and model counter are contained in the `Count` directory. Here we provide a few additional notes on the implementation and clarify the sense in which the toolchain has been verified.

3.1 Implementation notes

When thinking about the formal verification, it is helpful to distinguish between the data structures that play a role in the code that is executed and the definitions that serve as a mathematical reference, allowing us to state our specifications and prove that they are met. Among the former is our representation of CNF formulas: a literal is represented as a nonzero integer, a clause is an array of literals, and a CNF formula is an array of clauses.

```
def ILit := { i : Int // i ≠ 0 }
abbrev IClause := Array ILit
abbrev ICnf := Array IClause
```

We define the clause database `ClauseDb` as a hashmap that stores each clause together with a flag indicating whether it has been deleted. Each element of a POG is either a variable, a binary disjunction (sum), or an arbitrary conjunction (product):

```
inductive PogElt where
  | var  : Var → PogElt
  | disj : Var → ILit → ILit → PogElt
  | conj : Var → Array ILit → PogElt
```

In the first case, the argument `x` in the expression `var x` is the index of an original variable; in `disj x left right` and `conj x args` it is an extension variable appearing in the CPOG file. Note that representing edges as literals allows us to negate the arguments to `disj` and `conj`. A `Pog` is then an array of `PogElt` that is well-founded in the sense that each element depends only on prior elements in the array.

On the mathematical side, we define propositional formulas and their semantics in the usual way. Functions `ILit.toPropForm`, `IClause.toPropForm`, `ICnf.toPropForm`, and `Pog.toPropForm` relate our data structures to propositional formulas and their semantics. For example, given a literal `u`, `Pog.toPropForm P u` denotes the (unnegated or negated) interpretation of the node corresponding to `u` in the POG `P` as a propositional formula $\phi_{\mathtt{u}}/\neg\phi_{\mathtt{u}}$ over the original variables. Lean provides us with convenient "anonymous projector" notation that allows us to write `P.toPropForm u` instead of `Pog.toPropForm P u` when `P` has type `Pog`, `C.toPropForm` instead of `IClause.toPropForm C` when `C` has type `IClause`, etc.

In order to reason about the behavior of the checker, we found it important to work with propositional formulas modulo logical equivalence, a structure known in logic as the *Lindenbaum–Tarski algebra*. We defined the quotient and lifted the Boolean operations and the entailment relation to this new type. The advantage is that equivalent formulas give rise to equal elements in the quotient. This makes it much easier to prove equivalences involving complicated expressions, since it enables us to substitute elements of the quotient for others even when the corresponding formulas are merely equivalent but not equal. This gave rise to auxiliary challenges, however. For example, it is no longer straightforward to say that an element of the quotient "depends" on a variable, since equivalent formulas can have different variables—consider $x \lor \neg x$ and $\top$. Instead, we made use of a semantic notion of dependence, in which an element ϕ of the quotient depends on a variable if and only if for some truth assignment the value of ϕ changes when the assignment to that variable is flipped.

3.2 Trust

In this subsection, we clarify what has been verified and what has to be trusted. Recall that our first step is to parse CNF and CPOG files in order to read in the initial formula and the proof. We do not verify this step. Instead, the verified checker exposes flags `--print-cnf` and `--print-cpog` which reprint the consumed formula or proof, respectively. Comparing this to the actual files using `diff` provides an easy way of ensuring that what was parsed matches their contents. This involves trusting only the correctness of the print procedure and `diff`. Similarly, if one wants to establish the correctness of the POG contained in the CPOG file, one can print out the POG that is constructed by the checker and compare.

Lean's code extraction replaces calculations on natural numbers and integers with efficient but unverified arbitrary precision versions. Lean also uses an efficient implementation of arrays; within the formal system, these are defined in terms of lists, but code extraction replaces them with dynamic arrays and uses reference counting to allow destructive updates when it is safe to do so [4]. Finally, Lean's standard library implements hashmaps in terms of arrays. Many of the basic properties of hashmaps have been formally verified, but not all. In particular, we make use of a fold operation whose verification is not yet complete. Thus our proofs depend on the assumption that the fold operation has the expected properties.

In summary, in addition to trusting Lean's foundation and kernel checker, we also have to trust that code extraction respects that foundation, that the implementation of the fold operation for hashmaps satisfies its description, and that, after parsing, the computation uses the correct input formula. All of our specifications have been completely proven and verified relative to these assumptions.

4 Detailed Experimental Results

Our experiments are designed to evaluate the following:

- whether our toolchain can handle challenging benchmark problems,
- the effectiveness of some design choices and optimizations,
- whether our toolchain can perform end-to-end verification with preprocessed formulas,
- the comparative performance of the prototype and verified checkers, and
- how the performance of our toolchain compares to that of related verification toolchains.

4.1 Experimental Setup

As a test machine, we used a 2021 MacBook Pro laptop having a 3.2 Ghz Apple M1 processor and with 64 GB of physical memory. We also used an external 3 TB solid-state disk drive with an advertised read/write speed of one gigabit per second. Despite being a laptop, this configuration is somewhat more powerful than the nodes typically found in server clusters. The fast access to storage is especially important due to the very large files generated and processed by the tools. Our tools generated individual files with over 160 gigabytes of data.

For benchmarks, we downloaded 100 files each from the 2022 standard and weighted model counting competitions:

`https://mccompetition.org/2022/mc_description.html`

We found that 20 of these were duplicates across the two tracks, yielding a total of 180 unique benchmark problems, ranging in size from 250 to 2,753,207 clauses.

Our default configuration for the proof generator used the structural method for the forward implication proof, with the two optimizations of literal grouping and lemmas. Running with a time limit of 4000 seconds,[1] D4 was able to generate dec-DNNF representations for 124 of these, and it timed out on the rest. Our proof generator was able to convert all of the generated dec-DNNF graphs into POGs and determine how many defining clauses they would generate. A POG operation with k arguments requires $k + 1$ defining clauses, and so the total number of defining clauses in POG P equals the number of nonterminal nodes plus the number of edges in the graph, corresponding to our definition of $|P|$. The number of defining clauses ranged from 304 to 2,761,457,765 with a median of 848,784.

4.2 Performance of the Proof Generator and Checker

For each of the 124 dec-DNNF graphs generated by D4, we ran our proof generator and prototype checker, limiting proof generation to 10,000 seconds. We ran the programs to generate and check one-sided proofs for the graphs, again with a time limit of 10,000 seconds for proof generation.

Figure 3 summarizes our results in terms of the time required by D4 (X axis) versus the sum of the times for proof generation, checking, and counting (Y axis). We were able to complete a full validation for 108 of the 124 benchmarks, with times ranging from fractions of a second to 13,144 seconds, with a median of 71.6 seconds.

Relative to the runtime for D4, two problems ran faster with the generator, checker, and counter. These were for problems having small numbers of models (relative to the number of variables), and so most of the time spent by both D4 and our proof generator was in running a CDCL solver to search the very sparse solution space. CaDiCal generally outperforms the miniSAT-based solver used by D4. At the other extreme, one problem that required only 19 seconds for D4 required 8657 seconds to generate a proof and 45 seconds to check it (and count). This benchmark appears to stem from weak performance by our implementation of BCP. Overall the ratios between the combined time for generation, checking, and counting versus the time for D4 had a harmonic mean of 5.5.

Of the 16 benchmarks that could not be fully validated, one had so many defining clauses that it overflowed the 32-bit signed integers our programs use for clause identifiers. Another exited due to the virtual memory limit imposed by the operating system, and the other 14

[1] All measured times in this document are actual elapsed times, not CPU times.

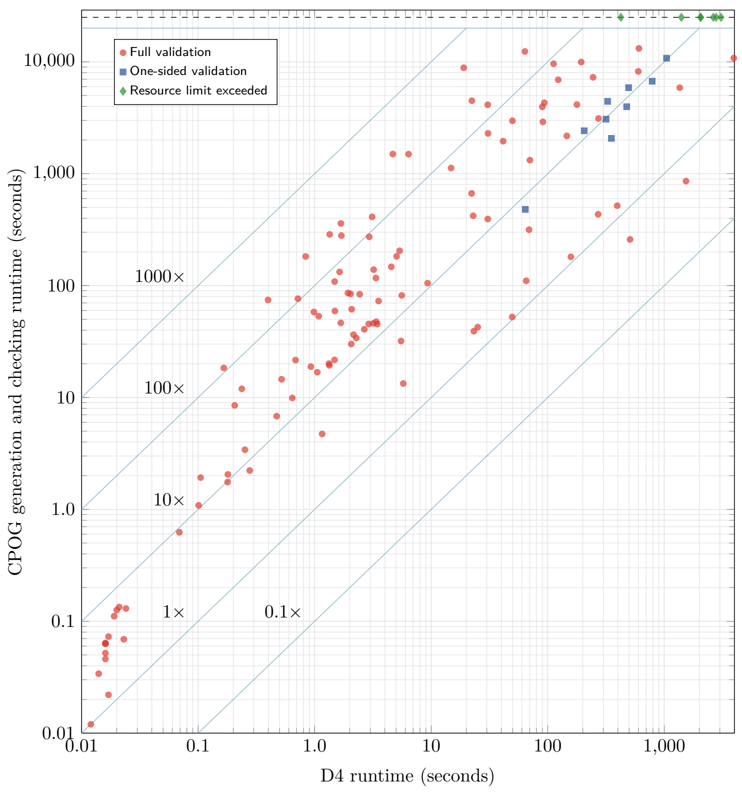


Figure 3 Combined runtime for CPOG proof generation and checking as function of D4 runtime. Timeouts are shown as points on the dashed line.

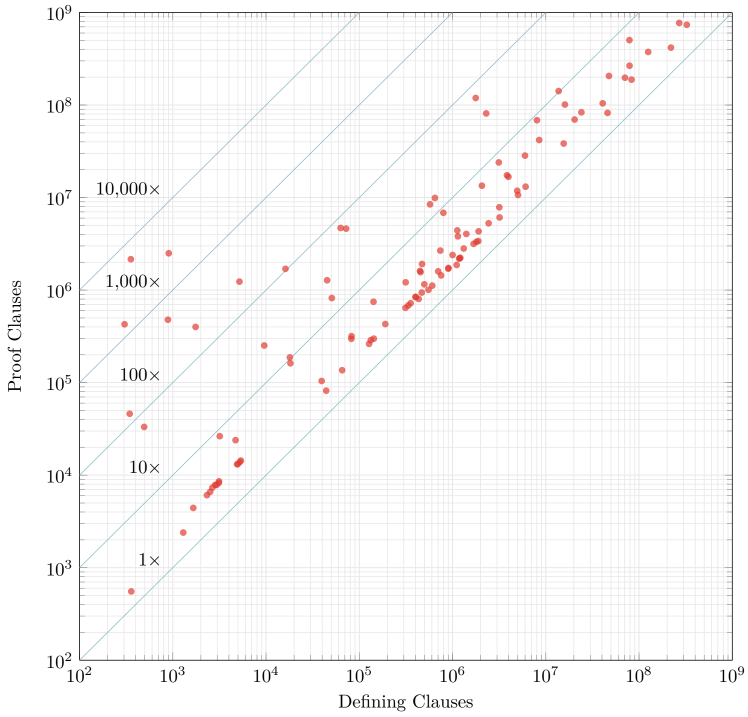


Figure 4 Total number of clauses in CPOG file as function of number of defining clauses

timed out. Of the 15 that did not overflow the clause counter, we were able to complete a one-sided verification for 9, but the other 6 timed out. Overall, we were able to provide some level of verification for 94% of the benchmark problems.

Figure 4 compares the number of defining clauses (X axis) with the total number of clauses (Y axis) for the 108 problems that were fully verified. These totals include the defining clauses for the POG, the additional defining clauses introduced to support literal grouping and lemmas, and the clauses added by RUP steps in the forward implication proof. These totals ranged from 554 to 770,382,773, with a median of 1,719,245. The ratios of the total clauses versus defining clauses ranged from 1.54 to 6073, with a harmonic mean of 3.13. The high numbers were for problems having very few models, and so the proof clauses must encode large proofs of unsatisfiability.

4.3 Monolithic Forward Implication Proofs

To assess the relative merits of the two approaches to forward implication proof generation, we ran the proof generator in monolithic mode on the 92 problems for which the combination of structural proof generation, checking, and counting required at most 3000 seconds and then kept only those measurements for which at least one of the two approaches had combined times of at most 1000 seconds. There were a total of 83 such problems. Figures 5 and 6 show comparisons between the two approaches in terms of time and total clauses.

Examining Figure 5, we can see that the monolithic approach ran faster than the structural approach for 23 of the problems, including a case where the generator, checker, and counter required only one second, versus 71 seconds for the structural approach. On the other hand, the monolithic approach timed out for 13 of the cases and was slower for the other 37. In general, we can see the monolithic approach faring worse on larger problems.

Figure 6 shows the total number of clauses for the two approaches for the 70 cases where both approaches completed within 1000 seconds. As can be seen, the monolithic approach consistently produces shorter proofs, perhaps due to the benefits of the proof trimming by DRAT-TRIM.

These experiments suggest that a hybrid of the two approaches might yield the best results in terms of consistency, runtime, and proof size. It would begin the recursive descent of validate according to a structural approach, but monolithically generate the proof once it encounters a sufficiently small subgraph.

4.4 Impact of Optimizations

To assess the impact of the two optimizations: literal grouping and lemmas, we ran the proof generator on the 80 problems for which the combination of the two optimizations yielded proofs totaling at most 10 million clauses. Compared to the problems used to assess monolithic proof generation (Section 4.3), this set included one that had a short proof but long runtime and excluded four that had short runtimes but long proofs. We then ran the proof generator on all three combinations of the optimizations being partially or totally disabled. Figure 7 shows how disabling these optimizations affected the total number of clauses in the proofs, where the X axis indicates the total number with both optimizations enabled, while the Y axis indicates the number with one (left) or both (right) optimizations disabled.

As the figures illustrated, each optimization on its own can shorten the proofs, sometimes substantially, and the two combine to have even greater effect. Of the 80 benchmark problems evaluated, and comparing with both optimizations disabled, 47 had the number of proof

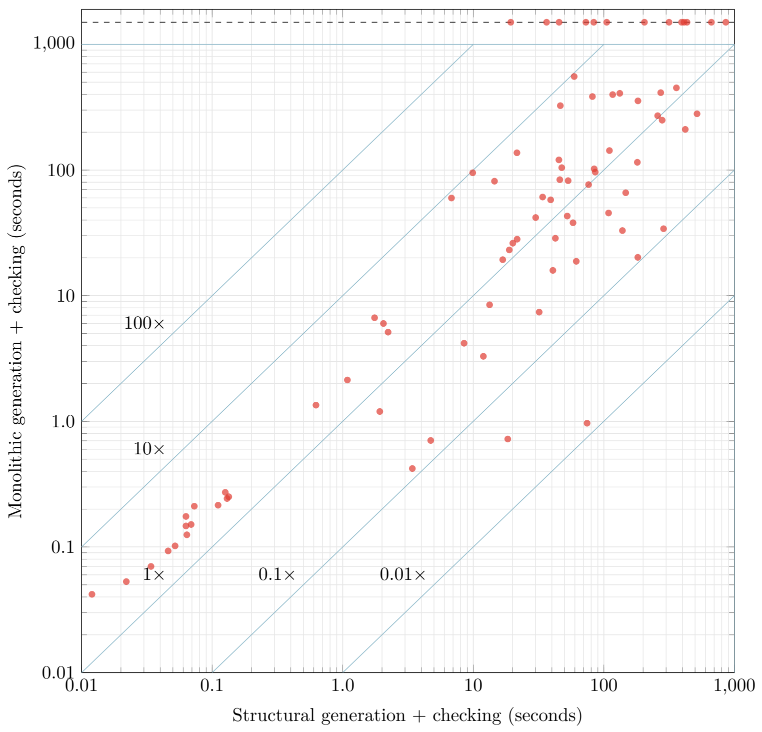


Figure 5 Monolithic versus Structural Validation: Time. Timeouts are shown as points on the dashed lines.

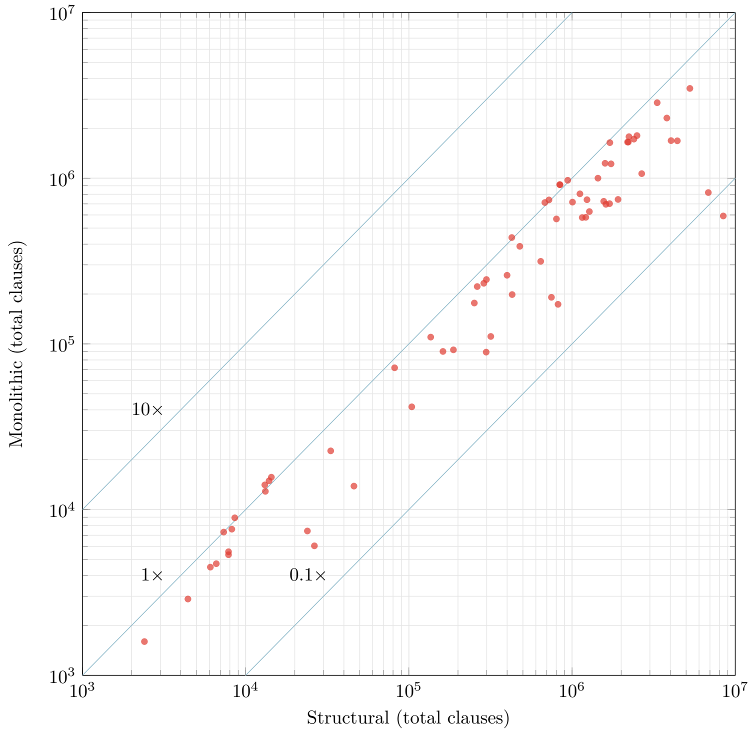


Figure 6 Monolithic versus Structural Validation: Clauses

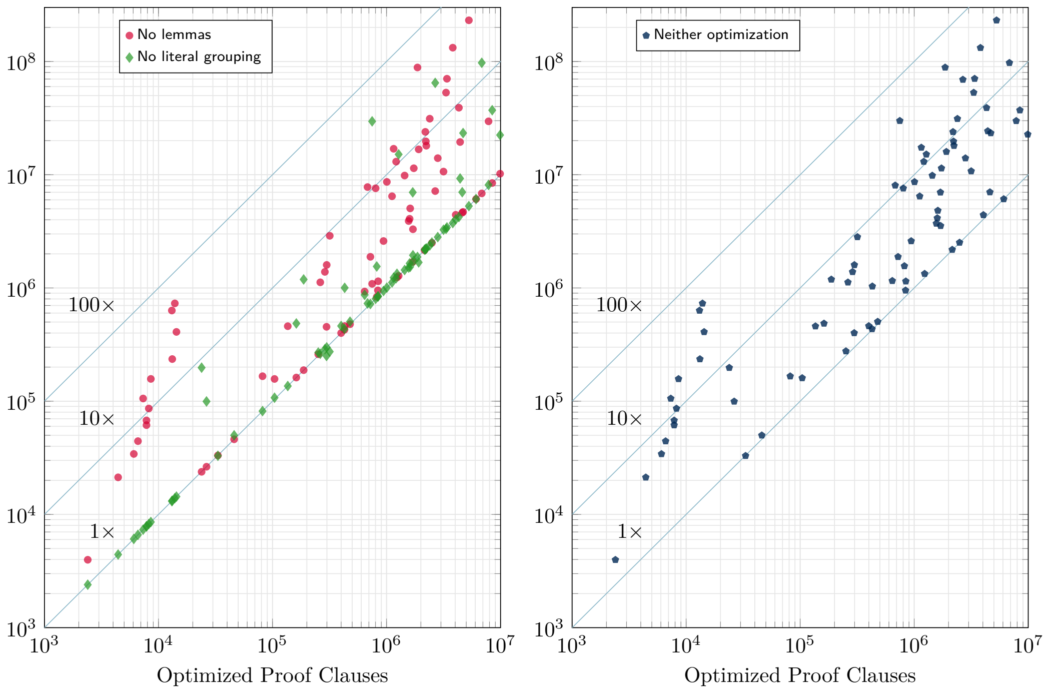


Figure 7 Proof clauses when one (left) or both (right) optimizations are disabled, versus when both optimizations are enabled

clauses reduced by at least a factor of 2 by using lemmas, 14 by using literal merging, and 60 by enabling both optimizations. In the extreme cases, a lack of lemmas caused one proof to grow by a factor of 52.5, while a lack of literal grouping caused another proof to grow by a factor of 39.6.

The time performance of these optimizations is less dramatic, but still significant. Lemmas sometimes made proof generation slower, but in one case ran 38× faster. Similarly, literal grouping could take longer, but in one case ran 3.9× faster.

Overall, it is clear that these two optimizations are worthwhile, and sometimes critical, to success for some benchmarks.

4.5 Certifying Preprocessed Formulas

The D4 knowledge compiler can optionally perform three different forms of preprocessing on the input formula. These transform the formula while preserving logical equivalence [3]. Proving that the combination of preprocessing plus knowledge compilation preserved logical equivalence would be a valuable capability. However, proof generators that operate based on the structure of the input clauses, including our structural approach, will generally fail in this case, since the compilation was not based on the original input clauses.

One strength of monolithic proof generation is that it makes no assumptions about how the compiled representation was generated. A complete proof-generating SAT solver could, in principle, always generate the forward implication proof, as long as forward implication holds. To test this hypothesis, we ran our proof framework with preprocessing enabled on the same 83 benchmarks as in Section 4.3. We measured the time for two entire toolchains: one with compilation, proof generation, checking, and counting, and the other starting with

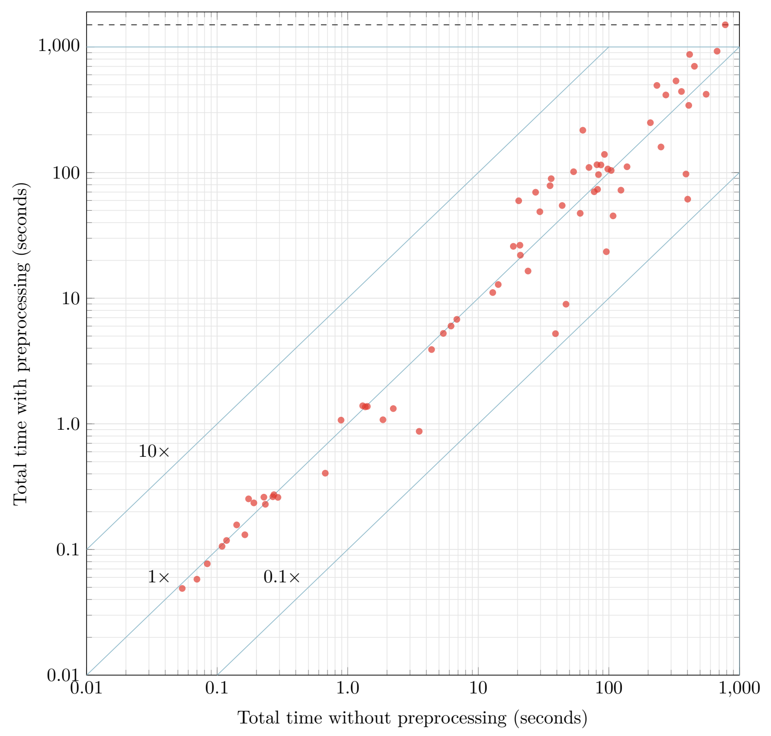


Figure 8 Runtime impact of preprocessing. All use monolithic generation. Times include running D4

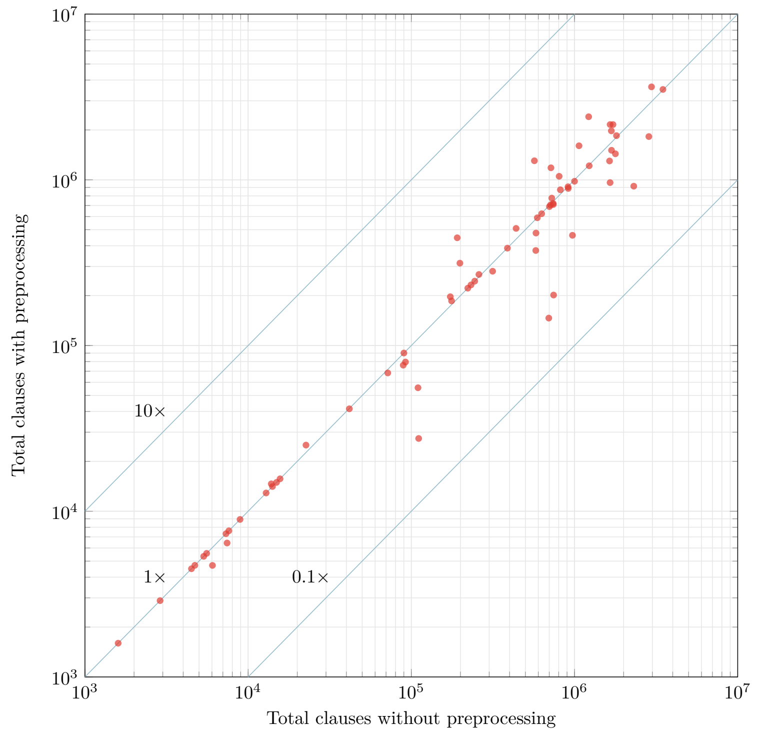


Figure 9 Impact of preprocessing on total clauses. All use monolithic generation

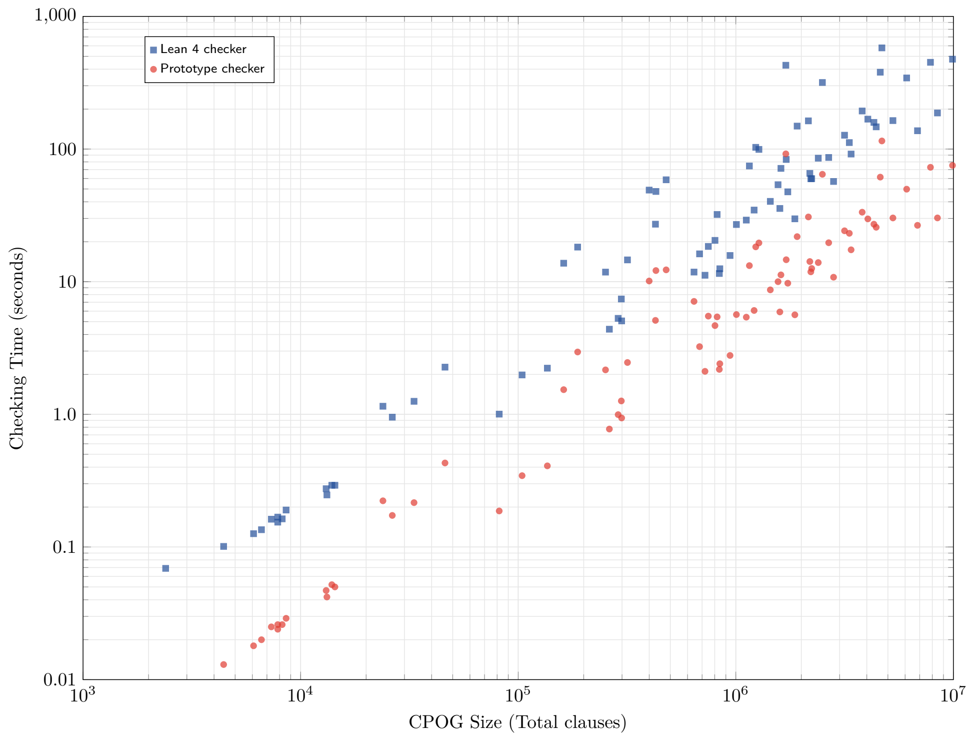


Figure 10 Times for Prototype and Verified Checker, as Functions of Total Clauses

preprocessing and then having the same steps as the other. The results are summarized in Figures 8 and 9. Setting a time limit of 1000 seconds, 70 of the formulas completed without preprocessing, while 69 did so with preprocessing. The one that timed out with preprocessing did so due to the excessive time required for preprocessing. Of the 69, 53 ran faster with preprocessing, while 15 ran slower. The sizes of the generated CPOG proofs were more variable: 31 had fewer clauses with preprocessing, 21 had more, and 17 were identical.

4.6 Evaluation of the Verified Checker

We ran the Lean 4 checker on the 80 benchmark problems for which the generated CPOG file had at most 10 million total clauses. These are the same benchmarks used in evaluating the optimizations (Section 4.4). All of the checks completed and confirmed the outcome of the prototype checker.

Figure 10 shows the performance of both the Lean 4 checker and the prototype checker on the Y axis, as functions of the total number of clauses on the X axis. As would be expected, the Lean 4 checker consistently runs slower than the prototype checker. The ratio of the runtime for Lean 4 versus the runtime for the prototype checker had a harmonic mean of 5.94.

Importantly, however, it can be seen that both checkers show the same overall performance trends. The ratio of the two runtimes was less than 8.0 for all but three small benchmarks. This seems like a reasonable price to pay for a rigorous guarantee of correctness, and we are confident that we can reduce this gap with more effort in optimizing the verified checker.

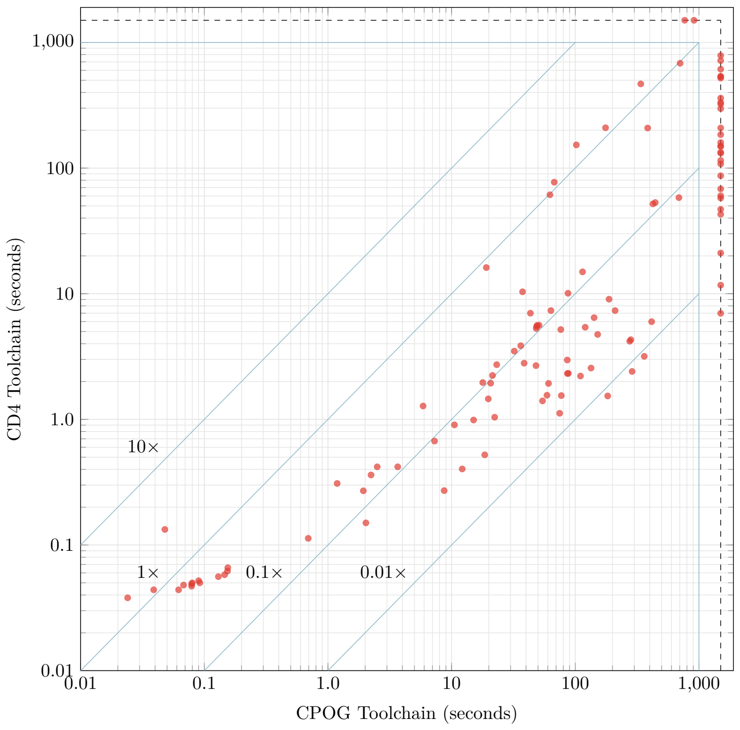


Figure 11 Times for CD4 Toolchain versus CPOG Toolchain. Times include knowledge compilation, proof generation, and checking

5 Comparison with other Certification Frameworks

This section provides more insights into how our proof framework (CPOG) compares with other work in certified knowledge compilation and model counting.

5.1 Comparison with CD4 Framework

Figure 11 shows the results of a set of experiments comparing the performance of our toolchain (CPOG) versus one based on CD4. It shows the timings for the 110 benchmarks for which either the CPOG toolchain or the CD4 toolchain completes in less than 1000 seconds. Both toolchains include compilation, in addition to proof generation and checking. Of these, 80 completed for both toolchains, 28 timed out or failed with the CPOG toolchain, and 2 timed out for the CD4 toolchain. The data clearly show that the CD4 toolchain can handle larger benchmarks and generally runs faster. Only 7 of the 80 that completed with both toolchains ran faster with the CPOG toolchain. For these 80, the toolchain based on CD4 ran faster by a harmonic mean factor of 19.8×.

On the other hand, the CD4 proof generation and checking is tightly connected to the operation of D4, and every change to the program could require changes to the proof steps and possibly changes to the proof framework. For example, we have found that CD4 does not work properly when the compiler makes use of its internal preprocessor.

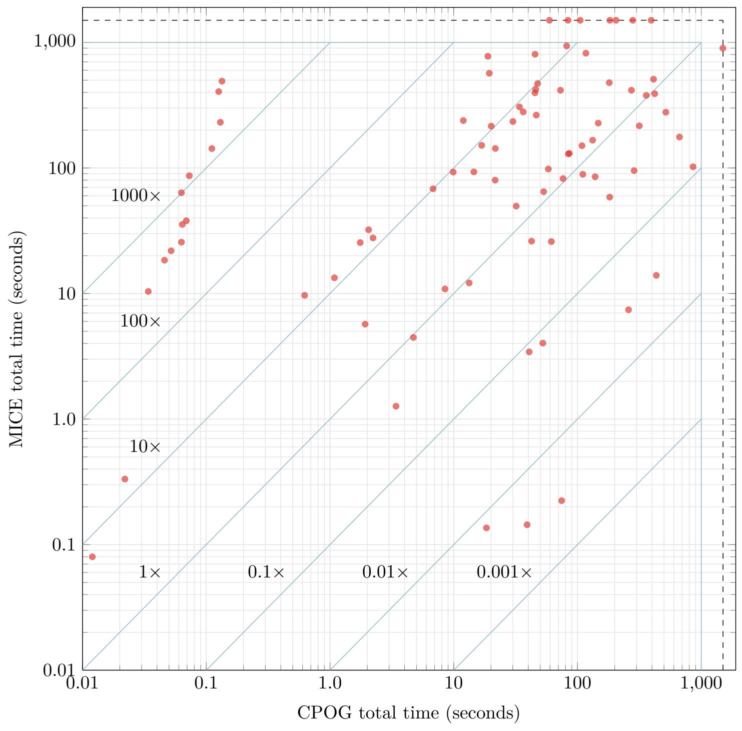


Figure 12 Running Time for MICE versus CPOG proof chains. Times include proof generation, checking, and counting. Timeouts are shown as points on the dashed lines.

5.2 Comparison with the MICE Framework

To compare the performance of our toolchain versus that of MICE [2], we consider how the combination of proof generation, proof checking, and counting in the CPOG toolchain compares to the time to run their proof generator NNF2TRACE and checker SHARPTRACE. We ran their tools on the 92 benchmark problems for which our toolchain requires at most 3000 seconds and retained the 84 data points for which at least one of the toolchains completes in under 1000 seconds.

We can summarize the results as:

- 84 total data points
- 76 were completed by both programs in under 1000 seconds
- Of those 21 were faster with MICE, 55 with CPOG
- 7 completed with CPOG under 1000 seconds but exceeded that threshold for MICE
- 1 completed with MICE under 1000 seconds but exceeded that threshold for CPOG

The detailed results are shown in Figure 12. As can be seen, the comparative runtimes are highly variable, reflecting the fact that the two toolchains differ fundamentally in their objectives and their approaches.

One shortcoming of NNF2TRACE can be seen in the near-vertical series of points in the upper-lefthand corner. These correlate with a similar set of points along the left in Figure 7. Like the naive implementation of validate, their program recursively traverses the graph representation of a formula, effectively expanding it into a tree. They lack an optimization analogous to lemmas.

We have not tried the MICE toolchain on larger benchmarks, but we suspect it would encounter significant scaling limitations.

References

1 Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2002.

2 Johannes K Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

3 Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

4 Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: reference counting optimized for purely functional programming. In *Implementation and Application of Functional Languages (IFL)*, pages 3:1–3:12. ACM, 2019.