Engineering an Efficient Preprocessor for Model Counting

Mate Soos
University of Toronto
Canada

Kuldeep S. Meel
University of Toronto
Canada

ABSTRACT

Given a formula F, the problem of model counting is to compute the number of solutions (also known as models) of F. Over the past decade, model counting has emerged as key building block of quantitative reasoning in design automation and artificial intelligence. Given the wide-ranging applications, scalability remains the major challenge. Motivated by the observation that the formula simplification can dramatically impact the performance of the state-of-the-art exact model counters, we design a new state-of-the-art preprocessor, Arjun2, that relies on tight integration of techniques. The design of Arjun2 is motivated from our observation that it is often beneficial to employ preprocessing techniques whose overhead may be prohibitive for the task of SAT solving but not for model counting: accordingly, we rely on a specifically tailored SAT solver design for redundancy detection, sampling-boosted backbone detection, as well as storing of redundancy information for the purposes of improving propagation within top-down model counters. Our detailed empirical evaluation demonstrates that Arjun2 achieves significant performance improvements over prior model counting preprocessors in terms of instance-size reductions achieved as well as the runtime improvements of the downstream model counters.

ACM Reference Format:
Mate Soos and Kuldeep S. Meel. 2024. Engineering an Efficient Preprocessor for Model Counting. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658489

1 INTRODUCTION

Given a formula F, the problem of model counting is to compute the number of models (i.e., satisfying assignments or solutions) of the formula F. Model counting is a fundamental problem in design automation and artificial intelligence, with applications ranging from quantified information flow analysis [14], network reliability [11], neural network verification [3], software reliability, and the like. Motivated by the wide ranging applications, there have been sustained effort in the development of exact (as well as approximate) model counters over the past decade; perhaps best illustrated by the presence of yearly organized model counting competitions [15]. While the efforts over the years have significantly improved the runtime performance of model counters, scalability continues to remain the primary challenge.

In this study, we focus on expanding the reach and efficiency of current state-of-the-art model counters by employing preprocessing methods specifically designed for model counting. It's important to note that in our work, we concentrate on exact model counters. Approximate model counters, such as ApproxMC [31], operate in a fundamentally different manner. Therefore, preprocessing techniques that enhance the performance of exact model counters may adversely affect the performance of ApproxMC [29].

One of the primary motivations for incorporating preprocessing to enhance the efficiency of model counters originates from Boolean satisfiability (SAT) solving. In SAT solving, preprocessing has proven to be a vital component and is seamlessly integrated into modern SAT solvers. However, it's essential to highlight a key distinction. While correctness in SAT preprocessing merely necessitates the maintenance of the satisfiability status of input instances, preprocessing for model counting imposes a much stricter requirement: the preservation of the *number* of satisfying assignments. This constraint limits the applicability of certain SAT preprocessing techniques to model counting. Conversely, since model counting is significantly more computationally challenging than SAT solving, it allows us to leverage more potent techniques that may not be viable in the realm of SAT solving.

The main contribution of this work is to develop an efficient preprocessing engine called Arjun2 for model counting. Arjun2 incorporates a wide range of preprocessing techniques. In particular, Arjun2 integrates three types of preprocessing techniques: (A) lightweight preprocessing techniques that draw their origins from the field of SAT solving but whose extent has been tailored to model counting, (B) SAT-based preprocessing techniques wherein we rely on lightweight SAT solving for redundancy detection, (C) sampling-based backbone detection wherein we rely on recent progress in the design of efficient CNF samplers. All these techniques preserve the count of models, allowing them to be used in conjunction with exact counters. Furthermore, we propose a new form of preprocessing that provides a model counter with information about redundant clauses detected during preprocessing to enhance propagation within the counter.

Through extensive empirical evaluation, we demonstrate that Arjun2 significantly simplifies problem instances compared to other existing model counting preprocessors, by removing a significantly larger number of variables and clauses from the formula. The integration of Arjun2 with state-of-the-art model counters greatly enhances their runtime efficiency and scalability.

2 BACKGROUND

Let $X = \{x_1, x_2, \ldots x_n\}$ be the set of Boolean variables and let F be a Boolean formula in conjunctive normal form (CNF) defined over X. An assignment $\sigma : X \mapsto \{0, 1\}$ is called a *satisfying assignment* or *solution* of F if it makes F evaluate to true. We denote the set of all solutions of F by $\mathsf{Sol}(F)$. Often, we are only interested in counting over a subset of variables $P \subseteq X$. Let $\mathsf{Sol}(F)_{\downarrow P}$ to indicate the projection of $\mathsf{Sol}(F)$ on P. Given a formula F over X, the problem

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658489

of model counting is to compute $|\mathsf{Sol}(F)|$. Similarly, given a formula F and projection set $P \subseteq X$, the problem of projected counting is to compute $|\mathsf{Sol}(F)_{\downarrow P}|$.

A formula is represented in Negation Normal Form (NNF) wherein every internal node is either a disjunction ($\vee$) or conjunction ($\wedge$), and every leaf is a literal. We say that a formula F is in d-DNNF if every $\vee$-node of F is deterministic and every $\wedge$-node of F is decomposable.

2.1 Top-Down Model Counters

Top-down model counters combine conflict-driven clause learning (CDCL) with component caching to achieve scalability. On a high level, the trace of a top-down model counter can be represented by a d-DNNF as the counters operate by branching on a variable, which can be represented by a deterministic $\vee$. Whenever the counter finds that the current residual formula can be decomposed into disjoint components such that the subformulas do not share variables, then the counter proceeds to perform decomposition followed by counting each of these subformulas independently, and multiplies the returned counts. Top-down counters refer to the formula represented at each node as a component. Caching refers to these counter's cache to store a component and check whether the current component has already been encountered by the counter in its execution. If so, it can retrieve the count of the component from the cache, saving on processing time.

Top-down counters have dominated the model counting competition over the past three years. As indicated earlier, scalability continues to remain a major challenge for model counters and therefore, we focus on achieving further scalability of top-down model counters through the design of efficient preprocessing techniques. To guide the development of preprocessing techniques, we rely on minimizing the following two features of the preprocessed CNF:

1. Variables With fewer variables, the counter has to make fewer decisions and propagations to reach a leaf, which can significantly reduce the size of the (implicit) d-DNNF it produces. Furthermore, the performance of top-down model counters is heavily affected by the efficacy of the component cache whose performance depends on the number of variables in the formula. The fewer the number of variables, the smaller is the expected size of each component in the cache, leading to higher expected cache hit rate.

2. Clauses With fewer clauses, the counter has a higher chance of performing disjunctions. Furthermore, removing clauses can further improve the cache hit rate, similarly to removing variables.

It is worth remarking that while these two features serve as a useful rules of thumb to help in improving preprocessing techniques, they are not the primary measures of effectiveness of a preprocessor. The primary measure for effectiveness is the reduction of total time taken by the preprocessor and the model counter.

2.2 Related Work

Preprocessing has emerged as a crucial component of modern CDCL solvers and as such all state-of-the-art SAT solvers heavily rely on pre- and in-processing. We refer the interested reader to [8] for an excellent survey. Motivated by the success of preprocessing techniques in the context of SAT solving, Lagniez and Marquis [24] initiated the study of preprocessing techniques in the context of model counting. They evaluated the effectiveness of standard preprocessing techniques proposed in SAT solving, such as vivification, occurrence reduction, backbone identification, as well as equivalence, AND, and XOR gate identification and replacement. These techniques were implemented in the tool pmc. Subsequently, Lagniez, Lonca, and Marquis sought to enhance the size reduction of preprocessed Conjunctive Normal Forms (CNFs) by leveraging the concept of definability. They developed the tool B+E (now referred to as BiPe), which has been widely adopted by researchers as a preprocessing step before invoking model counters, and has been utilized as a preprocessor in the model counting competition.

Over the past two years, we have witnessed the development of three new model counters which have demonstrated impressive runtime performance: GPMC [33], ExactMC [26], and SharpSAT-TD [21, 22]. Each of these counters have been accompanied by a dedicated preprocessing engine, which highlights the crucial role of preprocessors in the runtime performance of current state-of-the-art model counters.

3 Arjun2

In this section, we present the core technical contribution of our work: Arjun2[1], an efficient pre-processor that improves the runtime performance of state-of-the-art model counters.

Our algorithmic engineering is heavily motivated by the observation that model counting is a significantly harder problem than plain SAT solving, supported by theoretical as well as empirical evidence. Therefore, in contrast to SAT solving where preprocessing techniques need to be relatively lightweight, it is worthwhile to rely on preprocessing techniques for counting that may require calls to a SAT solver or even a sampler. Another novel aspect of Arjun2 is to allow model counters to take advantage of clauses that are removed during preprocessing. These redundant clauses are marked as such in the preprocessed formula and can be used by the model counter to improve conflict-driven clause learning search. This technique allows the model counter to achieve the best of both worlds: use clauses for conflicts but without making the resulting components larger.

In the rest of this section, we discuss the preprocessing techniques incorporated in Arjun2: Section 3.1 lists the lightweight preprocessing techniques, Section 3.2 lists techniques relying on lightweight SAT solving, Section 3.3 discusses sampling-based preprocessing, and Section 3.4 discusses redundancy recycling. Finally, Section 3.5 shows how these techniques are combined.

3.1 Lightweight Preprocessing

Preprocessing in the context of SAT solving has had a long history with many techniques proposed over the years [9, 12, 19, 20, 30]. We use the following set of techniques in Arjun2:

Backward Subsumption This is the standard clause subsumption algorithm used by SatELite [12] that generates a hash of each clause and uses a Bloom filter [10] to pre-filter candidates that can be subsumed. It does this in a *backward manner*, in the sense that instead of looking for a clause that could subsume C, we are looking for clauses that are *subsumed by* C.

[1] Code available at: github.com/meelgroup/arjun

Backward Subsumption with Resolvents This technique computes all resolvents of all variables and runs the backward subsumption query with the resulting resolvents. For example, if variable x_1 is in clauses $(x_1 \vee x_2)$ and $(\neg x_1 \vee x_3)$, its resolvent, $C_r = (x_2 \vee x_3)$. C_r, can be used to subsume $C_s = (x_2 \vee x_3 \vee x_4)$.

Replacement with OR-gates This algorithm finds *OR* gates of the form $x_1 = x_2 \vee x_3$ using [27], where each gate has two inputs. Then, it finds a clause of the form $(x_1 \vee x_2 \vee C)$ where C is any set of literals, and replaces it with the clause $(x_1 \vee C)$. This rewriting removes a literal from the formula via each successful execution.

Bounded Variable Elimination (BVE) Bounded variable elimination is ran as per SatELite [12]. However, we use all the ideas employed by the Kissat SAT solver to reduce the number of resolvents [7], via regular and irregular gate detection through UNSAT cores produced by PicoSAT [6] and through *weakening*. Weakening is a technique whereby a clause is copied and temporarily enlarged via reverse self-subsuming resolution using binary clauses. For example, the clause $(x_1 \vee x_2 \vee x_3)$ can be weakened via $(x_1 \vee \neg x_4)$ to $(x_1 \vee x_2 \vee x_3 \vee x_4)$. This helps improve the chance of obtaining more tautological resolvents during BVE.

Ternary Resolvents Arjun2 produces ternary resolvents as redundant clauses from all 3-long clauses that can be resolved with each other as per [5]. These clauses become useful later when they can subsume potentially larger irredundant clauses or can be used to strengthen them. In Section 3.4, we inject these ternary clauses into the redundant clause database of the model counter to improve its Boolean Constraint Propagation (BCP) performance, while not negatively impacting its ability to find disconnected components.

Tree-based Lookahead Tree-based lookahead [17] is a technique that can achieve three goals at the same time. It performs (1) hyper-binary resolution [2], (2) transitive reduction [1], and (3) BCP-based probing. Transitive reduction sparsifies the binary implication graph, and hyper-binary resolution adds redundant binary clauses to ensure stronger propagation properties of the formula. The new redundant binary clauses can later be used to subsume larger clauses or to strengthen them. Furthermore, they come useful later in Section 3.4 to improve the BCP strength of the model counter.

Backwards Subsumption and Strengthening This step is similar to **Backwards Subsumption** but this time we also perform backwards strengthening, using the algorithm from SatELite. Strengthening works by discovering clause pairs of the form $\{(x_1 \vee C), (\neg x_1 \vee D)\}$ where $D \subseteq C$ and replaces it with the pair $\{(C), (\neg x_1 \vee D)\}$, removing a literal.

BCP-based Clause Vivification Vivification [28] enqueues the negation of the literals of a clause one by one and performs BCP after each literal enqueued. If the BCP fails, the rest of the literals not yet enqueued can be deleted from the clause. Furthermore, conflict analysis is performed to determine the literals responsible for the conflict, potentially further strengthening the clause.

3.2 SAT-based Preprocessing

We now discuss preprocessing techniques that rely on a SAT solver.

We present a high-level overview of vivification in Algorithm 1. This algorithm attempts to check, for each clause cl and for each literal ℓ appearing in the clause cl, whether ℓ can be removed. To achieve this, we invoke a SAT solver with the assumption that it is the conjunction of the negation of all literals except ℓ appearing in cl. If the solver returns UNSAT, we can deduce that the formula φ implies cl $\setminus \ell$ (i.e., a stronger clause formed by removing the literal ℓ from cl). Therefore, ℓ can be removed from cl.

We now present a high-level overview of sparsification in Algorithm 2. Here, we create and add a new *indicator variable* for each clause. This indicator variable is later used to turn the clause on or off. To check whether a clause is needed, its relevant indicator variable is assumed to TRUE, along with all active clauses' indicator variables assumed FALSE, as well as the negation of the clause in question. A `solve()` call is then performed. If the result is UNSAT, the clause is implied by the rest of the clauses, and the clause can be removed. The clause is removed by letting its indicator literal be TRUE at toplevel, and continuing.

Both algorithms detailed above are special in that they require many assumptions for each SAT call, hence a solver that can handle assumptions at a high rate is of paramount importance. For example, if the number of clauses is m, and say 10% of clauses can be removed via sparsification, then m calls need to be made during sparsification, each with on average $.95m$ assumption literals. Here, m can be large, almost always above 1000. Normal SAT solvers would enqueue and propagate these $\approx m$ assumption literals one by one [13] for each `solve()` call and each restart within the `solve()` call. This is extremely time-consuming and would take the majority of the solving time for most SAT solvers.

The heavy usage of assumptions necessitates the design of a specialized solver, as evidenced by the development of Kitten inside the Kissat solver, and the specialized CDCL loop inside the Arjun tool [32]. In the same vein, we rely on a specialized SAT solver architecture implemented by SharpSAT-TD [22]. To handle the large number of assumptions, the lightweight SAT solver inside SharpSAT-TD, called Oracle[2], allows one to set and change assumptions at any point, without the need to propagate all assumptions at every call to `solve()` or after every restart. Instead, only the updated assumption(s) need to be propagated once, after `solve()`.

The above change necessitates that no literal can be learnt at toplevel, or re-propagation would be needed. It also makes it impossible to return a minimal reason clause in case of an UNSAT result. However, neither of these issues affect our use case.

Satisfying Assignments Cache for Vivification. During vivification every SAT result to `solve()` is a solution to the original CNF. Hence, if during any `solve()` call the set of assumptions has been seen in a solution, the call will terminate with SAT. To take advantage of this, for every `solve()` call, Oracle caches the solution and therefore, in the future instead of invoking the call to `solve()`, it can check if the set of assumptions is present in any solution already in the cache.

3.3 Sampling-based Preprocessing

Backbone detection [18] aims at discovering boolean values of variables that are shared by all solutions to a formula. These variables' values can be set to the shared fixed value since all solutions must agree on them. This in turn can help in e.g. counting solutions, as

[2] github.com/Laakeri/sharpsat-td/tree/main/src/preprocessor/oracle.cpp

Algorithm 1 Vivification (φ). As in SharpSAT-TD via Oracle

```
for cl ∈ clauses(φ) do
    assump ← ¬(c)
    for lit ℓ ∈ c do                          ▷ Try removing each literal one by one
        assump.Remove(ℓ)
        ret ← solve(assump)
        if ret == UNSAT then
            cl.Modify(¬assump)                                  ▷ Strengthen cl
            break
        else
            assump.Append(ℓ)                  ▷ Couldn't remove literal l from cl
```

Algorithm 2 Sparsification(clsList). As in SharpSAT-TD via Oracle

```
indices ← [n + 1, n + 2, ... n + len(clsList) + 1]
procClsList ← clsList
for i ∈ [0 ... len(clsList)] do                    ▷ Add extra literal to all clauses
    tmp ← clsList[i]
    tmp.Append(indices[i])                ▷ indices[i] is an indicator for i-th clause
    solver.AddClause(tmp)
for i ∈ indices do solver.SetAssumpsLit(¬indices[i])    ▷ ∀ clauses are active
for i ∈ [0 ... len(clsList)] do
    solver.SetAssumpsLit(indices[i])                          ▷ Disable this clause
    tmp ← ¬(clsList[i])
    ret ← solve(tmp)                          ▷ Check if the clause is redundant
    if ret== SAT then solver.SetAssumpsLit(¬indices[i])
    else if ret== UNSAT then procClsList.Remove(clsList[i])
return procClsList
```

these variables effectively disappear from the formula, reducing the variable, clause, and literal count of the formula.

Previous work on backbone computation has focused on formulas that were not specifically made for model counting. Formulas in the Model Counting Competition [15], in contrast to formulas in e.g. the SAT Competition [4] tend to be extremely easily satisfiable, i.e. it is very easy to find a satisfying assignment to them. This opens up the space to design backbone detection algorithms relying on subroutines harder than SAT solving. For example, SAT samplers could be used, which take in a formula as input and seek to return a satisfying assignment uniformly at random. While uniform samplers with rigorous guarantees struggle to scale, the past few years have witnessed the design of samplers that work well in practice.

We can take advantage of this by sampling the solution space and finding literals that are *rotatable* as per [18], since finding many solutions is comparably fast in a model counting setting. Rotatable literals are literals that exist both in positive and negative forms in different solutions. Clearly, these literals cannot be part of the backbone of a formula. To this end, we use the uniform-like sampler CMSGen [16] to randomly sample a number of solutions (30 in our implementation) from the solution space of the formula to discover rotatable literals. CMSGen attempts to find widely different solutions, hence, there is a high likelihood that many rotatable literals are discovered this way. This significantly reduces the number of potential candidates that could be backbone. We then run Algorithm 3 from [18] to check for the variables that have not been discovered to be rotatable.

3.4 Redundant Clause Recycling

During our running of **Tree-based Lookahead**, and **Ternary Resolution**, we collect binary (through hyper-binary resolution) and ternary (as ternary resolvents) redundant clauses. These two redundant clause classes are both known to improve the performance of SAT solvers [5, 17]. Furthermore, during sparsification, in Arjun2 the sparsified clauses are collected as redundant clauses. While redundant clauses are not necessary to get a correct count for the instance, they can significantly improve the BCP efficiency of a model counter, and hence improve its performance.

To take advantage of these collected redundant clauses, we propose an extension to the DIMACS format : these redundant clauses are emitted into the preprocessed CNF with a special syntax `'c red CLAUSE 0'`. To demonstrate the ease of usage of our extended format and integration of redundant clauses, we also implemented a modification of GANAK [29], a state-of-the-art model counter to support this format extension.

3.5 Putting it All Together

The order of preprocessing techniques matters a great deal in the quality of the final preprocessed instance. Given the large number of possible permutations, it is important to consider the potential benefits and challenges of different orders. Observe that sparsification benefits from backbone detection. Furthermore, when a clause contains a literal that is backbone, vivification reduces it to the literal, hence simulating backbone detection. Therefore, it is best to run backbone detection before vivification. While vivification and sparsification could be performed interchangeably, it is best to perform vivification first, as this will ensure that more clauses can later be sparsified away, given that after vivification clauses are in general stronger.

Arjun2 combines all the preprocessing techniques mentioned above in the following order. Firstly, we run Equivalent Literal Reasoning, BCP-based Probing, Backbone Detection, Subsumption via Binary Clauses, BCP-based Sparsification, Backward Subsumption with Resolvents, Backward Subsumption, Replacement with OR-gates, BVE, Ternary Resolvents, Tree-based Lookahead, Backwards Subsumption and Strengthening, and BCP-based Clause Vivification. This set of operations is performed twice, in the same order, as they may influence one another. We then perform Vivification and Sparsification using Oracle. We repeat the same set of lightweight operations twice more, to further simplify after the heavy-lifting performed by Oracle. Finally, the remaining CNF is renumbered such that active (i.e., non-set and uneliminated) variables are numbered from 1. This helps with the memory and cache efficiency of model counters.

4 EXPERIMENTAL EVALUATION

We conducted an extensive evaluation of Arjun2 to check its performance with respect to prior state-of-the-art preprocessors BiPe [23], KCBox [25], GPMC-Pre [33], and SharpSAT-TD-Pre [21, 22], where GPMC-Pre and SharpSAT-TD-Pre refer to the preprocessors that are integrated with the model counting tools GPMC, and SharpSAT-TD. To understand the impact of preprocessors on model counters, we experimented with an extensive set of state-of-the-art model counters: ExactMC, GPMC, D4, SharpSAT-TD, and GANAK where the experimental setup consisted of first executing a preprocessor on a given CNF file and then the output of the preprocessor is fed to the corresponding model counter. We use $P + C$ to refer to

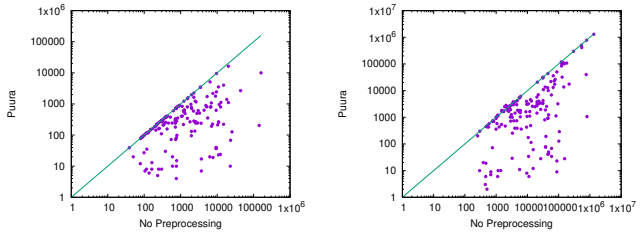

Figure 1: Impact of Arjun2 on the number of variables (left) and clauses (right)

running preprocessor P followed by the model counter C. As noted above, we have currently integrated redundant clause recycling into a modified version of GANAK. To present a fair evaluation that preserves the relative power of different model counters, we provide results for the version of GANAK without the capability of redundant clause cycling, except in the last subsection where we investigate the potential power of redundant clause recycling.

To conduct the experiments, we used the Model Counting Competition Track-1 non-projected benchmark instances that contain a number of different benchmark families, and total 200 CNF files. For all experiments, we used a cluster with AMD EPYC 7713 processors, with a total of 128 cores and 440GB of memory per node. We set 6GB of memory limit per process, and set the cache limit to 3 GB for each counter. We executed each run on a separate physical CPU core with a timeout of 3600 seconds per model counter process, as per Model Counting Competition[3] rules. Preprocessors were ran before the Model Counting process, with the same memory constraints and a separate 1800s timeout.

We sought to answer the following questions:

(1) How does Arjun2 compare to prior state-of-the-art preprocessors in terms of the size of the preprocessed formula?
(2) How does Arjun2 compare to prior state-of-the-art in terms of its impact on the downstream model counters?
(3) What is the impact of sparsification, vivification, and redundant clause generation by Arjun2 on model counting?

Analysis of Preprocessed CNFs. Figure 1 illustrates the reduction of the number of variables (left) and clauses (right), using Arjun2 on the instances. Overall, we observe that Arjun2 reduces the median number of variables by 43.2% the median number of clauses by 86.9%, and the median number of literals by 85.7%.

Analysis of Impact on Model Counters. Table 1 summarizes the PAR-2 scores[4] of all counters with all preprocessors. A cell in row X and column Y of Table 1 corresponds to the PAR-2 score of $X+Y$: for example, the column at the top left corner represents PAR-2 score of Arjun2+GPMC. We make three observations: First, when comparing against "No preprocessing", we witness an improvement in PAR-2 score of more than 1000 for nearly all counters except ExactMC. It's worth noting that ExactMC has internal preprocessing which can not be turned off. Secondly, we observe that in comparison to other preprocessors, Arjun2 achieves improvement in the PAR-2 score for all counters except GANAK. Of particular note is the difference of nearly 100 points between SharpSAT-TD-Pre+ExactMC and Arjun2+ExactMC.

Figure 2 shows in detail the performance difference that different preprocessors have on the most well-performing, state-of-the-art model counter ExactMC. Firstly, this figure shows that Arjun2, SharpSAT-TD-Pre, and BiPe are in a different category than the other preprocessors. Secondly, it demonstrates that using Arjun2 is strongly preferred for fast counting.

Comparison with other Preprocessors. Figure 3 presents the runtime performance comparison of Arjun2 with other preprocessors. Notice that while Arjun2 is slower at the start than some preprocessors, it overtakes all but GPMC-Pre by 300s, and all by 500s timeout. It is worth noting that all preprocessors except Arjun2 either time- or memory-outed on some instances.

Impact of different configurations of Arjun2. We sought to also understand the effectiveness of different preprocessing techniques in Arjun2: given the large number of techniques integrated in Arjun2, we present results for configurations that have a significant impact.

[3]See https://mccompetition.org/
[4]PAR-2 score is the average runtime of a tool, with penalization of two times the allocated runtime in case of a time- or memory-out.

	GPMC	D4	GANAK	SharpSAT-TD	ExactMC
Arjun2	2174.9	2149.5	1963.7	1669.3	**1574.8**
SharpSAT-TD-Pre	2082.0	2386.7	1906.9	1919.0	1680.8
BiPe	2191.9	2145.6	2035.5	2028.5	1735.4
GPMC-Pre	2538.2	2496.0	2249.6	2072.0	1949.3
KCBox	2814.0	2894.0	2688.8	2643.2	2223.9
No preprocessing	2953.6	3174.9	3319.2	3010.4	2175.7

Table 1: PAR-2 score of all preprocessors on all model counters. Note that ExactMC's preprocessor cannot be turned off

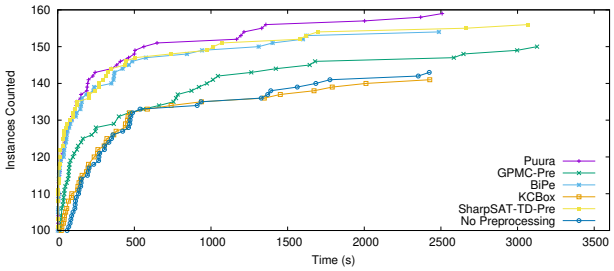

Figure 2: Runtime performance of ExactMC

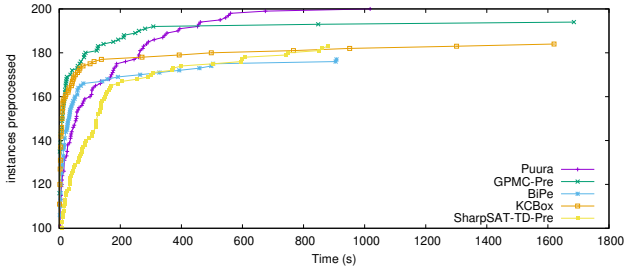

Figure 3: Runtime comparison of all preprocessors

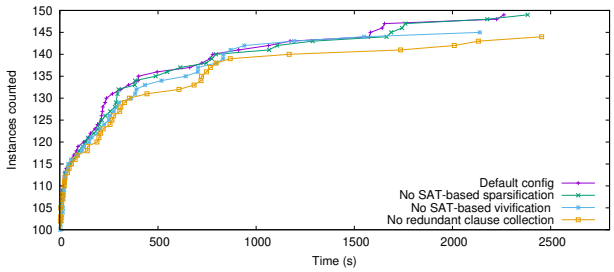

Figure 4: The effect of different configurations of Arjun2 on the GANAK model counter

To this end, we focus on understanding the effectiveness of vivification, sparsification, and redundant clause recycling. Since not all counters support our proposed format for redundant clauses, we demonstrate results for one of the counters, GANAK (our choice of GANAK was based purely on our familiarity of its code base). We plot the results for different configurations in Figure 4. The figure demonstrates that the default configuration, with all improvements turned on, performs the best. Interestingly, turning off redundant clause addition results in the most significant performance loss. This is likely because transitive reduction and sparsification remove clauses that are beneficial for propagation performance, even though they are redundant. Reintroducing them as redundant clauses enables the counter to use them for propagation while eliminating the need to verify whether they connect potentially disjoint components.

5 CONCLUSION

In this paper, we focus on the problem of algorithmic engineering of an efficient preprocessor for model counting, Arjun2. The design of Arjun2 is motivated by the observation that given the computational hardness of the problem of model counting, it is worthwhile to integrate a diverse array of techniques, even ones that require calls to SAT solvers and samplers. Our detailed empirical evaluation shows that Arjun2 improves the runtime performance of almost all the top-performing model counters. Furthermore, our empirical analysis demonstrating the usefulness of recycling of redundant clauses indicate that an interesting direction of future work would be to selectively reuse redundant clauses while avoiding the overhead on the component cache management.

Acknowledgment. The authors are grateful to Tuukka Korhonen and Matti Järvisalo for many helpful discussions. This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019- 0004], and Ministry of Education Singapore Tier 2 Grant [MOE- T2EP20121-0011]. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing.

REFERENCES

[1] Réka Albert, Bhaskar DasGupta, Riccardo Dondi, and Eduardo D. Sontag. 2006. Inferring (Biological) Signal Transduction Networks via Transitive Reductions of Directed Graphs. *Electron. Colloquium Comput. Complex.* TR06-010 (2006).
[2] Fahiem Bacchus and Jonathan Winter. 2003. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *Proc. of SAT*. 341–355.
[3] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S Meel, and Prateek Saxena. 2019. Quantitative verification of neural networks and its security applications. In *Proc. of CCS*. 1249–1264.
[4] Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2022. Proceedings of SAT Competition 2022 : Solver and Benchmark Descriptions. *Dept. of Computer Science Series, University of Helsinki* (2022).
[5] Armin Biere. [n. d.]. Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. In *Proc. of POS*.
[6] Armin Biere. 2008. PicoSAT Essentials. *J. Satisf. Boolean Model. Comput.* 4, 2-4 (2008), 75–97.
[7] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proc. of SAT Competition 2020*. University of Helsinki, 51–53.
[8] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. 2021. Preprocessing in SAT Solving. *Handbook of Satisfiability* 336 (2021), 391–435.
[9] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. 2021. Preprocessing in SAT Solving. In *Handbook of Satisfiability - Second Edition*. Vol. 336. 391–435.
[10] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426.
[11] Leonardo Duenas-Osorio, Kuldeep S Meel, Roger Paredes, and Moshe Y Vardi. 2017. Counting-Based Reliability Estimation for Power-Transmission Grids. In *Proc. of AAAI*.
[12] Niklas Eén and Armin Biere. 2005. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. of SAT*, Vol. 3569. 61–75.
[13] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proc. of SAT*. 502–518.
[14] William Eiers, Seemanta Saha, Tegan Brennan, and Tevfik Bultan. 2019. Subformula caching for model counting and quantitative program analysis. In *Proc. of ASE*. 453–464.
[15] Johannes K Fichte, Markus Hecher, and Florim Hamiti. 2021. The model counting competition 2020. *Journal of Experimental Algorithmics (JEA)* 26 (2021), 1–26.
[16] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. 2021. Designing Samplers is Easy: The Boon of Testers. In *Proc. of FMCAD*.
[17] Marijn Heule, Matti Järvisalo, and Armin Biere. 2013. Revisiting Hyper Binary Resolution. In *Proc. of CPAIOR*, Vol. 7874. 77–93.
[18] Mikolás Janota, Inês Lynce, and João Marques-Silva. 2015. Algorithms for computing backbones of propositional formulae. *AI Commun.* 28, 2 (2015), 161–177.
[19] Matti Järvisalo, Armin Biere, and Marijn Heule. 2010. Blocked Clause Elimination. In *Proc. of TACAS (Lecture Notes in Computer Science, Vol. 6015)*. Springer, 129–144.
[20] Matti Järvisalo, Marijn Heule, and Armin Biere. 2012. Inprocessing Rules. In *Proc. of IJCAR (Lecture Notes in Computer Science, Vol. 7364)*. 355–370.
[21] Tuukka Korhonen and Matti Järvisalo. 2021. Integrating tree decompositions into decision heuristics of propositional model counters. In *Proc. of CP*.
[22] Tuukka Korhonen and Matti Järvisalo. 2023. SharpSAT-TD in Model Counting Competitions 2021-2023. *arXiv e-prints* (Aug. 2023). arXiv:2308.15819
[23] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. 2016. Improving Model Counting by Leveraging Definability. In *Proc. of IJCAI*. 751–757.
[24] Jean-Marie Lagniez and Pierre Marquis. 2014. Preprocessing for propositional model counting. In *Proc. of AAAI*, Vol. 28.
[25] Yong Lai, Dayou Liu, and Minghao Yin. 2017. New Canonical Representations by Augmenting OBDDs with Conjunctive Decomposition. *J. Artif. Intell. Res.* 58 (2017), 453–521.
[26] Yong Lai, Kuldeep S Meel, and Roland HC Yap. 2021. The power of literal equivalence in model counting. In *Proc. of AAAI*. 3851–3859.
[27] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. [n. d.]. Recovering and Exploiting Structural Knowledge from CNF Formulas. In *CP 2002*.
[28] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. 2008. Vivifying Propositional Clausal Formulae. In *ECAI 2008*.
[29] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proc. of IJCAI*. 1169–1176.
[30] Mary Sheeran and Gunnar Stålmarck. 1998. A Tutorial on Stålmarcks's Proof Procedure for Propositional Logic. In *Proc. of FMCAD*, Vol. 1522. 82–99.
[31] Mate Soos and Kuldeep S Meel. 2019. BIRD: Engineering an Efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proc. of AAAI*.
[32] Mate Soos and Kuldeep S. Meel. 2022. Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling. In *Proc. of ICCAD*. 71.
[33] Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. 2017. *Improvement of projected model-counting solver with component decomposition using SAT solving in components*. Technical Report. SIG-FPAI-506-07.