BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting*

Mate Soos and **Kuldeep S. Meel**
School of Computing
National University of Singapore

Abstract

Given a Boolean formula ϕ, the problem of model counting, also referred to as #SAT is to compute the number of solutions of ϕ. Model counting is a fundamental problem in artificial intelligence with a wide range of applications including probabilistic reasoning, decision making under uncertainty, quantified information flow, and the like. Motivated by the success of SAT solvers, there has been surge of interest in the design of hashing-based techniques for approximate model counting for the past decade. We profiled the state of the art approximate model counter ApproxMC2 and observed that over 99.99% of time is consumed by the underlying SAT solver, CryptoMiniSat. This observation motivated us to ask: *Can we design an efficient underlying CNF-XOR SAT solver that can take advantage of the structure of hashing-based algorithms and would this lead to an efficient approximate model counter?*

The primary contribution of this paper is an affirmative answer to the above question. We present a novel architecture, called BIRD, to handle CNF-XOR formulas arising from hashing-based techniques. The resulting hashing-based approximate model counter, called ApproxMC3, employs the BIRD framework in its underlying SAT solver, CryptoMiniSat. To the best of our knowledge, we conducted the most comprehensive study of evaluation performance of counting algorithms involving 1896 benchmarks with computational effort totaling 86400 computational hours. Our experimental evaluation demonstrates significant runtime performance improvement for ApproxMC3 over ApproxMC2. In particular, we solve 648 benchmarks more than ApproxMC2, the state of the art approximate model counter and for all the formulas where both ApproxMC2 and ApproxMC3 did not timeout and took more than 1 seconds, the mean speedup is 284.40 – more than two orders of magnitude.

1 Introduction

Propositional model counting is a fundamental problem in artificial intelligence with a wide range of applications ranging from probabilistic reasoning (Roth 1996), network reliability (Dueñas-Osorio et al. 2017), decision making under uncertainty (Gomes, Sabharwal, and Selman 2009), quantified information leakage (Biondi et al. 2018) and the like (Roth 1996; Gomes, Sabharwal, and Selman 2009) Given a Boolean formula ϕ, the problem of propositional model counting, also referred to as #SAT, is to compute the number of solutions of ϕ. In his seminal paper, Valiant showed that #SAT is $\#\mathsf{P}$-complete, where $\#\mathsf{P}$ is the set of counting problems associated with NP decision problems (Valiant 1979).

Theoretical investigations of $\#\mathsf{P}$ have led to the discovery of deep connections in complexity theory, and there is strong evidence for its hardness (Arora and Barak 2009; Toda 1989). In particular, Toda showed that every problem in the polynomial hierarchy could be solved by just one call to a $\#\mathsf{P}$ oracle; more formally, $PH \subseteq P^{\#\mathsf{P}}$ (Toda 1989).

Given computational intractability of #SAT, attention has focused on approximation of #SAT. In a breakthrough, Stockmeyer provided a hashing-based randomized approximation scheme for counting that makes polynomially many invocations of NP oracle (Stockmeyer 1983). The procedure, however, was computationally prohibitive in practice at that time and no practical tools existed based on Stockmeyer's proposed algorithmic framework until the early 2000s (Gomes, Sabharwal, and Selman 2009). Motivated by the success of SAT solvers, there has been surge of interest in the design of hashing-based techniques for approximate model counting for the past decade (Gomes, Sabharwal, and Selman 2006; Chakraborty, Meel, and Vardi 2013b; Ermon et al. 2013; Chakraborty, Meel, and Vardi 2016; Meel et al. 2016; Meel 2017; Achlioptas, Hammoudeh, and Theodoropoulos 2018).

The core idea of hashing-based framework is to employ $2-$universal hash functions to partition the solution space into *roughly equal small* cells, wherein a cell is called *small* if it has less than or equal to thresh solutions, such that thresh is a function of ϵ. A SAT solver is employed to check if a cell is small by enumerating solutions one-by-one until either there are no more solutions or we have already enumerated $\mathrm{thresh}+1$ solutions. Following the terminology of (Chakraborty, Meel, and Vardi 2013b; 2016), we refer to the above described procedure as BSAT. To determine the number of cells, the frameworks such as ApproxMC2 perform a search that requires $\mathcal{O}(\log \mathsf{n})$ steps and the estimate is returned as the count of the solutions in a randomly picked small cell scaled by the total number of cells. To amplify confidence to the desired levels

*The open-source implementation along with accompanied technical report is available at `https://github.com/meelgroup/approxmc`

of $1-\delta$, the hashing-based algorithms invoke the estimation routine $\mathcal{O}(\log \frac{1}{\delta})$ times and reports the median of all such estimates. Hence, the number of BSAT invocations is $\mathcal{O}(\log \mathsf{n} \log(\frac{1}{\delta}))$. Over the past decade, the surge of interest in hashing-based techniques has led to several important significant advances: such as reduction of BSAT calls from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ (Chakraborty, Meel, and Vardi 2016), usage of sparser XORs (Zhao et al. 2016; Ivrii et al. 2015; Achlioptas and Theodoropoulos 2017), projection of counting over a smaller variable set (Aziz et al. 2015; Ivrii et al. 2015) and the like. While the state of art hashing-based counters such as ApproxMC2 can handle formulas involving hundreds of thousands of variables, there is still a large gap between theory and practice.

We profiled the state of the art approximate model counter, ApproxMC2 and observed that over 99.99% of time is consumed by BSAT. Since BSAT is invoked with a CNF formula conjuncted with random XOR constraints, prior work on hashing-based techniques have advocated usage of CryptoMiniSat, an efficient SAT solver designed to handle CNF-XOR formulas. Inspired by the success of SMT solvers, Soos et al (Soos, Nohl, and Castelluccia 2009) proposed an elegant architecture for CryptoMiniSat that keeps CNF and XOR clauses separately. The distributed storage allows CryptoMiniSat to apply Gauss-Jordan elimination on XOR clauses.

The distributed storage, however, comes at the cost of disabling execution of inprocessing steps, e.g., bounded variable elimination (Eén and Biere), on variables that are part of the XOR clauses. A careful reader might observe that the usage of distributed storage does not necessarily imply unsoundness of inprocessing steps but a sound implementation of inprocessing steps would require extensive study into effect of every inprocessing step on XOR clauses. Given the complexity of inprocessing implementations in the current state of the art SAT solvers, the odds of success of an efficient implementation following extensive study into the effect of inprocessing steps on XORs are very high. Since ApproxMC2 relies on the usage of random XORs, the disabling of pre- and inprocessing steps essentially restricts most of the pre- and inprocessing steps during the execution of BSAT. The lack of usage of pre- and inprocessing steps significantly hurts the performance of BSAT since these techniques have been shown to be crucial to the performance of state of the art SAT solvers. Furthermore, division of storage of CNF and XOR clauses induces significant overhead in synchronization of the state of the solve in CNF and XOR.

The asymmetry in the design and usage of CryptoMiniSat motivates us to ask: *Can we design an efficient approximate model counter where the underlying* SAT *solver can take advantage of the structure of hashing-based algorithms?* The primary contribution of this paper is an affirmative answer to the above question. We present a novel architecture, called BIRD, to handle CNF-XOR formulas arising from hashing-based techniques. The resulting hashing-based approximate model counter, called ApproxMC3, employs the BIRD framework in its underlying SAT solver, CryptoMiniSat. To the best of our knowledge, we conducted the most comprehensive study of evaluation performance of counting algorithms involving 1896 benchmarks with computational effort totaling 86400 computational hours. With a timeout of 5000 seconds, the state of the art exact model counter, DSharp and the state of the art approximate model counter, ApproxMC2 were able to solve only 1001 and 492 benchmarks while ApproxMC3 could solve 1140 benchmarks. Significantly, this marks the first time that an approximate model counter is able to not only beat exact counter for challenging problems but also outperform the exact counter for the entire set of benchmarks. In particular, ApproxMC3 solves 648 benchmarks more than ApproxMC2, the state of the art approximate model counter. Furthermore, for all the formulas where both ApproxMC2 and ApproxMC3 did not timeout and took more than 1 seconds, the mean speedup is 284.40 – more than 2 orders of magnitude.

The rest of the paper is organized as follows: We discuss notations and preliminaries in Section 2 and then focus on core technical features of ApproxMC3 in Section 3. We then present an extensive experimental evaluation in Section 4 and finally conclude in Section 5.

2 Notations and Preliminaries

Let F be a Boolean formula in conjunctive normal form (CNF), and let $\mathsf{Vars}(F)$ be the set of variables appearing in F. The set $\mathsf{Vars}(F)$ is also called the *support* of F. An assignment σ of truth values to the variables in $\mathsf{Vars}(F)$ is called a *satisfying assignment* or *witness* of F if it makes F evaluate to true. We denote the set of all witnesses of F by R_F. Given a set of variables $S \subseteq \mathsf{Vars}(F)$, we use $R_{F\downarrow S}$ to denote the projection of R_F on S.

We write $\mathsf{Pr}\left[X : \mathcal{P}\right]$ to denote the probability of outcome X when sampling from a probability space $\mathcal{P}$. For brevity, we omit $\mathcal{P}$ when it is clear from the context. The expected value of X is denoted $\mathsf{E}\left[X\right]$ and its variance is denoted $\mathsf{V}\left[X\right]$.

The *propositional model counting problem* is to compute $|R_{F\downarrow S}|$ for a given CNF formula F and sampling set $S \subseteq \mathsf{Vars}(F)$. A *probably approximately correct* (or PAC) counter is a probabilistic algorithm $\mathsf{ApproxCount}(\cdot,\cdot,\cdot,\cdot)$ that takes as inputs a formula F, a sampling set S, a tolerance $\varepsilon > 0$, and a confidence $1-\delta \in (0,1]$, and returns a count c such that $\mathsf{Pr}\Big[|R_{F\downarrow S}|/(1+\varepsilon) \leq c \leq (1+\varepsilon)|R_{F\downarrow S}|\Big] \geq 1-\delta$.

For positive integers n and m, a special family of 2-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, called $H_{xor}(n,m)$, plays a crucial role in our work. Let $y[i]$ denote the i^{th} component of a vector y. The family $H_{xor}(n,m)$ can then be defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,k} \in \{0,1\}, 1 \leq i \leq m, 0 \leq k \leq n\}$, where $\oplus$ denotes "XOR" and $\cdot$ denotes "and". By choosing values of $a_{i,k}$ randomly and independently, we can effectively choose a random hash function from $H_{xor}(n,m)$.

In (Chakraborty, Meel, and Vardi 2013b; Meel 2014), a new hashing-based strongly probably approximately correct counting algorithm, called ApproxMC, was shown to scale to formulas with thousands of variables, while providing rigorous PAC-style (ε,δ) guarantees. The core idea of ApproxMC is to use 2-universal hash functions to randomly partition the solution space of the original formula into "small" enough cells. Overall, ApproxMC makes a total of $\mathcal{O}(\frac{n\log(1/\delta)}{\varepsilon^2})$ calls

to CryptoMiniSat. Significantly, and unlike the algorithm in (Goldreich 1999), each call of CryptoMiniSat reasons about a formula with only n variables.

It is well-known that long XOR-based constraints make SAT solving significantly harder in practice (Gomes et al. 2007). Researchers have therefore investigated theoretical and practical aspects of using short XORs (Gomes et al. 2007; Chakraborty, Meel, and Vardi 2014; Zhao et al. 2016). The techniques for identifying small independent supports have been developed (Ivrii et al. 2015), and word-level hash functions have been used to count in the theory of bit-vectors (Chakraborty et al. 2016). A common aspect of all of these approaches was that a linear search was used to find the right parameters of the hash functions, where each search step involves multiple SAT solver calls. ApproxMC2 targeted this weak link and shown search for parameters can be searched in logarithmic time by relying on dependence of hash functions. To the best of our knowledge, ApproxMC2 is currently the state of the art approximate model counter. ApproxMC3 builds on top of ApproxMC2 by modifying the underlying SAT solver used by ApproxMC2. The underlying SAT solver is invoked through subroutine BSAT, which is implemented using CryptoMiniSat. Formally, BSAT takes as inputs a formula F, a threshold thresh, and a sampling set S, and returns a subset Y of $R_{F\downarrow S}$, such that $|Y| = \min(\mathrm{thresh}, |R_{F\downarrow S}|)$. Note that every invocation of BSAT is performed with F conjuncted with set of XOR constraints. We henceforth denote such formulas as CNF-XOR formulas. As mentioned earlier, our profiling of ApproxMC2 revealed that over 99% of the total time is taken by BSAT calls inside LogSATSearch. In the next section, we discuss a novel architecture of CryptoMiniSat that allows us to perform BSAT efficiently.

For lack of space, we refer the reader to (Chakraborty, Meel, and Vardi 2016) for description of LogSATSearch that invokes BSAT.

3 BIRD: A New Framework for Handling CNF-XOR Formulas

In this section, we discuss the primary contribution of this paper: a novel architecture for BSAT to efficiently handle CNF-XOR formulas. Before delving into the technical details of the architecture, we first review the architecture of BSAT employed in ApproxMC2.

3.1 Architecture of BSAT in ApproxMC2

Given extensive reliance on XORs for partitioning of the solution space, BSAT requires underlying SAT solver to have native support for XORs (Chakraborty, Meel, and Vardi 2013a). Therefore, ApproxMC2 employs CryptoMiniSat with native support for XORs. Inspired by the success of SMT solvers, Soos et al (Soos, Nohl, and Castelluccia 2009) proposed an elegant architecture for CryptoMiniSat that keeps CNF and XOR clauses separately. The distributed storage allows CryptoMiniSat to apply Gauss-Jordan elimination on XOR clauses.

The distributed storage, however, comes at the cost of disabling execution of inprocessing steps, e.g., bounded variable elimination (Eén and Biere), on variables that are part of the XOR clauses. A careful reader might observe that the usage of distributed storage does not necessarily imply unsoundness of inprocessing steps but a sound implementation of inprocessing steps would require extensive study into effect of every inprocessing step on XOR clauses. Given the complexity of inprocessing implementations in the current state of the art SAT solvers, the odds of success of an efficient implementation following extensive study into the effect of inprocessing steps on XORs are very high. Since ApproxMC2 employs random XORs over the sampling set S and we are interested in solutions projected over the sampling set S, the disabling of pre- and inprocessing steps essentially restricts most of the pre- and inprocessing steps during the execution of BSAT. The lack of usage of pre- and inprocessing steps significantly hurts the performance of BSAT since these techniques have been shown to be crucial to the performance of state of the art SAT solvers. Furthermore, division of storage of CNF and XOR clauses induces significant overhead in synchronization of the state of the solve in CNF and XOR. We target this weak link in this paper, and drastically improve the runtime of ApproxMC3 by redesigning the architecture of BSAT.

Algorithm 1 ComputeBloom

```
abst ← 0
for var in clause do
    abst ← abst | (var % 32)
return abst
```

3.2 BIRD: Blast, In-processing, Recover, and Destroy

To allow seamless integration of pre- and inprocessing techniques, it is important that the solver has access to XOR clauses in CNF form while ensuring native support for XORs to perform Gauss-Jordan elimination. We achieve this by our architecture, BIRD: Blast-Recover-Blast, described as follows:

BIRD: Blast, In-process, Recover, and Destroy

Step 1 **Blast** XOR clauses into normal CNF clauses

Step 2 **Inprocess** (and pre-process) over CNF clauses

Step 3 **Recover** simplified XOR clauses

Step 4 Perform CDCL on CNF clauses with on-the-fly Gauss-Jordan Elimination on XOR clauses until inprocessing is scheduled

Step 5 **Destroy** XOR clauses and goto **Step 2**

Note that we exit the above loop as soon as find a satisfying assignment or prove that the formula is UNSAT. The BIRD architecture allows *all* current or future techniques during inprocessing. Furthermore, as the benchmarks arising from circuits typically contain XOR clauses encoded in CNF, BIRD can efficiently recover such XORs and therefore, allow-

ing the usage of Gauss-Jordan elimination on such recovered XORs.

In comparison to BSAT in ApproxMC2, the primary challenge for BIRD is to ensure that **Step 1** and **Step 3** can be executed efficiently. Note that LogSATSearch invokes BSAT over the original formula F conjuncted with XORs until the number of solutions is less than thresh when the number of solutions of F is typically of the order of 2^{60}. Therefore, XOR constraints play a significant part in determining the solution space of the formula that BSAT takes as input. Consequently, we focus on efficient blasting of XORs into CNF in **Step 1** and efficient and full recovery of XORs in **Step 3** as described below:

3.3 Blasting of XORs to CNF

We employ the standard technique of blasting XORs into CNF. Observe that a XOR over k variables can be equivalently represented as CNF over k variables and $\mathcal{O}(2^k)$ clauses. Since we deal with long XORs, typically of size $|S|/2$, we first cut a long XORs into smaller XORs by introducing auxiliary variables. The size of small XORs is known as *cutting number*. We experimented with different cutting numbers and found that *cutting number = 4* is optimal for our use case.

3.4 XOR recovery

We now discuss the most technically challenging task of our BIRD architecture: recovery of XOR constraints from CNF clauses (i.e. **Step 3**). Formally, given a formula F in CNF, we would like to extract H, which is expressed as conjunction of XOR constraints such that $F \rightarrow H$. To put our contribution in context, we briefly review the prior work. Heule proposed the current state of the art algorithm, referred to as HeuleRecovery for XOR recovery in his PhD thesis, which is based on the observation that an XOR of size k is equivalently represented by 2^{k-1} CNF clauses. For example:

$$\begin{array}{r}(x_1 \vee x_2 \vee \neg x_3)\\(x_1 \vee \neg x_2 \vee x_3)\\(\neg x_1 \vee x_2 \vee x_3)\\(\neg x_1 \vee \neg x_2 \vee \neg x_3)\end{array} \Leftrightarrow x_1 \oplus x_2 \oplus x_3 = 0 \qquad (1)$$

Observe that if we only concentrate on the sign of a variable and represent a positive variable as 0 and the negative variable as 1, we see that the above set of clauses over x_1, x_2, x_3 can be represented by strings $\{001, 010, 100, 111\}$, whose decimal representation reads as $\{1, 3, 5, 7\}$. Similarly if we had CNF clauses encoding the XOR constraint, $x_1 \oplus x_2 \oplus x_3 = 1$, we would have decimal representation of the set of clauses as $\{0, 2, 4, 8\}$. Based on this observation, HeuleRecovery proceeds by sorting clauses by the sets of variables occurring in a clause and stores the decimal representation of all the combinations and accordingly recovers XORs. Executing HeuleRecovery is efficient in practice; recovery of XOR constraints takes only a few seconds for formulas with hundreds of thousands of clauses.

While HeuleRecovery is efficient in practice, it is not robust to modification of clauses due to **Step 2** and **Step 4** of BIRD. Consequently, HeuleRecovery fails to extract many of the XORs during the search procedure. The primary drawback of HeuleRecovery is its reliance on presence of all the equivalent CNF clauses. Let us assume XOR clause $x_1 \oplus x_2 \oplus x_3 = 0$ was blasted to the four clauses shown in Eq 1 in **Step 1**. Now let us assume that during **Step 2** and **Step 4**, the SAT solver learns the clause $x_1 \vee x_2$, which subsumes (i.e., replaces), the first clause in Eq 1: $(x_1 \vee x_2 \vee \neg x_3)$ Furthermore, another frequently used, self-subsuming resolution, would resolve $(x_1 \vee x_2)$ with $(x_1 \vee \neg x_2 \vee x_3)$ to obtain $(x_1 \vee x_3)$, which subsumes $(x_1 \vee \neg x_2 \vee x_3)$. Therefore, the four clauses in Eq 1 are reduced to the following set of clauses, denoted by G. $G := (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ Note that $G \rightarrow (x_1 \oplus x_2 \oplus x_3 = 0)$. Therefore, we would like to recover $(x_1 \oplus x_2 \oplus x_3 = 0)$. The above illustrated sequence of steps happen often and repeatedly during pre- and inprocessing of CNF formulas. Hence, a XOR recovery algorithm must be able to find any such mangled XOR from a reasonably sized CNF with hundreds of thousands of clauses.

Algorithm 2 Barbet(clauses, M)

```
1: xorclauses ← ∅
2: for base_cl ∈ clauses do
3:     if base_cl.size > M then continue
4:     if base_cl.used == 1 then continue
5:     FIND_ONE_XOR(base_cl)
   return xorclauses
```

Algorithm 3 Find XOR using base clause

```
1: function FindOneXOR(base_cl)
2:     quickcheck ← array of zeroes
3:     found_comb ← array of zeroes
4:     comb ← 0
5:     base_rhs ← 1            ▷ right-hand-side of the XOR
6:     for i ← 0 . . . base_cl size-1 do
7:         base_rhs ← base_rhs ⊕ base_cl[i].sign
8:         comb ← comb | (base_cl[i].sign << i)
9:         quickcheck[base_cl[i].var] ← 1
10:    base_abst ← CALC_ABST(base_cl)
11:    found_comb[comb] ← 1
12:    for v ∈ Vars(base_cl) do
13:        for abst, cl ∈ occurrence[v] do
14:            if CheckClause(abst, cl, base_cl, base_abst)
   then return
```

Barbet

We now describe our recovery algorithm Barbet presented in Algorithm 2. Barbet takes in a set of clauses and a parameter M and returns recovered XOR clauses of size $\leq M$. First, Barbet initializes an empty set xorclauses, to store the recovered XOR clauses. Then it makes a linear pass through all the clauses. If a clause is of size $> M$ or has already been used in construction of a XOR, we skip the clause. Otherwise, we invoke the subroutine FindOneXOR, which searches for recovery of an XOR clause from the set of clauses containing base_cl clause.

The algorithm FindOneXORis presented in Algorithm 3.

FindOneXORtakes in a clause, base_cl, and attempts to recover the XOR defined exactly over the variables in base_cl. Note that there is exactly one XOR defined exactly over variables in base_cl. For example, if base_cl is $(x_1 \vee x_2 \vee x_3)$, then the only XOR defined over x_1, x_2, x_3 in which base_cl can participate is $x_1 \oplus x_2 \oplus x_3 = 1$. Observe that we can compute the rhs of the XOR by computing the parity of the variables in the XOR. The key idea behind the search for XORs over variables in base_cl, say of size M, to perform a linear pass (in an efficient manner as detailed below) and check whether there exists a subset of clauses that would imply the required 2^{M-1} combination of CNF clauses over M variables.

Note that a clause may imply multiple CNF clauses over M variables. For example, let base_cl $:= (x_1 \vee x_2 \vee x_3)$, then a clause $cl := (x_1)$ would imply 4 clauses over $\{x_1, x_2, x_3\}$, i.e. $\{(x_1 \vee x_2 \vee \neg x_3), (x_1 \vee x_2 \vee x_3), (x_1 \vee \neg x_2 \vee x_3), (x_1 \vee \neg x_2 \vee \neg x_3)\}$. To this end, we maintain an array of possible combinations, denoted by foundcomb, of size 2^M and update the entry (indexed by comb, which is binary representation of the clause for a fixed ordering of variables) corresponding a clause cl' to 1 if $cl \rightarrow cl'$. Similar to other aspects of SAT solving, efficient data structures are vital to perform the above mentioned checks and updates efficiently. We now discuss few crucial optimizations below in the description of FindOneXOR:

FindOneXORassumes access to the map data structure, occurrence, such that for a variable x, occurrence$[x]$ is a list of all the clauses that contain v. Every entry of the list consists of pointer to the clause and the bitfield abst, which is a 32-bit bloom filter of the variables inside the clause. The algorithm to compute abst is presented in Algorithm 1.

FindOneXORmaintains a map of all the variables in base_cl in the data structure, quickcheck. FindOneXOR populates base_rhs, comb, quickcheck in lines 6– 9. Then we compute the bloom filter corresponding to base_cl in line 10. Next, we make a linear pass over all the clauses that contain a variable in base_cl and check for each clause cl if cl implies one of the combinations required for XOR corresponding to base_cl. The check is performed in the subroutine CheckClause.

CheckClause first checks if the variables in cl are subset of the variables in base_cl because for cl to imply one of the combinations required for XOR defined over variables in base_abst, variables in cl should be subset of variables in base_cl. We use bloom filter to perform this check by determining whether $abst|$base_abst $==$ base_abst holds. Note that by properties of bloom filter may have false positives but not false negatives. To check for false positive, we iterate through the clause to see whether every variable in cl is also present in base_cl. Note that performing this check requires dereferencing the pointer to the clause, therefore the bloom filter check is crucial. In our experiments the usage of the bloom filter led to up to 100 fold speedup on large instances for CheckClause calls. We now perform two more checks to see if cl is useful: first the rhs for XOR corresponding to cl should match with $base\_rhs$ if the sizes of both cl and base_cl are same. Finally, we update the foundcomb in the subroutine AddClause. The subroutine XORFound checks

Algorithm 4 Check if clause belongs to an XOR

```
1:  function CheckClause(abst, cl, base_cl, base_abst)
2:      if abst | base_abst != base_abst then
3:          return False                ▷ Bloom filter check fails
4:      rhs ← 1
5:      for lit ∈ cl do
6:          if quickcheck[lit.var] == 0 then
7:              return False      ▷ Bloom filter false positive
8:          rhs ← rhs ⊕ lit.sign
9:      if cl.used == 1 then
10:         return False
11:     if cl.size == base_cl.size and rhs != base_rhs then
12:         return False
13:     if cl.size == base_cl.size then cl.used ← 1
14:     AddClause(cl)
15:     if XORFound(base_rhs) then
16:         xorclauses.append(base_cl)
17:         return True
18:     return False
```

Algorithm 5 Add a clause to an XOR

```
1:  procedure AddClause(cl)
2:      basei, i, x ← 0
3:      missing ← clear
4:      for k = 0; k < cl.size; k++, i++, basei++ do
5:          while cl[i].var != base_cl[basei].var do
6:              missing.append(basei);
7:              basei ← basei + 1
8:          assert i == basei
9:          x ← x | (cl[k].sign ≪ basei)
10:     ▷ Mark every combination for the missing variables
11:     for j = 0; j < 1≪(missing.size); j++ do
12:         tx ← x
13:         for i2 = 0; i2 < missing.size; i2++ do
14:             if bit_set(j, i2) then
15:                 tx ← tx + (1≪missing[i2])
16:         found_comb[tx] ← 1
17: end procedure
```

if all the combinations required for XOR corresponding to base_cl have been found and if true, it adds base_cl to the list xorclauses. Recall that base_cl uniquely represents the XOR defined over all the variables in base_cl.

The algorithm AddClause first finds all the missing variables from the clause cl and computes all the combinations by taking account missing variables and updates the vector foundcomb. For example, if base_cl is $(x_1 \vee \neg x_2 \vee x_3)$ and cl is $(x_1 \vee \neg x_3)$, AddClause finds that x_2 is missing and then adds the combinations corresponding to clauses $(x_1 \vee x_2 \vee \neg x_3)$ and $(x_1 \vee \neg x_2 \vee \neg x_3)$. For efficiency, we perform a clever usage of bitfields as illustrated in the Algorithm 5.

The algorithm XORFound, presented in Algorithm 6, checks if all the possible combinations are present, which requires us to check if a given combination is required. Given base_rhs, we can check if a combination is required by check-

Algorithm 6 Check if XOR has been built

```
1: function XORFound(base_rhs)
2:     for i ∈ 0 . . . found_comb.size-1 do
3:         ▷ Only check combinations with the correct right-hand-side
4:         if (hamming_weight(i)%2) == base_rhs then
5:             continue
6:         if found_comb[i] == 0 then
7:             return False                ▷ XOR not complete
8:     return True                         ▷ Every combination found
```

ing the hamming weight of the bitfield. If base_rhs is 0, then the hamming weight[1] should be 0 otherwise it should be 1. If any of the required combination is missing, XORFound returns False, otherwise it returns True.

3.5 XOR Reconstruction

The XOR constraints recovered by Barbet are of length at most M, which is set to 5 in our experiments. Recall, we set the *cutting number* to 4 during blasting of XORs to CNF. Therefore, Barbet is able to recover the smaller XORs obtained during the blasting phase. Since the original XOR constraints added by ApproxMC3 are of length $|S|/2$, we now discuss the reconstruction of original XOR constraints so as to optimize the performance of Gauss-Jordan elimination. Similar to Barbet, we create a mapping from variables to the list of the XOR clauses that a variable is present. For each variable x, whose XOR-occurrence list is exactly of size 2 and where the XORs overlap exactly over this variable, we XOR together the two XOR clauses corresponding to the variable x to eliminate x and update the occurrence lists accordingly. These steps are performed until fixedpoint. We illustrate the technique with an example:

$$\left.\begin{array}{l} x_1 \oplus x_2 \oplus x_8 = 0 \\ x_8 \oplus x_3 \oplus x_4 \oplus x_9 = 0 \\ x_9 \oplus x_5 \oplus x_6 = 1 \end{array}\right. \rightarrow x_1 \oplus x_2 \ldots \oplus x_6 = 1$$

3.6 Correctness

Note that our reconstruction process may not be able to reconstruct all possible XOR operations. This, however, does not affect soundness and completeness of the SAT solving as CNF representation of all the constraints is still retained at CNF level. Therefore, one can view **Step 3** and **Step 4** of BIRD as *best effort* processes where we focus on helping CDCL as best as possible to deal with XOR clauses without having to be absolutely certain not to loose any information. This freedom is a property of the blasting-recovery method used by the system and greatly contributes to significant speedup achieved by our system as outline in the next section.

3.7 ApproxMC3

We augment ApproxMC2 with BIRD framework and call the resulting hashing-based counter as ApproxMC3. Given the soundness of BIRD, both ApproxMC2 and ApproxMC3 provide exactly same theoretical guarantees.

[1]Note that hamming weights are extremely fast to compute on modern CPUs thanks to dedicated assembly instructions

Benchmark	Vars	Clauses	ApproxMC2 time	ApproxMC3 time
or-50-10-9-UC-30	100	260	1814.78	2.66
blasted_squaring28	1060	3839	1845.47	2.48
55.sk_3_46	3128	12145	TO	1.35
s838_7_4	616	1490	TO	3.91
min-3s	431	1373	TO	3.86
blasted_case210	872	2937	TO	5.93
blasted_squaring16	1627	5835	TO	11.12
or-60-5-2-UC-30	120	315	TO	11.09
s5378a_3_2	3679	8372	TO	68.2
modexp8-4-1	79409	288110	TO	255.05
reverse.sk_ 11_ 258	75641	380869	TO	23.4
hash-6	282521	1133816	TO	246.04
modexp8-5-8	101553	402654	TO	1166.54
herman31.pm_20steps_6-int_1fract_stable_over	309496	159495	TO	1422.23
ConcreteRoleAffectation-Service.sk_119_273	395951	1520924	TO	2081.38
hash-11-8	518009	2078959	TO	4908.15
karatsuba.sk_7_41	19594	82417	TO	4865.53
01B-3	23393	103379	TO	4275.08

Table 1: Runtime performance comparison of ApproxMC3 vis-a-vis ApproxMC2. TO indicates timeout after 5000 seconds.

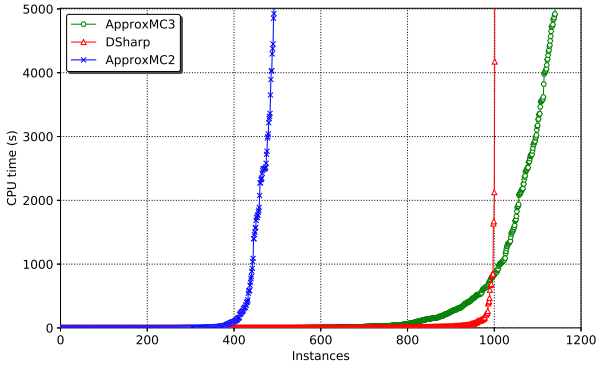

Figure 1: Cactus plot showing behavior of ApproxMC2, DSharp, and ApproxMC3

4 Experimental Evaluation

To evaluate the runtime performance and quality of approximations computed by ApproxMC3, we conducted the most comprehensive study of performance evaluation of counting algorithms involving 1896 benchmarks arising from wide range of application areas including probabilistic reasoning, plan recognition, DQMR networks, ISCAS89 combinatorial circuits, quantified information flow, program synthesis, functional synthesis, logistics, and the like as have been previously employed in studies on model counting (Chakraborty, Meel, and Vardi 2016; Lagniez and Marquis 2017). For the

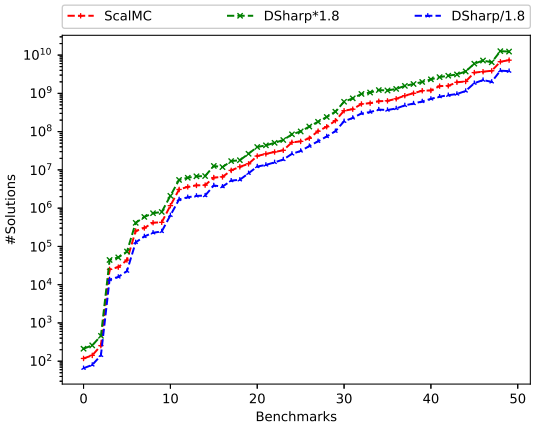

Figure 2: Plot showing counts obtained by ApproxMC2 vis-a-vis true counts from DSharp

sake of space, we discuss results for only a subset of these benchmarks here. The complete set of experimental results can be found at `https://github.com/meelgroup/approxmc`.

The objective of our experimental evaluation was to answer the following questions:

1. How does runtime performance of ApproxMC3 compare with that of ApproxMC2 and other state of the art counting techniques?
2. How far are the counts computed by ApproxMC3 from the exact counts.

The core difference between ApproxMC2 and ApproxMC3 is optimization of BSAT, therefore a fair comparison would be to compare ApproxMC3 vis-a-vis ApproxMC2. Such a comparison would provide a clear picture of progress achieved by ApproxMC3 but would still not provide a thorough analysis of the state of the art hashing-based techniques in comparison to other techniques. To this end, we also perform a comparison with the state of the art exact count technique DSharp which can handle projection over a sampling set. [2]

The experiments were conducted on high performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM. For all our experiments, we used $\varepsilon = 0.8$ and $\delta = 0.1$, unless stated otherwise. We used timeout of 5000 seconds for each experiment, which consisted of running a tool on particular benchmark. To further optimize the running time for both ApproxMC2 and ApproxMC3, we used improved estimates of the iteration count t following an analysis similar to that in (Chakraborty, Meel, and Vardi 2016).

[2] A curious reader might wonder whether DSharp achieves projection over sampling set at the cost of runtime performance. Instead, our analysis show DSharp is able to solve 100 more benchmarks that sharpSAT, which is unable to perform projection over sampling set

4.1 Results

Figure 1 shows the cactus plot for ApproxMC2 and ApproxMC3. We present the number of benchmarks on $x-$axis and the time taken on $y-$axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve. With a timeout of 5000 seconds, DSharp and ApproxMC2 were able to solve only 1001 and 492 benchmarks while ApproxMC3 could solve 1140 benchmarks. During the recent surge of interest in the development of hashing-based techniques, the community have consistently observed that exact counters are able to solve more instances than hashing-based schemes but the strength of hashing-based schemes lied in their ability to solve instances that were beyond the reach of exact counters. In this context, we believe ApproxMC3 has achieved an important milestone in being able to solve more instances than the state of the art exact counters. Note that ApproxMC3 retains the same theoretical guarantees of ApproxMC2. In particular, the performance improvement is solely due to improvement of the underlying SAT solver.

To present a clear picture of performance gain achieved by ApproxMC3 over ApproxMC2, we present runtime comparison of ApproxMC3 vis-a-vis ApproxMC2 in Table 1 on a subset of our benchmarks. Column 1 of this tables gives the benchmark name, while columns 2 and 3 list the number of variables and clauses, respectively. Columns 4 and 5 list the runtime (in seconds) of ApproxMC3 and ApproxMC2 respectively. Table 1 clearly demonstrates that ApproxMC3 outperforms ApproxMC2 by up to 3 orders of magnitude. We show that not only ApproxMC3 is able to compute estimates within few seconds for formulas where ApproxMC2 but ApproxMC3 is also significantly faster for formulas where both ApproxMC3 and ApproxMC2 do not timeout. In particular, for all the formulas where both ApproxMC2 and ApproxMC3 did not timeout and took more than 1 seconds, the mean speedup is 284.40 – more than 2 orders of magnitude. Recall, ApproxMC3 solved 648 instances more than ApproxMC2.

Approximation Quality To measure the quality of approximation, we compared the approximate counts returned by ApproxMC3 with the counts computed by an exact model counter, viz. DSharp. Figure 2 shows the model counts computed by ApproxMC3, and the bounds obtained by scaling the exact counts with the tolerance factor $(\varepsilon = 0.8)$ for a small subset of benchmarks. The y-axis represents model counts on log-scale while the x-axis represents benchmarks ordered in ascending order of model counts. We observe that for *all* the benchmarks, ApproxMC3 computed counts within the tolerance. Furthermore, for each instance, the observed tolerance (ε_{obs}) was calculated as $\max(\frac{|R_{F\downarrow S}|}{\mathrm{AprxCount}} - 1, \frac{\mathrm{AprxCount}}{|R_{F\downarrow S}|} - 1)$, where $\mathrm{AprxCount}$ is the estimate computed by ApproxMC3. We observe that the arithmetic mean of ε_{obs} across all benchmarks is 0.038 (resp. 0.37 for ApproxMC2) – far better than the theoretical guarantee of 0.8.

5 Conclusion

Model counting is a fundamental problem in artificial intelligence with a wide range of applications including probabilistic reasoning, quantified information flow, and the like. Hashing-based techniques have emerged as a promising approach that seeks to combine theoretical guarantees with scalability. Our profiling of the state of the art approximate model counter, ApproxMC2, revealed that over 99.99% of time is consumed by the underlying SAT solver, CryptoMiniSat, a specialized solver designed to handle CNF-XOR formulas. In this paper, we designed a new framework to handle the CNF-XOR formulas arising from hashing-based techniques and the new tool, called ApproxMC3, is able to achieve two to three orders of magnitude speedup over the existing state of the art approximate counters. With a timeout of 5000 seconds, ApproxMC2 could only solve 492 benchmarks while ApproxMC3 could solve 1140 benchmarks – a difference of 648 benchmarks. Note that ApproxMC3 retains the same theoretical guarantees of ApproxMC2.

Acknowledgements We are grateful to the anonymous Reviewer #3 for the excellent suggestions to rewrite the Introduction. We are thankful to the organizers of the 2014 SAT-SMT Summer School, the venue of our first in-person meeting. This work was supported in part by NUS ODPRT Grant R-252-000-685-133 and AI Singapore Grant R-252-000-A16-490. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore `https://www.nscc.sg`.

References

Achlioptas, D., and Theodoropoulos, P. 2017. Probabilistic model counting with short xors. In *In Proc. of SAT*.

Achlioptas, D.; Hammoudeh, Z.; and Theodoropoulos, P. 2018. Fast sampling of perfectly uniform satisfying assignments. In *Proc. of SAT*.

Arora, S., and Barak, B. 2009. *Computational Complexity: A Modern Approach*. Cambridge Univ. Press.

Aziz, R. A.; Chu, G.; Muise, C.; and Stuckey, P. 2015. Sat: Projected model counting. In *Proc. of SAT*, 121–137.

Biondi, F.; Enescu, M.; Heuser, A.; Legay, A.; Meel, K. S.; and Quilbeuf, J. 2018. Scalable approximation of quantitative information flow in programs. In *Proc. of VMCAI*.

Chakraborty, S.; Meel, K. S.; Mistry, R.; and Vardi, M. Y. 2016. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013a. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *Proc. of CAV*, 608–623.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013b. A scalable approximate model counter. In *Proc. of CP*, 200–216.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2014. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, 1–6.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*.

Dueñas-Osorio, L.; Meel, K. S.; Paredes, R.; and Vardi, M. Y. 2017. Sat-based connectivity reliability estimation for power transmission grids. Technical report, Rice University.

Eén, N., and Biere, A. Effective preprocessing in sat through variable and clause elimination. In Bacchus, F., and Walsh, T., eds., *Proc. of SAT*, 61–75.

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2013. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*, 334–342.

Goldreich, O. 1999. The Counting Class #P. Lecture notes of course on "Introduction to Complexity Theory", Weizmann Institute of Science.

Gomes, C. P.; Hoffmann, J.; Sabharwal, A.; and Selman, B. 2007. Short XORs for Model Counting; From Theory to Practice. In *SAT*, 100–106.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*, volume 21, 54–61.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2009. Model counting. In *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications. 633–654.

Ivrii, A.; Malik, S.; Meel, K. S.; and Vardi, M. Y. 2015. On computing minimal independent support and its applications to sampling and counting. *Constraints* 1–18.

Lagniez, J.-M., and Marquis, P. 2017. An improved decision-dnnf compiler. In *Proc. of IJCAI*, 667–673.

Meel, K. S.; Vardi, M. Y.; Chakraborty, S.; Fremont, D. J.; Seshia, S. A.; Fried, D.; Ivrii, A.; and Malik, S. 2016. Constrained sampling and counting: Universal hashing meets sat solving. In *Proc. of Beyond NP Workshop*.

Meel, K. S. 2014. *Sampling Techniques for Boolean Satisfiability*. Rice University. M.S. Thesis.

Meel, K. S. 2017. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. Ph.D. Dissertation, Rice University.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82(1):273–302.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, 244–257.

Stockmeyer, L. 1983. The complexity of approximate counting. In *Proc. of STOC*, 118–126.

Toda, S. 1989. On the computational power of PP and (+)P. In *Proc. of FOCS*, 514–519. IEEE.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3):410–421.

Zhao, S.; Chaturapruek, S.; Sabharwal, A.; and Ermon, S. 2016. Closing the gap between short and long xors for model counting. In *Proc. of AAAI*.