Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling *

Mate Soos[1], Stephan Gocht[2], and Kuldeep S. Meel[1]

[1] School of Computing, National University of Singapore
[2] Lund University

Abstract. Given a Boolean formula, the problem of counting seeks to estimate the number of solutions of F while the problem of uniform sampling seeks to sample solutions uniformly at random. Counting and uniform sampling are fundamental problems in computer science with a wide range of applications ranging from constrained random simulation, probabilistic inference to network reliability and beyond. The past few years have witnessed the rise of hashing-based approaches that use XOR-based hashing and employ SAT solvers to solve the resulting CNF formulas conjuncted with XOR constraints. Since over 99% of the runtime of hashing-based techniques is spent inside the SAT queries, improving CNF-XOR solvers has emerged as a key challenge.
In this paper, we identify the key performance bottlenecks in the recently proposed BIRD architecture, and we focus on overcoming these bottlenecks by accelerating the XOR handling within the SAT solver and on improving the solver integration through a smarter use of (partial) solutions. We integrate the resulting system, called BIRD2, with the state of the art approximate model counter, ApproxMC3, and the state of the art almost-uniform model sampler UniGen2. Through an extensive evaluation over a large benchmark set of over 1896 instances, we observe that BIRD2 leads to consistent speed up for both counting and sampling, and in particular, we solve 77 and 51 more instances for counting and sampling respectively.

1 Introduction

A CNF-XOR formula φ is represented as conjunction of two Boolean formulas $\varphi_{\text{CNF}} \wedge \varphi_{\text{XOR}}$ wherein φ_{CNF} is represented in Conjunctive Normal Form (CNF) and φ_{XOR} is represented as conjunction of XOR constraints. While owing to the NP-completeness of CNF, every CNF-XOR formula can be represented as a CNF formula with only a linear increase in the size of the resulting formula, such a transformation may not be ideal in several scenarios. In particular, it is well known that modern Conflict Driven Clause Learning (CDCL) SAT solvers perform poorly on XOR formulas represented in CNF form despite the existence of efficient polynomial time decision procedures for XOR constraints. Furthermore, constraints arising from domains such as cryptanalysis and circuits can be

* The resulting tools ApproxMC4 and UniGen3 are available open source at https://github.com/meelgroup/approxmc and https://github.com/meelgroup/unigen.

naturally described as CNF-XOR formulas and these domains served as the early inspiration for design of SAT solvers with native support for XORs through the usage of Gaussian Elimination. These efforts lead to the development of CryptoMiniSat, a SAT solver that sought to perform Conflict Driven Clause Learning and Gaussian Elimination in tandem. The architecture of the early verisons of CryptoMiniSat sought to employ disjoint storage of CNF and XOR clauses – reminiscent to the architecture of SMT solvers.

While CryptoMiniSat was originally designed for cryptanalysis, its ability to handle XORs natively has led it to be a fundamental building block of the hashing-based techniques for approximate model counting and sampling. Model counting, also known as #SAT, and uniform sampling of solutions for Boolean formulas are two fundamental problems in computer science with a wide variety of applications [1, 11, 18]. The core idea of hashing-based techniques for approximate counting and almost-uniform sampling is to employ XOR-based 3-wise independent hash functions [3] to partition the solution space of F into *roughly equal small* cells of solutions. The usage of XOR-based hash functions allows us to represent a cell as conjunction of a Boolean formula in conjunctive normal form (CNF) and XOR constraints, and a SAT solver is invoked to enumerate solutions inside a randomly chosen cell. The corresponding counting and sampling algorithms typically employ the underlying solver in an incremental fashion and invoke the solver thousands of times, thereby necessitating the need for runtime efficiency. In this context, Soos and Meel [19] observed that the original architecture of CryptoMiniSat did not allow a straightforward integration of pre- and in-processing which of late has emerged to be key techniques in SAT solving. Accordingly, Soos and Meel [19] proposed a new architecture, called BIRD, that relied on the key idea of keeping the XOR constraints in both CNF form and XOR form. Soos and Meel integrated BIRD into CryptoMiniSat, and showed that state of the art approximate model counter, ApproxMC, when integrated with the new version of CryptoMiniSat achieves significant runtime improvements. The resulting version of ApproxMC was called ApproxMC3.

Motivated by the success of BIRD in achieving significant runtime performance improvements, we sought to investigate the key bottlenecks in the runtime performance of CryptoMiniSat when handling CNF+XOR formulas. Given the prominent usage of CNF-XOR formulas by the hashing based techniques, we study the runtime behavior of CryptoMiniSat for the the queries issued by the hashing-based approximate counters and samplers, ApproxMC3 and UniGen2 respectively. Our investigation leads us to make five core technical contributions. The first four contributions contribute towards architectural advances in handling of CNF-XOR formulas while the fifth contribution focuses on algorithmic improvements in the hashing-based techniques for counting and sampling:

1. **Matrix row handling improvements** for efficient propagation and conflict checking of XOR constraints

[3] While approximate counting techniques [10] only require 2-wise independent hash functions, hashing-based sampling techniques [6, 9] require 3-wise independent hash functions

2. **XOR constraint detaching** from the standard unit propagation system for higher unit propagation speed
3. **Lazy reason clause generation** to reduce reason generation overhead for unused reasons generated from XOR constraints
4. **Allowing partial solution extraction** by the SAT solver
5. **Intelligent reuse of solutions** by hashing-based techniques to reduce the number of SAT calls

We integrate these improvements into the BIRD framework, the resulting framework is called BIRD2. The BIRD2 framework is applied to state of the art approximate model counter, ApproxMC3, and to the almost-uniform sampler UniGen2 [6, 9]. The resulting counter and sampler are called ApproxMC4 and UniGen3 respectively. We conducted an extensive empirical evaluation with over 1800 benchmarks arising from diverse domains with computational effort totalling 50,000 CPU hours. With a timeout of 5000 seconds, ApproxMC3 and UniGen2+BIRD were able to solve only 1148 and 1012 benchmarks, while ApproxMC4 and UniGen3 solved 1225 and 1063 benchmarks respectively. Furthermore, we observe a consistent speedup for most of the benchmarks that could be solved by ApproxMC3 and UniGen2+BIRD. In particular, the PAR-2[4] score improved from 4146 with ApproxMC3 to 3701 with ApproxMC4. Similarly, the corresponding PAR-2 scores for UniGen3 and UniGen2+BIRD improved to 4574 from 4878.

2 Notations and Preliminaries

Let F be a Boolean formula in conjunctive normal form (CNF) and $\mathsf{Vars}(F)$ the set of variables in F. Unless otherwise stated, we use n to denote the number of variables in F i.e., $n = |\mathsf{Vars}(F)|$. An assignment 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 $sol(F)$. If we are only interested in a subset of variables $S \subseteq \mathsf{Vars}(F)$ we will use $sol(F)_{\downarrow S}$ to indicate the projection of $sol(F)$ on S.

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

A *uniform sampler* outputs a solution $y \in sol(F)$ such that $\mathsf{Pr}[y \text{ is output}] = \frac{1}{|sol(F)|}$. An *almost-uniform sampler* relaxes the guarantee of uniformity and in particular, ensures that $\frac{1}{(1+\varepsilon)|sol(F)|} \leq \mathsf{Pr}[y \text{ is output}] \leq \frac{1+\varepsilon}{|sol(F)|}$.

[4] PAR-2 score, that is, penalized average runtime, assigns a runtime of two times the time limit (instead of a "not solved" status) for each benchmark not solved by a tool.

Universal hash functions Let $n, m \in \mathbb{N}$ and $\mathcal{H}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ be a family of hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} \mathcal{H}(n,m)$ to denote the probability space obtained by choosing a function h uniformly at random from $\mathcal{H}(n,m)$. To measure the quality of a hash function we are interested in the set of elements of S mapped to α by h, denoted $\mathsf{Cell}_{\langle S,h,\alpha\rangle}$ and its cardinality, i.e., $|\mathsf{Cell}_{\langle S,h,\alpha\rangle}|$. To avoid cumbersome terminology, we abuse notation slightly and we use $\mathsf{Cell}_{\langle F,m\rangle}$ (resp. $\mathsf{Cnt}_{\langle F,m\rangle}$) as shorthand for $\mathsf{Cell}_{\langle sol(F),h,\alpha\rangle}$ (resp. $|\mathsf{Cell}_{\langle sol(F),h,\alpha\rangle}|$).

Definition 1. *A family of hash functions $\mathcal{H}(n,m)$ is k-wise independent*[5] *if $\forall \alpha_1, \alpha_2, \ldots \alpha_k \in \{0,1\}^m$ and for distinct $\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_k \in \{0,1\}^n$, $h \xleftarrow{R} \mathcal{H}(n,m)$,*

$$\mathsf{Pr}\left[(h(\mathbf{y}_1) = \alpha_1) \wedge (h(\mathbf{y}_2) = \alpha_2) \ldots \wedge (h(\mathbf{y}_k) = \alpha_k)\right] = \left(\frac{1}{2^m}\right)^k \tag{1}$$

Note that every k-wise independent hash family is also $k-1$ wise independent.

Prefix Slicing While universal hash families have nice concentration bounds, they are not adaptive, in the sense that one cannot build on previous queries. In several applications of hashing, the dependence between different queries can be exploited to extract improvements in theoretical complexity and runtime performance. Thus, we are typically interested in prefix slices of hash functions [10] as follows.

Definition 2. *For every $m \in \{1, \ldots n\}$, the m^{th} prefix-slice of h, denoted $h^{(m)}$, is a map from $\{0,1\}^n$ to $\{0,1\}^m$, such that $h^{(m)}(\mathbf{y})[i] = h(\mathbf{y})[i]$, for all $y \in \{0,1\}^n$ and for all $i \in \{1, \ldots m\}$. Similarly, the m^{th} prefix-slice of α, denoted $\alpha^{(m)}$, is an element of $\{0,1\}^m$ such that $\alpha^{(m)}[i] = \alpha[i]$ for all $i \in \{1, \ldots m\}$.*

Explicit hash functions The most common explicit hash family used in state of the art sampling and counting techniques is based on random XOR constraints. Viewing $\mathsf{Vars}(F)$ as a vector $\boldsymbol{x}$ of dimension $n \times 1$, we can represent the hash family as follows: Let $\mathcal{H}_{xor}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ be the family of functions of the form $h(x) = \boldsymbol{M}\boldsymbol{x} + \boldsymbol{b}$ with $\boldsymbol{M} \in \mathbb{F}_2^{m \times n}$ and $\boldsymbol{b} \in \mathbb{F}_2^{m \times 1}$ where the entries of $\boldsymbol{M}$ and b are independently generated according to the Bernoulli distribution with probability $1/2$. Observe that $h^{(m)}(x)$ can be written as $h^{(m)}(\boldsymbol{x}) = \boldsymbol{M}^{(m)}\boldsymbol{x} + \boldsymbol{b}^{(m)}$, where $\boldsymbol{M}^{(m)}$ denotes the submatrix formed by the first m rows and n columns of $\boldsymbol{M}$ and $\boldsymbol{b}^{(m)}$ is the first m entries of the vector $\boldsymbol{b}$. It is well known that $\mathcal{H}_{xor}$ is 3-wise independent [9].

[5] The phrase *strongly 2-universal* is also used to refer to 2-wise independent as noted by Vadhan in [23], although the concept of 2-universal hashing proposed by Carter and Wegman [4] only required that $\mathsf{Pr}[h(x) = h(y)] \leq \frac{1}{2^m}$

3 Background

The general idea of hashing-based model counting and sampling is to use a hash function from a suitable family, e.g. $\mathcal{H}_{xor}$, to divide the solution space into cells that are sufficiently small such that all solutions within a cell can be enumerated efficiently. Given such a cell, its size can then be used to estimate the total count of solutions or we can return a random element of this small cell to produce a sample. Hence, hashing-based sampling and counting are closely related.

3.1 Hashing-Based Model Counting

The seminal work of Valiant [24] established that #SAT is #P-complete. Toda [22] showed that the entire polynomial hierarchy is contained inside the complexity class defined by a polynomial time Turing machine equipped with #P oracle. Building on Carter and Wegman's [4] seminal work of universal hash functions, Stockmeyer [21] proposed a probabilistic polynomial time procedure relative to an NP oracle to obtain an (ε, δ)-approximation of F.

The core theoretical idea of the hashing-based approximate solution counting framework proposed in ApproxMC [8], building on Stockmeyer [21], is to employ 2-universal hash functions to partition the solution space, denoted by $sol(F)$ for a formula F, into *roughly equal small* cells, wherein a cell is called *small* if it has solutions less than or equal to a pre-computed threshold, thresh. An NP oracle 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 thresh $+ 1$ solutions. In practice, a SAT solver is used to realize the NP oracle. To ensure polynomially many calls to the oracle, thresh is set to be polynomial in the input parameter ε. To determine the right number of cells, i.e., the value of m for $\mathcal{H}(n, m)$, a search procedure is invoked. Finally, the subroutine, called ApproxMCCore, computes the estimate as the number of solutions in the randomly chosen cell scaled by the number of cells (i.e, 2^m). To achieve probabilistic amplification of the confidence, multiple invocations of the underlying subroutine, ApproxMCCore, are performed with the final count computed as the median of estimates returned by ApproxMCCore.

Two key algorithmic improvements proposed in ApproxMC2 [10] are significant to practical performance: (1) the search for the right number of cells can be performed via galloping search, and (2) one can first perform linear search over a small enough interval (chosen to be of size 7) around the value of m found in the previous iteration of ApproxMCCore. The practical profiling of ApproxMC2 reveals that linear search is sufficient after the first invocation of ApproxMCCore. Note that the linear search seeks to identify a value of m such that $\mathsf{Cnt}_{\langle F,m-1\rangle} \geq$ thresh and $\mathsf{Cnt}_{\langle F,m\rangle} <$ thresh for an appropriately chosen thresh. ApproxMC is currently in its third generation: ApproxMC3.

3.2 Hashing-Based Sampling

Jerrum, Valiant, and Vazirani [14] showed that the approximate counting and almost-uniform counting are polynomially inter-reducible. Building on Jerrum et

al.'s result, Bellare, Goldreich, and Petrank [2] proposed a probabilistic uniform generator that makes polynomially many calls to an NP oracle where each NP query is the input formula F conjuncted with constraints encoding a degree n polynomially representing n-wise independent hash functions where n is the number of variables in F. The practical implementation of Bellare et al's technique did not scale beyond few tens of variables. Chakraborty, Meel, and Vardi [7, 9], sought to combine the inter-reducibility and the usage of independent hashing, and proposed a hashing-based framework, called UniGen, that employs 3-wise independent hashing and makes polynomially many calls to an NP oracle.

The core theoretical idea of the hashing-based sampling framework, proposed in UniGen, exploits the close relationship between counting and sampling. UniGen first invokes ApproxMC to compute an estimate of the number of solutions of the given formula F. It then uses the count to determine the number of cells that the solution space should be partitioned into using 3-wise independent hash functions. At this point, it is worth mentioning that the state of the art hashing-based sampling employ 3-wise independent hash functions. Fortunately, the family of hash functions, $\mathcal{H}_{xor}$, is also known to be 3-wise independent. There after, similar to ApproxMC, a linear search over a small enough interval (chosen to be of size 4) is invoked to find the *right* value of m where a randomly chosen cell's size is within the desired bounds. For such a cell, all its solutions are enumerated and one of the solutions is randomly chosen. Again, similar to ApproxMC2 (and ApproxMC3), the linear search seeks to identify a value of m such that $\mathsf{Cnt}_{\langle F,m-1\rangle} \geq \mathsf{thresh}$ and $\mathsf{Cnt}_{\langle F,m\rangle} < \mathsf{thresh}$ for an appropriately chosen thresh. UniGen is currently in its second generation: UniGen2 [6].

3.3 The Underlying SAT Solver

The underlying SAT solver is invoked through subroutine BoundedSAT, which is implemented using CryptoMiniSat. Formally, BoundedSAT takes as inputs a formula F, a threshold thresh, and a sampling set S, and returns a subset Y of $sol(F)_{\downarrow S}$, such that $|Y| = \min(\mathsf{thresh}, |sol(F)_{\downarrow S}|)$. The formula F consists of the original formula, which we want to count or sample, conjuncted with a set of XOR constraints defined through a hash function sampled from the family $\mathcal{H}_{xor}$. We henceforth denote such formulas as CNF-XOR formulas. Note that the efficient encoding of XOR constraints into CNF requires the introduction of new variables and hence the sampling set S usually does not contain all variables in F.

As is consistent with prior studies, profiling of ApproxMC3 and UniGen2 reveal that over 99% of the time is spent in the runtime of BoundedSAT. Therefore recent efforts have focused on improving BoundedSAT. Soos and Meel [19] sought to address the performance of the underlying SAT solver by proposing a new architecture, called BIRD, that allows the usage of in- and pre-processing techniques for a Gauss Jordan Elimination (GJE)-augmented SAT solver. ApproxMC2, integrated with BIRD, called ApproxMC3, gave up to three orders of magnitude runtime performance improvement. Such significant improvements are rare in the SAT community. Encouraged by Soos and Meel's observations, we seek to

build on top of BIRD to achieve an even tighter integration of the underlying SAT solver and ApproxMC3/UniGen2.

BIRD: Blast, Inprocess, Recover, and Destroy Pre- and inprocessing techniques are known to have a large impact on the runtime performance of SAT solvers. However, earlier Guassian elimination architectures were unable to perform these techniques. Motivated by this inability, Soos and Meel [19] proposed a new framework, called BIRD, that allows usage of inprocessing techniques for GJE-augmented CDCL solvers. The key idea of BIRD is to blast XOR clauses into CNF clauses so that any technique working solely on CNF clauses does not violate soundness of the solver. To perform Gauss-Jordan elimination, one needs efficient algorithms and data structures to extract XORs from CNF. The entire framework is presented 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 (GJE) on XOR clauses until inprocessing is scheduled
Step 5 **Destroy** XOR clauses and goto **Step 2**

The above loop terminates as soon as a satisfying assignment is found or the formula is proven UNSAT. The BIRD architecture separates inprocessing from CDCL solving and therefore every sound inprocessing step can be employed.

4 Technical Contributions to CNF-XOR Solving

Inspired by the success of BIRD, we seek to further improve the underlying SAT solver's architecture based on the queries generated by the hashing-based techniques. To this end, we relied on extensive profiling of CryptoMiniSat augmented with BIRD to identify the key performance bottlenecks, and propose solutions to overcome some of the challenges.

4.1 Detaching XOR Clauses From Watch-lists

Given a formula F in CNF, the recovery phase of BIRD attempts to construct a set of XORs, H such that $F \rightarrow H$. As detailed in [19], the core technique for recovery of an XOR of size k is to establish whether the required 2^{k-1} combinations of clauses are implied by the existing CNF clauses. For example, the XOR $x_1 \oplus x_2 \oplus x_3 = 0$ (where $k = 3$) can be recovered if the existing set of CNF clauses implies the following $4(= 2^{3-1})$ clauses: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. To this end, the first stage of the recovery phase of BIRD iterates over the CNF clauses and for a given

clause, called base_cl of size k, searches whether the remaining $2^{k-1}-1$ clauses are implied as well, in which case the resulting XOR is added. It is worth noting that a clause can imply multiple clauses over the the set of variables of base_cl; For example if the base_cl $= (x_1 \vee \neg x_2 \vee x_3)$, then the clause $(\neg x_1)$ would imply the two clauses $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ and $(\neg x_1 \vee x_2 \vee x_3)$. Note that given a base_cl, we are only interested in clauses over the variables in base_cl.

During blasting of XORs into CNF, XORs are first cut into smaller XORs by introducing auxiliary variables. Hence, the first stage of recovery phase must recover these smaller XORs and the second phase reconstructs the larger XORs by XOR-ing two XORs together if they differ only on one variable, referred to as a *clash variable*. For example, $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_3 \oplus x_4 \oplus x_5 = 1$ can be XOR-ed together over clash variable x_3 to obtain $x_1 \oplus x_2 \oplus x_4 \oplus x_5 = 1$.

Since BIRD performs CDCL in tandem with Gauss-Jordan elimination, it is worth noting that the Gauss-Jordan elimination (GJE)-based decision procedure is sound and complete, i.e., all unit propagations and conflicts implied by the given set of XORs would be discovered by a GJE-based decision procedure. For the initial formula (in CNF) F and the recovered set of XORs, H, if a set of CNF clauses G is implied by H, then presence or absence of G does not affect soundness and completeness of GJE-augmented CDCL engine. Our extensive profiling of the BIRD framework integrated in CryptoMiniSat revealed a significant time spent in examination of clauses in G during unit propagation. To this end, we sought to ask how to design an efficient technique to find all the CNF clauses implied by the recovered XORs. These clauses could be detached from unit propagation without any negative effect on correctness of execution.

A straightforward approach would be to mark all the clauses during the blasting phase of XORs into CNF. However, the incompleteness of the recovery phase of BIRD does not guarantee that all such marked clauses are indeed implied by the recovered set of XORs. Another challenge in the search for detachable clauses arises due to construction of larger XORs by combining smaller XORs. For example, while $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_3 \oplus x_4 \oplus x_5 = 1$ imply $(x_1 \vee x_2 \vee \neg x_3)$ and $(x_3 \vee x_4 \vee x_5)$, the combined XOR $x_1 \oplus x_2 \oplus x_4 \oplus x_5 = 1$ does not imply $(x_1 \vee x_2 \vee \neg x_3)$ and $(x_3 \vee x_4 \vee x_5)$.

Two core insights inform our design of the modification of the recovery phase and search for detachable clauses. Firstly, given a base clause base_cl, if a clause cl participates in the recovery of XORs over the variables in base_cl, then cl is implied by the recovered XOR if the number of variables in cl is the same as that of base_cl. We call such a clause cl a *fully participating clause*. Secondly, let G_1 and G_2 be the set of CNF clauses implied by two XORs q_1 and q_2 that share exactly one variable, say x_i. Let $U = (\mathsf{Vars}(q_1) \cup \mathsf{Vars}(q_2)) \setminus x_i$. Let q_3 be the XOR obtained by XORing together q_1 and q_2, then, $sol(q_3)_{\downarrow U} \subseteq sol(G_1 \wedge G_2)_{\downarrow U}$ if x_i does not appear in the remaining clauses, i.e., $x_i \notin \mathsf{Var}[F \setminus (G_1 \cup G_2)]$.

The above two insights lead us to design a modified recovery and detachment phase as follows. During recovery, we add every *fully participating clause* to the set of detachable clauses D. Let $\mathcal{U} = S \cup (\mathsf{Vars}(D) \cap \mathsf{Vars}(F \setminus D))$. Then, the recovery of longer XORs is only performed over clash variables that do not belong to $\mathcal{U}$.

We then detach the clauses in D from watch-lists during GJE-augmented CDCL phase, mark the clash variables as non-decision variables, perform CDCL, and only reattach the clauses and re-set the clash variables to be decision variables after the Destroy phase of BIRD.

If the formula is satisfiable, the design of the solver is such that the solution is always found during the GJE-augmented CDCL solving phase. Since clauses in D are detached and the clash variables are set to be not decided on during this phase, the clash variables are always left unassigned. As discussed below, however, we only need to extract solutions over the sampling set S, therefore the solution found is adequate as-is, without the clash variables, which are by definition not over S as they are only introduced for having short encodings of XORs into CNF.

Conceptually, this approach reconciles the overhead introduced by BIRD, i.e., that XOR constraints are also present as regular clauses, with the neatness of the original CryptoMiniSat that stored XOR and regular constraints in different data structures. This reconciliation takes the best of both worlds.

4.2 Fast Propagation/Conflict Detection and Reason Generation

We identified two key bottlenecks in the the current GJE component of BIRD framework integrated in CryptoMiniSat, which we sought to improve upon. To put our contributions in the context, we first describe the technical details of the core data structures and algorithms.

Han-Jiang's GJE To perform Gaussian elimination on a set of XORs, the XORs are represented as a matrix where each row represents an XOR and each column represents a variable. The framework proposed by Soos et al. updates the matrix whenever a variable is assigned and removes the assigned variable from all XORs by zeroing out the corresponding column. However, using the matrix in such a way involves significant memory copying during backtracking due to having to revert the matrix to a previous version.

To avoid the overhead, Han and Jiang proposed a new framework [13] building on Simplex-like techniques that performs Gauss-Jordan elimination, i.e., using reduced row echelon form instead of row echelon form. The key data structure innovation was to employ a two-watched variable scheme for each row of the matrix wherein the watched variables are called basic and non-basic variables. Essentially, the basic variables are the variables on the diagonal of a matrix in reduced row echelon form and hence every row has exactly one basic variable and the basic variable only occurs in one row. Similar to standard CDCL solving, when a matrix row's watch is assigned, the GJE component must determine whether the row (1) propagates, (2) needs to assign a new watch, (3) is satisfied, or (4) is conflicted. It is worth recalling that a row would propagate if all except one variable has been assigned and would conflict or be satisfied if all the variables in a row have been assigned. Furthermore, we need to find a new watch if a watched variable was assigned and there is more than one unassigned variable left. If a

basic variable is replaced by a new watch then the two corresponding columns are swapped and the reduced row echelon form is recomputed. In practice swapping columns is avoided by keeping track of which column is a basic variable.

For propagation, checking for conflict, and conflict clause generation Han-Jiang proposed a sequential walk through a row that eagerly computes the reason clause and stops when it encounters a new watch variable or reaches until the end of the row. At that point, the system (1) knows whether the row is satisfied, propagating, or conflicted, and (2) if not satisfied, has eagerly computed the reason clause for the propagation or the conflict.

For general benchmarks where XOR constraints do not play an influential role in determining satisfiability of the underlying problem, the GJE component can be as small as 10% of the entire solving time. However, for formulas generated generated by hashing-based techniques, our profiling demonstrated several cases where the Gaussian elimination component could be very time consuming, taking up to 90% of solving time.

While the choice of GJE combined with clever data structure maintenance led to significant improvements of the runtime of Gaussian Elimination component, our profiling identified two processes as key bottlenecks: propagation checking and reason generation. We next discuss our proposed algorithmic improvements that achieve significant runtime improvement by addressing these bottlenecks.

Tinted Fast Unit Propagation The core idea to achieve faster propagation is based on bit-level parallelism via the different native operations supported by modern CPUs. In particular, modern CPUs provide native support for basic bitwise operations on bit fields such as AND, INVERT, hamming weight computation (i.e., the number of non-zero entries), and *find first set* (i.e., finding the index of first non-zero bit). Given the widespread support of SIMD extensions, the above operations can be performed at the rate of 128...512 bits per instruction. Therefore, the core data structure represents every 0-1 vector as a bit field.

A set of XORs over n variables $x_1, \ldots, x_n$ is represented as $\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b}$ for a 0-1 matrix $\boldsymbol{M}$ of size $m \times n$, 0-1 vector b of length m and $\boldsymbol{x} = (x_1, \ldots, x_n)^T$. Consider the $i-$th row of $\boldsymbol{M}$, denoted by $\boldsymbol{M}[i]$. Let $\boldsymbol{a}$ be a 0-1 vector of size n such that $\boldsymbol{a}[j]$=1 if the variable x_j is assigned True or False, and 0 in case x_j is unassigned. Let $\boldsymbol{v}$ be a 0-1 vector of size n such that $\boldsymbol{v}[j] = 1$ if x_j is set to True and 0 otherwise. Let $\overline{\boldsymbol{z}}$ be the bitwise inverse of a 0-1 vector $\boldsymbol{z}$ and & be the bitwise AND operation. Let $W_{unass} = \mathsf{hamming\_weight}(\overline{\boldsymbol{a}}\&\boldsymbol{M}[i])$ the number of unassigned variables in the XOR represented by row i, and $W_{val} = \mathsf{hamming\_weight}(\boldsymbol{v}\&\boldsymbol{M}[i])$ the number of satisfied variables. We view the computation of W_{unass} and W_{val} as viewing the world of $\boldsymbol{M}$ through the tinted lens of $\boldsymbol{v}$ and $\overline{\boldsymbol{a}}$. Now, the following holds:

1. Row i is satisfied if and only if $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 0$.
2. Row i causes a conflict if and only if $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$.
3. Row i propagates if and only if $W_{unass} = 1$. Propagated variable is the one that corresponds to the column with the only bit set in $\overline{\boldsymbol{a}}\&M[i]$. The value propagated is $(W_{val} \mod 2) \oplus \boldsymbol{b}[i]$.

4. A new watch needs to be found for row i if and only if $W_{unass} \geq 2$. The new watch is any one of the variables corresponding to columns with the bits set to 1 in $\overline{\boldsymbol{a}}\&\boldsymbol{M}[i]$, except for the already existing watch variable.

Reason Generation For propagation and conflict we generate the reason clauses for row i as follows. We forward-scan $\boldsymbol{M}[i]$ for all set bits and insert the corresponding variable into the reason clause as a literal that evaluates to false under the current assignment. In the case of propagation, the literal added for the propagated variable, say x_j, is added as literal $\neg x_j$ if $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 0$ and x_j otherwise.

Example For example, let $\boldsymbol{b}[i] = 1$ and $\boldsymbol{M}[i] = 10011$ corresponding to variables $x_1, x_2, \ldots x_5$ and assignments 1?11? respectively, where "?" indicates an unassigned variable. Then $\boldsymbol{a} = 10110, \overline{\boldsymbol{a}}\&\boldsymbol{M}[i] = 00001, W_{unass} = 1, \boldsymbol{v} = 10110, \boldsymbol{v}\&\boldsymbol{M}[i] = 10010, W_{val} = 2$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$. Therefore, this row propagates (case 3 above), and the reason generated is $(\neg x_1 \vee \neg x_4 \vee x_5)$. If the assignements were 11110, then $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$ so this row conflicts (case 2 above), with conflict clause $(\neg x_1 \vee \neg x_4 \vee x_5)$.

Performance Notice that all cases only require bitwise and, inverse, hamming weight and find first set operations. To find a new watch in case 4 we first find the first bit that is set to 1 in $\bar{\boldsymbol{a}}\&\boldsymbol{M}$ by invoking find first set. In case the obtained index is the same as the existing watch variable, we remove the first 1-bit by left shifting and run find first set again to find the second 1-bit. Bitwise and and inverse are trivially single-assembly instructions. We use compiler intrinsics to execute find first set and hamming weight functions, which compile down to BSF and POPCNT in x86 assembly, respectively. It is worth pointing out that we keep the bit field representations of $\boldsymbol{a}$ and $\boldsymbol{v}$ synchronized when variables are assigned. During backtracking we reset these to zero and refill them as needed. For better cache efficiency, we use sequential set of bit-packed 64-bit integers to represent all bit-fields, rows, and matrices.

Although bit-packing is not a novel concept in the context of CNF-XOR solving, let us elaborate why we believe that our conribution is conceptually interesting. Soos et al. [20] used bit-packed pre- and post-evaluated matrices. Since post-evaluated matrices lose information, they have to be saved and reloaded on backtracking. Han and Jiang's code [13] changed this to using pre-evaluated matrices only, which free the system from having to save and reload. But it was slow, because bit-by-bit evaluation had to happen on every matrix row read (thanks to the missing post-evaluation matrix). Our improved approach is essentially merging the best of both worlds: fast evaluation, without having to save and reload.

4.3 Lazy Reason Clause Generation

As discussed earlier, the current BIRD performs eager reason clause generation in a spirit similar to the original proposal by Han and Jiang. At the time of

proposal of eager clause generation by Han and Jiang, the state of the art SAT solver at that time could solve problems with XOR clauses of sizes in few tens to few hundreds. The improved scalability, however, highlights the overhead due to eager reason clause generation. During our profiling, we observed that for several problems, the independent support of the underlying formula ranges in thousands, and therefore, leading to generation of reason clauses involving thousands of variables. The generation of such long reason clauses is time consuming and tedious. Furthermore, a significant fraction of reason clauses are never required during conflict analysis phase as we are, often, focused only on finding a 1UIP clause. Therefore, we seek to explore lazy reason clause generation.

Let the state of a clause c indicate whether c is satisfied, conflicted or undetermined (i.e., the clause is neither satisfied nor conflicted). The core design of our lazy generation technique is based on the following invariant satisfied by CDCL-based techniques: Once a (CNF/XOR) clause is satisfied or conflicted, the assignment to the variables in the clause does not change as long the state of the clause does not change. Observe that when a clause propagates, the propagated literal changes the state of the clause to satisfied. Furthermore, as long as all variables are assigned, the row will not participate in GJE because none of the contained variables can become a basic watch. Therefore, whenever an XOR clause propagates, we keep an index of the row and the propagating literal but do not compute the reason clause. Now, whenever a reason clause is requested, we compute the reason clause as detailed above and return a pointer to the computed reason clause, and index the computed clause by the corresponding row. To ensure correctness, whenever a row causes a propagation, we delete the existing reason clause but we do not eagerly compute the new corresponding reason clause. On the other hand, if a row is conflicting, the conflict analysis requires the reason clause immediately and as such the reason clause is eagerly computed.

Lazy reason clause generation allows us to skip the majority of reason clauses to be generated. Furthermore, given that a row cannot lead to more than one reason clause, it allows us to statically allocate memory for them. This is in stark contrast to the original implementation that not only eagerly computed all reason clauses, but also dynamically allocated memory for them, freeing the memory up during backtracking.

4.4 Skipping Solution Extension of Eliminated Variables

SAT solvers aim to present a clean and uncomplicated API interface with internal behavior typically hidden to enable fast pacing development of heuristics without necessitating change in the interface for the end users. While such a design philosophy allows easier integration, it may be an hindrance to achieving efficiency for the use cases that may not be seeking a simple off-the-shelf behavior. Given the surge of projected counting and sampling as the desired formulation, BoundedSAT is invoked with a sampling set and we are interested only in the assignment to variables in the sampling set. A naive solution would be to obtain a complete assignment over the entire set of variables and then extract an assignment over the

desired sampling set. In this context, we wonder if we can terminate early after the variables in the sampling set are assigned. In modern SAT solvers, once the solver has determined that the formula is satisfied, the *solution extension* subroutine is invoked that extends the current partial assignment to a complete assignment. Upon profiling, we observed that, during solution extension, a significant time is spent in computing an assignment to the variables eliminated due to Bounded Variable Elimination (BVE) [12] during pre- and inprocessing. When a solution is found, the eliminated clauses must be re-examined in reverse, linear, order to make sure the eliminated variables in the model are correctly assigned. This examination process can be time-consuming on large instances with large portions of the CNF eliminated.

BVE is widely used in modern SAT solvers owing to its ability to eliminate a large subset of the input formula and thereby allowing compact data structures. While disabling BVE would eliminate the overhead during solution extension phase, it would also significantly degrade performance during solving phase. Since we are interested in solutions only over the sampling set, we disable the invocation of bounded variable elimination for variables in the sampling set. Therefore, whenever the SAT solver determines that the current partial assignment satisfies the formula, all the variables in the sampling set are assigned and we do not invoke solution extension. The disabling of solution extension can save significant (over 20%) time on certain instances.

4.5 Putting it All Together: BIRD2

We combine improvements proposed above into our new framework, called BIRD2, a namesake to capture the primary architecture of Blast, In-process, Recover, Detach, and Destroy. For completeness, we present the core skeleton of BIRD2 in Algorithm 1. BIRD2 terminates as soon as a satisfying assignment is found or the formula is proven UNSAT. Similar to BIRD, BIRD2 architecture separates inprocessing from CDCL solving and therefore every sound inprocessing step can be employed.

Algorithm 1 BIRD2(φ) ▷ *φ has a mix of CNF and XOR clauses*

1: **Blast** XOR clauses into normal CNF clauses
2: **In-process** (and pre-process) over CNF clauses
3: **Recover** XOR clauses
4: **Detach** CNF clauses implied by recovered XOR clauses
5: Perform CDCL on CNF clauses with on-the-fly *improved* GJE on XOR clauses until: (a) in-processing is scheduled, (b) a satisfying assignment is found, or (c) formula is found to be unsatisfiable
6: **Destroy** XOR clauses and reattach detached CNF clauses. Goto line 2 if conditions (b) or (c) above don't hold. Otherwise, return satisfying assignment or report unsatisfiable.

5 Technical Contribution to Counting and Sampling

In this section, we discuss our primary technical contribution to hashing-based sampling and counting techniques.

5.1 Reuse of Previously Found Solutions

The usage of a prefix-slicing ensures monotonicity of the random variable, $\mathsf{Cnt}_{\langle F,i\rangle}$, since from the definition of prefix-slicing, we have that for all i, $h^{(i+1)}(x) = \alpha^{(i+1)} \implies h^{(i)}(x) = \alpha^{(i)}$. Formally,

Proposition 1. *For all* $1 \leq i < m$, $\mathsf{Cell}_{\langle F,i+1\rangle} \subseteq \mathsf{Cell}_{\langle F,i\rangle}$

Furthermore as is evident from the analysis of ApproxMC3 [10], the pairwise independence of the family $\mathcal{H}_{xor}$ implies $\frac{\mathsf{E}[\mathsf{Cnt}_{\langle F,i\rangle}]}{\mathsf{E}[\mathsf{Cnt}_{\langle F,j\rangle}]} = 2^{j-i}$. Therefore, once we obtain the set of solutions from invocation of BoundedSAT for $F \wedge (h^i)^{-1}(\mathbf{0})$ (i.e., after putting i XORs), we can potentially reuse the returned solutions when we are interested in enumerating solutions for $F \wedge (h^j)^{-1}(\mathbf{0})$. In particular, note that if $i > j$, then Proposition 1 implies that all the solutions $F \wedge (h^i)^{-1}(\mathbf{0})$ are indeed solutions for $F \wedge (h^j)^{-1}(\mathbf{0})$ and we can invoke BoundedSAT with adjusted threshold. On the other hand, for $i < j$, we can check if the solutions of $F \wedge (h^i)^{-1}(\mathbf{0})$ also satisfy $F \wedge (h^{i+1})^{-1}(\mathbf{0})$.

On closer observation, we find that the latter case may not be always helpful when i and j differ by more than a small constant since the ratio of their expected number of solutions decreases exponentially with $j - i$. Interestingly, as discussed in Section 3, both ApproxMC3 and UniGen2 employ linear search over intervals of sizes 4 to 7. for the right values of m. In particular, for both ApproxMC3 and UniGen2, the linear search seeks to identify a value of m^* such that $\mathsf{Cnt}_{\langle F,m^*-1\rangle} \geq \mathsf{thresh}$ and $\mathsf{Cnt}_{\langle F,m^*\rangle} < \mathsf{thresh}$ for an appropriately chosen thresh. Therefore, when invoking BoundedSAT for $i = k$ after determining that for $i = k + 1$, $\mathsf{Cnt}_{\langle F,k+1\rangle} < \mathsf{thresh}$, we can replace thresh with $\mathsf{thresh} - \mathsf{Cnt}_{\langle F,k+1\rangle}$. Similarly, when invoking BoundedSAT for $i = k$ after determining that for $i = k-1$, $\mathsf{Cnt}_{\langle F,k-1\rangle} \geq \mathsf{thresh}$, we first check how many solutions of $F \wedge (h^{k-1})^{-1}(\mathbf{0})$ satisfy $F \wedge (h^k)^{-1}(\mathbf{0})$. As noted above, in expectation, thresh/2 out of thresh solutions of $F \wedge (h^{k-1})^{-1}(\mathbf{0})$ would satisfy $F \wedge (h^k)^{-1}(\mathbf{0})$.

5.2 ApproxMC4 and UniGen3

That said, we turn our focus to hashing-based sampling and counting techniques to showcase the impact of BIRD2. To this end, we integrate BIRD2 along with the proposed technique in Section 5.1 into the state of the art hashing-based counting and sampling tools: ApproxMC3 and UniGen2 respectively. We call our improved counting tool ApproxMC4 and our improved sampling tool UniGen3.

Assurance of Correctness We believe it to be imperative to strongly verify correctness and quality of results provided by our tools, as it is not only possible but indeed easy to accidentally generate incorrect or low quality results, as

demonstrated by Chakraborty and Meel [5]. To ensure the quality and correctness of our sampler and counter, we used three methods: (1) fuzzed the system as first demonstrated in SAT by Brummayer et al. [3], (2) compared the approximate counts returned by ApproxMC4 with the counts computed by a known good exact model counter as previously performed by Soos and Meel [19], and (3) compared the distribution of samples generated by UniGen4 on an example problem against that of a known good uniform sampler as previously performed by Chakraborty et al. [9]. We focus on (1), i.e. fuzzing, here and defer the discussion about (2) and (3) to the next section.

Fuzzing is a technique [17] used to find bugs in code by generating random inputs and observing crashes, invariant check fails, and other errors from the output of the system under test. CryptoMiniSat has such a built-in fuzzer generating random CNFs and verifying the output of the solver. To account for XOR constraints, we improved the built-in fuzzer of CryptoMiniSat by adding a counting- and sampling-specific XOR-CNF generator. This inserts randomly generated XORs that form distinct matrices inside the generated CNFs and adds a randomly generated sampling set over some of these matrices. We also added hundreds of lines of invariant checks to our improved Gauss-Jordan elimination algorithm, running throughout our fuzzing tests. Running this improved fuzzer for many hundreds of CPU hours has greatly helped debugging and gaining confidence in our implementation.

6 Evaluation

To evaluate the performance and quality of approximations and samples computed by ApproxMC4 and UniGen3, we conducted a comprehensive study involving 1896 benchmarks as released by Soos and Meel [16] comprising a 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.

In the context of counting, we focused on a comparison of the performance of ApproxMC4 vis-a-vis ApproxMC3. In the context of sampling, a simple methodology would have been a comparison of UniGen3 vis-a-vis the state of the art sampler, UniGen2. Such a comparison, in our view, would be unfair to UniGen2 as while ApproxMC3 builds on BIRD framework, such is not the case for UniGen2. It is worth noting that the BIRD framework, proposed by Soos and Meel [19], can work as a drop-in replacement for the SAT solver in UniGen2, as it only changes the underlying SAT solver. Therefore, we used UniGen2 augmented with BIRD, called UniGen2+BIRD henceforth, as baseline for performance comparisons in the rest of this paper, as it is significantly faster than UniGen2, and therefore, will lead to a fair comparison and showcase improvements solely due to BIRD2.

To keep in line with prior studies, we set $\varepsilon = 0.8$ and $\delta = 0.8$ for ApproxMC3 and ApproxMC4 respectively. Similarly, we set $\varepsilon = 16$ for both UniGen3 and UniGen2+BIRD respectively. The experiments were conducted on a high performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with

2x12 real cores and 96GB of RAM. We use a timeout of 5000 seconds for each experiment, which consisted of running a tool on a particular benchmark.

6.1 Performance

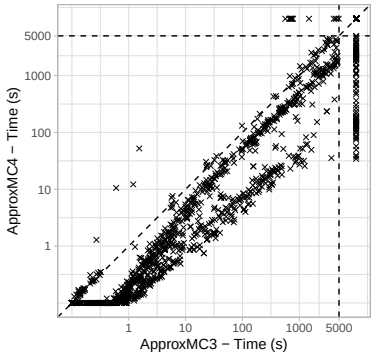

Fig. 1. Comparison of ApproxMC4 and ApproxMC3. ApproxMC4 is faster below the diagonal. Time outs are plotted behind the 5000s mark.

ApproxMC4 vis-a-vis ApproxMC3 Figure 1 shows a scatter plot comparing ApproxMC4 and ApproxMC3. Although, there are some benchmarks that are solved faster with ApproxMC3 there is a clear trend demonstrating the speed up achieved through our improvements: ApproxMC4 can solve many benchmarks more than 10 times faster and in total solves 77 more instances than ApproxMC3. In particular, ApproxMC3 and ApproxMC4 solved 1148 and 1225 instances respectively, while achieving PAR-2 scores of 4146 and 3701 respectively.

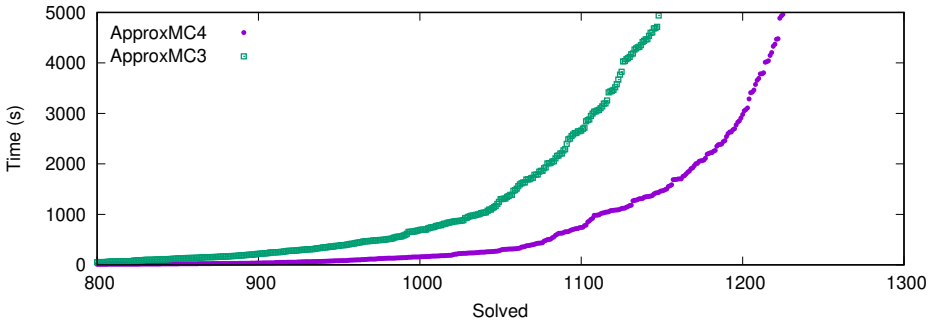

Fig. 2. Cactus plot showing behavior of ApproxMC4 and ApproxMC3

Figure 3 shows the cactus plot for ApproxMC3 and ApproxMC4. We present the number of benchmarks on the x-axis and the time taken on the y-axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve for the corresponding tool.

To present a detailed picture of performance gain achieved by ApproxMC4 over ApproxMC3, we present a runtime comparison of ApproxMC4 vis-a-vis ApproxMC3 in Table 1 on a subset of benchmarks. Column 1 of the table presents benchmarks names, while columns 2 and 3 list the number of variables and clauses. Column 4 and 5 list the runtime (in seconds) of ApproxMC4 and ApproxMC3, respectively.

While investigating the large improvements in performance, we observed that when both the sampling set and the number of solutions is large for a problem, the new system can be up to an order of magnitude faster. In these cases the Gauss-Jordan elimination (GJE) component of the SAT solver dominated the runtime of ApproxMC3 due to the large matrices involved in such problems. The improvements of BIRD2 has led to significant improvement in efficiency of GJE component and we observe that the runtime, in such instance, is now often dominated by the CDCL solver's propagation and conflict clause generation routines.

UniGen3 vis-a-vis UniGen2+BIRD Similar to Figure 2, Figure 3 shows the cactus plot for UniGen3, UniGen2+BIRD, and UniGen2. We present the number of benchmarks on the x-axis and the time taken on the y-axis. UniGen3 and UniGen2+BIRD were able to solve 1012 and 1063 instances, respectively while achieving PAR-2 scores of 4574 and 4878, respectively. UniGen2 could solve only 360 benchmarks, thereby justifying our choice of implementing UniGen2+BIRD as a baseline for fair comparison to showcase strengths of BIRD2. We would like to highlight that the cactus plot shows that given a 2600 second timeout, UniGen can sample as many benchmarks as UniGen2+BIRD would do for a 5000s timeout.

To present a clear picture of performance gain by UniGen3 over UniGen2+BIRD, we present runtime comparison for UniGen3 vis-a-vis UniGen2+BIRD in Table 1, where in addition to data on ApproxMC3 and ApproxMC4, columns 5 and 6 lists the runtime for UniGen3 and UniGen2+BIRD respectively. Similar to the observation above, we note that UniGen3 is able to sample for instances that timed out even for ApproxMC3. It is worth to recall that UniGen3 (and UniGen2) first makes a call to an approximate counter during its parameter search phase.

Remark 1. Since the runtime improvements of ApproxMC4 and UniGen3 are primarily due to improvements in the underlying SAT solver, it is worth pointing out, to put our contribution in context, that the difference between average PAR-2 scores of the top two solvers in a SAT competition is usually less than 100.

Benchmark	Vars	Cls	ApproxMC3 time (s)	ApproxMC4 time (s)	UniGen2 +BIRD 500 samples time (s)	UniGen3 500 samples time (s)
or-70-5-1-UC-20	140	350	6.03	2.07	14.21	6.08
prod-4	7497	37358	56.65	7.09	171.57	36.54
min-8	1545	4230	152.53	5.58	471.47	35.04
parity.sk_11_11	13116	47506	389.26	436.32	705.85	809
leader_sync4_11	205198	129149	346.4	20.55	1019.09	106.93
blasted_TR_b12_2	2426	8373	308.08	20.46	1218.01	546.62
hash-8-6	377545	1517574	462.28	266.59	1321.91	633.84
s15850a_15_7	10995	24836	1206.17	31.69	2782.96	230.17
ConcreteRole	395951	1520924	1694.19	309.07	3083.99	923.69
tire-3	577	2004	3059.19	233.28	3876.03	797.42
04B-2	19510	86961	1860.97	625.81	TO	2236.31
blasted_case138	849	2253	TO	3691.9	TO	TO
hash-11-4	518449	2082039	4602.95	4043.4	TO	TO
karatsuba.sk_7_41	19594	82417	3192.85	3410.36	TO	TO
log-3	1413	29487	TO	123.15	TO	408.25
modexp8-8-6	167793	633614	4439.21	TO	TO	TO
or-100-5-6-UC-20	200	500	TO	1689.47	TO	4898.43
prod-28	52233	261422	TO	235.02	TO	1053.9
s38417_15_7	25615	57946	TO	187.71	TO	TO
signedAvg	30335	91854	TO	114.15	TO	582.01

Table 1. Performance comparison of ApproxMC3 vis-a-vis ApproxMC4 and UniGen2+BIRD vis-a-vis UniGen3. TO indicates timeout after 5000 seconds or out of memory. Notice that on many problems that used to time out even for counting, we can now confidently sample.

6.2 Quality and Correctness

Quality of counting. To evaluate the quality of approximation we follow the same approach as Soos and Meel [19] and compare the approximate counts returned by ApproxMC4 with the counts computed by an exact model counter, namely DSharp[6]. The approximate counts and the exact counts are used to compute the observed tolerance ε_{obs}, which is defined as $\max(\frac{|sol(F)_{\downarrow S}|}{\text{AprxCount}} - 1, \frac{\text{AprxCount}}{|sol(F)_{\downarrow S}|} - 1)$, where AprxCount is the estimate computed by ApproxMC4 for a formula F and a sampling set S, which are both given for each benchmark. Note that, using ε_{obs}, we can rewrite the theoretical (ε, δ)-guarantee to $\mathsf{Pr}[\varepsilon_{obs} \leq \varepsilon] \geq 1 - \delta$ and hence we expect that ε_{obs} is mostly below $\varepsilon = 0.8$. The observed tolerance ε_{obs} over all benchmarks is shown in Figure 4. We observe a maximal value for ε_{obs} of 0.3333 and the the arithmetic mean of ε_{obs} across all benchmarks is 0.0411. Hence, the approximate counts are much closer to the exact counts than is theoretically guaranteed.

[6] DSharp is used because of its ability to handle sampling sets.

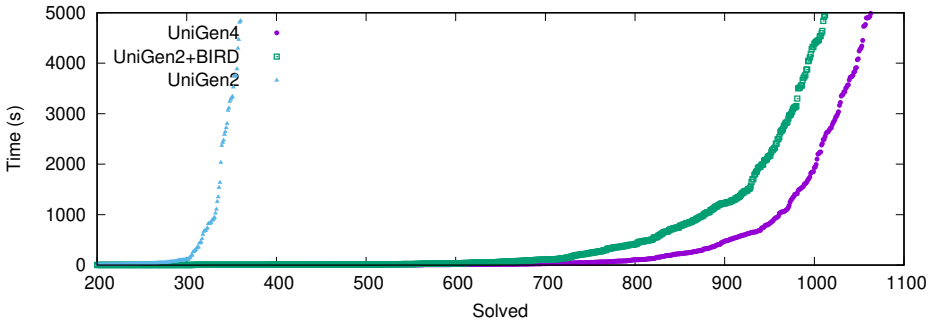

Fig. 3. Sampling performance of UniGen2 and UniGen2+BIRD versus UniGen3.

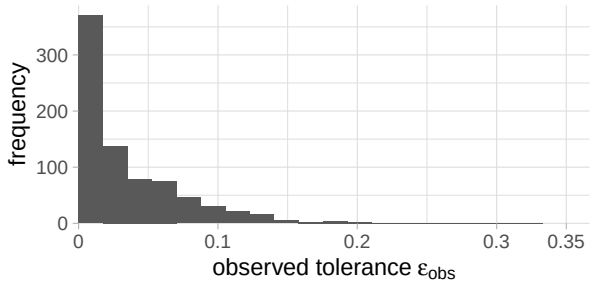

Fig. 4. The histogram of the observed tolerance ε_{obs} shows that the approximate counts are very close to the exact counts.

Quality of sampling. To evaluate the quality of sampling, we employed the uniformity tester, Barbarik, proposed by Chakraborty and Meel [5]. To this end, we selected 35 benchmarks from the pool of benchmarks employed by Chakraborty and Meel in their work and we tested UniGen3 for all the 35 benchmarks. We observed that Barbarik accepts UniGen3 for all the 35 instances, thereby providing a certificate for uniformity. We refer the reader to [5] for detailed discussion of the guarantees provided by Barbarik. Keeping in line with past work on sampling that tries to demonstrate the quality of sampling on a representative benchmark where exact uniform sampling is feasible via enumeration-based techniques, we chose the CNF instance *blasted_case110* (287 variables and 16384 solutions), which has been chosen in the previous studies as well. To this end, we implemented a simple ideal uniform sampler, denoted by US henceforth, by enumerating all the solutions and then picking a solution uniformly at random. We then generate $4,039,266$ samples from both UniGen3 and US. In each case, the number of times various witnesses were generated was recorded, yielding a distribution of the counts. Figure 5 shows the distributions of counts generated versus # of solutions. The x-axis represents counts and the y-axis represents the number of witnesses appearing the specified number of times. Thus, the point (230,212) represents

the fact that each of 212 distinct witnesses were generated 230 times among the 4,039,266 samples. While UniGen3 provides guarantees of almost-uniformity only, the two distributions are statistically indistinguishable. In particular, the KL divergence [15] of the distribution by UniGen from that of US is 0.003989.

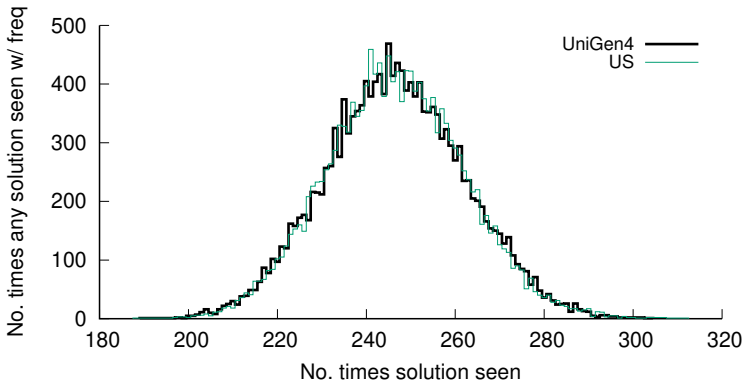

Fig. 5. Distribution of solution recurrence as generated by UniGen3 and US for the CNF `blasted_case110.cnf`.

7 Conclusions

We investigated the bottlenecks of CNF-XOR solving in the context of hashing-based approximate model counting and almost uniform sampling as implemented in ApproxMC3 and UniGen2 respectively. In this paper, we proposed five technical improvements, as follows: (1) detaching the clausal representation of XOR constraints from unit propagation, (2) lazy reason generation for XOR constraints, (3) bit-level parallelism for XOR constraint propagation, (4) partial solution extraction only covering the sampling set and (5) solution reuse. These improvements were incorporated into the new framework BIRD2, which led to the construction of improved approximate model counter ApproxMC4 and almost uniform sampler UniGen3. Experiments over a large set of benchmarks from various domains clearly show an improvement in running time and 77 more problems could be solved for counting and 51 more for sampling.

Acknowledgments We are grateful to Yash Pote for the early discussions of solution reuse. Work done in part while the second author visited NUS.

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The second author was funded by the Swedish Research Council (VR) grant 2016-00782. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg

References

1. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019. pp. 1249–1264 (2019)
2. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. Information and Computation 163(2), 510–526 (2000)
3. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT. LNCS, vol. 6175, pp. 44–57. Springer (2010)
4. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. In: ACM Symposium on Theory of Computing. pp. 106–112. ACM (1977)
5. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: Proceedings of AAAI Conference on Artificial Intelligence (AAAI) (1 2019)
6. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform sat witness generation. In: Proc. of TACAS. pp. 304–319 (2015)
7. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A Scalable and Nearly Uniform Generator of SAT Witnesses. In: Proc. of CAV. pp. 608–623 (2013)
8. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Proc. of CP. pp. 200–216 (2013)
9. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proc. of DAC. pp. 1–6 (2014)
10. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In: Proc. of IJCAI (2016)
11. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proc. of AAAI (2017)
12. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Proc. of SAT. pp. 61–75 (2005)
13. Han, C., Jiang, J.R.: When boolean satisfiability meets Gaussian elimination in a Simplex way. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification, CAV. LNCS, vol. 7358, pp. 410–426. Springer (2012)
14. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. Theoretical Computer Science 43(2-3), 169–188 (1986)
15. Kullback, S., Leibler, R.A.: On information and sufficiency. Ann. Math. Statist. 22(1), 79–86 (03 1951)
16. Meel, K.S.: Model counting and uniform sampling instances (May 2020), https://doi.org/10.5281/zenodo.3793090
17. Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1995)

18. Roth, D.: On the hardness of approximate reasoning. Artificial Intelligence 82(1), 273–302 (1996)
19. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI Conference on Artificial Intelligence, AAAI. pp. 1592–1599. AAAI Press (2019)
20. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Proc. of SAT (2009)
21. Stockmeyer, L.: The complexity of approximate counting. In: Proc. of STOC. pp. 118–126 (1983)
22. Toda, S.: On the computational power of PP and (+)P. In: Proc. of FOCS. pp. 514–519. IEEE (1989)
23. Vadhan, S.P., et al.: Pseudorandomness. Foundations and Trends® in Theoretical Computer Science 7(1–3), 1–336 (2012)
24. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing 8(3), 410–421 (1979)