Approximate Probabilistic Inference via Word-Level Counting * †

Supratik Chakraborty
Indian Institute of Technology,
Bombay

Kuldeep S. Meel
Department of Computer Science,
Rice University

Rakesh Mistry
Indian Institute of Technology,
Bombay

Moshe Y. Vardi
Department of Computer Science,
Rice University

Abstract

Hashing-based model counting has emerged as a promising approach for large-scale probabilistic inference on graphical models. A key component of these techniques is the use of xor-based 2-universal hash functions that operate over Boolean domains. Many counting problems arising in probabilistic inference are, however, naturally encoded over finite discrete domains. Techniques based on bit-level (or Boolean) hash functions require these problems to be propositionalized, making it impossible to leverage the remarkable progress made in SMT (Satisfiability Modulo Theory) solvers that can reason directly over words (or bit-vectors). In this work, we present the first approximate model counter that uses word-level hashing functions, and can directly leverage the power of sophisticated SMT solvers. Empirical evaluation over an extensive suite of benchmarks demonstrates the promise of the approach.

1 Introduction

Probabilistic inference on large and uncertain data sets is increasingly being used in a wide range of applications. It is well-known that probabilistic inference is polynomially inter-reducible to model counting (Roth 1996). In a recent line of work, it has been shown (Chakraborty, Meel, and Vardi 2013; Chakraborty et al. 2014; Ermon et al. 2013; Ivrii et al. 2015) that one can strike a fine balance between performance and approximation guarantees for propositional model counting, using 2-universal hash functions (Carter and Wegman 1977) on Boolean domains. This has propelled the model-counting formulation to emerge as a promising "assembly language" (Belle, Passerini, and Van den Broeck 2015) for inferencing in probabilistic graphical models.

In a large class of probabilistic inference problems, an important case being lifted inference on first order representations (Kersting 2012), the values of variables come from finite but large (exponential in the size of the representation) domains. Data values coming from such domains are naturally encoded as fixed-width words, where the width is logarithmic in the size of the domain. Conditions on observed values are, in turn, encoded as word-level constraints, and the corresponding model-counting problem asks one to count the number of solutions of a word-level constraint. It is therefore natural to ask if the success of approximate propositional model counters can be replicated at the word-level.

The balance between efficiency and strong guarantees of hashing-based algorithms for approximate propositional model counting crucially depends on two factors: (i) use of XOR-based 2-universal bit-level hash functions, and (ii) use of state-of-the-art propositional satisfiability solvers, viz. CryptoMiniSAT (Soos, Nohl, and Castelluccia 2009), that can efficiently reason about formulas that combine disjunctive clauses with XOR clauses.

In recent years, the performance of SMT (Satisfiability Modulo Theories) solvers has witnessed spectacular improvements (Barrett et al. 2012). Indeed, several highly optimized SMTsolvers for fixed-width words are now available in the public domain (Brummayer and Biere 2009; Jha, Limaye, and Seshia 2009; Hadarean et al. 2014; De Moura and Bjørner 2008). Nevertheless, 2-universal hash functions for fixed-width words that are also amenable to efficient reasoning by SMT solvers have hitherto not been studied. The reasoning power of SMTsolvers for fixed-width words has therefore remained untapped for word-level model counting. Thus, it is not surprising that all existing work on probabilistic inference using model counting (viz. (Chistikov, Dimitrova, and Majumdar 2015; Belle, Passerini, and Van den Broeck 2015; Ermon et al. 2013)) effectively reduce the problem to propositional model counting. Such approaches are similar to "bit blasting" in SMT solvers (Kroening and Strichman 2008).

The primary contribution of this paper is an efficient word-level approximate model counting algorithm SMTApproxMC that can be employed to answer inference queries over high-dimensional discrete domains. Our algorithm uses a new class of word-level hash functions that are 2-universal and can be solved by word-level SMTsolvers capable of reasoning about linear equalities on words. Therefore, unlike previous works, SMTApproxMC is able to leverage the power of sophisticated SMT solvers.

To illustrate the practical utility of SMTApproxMC, we implemented a prototype and evaluated it on a

*The author list has been sorted alphabetically by last name; this should not be used to determine the extent of authors' contributions.

†The full version is available at http://arxiv.org/abs/1511.07663

suite of benchmarks. Our experiments demonstrate that SMTApproxMC can significantly outperform the prevalent approach of bit-blasting a word-level constraint and using an approximate propositional model counter that employs XOR-based hash functions. Our proposed word-level hash functions embed the domain of all variables in a large enough finite domain. Thus, one would not expect our approach to work well for constraints that exhibit a hugely heterogeneous mix of word widths, or for problems that are difficult for word-level SMT solvers. Indeed, our experiments suggest that the use of word-level hash functions provides significant benefits when the original word-level constraint is such that (i) the words appearing in it have long and similar widths, and (ii) the SMTsolver can reason about the constraint at the word-level, without extensive bit-blasting.

2 Preliminaries

A *word* (or *bit-vector*) is an array of bits. The size of the array is called the *width* of the word. We consider here *fixed-width* words, whose width is a constant. It is easy to see that a word of width k can be used to represent elements of a set of size 2^k. The first-order theory of fixed-width words has been extensively studied (see (Kroening and Strichman 2008; Bruttomesso 2008) for an overview). The vocabulary of this theory includes interpreted predicates and functions, whose semantics are defined over words interpreted as signed integers, unsigned integers, or vectors of propositional constants (depending on the function or predicate). When a word of width k is treated as a vector, we assume that the component bits are indexed from 0 through $k-1$, where index 0 corresponds to the rightmost bit. A *term* is either a word-level variable or constant, or is obtained by applying functions in the vocabulary to a term. Every term has an associated width that is uniquely defined by the widths of word-level variables and constants in the term, and by the semantics of functions used to build the term. For purposes of this paper, given terms t_1 and t_2, we use $t_1 + t_2$ (resp. $t_1 * t_2$) to denote the sum (resp. product) of t_1 and t_2, interpreted as unsigned integers. Given a positive integer p, we use $t_1 \mod p$ to denote the remainder after dividing t_1 by p. Furthermore, if t_1 has width k, and a and b are integers such that $0 \leq a \leq b < k$, we use $\mathsf{extract}(t_1, a, b)$ to denote the slice of t_1 (interpreted as a vector) between indices a and b, inclusively.

Let F be a formula in the theory of fixed-width words. The *support* of F, denoted $\mathsf{sup}(F)$, is the set of word-level variables that appear in F. A *model* or *solution* of F is an assignment of word-level constants to variables in $\mathsf{sup}(F)$ such that F evaluates to True. We use R_F to denote the set of *models* of F. The model-counting problem requires us to compute $|R_F|$. For simplicity of exposition, we assume henceforth that all words in $\mathsf{sup}(F)$ have the same width. Note that this is without loss of generality, since if k is the maximum width of all words in $\mathsf{sup}(F)$, we can construct a formula $\widehat{F}$ such that the following hold: (i) $|\mathsf{sup}(F)| = |\mathsf{sup}(\widehat{F})|$, (ii) all word-level variables in $\widehat{F}$ have width k, and (iii) $|R_F| = |R_{\widehat{F}}|$. The formula $\widehat{F}$ is obtained by replacing every occurrence of word-level variable x having width m $(< k)$ in F with $\mathsf{extract}(\widehat{x}, 0, m-1)$, where $\widehat{x}$ is a new variable of width k.

We write $\mathsf{Pr}[X : \mathcal{P}]$ for 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.

Given a word-level formula F, an *exact model counter* returns $|R_F|$. An *approximate model counter* relaxes this requirement to some extent: given a *tolerance* $\varepsilon > 0$ and *confidence* $1 - \delta \in (0, 1]$, the value v returned by the counter satisfies $\mathsf{Pr}[\frac{|R_F|}{1+\varepsilon} \leq v \leq (1+\varepsilon)|R_F|] \geq 1 - \delta$. Our model-counting algorithm belongs to the class of approximate model counters.

Special classes of hash functions, called 2-*wise independent universal* hash functions play a crucial role in our work. Let $\mathsf{sup}(F) = \{x_0, \ldots x_{n-1}\}$, where each x_i is a word of width k. The space of all assignments of words in $\mathsf{sup}(F)$ is $\{0,1\}^{n.k}$. We use hash functions that map elements of $\{0,1\}^{n.k}$ to p bins labeled $0, 1, \ldots p-1$, where $1 \leq p < 2^{n.k}$. Let $\mathbb{Z}_p$ denote $\{0, 1, \ldots p-1\}$ and let $\mathcal{H}$ denote a family of hash functions mapping $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$. We use $h \xleftarrow{R} \mathcal{H}$ to denote the probability space obtained by choosing a hash function h uniformly at random from $\mathcal{H}$. We say that $\mathcal{H}$ is a 2-wise independent universal hash family if for all $\alpha_1, \alpha_2 \in \mathbb{Z}_p$ and for all distinct $\mathbf{X}_1, \mathbf{X}_2 \in \{0,1\}^{n.k}$, $\mathsf{Pr}\left[h(\mathbf{X}_1) = \alpha_1 \wedge h(\mathbf{X}_2) = \alpha_2 : h \xleftarrow{R} \mathcal{H}\right] = 1/p^2$.

3 Related Work

The connection between probabilistic inference and model counting has been extensively studied by several authors (Cooper 1990; Roth 1996; Chavira and Darwiche 2008), and it is known that the two problems are inter-reducible. Propositional model counting was shown to be #P-complete by Valiant (Valiant 1979). It follows easily that the model counting problem for fixed-width words is also #P-complete. It is therefore unlikely that efficient exact algorithms exist for this problem. (Bellare, Goldreich, and Petrank 2000) showed that a closely related problem, that of almost uniform sampling from propositional constraints, can be solved in probabilistic polynomial time using an NP oracle. Subsequently, (Jerrum, Valiant, and Vazirani 1986) showed that approximate model counting is polynomially inter-reducible to almost uniform sampling. While this shows that approximate model counting is solvable in probabilstic polynomial time relative to an NP oracle, the algorithms resulting from this largely theoretical body of work are highly inefficient in practice (Meel 2014).

Building on the work of Bellare, Goldreich and Petrank (2000), Chakraborty, Meel and Vardi (2013) proposed the first scalable approximate model counting algorithm for propositional formulas, called ApproxMC. Their technique is based on the use of a family of 2-universal bit-level hash functions that compute XOR of randomly chosen propositional variables. Similar bit-level hashing techniques were also used in (Ermon et al. 2013; Chakraborty et al. 2014) for weighted model counting. All of these works leverage the significant advances made in propositional satisfiability solving in the recent past (Biere et al. 2009).

Over the last two decades, there has been tremendous progress in the development of decision procedures, called Satisfiability Modulo Theories (or SMT) solvers, for combinations of first-order theories, including the theory of fixed-width words (Barrett, Fontaine, and Tinelli 2010; Barrett, Moura, and Stump 2005). An SMT solver uses a core propositional reasoning engine and decision procedures for individual theories, to determine the satisfiability of a formula in the combination of theories. It is now folklore that a well-engineered word-level SMT solver can significantly outperform the naive approach of *blasting* words into component bits and then using a propositional satisfiability solver (De Moura and Bjørner 2008; Jha, Limaye, and Seshia 2009; Bruttomesso et al. 2007). The power of word-level SMT solvers stems from their ability to reason about words directly (e.g. $a + (b - c) = (a - c) + b$ for every word a, b, c), instead of *blasting* words into component bits and using propositional reasoning.

The work of (Chistikov, Dimitrova, and Majumdar 2015) tried to extend ApproxMC (Chakraborty, Meel, and Vardi 2013) to non-propositional domains. A crucial step in their approach is to propositionalize the solution space (e.g. bounded integers are equated to tuples of propositions) and then use XOR-based bit-level hash functions. Unfortunately, such propositionalization can significantly reduce the effectiveness of theory-specific reasoning in an SMT solver. The work of (Belle, Passerini, and Van den Broeck 2015) used bit-level hash functions with the propositional abstraction of an SMT formula to solve the problem of *weighted model integration*. This approach also fails to harness the power of theory-specific reasoning in SMT solvers.

Recently, (de Salvo Braz et al. 2015) proposed SGDPLL(T), an algorithm that generalizes SMT solving to do lifted inferencing and model counting (among other things) modulo background theories (denoted T). A fixed-width word model counter, like the one proposed in this paper, can serve as a theory-specific solver in the SGDPLL(T) framework. In addition, it can also serve as an alernative to SGDPLL(T) when the overall problem is simply to count models in the theory T of fixed-width words, There have also been other attempts to exploit the power of SMT solvers in machine learning. For example, (Teso, Sebastiani, and Passerini 2014) used optimizing SMT solvers for structured relational learning using Support Vector Machines. This is unrelated to our approach of harnessing the power of SMT solvers for probabilistic inference via model counting.

4 Word-level Hash Function

The performance of hashing-based techniques for approximate model counting depends crucially on the underlying family of hash functions used to partition the solution space. A popular family of hash functions used in propositional model counting is $\mathcal{H}_{xor}$, defined as the family of functions obtained by XOR-ing a random subset of propositional variables, and equating the result to either 0 or 1, chosen randomly. The family $\mathcal{H}_{xor}$ enjoys important properties like 2-independence and easy implementability, which make it ideal for use in practical model counters for propositional formulas (Gomes, Sabharwal, and Selman 2007; Ermon et al. 2013; Chakraborty, Meel, and Vardi 2013). Unfortunately, word-level universal hash families that are 2-independent, easily implementable and amenable to word-level reasoning by SMT solvers, have not been studied thus far. In this section, we present $\mathcal{H}_{SMT}$, a family of word-level hash functions that fills this gap.

As discussed earlier, let $\mathsf{sup}(F) = \{x_0, \ldots x_{n-1}\}$, where each x_i is a word of width k. We use $\mathbf{X}$ to denote the n-dimensional vector $(x_0, \ldots x_{n-1})$. The space of all assignments to words in $\mathbf{X}$ is $\{0,1\}^{n.k}$. Let p be a prime number such that $2^k \leq p < 2^{n.k}$. Consider a family $\mathcal{H}$ of hash functions mapping $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$, where each hash function is of the form $h(\mathbf{X}) = (\sum_{j=0}^{n-1} a_j * x_j + b) \mod p$, and the a_j's and b are elements of $\mathbb{Z}_p$, represented as words of width $\lceil \log_2 p \rceil$. Observe that every $h \in \mathcal{H}$ partitions $\{0,1\}^{n.k}$ into p bins (or cells). Moreover, for every $\xi \in \{0,1\}^{n.k}$ and $\alpha \in \mathbb{Z}_p$, $\mathsf{Pr}\left[h(\xi) = \alpha : h \xleftarrow{R} \mathcal{H}\right] = p^{-1}$. For a hash function chosen uniformly at random from $\mathcal{H}$, the expected number of elements per cell is $2^{n.k}/p$. Since $p < 2^{n.k}$, every cell has at least 1 element in expectation. Since $2^k \leq p$, for every word x_i of width k, we also have $x_i \mod p = x_i$. Thus, distinct words are not aliased (or made to behave similarly) because of modular arithmetic in the hash function.

Suppose now we wish to partition $\{0.1\}^{n.k}$ into p^c cells, where $c > 1$ and $p^c < 2^{n.k}$. To achieve this, we need to define hash functions that map elements in $\{0,1\}^{n.k}$ to a tuple in $(\mathbb{Z}_p)^c$. A simple way to achieve this is to take a c-tuple of hash functions, each of which maps $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$. Therefore, the desired family of hash functions is simply the iterated Cartesian product $\mathcal{H} \times \cdots \times \mathcal{H}$, where the product is taken c times. Note that every hash function in this family is a c-tuple of hash functions. For a hash function chosen uniformly at random from this family, the expected number of elements per cell is $2^{n.k}/p^c$.

An important consideration in hashing-based techniques for approximate model counting is the choice of a hash function that yields cells that are neither too large nor too small in their expected sizes. Since increasing c by 1 reduces the expected size of each cell by a factor of p, it may be difficult to satisfy the above requirement if the value of p is large. At the same time, it is desirable to have $p > 2^k$ to prevent aliasing of two distinct words of width k. This motivates us to consider more general classes of word-level hash functions, in which each word x_i can be split into thinner slices, effectively reducing the width k of words, and allowing us to use smaller values of p. We describe this in more detail below.

Assume for the sake of simplicity that k is a power of 2, and let q be $\log_2 k$. For every $j \in \{0, \ldots q-1\}$ and for every $x_i \in \mathbf{X}$, define $\mathbf{x_i}^{(j)}$ to be the 2^j-dimensional vector of slices of the word x_i, where each slice is of width $k/2^j$. For example, the two slices in $\mathbf{x_1}^{(1)}$ are $\mathsf{extract}(x_1, 0, k/2-1)$ and $\mathsf{extract}(x_1, k/2, k-1)$. Let $\mathbf{X}^{(j)}$ denote the $n.2^j$-dimensional vector $(\mathbf{x_0}^{(j)}, \mathbf{x_1}^{(j)}, \ldots \mathbf{x_{n-1}}^{(j)})$. It is easy to see that the m^{th} component of $\mathbf{X}^{(j)}$, denoted $\mathbf{X}_m^{(j)}$, is $\mathsf{extract}(x_i, s, t)$, where $i = \lfloor m/2^j \rfloor$, $s = (m \mod 2^j) \cdot (k/2^j)$ and $t = s + (k/2^j) - 1$. Let p_j denote the smallest prime larger than or equal to $2^{(k/2^j)}$. Note that this implies $p_{j+1} \leq p_j$ for all $j \geq 0$. In order to obtain a family of hash functions that maps $\{0,1\}^{n.k}$ to $\mathbb{Z}_{p_j}$, we split each word x_i into slices of width $k/2^j$, treat these slices as words of reduced width, and use a technique similar to the one used above to map $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$. Specifically, the family $\mathcal{H}^{(j)} = \left\{h^{(j)} : h^{(j)}(\mathbf{X}) = \left(\sum_{m=0}^{n.2^j-1} a_m^{(j)} * \mathbf{X}_m^{(j)} + b^{(j)}\right) \mod p_j\right\}$ maps $\{0,1\}^{n.k}$ to $\mathbb{Z}_{p_j}$, where the values of $a_m^{(j)}$ and $b^{(j)}$ are chosen from $\mathbb{Z}_{p_j}$, and represented as $\lceil \log_2 p_j \rceil$-bit words.

In general, we may wish to define a family of hash functions that maps $\{0,1\}^{n.k}$ to $\mathcal{D}$, where $\mathcal{D}$ is given by $(\mathbb{Z}_{p_0})^{c_0} \times (\mathbb{Z}_{p_1})^{c_1} \times \cdots (\mathbb{Z}_{p_{q-1}})^{c_{q-1}}$ and $\prod_{j=0}^{q-1} p_j^{c_j} < 2^{n.k}$. To achieve this, we first consider the iterated Cartesian product of $\mathcal{H}^{(j)}$ with itself c_j times, and denote it by $\left(\mathcal{H}^{(j)}\right)^{c_j}$, for every $j \in \{0, \ldots q-1\}$. Finally, the desired family of hash functions is obtained as $\prod_{j=0}^{q-1} \left(\mathcal{H}^{(j)}\right)^{c_j}$. Observe that every hash function h in this family is a $\left(\sum_{l=0}^{q-1} c_l\right)$-tuple of hash functions. Specifically, the r^{th} component of h, for $r \leq \left(\sum_{l=0}^{q-1} c_l\right)$, is given by $\left(\sum_{m=0}^{n.2^j-1} a_m^{(j)} * \mathbf{X}_m^{(j)} + b^{(j)}\right) \mod p_j$, where $\left(\sum_{i=0}^{j-1} c_i\right) < r \leq \left(\sum_{i=0}^{j} c_i\right)$, and the $a_m^{(j)}$s and $b^{(j)}$ are elements of $\mathbb{Z}_{p_j}$.

The case when k is not a power of 2 is handled by splitting the words x_i into slices of size $\lceil k/2 \rceil$, $\lceil k/2^2 \rceil$ and so on. Note that the family of hash functions defined above depends only on n, k and the vector $C = (c_0, c_1, \ldots c_{q-1})$, where $q = \lceil \log_2 k \rceil$. Hence, we call this family $\mathcal{H}_{SMT}(n,k,C)$. Note also that by setting c_i to 0 for all $i \neq \lfloor \log_2(k/2) \rfloor$, and c_i to r for $i = \lfloor \log_2(k/2) \rfloor$ reduces $\mathcal{H}_{SMT}$ to the family $\mathcal{H}_{xor}$ of XOR-based bit-wise hash functions mapping $\{0,1\}^{n.k}$ to $\{0,1\}^r$. Therefore, H_{SMT} strictly generalizes $\mathcal{H}_{xor}$.

We summarize below important properties of the $\mathcal{H}_{SMT}(n,k,C)$ class. All proofs are available in (Chakraborty et al. 2015).

Lemma 1. *For every* $\mathbf{X} \in \{0,1\}^{n.k}$ *and every* $\alpha \in \mathcal{D}$, $\Pr[h(\mathbf{X}) = \alpha \mid h \xleftarrow{R} \mathcal{H}_{SMT}(n,k,C)] = \prod_{j=0}^{|C|-1} {p_j}^{-c_j}$

Theorem 1. *For every* $\alpha_1, \alpha_2 \in \mathcal{D}$ *and every distinct* $\mathbf{X}_1, \mathbf{X}_2 \in \{0,1\}^{n.k}$, $\Pr[(h(\mathbf{X}_1) = \alpha_1 \wedge h(\mathbf{X}_2) = \alpha_2) \mid h \xleftarrow{R} \mathcal{H}_{SMT}(n,k,C)] = \prod_{j=0}^{|C|-1} (p_j)^{-2.c_j}$. *Therefore,* $\mathcal{H}_{SMT}(n,k,C)$ *is pairwise independent.*

Gaussian Elimination The practical success of XOR-based bit-level hashing techniques for propositional model counting owes a lot to solvers like CryptoMiniSAT (Soos, Nohl, and Castelluccia 2009) that use Gaussian Elimination to efficiently reason about XOR constraints. It is significant that the constraints arising from $\mathcal{H}_{SMT}$ are linear modular equalities that also lend themselves to efficient Gaussian Elimination. We believe that integration of Gaussian Elimination engines in SMT solvers will significantly improve the performance of hashing-based word-level model counters.

5 Algorithm

We now present SMTApproxMC, a word-level hashing-based approximate model counting algorithm. SMTApproxMC takes as inputs a formula F in the theory of fixed-width words, a tolerance ε (> 0), and a confidence $1-\delta \in (0,1]$. It returns an estimate of $|R_F|$ within the tolerance ε, with confidence $1-\delta$. The formula F is assumed to have n variables, each of width k, in its support. The central idea of SMTApproxMC is to randomly partition the solution space of F into "small" cells of roughly the same size, using word-level hash functions from $\mathcal{H}_{SMT}(n,k,C)$, where C is incrementally computed. The check for "small"-ness of cells is done using a word-level SMT solver. The use of word-level hash functions and a word-level SMT solver allows us to directly harness the power of SMT solving in model counting.

The pseudocode for SMTApproxMC is presented in Algorithm 1. Lines 1– 3 initialize the different parameters. Specifically, pivot determines the maximum size of a "small" cell as a function of ε, and t determines the number of times SMTApproxMCCore must be invoked, as a function of δ. The value of t is determined by technical arguments in the proofs of our theoretical guarantees, and is not based on experimental observations Algorithm SMTApproxMCCore lies at the heart of SMTApproxMC. Each invocation of SMTApproxMCCore either returns an approximate model count of F, or $\perp$ (indicating a failure). In the former case, we collect the returned value, m, in a list M in line 8. Finally, we compute the median of the approximate counts in M, and return this as $\mathrm{FinalCount}$.

Algorithm 1 SMTApproxMC$(F, \varepsilon, \delta, k)$

1: counter $\leftarrow 0$; $M \leftarrow$ emptyList;
2: pivot $\leftarrow 2 \times \lceil e^{3/2} \left(1 + \frac{1}{\varepsilon}\right)^2 \rceil$;
3: $t \leftarrow \lceil 35 \log_2(3/\delta) \rceil$;
4: **repeat**
5:     $m \leftarrow$ SMTApproxMCCore(F, pivot, k);
6:     counter $\leftarrow$ counter $+ 1$;
7:     **if** $m \neq \perp$ **then**
8:         AddToList(M, m);
9: **until** (counter $< t$)
10: FinalCount $\leftarrow$ FindMedian(M);
11: **return** FinalCount;

The pseudocode for SMTApproxMCCore is shown in Algorithm 2. This algorithm takes as inputs a word-level SMT formula F, a threshold pivot, and the width k of words in $\mathsf{sup}(F)$. We assume access to a subroutine BoundedSMT that accepts a word-level SMT formula φ and a threshold pivot as inputs, and returns pivot $+ 1$ solutions of φ if $|R_\varphi| >$ pivot; otherwise it returns R_φ. In lines 1– 2 of Algorithm 2, we return the exact count if $|R_F| \leq$ pivot. Otherwise, we initialize C by setting $C[0]$ to 0 and $C[1]$ to 1, where $C[i]$ in the pseudocode refers to c_i in the previous section's discussion. This choice of initialization is motivated by our experimental observations. We also count the number of cells generated by an arbitrary hash function from

Algorithm 2 SMTApproxMCCore(F, pivot, k)

1: $Y \leftarrow$ BoundedSMT(F, pivot);
2: **if** $|Y| \leq$ pivot) **then return** $|Y|$;
3: **else**
4: $\quad C \leftarrow$ emptyVector; $C[0] \leftarrow 0$; $C[1] \leftarrow 1$;
5: $\quad i \leftarrow 1$; numCells $\leftarrow p_1$;
6: $\quad$ **repeat**
7: $\quad\quad$ Choose h at random from $\mathcal{H}_{SMT}(n, k, C)$;
8: $\quad\quad$ Choose α at random from $\prod_{j=0}^{i} \left(\mathbb{Z}_{p_j}\right)^{C[j]}$;
9: $\quad\quad$ $Y \leftarrow$ BoundedSMT($F \wedge (h(\mathbf{X}) = \alpha)$, pivot);
10: $\quad\quad$ **if** ($|Y| >$ pivot) **then**
11: $\quad\quad\quad C[i] \leftarrow C[i] + 1$;
12: $\quad\quad\quad$ numCells $\leftarrow$ numCells $\times\, p_i$;
13: $\quad\quad$ **if** ($|Y| = 0$) **then**
14: $\quad\quad\quad$ **if** $p_i > 2$ **then**
15: $\quad\quad\quad\quad C[i] \leftarrow C[i] - 1$;
16: $\quad\quad\quad\quad i \leftarrow i + 1$; $C[i] \leftarrow 1$;
17: $\quad\quad\quad\quad$ numCells $\leftarrow$ numCells $\times\, (p_{i+1}/p_i)$;
18: $\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad$ break;
20: $\quad$ **until** ($(0 < |Y| \leq$ pivot) or (numCells $> 2^{n.k}$))
21: $\quad$ **if** (($|Y| >$ pivot) **or** ($|Y| = 0$)) **then return** $\perp$;
22: $\quad$ **else return** $|Y| \times$ numCells;

$\mathcal{H}_{SMT}(n, k, C)$ in numCells. The loop in lines 6–20 iteratively partitions R_F into cells using randomly chosen hash functions from $\mathcal{H}_{SMT}(n, k, C)$. The value of i in each iteration indicates the extent to which words in the support of F are sliced when defining hash functions in $\mathcal{H}_{SMT}(n, k, C)$ – specifically, slices that are $\lceil k/2^i \rceil$-bits or more wide are used. The iterative partitioning of R_F continues until a randomly chosen cell is found to be "small" (i.e. has ≥ 1 and $\leq$ pivot solutions), or the number of cells exceeds $2^{n.k}$, rendering further partitioning meaningless. The random choice of h and α in lines 7 and 8 ensures that we pick a random cell. The call to BoundedSMT returns at most pivot$+1$ solutions of F within the chosen cell in the set Y. If $|Y| >$ pivot, the cell is deemed to be large, and the algorithm partitions each cell further into p_i parts. This is done by incrementing $C[i]$ in line 11, so that the hash function chosen from $\mathcal{H}_{SMT}(n, k, C)$ in the next iteration of the loop generates p_i times more cells than in the current iteration. On the other hand, if Y is empty and $p_i > 2$, the cells are too small (and too many), and the algorithm reduces the number of cells by a factor of p_{i+1}/p_i (recall $p_{i+1} \leq p_i$) by setting the values of $C[i]$ and $C[i+1]$ accordingly (see lines15 –17). If Y is nonempty and has no more than pivot solutions, the cells are of the right size, and we return the estimate $|Y| \times$ numCells. In all other cases, SMTApproxMCCore fails and returns $\perp$.

Similar to the analysis of ApproxMC (Chakraborty, Meel, and Vardi 2013), the current theoretical analysis of SMTApproxMC assumes that for some C during the execution of SMTApproxMCCore, $\log |R_F| - \log(\text{numCells}) - 1 = \log(pivot)$. We leave analysis of SMTApproxMC without above assumption to future work. The following theorems concern the correctness and performance of SMTApproxMC.

Theorem 2. *Suppose an invocation of* SMTApproxMC($F, \varepsilon, \delta, k$) *returns* FinalCount. *Then* $\Pr\left[(1+\varepsilon)^{-1}|R_F| \leq \text{FinalCount} \leq (1+\varepsilon)|R_F|\right] \geq 1-\delta$

Theorem 3. SMTApproxMC($F, \varepsilon, \delta, k$) *runs in time polynomial in* $|F|$, $1/\varepsilon$ *and* $\log_2(1/\delta)$ *relative to an* NP*-oracle.*

The proofs of Theorem 2 and 3 can be found in (Chakraborty et al. 2015).

6 Experimental Methodology and Results

To evaluate the performance and effectiveness of SMTApproxMC, we built a prototype implementation and conducted extensive experiments. Our suite of benchmarks consisted of more than 150 problems arising from diverse domains such as reasoning about circuits, planning, program synthesis and the like. For lack of space, we present results for only for a subset of the benchmarks.

For purposes of comparison, we also implemented a state-of-the-art bit-level hashing-based approximate model counting algorithm for bounded integers, proposed by (Chistikov, Dimitrova, and Majumdar 2015). Henceforth, we refer to this algorithm as CDM, after the authors' initials. Both model counters used an overall timeout of 12 hours per benchmark, and a BoundedSMT timeout of 2400 seconds per call. Both used Boolector, a state-of-the-art SMT solver for fixed-width words (Brummayer and Biere 2009). Note that Boolector (and other popular SMT solvers for fixed-width words) does not yet implement Gaussian elimination for linear modular equalities; hence our experiments did not enjoy the benefits of Gaussian elimination. We employed the Mersenne Twister to generate pseudo-random numbers, and each thread was seeded independently using the Python `random` library. All experiments used $\varepsilon = 0.8$ and $\delta = 0.2$. Similar to ApproxMC, we determined value of t based on tighter analysis offered by proofs. For detailed discussion, we refer the reader to Section 6 in (Chakraborty, Meel, and Vardi 2013). Every experiment was conducted on a single core of high-performance computer cluster, where each node had a 20-core, 2.20 GHz Intel Xeon processor, with 3.2GB of main memory per core.

We sought answers to the following questions from our experimental evaluation:

1. How does the performance of SMTApproxMC compare with that of a bit-level hashing-based counter like CDM?
2. How do the approximate counts returned by SMTApproxMC compare with exact counts?

Our experiments show that SMTApproxMC significantly outperforms CDM for a large class of benchmarks. Furthermore, the counts returned by SMTApproxMC are highly accurate and the observed geometric tolerance(ε_{obs}) = 0.04.

Performance Comparison Table 1 presents the result of comparing the performance of SMTApproxMC vis-a-vis CDM on a subset of our benchmarks. In Table 1, column 1 gives the benchmark identifier, column 2 gives the sum of widths of all variables, column 3 lists the number of

Benchmark	Total Bits	Variable Types	# of Operations	SMTApproxMC time(s)	CDM time(s)
squaring27	59	{1: 11, 16: 3}	10	–	2998.97
squaring51	40	{1: 32, 4: 2}	7	3285.52	607.22
1160877	32	{8: 2, 16: 1}	8	2.57	44.01
1160530	32	{8: 2, 16: 1}	12	2.01	43.28
1159005	64	{8: 4, 32: 1}	213	28.88	105.6
1160300	64	{8: 4, 32: 1}	1183	44.02	71.16
1159391	64	{8: 4, 32: 1}	681	57.03	91.62
1159520	64	{8: 4, 32: 1}	1388	114.53	155.09
1159708	64	{8: 4, 32: 1}	12	14793.93	–
1159472	64	{8: 4, 32: 1}	8	16308.82	–
1159115	64	{8: 4, 32: 1}	12	23984.55	–
1159431	64	{8: 4, 32: 1}	12	36406.4	–
1160191	64	{8: 4, 32: 1}	12	40166.1	–

Table 1: Runtime performance of SMTApproxMC vis-a-vis CDM for a subset of benchmarks.

variables ($\mathrm{numVars}$) for each corresponding width (w) in the format $\{\mathrm{w} : \mathrm{numVars}\}$. To indicate the complexity of the input formula, we present the number of operations in the original SMT formula in column 4. The runtimes for SMTApproxMC and CDM are presented in columns 5 and column 6 respectively. We use "–" to denote timeout after 12 hours. Table 1 clearly shows that SMTApproxMC significantly outperforms CDM (often by 2-10 times) for a large class of benchmarks. In particular, we observe that SMTApproxMC is able to compute counts for several cases where CDM times out.

Benchmarks in our suite exhibit significant heterogeneity in the widths of words, and also in the kinds of word-level operations used. Propositionalizing all word-level variables eagerly, as is done in CDM, prevents the SMT solver from making full use of word-level reasoning. In contrast, our approach allows the power of word-level reasoning to be harnessed if the original formula F and the hash functions are such that the SMT solver can reason about them without bit-blasting. This can lead to significant performance improvements, as seen in Table 1. Some benchmarks, however, have heterogenous bit-widths and heavy usage of operators like $\mathsf{extract}(x, n_1, n_2)$ and/or word-level multiplication. It is known that word-level reasoning in modern SMT solvers is not very effective for such cases, and the solver has to resort to bit-blasting. Therefore, using word-level hash functions does not help in such cases. We believe this contributes to the degraded performance of SMTApproxMC vis-a-vis CDM in a subset of our benchmarks. This also points to an interesting direction of future research: to find the right hash function for a benchmark by utilizing SMT solver's architecture.

Quality of Approximation To measure the quality of the counts returned by SMTApproxMC, we selected a subset of benchmarks that were small enough to be bit-blasted and fed to sharpSAT (Thurley 2006) – a state-of-the-art exact model counter. Figure 1 compares the model counts computed by SMTApproxMC with the bounds obtained by scaling the exact counts (from sharpSAT) with the tolerance factor ($\varepsilon = 0.8$). The y-axis represents model counts on log-scale while the x-axis presents benchmarks ordered in ascending order of model counts. We observe that for *all* the benchmarks, SMTApproxMC computes counts within the tolerance. Furthermore, for each instance, we computed observed tolerance (ε_{obs}) as $\frac{\mathrm{count}}{|R_F|} - 1$, if count $\geq |R_F|$, and $\frac{|R_F|}{\mathrm{count}} - 1$ otherwise, where $|R_F|$ is computed by sharpSAT and count is computed by SMTApproxMC. We observe that the geometric mean of ε_{obs} across all the benchmarks is only 0.04 – far less (i.e. closer to the exact count) than the theoretical guarantee of 0.8.

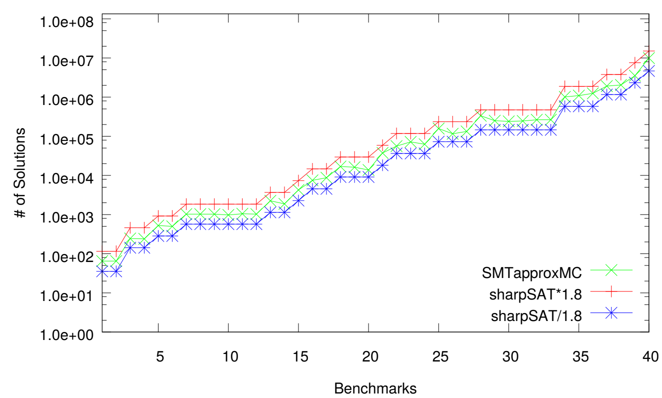

Figure 1: Quality of counts computed by SMTApproxMC vis-a-vis exact counts

7 Conclusions and Future Work

Hashing-based model counting has emerged as a promising approach for probabilistic inference on graphical models. While real-world examples naturally have word-level constraints, state-of-the-art approximate model counters effectively reduce the problem to propositional model counting due to lack of non-bit-level hash functions. In this work, we presented, $\mathcal{H}_{SMT}$, a word-level hash function and used it to build SMTApproxMC, an approximate word-level model counter. Our experiments show that SMTApproxMC can significantly outperform techniques based on bit-level hashing.

Our study also presents interesting directions for future work. For example, adapting SMTApproxMC to be aware of SMT solving strategies, and augmenting SMT solving strategies to efficiently reason about hash functions used in counting, are exciting directions of future work.

Our work goes beyond serving as a replacement for other approximate counting techniques. SMTApproxMC can also be viewed as an efficient building block for more sophisticated inference algorithms (de Salvo Braz et al. 2015). The development of SMT solvers has so far been primarily driven by the verification and static analysis communities. Our work hints that probabilistic inference could well be another driver for SMT solver technology development.

Acknowledgements

We thank Daniel Kroening for sharing his valuable insights on SMT solvers during the early stages of this project and Amit Bhatia for comments on early drafts of the paper. This work was supported in part by NSF grants IIS-1527668, CNS 1049862, CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, by a gift from Intel, by a grant from the Board of Research in Nuclear Sciences, India, Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097.

References

Barrett, C.; Deters, M.; de Moura, L.; Oliveras, A.; and Stump, A. 2012. 6 Years of SMT-COMP. *Journal of Automated Reasoning* 1–35.

Barrett, C.; Fontaine, P.; and Tinelli, C. 2010. The SMT-LIB standard - Version 2.5. `http://smtlib.cs.uiowa.edu/`.

Barrett, C.; Moura, L.; and Stump, A. 2005. SMT-COMP: Satisfiability Modulo Theories Competition. In *Proc. of CAV*, 20–23.

Bellare, M.; Goldreich, O.; and Petrank, E. 2000. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation* 163(2):510–526.

Belle, V.; Passerini, A.; and Van den Broeck, G. 2015. Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of IJCAI*, 2770–2776.

Biere, A.; Heule, M.; Van Maaren, H.; and Walsh, T. 2009. *Handbook of Satisfiability*. IOS Press.

Brummayer, R., and Biere, A. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. of TACAS*, 174–177.

Bruttomesso, R.; Cimatti, A.; Franzén, A.; Griggio, A.; Hanna, Z.; Nadel, A.; Palti, A.; and Sebastiani, R. 2007. A lazy and layered smt(bv) solver for hard industrial verification problems. In *Proc. of CAV*, 547–560.

Bruttomesso, R. 2008. *RTL Verification: From SAT to SMT(BV)*. Ph.D. Dissertation, DIT, University of Trento/FBK - Fondazione Bruno Kessler.

Carter, J. L., and Wegman, M. N. 1977. Universal classes of hash functions. In *Proc. of ACM symposium on Theory of computing*, 106–112. ACM.

Chakraborty, S.; Fremont, D. J.; Meel, K. S.; Seshia, S. A.; and Vardi, M. Y. 2014. Distribution-aware sampling and weighted model counting for SAT. In *Proc. of AAAI*, 1722–1730.

Chakraborty, S.; Meel, K. S.; Mistry, R.; and Vardi, M. Y. 2015. Approximate Probabilistic Inference via Word-Level Counting (Technical Report). http://arxiv.org/abs/1511.07663.

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

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172(6):772–799.

Chistikov, D.; Dimitrova, R.; and Majumdar, R. 2015. Approximate Counting for SMT and Value Estimation for Probabilistic Programs. In *Proc. of TACAS*, 320–334.

Cooper, G. F. 1990. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence* 42(2):393–405.

De Moura, L., and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proc. of TACAS*. Springer. 337–340.

de Salvo Braz, R.; O'Reilly, C.; Gogate, V.; and Dechter, R. 2015. Probabilistic inference modulo theories. In *Workshop on Hybrid Reasoning at IJCAI*.

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.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2007. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Proc. of NIPS*, 670–676.

Hadarean, L.; Bansal, K.; Jovanovic, D.; Barrett, C.; and Tinelli, C. 2014. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Proc. of CAV*, 680–695.

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.

Jerrum, M.; Valiant, L.; and Vazirani, V. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43(2-3):169–188.

Jha, S.; Limaye, R.; and Seshia, S. 2009. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, 668–674.

Kersting, K. 2012. Lifted probabilistic inference. In *Proc. of ECAI*, 33–38.

Kroening, D., and Strichman, O. 2008. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition.

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

Teso, S.; Sebastiani, R.; and Passerini, A. 2014. Structured learning modulo theories. *CoRR* abs/1405.1675.

Thurley, M. 2006. SharpSAT: counting models with advanced component caching and implicit BCP. In *Proc. of SAT*, 424–429.

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

Appendix

In this section, we provide proofs of various results stated previously. Our proofs borrow key ideas from (Bellare, Goldreich, and Petrank 2000; Chakraborty, Meel, and Vardi 2013; Gomes, Sabharwal, and Selman 2007); however, there are non-trivial adaptations specific to our work. We also provide extended versions of the experimental results reported in Section 6.

Detailed Proofs

Let $\mathcal{D}$ denote $(\mathbb{Z}_{p_0})^{c_0} \times (\mathbb{Z}_{p_1})^{c_1} \times \cdots (\mathbb{Z}_{p_{q-1}})^{c_{q-1}}$, where $\prod_{j=0}^{q-1} p_j^{c_j} < 2^{n.k}$. Let C denote the vector $(c_0, c_1, \ldots c_{q-1})$.

Lemma 1. *For every* $\mathbf{X} \in \{0,1\}^{n.k}$ *and every* $\alpha \in \mathcal{D}$, $\mathsf{Pr}[h(\mathbf{X}) = \alpha \mid h \xleftarrow{R} \mathcal{H}_{SMT}(n,k,C)] = \prod_{i=0}^{|C|-1} {p_i}^{-c_i}$

Proof. Let h_r, the r^{th} component of h, for $r \leq \left(\sum_{j=0}^{|C|-1} c_j\right)$, be given by $\left(\sum_{m=0}^{n.2^j-1} a_m^{(j)} * \mathbf{X}_m^{(j)} + b^{(j)}\right) \mod p_j$, where $\left(\sum_{i=0}^{j-1} c_i\right) < r \leq \left(\sum_{i=0}^{j} c_i\right)$, and the $a_m^{(j)}$s and $b^{(j)}$ are randomly and independently chosen elements of $\mathbb{Z}_{p_j}$, represented as words of width $\lceil \log_2 p_j \rceil$. Let $\mathcal{H}^{(j)}$ denote the family of hash functions of the form $\left(\sum_{m=0}^{n.2^j-1} u_m^{(j)} * \mathbf{X}_m^{(j)} + v^{(j)}\right) \mod p_j$, where $u_m^{(j)}$ and $v^{(j)}$ are elements of $\mathbb{Z}_{p_j}$. We use α_r to denote the rth component of α. For every choice of $\mathbf{X}$, $a_m^{(j)}$s and α_r, there is exactly one $b^{(j)}$ such that $h_r(\mathbf{X}) = \alpha_r$. Therefore, $\mathsf{Pr}[h_r(\mathbf{X}) = \alpha_r | h_r \xleftarrow{R} \mathcal{H}^{(j)}] = p_i^{-1}$.

Recall that every hash function h in $\mathcal{H}_{SMT}(n,k,C)$ is a $\left(\sum_{j=0}^{q-1} c_j\right)$-tuple of hash functions. Since h is chosen uniformly at random from $\mathcal{H}_{SMT}(n,k,C)$, the $\left(\sum_{j=0}^{q-1} c_j\right)$ components of h are effectively chosen randomly and independently of each other. Therefore, $\mathsf{Pr}[h(\mathbf{X}) = \alpha \mid h \xleftarrow{R} \mathcal{H}_{SMT}(n,k,C)] = \prod_{i=0}^{|C|-1} {p_i}^{-c_i}$ □

Theorem 1. *For every* $\alpha_1, \alpha_2 \in \mathcal{D}$ *and every distinct* $\mathbf{X}_1, \mathbf{X}_2 \in \{0,1\}^{n.k}$, $\mathsf{Pr}[(h(\mathbf{X}_1) = \alpha_1 \wedge h(\mathbf{X}_2) = \alpha_2) \mid h \xleftarrow{R} \mathcal{H}_{SMT}(n,k,C)] = \prod_{i=0}^{|C|-1} (p_i)^{-2.c_i}$. *Therefore,* $\mathcal{H}_{SMT}(n,k,C)$ *is pairwise independent.*

Proof. We know that $\mathsf{Pr}[(h(\mathbf{X}_1) = \alpha_1 \wedge h(\mathbf{X}_2) = \alpha_2)] = \mathsf{Pr}[h(\mathbf{X}_2) = \alpha_2 \mid h(\mathbf{X}_1) = \alpha_1] \times \mathsf{Pr}[h(\mathbf{X}_1) = \alpha_1]$. Theorem 1 implies that in order to prove pairwise independence of $\mathcal{H}_{SMT}(n,k,C)$, it is sufficient to show that $\mathsf{Pr}[h(\mathbf{X}_2) = \alpha_2 \mid h(\mathbf{X}_1) = \alpha_1] = \mathsf{Pr}[h(\mathbf{X}_2) = \alpha_2]$.

Since $h(\mathbf{X}) = \alpha$ can be viewed as conjunction of $\left(\sum_{j=0}^{q-1} c_j\right)$ ordered and *independent* constraints, it is sufficient to prove 2-wise independence for every ordered constraint. We now prove 2-wise independence for one of the ordered constraints below. Since the proof for the other ordered constraints can be obtained in exactly the same way, we omit their proofs.

We formulate a new hash function based on the first constraint as $g(\mathbf{X}) = (\left(\sum_{m=0}^{n.2^j-1} a_m^{(0)} * \mathbf{X}_m^{(0)} + b^{(0)}\right) \mod p_0$, where the $a_m^{(0)}$'s and $b^{(0)}$ are randomly and independently chosen elements of $\mathbb{Z}_{p_0}$, represented as words of width $\lceil \log_2 p_0 \rceil$. It is sufficient to show that $g(\mathbf{X})$ is 2-universal. This can be formally stated as $\mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0} \mid g(\mathbf{X}_1) = \alpha_{1,0}] = \mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0}]$, where $\alpha_{2,0}, \alpha_{1,0}$ are the 0^{th} components of α_2 and α_1 respectively. We consider two cases based on linear independence of $\mathbf{X}_1$ and $\mathbf{X}_2$.

- **Case 1:** $\mathbf{X}_1$ and $\mathbf{X}_2$ are linearly dependent. Without loss of generality, let $\mathbf{X}_1 = (0,0,0,\ldots 0)$ and $\mathbf{X}_2 = (r_1, 0, 0, \ldots 0)$ for some $r_1 \in \mathbb{Z}_{p_0}$, represented as a word. From $g(\mathbf{X}_1)$ we can deduce $b^{(0)}$. However for $g(\mathbf{X}_2) = \alpha_{2,0}$ we require $a_1^{(0)} * r_1 + b^{(0)} = \alpha_{2,0} \mod p_0$. Using Fermat's Little Theorem, we know that there exists a unique $a_1^{(0)}$ for every r_1 that satisfies the above equation. Therefore, therefore $\mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0} | g(\mathbf{X}_1) = \alpha_{1,0}] = \mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0}] = \frac{1}{p_0}$.

- **Case 2:** $\mathbf{X}_1$ and $\mathbf{X}_2$ are linearly independent. Since $2^k < p_0$, every component of $\mathbf{X}_1$ and $\mathbf{X}_2$ (i.e. an element of $\{0,1\}^k$) can be treated as an element of $\mathbb{Z}_{p_0}$. The space $\{0,1\}^{n.k}$ can therefore be thought of as lying within the vector space $(\mathbb{Z}_{p_0})^n$, and any $\mathbf{X} \in \{0,1\}^{n.k}$ can be written as a linear combination of the set of basis vectors over $(\mathbb{Z}_{p_0})^n$. It is therefore sufficient to prove pairwise independence when $\mathbf{X}_1$ and $\mathbf{X}_2$ are basis vectors. Without loss of generality, let $\mathbf{X}_1 = (r_1, 0, 0, \ldots 0)$ and $\mathbf{X}_2 = (0, r_2, 0, 0, \ldots 0)$ for some $r_1, r_2 \in \mathbb{Z}_{p_0}$. From $g(\mathbf{X}_1)$, we can deduce $\left(a_1^{(0)} * r_1 + b^{(0)} = \alpha_{1,0}\right) \mod p_0$. But since $a_1^{(0)}$ is randomly chosen, therefore $\mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0} \mid g(\mathbf{X}_1) = \alpha_{1,0}] = \mathsf{Pr}[(a_2^{(0)} * r_2 + \alpha_{1,0} - a_1^{(0)} * r_1 = \alpha_{2,0}) \mod p_0] = \mathsf{Pr}[(a_2^{(0)} * r_2 - a_1^{(0)} * r_1 = \alpha_{2,0} - \alpha_{1,0}) \mod p_0]$, where $-a$ refers to the additive inverse of a in the field $\mathbb{Z}_{p_0}$. Using Fermat's Little Theorem, we know that for every choice $a_1^{(0)}$ there exists a unique $a_2^{(0)}$ that satisfies the above requirement, given $\alpha_{1,0}$, $\alpha_{2,0}$, r_1 and r_2. Therefore $\mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0} \mid g(\mathbf{X}_1) = \alpha_{1,0}] = \frac{1}{p_0} = \mathsf{Pr}[g(\mathbf{X}_2) = \alpha_{2,0}]$.

□

Analysis of SMTApproxMC

For a given h and α, we use $R_{F,h,\alpha}$ to denote the set $R_F \cap h^{-1}(\alpha)$, i.e. the set of solutions of F that map to α under h. Let $\mathsf{E}[Y]$ and $\mathsf{V}[Y]$ represent expectation and variance of a random variable Y respectively. The analysis below focuses on the random variable $|R_{F,h,\alpha}|$ defined for a chosen α. We use μ to denote the expected value of the random variable $|R_{F,h,\alpha}|$ whenever h and α are clear from the context. The following lemma based on pairwise independence of $\mathcal{H}_{SMT}(n,k,C)$ is key to our analysis.

Lemma 2. *The random choice of* h *and* α *in* SMTApproxMCCore *ensures that for each* $\varepsilon > 0$, *we have* $\mathsf{Pr}\left[(1 - \frac{\varepsilon}{1+\varepsilon})\mu \leq |R_{F,h,\alpha}| \leq (1 + \frac{\varepsilon}{1+\varepsilon})\mu\right] \geq 1 - \frac{(1+\varepsilon)^2}{\varepsilon^2 \ \mu}$, *where* $\mu = \mathsf{E}[|R_{F,h,\alpha}|]$

Proof. For every $y \in \{0,1\}^{n.k}$ and for every $\alpha \in \prod_{i=0}^{|C|-1} (\mathbb{Z}_{p_i})^{C[i]}$, define an indicator variable $\gamma_{y,\alpha}$ as follows: $\gamma_{y,\alpha} = 1$ if $h(y) = \alpha$, and $\gamma_{y,\alpha} = 0$ otherwise. Let us fix α and y and choose h uniformly at random from $\mathcal{H}_{SMT}(n,k,C)$. The 2-wise independence $\mathcal{H}_{SMT}(n,k,C)$ implies that for every distinct $y_1, y_2 \in R_F$, the random variables $\gamma_{y_1}, \gamma_{y_2}$ are 2-wise independent. Let $|R_{F,h,\alpha}| = \sum_{y \in R_F} \gamma_{y,\alpha}$, $\mu = \mathsf{E}\left[|R_{F,h,\alpha}|\right]$ and $\mathsf{V}[|R_{F,h,\alpha}|] = \mathsf{V}[\sum_{y \in R_F} \gamma_{y,\alpha}]$. The pairwise independence of $\gamma_{y,\alpha}$ ensures that $\mathsf{V}[|R_{F,h,\alpha}|] = \sum_{y \in R_F} \mathsf{V}[\gamma_{y,\alpha}] \le \mu$. The result then follows from Chebyshev's inequality. □

Let Y be the set returned by $\mathsf{BoundedSMT}(F \wedge (h(\mathbf{X}) = \alpha), \text{pivot})$ where pivot is as calculated in Algorithm 1.

Lemma 3. $\mathsf{Pr}\left[(1+\varepsilon)^{-1}|R_F| \le |Y| \le (1+\varepsilon)|R_F| \mid \log() + \log(\text{pivot}) - 1 \le \log|R_F|\right] \ge 1 - \frac{e^{-3/2}}{\log|R_F| - - \log(\text{pivot})+1}$

Proof. Applying Lemma 2 with $\frac{\varepsilon}{1+\varepsilon} < \varepsilon$, we have $\mathsf{Pr}\left[(1+\varepsilon)^{-1}|R_F| \le |Y| \le (1+\varepsilon)|R_F| \mid + \log(\text{pivot}) - 1 \le \log|R_F|\right] \ge 1 - \frac{e^{-3/2}}{\log|R_F| - \ell - \log(\text{pivot})+1}$ □

Lemma 4. *Let an invocation of* SMTApproxMCCore *from* SMTApproxMC *return* m. *Then* $\mathsf{Pr}\left[(1+\varepsilon)^{-1}|R_F| \le m \le (1+\varepsilon)|R_F|\right] \ge 0.6$

Proof. For notational convenience, we use $(_l)$ to denote the value of when $i = l$ in the loop in SMTApproxMCCore. As noted earlier, we assume, for some $i = \ell^*$, $\log|R_F| - \log(_{\ell^*}) - 1 = \log(pivot)$. Furthermore, note that for all $i \neq j$ and $_i >_j$, $_i/_j > 2$. Let F_l denote the event that $|Y| < \text{pivot} \wedge (|Y| > (1+\varepsilon)|R_F| \vee |Y| < (1+\varepsilon)^{-1}|R_F|)$ for $i = l$. Let ℓ_1 be the value of i such that $_{\ell_1} <_{\ell^*}/2 \wedge \forall j,_j <_{\ell^*}/2 \implies _{\ell_1} \ge_j$. Similarly, let ℓ_2 be the value of i such that $_{\ell_2} <_{\ell^*}/4 \wedge \forall j,_j <_{\ell^*}/4 \implies _{\ell_2} \ge_j$

Then, $\forall_{i|_i <_{\ell^*}/4}$, $F_i \subseteq F_{\ell_2}$. Furthermore, the probability of $\mathsf{Pr}\left[(1+\varepsilon)^{-1}|R_F| \le m \le (1+\varepsilon)|R_F|\right]$ is at least $1 - \mathsf{Pr}[F_{\ell_2}] - \mathsf{Pr}[F_{\ell_1}] - \mathsf{Pr}[F_{\ell^*}] = 1 - \frac{e^{-3/2}}{4} - \frac{e^{-3/2}}{2} - e^{-3/2} \ge 0.6$
.

□

Now, we apply standard combinatorial analysis on repetition of probabilistic events and prove that SMTApproxMC is (ε, δ) model counter.

Theorem 2. *Suppose an invocation of* $\mathsf{SMTApproxMC}(F, \varepsilon, \delta, k)$ *returns* FinalCount. *Then* $\mathsf{Pr}\left[(1+\varepsilon)^{-1}|R_F| \le \text{FinalCount} \le (1+\varepsilon)|R_F|\right] \ge 1 - \delta$

Proof. Throughout this proof, we assume that SMTApproxMCCore is invoked t times from SMTApproxMC, where $t = \lceil 35 \log_2(3/\delta) \rceil$ in Section 5). Referring to the pseudocode of SMTApproxMC, the final count returned by SMTApproxMC is the median of non-$\perp$ counts obtained from the t invocations of SMTApproxMCCore. Let Err denote the event that the median is not in $\left[(1+\varepsilon)^{-1} \cdot |R_F|, (1+\varepsilon) \cdot |R_F|\right]$. Let "$\#non\perp = q$" denote the event that q (out of t) values returned by SMTApproxMCCore are non-$\perp$. Then, $\mathsf{Pr}[Err] = \sum_{q=0}^{t} \mathsf{Pr}[Err \mid \#non\perp = q] \cdot \mathsf{Pr}[\#non\perp = q]$.

In order to obtain $\mathsf{Pr}[Err \mid \#non\perp = q]$, we define a 0-1 random variable Z_i, for $1 \le i \le t$, as follows. If the i^{th} invocation of SMTApproxMCCore returns c, and if c is either $\perp$ or a non-$\perp$ value that does not lie in the interval $[(1+\varepsilon)^{-1} \cdot |R_F|, (1+\varepsilon) \cdot |R_F|]$, we set Z_i to 1; otherwise, we set it to 0. From Lemma 4, $\mathsf{Pr}[Z_i = 1] = p < 0.4$. If Z denotes $\sum_{i=1}^{t} Z_i$, a necessary (but not sufficient) condition for event Err to occur, given that q non-$\perp$s were returned by SMTApproxMCCore, is $Z \ge (t - q + \lceil q/2 \rceil)$. To see why this is so, note that $t-q$ invocations of SMTApproxMCCore must return $\perp$. In addition, at least $\lceil q/2 \rceil$ of the remaining q invocations must return values outside the desired interval. To simplify the exposition, let q be an even integer. A more careful analysis removes this restriction and results in an additional constant scaling factor for $\mathsf{Pr}[Err]$. With our simplifying assumption, $\mathsf{Pr}[Err \mid \#non\perp = q] \le \mathsf{Pr}[Z \ge (t - q + q/2)] = \eta(t, t - q/2, p)$. Since $\eta(t, m, p)$ is a decreasing function of m and since $q/2 \le t - q/2 \le t$, we have $\mathsf{Pr}[Err \mid \#non\perp = q] \le \eta(t, t/2, p)$. If $p < 1/2$, it is easy to verify that $\eta(t, t/2, p)$ is an increasing function of p. In our case, $p < 0.4$; hence, $\mathsf{Pr}[Err \mid \#non\perp = q] \le \eta(t, t/2, 0.4)$.

It follows from above that $\mathsf{Pr}[Err] = \sum_{q=0}^{t} \mathsf{Pr}[Err \mid \#non\perp = q] \cdot \mathsf{Pr}[\#non\perp = q] \le \eta(t, t/2, 0.4) \cdot \sum_{q=0}^{t} \mathsf{Pr}[\#non\perp = q] = \eta(t, t/2, 0.4)$. Since $\binom{t}{t/2} \ge \binom{t}{k}$ for all $t/2 \le k \le t$, and since $\binom{t}{t/2} \le 2^t$, we have $\eta(t, t/2, 0.4) = \sum_{k=t/2}^{t} \binom{t}{k} (0.4)^k (0.6)^{t-k} \le \binom{t}{t/2} \sum_{k=t/2}^{t} (0.4)^k (0.6)^{t-k} \le 2^t \sum_{k=t/2}^{t} (0.6)^t (0.4/0.6)^k \le 2^t \cdot 3 \cdot (0.6 \times 0.4)^{t/2} \le 3 \cdot (0.98)^t$. Since $t = \lceil 35 \log_2(3/\delta) \rceil$, it follows that $\mathsf{Pr}[Err] \le \delta$. □

Theorem 3. $\mathsf{SMTApproxMC}(F, \varepsilon, \delta, k)$ *runs in time polynomial in* $|F|$, $1/\varepsilon$ *and* $\log_2(1/\delta)$ *relative to an* NP*-oracle.*

Proof. Referring to the pseudocode for SMTApproxMC, lines 1– 3 take time no more than a polynomial in $\log_2(1/\delta)$ and $1/\varepsilon$. The repeat-until loop in lines 4– 9 is repeated $t = \lceil 35 \log_2(3/\delta) \rceil$ times. The time taken for each iteration is dominated by the time taken by SMTApproxMCCore. Finally, computing the median in line 10 takes time linear in t. The proof is therefore completed by showing that SMTApproxMCCore takes time polynomial in $|F|$ and $1/\varepsilon$ relative to the SAT oracle.

Referring to the pseudocode for SMTApproxMCCore, we find that BoundedSMT is called $\mathcal{O}(|F|)$ times. Each such call can be implemented by at most $pivot + 1$ calls to a NP oracle (SMT solver in case), and takes time polynomial in $|F|$ and $pivot + 1$ relative to the oracle. Since $pivot + 1$ is in $\mathcal{O}(1/\varepsilon^2)$, the number of calls to the NP oracle, and the total time taken by all calls to BoundedSMT in each invocation of SMTApproxMCCore is a polynomial in $|F|$ and $1/\varepsilon$ relative to the oracle. The random choices in lines 7 and 8 of SMTApproxMCCore can be implemented in time polynomial in $n.k$ (hence, in $|F|$) if we have access to a source of random bits. Constructing $F \wedge (h(\mathbf{X}) = \alpha)$ in line 9 can also be done in time polynomial in $|F|$. □

Detailed Experimental Results

The Table 2 represents the counts corresponding to benchmarks in Figure 1. Column 1 lists the ID for every benchmark, which corresponds to the position on the x-axis in Figure 1. Column 2 lists the name of every benchmark. The exact count computed by sharpSAT on bit-blasted versions of the benchmarks and column 4 lists counts computed by SMTApproxMC. Similar to the observation based on Figure 1, the Table 2 clearly demonstrates that the counts computed by SMTApproxMC are very close to the exact counts.

Table 3 is an extended version of Table 1. Similar to Table 1, column 1 gives the benchmark name, column 2 gives the sum of widths of all variables, column 3 lists the number of variables ($\mathrm{numVars}$) for each corresponding width (w) in the format $\{\mathrm{w} : \mathrm{numVars}\}$. To indicate the complexity of the input formula, we present the number of operations in the original SMT formula in column 4. The runtimes for SMTApproxMC and CDM are presented in columns 5 and column 6 respectively. We use "–" to denote timeout after 12 hours.

Table 2: Comparison of exact counts vs counts returned by SMTApproxMC

Id	Benchmark	Exact Count	SMTApproxMC Count
1	case127	64	65
2	case128	64	65
3	case24	256	245
4	case29	256	240
5	case25	512	525
6	case30	512	500
7	case28	1024	1025
8	case33	1024	1025
9	case27	1024	1025
10	case32	1024	975
11	case26	1024	1050
12	case31	1024	1025
13	case17	2048	2250
14	case23	2048	1875
15	case38	4096	4250
16	case21	8192	7500
17	case22	8192	8750
18	case11	16384	16875
19	case43	16384	16250
20	case45	16384	13750
21	case4	32768	37500
22	case44	65536	56250
23	case46	65536	71875
24	case108	65536	62500
25	case7	131072	157216
26	case1	131072	115625
27	case68	131072	132651
28	case47	262144	334084
29	case51	262144	250563
30	case52	262144	240737
31	case53	262144	245650
32	case134	262144	264196
33	case137	262144	264196
34	case56	1048576	1015625
35	case54	1048576	1093750
36	case109	1048576	1250000
37	case100	2097152	1915421
38	case101	2097152	2047519
39	case2	4194304	3515625
40	case8	8388608	9938999

Table 3: Extended Runtime performance of SMTApproxMC vis-a-vis CDM for a subset of benchmarks.

Benchmark	Total Bits	Variable Types	# of Operations	SMTApproxMC time(s)	CDM time(s)
squaring27	59	{1: 11, 16: 3}	10	–	2998.97
1159708	64	{8: 4, 32: 1}	12	14793.93	–
1159472	64	{8: 4, 32: 1}	8	16308.82	–
1159115	64	{8: 4, 32: 1}	12	23984.55	–
1159520	64	{8: 4, 32: 1}	1388	114.53	155.09
1160300	64	{8: 4, 32: 1}	1183	44.02	71.16
1159005	64	{8: 4, 32: 1}	213	28.88	105.6
1159751	64	{8: 4, 32: 1}	681	143.32	193.84
1159391	64	{8: 4, 32: 1}	681	57.03	91.62
case1	17	{1: 13, 4: 1}	13	17.89	65.12
1159870	64	{8: 4, 32: 1}	164	17834.09	9152.65
1160321	64	{8: 4, 32: 1}	10	117.99	265.67
1159914	64	{8: 4, 32: 1}	8	230.06	276.74
1159064	64	{8: 4, 32: 1}	10	69.58	192.36
1160493	64	{8: 4, 32: 1}	8	317.31	330.47
1159197	64	{8: 4, 32: 1}	8	83.22	176.23
1160487	64	{8: 4, 32: 1}	10	74.92	149.44
1159606	64	{8: 4, 32: 1}	686	431.23	287.85
case100	22	{1: 6, 16: 1}	8	32.62	89.69
1160397	64	{8: 4, 32: 1}	70	126.08	172.24
1160475	64	{8: 4, 32: 1}	67	265.58	211.16
case108	24	{1: 20, 4: 1}	7	37.33	100.2
case101	22	{1: 6, 16: 1}	12	44.74	90
1159244	64	{8: 4, 32: 1}	1474	408.63	273.57
case46	20	{1: 8, 4: 3}	12	16.95	76.4
case44	20	{1: 8, 4: 3}	8	13.69	72.05
case134	19	{1: 3, 16: 1}	8	5.36	54.22
case137	19	{1: 3, 16: 1}	9	10.98	56.12
case68	26	{8: 3, 1: 2}	7	34.9	67.48
case54	20	{1: 16, 4: 1}	8	50.73	103.91
1160365	64	{8: 4, 32: 1}	286	98.38	99.74
1159418	32	{8: 2, 16: 1}	7	3.73	43.68
1160877	32	{8: 2, 16: 1}	8	2.57	44.01
1160988	32	{8: 2, 16: 1}	8	4.4	44.64
1160521	32	{8: 2, 16: 1}	7	4.96	44.52
1159789	32	{8: 2, 16: 1}	13	6.35	43.09
1159117	32	{8: 2, 16: 1}	13	5.55	43.18
1159915	32	{8: 2, 16: 1}	11	7.02	45.62
1160332	32	{8: 2, 16: 1}	12	3.94	44.35
1159582	32	{8: 2, 16: 1}	8	5.37	43.98
1160530	32	{8: 2, 16: 1}	12	2.01	43.28
1160482	64	{8: 4, 32: 1}	36	153.99	120.55
1159564	32	{8: 2, 16: 1}	12	7.36	41.77
1159990	64	{8: 4, 32: 1}	34	71.17	97.25
case7	18	{1: 10, 8: 1}	12	17.93	51.96
case56	20	{1: 16, 4: 1}	12	41.54	109.3
case43	15	{1: 11, 4: 1}	12	8.6	37.63
case45	15	{1: 11, 4: 1}	12	9.3	35.77
case53	19	{1: 7, 8: 1, 4: 1}	9	53.66	69.96
case4	16	{1: 12, 4: 1}	12	8.42	35.49
1160438	64	{8: 4, 32: 1}	2366	199.08	141.84
case109	29	{1: 21, 4: 2}	12	171.51	179.98
case38	13	{1: 9, 4: 1}	7	6.21	30.27
case11	15	{1: 11, 4: 1}	8	7.26	33.75

Continued on next page

Benchmark	Total Bits	Variable Types	# of Operations	SMTApproxMC time(s)	CDM time(s)
1158973	64	{8: 4, 32: 1}	94	366.6	270.17
case22	14	{1: 10, 4: 1}	12	5.46	26.03
case21	14	{1: 10, 4: 1}	12	5.57	24.59
case52	19	{1: 7, 8: 1, 4: 1}	9	45.1	70.72
case23	12	{1: 8, 4: 1}	11	2.29	12.84
case51	19	{1: 7, 8: 1, 4: 1}	12	40	67.22
case17	12	{1: 8, 4: 1}	12	2.75	11.09
case33	11	{1: 7, 4: 1}	12	1.7	9.66
case30	13	{1: 5, 4: 2}	13	1.41	8.69
case28	11	{1: 7, 4: 1}	12	1.66	8.73
case25	13	{1: 5, 4: 2}	12	1.39	8.27
case27	11	{1: 7, 4: 1}	12	1.69	8.57
case26	11	{1: 7, 4: 1}	12	1.68	8.35
case32	11	{1: 7, 4: 1}	12	1.46	8.16
case31	11	{1: 7, 4: 1}	12	1.64	7.64
case29	12	{1: 4, 4: 2}	8	0.67	5.16
case24	12	{1: 4, 4: 2}	12	0.77	4.94
1160335	64	{8: 4, 32: 1}	216	0.31	0.54
1159940	64	{8: 4, 32: 1}	94	0.17	0.04
1159690	32	{8: 2, 16: 1}	8	0.12	0.04
1160481	32	{8: 2, 16: 1}	12	0.13	0.03
1159611	64	{8: 4, 32: 1}	73	0.2	0.09
1161180	32	{8: 2, 16: 1}	12	0.11	0.04
1160849	32	{8: 2, 16: 1}	7	0.1	0.03
1159790	64	{8: 4, 32: 1}	113	0.15	0.04
1160315	64	{8: 4, 32: 1}	102	0.17	0.04
1159720	64	{8: 4, 32: 1}	102	0.17	0.05
1159881	64	{8: 4, 32: 1}	102	0.16	0.04
1159766	64	{8: 4, 32: 1}	73	0.15	0.03
1160220	64	{8: 4, 32: 1}	681	0.17	0.03
1159353	64	{8: 4, 32: 1}	113	0.16	0.04
1160223	64	{8: 4, 32: 1}	102	0.17	0.04
1159683	64	{8: 4, 32: 1}	102	0.17	0.03
1159702	64	{8: 4, 32: 1}	102	0.19	0.04
1160378	64	{8: 4, 32: 1}	476	0.17	0.04
1159183	64	{8: 4, 32: 1}	172	0.17	0.03
1159747	64	{8: 4, 32: 1}	322	0.18	0.03
1159808	64	{8: 4, 32: 1}	539	0.17	0.03
1159849	64	{8: 4, 32: 1}	322	0.18	0.03
1159449	64	{8: 4, 32: 1}	540	0.3	0.05
case47	26	{1: 6, 8: 2, 4: 1}	11	81.5	80.25
case2	24	{1: 20, 4: 1}	10	273.91	194.33
1159239	64	{8: 4, 32: 1}	238	1159.32	449.21
case8	24	{1: 12, 8: 1, 4: 1}	8	433.2	147.35
1159936	64	{8: 4, 32: 1}	238	5835.35	1359.9
squaring51	40	{1: 32, 4: 2}	7	3285.52	607.22
1159431	64	{8: 4, 32: 1}	12	36406.4	–
1160191	64	{8: 4, 32: 1}	12	40166.1	–