On Parallel Scalable Uniform SAT Witness Generation*,**

Supratik Chakraborty[1], Daniel J. Fremont[2], Kuldeep S. Meel[3], Sanjit A. Seshia[2], and Moshe Y. Vardi[3]

[1] Indian Institute of Technology, Bombay
[2] University of California, Berkeley
[3] Department of Computer Science, Rice University

Abstract. Constrained-random verification (CRV) is widely used in industry for validating hardware designs. The effectiveness of CRV depends on the uniformity of test stimuli generated from a given set of constraints. Most existing techniques sacrifice either uniformity or scalability when generating stimuli. While recent work based on random hash functions has shown that it is possible to generate almost uniform stimuli from constraints with 100,000+ variables, the performance still falls short of today's industrial requirements. In this paper, we focus on pushing the performance frontier of uniform stimulus generation further. We present a random hashing-based, easily parallelizable algorithm, UniGen2, for sampling solutions of propositional constraints. UniGen2 provides strong and relevant theoretical guarantees in the context of CRV, while also offering significantly improved performance compared to existing almost-uniform generators. Experiments on a diverse set of benchmarks show that UniGen2 achieves an average speedup of about 20× over a state-of-the-art sampling algorithm, even when running on a single core. Moreover, experiments with multiple cores show that UniGen2 achieves a near-linear speedup in the number of cores, thereby boosting performance even further.

1 Introduction

Functional verification is concerned with the verification and validation of a *Design Under Verification* (DUV) with respect to design specifications. With

* The full version is available at `http://www.cs.rice.edu/CS/Verification/Projects/UniGen/`

** The authors would like to thank Suguman Bansal and Karthik Murthy for valuable comments on the earlier drafts, Armando Solar-Lezama for benchmarks, and Mate Soos for tweaking CMS to support UniGen2. This work was supported in part by NSF grants CNS 1049862, CCF-1139011, CCF-1139138, 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 Board of Research in Nuclear Sciences, India, by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467 and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc., and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 304–319, 2015.
DOI: 10.1007/978-3-662-46681-0_25

the increasing complexity of DUVs, functional verification has become one of the most challenging and time-consuming steps in design validation [3]. In view of the high computational cost of formal verification, simulation-based techniques have been extensively employed in industrial practice. The success of such techniques depends on the *quality* of input stimuli with which the design is simulated. The generation of high-quality stimuli that uncover hidden bugs continues to be a challenging problem even today [21].

The problem of high-quality stimulus generation has led to the emergence of *constrained-random simulation*, also known as *constrained-random verification* (CRV) [22]. In CRV, a verification engineer is tasked with the construction of verification scenarios, expressed as constraints over stimuli. Typically, constructing these scenarios involves applying past user experience, inputs from design engineers, and domain-specific knowledge. A constraint solver is then invoked to generate random stimuli satisfying the constraints. Since the distribution of errors in the design is not known *a priori*, each random stimulus is just as likely to produce an error as any other. Therefore, achieving a uniformly random distribution over stimuli satisfying the constraints is highly desirable.

While constraint-solving technologies have witnessed significant advancements over the last decade, methods of generating uniformly distributed solutions still face huge scalability hurdles. This has been observed repeatedly in the literature [6] and by industry practitioners[1]. In this paper, we take a step towards remedying the current situation by proposing an easily parallelizable sampling algorithm for Boolean constraints that provides strong theoretical guarantees (similar to those provided by an almost-uniform generator) in the context of CRV, and also runs significantly faster than current state-of-the-art techniques on a diverse set of benchmark problems.

Since constraints arising in CRV can often be encoded as propositional formulae in conjunctive normal form (CNF), we focus on almost-uniform sampling of satisfying assignments of CNF formulae (known as SAT *witnesses*). This problem has been extensively studied in both theoretical and practical contexts, and has many applications, including probabilistic reasoning, approximate model counting, and Markov logic networks [4]. Until recently, approaches to solving this problem belonged to one of two classes: those which provide strong guarantees of uniformity but scale poorly [2,24], and those which scale to large problem instances but rely on heuristics and hence offer very weak or no uniformity guarantees [11,16,14].

Recently, Chakraborty, Meel, and Vardi [4] proposed a new algorithmic approach to bridge the gap between these two extremes. The main idea behind their approach is to use universal hashing in order to partition the space of witnesses into roughly equal "cells". Under an appropriate partitioning scheme, choosing a random witness from a randomly chosen cell provides strong uniformity guarantees. The most recent instance of this approach is called UniGen [6]. While UniGen scales to formulae much larger than those that can be handled

[1] Private Communication: R. Kurshan.

by previous state-of-the-art techniques, the runtime performance of UniGen still falls short of industry requirements.

Since the end of Dennard scaling, there has been a strong revival of interest in parallelizing a wide variety of algorithms to achieve improved performance [10]. One of the main goals in parallel-algorithm design is to achieve a speedup nearly linear in the number of processors, which requires the avoidance of dependencies among different parts of the algorithm [8]. Most of the sampling algorithms used for uniform witness generation fail to meet this criterion, and are hence not easily parallelizable. In contrast, the algorithm proposed in this paper is inherently parallelizable, and achieves a near-linear speedup.

Our primary contribution is a new algorithm, UniGen2, that addresses key performance deficiencies of UniGen. Significantly, UniGen2 generates many more samples (witnesses) per iteration compared to UniGen, thereby reducing the number of SAT calls required per sample to a *constant*. While this weakens the guarantee of independence among samples, we show that this does not hurt the primary objective of CRV. Specifically, we prove that UniGen2 provides almost as strong guarantees as UniGen with respect to discovery of bugs in a CRV setting. On the practical front, we present an implementation of UniGen2, and show by means of extensive experiments that it significantly outperforms existing state-of-the-art algorithms, while generating sample distributions that are indistinguishable from those generated by an ideal uniform sampler. UniGen2 is also inherently parallelizable, and we have implemented a parallel version of it. Our experiments show that parallel UniGen2 achieves a near-linear speedup with the number of cores.

2 Notation and Preliminaries

Let F denote a Boolean formula in conjunctive normal form (CNF), and let X be the set of variables appearing in F. The set X is called the *support* of F. Given a set of variables $S \subseteq X$ and an assignment σ of truth values to the variables in X, we write $\sigma_{\downarrow S}$ for the projection of σ onto S. A *satisfying assignment* or *witness* of F is an assignment that makes F evaluate to true. We denote the set of all witnesses of F by R_F and the projection of R_F on S by $R_{F\downarrow S}$. For the rest of the paper, we use S to denote the *sampling set*, the set of variables on which we desire assignments to be projected. Even when no projection is desired explicitly, S can often be restricted to a small subset of X, called an *independent support* (see [6] for details) such that $|R_F| = |R_{F\downarrow S}|$. For notational simplicity, we omit mentioning F and S when they are clear from the context.

We use $\mathsf{Pr}[X]$ to denote the probability of event X. We say that a set of events $\{X_1, X_2, \ldots, X_n\}$ are (l, u) *almost-independent almost-identically distributed* (henceforth, called (l, u)-*a.a.d.*) if $\forall i \in \{1, \ldots n\},\ l \leq \mathsf{Pr}[X_i] \leq u$ and $l \leq \mathsf{Pr}[X_i|(\{X_1, \ldots, X_n\} \setminus X_i)] \leq u$. Note that this notion is similar to that of independently identically distributed (i.i.d.) events, but somewhat weaker.

Given a Boolean formula F and sampling set S, a *probabilistic generator* of witnesses of F is a probabilistic algorithm that generates a random element of

$R_{F\downarrow S}$. A *uniform generator* $\mathcal{G}^u(\cdot,\cdot)$ is a probabilistic generator that guarantees $\mathsf{Pr}\left[\mathcal{G}^u(F,S)=y\right]=1/|R_{F\downarrow S}|$, for every $y \in R_{F\downarrow S}$. An *almost-uniform generator* $\mathcal{G}^{au}(\cdot,\cdot,\cdot)$ relaxes the above guarantees, ensuring only that $1/((1+\varepsilon)|R_{F\downarrow S}|) \leq \mathsf{Pr}\left[\mathcal{G}^{au}(F,S,\varepsilon)=y\right] \leq (1+\varepsilon)/|R_{F\downarrow S}|$ for every $y \in R_{F\downarrow S}$ and tolerance ε (>0). Probabilistic generators are allowed to occasionally "fail" by returning no witness although $R_{F\downarrow S} \neq \emptyset$. The failure probability must be bounded by a constant strictly less than 1.

A special class of hash functions, called *r-wise independent* hash functions, play a crucial role in our work. Let n, m and r be positive integers, and let $H(n,m,r)$ denote a family of r-wise independent hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} H(n,m,r)$ to denote the probability space obtained by choosing a hash function h uniformly at random from $H(n,m,r)$. The property of r-wise independence guarantees that for all $\alpha_1,\ldots\alpha_r \in \{0,1\}^m$ and for all distinct $y_1,\ldots y_r \in \{0,1\}^n$, $\mathsf{Pr}\left[\bigwedge_{i=1}^{r} h(y_i)=\alpha_i \ : h \xleftarrow{R} H(n,m,r)\right] = 2^{-mr}$. For every $\alpha \in \{0,1\}^m$ and $h \in H(n,m,r)$, let $h^{-1}(\alpha)$ denote the set $\{y \in \{0,1\}^n \mid h(y)=\alpha\}$. Given $R_{F\downarrow S} \subseteq \{0,1\}^{|S|}$ and $h \in H(|S|,m,r)$, we use $R_{F\downarrow S,h,\alpha}$ to denote the set $R_{F\downarrow S} \cap h^{-1}(\alpha)$. If we keep h fixed and let α range over $\{0,1\}^m$, the corresponding sets $R_{F\downarrow S,h,\alpha}$ form a partition of $R_{F\downarrow S}$.

We use a particular class of hash functions from $\{0,1\}^m$ to $\{0,1\}^n$, denoted by $H_{xor}(n,m)$, which is defined as follows. Let $h : \{0,1\}^m \to \{0,1\}^n$ be a hash function, y be a vector in $\{0,1\}^m$ and $h(y)[i]$ be the i^{th} component of the vector $h(y)$. The family $H_{xor}(n,m)$ is defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,j} \in \{0,1\}, 1 \leq i \leq m, 0 \leq j \leq n\}$, where $\oplus$ denotes XOR. By choosing values of $a_{i,j}$ randomly and independently, we can choose a random function from $H_{xor}(n,m)$. It was shown in [12] that the family $H_{xor}(n,m)$ is 3-wise independent.

3 Related Work

Uniform generation of SAT witnesses was studied by Jerrum, Valiant, and Vazirani [15], who showed that the problem can be solved in probabilistic polynomial time, given access to a Σ_2^{P} oracle. In addition, they showed that almost-uniform generation is polynomially inter-reducible with approximate model counting. Bellare, Goldreich, and Petrank [2] improved this result and provided an algorithm in $\mathsf{BPP}^{\mathsf{NP}}$. Unfortunately, their algorithm fails to scale beyond few tens of variables in practice [4]. A completely different approach to uniform generation of SAT witnesses is due to Yuan et al. [24], wherein a sample is generated by performing a random walk over a weighted binary decision diagram (WBDD). The high space requirement of this technique limits its applicability in practice.

In several settings (some industrial), generation of stimuli for CRV is typically done via heuristic methods that provide very weak or no guarantees of uniformity. One of the earliest such approaches was to randomly seed a SAT solver [19]. While this is simple in principle, the distributions generated by random seeding have been shown to be highly skewed in practice [17]. An alternative approach focusing on the generation of "diverse" solutions was proposed by Nadel [20], but it also fails to provide theoretical guarantees of coverage.

Markov Chain Monte Carlo (MCMC) algorithms, such as those based on simulated annealing or the Metropolis-Hastings algorithm, have been studied extensively in the literature [18] in the context of generating samples from a probability space. The eventual convergence to the target distribution for MCMC methods is often impractically slow in practice under mild requirements. Most MCMC-based sampling tools therefore use heuristic adaptations [17,16] to improve performance and reduce correlation between samples. Unfortunately, these heuristics significantly weaken or even destroy the theoretical guarantees.

Interval propagation [14] has been used extensively in industrial practice to achieve scalable stimulus generation. Techniques based on interval propagation, however, generate highly non-uniform distributions. Recent efforts via the conversion of constraints into belief networks [11,7] have also failed to achieve the desired balance between performance and guarantees of uniformity.

Recently, several random hashing-based techniques have been proposed to bridge the wide gap between scalable algorithms and those that give strong guarantees of uniformity when sampling witnesses of propositional constraints [4,6,9]. Hashing-based sampling techniques were originally pioneered by Sipser [23] and further used by Jerrum et al [15], and Bellare et al [2]. The key idea in hashing-based techniques is to first partition the space of satisfying assignments into small "cells" of roughly equal size using r-wise independent hash functions (for a suitable value of r), and then randomly choose a solution from a randomly picked cell. Bellare et al. showed that by choosing $r = n$ (where the propositional constraint has n variables), we can guarantee uniform generation. The resulting algorithm, however, does not scale in practice. Chakraborty, Meel, and Vardi [4] subsequently showed that with $r = 3$, a significantly more scalable near-uniform generator named UniWit can be designed. Building on the principle underlying UniWit, Ermon et al. [9] suggested further algorithmic improvements to uniform generation of witnesses.

Recently, Chakraborty et al. proposed a new algorithm named UniGen [5], which improves upon the ideas of UniWit. In particular, UniGen provides stronger guarantees of uniformity by exploiting a deep connection between approximate counting and almost-uniform sampling [15]. Furthermore, UniGen has been shown to scale to formulae with hundreds of thousands of variables. Even so, UniGen is typically 2-3 orders of magnitude slower than a single call to a SAT solver and therefore, its runtime performance falls short of the performance of heuristic methods commonly employed in industry to generate stimuli for CRV [2]. In this paper, we offer several improvements to UniGen and obtain an algorithm with substantially improved performance that can be further scaled by parallelization to match the requirements of industry.

[2] A random-constrained test case generator is typically allowed to be 10× slower than a constraint solver (private communication with industry expert W. Hung).

4 A Parallel SAT Sampler

In this section, we first motivate the need for sampling solutions of constraints in parallel, and then provide technical details of our algorithm, named UniGen2.

Parallelization. While simulation-based verification typically involves running in parallel many simulations with different input stimuli, the generation of these stimuli is often done sequentially. This is because existing approaches to stimulus generation are not efficiently parallelizable without degrading guarantees of uniformity. For example, approaches based on random seeding of a SAT solver maintain information about which regions of the solution space have already been explored, since choosing random seeds is often not good enough to steer the solver towards new regions of the solution space [17]. Different threads generating solutions must therefore communicate with each other, impeding efficient parallelization. In MCMC-based approaches, to generate independent samples in parallel, each thread has to take a random walk until a stationary distribution is reached. The length of this walk is often impractically long in the case of combinatorial spaces with complex internal structure [9]. Heuristics to speed up MCMC-based techniques destroy guarantees of uniformity even in the sequential case [17]. Methods based on random walks on WBDDs are amenable to parallelization, but they are known not to scale beyond a few hundred variables. The lack of techniques for sampling solutions of constraints in parallel while preserving guarantees of effectiveness in finding bugs is therefore a major impediment to high-performance CRV.

The algorithm UniGen2 presented in this section takes a step forward in addressing the above problem. It has an initial preprocessing step that is sequential but low-overhead, followed by inherently parallelizable sampling steps. It generates samples (stimuli) that are provably nearly as effective as those generated by an almost-uniform sampler for purposes of detecting a bug. Furthermore, our experiments demonstrate that a parallel implementation of UniGen2 achieves a near-linear speedup in the number of processor cores. Given that current practitioners are forced to trade guarantees of effectiveness in bug hunting for scalability, the above properties of UniGen2 are significant. Specifically, they enable a new paradigm of CRV wherein parallel stimulus generation and simulation can provide the required runtime performance while also providing theoretical guarantees.

Algorithm. Our algorithm, named UniGen2, bears some structural similarities with the UniGen algorithm proposed earlier in [6]. Nevertheless, there are key differences that allow UniGen2 to outperform UniGen significantly. Like UniGen, UniGen2 takes a CNF formula F, a sampling set S and a tolerance ε (that is chosen to be at least 6.84 for technical reasons). Note that the formula F and set S uniquely define the solution set $R_{F\downarrow S}$.

Similarly to UniGen, UniGen2 works by partitioning $R_{F\downarrow S}$ into "cells" using random hash functions, then randomly selecting a cell by adding appropriate constraints to F. If the chosen cell has the right size (where the acceptable size

range depends on the desired tolerance ε), we can enumerate all the solutions in it and return a uniform random sample from among them. Unlike UniGen, however, UniGen2 samples multiple times from the same cell. This decreases the generation time per sample by a large factor (about 10× in our experiments), while preserving strong guarantees of effectiveness of the samples in finding bugs.

Algorithm 1. EstimateParameters(F, S, ε)

/* Returns (hashBits, loThresh, hiThresh) as required by GenerateSamples */
Find $\kappa \in (0,1)$ such that $\varepsilon = (1+\kappa)(7.44 + \frac{0.392}{(1-\kappa)^2}) - 1$
pivot $\leftarrow \left\lceil 4.03\left(1+\frac{1}{\kappa}\right)^2 \right\rceil$
hiThresh $\leftarrow \lceil 1 + \sqrt{2}(1+\kappa)\text{pivot} \rceil$; loThresh $\leftarrow \left\lfloor \frac{1}{\sqrt{2}(1+\kappa)}\text{pivot} \right\rfloor$
$i \leftarrow 0$
while $i < n$ **do**
    $i \leftarrow i + 1$
    Choose h at random from $H_{xor}(|S|, i)$
    Choose α at random from $\{0,1\}^i$
    $Y \leftarrow \mathsf{BSAT}(F \wedge (h(S) = \alpha), 61, S)$
    if $1 \leq |Y| \leq 60$ **then**
        return (round $(\log|Y| + i + \log 1.8 - \log \text{pivot})$, loThresh, hiThresh)
return $\perp$

Algorithm 2. GenerateSamples$(F, S, \text{hashBits}, \text{loThresh}, \text{hiThresh})$

Pick an order V of the values $\{\text{hashBits}-2, \text{hashBits}-1, \text{hashBits}\}$
for $i \in V$ **do**
    Choose h at random from $H_{xor}(|S|, i)$
    Choose α at random from $\{0,1\}^i$
    $Y \leftarrow \mathsf{BSAT}(F \wedge (h(S) = \alpha), \text{hiThresh}, S)$
    if (loThresh $\leq |Y| <$ hiThresh) **then**
        return loThresh distinct random elements of Y
return $\perp$

UniGen2 is an algorithmic framework that operates in two stages: the first stage, EstimateParameters (Algorithm 1), performs low-overhead one-time preprocessing for a given F, S, and ε to compute numerical parameters 'hashBits', 'loThresh', and 'hiThresh'. The quantity hashBits controls how many cells $R_{F\downarrow S}$ will be partitioned into, while loThresh and hiThresh delineate the range of acceptable sizes for a cell. In the second stage, GenerateSamples (Algorithm 2) uses these parameters to generate loThresh samples. If more samples are required, GenerateSamples is simply called again with the same parameters. Theorem 3 below shows that invoking GenerateSamples multiple times does not cause the loss of any theoretical guarantees. We now explain the operation of the two subroutines in detail.

Lines 1–3 of EstimateParameters compute numerical parameters based on the tolerance ε which are used by GenerateSamples. The variable 'pivot' can be thought of as the ideal cell size we are aiming for, while as mentioned above 'loThresh' and 'hiThresh' define the allowed size range around this ideal. For simplicity of exposition, we assume that $|R_{F\downarrow S}| > \max(60, \text{hiThresh})$. If not, there are very few solutions and we can do uniform sampling by enumerating all of them as in UniGen [6].

Lines 4–11 of EstimateParameters compute 'hashBits', an estimate of the number of hash functions required so that the corresponding partition of $R_{F\downarrow S}$ (into 2^{hashBits} cells) has cells of the desired size. This is done along the same lines as in UniGen, which used an approximate model counter such as ApproxMC [5]. The procedure invokes a SAT solver through the function $\mathsf{BSAT}(\phi, m, S)$. This returns a set, consisting of models of the formula ϕ which all differ on the set of variables S, that has size m. If there is no such set of size m, the function returns a maximal set. If the estimation procedure fails, EstimateParameters returns $\perp$ on line 12. In practice, it would be called repeatedly until it succeeds. Theorem 1 below shows that on average few repetitions are needed for EstimateParameters to succeed, and this is borne out in practice.

The second stage of UniGen2, named GenerateSamples, begins on lines 1–2 by picking a hash count i close to hashBits, then selecting a random hash function from the family $H_{xor}(|S|, i)$ on line 3. On line 4 we pick a random output value α, so that the constraint $h(S) = \alpha$ picks out a random cell. Then, on line 5 we invoke BSAT on F with this additional constraint, obtaining at most hiThresh elements Y of the cell. If $|Y| <$ hiThresh then we have enumerated every element of $R_{F\downarrow S}$ in the cell, and if $|Y| \geq$ loThresh the cell is large enough for us to get a good sample. So if loThresh $\leq |Y| <$ hiThresh, we randomly select loThresh elements of Y and return them on line 7.

If the number of elements of $R_{F\downarrow S}$ in the chosen cell is too large or too small, we choose a new hash count on line 2. Note that line 1 can pick an arbitrary order for the three hash counts to be tried, since our analysis of UniGen2 does not depend on the order. This allows us to use an optimization where if we run GenerateSamples multiple times, we choose an order which starts with the value of i that was successful in the previous invocation of GenerateSamples. Since hashBits is only an estimate of the correct value for i, in many benchmarks on which we experimented, UniGen2 initially failed to generate a cell of the right size with $i =$ hashBits-2, but then succeeded with $i =$ hashBits-1. In such scenarios, beginning with $i =$ hashBits-1 in subsequent iterations saves considerable time. This heuristic is similar in spirit to "leapfrogging" in ApproxMC [5] and UniWit [4], but does not compromise the theoretical guarantees of UniGen2 in any way.

If all three hash values tried on line 2 fail to generate a correctly-sized cell, GenerateSamples fails and returns $\perp$ on line 8. Theorem 1 below shows that this happens with probability at most 0.38. Otherwise, UniGen2 completes by returning loThresh samples.

Parallelization of UniGen2**.** As described above, UniGen2 operates in two stages: EstimateParameters is initially called to do one-time preprocessing, and then GenerateSamples is called to do the actual sampling. To generate N samples, we can invoke EstimateParameters once, and then GenerateSamples $N/\text{loThresh}$ times, since each of the latter calls generates loThresh samples (unless it fails). Furthermore, each invocation of GenerateSamples is completely independent of the others. Thus if we have k processor cores, we can just perform $N/(k{\cdot}\text{loThresh})$ invocations of GenerateSamples on each. There is no need for any inter-thread communication: the "leapfrogging" heuristic for choosing the order on line 1 can simply be done on a per-thread basis. This gives us a linear speedup in the number of cores k, since the per-thread work (excluding the initial preprocessing) is proportional to $1/k$. Furthermore, Theorem 3 below shows that assuming each thread has its own source of randomness, performing multiple invocations of GenerateSamples in parallel does not alter its guarantees of uniformity. This means that UniGen2 can scale to an arbitrary number of processor cores as more samples are desired, while not sacrificing any theoretical guarantees.

5 Analysis

In this section, we present a theoretical analysis of the uniformity, effectiveness in discovering bugs, and runtime performance of UniGen2. For lack of space, we defer all proofs to the full version. For technical reasons, we assume that $\varepsilon > 6.84$. Our first result bounds the failure probabilities of EstimateParameters and GenerateSamples.

Theorem 1. EstimateParameters *and* GenerateSamples *return* $\perp$ *with probabilities at most* 0.009 *and* 0.38 *respectively.*

Next we show that a single invocation of GenerateSamples provides guarantees nearly as strong as those of an almost-uniform generator.

Theorem 2. *For given* F*,* S*, and* ε*, let* L *be the set of samples generated using* UniGen2 *with a single call to* GenerateSamples*. Then for each* $y \in R_{F\downarrow S}$*, we have*

$$\frac{\text{loThresh}}{(1+\varepsilon)|R_{F\downarrow S}|} \leq \mathsf{Pr}[y \in L] \leq 1.02 \cdot (1+\varepsilon)\frac{\text{loThresh}}{|R_{F\downarrow S}|}.$$

Now we demonstrate that these guarantees extend to the case when GenerateSamples is called multiple times, sequentially or in parallel.

Theorem 3. *For given* F*,* S*, and* ε*, and for* hashBits*,* loThresh*, and* hiThresh *as estimated by* EstimateParameters*, let* GenerateSamples *be called* N *times with these parameters in an arbitrary parallel or sequential interleaving. Let* $E_{y,i}$ *denote the event that* $y \in R_{F\downarrow S}$ *is generated in the* i^{th} *call to* GenerateSamples*. Then the events* $E_{y,i}$ *are* (l,u)-a.a.d. *with* $l = \frac{\text{loThresh}}{(1+\varepsilon)|R_{F\downarrow S}|}$ *and* $u = \frac{1.02\cdot(1+\varepsilon)\text{loThresh}}{|R_{F\downarrow S}|}$.

Next we show that the above result establishes very strong guarantees on the effectiveness of UniGen2 in discovering bugs in the CRV context. In this context,

the objective of uniform generation is to maximize the probability of discovering a bug by using a diverse set of samples. Let us denote the fraction of stimuli that trigger a bug by f, i.e. if B is the set of stimuli that trigger a bug, then $f = |B|/|R_{F\downarrow S}|$. Furthermore, if N is the desired number of stimuli we wish to generate, we want to minimize the failure probability, i.e. the probability that the N randomly generated stimuli fail to intersect the set B. If the stimuli are generated uniformly, the failure probability is $(1-f)^N$. Using binomial expansion, the failure probability can be shown to decrease exponentially in N, with decay rate of f (henceforth denoted as *failure decay rate*). We can evaluate the effectiveness of a stimulus-generation method by comparing the failure decay rate it achieves to that of a uniform generator. Alternatively, given some $\delta > 0$, we can ask how many samples are needed to ensure that the failure probability is at most δ. Normalizing by the number of samples needed by an ideal uniform generator gives the *relative number of samples needed* to find a bug. Our next theorem shows that UniGen2 is as effective as an almost-uniform generator according to both of these metrics but needs many fewer SAT calls.

Theorem 4. *Given F, S, ε, and $B \subseteq R_{F\downarrow S}$, let $f = |B|/|R_{F\downarrow S}| < 0.8$, $\nu = \frac{1}{2}(1+\varepsilon)f$, and $\widehat{\nu} = 1.02 \cdot \text{loThresh} \cdot \nu < 1$. Then we have the following bounds:*

generator type	*uniform*	UniGen	UniGen2
failure decay rate	f	$\frac{f}{1+\varepsilon}$	$(1-\widehat{\nu})\frac{f}{1+\varepsilon}$
relative # of samples needed	1	$(1+\nu)(1+\varepsilon)$	$\frac{1+\widehat{\nu}}{1-\widehat{\nu}}(1+\varepsilon)$
relative expected # of SAT calls	1	$\frac{3\cdot\text{hiThresh}(1+\nu)(1+\varepsilon)}{0.52}$	$\frac{3\cdot\text{hiThresh}}{0.62\cdot\text{loThresh}}\frac{1+\widehat{\nu}}{1-\widehat{\nu}}(1+\varepsilon)$

If $8.09 \leq \varepsilon \leq 242$ and $f \leq 1/1000$, then UniGen2 *uses fewer SAT calls than* UniGen *on average.*

Thus under reasonable conditions such as occur in industrial applications, UniGen2 is more efficient than UniGen at finding bugs. We illustrate the significance of this improvement with an example. Suppose 1 in 10^4 inputs causes a bug. Then to find a bug with probability $1/2$, we would need approximately $6.93 \cdot 10^3$ uniformly generated samples. To achieve the same target, we would need approximately $1.17 \cdot 10^5$ samples from an almost-uniform generator like UniGen, and approximately $1.20 \cdot 10^5$ samples from UniGen2, using a tolerance (ε) of 16 in both cases. However, since UniGen2 picks multiple samples from each cell, it needs fewer SAT calls. In fact, the expected number of calls made by UniGen2 is only $3.38 \cdot 10^6$, compared to $4.35 \cdot 10^7$ for UniGen – an order of magnitude difference! Therefore, UniGen2 provides as strong guarantees as UniGen in terms of its ability to discover bugs in CRV, while requiring far fewer SAT calls. Note that while the rest of our results hold for $\varepsilon > 6.84$, our guarantee of fewer expected SAT calls in UniGen2 holds for a subrange of values of ε, as indicated in Theorem 4.

Finally, since the ratio of hiThresh to loThresh can be bounded from above, we have the following result.

Theorem 5. *There exists a fixed constant* $\lambda = 40$ *such that for every* F, S, *and* ε, *the expected number of* SAT *queries made by* UniGen2 *per generated sample is at most* λ.

In contrast, the number of SAT calls per generated sample in UniGen is proportional to hiThresh and thus to ε^{-2}. An upper bound on the expected number of SAT queries makes it possible for UniGen2 to approach the performance of heuristic methods like random seeding of SAT solvers, which make only one SAT query per generated sample (but fail to provide any theoretical guarantees).

6 Evaluation

To evaluate the performance of UniGen2, we built a prototype implementation in C++ that employs the solver CryptoMiniSAT [1] to handle CNF-SAT augmented with XORs efficiently[3]. We conducted an extensive set of experiments on diverse public domain benchmarks, seeking to answer the following questions:

1. How does UniGen2's runtime performance compare to that of UniGen, a state-of-the-art almost-uniform SAT sampler?
2. How does the performance of parallel UniGen2 scale with the # of cores?
3. How does the distribution of samples generated by UniGen2 compare with the ideal distribution?
4. Does parallelization affect the uniformity of the distribution of the samples?

Our experiments showed that UniGen2 outperforms UniGen by a factor of about 20× in terms of runtime. The distribution generated by UniGen2 is statistically indistinguishable from that generated by an ideal uniform sampler. Finally, the runtime performance of parallel UniGen2 scales linearly with the number of cores, while its output distribution continues to remain uniform.

6.1 Experimental Setup

We conducted experiments on a heterogeneous set of benchmarks used in earlier related work [6]. The benchmarks consisted of ISCAS89 circuits augmented with parity conditions on randomly chosen subsets of outputs and next-state variables, constraints arising in bounded model checking, bit-blasted versions of SMTLib benchmarks, and problems arising from automated program synthesis. For each benchmark, the sampling set S was either taken to be the independent support of the formula or was provided by the corresponding source. Experiments were conducted on a total of 200+ benchmarks. We present results for only a subset of representative benchmarks here. A detailed list of all the benchmarks is available in the Appendix.

For purposes of comparison, we also ran experiments with UniGen [6], a state-of-the-art almost-uniform SAT witness generator. We employed the Mersenne

[3] The tool (with source code) is available at `http://www.cs.rice.edu/CS/Verification/Projects/UniGen/`

Twister to generate pseudo-random numbers, and each thread was seeded independently using the C++ class `random_device`. Both tools used an overall timeout of 20 hours, and a BSAT timeout of 2500 seconds. All experiments used $\varepsilon = 16$, corresponding to loThresh = 11 and hiThresh = 64. The experiments were conducted on a high-performance computer cluster, where each node had a 12-core, 2.83 GHz Intel Xeon processor, with 4GB of main memory per core.

6.2 Results

Runtime Performance. We compared the runtime performance of UniGen2 with that of UniGen for all our benchmarks. For each benchmark, we generated between 1000 and 10000 samples (depending on the size of the benchmark) and computed the average time taken to generate a sample on a single core. The results of these experiments for a representative subset of benchmarks are shown in Table 1. The columns in this table give the benchmark name, the number of variables and clauses, the size of the sampling set, the success probability of UniGen2, and finally the average runtime per sample for both UniGen2 and UniGen in seconds. The success probability of UniGen2 was computed as the fraction of calls to GenerateSamples that successfully generated samples.

Table 1. Runtime performance comparison of UniGen2 and UniGen (on a single core)

				UniGen2		UniGen
Benchmark	**#vars**	**#clas**	**|S|**	**Succ. Prob**	**Runtime(s)**	**Runtime(s)**
s1238a_3_2	686	1850	32	1.0	0.3	7.17
s1196a_3_2	690	1805	32	1.0	0.23	4.54
s832a_15_7	693	2017	23	1.0	0.04	0.51
case_1_b12_2	827	2725	45	1.0	0.24	6.77
squaring16	1627	5835	72	1.0	4.16	79.12
squaring7	1628	5837	72	1.0	0.79	21.98
doublyLinkedList	6890	26918	37	1.0	0.04	1.23
LoginService2	11511	41411	36	1.0	0.05	0.55
Sort	12125	49611	52	1.0	4.15	82.8
20	15475	60994	51	1.0	19.08	270.78
enqueue	16466	58515	42	1.0	0.87	14.67
Karatsuba	19594	82417	41	1.0	5.86	80.29
lltraversal	39912	167842	23	1.0	0.18	4.86
llreverse	63797	257657	25	1.0	0.73	7.59
diagStencil_new	94607	2838579	78	1.0	3.53	60.18
tutorial3	486193	2598178	31	1.0	58.41	805.33
demo2_new	777009	3649893	45	1.0	3.47	40.33

Table 1 clearly shows that UniGen2 significantly outperforms UniGen on all types of benchmarks, even when run on a single core. Over the entire set of 200+ benchmarks, UniGen2's runtime performance was about 20× better than that of UniGen on average (using the geometric mean). The observed performance gain can be attributed to two factors. First, UniGen2 generates loThresh (11 in our experiments) samples from every cell instead of just 1 in the case of UniGen. This provides a speedup of about 10×. Second, as explained in Section 4, UniGen2

uses "leapfrogging" to optimize the order in which the values of i in line 2 of Algorithm 2 are chosen. In contrast, UniGen uses a fixed order. This provides an additional average speedup of 2× in our experiments. Note also that the success probability of UniGen2 is consistently very close to 1 across the entire set of benchmarks.

Parallel Speedup. To measure the effect of parallelization on runtime performance, we ran the parallel version of UniGen2 with 1 to 12 processor cores on our benchmarks. In each experiment with C cores, we generated 2500 samples per core, and computed the C-core resource usage as the ratio of the average individual core runtime to the total number of samples (i.e. $C \times 2500$). We averaged our computations over 7 identical runs. The speedup for C cores was then computed as the ratio of 1-core resource usage to C-core resource usage. Figure 1 shows how the speedup varies with the number of cores for a subset of our benchmarks. The figure illustrates that parallel UniGen2 generally scales almost linearly with the number of processor cores.

To obtain an estimate of how close UniGen2's performance is to real-world requirements (roughly 10× slowdown compared to a simple SAT call), we measured the slowdown of UniGen2 (and UniGen) running on a single core relative to a simple SAT call on the input formula. The (geometric) mean slowdown for UniGen2 turned out to be 21 compared to 470 for UniGen. This shows that UniGen2 running in parallel on 2–4 cores comes close to matching the requirements of CRV in industrial practice.

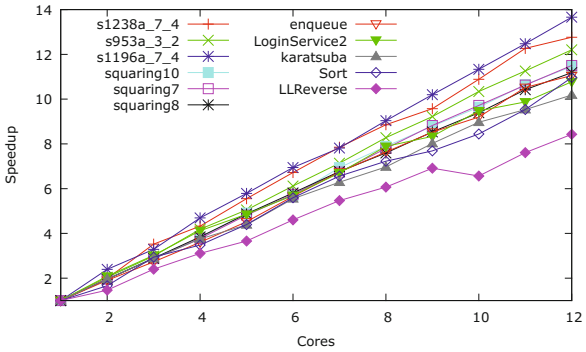

Fig. 1. Effect of parallelization on the runtime performance of UniGen2

Uniformity Comparison. To measure the quality of the distribution generated by UniGen2 and parallel UniGen2 in practice, we implemented an *ideal sampler*, henceforth denoted as IS. Given a formula F, the sampler IS first enumerates all witnesses in $R_{F\downarrow S}$, and then picks an element of $R_{F\downarrow S}$ uniformly at random. We compared the distribution generated by IS with that generated by UniGen2 run sequentially, and with that generated by UniGen2 run in parallel on 12 cores.

In the last case, the samples generated by all the cores were aggregated before comparing the distributions. We had to restrict the experiments for comparing distributions to a small subset of our benchmarks, specifically those which had less than 100,000 solutions. We generated a large number N ($\geq 4 \times 10^6$) of samples for each benchmark using each of IS, sequential UniGen2, and parallel UniGen2. Since we chose N much larger than $|R_{F\downarrow S}|$, all witnesses occurred multiple times in the list of samples. We then computed the frequency of generation of individual witnesses, and grouped witnesses appearing the same number of times together. Plotting the distribution of frequencies — that is, plotting points (x, y) to indicate that each of x distinct witnesses were generated y times — gives a convenient way to visualize the distribution of the samples. Figure 2 depicts this for one representative benchmark (case110, with 16,384 solutions). It is clear from Figure 2 that the distribution generated by UniGen2 is practically indistinguishable from that of IS. Furthermore, the quality of the distribution is not affected by parallelization. Similar observations also hold for the other benchmarks for which we were able to enumerate all solutions. For the example shown in Fig. 2, the Jensen-Shannon distance between the distributions from sequential UniGen2 and IS is 0.049, while the corresponding figure for parallel UniGen2 and IS is 0.052. These small Jensen-Shannon distances make the distribution of UniGen2 (whether sequential or parallel) indistinguishable from that of IS (See Section IV(C) of [13]).

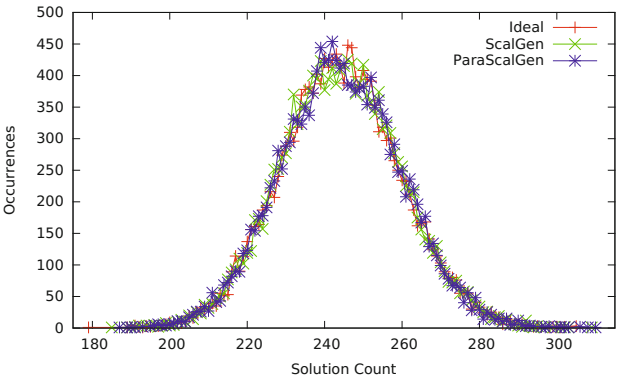

Fig. 2. Uniformity comparison between an ideal sampler (IS), UniGen2, and parallel UniGen2. Results from benchmark 'case110' with $N = 4 \cdot 10^6$.

7 Conclusion

Constrained-random simulation has been the workhorse of functional verification for the past few decades. In this paper, we introduced a new algorithm, UniGen2, that outperforms state-of-the-art techniques by a factor of about 20×. UniGen2

trades off independence of samples for speed while still providing strong guarantees of discovering bugs with high probability. Furthermore, we showed that the parallel version of UniGen2 achieves a linear speedup with increasing number of cores. This suggests a new paradigm for constrained-random verification, wherein we can obtain the required runtime performance through parallelization without losing guarantees of effectiveness in finding bugs.

References

1. CryptoMiniSAT, http://www.msoos.org/cryptominisat2/
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. Bening, L., Foster, H.: Principles of verifiable RTL design – A functional coding style supporting verification processes. Springer (2001)
4. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of SAT witnesses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 608–623. Springer, Heidelberg (2013)
5. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 200–216. Springer, Heidelberg (2013)
6. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT-witness generator. In: Proc. of DAC, pp. 1–6 (2014)
7. Dechter, R., Kask, K., Bin, E., Emek, R.: Generating random solutions for constraint satisfaction problems. In: AAAI, pp. 15–21 (2002)
8. Eager, D.L., Zahorjan, J., Lazowska, E.D.: Speedup versus efficiency in parallel systems. IEEE Trans. on Computers 38(3), 408–423 (1989)
9. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Embed and project: Discrete sampling with universal hashing. In: Proc. of NIPS (2013)
10. Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: Proc. of ISCA, pp. 365–376 (2011)
11. Gogate, V., Dechter, R.: A new algorithm for sampling CSP solutions uniformly at random. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 711–715. Springer, Heidelberg (2006)
12. Gomes, C.P., Sabharwal, A., Selman, B.: Near uniform sampling of combinatorial spaces using XOR constraints. In: Proc. of NIPS, pp. 670–676 (2007)
13. Grosse, I., Bernaola-Galván, P., Carpena, P., Román-Roldán, R., Oliver, J., Stanley, E.: Analysis of symbolic sequences using the Jensen-Shannon divergence. Physical Review E 65(4), 41905 (2002)
14. Iyer, M.A.: Race: A word-level ATPG-based constraints solver system for smart random simulation. In: Proc. of ITC, pp. 299–308. Citeseer (2003)
15. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. TCS 43(2-3), 169–188 (1986)
16. Kitchen, N.: Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation. PhD thesis, University of California, Berkeley (2010)
17. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: Proc. of ICCAD, pp. 258–265 (2007)
18. Madras, N.: Lectures on Monte Carlo Methods. Fields Institute Monographs, vol. 16. AMS (2002)

19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of DAC, pp. 530–535. ACM (2001)
20. Nadel, A.: Generating diverse solutions in SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 287–301. Springer, Heidelberg (2011)
21. Naveh, R., Metodi, A.: Beyond feasibility: CP usage in constrained-random functional hardware verification. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 823–831. Springer, Heidelberg (2013)
22. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. In: Proc. of AAAI, pp. 1720–1727 (2006)
23. Sipser, M.: A complexity theoretic approach to randomness. In: Proc. of STOC, pp. 330–335 (1983)
24. Yuan, J., Aziz, A., Pixley, C., Albin, K.: Simplifying Boolean constraint solving for random simulation vector generation. TCAD 23(3), 412–420 (2004)