Fast Sampling of Perfectly Uniform Satisfying Assignments

Dimitris Achlioptas[1,2] **Zayd Hammoudeh**[1(✉)]
& Panos Theodoropoulos[2]

[1]**University of California, Santa Cruz**
Santa Cruz, CA, USA
{dimitris, zayd}@ucsc.edu

[2]**University of Athens**
Athens, Greece
ptheodor@di.uoa.gr

July 10, 2018

Setting

- From Boolean Formula F, select $s \geq 1$ satisfying assignments u.i.r.
- Primary Application: Constrained Random Verification (CRV)
  - Basic Idea: Input test vector into hardware design and verify outputs
  - Problem: Too many valid inputs to test all of them
  - Standard Approach: Select valid test inputs u.a.r.

Key Contribution

SPUR: Perfectly uniform SAT sampler $>400\times$ *faster than the state-of-the-art*

- *We discuss approximate counting in the* ***next session***

Exact Model Counting

sharpSAT

- sharpSAT: State-of-the-art *exact* model counter [1]
    - Developed by Marc Thurley
    - First published at SAT-06 and iteratively improved since.
    - DPLL-based
- Integrates both SAT techniques like CDCL, UP, two-watch literals, VS heuristics and counting specific optimizations e.g., component decomposition, subformula caching
- **New Caching Paradigm**: sharpSAT's primary differentiator

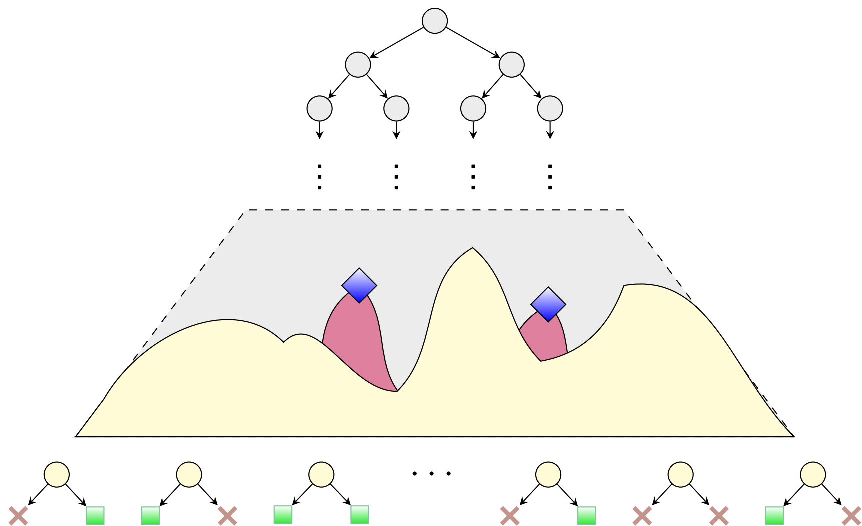
Caching

CDCL: Memoizes UNSAT assignments in new, implied clauses

Caching: Analogue of CDCL for SAT residual formulas

- **Benefit**: Prevents re-analysis of duplicate residual formulas
- **Structure**: Set of key-value pairs
  - *Key*: Boolean formula F'
  - *Value*: Model count of F'
- sharpSAT places all SAT residual formulas, $F(\sigma)$, into the cache
- Before analyzing any $F(\sigma)$, sharpSAT checks if $F(\sigma)$ is in the cache.
  - If yes, sharpSAT uses the stored model count and backtracks
  - Otherwise, sharpSAT analyzes $F(\sigma)$ as normal

The sharpSAT Tree



Reservoir Sampling

Framing the Sampling Problem

Setting: From arbitrary, finite set $\mathcal{P}$ select s elements u.i.r.

- $P_1 \cup P_2 \cup \ldots \cup P_j \cup \ldots P_m$ is an arbitrary partition of $\mathcal{P}$'s elements
- $\mathcal{P}$ is revealed as a stream of disjoint subsets (bundles) $P_1 P_2 \ldots P_m$

Additional Constraints:

1. $|\mathcal{P}|$ and m are unknown
2. "No looking back" on the stream
3. Storage Capacity: s

Solution: **Reservoir Sampling**

- Select s bundles P_j **with replacement** where $\Pr[\text{Select } P_j] \propto |P_j|$
- From each selected bundle P_j, select one member element $p \in P_j$ u.i.r.

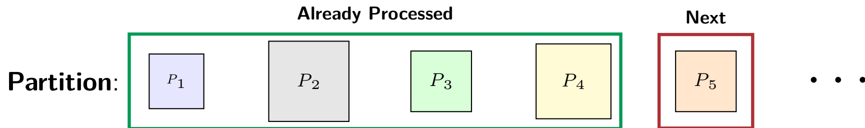
Let's Build an Intuition about Reservoir Sampling

- **Question**: What is a reservoir?
- **Answer**: A multiset (*think memory*) of fixed size s

- **Question**: How can reservoir sampling provide statistical guarantees if $|P|$ and m are unknown?
- **Answer**: *Invariant* – After analyzing each bundle P_k, the statistical guarantees are satisfied w.r.t. $\cup_{j=1}^{k} P_j$ (i.e., *everything seen so far*).

- **Question**: Is reservoir sampling an iterative procedure?
- **Answer**: Yes. Identical iterative procedure for each stream bundle.

An example will help clarify...

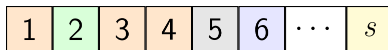


Bent Coin

$$\Pr[\text{Success}] = \frac{|P_{\mathbf{5}}|}{Z}$$

s times

k successes



Reservoir

$$Z = \sum_{i=1}^{\mathbf{5}} |P_i|$$

Total Weight

Bringing Together Model Counting & Reservoir Sampling

Overview

- SPUR – Satisfying Perfectly Uniform Random assignment sampler
- SPUR := sharpSAT + Reservoir Sampling
  - Reservoir sampling performed on the stream of leaves encountered by sharpSAT during its normal execution
  - Probability a particular leaf selected is proportional to its model count
- **Rule of Thumb**: Generating 1,000 samples takes $\sim 10\times$ as long as counting with sharpSAT.
  - **Low overhead** – *If* sharpSAT *can count the models,* SPUR *can sample them*

Implementation

- C++ Based and Open Source
  - **Repository**: https://github.com/ZaydH/spur
- **sharpSAT** + **Reservoir Sampling**: Straightforward idea, complex implementation
- sharpSAT's caching interferes with reservoir sampling
  - Necessitates change to caching paradigm

Experimental Results

Overview

- **State-of-the-Art SAT Sampler**: UniGen2 [2]
  - Developed by Chakraborty, Fremont, Meel, Seshia, Vardi
  - Probabilistic "*almost-independent*," "*almost uniform*"
- **Dataset**: 373 formulas from diverse domains
  - Exclusively **all** the formulas in the UniGen2 and SHARPSAT [1] papers
  - $\#$ *Variables*: 17 to $>$300K
  - $\#$ *Clauses*: 43 to 1.7M
- We compare SPUR and UniGen2 in two sets of experiments:
  - Uniformity of selected samples
  - Execution time to generate 1K samples

Uniformity – Both About the Same in Practice

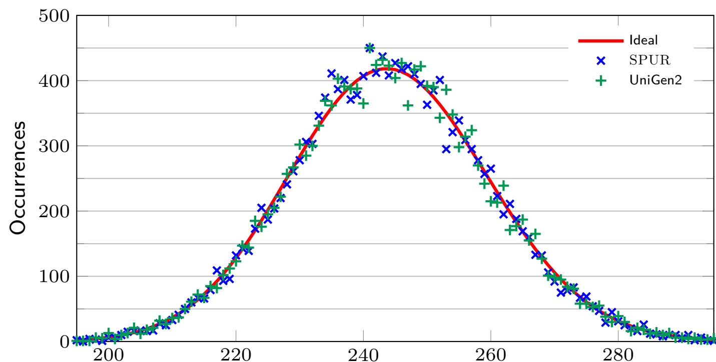
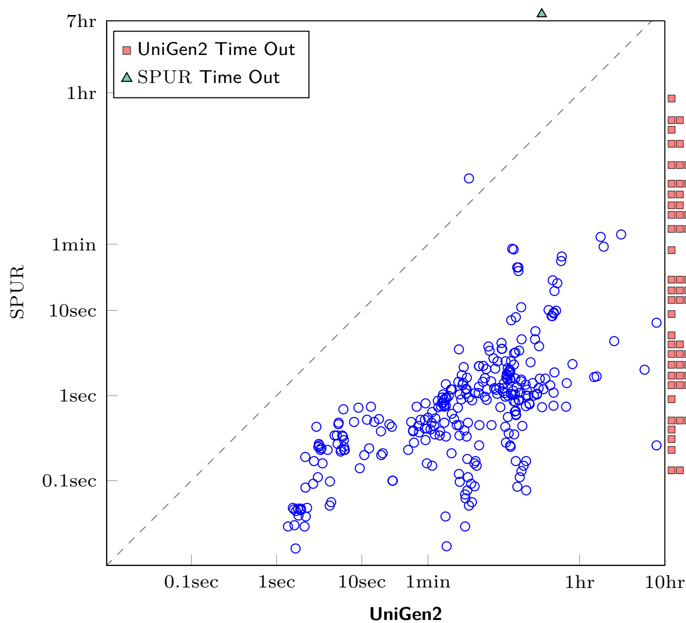


Figure 1: Uniformity comparison between the ideal distribution, SPUR and UniGen2


UniGen2 Time Out
SPUR Time Out
SPUR
7hr
1hr
1min
10sec
1sec
0.1sec
0.1sec
1sec
10sec
1min
1hr
10hr
UniGen2

Execution Time Results Summary

- **Average speed-up**: $>400\times$
  - **Takeaway**: **7 hours $\rightarrow$ 1 minute**
- SPUR is faster than state-of-the-art on 371/373 benchmarks
  - $>10\times$ faster on 369/373 benchmarks
  - $>100\times$ faster on $>2/3$ benchmarks
- On $>70\%$ of benchmarks, SPUR generated 1K samples within $10\times$ the time sharpSAT takes to count
- SPUR is $3\times$ more likely than UniGen2 to successfully generate 1K samples for formulas of >10K variables

Inner Workings of SPUR

Turning SHARPSAT into a Sampler

- **A First Idea**: Enhance SHARPSAT to cache model counts and satisfying assignments
  - Enables trivially building complete assignments even at cache hit leaves

- **Problem**: This simply doesn't work...
  - **Key Observation #1**: "*Model counts are reusable but samples are not.*"
  - Sharing cached satisfying assignments induces dependencies between cache hit leaves in the SHARPSAT tree
  - Inter-leaf dependencies undermines sample independence

- **Key Observation #2**: If only sampling $s = 1$ model, there are no dependencies
  - In only this case, caching satisfying assignments is safe

The sharpSAT Tree Meets Reservoir Sampling

- $\mathcal{P}$: Set of satisfying assignments for Boolean formula F
- $P_j \in \mathcal{P}$: A leaf in the sharpSAT tree
- **Procedure**: Perform reservoir sampling on each leaf P_j
  - k: Number of samples to be selected from P_j
  - If $k > 0$, store k and P_j's partial assignment in the reservoir
- Since the reservoir only has partial assignments, SPUR *must convert them to complete assignments* eventually.

Top-Level SPUR Algorithm

- **Input**: Boolean formula F and sample count s
- **Output**: $\mathcal{S} = \{<s_i, \sigma_i>\}$
  - s_i: Sample count where $\sum_i s_i = s$
  - σ_i: Sample partial variable assignment
- For each $<s_i, \sigma_i> \in \mathcal{S}$:
  a. $F(\sigma_i)$ is empty
     - Set remaining bits in σ_i u.a.r.

  b. $F(\sigma_i)$ is **non**-empty
     - $s_i > 1$: Invoke SPUR recursively with formula $F(\sigma_i)$ and s_i as normal
     - $s_i = 1$: Invoke SPUR recursively with sample caching using **key observation #2**

Advantages:

- Recursion terminates faster
- Preserves sample independence

Conclusions

Review

Our Perfectly Uniform SAT Sampler

$\text{SPUR} := \text{SHARPSAT} +$ Reservoir Sampling

Execution Time

SPUR is $>400\times$ faster than the state-of-the-art.

Rule of Thumb

Generating 1,000 samples takes $\sim 10\times$ as long as counting with SHARPSAT.

Source Code

SPUR – Satisfying Perfectly Uniform Random assignment sampler

https://github.com/ZaydH/spur

Rule of Thumb

Generating 1,000 samples takes $\sim 10\times$ as long as counting.

Appendix & Backup Slides

Summary of Topics

1. Exact Model Counting
   - sharpSAT
2. Reservoir Sampling
3. Bringing Together Model Counting & Reservoir Sampling
4. Experimental Results
   - Sampler Uniformity
   - Execution Time Comparison
5. Inner Workings of SPUR
6. Conclusions
7. Appendix & Backup Slides
8. Experimental Results
9. References

Experimental Results

Selected Results

Benchmark	#Var	#Clause	$\frac{\text{SPUR}}{\text{sharpSAT}}$	UniGen2 (sec)	SPUR (sec)	Speedup
case5	176	518	19.1	633	0.84	753
registerlesSwap	372	1,493	7.0	28,778	0.26	110,684
s953a_3_2	515	1,297	13.4	1139	1.03	1,105
s1238a_3_2	686	1,850	7.0	610	2.31	264
s832a_15_7	693	2,017	13.5	56	0.81	69
case_1_b12_2	827	2,725	1.4	689	29	23
squaring30	1,031	3,693	3.7	1,079	4.58	235
27	1,509	2,707	1.0	99	0.017	5,823
squaring16	1,627	5,835	1.9	11,053	78	141
squaring7	1,628	5,837	1.4	2,185	38	57
111	2,348	5,479	1.0	163	0.029	5,620
51	3,708	14,594	1.5	714	0.11	6,490
32	3,834	13,594	1.0	181	0.051	3,549
70	4,670	15,864	1.0	196	0.056	3,500
7	6,683	24,816	1.0	173	0.077	2,246
Pollard	7,815	41,258	6.0	181	355	0.51
17	10,090	27,056	1.6	192	0.092	2,086
20	15,475	60,994	2.7	289	2.05	140
reverse	75,641	380,869	6.2	TIMEOUT	2.66	$>$13,533

References

References I

[1] M. Thurley, “sharpSAT: Counting models with advanced component caching and implicit BCP,” in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT-06, pp. 424–429, 2006.

[2] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, “On parallel scalable uniform SAT witness generation,” in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS-15, pp. 304–319, 2015.