Backtracking Search for Optimal Parameters of a PLL-based True Random Number Generator

Brice Colombier, Nathalie Bochard, Florent Bernard, Lilian Bossuet
Univ Lyon, UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023, SAINT-ÉTIENNE, France
{b.colombier, nathalie.bochard, florent.bernard, lilian.bossuet}@univ-st-etienne.fr

Abstract—**The phase-locked loop-based true random number generator (PLL-TRNG) extracts randomness from clock jitter. It is an interesting construct because it comes with a stochastic model, making it certifiable by certification bodies. However, bringing it to good performance is difficult since it comes with multiple parameters to tune. This article proposes to use backtracking to determine these parameters. Compared to existing methods, based on genetic algorithms or exhaustive search of a feasible set of parameters, backtracking has several advantages. Indeed, since this method is expressible by constraint programming, it provides very good readability. Constraints can be specified in a very straightforward and maintainable way. It also exhibits good performance and generates PLL-TRNG configurations rapidly. Finally, it allows to integrate new exploratory design constraints for the PLL-TRNG very easily. We provide experimental results with a PLL-TRNG implemented on three FPGA families that come with different physical constraints, showing that the method allows to find good parameters for every one of them. Moreover, we were able to obtain configurations that lead to an increase 59 % in throughput and 82 % in jitter sensitivity on average, thereby generating random numbers of higher quality at a faster rate. This approach also paves the way for new design exploration strategies for PLL-TRNG. The source code of our implementation is open source and available online for reproducibility and reuse.**

I. Introduction

Random number generators are a cornerstone of many security applications and are used to generate encryption keys, initialisation vectors or masks for implementations of block ciphers secure against side-channel attacks. From certification requirements arose the need for true random number generators (TRNGs) with a stochastic model that specifies where randomness is extracted from [1], [2]. Among them, the PLL-TRNG, based on phase-locked loop(s) [3], [4], is a promising candidate since it has been modeled precisely [5]. However, since this TRNG has multiple parameters to tune, finding a suitable set of parameters that satisfy the throughput and entropy requirements for a given application is complicated.

To achieve this goal, we propose in this article to use *backtracking*, expressed in the framework of constraint programming. As a subset of declarative programming [6], constraint programming aims at finding possible values for the variables of a problem that satisfy a set of constraints. The problem of finding good parameters for a PLL-TRNG can be directly expressed in this framework. Indeed, when searching for optimal PLL-TRNG parameters, three groups of constraints can be identified. The first group are the physical constraints of the PLL itself, *i.e.* the minimum and maximum frequencies for the inputs and outputs of its functional blocks. The second group is related to the TRNG setting for the PLL. In a PLL-TRNG, the PLLs must be set up appropriately to obtain random numbers, as described in [3]. Finally, the third group is related to the system requirements, namely the minimum throughput and the entropy. This again is directly expressible by constraints.

From the point of view of a designer who needs to implement a PLL-TRNG, this approach has several benefits. First of all, since the framework of constraint programming is used, expressing constraints is very easy and readable. Moreover, it makes it simple to maintain and modify. Second, the performance of parameters search is very good and in the order of magnitude of the results found in the state-of-the-art [7]. Finally, we benefit from the ecosystem of constraint programming. Even though we use the simple backtracking method in our case to actually perform the search, more efficient methods developed for constraint programming in general could be applied to this use case in particular.

The contributions of this article are the following. First, we describe the constraint programming framework and the use of backtracking to search for PLL-TRNG parameters. Second, we detail the possibilities opened by this new search method. Indeed, expressing constraints allows to take into account new exploratory design constraints for the PLL-TRNG architecture. Third, we relax one design constraint about the output frequency of one of the PLL used in the PLL-TRNG. Indeed, we show that only one of the two PLLs is limited by the surrounding design, not both as said in [7].

The remainder of this article is organised as follows. In Section II, we start by reviewing existing methods to search for good PLL-TRNG parameters. In Section III, we recall the architecture of the PLL-TRNG and its associated parameters. We then show in Section IV how to implement the parameters search with backtracking by constraint programming. In Section V, we provide experimental results of parameters search on three different FPGA families that come with different physical constraints. We discuss the possibilities offered by the constraint programming approach in Section VI before concluding this article in Section VII. The source code related to this work is made open source and available online[1].

[1]https://gitlab.univ-st-etienne.fr/sesam/pll-trng-constraint-programming/tree/master

II. Related work

The problem of finding PLL-TRNG parameters has been studied before. In the original article proposing this TRNG architecture [3], the choice of parameters for the PLLs is done by the expert who designs the system. Given the size of the parameters space, this approach might give good results but will probably miss better configurations available. This trial-and-error process also takes a lot of time.

In [8], Petura *et al.* question this "expert input" approach and derive the parameters of the PLL-TRNG using a genetic algorithm. They indeed find better configurations than previously known, having better throughput and entropy. However, this proposition is limited by the capabilities of genetic algorithms, namely that they are not guaranteed to converge towards a globally optimal solution.

From this observation, Noumon Allini *et al.* proposed an analytical method [7] to search for the PLL-TRNG parameters. For each parameter, they derive exact bounds based on physical constraints found in the technical documentation of the FPGAs they considered, as well as bounds computed for other parameters. Then they adopt a depth-first search strategy, as illustrated by their search algorithm which uses eight nested *for* statements. The order in which these *for* loops are nested is carefully chosen so that the depth-first search is performed optimally by computing the bounds in the correct order since they form a chain of dependency. The outcome of this search is a set of parameters that are physically possible to implement on the target FPGA. These sets of parameters must then be filtered to implement a PLL-TRNG suited to the system requirements. This search method, although computationally efficient, is very rigid since it is required that the chain of dependency between the parameters bounds is verified. If other constraints must be met during the parameters generation, then there is no guarantee that it would still be the case.

Before detailing the search method that we propose based on constraint programming, we recall the architecture of a PLL-TRNG and the associated parameters.

III. PLL-TRNG architecture and parameters

A. PLL parameters

Figure 1 shows a block diagram of a PLL. Grey blocks are the analog part and cannot be parameterised, while the M, N and C integer division coefficients, in white blocks, must be set. These three coefficients are used to obtain the output frequency of the PLL (f_{out}) from the reference frequency (f_{ref}) as given in Equation (1).

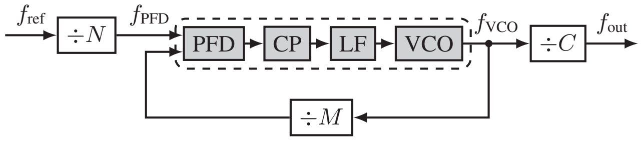


Fig. 1: Block diagram of a PLL (PFD: phase frequency detector, CP: charge pump, LF: loop filter, VCO: voltage-controlled oscillator)

$$f_{\text{out}} = f_{\text{ref}} \times \frac{M}{N \times C} \tag{1}$$

B. PLL-TRNG architecture

The architecture of a PLL-TRNG with two PLLs is given in Figure 2. The output clock of PLL$_0$ is used to sample the output clock of PLL$_1$ by using the coherent sampling technique [5]. Since both those clocks are jittery, then the number of ones sampled after several periods of the output clock of PLL$_0$ is random if the frequencies ratio is correctly chosen. This number is stored in a counter and then used to generate a random number, typically by extracting its least-significant bit.

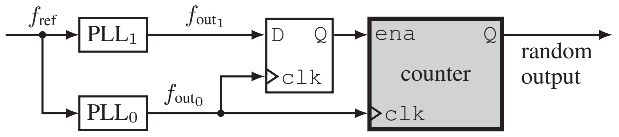


Fig. 2: PLL-based TRNG using two PLLs

From the individual PLL parameters, the global parameters of the PLL-TRNG can be derived. The overall multiplication coefficient K_M is given in Equation (2). The overall division coefficient K_D is given in Equation (3).

$$K_M = M_1 \times N_0 \times C_0 \tag{2}$$

$$K_D = M_0 \times N_1 \times C_1 \tag{3}$$

The figures of merit that are classically used to evaluate a TRNG are its ability to extract randomness and its throughput [9]. For the PLL-TRNG, the ability to extract randomness is expressed by the sensitivity to jitter S, in μs^{-1}, given in Equation (4), where the frequencies f_{out_1} and f_{ref} are in MHz.

$$S = f_{\text{out}_1} \times K_D = f_{\text{ref}} \times M_0 \times M_1 \tag{4}$$

The throughput R, in Mbit s^{-1}, is given in Equation (5), where the frequencies f_{out_0} and f_{ref} are in MHz.

$$R = \frac{f_{\text{out}_0}}{K_D} = \frac{f_{\text{ref}}}{N_0 \times N_1 \times C_0 \times C_1} \tag{5}$$

For a given application, the system requirements are typically given in these terms. The TRNG must supply random bits at a given throughput (R_{min}) with sufficient jitter sensitivity (S_{min}). The minimum sensitivity to jitter to obtain sufficient entropy depends on the target device in which the PLL-TRNG is implemented. For example in [7], for an FPGA implementation, it is said that the minimum sensitivity to jitter must be 0.09 ps^{-1}.

In addition to these requirements, the target design along which the PLL-TRNG is instantiated also has an influence on the set of feasible parameters. Indeed, the output clock of one of the PLLs is used as a general-purpose clock for the rest of the design. Therefore, one of the PLL sees its maximal frequency not only limited by physical constraints but also by the maximum operating frequency of the design. Freeing both PLLs from this constraint would require a third PLL dedicated to the design only, which might not be possible. However, contrary to what is said in [7], one of the PLLs is not subject to this constraint and can run at its maximum operating frequency. The other PLL is then used to clock the rest of the design.

Now that we examined the requirements which are commonly admitted when it comes to PLL-TRNG configuration, we show in the next section how to express them very simply using constraint programming.

IV. Parameters search by constraint programming with backtracking

Constraint programming is a subset of declarative programming [6]. One possible way of solving a problem stated by constraint programming is by backtracking. Given variables and a set of constraints over these variables, this approach iteratively explores the possible solutions. However, contrary to a brute-force approach, possible values for the variables are immediately dropped as soon as one of the constraints is not satisfied. It then *backtracks* to other possible values until all valid values are found. Here, Declaration 1 describes the problem we aim at solving, namely finding PLL-TRNG parameters that satisfy both the physical constraints as well as the application requirements.

Declaration 1 Search for optimal PLL-TRNG parameters with constraint programming

```
1: Instantiate the problem to solve $\mathcal{P}$
   /* Problem variables */
2: Add variable: $M_0 \in \{M_{min}, \dots, M_{max}\} \mid M_0$ odd
3: Add variable: $M_1 \in \{M_{min}, \dots, M_{max}\}$
4: Add variable: $N_0 \in \{N_{min}, \dots, N_{max}\}$
5: Add variable: $N_1 \in \{N_{min}, \dots, N_{max}\} \mid N_1$ odd
6: Add variable: $C_0 \in \{C_{min}, \dots, C_{max}\}$
7: Add variable: $C_1 \in \{C_{min}, \dots, C_{max}\} \mid C_1$ odd
   /* Physical constraints */
8: for $i \in \{0, 1\}$ do
9:     Add constraint: $f_{\mathrm{PFD}_{min}} < \frac{f_{\mathrm{ref}}}{N_i} < f_{\mathrm{PFD}_{max}}$
10:    Add constraint: $f_{\mathrm{VCO}_{min}} < \frac{f_{\mathrm{ref}} \times M_i}{N_i} < f_{\mathrm{VCO}_{max}}$
11:    Add constraint: $f_{\mathrm{out}_{min}} < \frac{f_{\mathrm{ref}} \times M_i}{C_i \times N_i} < f_{\mathrm{out}_{max}}$
12: end for
   /* Constraints related to the TRNG setting */
13: Add constraint: $\mathrm{GCD}(K_M, K_D) = 1$
14: Add constraint: $K_D < K_{D_{max}}$
15: Add constraint: $K_M < K_{M_{max}}$
   /* Application requirements */
16: Add constraint: $\frac{f_{\mathrm{ref}} \times M_0}{C_0 \times N_0} < f_{\mathrm{max}}$
17: Add constraint: $R > R_{min}$
18: Add constraint: $S > S_{min}$
   /* Problem solving */
19: Solve $\mathcal{P}$ to obtain the set(s) of feasible parameters
```

The first step is to instantiate the problem to solve (line 1) before adding the variables to determine (lines 2 to 7). For each variable, we specify a range in which the search is carried out. The range for the M_0, N_1 and C_1 variables must start at an odd number and continue with a step of 2, so that K_D, defined in Equation (3), is odd. The minimum and maximum values are given in the FPGA technical documentation. However, tighter bounds can be derived, as shown in [7]. We reuse those bounds here. However, we compute them statically when initialising the variable domains, not dynamically for each parameters set.

Then, we declare the physical constraints for both PLLs, namely the range of possible frequencies for the input of the phase frequency detector (line 9), the output of the voltage controlled oscillator (line 10) and the output of the PLL (line 11).

Next, we specify the conditions on the global multiplication and division coefficients K_M and K_D to perform coherent sampling. K_M and K_D must be co-prime (line 13). As stated in [5], the jitter accumulation time, related to K_D, must be within a certain limit. From this limit, an upper bound for K_D, denoted as $K_{D_{max}}$, is derived in [5]. The associated constraint is shown on line 14. Similarly, an upper limit is also set for K_M, as shown on line 15. We set $K_{D_{max}} = 500$ and $K_{M_{max}} = 1000$.

Finally, we declare the constraints related to the system requirements, namely the minimum throughput R_{min} (line 17) and the minimum sensitivity S_{min} (line 18). We also add a constraint on the maximum output frequency of PLL_0 so that it does not exceed the maximum frequency of the surrounding design (line 16). We then solve the problem (line 19) by backtracking to get the set(s) of feasible parameters.

After running the program, we obtain one or several sets of suited parameters or no suited parameters at all. In the former case, the designer can pick the one that is best suited to the system or run the program again with more stringent requirements to reduce the number of sets of parameters. In the latter case, the designer must relax the system requirements until at least one set of suited parameters is found. Due to the computational efficiency of the method, such a trial-and-error approach is actually very practical and quickly converges towards a good solution.

V. Experimental results

A. Implementation

To implement Declaration 1, we used the Python programming language and the python-constraint package[2]. We ran it on a desktop computer embedding a 6-core CPU operating at 2.80 GHz and 32 GB of RAM.

When actually implementing Declaration 1, one must be careful about the order in which constraints are added to the problem. Indeed, for backtracking to be efficient, it should be able to evaluate partial solutions rapidly without having to satisfy every constraint for each of them. Otherwise, it amounts to brute-force search. Therefore, constraints should

[2]https://labix.org/python-constraint

be sorted according to their computational complexity before being added to the problem. Computationally-easy constraints should be added first, while computationally-hard constraints should be added last. This way, when exploring the feasible sets of parameters, computationally-hard constraints are only evaluated if computationally-easy constraints were satisfied. In our case, for example, it is important that computationally-hard constraints like $\text{GCD}(K_M, K_D) = 1$ are added after computationally-easy constraints like $K_D < K_{D_{max}}$.

B. Generated configurations

We apply the method to PLL-TRNGs implemented in three FPGA families. For each FPGA family, the possible values for the parameters of the PLL are different. These are detailed in Table I, taken from [7]. We can see that the three families have quite different possible values for the PLL parameters, leading to different optimal configurations for a PLL-TRNG implemented in them. For a fair comparison with [7], we set the input frequency (f_{ref}) of the PLL-TRNG to 125 MHz and the maximum frequency of the design (f_{max}) to 250 MHz.

TABLE I: Ranges of possible values for the PLL parameters and frequencies for the selected FPGA families (from [7])

Parameter	Intel Cyclone V		Xilinx Spartan 6		Microsemi SmartFusion 2	
	Min.	Max.	Min.	Max.	Min.	Max.
f_{ref} [MHz]	5	500	19	540	1	200
M	1	512	1	64	1	4194304
N	1	512	1	52	1	16384
C	1	512	1	128	1	255
f_{PFD} [MHz]	5	325	19	500	1	200
f_{VCO} [MHz]	600	1300	400	1080	500	1000
f_{out} [MHz]	0	460	3.125	400	20	400

Experimental results are given in Table II and compared to [7]. For each FPGA family, two configurations are saved. The first one, referred to as "Max. R", is the one with the best throughput for a sensitivity to jitter S of at least 0.09 ps^{-1}. The second one, referred to as "Max. S", is the one with the best sensitivity to jitter for a throughput R of at least 0.5 Mbit s^{-1}.

For every possible configuration, we see a systematic improvement over the configurations given in [7]. For the "Max. R" configuration, the throughput is increased by at least 25 %, for Microsemi SmartFusion 2, up to 88 % for Xilinx Spartan 6. On average, the best throughput increased by 59 %. For the "Max. S" configuration, the sensitivity to jitter is increased by at least 62 %, for Microsemi SmartFusion 2, up to 101 % for Intel Cyclone V. On average, the best sensitivity to jitter increased by 82 %. This improvement was obtained thanks to the relaxation of the maximum operating frequency of the second PLL. Taking into account this modification, the method described in [7] would find these configurations too. However, as mentioned before, our method could easily integrate new constraints, which is not the case for the method in [7] that would require to completely redesign the algorithm.

Another interesting point of view on these performance metrics is to plot the sensitivity to jitter against the throughput for the best (R, S) pairs. We call *best* (R, S) pairs the ones for which the sensitivity to jitter is maximum for a given throughput. This is depicted in Figure 3. As we can observe, the best (R, S) pairs lie on a curve which equation is of the form ($f : x \mapsto \frac{a}{x},\ a \in \mathbb{R}$). The a coefficient is dependent on the FPGA family and is equal to 0.0999 for Intel Cyclone V, 0.0893 for Xilinx Spartan 6 and 0.0816 for Microsemi SmartFusion 2. The possibility to use this coefficient when searching for optimal PLL-TRNG parameters is discussed in Section VI-C.

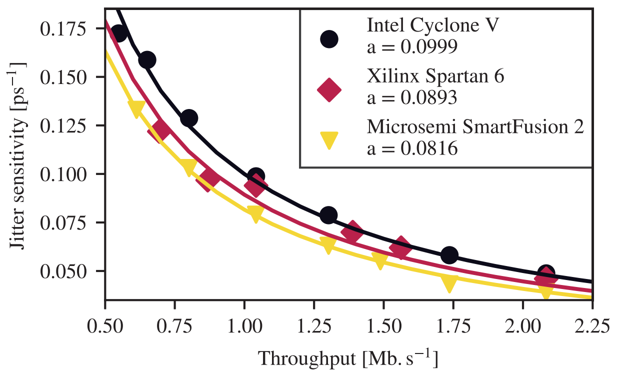


Fig. 3: Scatter plot and curve fit with a reciprocal function ($f : x \mapsto \frac{a}{x},\ a \in \mathbb{R}$) of the jitter sensitivity against the throughput for several PLL-TRNG configurations for each FPGA family

Due to the different ranges of values for each FPGA family, the time to generate the configurations varies. It takes less than one second for Xilinx Spartan 6, a few seconds for Intel Cyclone V and around ten seconds for Microsemi SmartFusion 2. It also depends on the application requirements, namely the throughput and sensitivity to jitter. These timings are of the same order of magnitude as those of the sate-of-the-art method proposed in [7].

After obtaining these configurations, we performed the implementation and acquired random data. This is evaluated in the next section.

C. Random data evaluation with statistical tests

In order to assess the quality of the random numbers generated by each configuration, we conducted the following experiment. We implemented each of the aforementioned configurations in the associated FPGA family. For each of them, we generated 2 MB of random data and then used the AIS-31 [1] and NIST 800-90B [2] statistical tests.

The results returned by these tests are split in two here, and presented in Table III. For each test, the "PASS" column indicates if the tests passed successfully. The second column, "entropy", provides the result of the entropy estimation performed by the tests. AIS-31 statistical tests employ a method described in [10] to evaluate the entropy per bit. The entropy evaluated here is the Shannon entropy (see Equation (6), where

TABLE II: Best set of PLL-TRNG parameters found by backtracking and comparison with the results from [7]

Configuration	PLL_0 (M_0, N_0, C_0)	PLL_1 (M_1, N_1, C_1)	(K_M, K_D)	f_{out_0} [MHz]	f_{out_1} [MHz]	Results from [7] R [Mbit s^{-1}]	Results from [7] S [ps^{-1}]	This work R [Mbit s^{-1}]	This work S [ps^{-1}]
Intel Cyclone V									
Max. R	(79, 8, 5)	(10, 1, 3)	(400, 237)	246.875	416.667	0.631	0.0913	**1.042** (+65%)	0.099
Max. S	(159, 16, 5)	(10, 1, 3)	(800, 477)	248.438	416.667	0.548	0.0988	0.521	**0.199** (+101%)
Xilinx Spartan-6									
Max. R	(47, 6, 4)	(16, 5, 1)	(384, 235)	244.792	400.000	0.555	0.0913	**1.042** (+88%)	0.094
Max. S	(31, 4, 4)	(43, 5, 3)	(688, 465)	242.188	358.333	0.555	0.0913	0.521	**0.167** (+83%)
Microsemi SmartFusion 2									
Max. R	(103, 13, 4)	(8, 1, 3)	(416, 309)	247.596	333.333	0.641	0.090	**0.801** (+25%)	0.103
Max. S	(159, 20, 4)	(8, 1, 3)	(640, 477)	248.438	333.333	0.548	0.098	0.521	**0.159** (+62%)

X is the random variable and p_i refers to the probability of the outcome i).

$$H_1(X) = -\sum_{i=1}^{n} p_i \log p_i \qquad (6)$$

NIST 800-90B statistical tests use ten different entropy estimation methods and keep the minimum of all the estimates. However, in this case, it is the min-entropy which is estimated (see Equation (7)). The min-entropy is always lower than the Shannon entropy.

$$H_\infty(X) = \min_i(-\log p_i) \qquad (7)$$

We repeated this experiment 50 times for each configuration for better statistical significance. The results are presented in Table III. For each configuration, we considered that the statistical tests passed successfully if it was the case for all 50 random data acquisitions. The entropy value given in Table III is the average of the estimated entropy over all 50 data sets.

For each configuration, the statistical tests passed successfully. Moreover, the Shannon entropy per bit was estimated to be between 0.99999 and 1, well above the 0.997 threshold fixed by the AIS-31 standard [1]. For NIST 800-90B tests, the min-entropy was estimated to be between 0.98950 and 0.98990, which again shows very good randomness quality. Therefore, we can conclude that all PLL-TRNG configurations are valid and generate random numbers of high entropy.

VI. Exploratory design strategies for the PLL-TRNG architecture

The framework of constraint programming allows to specify new constraints very easily to search for optimal PLL-TRNG parameters. We mention here some new exploratory design strategies which could be used to generate parameters for the PLL-TRNG architecture.

A. PLL disparity

The PLL-TRNG can be considered as a differential architecture. Indeed, by using two PLLs and exploiting the jitter between the two generated clocks as the source of randomness, we assume that any global perturbation will have a similar effect on both PLLs. Since the PLL-TRNG architecture is differential, these global effects will be cancelled out and have a lower impact on the randomness quality.

TABLE III: Experimental validation of the parameters with AIS-31 and NIST 800-90B statistical tests

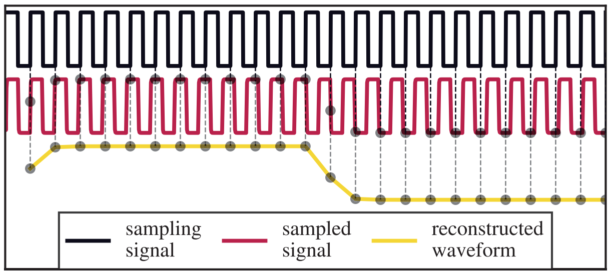
Configuration	AIS-31 PASS	AIS-31 entropy	NIST 800-90B PASS	NIST 800-90B entropy
Intel Cyclone V				
Max. R	✓	1	✓	0.98953
Max. S	✓	1	✓	0.98989
Xilinx Spartan-6				
Max. R	✓	1	✓	0.98990
Max. S	✓	0.99999	✓	0.98976
Microsemi SmartFusion 2				
Max. R	✓	1	✓	0.98950
Max. S	✓	0.99999	✓	0.98959

However, for both PLLs to be affected similarly by global perturbations, their M, N and C parameters should be as close as possible. Therefore, the $\frac{M_0}{M_1}$, $\frac{N_0}{N_1}$ and $\frac{C_0}{C_1}$ ratios should be close to 1. In the framework of constraint programming, this constraint can be quickly added to the parameters search.

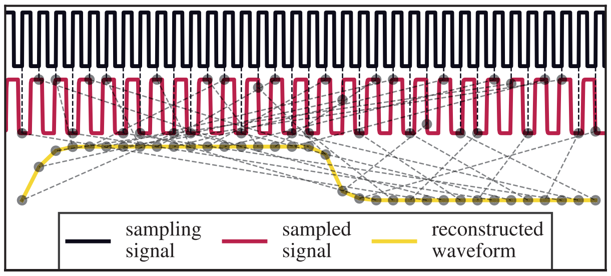
B. Distance between points in the reconstructed waveform

The coherent sampling principle used in the PLL-TRNG aims at reconstructing a periodic waveform by sampling it at a specific sampling frequency. The ratio between the sampling frequency and the periodic waveform frequency is a rational number. This is illustrated in Figure 4 where two different ratios are used. The counter of the PLL-TRNG (see Figure 2) stores the number of ones found in the reconstructed waveform. Since the clocks are jittery, some points are not always at a high or low logic level but lie on the edge of the reconstructed waveform. They correspond to samples taken near the clock edge, that are affected by the jitter and contain randomness. As mentioned in Section III-B, the least-significant bit of the counter is used as a random bit. This bit is by definition the exclusive-OR of all the samples.

If jittery samples found on the edge were sampled successively, as shown in Figure 4a, that could imply that they are not independent, affecting the randomness quality. Conversely, they could be sampled far from one another, at a minimum distance d_{min}, as shown in Figure 4b. This could prevent correlation between jittery samples. This intuition should be studied in future works. Again, filtering configurations according to the minimum distance between samples in the reconstructed waveform is easy by constraint programming.



(a) Consecutive points in the reconstructed waveform were sampled consecutively



(b) Consecutive points in the reconstructed waveform were sampled with a minimum distance $d_{min} = 10$

Fig. 4: Waveform reconstruction by coherent sampling

C. $R \times S$ product

As graphically shown in Figure 3, optimal (R, S) pairs lie on the curve of a reciprocal function. Therefore, for optimal (R, S) pairs, the $R \times S$ product is constant for each FPGA family. Knowing the target family in which the PLL-TRNG must be implemented, a designer can then use this product to generate configurations. For a given throughput, only configurations with the highest sensitivity to jitter will be generated. Since the throughput is usually a primary criterion when implementing a TRNG, a designer can then generate optimal configurations easily without having to deal with sensitivity to jitter.

VII. Conclusion and perspectives

In this article, we propose to use constraint programming to search for optimal parameters for a PLL-based TRNG. We also relax one design constraint about this TRNG architecture, namely that only one of the two PLLs sees its output frequency constrained by the surrounding digital design. Using the back-tracking method, we were able to generate sets of parameters for maximum throughput or maximum sensitivity to jitter for three different FPGA families that improve significantly over state-of-the-art methods. In comparison, we obtain an increase in throughput of 59 % and increase in sensitivity to jitter of 82 % on average. We assess the randomness quality of data obtained with the generated configurations with AIS-31 and NIST 800-90B statistical tests.

In future works, integrating more constraints will be much easier in the framework of constraint programming than with previous approaches. As sketched out in the last section of the article, exploratory strategies taking into account more constraints are now possible. In particular, a better integration of the physical model for randomness extraction could help generate faster PLL-TRNG that meet the randomness quality requirements.

Acknowledgements

This work was carried out in the framework of the FUIAAP22-Project PILAS supported by Bpifrance.

The authors would like to thank Elie Noumon Allini for the discussions about this work.

References

[1] W. Killmann and W. Schindler, "A proposal for: Functionality classes for random number generators," 2011. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_pdf
[2] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, "SP 800-90B: Recommendation for the entropy sources used for random bit generation," 2018. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-90b/final
[3] V. Fischer and M. Drutarovský, "True random number generator embedded in reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*, ser. Lecture Notes in Computer Science, B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523. Redwood Shores, CA, USA: Springer, Aug. 2002, pp. 415–430.
[4] M. Drutarovský, M. Simka, V. Fischer, and F. Celle, "A simple PLL-based true random number generator for embedded digital systems," *Computers and Artificial Intelligence*, vol. 23, no. 5, pp. 501–515, 2004.
[5] F. Bernard, V. Fischer, and B. Valtchanov, "Mathematical model of physical RNGs based on coherent sampling," *Tatra Mountains - Mathematical Publications*, vol. 45, pp. 1–14, 2010.
[6] D. E. Knuth, "Backtrack programming," in *The Art of Computer Programming*. Addison-Wesley, 2019, vol. 4, ch. 7.2.2.
[7] E. Noumon Allini, O. Petura, V. Fischer, and F. Bernard, "Optimization of the PLL configuration in a PLL-based TRNG design," in *Design, Automation & Test in Europe Conference*. Dresden, Germany: IEEE, Mar. 2018, pp. 1265–1270.
[8] O. Petura, U. Mureddu, N. Bochard, and V. Fischer, "Optimization of the PLL based TRNG design using the genetic algorithm," in *International Symposium on Circuits and Systems*. Baltimore, MD, USA: IEEE, May 2017, pp. 1–4.
[9] O. Petura, U. Mureddu, N. Bochard, V. Fischer, and L. Bossuet, "A survey of AIS-20/31 compliant TRNG cores suitable for FPGA devices," in *International Conference on Field Programmable Logic and Applications*, P. Ienne, W. A. Najjar, J. Anderson, P. Brisk, and W. Stechele, Eds. Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–10.
[10] J.-S. Coron and D. Naccache, "An accurate evaluation of Maurer's universal test," in *Selected Areas in Cryptography*, S. E. Tavares and H. Meijer, Eds., vol. 1556. Kingston, Ontario, Canada: Springer, Aug. 1998, pp. 57–71.