From Hardware to Software Synthesis of Linear Feedback Shift Registers

Cédric Lauradoux

INRIA Rocquencourt, Projet CODES
78153 Le Chesnay Cedex, FRANCE
cedric.lauradoux@inria.fr

Abstract

Linear Feedback Shift Registers (LFSRs) have always received considerable attention in computer science especially in coding theory and in cryptography. The scope of applications of LFSRs is wide: data scrambling, spread spectrum, build in self tests (BISTs)... They have to be implemented either in hardware or in software. Unlike hardware, software applications have not been very popular. The main reason is that, even if the LFSR synthesis in software is very similar to the LFSR synthesis on Xilinx FPGA, the overall processing is parallel in hardware while it is almost sequential in software, leading to low throughput implementations. If the naive LFSR implementation is in favor of hardware, increasing the number of LFSR steps computed at the same time can considerably improve software implementation. For instance, we obtain a 103 speedup factor for a 128-bit LFSR on 64-bit processors. Unfortunately, this cannot be obtain for all LFSRs. We here describe how LFSR parameters must be chosen to obtain an efficient implementation.

1. Introduction

A common feature among various algorithms for communication is the computation on bit-level data. However, current General Purpose Processors (GPPs) and Digital Signal Processors (DSPs) cannot operate directly on bit-level data without packing and unpacking the bits in memory. GPPs and DSPs operate on bit-level data in large register-wide ALUs. Hence the throughputs of those algorithms are low since many operations are required for packing/unpacking within a register and since the computation itself produces only a single bit. A typical class of algorithms working at bit-level is the class of Linear Feedback Shift Registers (LFSRs). LFSRs are present in every coding scheme as scramblers and in many stream ciphers because they produce sequences with good statistical properties, and they can be easily analyzed. Moreover they may have a low-cost realization in hardware.

An attempt to improve the data-path usage consists in computing several LFSR steps at the same time. This solution is known as leap-forwarding in hardware synthesis. It was first used to improve the efficiency of the software stream-cipher SSC2 [10]. Later, Chowdhury and Maitra partially analyzed software leap-forwarding in [3] and [4].

In this paper, we address the problem of generating efficient implementations of LFSRs for any parameters. Leap-forward implementations can be very efficient on a 64-bit processor: we obtain a 103 speedup factor for some LFSR parameters. Moreover, we investigate the link between the LFSR definitions and their implementation characteristics (throughput, code size). We have developed a new tool for generating optimized LFSR implementation in C. This new tool is available at [7].

The next section introduces the different representations of an LFSR. Section 3 compares the synthesis on Xilinx FPGA and on an r-bit processor. We detail the properties of leap-forward implementation and we point out why it is suitable for software and less for Xilinx FPGA. In Section 4, we provide an overview of LFSRs performance in software using the previously discussed leap-forward implementations. Most notably, we exhibit the influence of the properties of the characteristic polynomial on the LFSR performance in software.

2 Linear Feedback Shift Register representations

Let α be a root of a primitive polynomial P:

$$P(X) = X^m + a_{m-1}X^{m-1} + \cdots + a_1X + a_0 \quad (1)$$

in $\mathbf{F}_q[X]$. Then $\mathbf{F}_{q^m}$ can be defined as the quotient $\mathbf{F}_q[X]/P(X)$. As shown in [9], the multiplication of an

1-4244-0910-1/07/$20.00

element $s \in \mathbf{F}_{q^m}$ by the root α can be done easily in the polynomial basis $(1, \alpha, \alpha^2, \cdots, \alpha^{m-1})$ or in the dual basis $(\beta_0, \beta_1, \cdots, \beta_{m-1})$ defined by:

$$\begin{array}{ll} Tr(\beta_i \alpha^j) = 1 & \text{if } i = j \\ Tr(\beta_i \alpha^j) = 0 & \text{if } i \neq j. \end{array} \quad (2)$$

Independently from the basis representation, the device that implements this multiplication is called a Linear Feedback Shift Register (LFSR) of length m over $\mathbf{F}_q$. It consists of m delay cells of $\lceil \log_2(q) \rceil$ bits. The LFSRs are split into two families of devices, the so-called Fibonacci and Galois representations.

The Galois representation corresponds to the multiplication in the primal basis. Let s be expressed in the polynomial basis by:

$$s = s_{m-1}\alpha^{m-1} + \cdots + s_1\alpha + s_0. \quad (3)$$

The product αs is given by:

$$\alpha s = s_{m-1}\alpha^m + s_{m-2}\alpha^{m-1} + \cdots + s_1\alpha^2 + s_0\alpha.$$

Thus, as α is a root of P, we obtain:

$$\alpha s = (s_{m-1}a_{m-1} + s_{m-2})\alpha^{m-1} + \cdots + s_{m-1}. \quad (4)$$

The above equation is the main description of the Galois device feedback computation. In Figure 1, a Galois device is represented for the field $\mathbf{F}_{2^4}$ with $P(X) = X^4 + X + 1$.

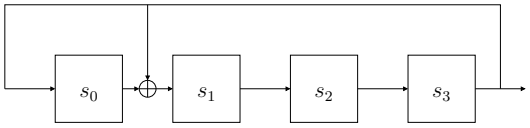


Figure 1. Galois LFSR setup

The multiplication in the dual basis is given by the Fibonacci representation. Let s be expressed in the dual basis by:

$$\begin{array}{l} s = s'_{m-1}\beta_{m-1} + \cdots + s'_1\beta_1 + s'_0\beta_0 \\ \text{with } s'_j = Tr(s\alpha^j)\ , j = 0, \cdots, m-1 \end{array} \quad (5)$$

Then, the components of αs in the dual basis are given by:

$$(\alpha s)'_j = Tr(s\alpha^{j+1}) \quad (6)$$

We obtain:

$$\begin{aligned} (\alpha s)'_j &= s'_{j+1},\ j = 0, 1, \cdots, m-2 \quad (7) \\ (\alpha s)'_{m-1} &= Tr(s\alpha^m) \\ &= a_{m-1}s'_{m-1} + \cdots + s'_0 \quad (8) \end{aligned}$$

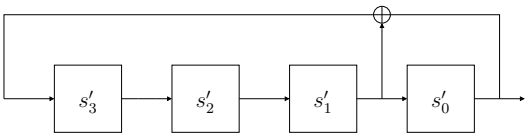


Figure 2. Fibonacci LFSR setup

Equations 7 and 8 are the main description of the feedback computation for Fibonacci device. In Figure 2, a Fibonacci device is represented for the field $\mathbf{F}_{2^4}$ with $P(X) = X^4 + X + 1$. The polynomial P is called the characteristic polynomial of the LFSR.

The properties of LFSR have been deeply studied [6, 8, 9]. Both Galois and Fibonacci setups generate the same set of sequences, even if a given output sequence corresponds to different initial states in both setups. Throughout this paper, we will only refer to LFSR over $\mathbf{F}_2$ and most of our techniques will be only described for Fibonacci setup. In this field, the characteristic polynomial can be written as:

$$\begin{array}{c} P(X) = 1 + X^{d_1} + X^{d_2} + \cdots + X^{d_{w-2}} + X^m \\ \text{with } 0 < d_1 < d_2 < \cdots < d_{w-2} < m. \end{array}$$

The LFSR characteristic polynomial is of degree m with w monomials and with tail monomial d_1. For Galois devices, we will consider d_{w-2} instead of d_1 as the tail monomial. The setup studied will be always specified in the following.

3 Synthesis of LFSRs

The software synthesis of LFSRs is very similar to the hardware synthesis on Xilinx FPGA [1, 5]. For Xilinx FPGA (Spartan 2, Spartan 3, Virtex II) synthesis, 16-bit shift registers (SRL16 and SRLC16) are the basic components. Larger shift registers are obtained by chaining basic components. To compute the feedback, a given set of bits must be accessed. This is particularly difficult with SRL16 or SRLC16 components. This problem is addressed in [1, 5]. In software synthesis, the basic component is the processor register (more precisely a memory block). Chaining those components can be difficult depending on the processor instruction set. Moreover, accessing a single bit within a register is also difficult (1 or 2 instructions). The main difference between software and hardware synthesis is the level of parallelism (Table 1).

We have described the implementation of a single step LFSR. The state register is shifted once per step and one feedback is computed. An attempt to increase the throughput of the device without increasing the device frequency consists in shifting the state register k times and in computing k feedbacks on the same clock cycle. This can be very suitable for software and it leads to leap-forward representation.

Characteristics	Spartan 2	r-bit processor
Basic block	SRL16	register
Chaining	Easy	ISA specific
Bit access	replication	shift and xor
Processing	parallel	sequential

Table 1. Software and hardware synthesis of LFSR

3.1 Leap-forward representations

Leap-forwarding is an appropriate trade-off between the feedback computation and the shift operation for both Fibonacci and Galois LFSR setups in VLSI design. Leap-forwarding exploits the representation of a single step of an LFSR as a vector-matrix multiplication:

$$s^{(t)} = s^{(t-1)}\, A,\ t \geq 0 \tag{9}$$

where $s^{(t-1)}$ is the current state of the LFSR, $s^{(t)}$ is the next state and A is the $m \times m$ companion matrix of the LFSR. Thus, the state of the LFSR after k steps is given by:

$$s^{(t+k)} = s^{(t)}\, A^k,\ t \geq 0. \tag{10}$$

Then the above equation can be transposed to a circuit which is called the k-bit (or element, depending on the field size) leap-forward LFSR representation. For instance, let us consider an 8-bit LFSR in Fibonacci setup with characteristic polynomial $P(X) = X^8 + X^6 + X^5 + X^4 + 1$ (Figure 3).

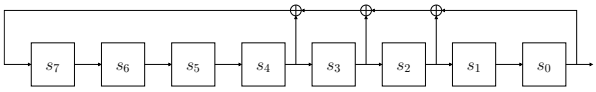


Figure 3. 8-bit Fibonacci LFSR setup

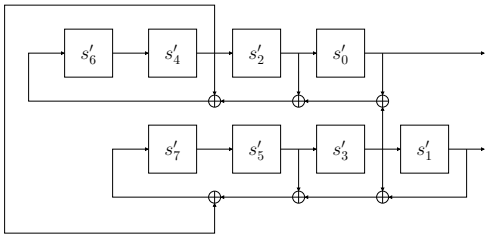


Figure 4. 2-bit leap-forward Fibonacci LFSR setup

The circuit drawn in Figure 3 implements multiplication by matrix A while the circuit drawn in Figure 4 corresponds to the multiplication by A^2. From a hardware point of view, leap-forwarding an LFSR of length m by k elements implies splitting the LFSR into k LFSRs and duplicating the feedback k times. Then the throughput is multiplied by k. An equivalent implementation can be obtained for Galois setup.

In a single step implementation (Figure 3), a given set of state bits is used in the feedback computation. In a 2-bit leap-forward implementation (Figure 4), the same set is used but also the successors of all those bits. On Xilinx FPGAs, it will considerably increase the area cost (more replications) as it is shown in Figure 5.

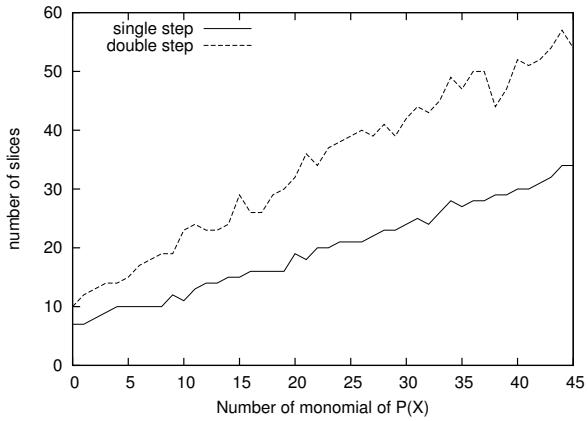


Figure 5. Number of Spartan 2E slices for 128-bit shift registers

In software, the situation is completely different since computing with windows of bits has the same cost as computing with single bit. A k-bit leap-forward implementation equivalently works on k-bit windows instead of one single bit. We have one window per monomial of P. For the 8-bit leap-forward LFSR with polynomial $P(X) = X^8 + X^6 + X^5 + X^4 + 1$, we obtain the implementation given in Code 1 on an 8-bit processor with 4 windows: t_0, t_1, t_2, t_3, t_4.

The previous code (Code 1) is obtained by analyzing the different windows we may have to treat.

3.2 Windows representation

From the LFSR processor register representation, we need to extract all the windows involved in the feedback computation. For efficiency purpose, the maximal size of a window is the processor register width: $k \leq r$. The windows are divided into four families: aligned windows, unaligned windows, overlapping windows and truncated windows.

The *aligned* windows correspond to the LFSR register representation. For those windows there is nothing to do.

Algorithm 1 8-bit leap-forward LFSR. v_0 is the LFSR state (8-bit variable). The variables t_0, t_1, t_2, t_3 and could be removed and all computations could have take place in t_4. We keep them for clarity.

1: $t_0 \leftarrow v_0$	$\{t_0\ (X^8)\}$
2: $t_1 \leftarrow v_0 \gg 2$	$\{t_1$ alignment $(X^6)\}$
3: $t_2 \leftarrow v_0 \gg 3$	$\{t_2$ alignment $(X^5)\}$
4: $t_3 \leftarrow v_0 \gg 4$	$\{t_3$ alignment $(X^4)\}$
5: $t_4 \leftarrow t_0 \oplus t_1 \oplus t_2 \oplus t_3$	$\{$partial $t_4\ (1)\}$
6: $t_4 \leftarrow t_4 \oplus (t_4 \ll 4)$	$\{t_3$ completion$\}$
7: $t_4 \leftarrow t_4 \oplus (t_4 \ll 5)$	$\{t_2$ completion$\}$
8: $t_4 \leftarrow t_4 \oplus (t_4 \ll 6)$	$\{t_1$ completion$\}$

To extract an *unaligned* window an offset is required. One shift is done per *unaligned* window. A window can overlap two processor registers. Such an *overlapping* window is extracted with one xor and two shifts.

The *truncated* windows are the most difficult to extract. Some bits in a window may not be ready at the beginning of the computation. First, all windows are aligned and a partial computation of the feedback is performed, assuming that all the missing bits are zero. This provides the first d_1 feedback bits. Then, the new values resulting from the previous partial computation are propagated to fill all the truncated windows. The completion of the truncated windows highly depends on the tail coefficient d_1 of P for Fibonacci setups. It depends on d_{w-2} for Galois setups.

Let us consider the truncated window associated to d_j. Then, in order to fill the window, the result of the partial computation must be propagated $\lceil \frac{k}{d_j} \rceil - 1 = \lfloor \frac{k-1}{d_j} \rfloor$ times. Each propagation corresponds to one mask (bitwise and), one bitwise xor and one shift.

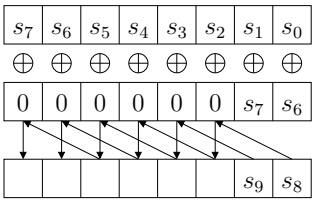


Figure 6. Completion of a truncated window with $d_1 = 2$

Therefore, if $d_j \geq k/2$, we need only one shift and one xor: the situation is similar to the case of overlapping windows. This is the case in Code 1. Then, it costs roughly $k \sum_{d_j < k/2} 1/d_j$ steps to fill all truncated windows.

For $d_1 = 1$, the results of the partial computation are propagated as one bit. We can use the window propagation method but an alternative method without shift can be more efficient. It uses properties of the two's complement of power of a 2. To propagate a bit, we first isolate it with a bitwise and (we get a power of two). We compute its two's complement to expand it. And then we made an additional mask before the bitwise xor. An example is given in Table 2.

Instruction	Result
$y = x \wedge$ 0x01	$0, 0, 0, 0, 0, 0, 0, x_1$
$y = 0 - y$	$x_1, x_1, x_1, x_1, x_1, x_1, x_1, x_1$
$y = y \wedge$ 0x82	$x_1, 0, 0, 0, 0, 0, x_1, 0$
$x = x \oplus y$	$x_8 \oplus x_1, x_7, x_6, x_5, x_4, x_3, x_1 \oplus x_2, x_1$

Table 2. Bit propagation $d_1 = 1$

The cost of bit propagation (Table 3) is the same for one truncated than for k *truncated* windows. The different situations are summed up in Table 3.

Cost per truncated windows				
Case	Shifts	Xors	And	Subtractions
$d_j \geq k/2$	2	1	0	0
$2 \leq d_j < k/2$	$\lfloor \frac{k-1}{d_j} \rfloor$	$\lfloor \frac{k-1}{d_j} \rfloor$	$\lfloor \frac{k-1}{d_j} \rfloor$	0
$d_1 = 1$	0	$k-1$	$2(k-1)$	$k-1$

Table 3. Computation of truncated windows

It clearly appears that characteristic polynomials with $d_1 = 1$ must be avoided since this corresponds to the highest implementation cost.

4 Performance analysis

We have developed a new tool which generates the k-bit leap-forward implementation of an LFSR with a given polynomial. Using this tool, we have explored Fibonacci LFSR implementations on an Opteron M250 (1Ghz) using GCC 4.0 (with -O3) and with different levels of leap-forwarding k. The generic trends on the effect of d_1, w and m are given in Figures (a), (b), (c) and (d) for 32-bit programming. The parameters with the most important effect on the throughput are d_1 and w.

In Figure (a), we studied the effect of d_1 on 128-bit LFSRs with $w = 10$. When $d_1 = 1$ we achieved only a 4.5 speedup factor on the naive implementation with a 32-bit leap-forward implementation. For higher values of d_1, the windows propagation algorithm considerably increases the throughput. For $d_1 = 12$ we already obtain a 24 speedup factor compared to the naive implementation for $k = 32$. We obtain nearly the same performance for higher values of d_1. In practice, we need to avoid to choose a characteristic polynomial with $d_1 = 1$.

(r, d_1, w, m)

Then, we have studied the performance of 128-bit LF-

SRs implementations with a fixed-d_1 (d_1=20) and an arbitrary number of monomials w (Figure (c)). For trinomials ($w = 3$), we achieve impressive performance: a 92.75 speedup factor compared to the naive implementation and we manage to get more than one bit per cycle (for instance 3.75 bits/cycle for $k = 128$). However, the throughput decreases quickly as w is increased. A threshold is observed after $w = 30$. The throughput is not affected by w anymore. This is explained by shift factorization and by an increasing number of common sub-expressions. If we need to compute $x = (x_1 \gg 3) \oplus (x_2 \gg 3)$ it can be simplified by $x = (x_1 \oplus x_2) \gg 3$. We can bound the number of shift operations for unaligned windows to $max(\text{right shift}) = r - 1$. The same result holds for overlapping windows: $max(\text{right shift}) = r - 1$ and $max(\text{left shift}) = r - 1$. This is explicitly done by our generator. The search for common-subexpressions is left to the compiler since we do not make any assumptions on the number of processor registers. It can reduce the number of xors to compute the feedback.

The degree m of P, *i.e.* the LFSR length, only affects the shift computation. Figure (b) shows that peak performance is always reached when k is a multiple of 32 for 32-bit code. In Figure (b), it seems that when $m = 32 \times n$, we get better performance. This impression is confirmed on Figure (c): the peak performance of a polynomial with $m = 32 \times n$ is clearly higher than for other values of m. The impact of n on the throughput is limited. For instance, the best implementation ($k = 128$) for $m = 32$ is only 1.5 times faster than the best implementation ($k = 96$) for $m = 256$.

To produce k-bit leap-forward implementations for $k > r$, our code generator iterates $\lfloor k/r \rfloor$ an r-bit leap-forward feedback. Then, it generates the remaining steps with a $(k \bmod r)$-bit leap-forward feedback. The generated code is always fully unrolled. Thus the study of Figure (c) provides the best unrolling factor. For instance, the best performance for $m = 160$ is reached for $k = 96$, *i.e.* unrolling the feedback loop 3 times.

Figures (e), (f) show two typical pentanomial profiles for throughput and code size with 64-bit programming. The polynomial with $d_1 = 1$ introduces an important code size overhead (Figure (e)). The best performance is achieved by $P(X) = 1 + X^{64} + X^{78} + X^{123} + X^{128}$ (Figure (f)). We already obtain a 100 speedup factor on the naive implementation for $k = 256$. Then the code size is multiply by 5.

Figure (g) shows the effect of the monomials repartition, *i.e.* the choice of d_2 and d_{w-2} on the throughput/code size trade-off. When the monomials are all gather in a small area, we manage to get between 5% and 10% improvement over other repartitions. Figure (h) shows the effect of the number of truncated windows ($d_2 = 2$ and $d_{w-2} = 31$) on the throughput.

5 Conclusions

During our test, we have only considered random characteristic polynomials. For many purposes, mostly cryptography, we need to generate Maximum Length sequences. Then, the characteristic polynomial must be primitive. Thus, we have computed a list of the best primitive trinomials, which can be accessed on [7]. It is worth mentioning that primitive trinomails exist for $m \neq 2n$ only [2]. For instance, we achieve 9.9 bits/cycle for $P(X) = 1 + X^{64} + X^{127}$ with $k = 576$.

For a given set of parameters, a multiple of the characteristic polynomial can also be used to produce the same sequence. Using a multiple of the characteristic polynomial instead of the original polynomial can reduce the number of monomials and can increase the tail coefficient. Let consider the polynomial P defined by:

$$P(X) = X^{128} + X^{32} + X^4 + X^3 + X^2 + X + 1.$$

In Table 4, we compare the parameters of the original polynomial P with some multiple polynomials. The performance of P are low since $d_1 = 1$ (Table 5). The throughput is multiplied by 7 if we use $(1 + X) \times P(X)$ and $(X^6 + X^5 + X + 1) \times P(X)$.

Polynomial	d_1	w	m
$P(X)$	1	7	128
$(1 + X) \times P(X)$	5	6	129
$(X^6 + X^5 + X + 1) \times P(X)$	10	10	134

Table 4. Multiple of the polynomial P

Polynomial	best k	Throughput
$P(X)$	64	0.21
$(1 + X) \times P(X)$	512	1.54
$(X^6 + X^5 + X + 1) \times P(X)$	256	1.52

Table 5. Throughput of multiple of P

We have also considered ANSI C implementation only. An interesting perspective could be to evaluate the influence of all LFSRs parameters for SIMD implementations. SIMD instructions can not speedup the treatment of truncated windows: it sequential by nature. However when $d_1 > k$, SIMD programming is expected to be very efficient. For instance, the overlapping windows can be treated with only one *vperm* instruction (Altivec instruction set). The effect of the monomials repartition is also expected to be different.

References

[1] P. Alfke. Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators. Technical report, Xilinx, 1996. www.xilinx.com/bvdocs/appnotes/xapp052.pdf.

[2] I. F. Blake, S. Gao, and R. J. Lambert. Construction and distribution problems for irreducible trinomials over finite fields. pages 19–32, 1996.

[3] S. Chowdhury and S. Maitra. Efficient Software Implementation of Linear Feedback Shift Registers. In *International Conference in Cryptology in India - INDOCRYPT '01*, Lecture Notes in Computer Science 2247, pages 297–307. Springer Verlag, 2001.

[4] S. Chowdhury and S. Maitra. Efficient Software Implementation of LFSR and Boolean Function and Its Application in Nonlinear Combiner Model. In *Applied Cryptography and Network Security - ACNS 2003*, Lecture Notes in Computer Science 2846, pages 387–402. Springer Verlag, 2003.

[5] M. George and P. Alfke. Linear Feedback Shift Registers in Virtex Devices. Technical report, Xilinx, 2001. www.xilinx.com/bvdocs/appnotes/xapp210.pdf.

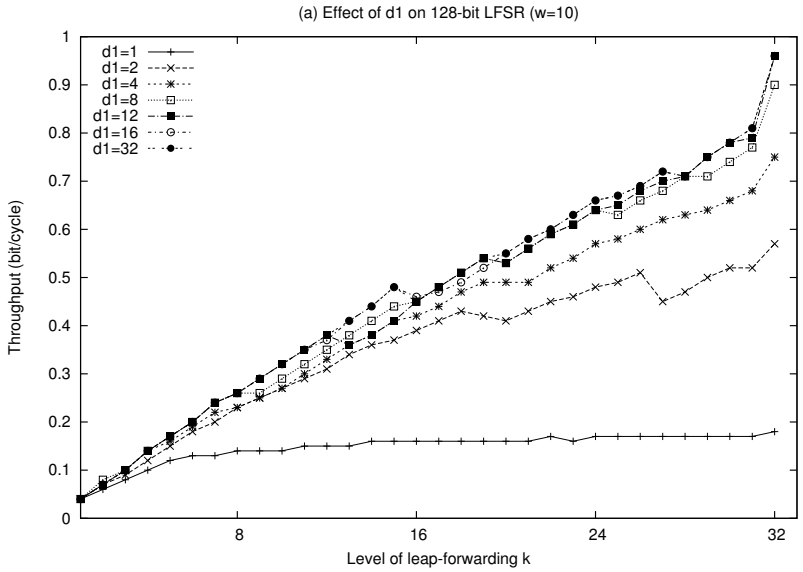
[6] S. W. Golomb. *Shift Register Sequences*. Aegean Park Press, 1981.

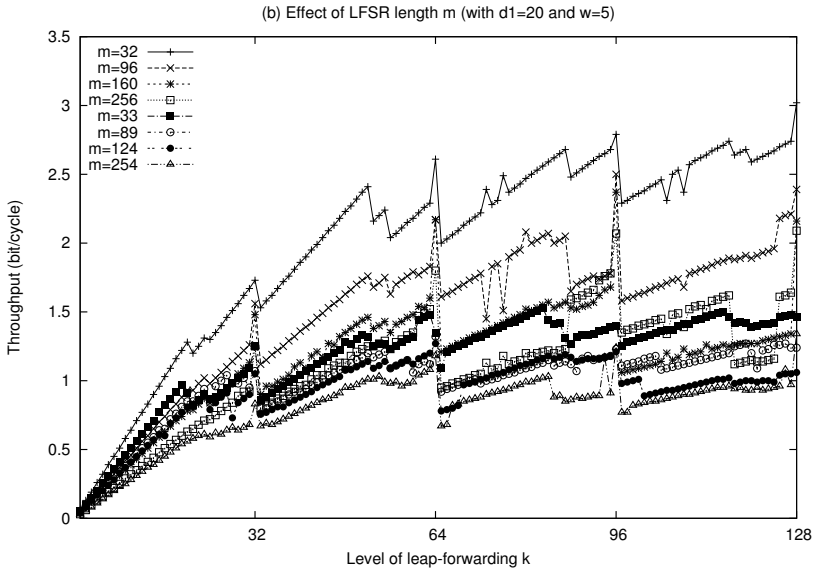
[7] C. Lauradoux. Shift generator, 2006. www-rocq.inria.fr/codes/LFSR/.

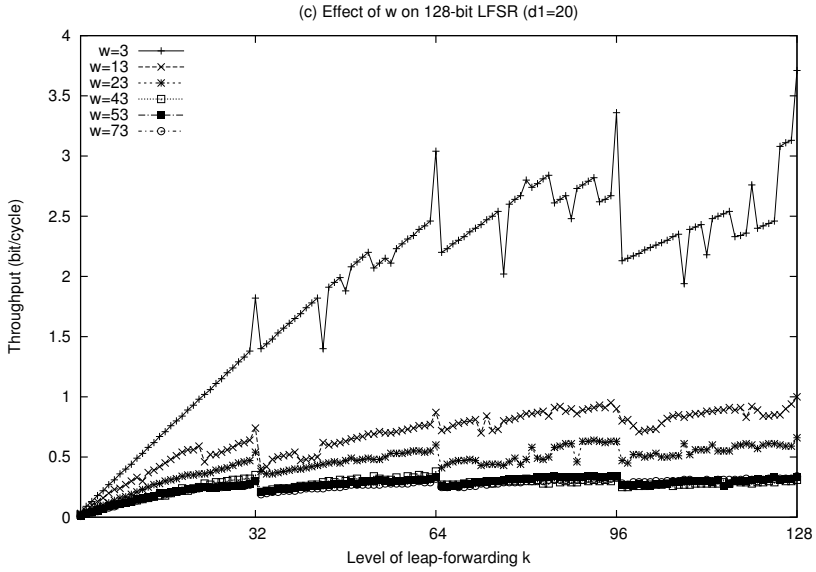
[8] R. Lidl and H. Niederreiter. *Finite Fields*. Cambridge University Press, second edition, 1997.

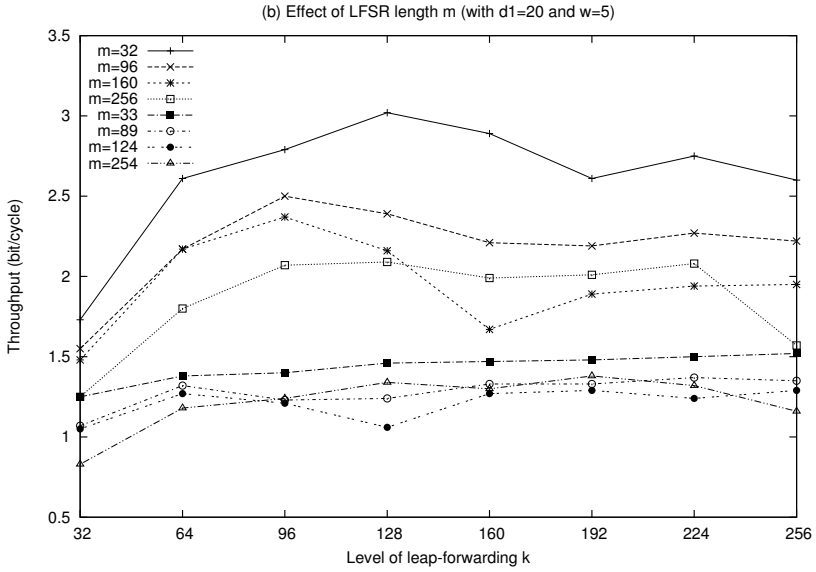
[9] R. J. McEliece. *Finite field for scientists and engineers*. Kluwer Academic Publishers, 1987.

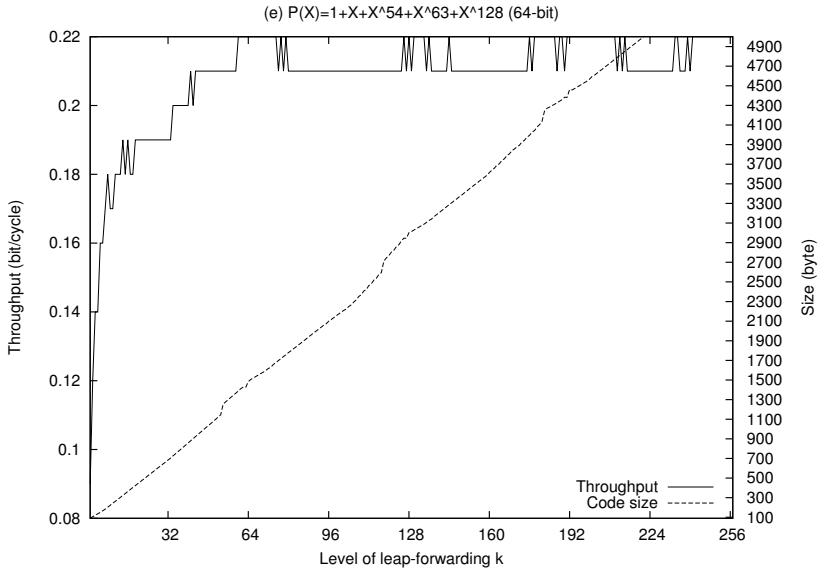
[10] M. Zhang, C. Carroll, and A. H. Chan. The Software-Oriented Stream Cipher SSC2. In *Fast Software Encryption - FSE 2000*, Lecture Notes in Computer Science 1978, pages 31–48. Springer-Verlag, 2001.

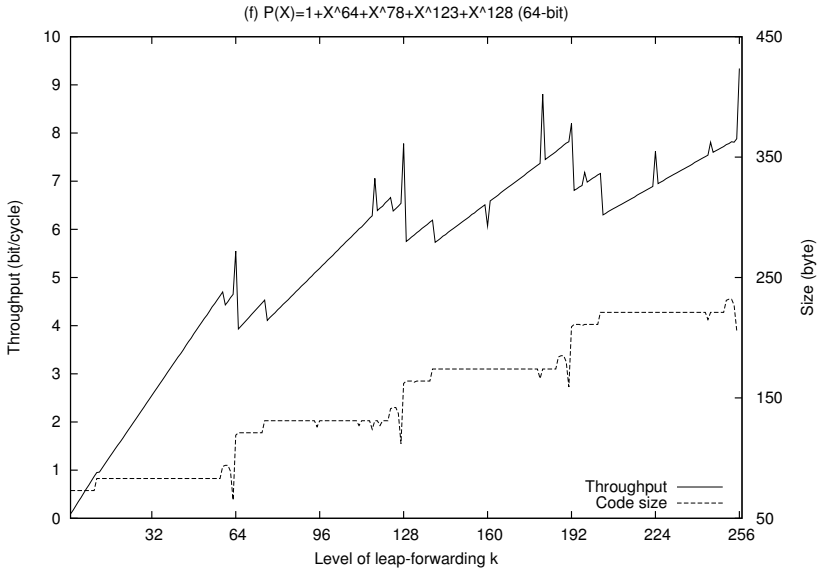

(a) Effect of d1 on 128-bit LFSR (w=10)
Throughput (bit/cycle)
Level of leap-forwarding k
d1=1
d1=2
d1=4
d1=8
d1=12
d1=16
d1=32

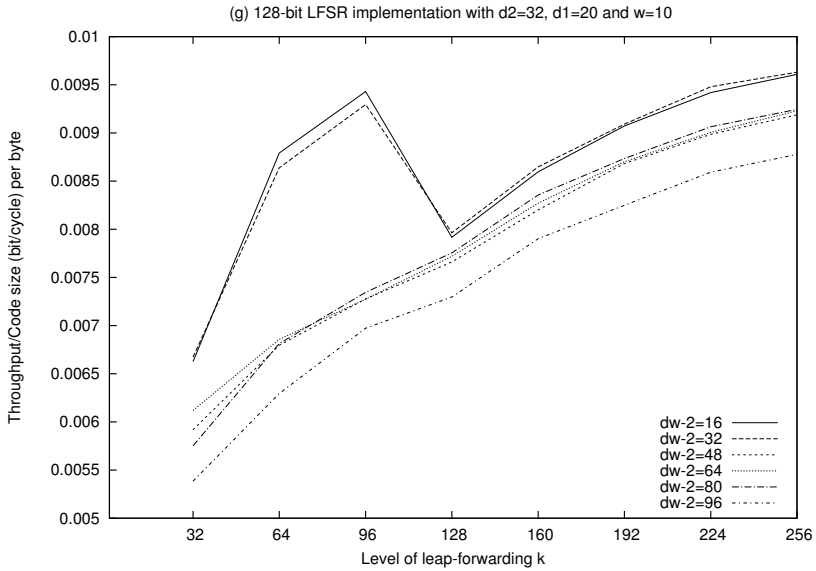

(b) Effect of LFSR length m (with d1=20 and w=5)
Throughput (bit/cycle)
Level of leap-forwarding k
m=32
m=96
m=160
m=256
m=33
m=89
m=124
m=254

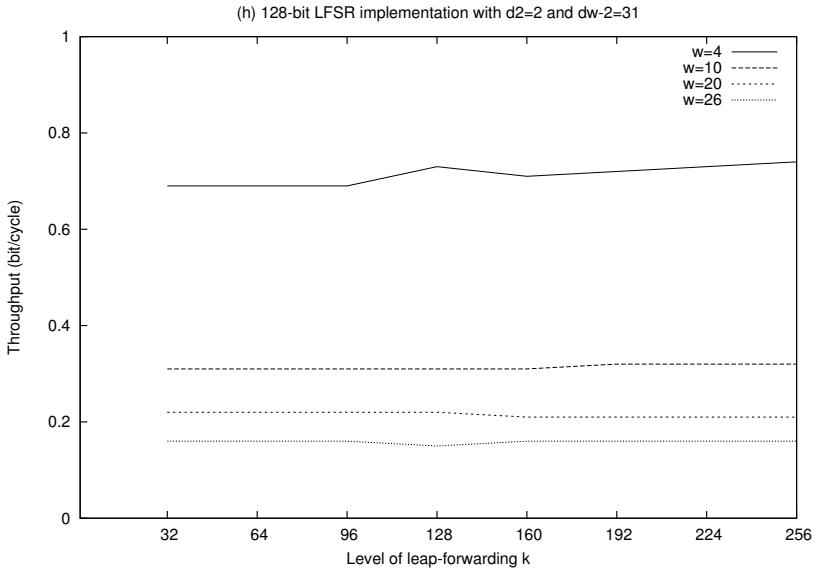

(c) Effect of w on 128-bit LFSR (d1=20)
Throughput (bit/cycle)
Level of leap-forwarding k
w=3
w=13
w=23
w=43
w=53
w=73


(b) Effect of LFSR length m (with d1=20 and w=5)
Throughput (bit/cycle)
Level of leap-forwarding k
m=32
m=96
m=160
m=256
m=33
m=89
m=124
m=254


(e) P(X)=1+X+X^54+X^63+X^128 (64-bit)
Throughput (bit/cycle)
Size (byte)
Level of leap-forwarding k
Throughput
Code size


(f) P(X)=1+X^64+X^78+X^123+X^128 (64-bit)
Throughput (bit/cycle)
Size (byte)
Level of leap-forwarding k
Throughput
Code size


(g) 128-bit LFSR implementation with d2=32, d1=20 and w=10
Throughput/Code size (bit/cycle) per byte
Level of leap-forwarding k
dw-2=16
dw-2=32
dw-2=48
dw-2=64
dw-2=80
dw-2=96


(h) 128-bit LFSR implementation with d2=2 and dw-2=31
Throughput (bit/cycle)
Level of leap-forwarding k
w=4
w=10
w=20
w=26