The comeback of Reed Solomon codes

Nir Drucker
University of Haifa, Israel,
and
Amazon Web Services Inc.[1]

Shay Gueron
University of Haifa, Israel,
and
Amazon Web Services Inc.[1]

Vlad Krasnov
CloudFlare, Inc.
San Francisco, USA

***Abstract*—Distributed storage systems utilize erasure codes to reduce their storage costs while efficiently handling failures. Many of these codes (e. g., Reed-Solomon (RS) codes) rely on Galois Field (GF) arithmetic, which is considered to be fast when the field characteristic is 2. Nevertheless, some developments in the field of erasure codes offer new efficient techniques that require mostly XOR operations, and are thus faster than GF operations.**

Recently, Intel announced [1] that its future architecture (codename "Ice Lake") will introduce new set of instructions called Galois Field New Instruction (GF-NI). These instructions allow software flows to perform vector and matrix multiplications over $GF(2^8)$ on the wide registers that are available on the AVX512 architectures. In this paper, we explain the functionality of these instructions, and demonstrate their usage for some fast computations in $GF(2^8)$. We also use the Intel® Intelligent Storage Acceleration Library (ISA-L) in order to estimate potential future improvement for erasure codes that are based on RS codes. Our results predict ≈ 1.4x speedup for vectorized multiplication, and 1.83x speedup for the actual encoding.

I. Introduction

Distributed storage systems (e. g., Google File System (GFS) [2], Hadoop Distributed File System (HDFS) [3], and Windows Azure [4]) need to employ redundancy in order to guarantee availability and reliability. Traditionally, such systems used the "triple replication" approach of storing three copies of each data blocks for this purpose. While this redundancy mechanism is speed-wise efficient, it is very expensive from the viewpoint of storage costs.

To overcome this challenge, different types of erasure codes were developed, offering a different balance between storage costs and speed. Triple replication is on one side of the equation, and Maximum Distance Separable (MDS) (e. g., RS [5]) codes are on the other. For example, Facebook [6] saved multiple petabytes of storage by using RS codes in their data warehouse. Another example is the Linux Redundant Array of Independent Disks (RAID)-6 file system that uses RS codes and is designed to tolerate any failure of two disks [7], [8].

However, although MDS codes reduce storage requirements significantly, they come at the cost of increased amount of disk operations and network traffic (that typically grows beyond the amount of missing data). Other types of codes (called regenerating codes) have been suggested in order to address this problem: a) an Local Reconstruction Code (LRC) [9] that achieves efficiency through piggybacking RS codes. Since it is no longer an MDS code, it is less storage-wise efficient; b) a Simple Regenerating Code (SRC) [10], which is a random linear code that can achieve the theoretically optimal tradeoff between storage efficiency and traffic bandwidth [11]. Nevertheless, these codes still suffer from excessive I/O consumption, and high computational overheads.

Modern research on regenerating codes is mostly focus on bandwidth and storage, but sacrifices performance considerations (compared to other erasure codes [12]). One slow down element is the extensive use of GF arithmetic [13]. This drives the common belief that "xor" based erasure codes (array codes) are significantly faster than those based on GF arithmetic (e. g., [14]). This was debunked in [15], suggesting that codes based on GF arithmetic can become a viable alternative to array codes if the GF multiplication is cache-limited and highly vectorized. However, subsequent research past [15], tends to favor array codes over GF based codes.

Recently, Intel has announced [1] that its future architecture, microarchitecture codename "Ice Lake", will add new instructions, which we call "GF-NI" to accelerate $GF(2^8)$ multiplication and inversion. These have various usages. In this paper, we show their application to speeding up erasure codes. The paper focuses on Intel's new GF-NI instructions, but for completeness we mention also [16], [17] that discuss a variant of the vectorized GF-NI on other processors. The use of other forthcoming instructions is described in [18], [19].

We first show how to leverage the GF-NI instructions for performing basic arithmetic over $GF(2^8)$ (e. g., vector, matrix or polynomial multiplications). We then explain how storage systems that use RAID-6 or RS codes can benefit from it. Finally, we modified the code of ISA-L in order to predict the potential improvement of these instructions.

The paper is organized as follows. Section II describes the new GF-NI instructions. In Section III we demonstrate some workloads that use arithmetic operations over $GF(2^8)$, and can be sped up by these instructions. Section IV presents RAID systems, and how GF-NI can contribute, and Section V describes the RS code. Finally, in Section VI, we modify the code of ISA-L to use GF-NI, and predict the potential improvement. We conclude in Section VII.

II. Preliminaries

In this paper, the "xor" operation is denoted by $\oplus$, and the "and" operation by $\&$. The function $parity(x)$ returns 1 if x has an odd number of set bits, and 0 otherwise. When using

[1]This work was done prior to joining Amazon.

hexadecimal notation, the LSB is positioned on the right e. g., 0x11b is 000100011011 in binary. The notation $X[j:i]$, $j>i$ refers bits of the sub-array of X between positions i and j (included). The case $i=j$ degenerates to $X[i]$. The assembly snippets are written in AT&T assembly syntax.

A. *Vectorized GF-NI*

Intel's GF-NI instructions include the VGF2P8MULB, VGF2P8AFFINEQB, and VGF2P8AFFINEINVQB instructions (denoted for short by MULB, AFFINEB, and AFFINEINVB, respectively). Alg. 1 describes the MULB instruction. It performs vectorized multiplication in $GF(2^8)$, of $KL = 16/32/64$ 8-bit elements that reside in two $128/256/512$-bit registers (named xmm, ymm, zmm, respectively). The field $GF(2^8)$ is represented in polynomial representation (this defines the field multiplication) with the reduction polynomial $p = x^8 + x^4 + x^3 + x + 1$.

Algorithm 1 MULB instruction

```
    Inputs: SRC1, SRC2 (wide registers)
    Outputs: DST (a wide register)
 1: procedure VGF2P8MULB(SRC1, SRC2)
 2:     for j in 0 to (KL-1) do
 3:         DEST.byte[j] ← GF2P8MULBYTE(SRC1.byte[j], SRC2.byte[j])

 4: procedure GF2P8MULBYTE(s1b, s2b)                    ▷ s1b,s2b (8 bits)
 5:     T[15:0] = 0
 6:     for i in 0 to 7 do
 7:         if s2b[i] then
 8:             T[15:0] = T[15:0] $\oplus (s1b \ll i)$
 9:     for i in 14 downto 8 do
10:         if T[i] then
11:             T[15:0] = T[15:0] $\oplus$ (0x11b $\ll (i-8)$)
12:     return T[7:0]
```

MULB can also be used in cases where $GF(2^8)$ is represented with a different polynomial representations. This requires some linear transformation to/from the representations, and can be performed with the AFFINEB, and AFFINEINVB instructions that are supplemented to GF-NI, and described in Alg. 2. Here, an affine transformation is defined by $A \cdot x + b$ or $A \cdot inv(x) + b$, accordingly; A is an 8×8-bit matrix vectorized $KL = 2/4/8$ times; x and b are 8-bit vectors. In the AFFINEB instruction, the value of b is a constant prescribed in the immediate byte. The inverse of x is defined in $GF(2^8)[x] \Big/ p$ (a lookup table is given in [1]).

Algorithm 2 AFFINEB and AFFINEINVB instructions

```
    Inputs: S1, S2 (wide registers) imm8 (8 bits)
    Outputs: D (a wide register)
 1: procedure VGF2P8AFFINE[INV]QB(S1, S2)
 2:     for $j$ in 0 to $KL-1$ do
 3:         for $b$ in 0 to 7 do
 4:             $k = 64j$, $q = k + 8b$
 5:             D[$q+7:q$] = [Inv]AffB(S2[$k+63:k$], S1[$q+7:q$], imm8)
 6:     return D[$64KL-1:0$]

 7: procedure [INV]AFFB(s2, s1, imm8)
 8:     for i = 0 to 7 do
 9:         T[7-i] = parity(s2[8i+7 : 8i] & [inv](s1)) $\oplus$ imm8[i]
10:     return T[7:0]
```

III. Basic arithmetic over $GF(2^8)$

We show the use of GF-NI for $GF(2^8)$ basic arithmetic.

A. *Multiplying by* 2

Let $r, a \in GF(2^8)$, where $a = \sum_{i=0}^{7} a_i x^i$, with coefficients in $GF(2)$. The coefficients of $r =$ "2"a (i. e., the element 00000010) are given by:

$$r_7 = a_6, r_6 = a_5, r_5 = a_4, r_4 = (a_3 \oplus a_7),$$
$$r_3 = (a_2 \oplus a_7), r_2 = (a_1 \oplus a_7), r_1 = a_0, r_0 = a_7$$

Fig. 1 compares the existing "legacy" (panel a) and the GF-NI (panel b) vectorized implementations. In the legacy implementation, the Most Significant Bit (MSB) (a_7) of each value is stored in a mask register $\%k1$ (Step 6), and cleared (Step 8). Then, the vector is left-shifted, and xored (Step 7,9); this puts the final results in t (if $a_7 = 0$) or in $zmm1$ (if $a_7 = 1$). The final results are blended into t, according to the original value of a_7 (Step 10). In comparison, the GF-NI implementation requires only one AFFINEB instruction.

(a) Legacy

```
1  .set t, %zmm0
2  .mask1:
3   .qword 0xfefefefefefefefe
4  .mask2:
5   .qword 0x1d1d1d1d1d1d1d1d
6  vpmovb2m t,%k1
7  vpsllq $1,t,t
8  vpandq .mask1(%rip),t,t
9  vpxorq .mask2(%rip),t,%zmm1
10 vpblendmb %zmm1,t,t{%k1}
```

(b) GF-NI

```
1  .set t, %zmm0
2  .mtrx:
3   .byte 0x40,0x20,0x10,0x88,
4   .byte 0x84,0x82,0x01,0x80
5  vgf2p8affineqb $0x00,.mtrx(%rip),t,t
```

Fig. 1. Using affine transformation for multiplying a vector by 2.

B. *Matrix multiplication*

a) Multiplying two 8×8-bit matrices: Calculating the product $C = AB$, of two 8×8-bit matrices (A, B) can be implemented easily with the AFFINEB. The two matrices are first loaded into two zmm registers (where each byte represents a column of the matrix). If AFFINEB is run over these inputs directly, we get the columns of A multiplied by with the columns of B, so we need to transpose A in order to get the desired result. To this end, we use AFFINEB with A and I (the identity matrix encoded as 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01) as the first and second operands, respectively.

b) Multiplying two 64×64-bit matrices: For calculating the product $C = AB$ of two 64×64-bit matrices (A, B) we divide each matrix to 8×8 sub-matrices, each one is an 8×8-bit matrix. Let $X_{i,j}$ denote the sub-matrix of X in row i and column j. Then,

$$C_{i,j} = \bigoplus_{k=0}^{7} \left(A_{i,k} B_{k,j}\right)$$

Calculating each of the sub-matrices of C requires only 8×2 AFFINEB invocations (assuming a transpose is required). This costs 512×2 AFFINEB invocations for the two matrices product. We point out that the columns of the sub-matrix $A_{i,j}$ are stored in bytes $j+i+8k$, $k=0,\ldots,7$ in memory. Collecting them can be done with the VPGATHERQQ instruction. Moreover, using an "AFFINEB friendly" memory arrangement, can further accelerate the loading/storing operations.

C. Polynomial multiplication

Let C, A, B, P be polynomials with coefficients in $GF(2^8)[x]$. We use the MULB instruction to compute $C = A\cdot B \pmod{P}$. Fig. 2 illustrates the code flow for multiplying polynomial of degree 7. The inputs A, B, P are loaded into the lower quadword (64 bits) of xmm8, xmm9, and xmm10, respectively. The upper quadwords of these registers are padded with zeros.

The illustrated code executes the Schoolbook multiplication (Steps 9-14), placing the resulting product in xmm11. The reduction (Steps 16-25) is performed according to the iterative equation $C = C \oplus \left(C_{i+8}\cdot P\cdot x^i\right)$, $i = 6,\ldots,0$. In the end, xmm11 holds the result. We note that using this code with larger registers (ymm or zmm) allows to multiply polynomials of higher degrees (15 and 31, respectively) in a similar way. Alternatively, these register can vectorized the multiplication of two/four 7-degree polynomials in parallel.

```
 1 .irp i, 0,1,2,3,4,5,6,7
 2   .mask\i:
 3   .byte 0x0\i, 0x0\i, ..., 0x0\i
 4 .endr
 5 .irp i, 0,1,2,3,4,5,6,7
 6   vmovdqa .mask\i(%rip), %xmm\i
 7 .endr
 8
 9 .irp i, 0,1,2,3,4,5,6,7 #Multiplication
10   vpshufb     %xmm\i,  %xmm8,  %xmm12
11   vgf2p8mulb  %xmm9,   %xmm12, %xmm13
12   vpxor       %xmm13,  %xmm11, %xmm11
13   vpslldq     $1,      %xmm9,  %xmm9
14 .endr
15
16 .irp i, 0,1,2,3,4,5,6 #Reduction
17   vpaddq .mask7(%rip), %xmm\i, %xmm\i
18 .endr
19
20 .irp i,6,5,4,3,2,1,0
21   vpshufb    %xmm\i,  %xmm11, %xmm12 #c[8+i],c[8+i],...
22   vgf2p8mulb %xmm12,  %xmm10, %xmm13 #c[8+i]*P
23   vpslldq    $\i,     %xmm13, %xmm13 #c[8+i]*P*x^i
24   vpxorq     %xmm13,  %xmm11, %xmm11
25 .endr
```

Fig. 2. Multiplication of polynomials of degree 7 (coefficients in $GF(2^8)[x]$.)

IV. RAID-6

RAID is a virtualized storage system based on multiple physical hard drives (d_i, $i = 1,\ldots,k$) each has multiple stripes ($d_{i,j}$, $j = 1,\ldots,l$). It uses striping, mirroring, and additional checksum disk drives, in order to improve performance, while maintaining high data redundancy. For example, RAID-4 and RAID-5 use the same simple erasure code (a parity MDS code), but have a different stripes layout. RAID-4 uses fixed disk identities and an additional disk P, dedicated for checksum data. The stripes of P are defined by ($p_i = d_{1,i} \oplus d_{2,i} \oplus \ldots \oplus d_{k,i}$). In contrast, RAID-5 uses rotational identities on a stripe-by-stripe basis (each disk holds the same amount of data and coding). This offers better balancing of the system.

RAID-6 systems are a bit more complicated. They use a similar parity disk P as in RAID-4/5, and an additional disk Q. The data stored on Q is based on $GF(2^8)$ arithmetic, setting $q_i = d_{1,i} \oplus 2^0 d_{2,i} \oplus \cdots \oplus 2^{k-1} d_{k,i}$. This allows the system to tolerate failures of any two disks. Common RAID-6 implementations use a rearranged version of this equation: $q_i = 2(2(\ldots 2(2d_k \oplus d_{k1})\ldots)\oplus d_2)\oplus d_1$, which requires only XORs and multiplication by 2. The latter can be accelerated with the method described in Section III-A.

V. Accelerating Reed Solomon Codes

RS codes are MDS codes that are able to correct multiple errors (e. g., in RAID systems). An RS(n,k) code uses n-symbol codewords with k data symbols, and $r = n-k$ checksum symbols ($k < n$ and can be defined by design). Common RS(n,k) codes operate over $GF(2^w)$, $w \in \{4, 8, 16, 32, 64, 128\}$, and $n < 2^w$, where symbols are bit strings. For example, an RS code that operates on 8-bit symbols, has codewords of length $n < 2^8 - 1 = 255$ symbols. Fixing $k = 223$ data symbols leaves room for up to 32 checksum symbols. This code can correct up to $32/2 = 16$ symbol errors per codeword.

We briefly describe RS(n,k) encoding (see details in [20], [21]). Denote each data disk by d_i, $i = 1,\ldots,k$ (for simplicity, assume that each disk contains one strip). We denote checksum disks by c_j $j = 1,\ldots,r$ (c_1=P, c_2=Q is used in Section IV). The encoding function is based on r functions (f_i) that are defined as a linear combination of the data symbols $c_i = f_i(d_1,\ldots,d_k) = \sum d_j f_{i,j}$. Therefore, the encoding is given as a matrix $A = (I; F)$ where $A\cdot d = (d; c)$ and I is the identity matrix. $A = (I; F)$ needs to be chosen in a way that submatrix of A, with k rows, is invertible. Typically, a common choice is the Vandermonde matrix where $f_{i,j} = j^{i-1}$

$$A = \begin{pmatrix} & & I_{k\times k} & & \\ f_{1,1} & f_{1,2} & f_{1,3} & \ldots & f_{1,k} \\ f_{2,1} & f_{2,2} & f_{2,3} & \ldots & f_{2,k} \\ f_{3,1} & f_{3,2} & f_{3,3} & \ldots & f_{3,k} \\ . & . & \ldots & . & . \\ f_{r,1} & f_{r,2} & f_{r,3} & \ldots & f_{r,k} \end{pmatrix} = \begin{pmatrix} & & I_{k\times k} & & \\ 1 & 1 & 1 & \ldots & 1 \\ 1 & 2 & 3 & \ldots & k \\ 1 & 2^2 & 3^2 & \ldots & k^2 \\ . & . & . & \ldots & . \\ 1 & 2^{r-1} & 3^{r-1} & \ldots & k^{r-1} \end{pmatrix}$$

When data in one of the data disks (d_j) is modified (to d_j'), the checksum disks can be modified by $c_i = f_{i,j}(d_j' - d_j)$.

VI. GF-NI usage for erasure codes

We predict the potential improvement that GF-NI can contribute in future platforms, before real processors are available. To this end, we used the ISA-L [22] that provides an implementation of erasure codes, based on GF arithmetic. ISA-L performs fast GF multiplication of $a, b \in GF(2^w)$, using a technique that is similar to [15]. First, calculate all the possible products of a by any element in $GF(2^w)$ (there are 2^w options), and place them in a table. Subsequently, choose the relevant value from the table, according to the value of b. This method becomes more efficient when it is applied for multiplying a by a vector of values B, because the table need to be calculated only once.

For sufficiently small w, the full table can be embedded in some wide registers. For example, for $w = 4$, the table contains 16 symbols of 16-bit. These can be embedded in one ymm or register two xmm registers. We are interested in the case $w = 8$. Here, the table is larger, and holds $2^8 = 256$ symbols. Reducing the size of this table [15] can be achieved by observing that $b = (b_h \ll 4) \oplus b_l$ and $ab = (a(b_h \ll 4)) \oplus a(b_l)$. Thus, each byte (degenerated to have 4 zero bits) can be calculated separately, by using only 16 symbols as in the case for w=4 (a similar method was also proposed in [23]).

The ISA-L code consists of three main primitives that operate over $GF(2^8)$: a) `gf_vect_mul` - performs vectorized multiplication; b) `gf_vect_dot_prod_perf` - performs vectorized multiplication, while accumulating the results (of each one) into a single value; c) `gf_vect_mad`- performs vectorized multiplication while adding the results of a previously vector. The `gf_vect_dot_prod_perf` method is used to encode the data, as explained in Section V. The `gf_vect_mad` function is used for updating an existing codeword, after some change to the data has occurred.

To evaluate the performance of these functions, and of the encoding and encoding update functions, we used the performance tests built-in with the ISA-L package. We compared AVX2 implementations, because the ISA-L AVX512 code based on lookup tables, was slower than its AVX2 counterpart. The results are given in Table I. To overcome the fact that the GF-NI are not yet available in real silicon, we use "stand in" replacements to estimate the predicted performance. Here, we replaced the assembly code that handles the tables (as above) with the single instruction `VPMULDQ` instruction. We believe it would have the same latency and throughput as GF-NI.

The experiments were carried out on a platform with the 7th Generation Intel® Core^{TM} processor ("Kaby Lake") - $i7 - 7700$ CPU at 3.60 GHz. The platform has 32K L1d and L1i cache, 256K L2 cache, and $8,192$K L3 cache. It was configured to disable the Intel® Turbo Boost Technology. The runs were carried out on a Linux (Ubuntu 16.04.3 LTS) OS.

TABLE I
SPEED UP PREDICTIONS FROM USING GF-NI IN ISA-L. PERFORMANCE NUMBERS ARE REPORTED IN MILLISECONDS (LOWER IS BETTER).

Function	Reference impl.	Stand-in impl.	Speedup
`gf_vect_mul`	819	624	1.31
`gf_vect_dot_prod_perf`	223	159	1.4
`gf_vect_mad`	771	700	1.1
`Encode`	2,686	1,462	1.834
`Encode update`	3,135	2,800	1.11

VII. CONCLUSION

This paper shows Intel's new forthcoming vectorized GF-NI, and demonstrates several usages that can enjoy their faster $GF(2^8)$ arithmetic offering. We used the erasure code implementation of ISA-L, to provide estimations for future performance improvements. Our results predict a speedup of 1.4x over the current ISA-L implementation. Finally, we note that erasure codes based on $GF(2^8)$ constructions have been replaced by array codes, due to performance considerations. We hope that the coming performance improvements contributed by GF-NI, could reopen the door for deploying $GF(2^8)$ based erasure codes.

ACKNOWLEDGEMENTS

This research was supported by: The Israel Science Foundation (grant No. 1018/16); The Ministry of Science and Technology, Israel, and the Department of Science and Technology, Government of India; The BIU Center for Research in Applied Cryptography and Cyber Security, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office; The Center for Cyber Law and Policy at the University of Haifa.

REFERENCES

[1] —, "Intel architecture instruction set extensions programming reference," https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, October 2017.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003. [Online]. Available: http://doi.acm.org.ezproxy.haifa.ac.il/10.1145/1165389.945450

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1–10.

[4] B. Calder, J. Wang, and et al. , "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 143–157. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571

[5] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: https://doi.org/10.1137/0108018

[6] —, "Hdfs raid," http://www.slideshare.net/ydn/hdfs-raid-facebook, December 2010.

[7] H. P. Anvin, "The mathematics of raid-6," https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf, December 2011.

[8] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Berkeley, CA, USA, 2004, pp. 1–14.

[9] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 15–26. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[10] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple regenerating codes: Network coding for cloud storage," in *2012 Proceedings IEEE INFOCOM*, March 2012, pp. 2801–2805.

[11] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, Sept 2010.

[12] H. C. H. Chen, Y. Hu, P. P. C. Lee, and Y. Tang, "Nccloud: A network-coding-based storage system in a cloud-of-clouds," *IEEE Transactions on Computers*, vol. 63, no. 1, pp. 31–44, Jan 2014.

[13] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proccedings of the 7th Conference on File and Storage Technologies*, ser. FAST '09. Berkeley, CA, USA: USENIX Association, 2009, pp. 253–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=1525908.1525927

[14] L. Xu and J. Bruck, "X-code: Mds array codes with optimal encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, Jan 1999.

[15] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel SIMD instructions," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, 2013, pp. 298–306. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/plank_james_simd

[16] A. Kumar and K. van Berkel, "Vectorization of reed solomon decoding and mapping on the evp," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 450–455.

[17] S. Mamidi, M. J. Schulte, D. Iancu, A. Iancu, and J. Glossner, "Instruction set extensions for reed-solomon encoding and decoding," in *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, July 2005, pp. 364–369.

[18] N. Drucker, S. Gueron, and V. Krasnov, "Fast multiplication of binary polynomials with the forthcoming vectorized VPCLMULQDQ instruction," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018.

[19] ——, "Making aes great again: the forthcoming vectorized aes instruction," Cryptology ePrint Archive, Report 2018/392, 2018, https://eprint.iacr.org/2018/392.

[20] J. S. Plank *et al.*, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Softw., Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[21] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on reedsolomon coding," *Software: Practice and Experience*, vol. 35, no. 2, pp. 189–194, 2005. [Online]. Available: http://dx.doi.org/10.1002/spe.631

[22] −, "Intel Intelligent Storage Acceleration Library (Intel ISA-L)," https://software.intel.com/en-us/storage/ISA-L, Feb 2018.

[23] S. Gueron and M. Kounavis, "Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm," *Information Processing Letters*, vol. 110, no. 14, pp. 549 – 553, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002001901000092X