Optimization of RAID Erasure Coding Algorithms for Intel Xeon Phi

Aleksei Marov, *Researcher,* R&D department
Raidix LLC,
PhD student, Saint-Petersburg State University
Saint Petersburg, Russia
marov.a@raidixstorage.com

Andrey Fedorov, *Founder and Board Member*,
Raidix LLC,
Saint Petersburg, Russia
andrew@digdes.com

***Abstract*—In this work we describe and consider some features of implementing RAID erasure coding algorithms for Intel Xeon Phi coprocessor. We propose some algorithmic and technical improvements of encoding and decoding performance both in native and offload modes. Proposed approaches are designed to maximize the efficiency of Intel MIC architecture. We suggest new approach to Galois fields arithmetic vectorization which allows to achieve high encoding and decoding speed.**

Keywords—erasure coding; Galois fields; MIC architecture; vectorization;

I. Introduction

In modern storage systems as well as in many file systems fault tolerance is achieved by using replication or erasure coding. Although erasure coding has a much lower storage overhead, it has some computational cost, which depends on the number and length of encoded or decoded blocks. Many optimizations have been suggested in erasure coding procedures in recent years. Some results allow one to use different matrices to reduce the number of operations during coding [1]. The others to implement some internal coding procedures such as Galois fields arithmetic using SIMD instructions [2, 3]. These results are used in very efficient ECC libraries e.g. Jerasure [4], ISA-L [5] and others. However, at the same time storage systems execute many other computationally expensive jobs like data compression, deduplication, encryption and decryption, virtualization, that require a lot of CPU time. Due to the fact that data encoding is performed for every write and read (in degraded mode) request, the opportunity of moving calculations to coprocessors looks very attractive.

From this point of view, Intel Xeon Phi is a good platform to apply ECC, unlike other coprocessors it does not require learning a new programming language or parallelization paradigm. The programming languages for this platform are C/C++ or Fortran and OpenMP/OpenCL/MPI standards of parallelization, which are very popular and well known for software developers. The main challenges in applying algorithms for this coprocessor are correct usage of SIMD registers of many cores, a lot of threads and cores to utilize, threads locks and data transfers (in offload mode). In the present paper we will describe this platform, make a brief introduction to RAID coding algorithms, Galois fields arithmetic and show some ways to optimize them for MIC.

II. MIC Architecture

We start by recalling basic technical aspects of MIC. Intel Many Integrated Core Architecture (MIC) combines a lot of Intel x86 based processors onto a single chip. MIC architecture utilizes a high degree of parallelism in smaller, lower-power performance Intel processor cores.

In our tests we used Intel Xeon Phi Coprocessor 3120P. It has 6 GB RAM, 57 cores 1.1 GHz each and supports 512-bit SIMD instructions. The cores can run 4 hardware threads for a total up to 228 threads in SMP model. Each core contains 32 KB L1 I/D cache and 512 KB L2 cache. MIC card communicates with a host system via PCIe and runs Linux-based OS to control and manage all communications, memory and threads.

Programming models for Intel Xeon Phi are Offload mode and Native mode. In the offload mode, the main program is executed at the host system while hard computational parts are offloaded to the coprocessor. On completion the computing, the results are transferred to the host system. In the native mode, the whole program is executed in Xeon Phi. In Section VII we will show coding algorithms performance in both modes.

III. Galois Fields Arithmetic in RAID

In storage systems implemented as RAID all drives are divided into blocks of equal sizes. Sequence of blocks with the same positions in different drives composes *stripe*. All operations such as checksum calculation or data recovery are performed for each stripe. Fig. 1 represents stripe with n data blocks and 2 checksum blocks of length w. This case corresponds to RAID-6 stripe. In our work we mostly consider RAID arrays with 2 or 3 checksums because of their popularity in modern storage systems. These technologies allow recovering up to three failed drives, detecting and recovering Silent Data Corruption (SDC) also known as erasures, and have low storage overhead.

In order to perform calculations with blocks in stripe, each data block or checksum block is represented as a set of q-bit codewords. Considering codewords as elements of Galois field $GF(2^q)$ allows one to define such operations as multiplication, addition, inversion and others. A Galois field [6] of characteristic 2 is a field of 2^q elements $GF(2^q) = \mathbb{Z}_2[x]/\langle f(x)\rangle$ where $f(x)$ is a field generating polynomial, i.e. an irreducible over $\mathbb{Z}_2$ polynomial of degree q. An element of this field can be treated either as a q-bit codeword or as a polynomial

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

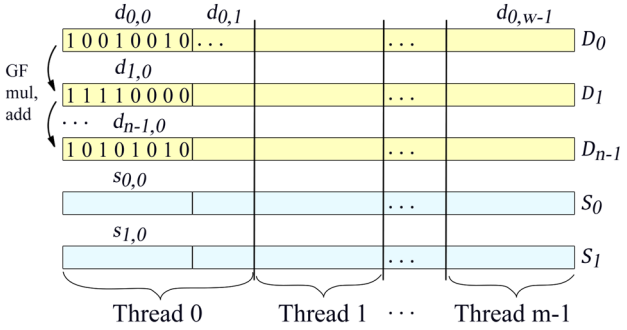

Fig. 1. Stripe representation with n data blocks and 2 checksum blocks. Each thread calculate its own set of codewords.

in the variable x of degree less than q with the coefficients vector coinciding with the codeword. For instance, the codeword (10000101) and the polynomial $x^7 + x^2 + 1$ are the equivalent representations of the same element of $GF(2^8)$. For elements $a(x)$ and $b(x)$ of the field, the addition operation is defined as the sum $a(x) + b(x)$ with reduction coefficients modulo 2 (XOR). The multiplication operation $a(x) * b(x)$ is defined as the product of polynomials modulo $f(x)$ with coefficients reduction modulo 2. A quick introduction to Galois fields in RAID is represented in [3]. In RAID-6 the following formulas can be used for checksum calculation:

$$\begin{aligned} &S_0 = D_0 + D_1 + \cdots + D_{n-1}, \\ &S_1 = D_0 a^{n-1} + D_1 a^{n-2} + \cdots + D_{n-2} a + D_{n-1} \\ &= (((D_0 a + D_1)a + D_2)a + \cdots + D_{n-2})a + D_{n-1}. \end{aligned} \quad (1)$$

The addition and multiplication operations in the right-hand sides of the formulas (1) are performed word-wise, i.e. if

$$D_0 = (d_{0,0}, d_{0,1}, d_{0,2}, \ldots),\ D_1 = (d_{1,0}, d_{1,1}, d_{1,2}, \ldots),$$

then

$$D_0 + D_1 = (d_{0,0} + d_{1,0}, d_{0,1} + d_{1,1}, \ldots), \ldots,$$

$$D_0 a^{n-1} = (d_{0,0} a^{n-1}, d_{0,1} a^{n-1}, \ldots), \ldots$$

with the word operations defined in an appropriate Galois field $GF(2^q)$. The third checksum can be calculated by the formula:

$$\begin{aligned} &S_2 = D_0 a^{2(n-1)} + D_1 a^{2(n-2)} + \cdots + D_{n-2} a^2 + D_{n-1} \\ &= (((D_0 a^2 + D_1)a^2 + D_2)a^2 + \cdots + D_{n-2})a^2 + D_{n-1}. \end{aligned} \quad (2)$$

In RAID with three checksums up to three failed drives can be recovered.

According to Horner scheme in (1) and (2), main operations in checksum calculation are codewords addition and multiplication by a (primitive element of a field). Next, we will briefly describe how checksums S_0, S_1 help to recover failed drives in RAID-6.

IV. Data Recovery in RAID

Suppose two checksums S_0, S_1 were calculated for some data stripe and the failure of the drives occurred. Suppose the numbers of failed drives are j, k and failed blocks are D_j, D_k. Then the recovery of the failed blocks can be performed by the following steps:

1) *Set failed blocks to zero* $D_j = 0,\ D_k = 0$
2) *Calculate new values of checksums* $\overline{S_0}, \overline{S_1}$ *by (1).*
3) *Use this values to recover failed blocks*

$$\begin{aligned} &D_j = \left((S_1 + \overline{S_1})a^{-(n-k-1)} + S_0 + \overline{S_0}\right)(a^{k-j} + 1)^{-1} \\ &D_k = S_0 + \overline{S_0} + D_j \end{aligned} \quad (3)$$

These steps are based on Galois field multiplication operation, which we intend to optimize. For any polynomials $a(x), b(x)$ we have

$$\begin{aligned} a(x)b(x) &= (a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0) \\ &* (b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1 x + b_0) \\ &= (\ldots((b_{n-1}a(x)x + b_{n-2}a(x))x + b_{n-3}a(x))x + \cdots \\ &+ b_1 a(x))x + b_0 a(x). \end{aligned} \quad (4)$$

From (4) it follows that multiplication of two Galois fields elements can be performed by multiplications by x (commonly used as a primitive element) and additions in this field. Therefore, all operations in the recovery process can be reduced to multiplications by x and additions.

From (1), (3) and (4) it follows that multiplication by x is a crucial point in checksums calculation and data recovery. Hence, optimization of this multiplication may have significant influence on the entire coding process.

V. Optimization of basic GF Operations

In this section, we discuss some ways to perform multiplication by primitive element efficiently. We will consider all further examples in Galois field $GF(2^8)$ generated by irreducible polynomial $f = x^8 + x^4 + x^3 + x^2 + 1 = (100011101)$. We choose x as primitive element to simplify computation. Since $f(x)$ is a sparse polynomial, i.e. it has a few nonzero coefficient, this field is quite convenient for fast computations. At the same time, it allows one to encode up to 255 drives in one array, which is quite enough for practical applications. Nevertheless, suggested ideas can be generalized for any other fields.

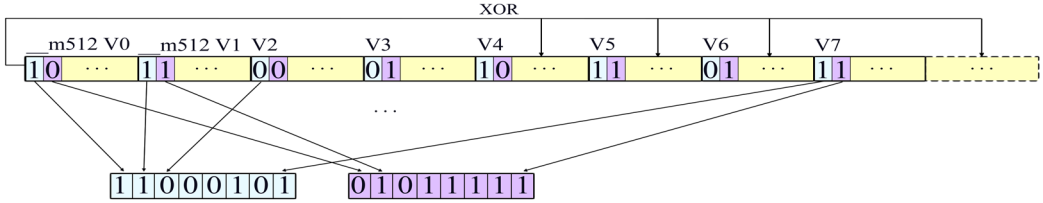

Fig. 3. Multiplication by x of the polynomial $F(x) = V_0 x^7 + V_1 x^6 + \cdots + V_6 x + V_7, F \in \mathbb{I},\ V_i \in \{0,1\}_{512}$.

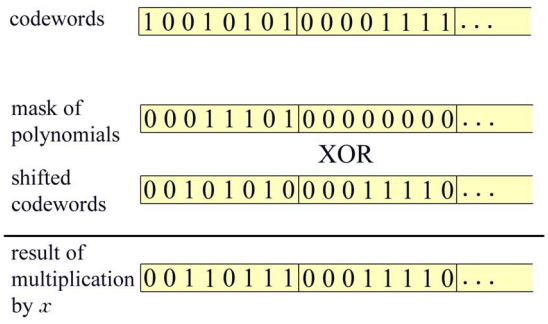

Fig. 2. Multiplication by x in $GF(2^8)$ using vector operations

Next, we will describe the well-known method of vectorization of GF multiplication [3]. All bytes of vector register are considered as elements of the same field. To perform multiplication by x one should:

a) Calculate mask of genereating polynomial. This mask consists of the values of generating polynomial on the positions corresponding to bytes of an initial vector having leading bit set to 1.

b) Shift packed bytes in initial vector left by 1.

c) XOR shifted vector with generated mask.

The resulting vector contains initial values multiplied by x in the Galois field. On Fig. 2 one can see the example of these operations. This approach efficiently utilizes 512-bit registers of Intel Xeon Phi, and requires small number of operations for the whole multiplication process. We will compare performance of this method with our own approach in foregoing sections.

An alternative method to perform Galois fields arithmetic assumes using SIMD shuffle instructions [2]. Coding libraries such as Jereasure and ISA-L are based on this idea. However, current generation of Intel Xeon Phi instruction set does not support these operations, so this approach is not yet applicable. Nevertheless, this method looks very promising for future generations of Xeon Phi coprocessors.

We will now describe our suggested method of multiplication in Galois fields. This method is based on simple XOR operations with vector registers and allows performing multiplication of large numbers of field elements by few processor operations.

The main idea is as following. We consider data blocks $d_{i,j}$ as a polynomials $F(x) = V_0x^k + V_1x^k + \cdots + V_{k-1}x + V_k$, such that $V_i \in \{0,1\}_q$, i.e. vectors of length q consisting of 0 and 1. These polynomials with vector coefficients form an ideal $\mathbb{I}$ of polynomial ring $\{0,1\}_q[x]$. Multiplication by x for these polynomials can be easily performed by a few XOR operations. We will demonstrate it by example, when $k = 512, q = 8$, according to Intel Xeon Phi vector register size. Let the generating polynomial of this ideal be

$$f(x) = Ex^8 + Ex^4 + Ex^3 + Ex^2 + E,$$

where E is q-bit vector equal to (1…11), and $F(x) = V_0x^7 + V_1x^6 + \cdots + V_6x + V_7$ is polynomial in some data block. Then after multiplication F by x modulo f we get

$$xF(x)\ mod\ f = V_1x^7 + V_2x^6 + V_3x^5 + (V_4 + V_0)x^4 + (V_5 + V_0)x^3 + (V_6 + V_0)x^2 + V_7x + V_0,$$

so coefficients will be $(V_1, V_2, V_3, V_4 + V_0, V_5 + V_0, V_6 + V_0, V_7, V_0)$. We perform three XOR operations and may just reorder vectors V_i to avoid data transfers.

At the same time, $F(x)$ can be considered as a set of 512 elements of $GF(2^8)$ where V_0 contains all first bits of these elements, V_1 contains all second bits and so on, see Fig. 3. After performing multiplication of $F(x)$ by x in $\mathbb{I}$, all 512 elements of $GF(2^8)$ will be multiplied by x in the corresponding field.

As a result, we perform three XOR operation on 512-bit registers, reorder these vectors, and get 512 elements multiplied by x. These steps are repeated according to Horner scheme in checksum calculation or data recovery. This approach is very effective with vector registers. We will discuss the performance of this approach in Section VII.

VI. XEON PHI SPECIFIC OPTIMIZATIONS

Aside from algorithmic optimizations in Galois fields, one can utilize the following Xeon Phi specific optimizations to achieve better performance.

1) Many processors and threads

All threads of Intel Xeon Phi coprocessor should be used for calculations to maximize performance. We used OpenMP to organize threading. As it is shown in Fig. 1, each thread calculates its own set of codewords so there are no data dependencies. Threads perform calculations independently from each other.

2) Data allocation and transfer

This optimization applies only to offload mode, in which data stripes are allocated in the host and are transferred to the coprocessor to perform calculations. Each RAID in the storage systems has a fixed number of drives so all stripes from the same RAID have equal memory size. For this reason, memory for these stripes can be allocated at once using clauses in data transfer pragmas: `alloc_if(1), free_if(0)`.After that allocation on the coprocessor, memory can be reused by clauses: `alloc_if(0), free_if(0)` in all repeating data transfers. Implementation of the memory re-usage significantly increases encoding and decoding speed in offload mode.

3) Threads placement

Intel OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. Depending upon the topology of the computer, thread affinity can have a significant effect on a program performance. Available values of thread binding are compact, scatter and balanced. In our performance tests compact and balanced demonstrated 5-10% better results for both coding algorithms.

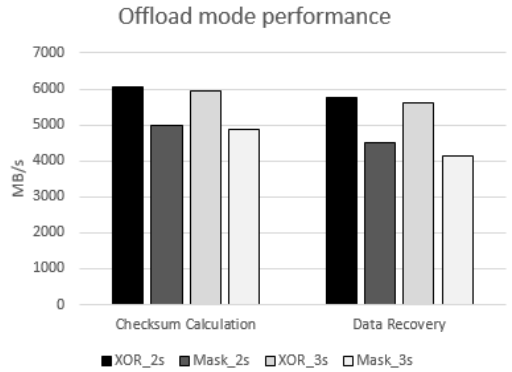

Fig. 4. Algorithms performance in offload mode.

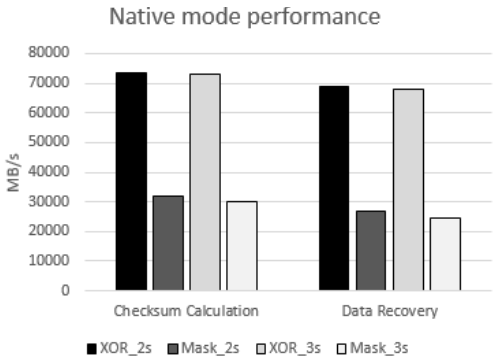

Fig. 5. Algorithms performance in native mode.

We used the aforesaid algorithmic and compiler optimizations to achieve the performance rate which will be discussed in the next section.

VII. Performance Measurements

To perform coding speed measurements in offload mode we allocated data stripes in RAM, measured time required to transfer data to Xeon Phi, performed checksum calculations or data recovery, and transferred calculated checksums or recovered data back. We made this test 1000 times for a fixed number of data drives (16-128) and 2 or 3 checksum drives in stripes. In native mode, we performed the same steps except the data transfers.

The system in our performance tests has the following configuration and components: Linux Red Hat 6.6, Host CPU Intel Core i5-4460 3.20GHz, MPSS 3.5.1, Intel C Compiler 15.0.3, Intel Xeon Phi 3120P.

Fig. 4 demonstrate the measurement results of checksum calculation and data recovery in offload mode. The algorithm performing Galois field operations by generating masks of irreducible polynomial is designated as "Mask", and our suggested XOR-based algorithm is designated as "XOR". "2s" or "3s" correspond to two or three checksums calculations or two or three blocks recoveries. It should be noted that in offload mode maximum bandwidth of data transfer through PCIe is up to 6.5 GB/s. Our XOR-based algorithm demonstrated an almost maximal achievable speed. We implemented this algorithm for the host CPU using AVX instructions. Although it showed a good performance rate, the CPU utilization was very high (up to 95-100%), whereas using the co-processor reduced it to 25-30% for all tests. This result was achieved by offloading computational parts, so the CPU managed only data transfers.

Fig. 5 show the performance results in native mode. As one can see, XOR-based algorithm in this mode significantly overcomes algorithm with masks.

Moreover, the current generation of Intel Xeon Phi is based on a chip codenamed Knights Corner (KNC). The next generation of Intel MIC architecture will be based on the Knights Landing chip (KNL). It will be available not only as a coprocessor but also as a standalone processor. In standalone version the MIC processor will have direct access to the RAM memory, so there will be no PCIe data transfer bottleneck. In this configuration, the performance we achieved in native mode will be available for the host system.

VIII. Conclusion

This paper provides an overview of using the modern coprocessor Intel Xeon Phi for RAID erasure coding computations. We have considered some technical optimizations such as data transfers, vectorization and others to achieve good performance rates. From the algorithmic point of view, we suggested a new approach to perform basic Galois field operations as vector XOR operations. The suggested method demonstrates a good performance rate both in offload and in native modes, allows utilization of 512-bit vector registers of the coprocessor and is highly parallelizable. Suggested method can be easily implemented for the host CPU using AVX or SSE extension. The significant result of using Intel Xeon Phi is that along with demonstrating a high speed of encoding and decoding in RAID, it reduces host CPU utilization, which is very important in storage systems.

IX. Acknowledgments

The authors gratefully acknowledge the constructive suggestions of Prof Alexei Uteshev which helped to improve the quality of the paper. We also thank the anonymous reviewers of this paper for their useful remarks.

X. References

[1] James S. Plank and Lihao Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications" *The 5th IEEE International Symposium on Network Computing and Applications*, 2006.

[2] James S. Plank, Kevin M. Greenan and Ethan L. Miller. "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions" *FAST 2013: 11th USENIX Conference on File and Storage Technologies*, San Jose, CA, February, 2013.

[3] H. Peter Anvin, "The mathematics of RAID-6", https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf

[4] GF-Complete and Jerasure, http://web.eecs.utk.edu/~plank/plank/www/software.html

[5] Intel Storage Acceleration Laibrary https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version

[6] R. Lidl and H. Niederreiter. "Introduction to finite fields and their applications", revised edition. Cambridge University Press, 1994