Erasure and Corruption Resilience Methods for High Performance Parallel File Systems

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim
Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Gvozden Nešković
aus Gračanica, Bosnien und Herzegowina

Frankfurt am Main 2018
(D 30)

vom Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
als Dissertation angenommen.

Dekan	Prof. Dr. Andreas Bernig
Gutachter	Prof. Dr. Volker Lindenstruth
	Prof. Dr. Ivan Kisel
Datum der Disputation	26.10.2018

Abstract

We live in age of data ubiquity. Even the most conservative estimates predict exponential growth in produced, transmitted and stored data. Big data is used to power business analytics as well as to foster scientific discoveries. In many cases, explosion of produced data exceeds capabilities of digital storage systems. Scientific high-performance computing environments cope with this problem by utilizing large, distributed, storage systems. These complex systems can only provide a high degree of reliability and durability by means of data redundancy. The most straight-forward way of doing that is by replicating the data over different physical devices. However, more elaborate approaches, such as erasure coding, can provide similar data protection while utilizing less storage. Recently, software-defined reliability methods began to replace traditional, hardware-based, solutions. Complicated failure modes of storage system components also warrant checksums to guaranty long-term data integrity. To cope with ever increasing data volumes, flexible and efficient software implementation of error correction codes is of great importance. This thesis introduces a method for realizing a flexible Reed-Solomon erasure code using the "Just-In-Time" compilation technique. By exploiting intrinsic arithmetic redundancy in the algorithm, and by relying on modern optimizing compilers, we obtain a throughput-efficient erasure code implementation. Additionally, exploitation of data parallelism is achieved effortlessly by instructing the compiler to produce SIMD code for desired execution platform. We show results of codes implemented using SSE and AVX2 SIMD instruction sets for x86, and NEON instruction set for ARM platforms. Next, we introduce a framework for efficient vectorized RAID-Z redundancy operations of ZFS file system. Traditional, table-based Galois field multiplication algorithms are replaced with custom SSE and AVX2 parallel methods, providing significantly faster and more efficient parity operations. The implementation of this framework was made publicly available as a part of *ZFS on Linux* project, since version 0.7. Finally, we propose a new erasure scheme for use with existing, high performance, parallel filesystems. Described reliability middleware (ECCFS) allows definition of flexible, file-based, reliability policies, adapting to customized user needs. By utilizing the block erasure code, the ECCFS achieves optimal storage, computation, and network resource utilization, while providing a high level of reliability. The distributed nature of the middleware allows greater scalability and more efficient utilization of storage and network resources, in order to improve availability of the system.

Contents

1 Introduction **1**
- 1.1 Storage technologies . . . 2
- 1.2 Storage reliability . . . 2
- 1.3 RAID . . . 8
- 1.4 ZFS . . . 10
- 1.5 Lustre . . . 11
- 1.6 Summary . . . 13

2 Introduction to Error Correction Codes **15**
- 2.1 Mathematical concepts . . . 16
  - 2.1.1 Groups . . . 16
    - Finite Groups . . . 18
  - 2.1.2 Fields . . . 19
    - Finite Fields . . . 20
    - Vector Spaces over Finite Fields . . . 23
    - Construction of Galois Fields . . . 25
- 2.2 Linear Block Codes . . . 29
  - 2.2.1 Introduction to block codes . . . 30
  - 2.2.2 Generator matrix . . . 31
  - 2.2.3 Bounds of linear block codes . . . 33
  - 2.2.4 Cyclic block codes . . . 35
- 2.3 Reed-Solomon Code . . . 36
  - 2.3.1 Vandermonde Reed-Solomon code . . . 37
    - Erasure decoding . . . 38
  - 2.3.2 Distributed Reed-Solomon code . . . 39
- 2.4 Algebraic signatures . . . 41

3 On Implementing Reed-Solomon Codes **45**
- 3.1 Galois Field multiplication . . . 46
  - 3.1.1 Carry-less multiplication . . . 46
  - 3.1.2 Modulo operation . . . 47
- 3.2 Reed-Solomon Encoding . . . 50

3.3 Summary 53

4 Micro Benchmarks **55**
4.1 Arithmetic and logic operations 56
4.2 Memory load throughput 56
4.2.1 Multi stream memory throughput 58
4.2.2 Memory load latency 60
4.3 Conclusion 63

5 JIT Generation of Reed-Solomon Erasure Codes **65**
5.1 Vectorization 66
5.1.1 Carry-less multiplication 67
5.1.2 Modulo operation 71
5.1.3 Evaluation 72
5.2 Just-In-Time compilation of Reed-Solomon codes 73
5.2.1 LLVM as a JIT compiler 74
LLVM IR representation 75
5.2.2 Reed-Solomon Encoding 77
5.2.3 Carry-less multiplication 79
5.3 Evaluation 80
5.4 Summary 86

6 Vectorization of ZFS Erasure Codes **89**
6.1 RAID-Z theoretical background 90
6.2 Implementation 92
6.2.1 RAID-Z parity generation 94
6.2.2 RAID-Z data reconstruction 96
6.3 Evaluation 100
6.4 Summary 104

7 ECCFS Middleware **105**
7.1 Motivation 106
7.2 Design of ECCFS 108
7.2.1 ECCFS Monitor 110
7.2.2 ECCFS Changelog 111
7.2.3 ECCFS MD Worker 111
7.2.4 ECCFS ECC Worker 112
7.3 Implementation 112
7.3.1 ECCFS Monitor 113
7.3.2 ECCFS Changelog 115
7.3.3 ECCFS MD Worker 116
7.3.4 ECCFS ECC Worker 117

7.3.5 Deployment and Administration 119
7.4 Summary 120

8 Summary **123**

References **127**

A Appendix **135**
A Test-bed platforms 136
B Example of Hexagon DSP VLIW code 137
C Galois Field used in RAID-Z 138

List of Tables **143**

List of Figures **145**

List of Algorithms **147**

List of Listings **149**

Zusammenfassung **151**

CHAPTER 1

Introduction

This chapter gives overview of current storage technologies and file systems used in distributed storage environments. Fundamental concepts of error correction coding theory, necessary for understanding the work in the thesis, are found in chapter 2. Next, in chapter 3, implementation aspects of the Reed-Solomon block-based erasure code are discussed. chapter 4 gives an overview of the micro-benchmarks that are used to understand limitations and performance optimization of different computing platforms. An implementation of the multi-platform, JIT generated, vectorized Reed-Solomon block-based erasure codes is presented in chapter 5. chapter 6 shows a practical implementation of erasure codes for storage systems used in the ZFS file system. Design and implementation of middleware for adding error correction codes for the data stored in a parallel file system is given in chapter 7. Finally, chapter 8 provides a summary and outlook for future work.

1.1 Storage technologies

Memory hierarchy in computer architecture distinguishes each level by capacity and response time. In the most general form, observed from a data processing unit's perspective, memory hierarchy has four levels:

1. **Internal** includes CPU registers and caches. As such it is the most capacity constrained but it is able to operate at, or close to, the speed of CPU.
2. **Local** memory includes directly attached random access memory, RAM, but also memory of other CPUs in *non-uniform memory architecture* (NUMA) configurations.
3. **Mass, secondary**, level consists of *hard disk drives* (HDD) and *solid state drives* (SDD). All devices of this level are online and support non- volatile operation.
4. **Off-line, tertiary**, storage encompasses data storage devices and mediums that are not directly controlled, or accessible, from a processing unit. Advantages of these mediums, such as magnetic tape and optical disc, are long data retention and much better *price per gigabyte* ratio.

In reality, however, storage hierarchy is not demarcated so precisely. Faster, non-volatile storage technologies, such as *non-volatile RAM* (NVRAM), are used as an intermediate cache layer between RAM and hard drives[57][64]. *Nearline storage*[83] is a class of long-term storage that can be made online automatically, but with a delay. This is achieved by using a robotic arm to place magnetic tape into the tape drive, or by optical disc jukebox systems[39]. *The Massive array of idle drives* (MAID), is a nearline technology that utilizes hard drives for data archival. It relies on the fact that disk drives are powered only when in use, enabling higher capacity densities with a fraction of the power and cooling requirements used in comparable online storage. A comparison of desktop, enterprise and nearline hard drives is given in Table 1-1.

Reliability specifications vary form one HDD vendor to another significantly, and are given by different metrics[1]. Big studies [100][86][54][75] found significantly larger failure rates than were specified. To better understand how these reliability metrics affect reliability of a storage system, we present two Markov chain models for reliability of more complex storage systems.

1.2 Storage reliability

Capacity increase of hard disk drives has not shown signs of slowing, but other characteristics have not kept the same pace. Linear read/write throughput is limited by physical rotational speed of the platters. Unfortunately, reliability characteristics have

[1] Power-on hours (POH), Annualized Failure Rate (AFR), Mean Time To Failure (MTTF)

Specification (Unit)	**Enterprise**	**Desktop**	**Nearline**
Capacity (TB)	0.3-1.8	1.0-6.0	6.0-10.0
Rotational speed (RPM)	10.5 - 15	5.4 - 7.2	5.4 - 7.2
Throughput ($\mathrm{MB\,s^{-1}}$)	250	180	150
Reliability (MTTF)	2 000 000 h	600 000 h	1 500 000 h
Un-recoverable error rate ($\mathrm{bit^{-1}}$)	10^{-16}	10^{-14}	10^{-15}
Load/Unload (cycles)	600 000	300 000	300 000
Interface	SAS	SATA	SATA
Interface speed(Gbps)	12 Gbps	3/6	6
Power operation/idle (W)	7.4/5.1	5.6/4	6.5/4.8

Table 1-1: Comparison of hard disks (mid 2016) [110][108][102][46]

not improved at the same rate, if at all. Current server grade hard drives[2] offer capacities from 1 TB up to 8 TB, with a sustained data transfer rate of $200\,\mathrm{MB\,s^{-1}}$. The hard drive has a *mean time between failures*[3] (MTBF) of 2 000 000 h, and a *bit error rate*[4] (BER), of 10^{-15}. However, reliability studies have shown that the specified numbers are usually much different. Authors of [25] have fund HDD MTBF to be between 87 600 and 438 000 h, while another big survey of cloud storage installation[75] has found the average *annualized failure rate* (AFR) to be 2.12%. The relation between MTTF and AFR is:

$$\mathrm{MTBF} = \frac{8760}{\mathrm{AFR}}\, 100 \quad [\mathrm{days}] \tag{1.1}$$

This gives the MTBF of 413 207 h for the [75] study. Another big concern with increasing HDD capacities are *unrecoverable read errors* (URE). These errors can be transient, caused by an undetected error in magnetic flux density decoding, or permanent, caused by physical errors of the magnetic medium. This presents problems, especially for traditional RAID systems, where the integrity of data is not verified with checksums. With increasing HDD capacities, the likelihood of URE during a RAID rebuild operation has increased significantly. Even worse, the process is likely to spread the error to newly reconstructed blocks[17]. The solution proposed in [65] tries to address the issue by using triple-parity RAID systems.

To illustrate the problem of data integrity with traditional erasure coding, such as RAID and Reed-Solomon codes, we extended the reliability of Markov models for the code given in [94] and [40]. Instead of solving for *mean time to data loss* (MTTDL), as shown in [55], we are going to investigate the influence of data checksumming on the

[2]Based on WD Gold™ datacenter hard drives specifications (model number WD6002FRYZ, 2016)
[3]Since HDDs are typically non-repairable, this is a measure of Mean Time To Failure (MTTF)
[4]Also called *unrecoverable read errors*, URE

erasure recovery process of the δ-erasure resistant code. We will call this metric *mean time to data corruption* (MTTDC). Silent errors can be caused by many factors, for example during the reading of a magnetic medium, during transmission, RAM bit flip, etc. Since the rate of these errors is not known, we assume that the specified *read bit rate* is a combination of detectable and silent errors. The transition rate diagram of the Markov model for a traditional k-erasure resistant code is shown in Figure 1-1. The model has the following parameters:

d	Number of disks in an erasure code block
δ	Number of erasures the code can recover from
λ	Failure rate of an HDD, given as MTBF^{-1} [h^{-1}]
μ	Rebuild rate of the array [h^{-1}]
RER	Read error rate, per HDD [B^{-1}]
SR	Proportion of silent errors in RER

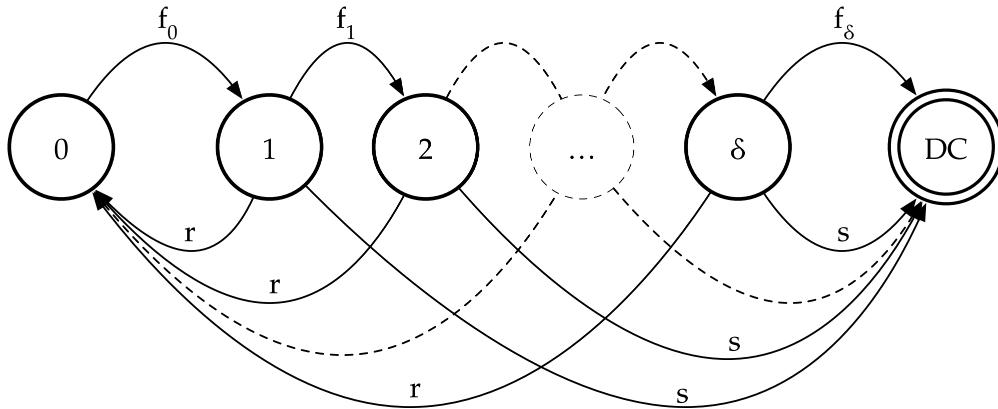


Figure 1-1: Transition rate diagram of parallel rebuilding of δ-erasure tolerant code without data checksum validation.

Model's states represent operation modes, where 0 is a fully operational state with no erasures, states 1 to k represent a state with a corresponding number of erasures, and finally, **DC**, which is a data corruption state. For our analysis, we'll assume that the system is capable of returning from an arbitrary number of erasures to a fully operational state at the same rate r, shown by arcs from states i to state 0 ($0 < i \leqslant \delta$). In other words, rebuilding is performed in parallel, utilizing exactly $(d - \delta)$ operational HDDs. Transitions from state i to state $i+1$, labeled f_i, represent the rate at which the system encounters a new HDD failure after already experiencing i failures. Finally, the system can transition from any of the erasure states to the **DC** state, with a rate s, which is the rate of encountering silent errors during rebuild. Probability of read error during rebuild, per single HDD, is calculated as $h = 1-(1-\mathrm{RER})^{C}$, and probability of read error happening

on any of $d - \delta$ disks participating in rebuild[33] is given by $h_r = 1 - (1 - \text{RER})^{(d-\delta)C}$. Compound transition rates for this model are:

$$\begin{aligned} \mathbf{f}_i &= (d - i) \times \lambda + (1 - SR) \times h_r \times \mu \\ \mathbf{r} &= (1 - h_r) \times \mu \\ \mathbf{s} &= SR \times h_r \times \mu \end{aligned} \tag{1.2}$$

From this formulation, we derive steady state probability distribution using a transition generator matrix of the system. Since state **DC** is clearly an absorbing state of the system, we define the total system failure rate as the total flow rate into that state. The infinitesimal generator matrix of the model in Figure 1-1 is:

$$\mathbf{Q} = \begin{bmatrix} -f_0 & f_0 & 0 & \cdots & 0 & 0 \\ r & -(r + f_1 + s) & f_1 & \cdots & 0 & s \\ r & 0 & -(r + f_2 + s) & \cdots & 0 & s \\ \vdots & \vdots & \vdots & \ddots & f_{\delta-1} & \vdots \\ r & 0 & 0 & \cdots & -(r + f_\delta + s) & f_\delta + s \end{bmatrix} \tag{1.3}$$

Each column of the generator matrix represents a state of the model, with the last being the absorbing state **DC**, while rows represent non-absorbing states ($\delta + 1$ in total). Non-diagonal entries represent transition rates from *row* state to *column* state. Entries on the diagonal a are a negative row sum of off-diagonal elements, so that each row sums to 0. The MTTDC($\mathbf{Q}$) is calculated by constructing matrix $\mathbf{Q}'$, as a negative matrix of $\mathbf{Q}$ after removing the last column [40]. The following formula gives the mean time to data corruption of the Markov model:

$$\begin{aligned} \text{col}_i(\mathbf{Q}') &:= \text{col}_i(-\mathbf{Q}) \quad (i = 1, \ldots, \delta + 1) \\ \text{MTTDC}(\mathbf{Q}) &= \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix} \mathbf{Q}'^{-1} \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T \end{aligned} \tag{1.4}$$

Using data integrity verification during rebuilding can increase MTTDC of the system. Silent errors during erasure reconstruction can be detected, and the system can use another parity HDD to finish the process. The Markov model of such a system is shown in Figure 1-2. Recovery from states less than δ is not affected by silent errors since another HDD can be chosen to complete the reconstruction. This holds if the system has less than δ erasures. If a silent error is encountered in state δ, the system transitions into

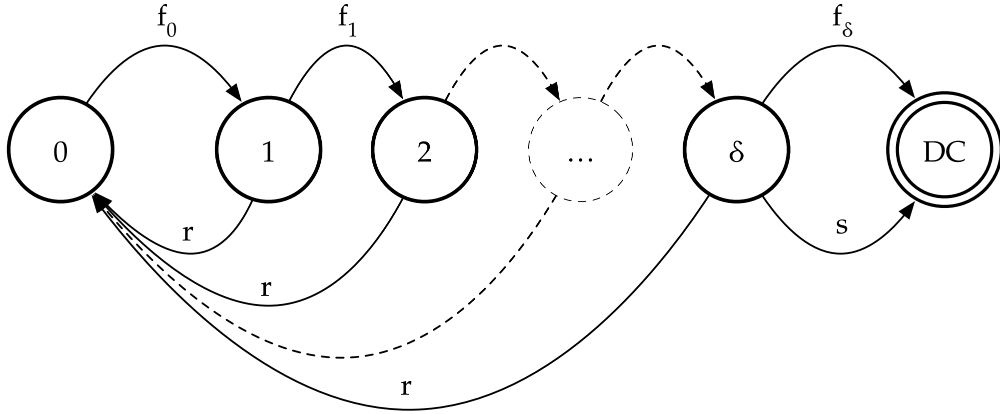


Figure 1-2: A revised transition rate diagram of the parallel rebuilding of an δ-erasure tolerant code with data integrity validation

state **DC** because no more available replicas exist, as shown by transition s. Transition rates for this model are:

$$\begin{aligned} f_i &= (d - i) \times \lambda + h_r \times \mu \\ r &= (1 - h_r) \times \mu \\ s &= SR \times h_r \times \mu \end{aligned} \tag{1.5}$$

To evaluate these two models, we use typical values for parameters [32][75], given in Table 1-2. We use disk capacity of $4\,\mathrm{TB}$ and assume a rebuild time of $12\,\mathrm{h}$.

Parameter	Value
λ	$1/(1 \times 10^{6})\,\mathrm{h}^{-1}$
μ	$1/12\,\mathrm{h}^{-1}$
C	$4\,\mathrm{TB}$
RER	$1 \times 10^{-15}\,\mathrm{bit}^{-1}$
SR	$1/100$
h	$0.003\,546\,4$

Table 1-2: Parameters of the δ-erasure resilient Markov models

We evaluated both models using the symbolic computation in SageMath[99] mathematics software, because resulting matrices are numerically ill-conditioned. Constants were represented using *Rational number* data type, and final evaluation is carried out using high precision *Real number* types that provide arbitrarily high numerical precision. Results are shown in Figure 1-3.

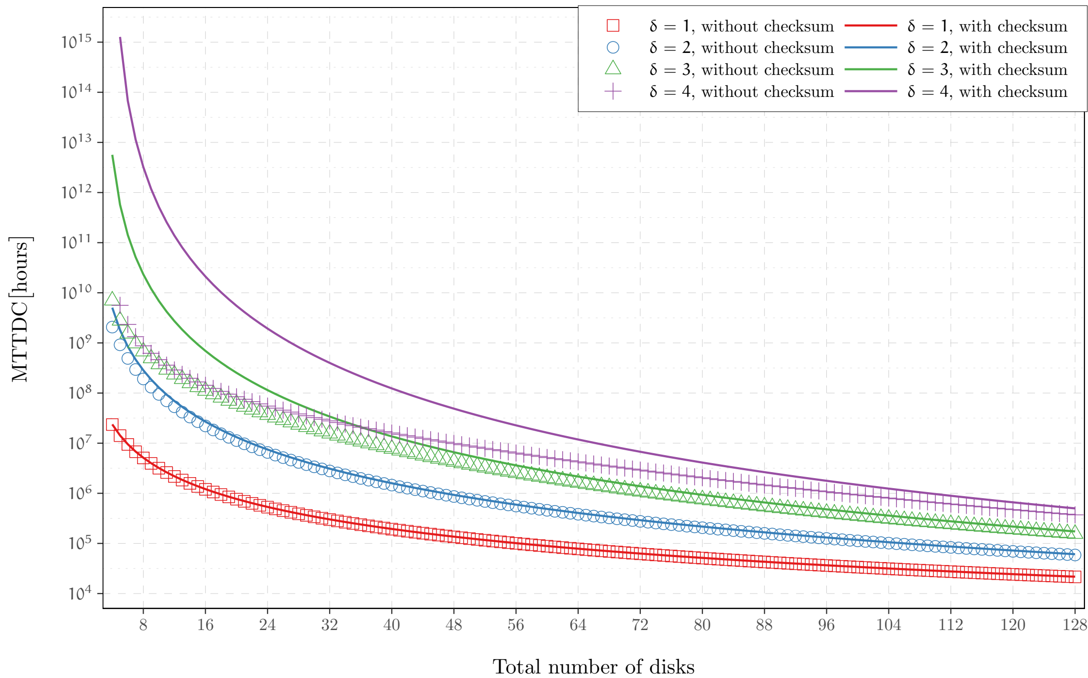


Figure 1-3: Mean Time To Data Corruption of δ-erasure resilient code; with and without checksumming during reconstruction

The graph shows MTTDC for different erasure resiliency code configurations, with and without data verification on rebuild. The total number of disks is shown on the x-axis. Both models show the same value for $\delta = 1$, because for that case, their transition rate Markov chains are equivalent. The results show two important aspects of redundant arrays. First, solutions of both models show that MTTDC is directly proportional to $\sim (\mu)^{\delta}$, also observed in [25][40]. This indicates that the best recovery strategy is to rebuild as fast as possible. Second, MTTDC decreases with adding more disks into the erasure code block, with a rate of $\sim 1/(d\lambda)^{\delta}$.

Codes with low resiliency, $\delta \leqslant 2$, show only marginal improvement in the enhanced model. This is explained by the small finite number of states from which the system can recover (δ) and a relatively high likelihood of read error during the rebuild. The likelihood of encountering the RER for $d \times$ HDD array is proportional to $\sim \text{RER}^{(d-\delta)C}$. Finally, for more erasure-resilient codes, $\delta > 2$, results show improvement in MTTDC when data verification is used. Even with a conservative amount of silent errors, 1 % of all RER errors, data verification provided a significant increase in MTTDC.

The described models can also be used to describe replicated storage systems. To compare the reliability metric of replication and erasure coding, we use the second Markov model in Figure 1-2. Data replication is modeled as a δ-resilient erasure code where $\delta = d - 1$. We use the same recovery rate for both cases because replicating the entire

HDD is limited mostly by single disk I/O bandwidth. Mean time to data corruption, as a function of storage overhead that is used for redundancy, is shown in Figure 1-4.

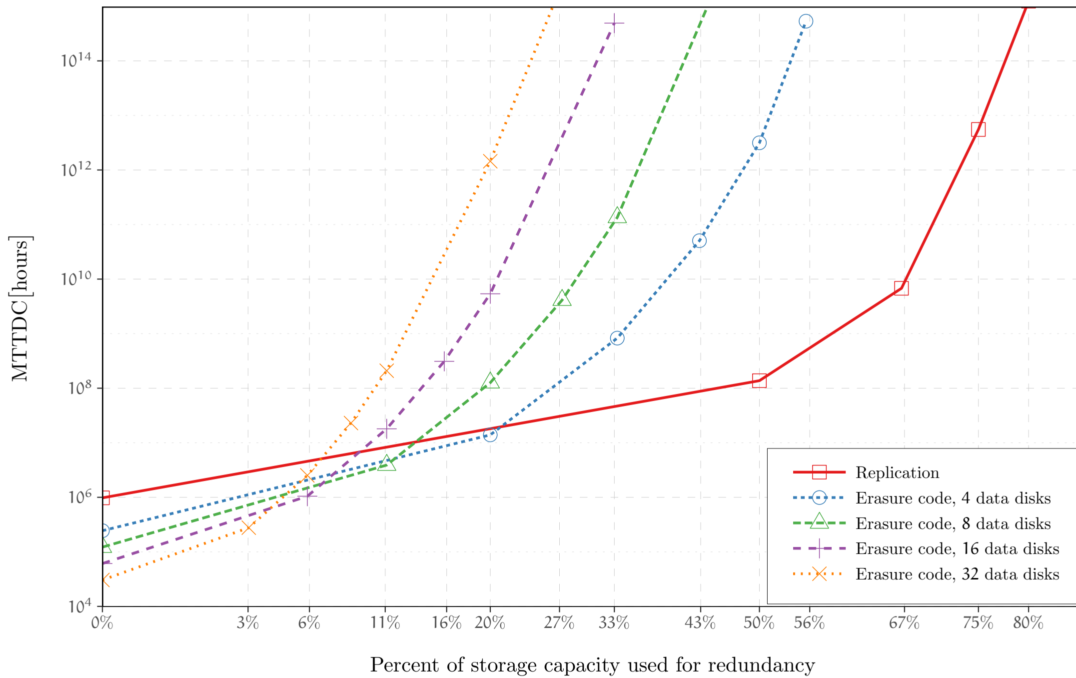


Figure 1-4: MTTDC as a function of redundancy overhead for replication and δ-resilient erasure codes

Results show that erasure coding requires less disk capacity for the same level of reliability. Paper [13], which compares the erasure coding and replication, concludes that while replication has lower storage costs, replicated systems may provide lower latency and simpler implementation. A hybrid approach is described in [48], where new data is first replicated, and erasure encoded later for long term storage.

1.3 RAID

The most dominant technology for data redundancy and performance improvement used today is RAID, which stands for redundant array of inexpensive disks. Since its introduction [82], different configurations of disks, so called RAID *levels*, have been used for grouping disks into fault-tolerant arrays. Standard RAID levels include:

RAID0 array has no parity. Drives are organized in a striping pattern, which distributes data equally among all drives. Concurrent requests to RAID0 array are serviced by rate of a single disk multiplied with number of disks.

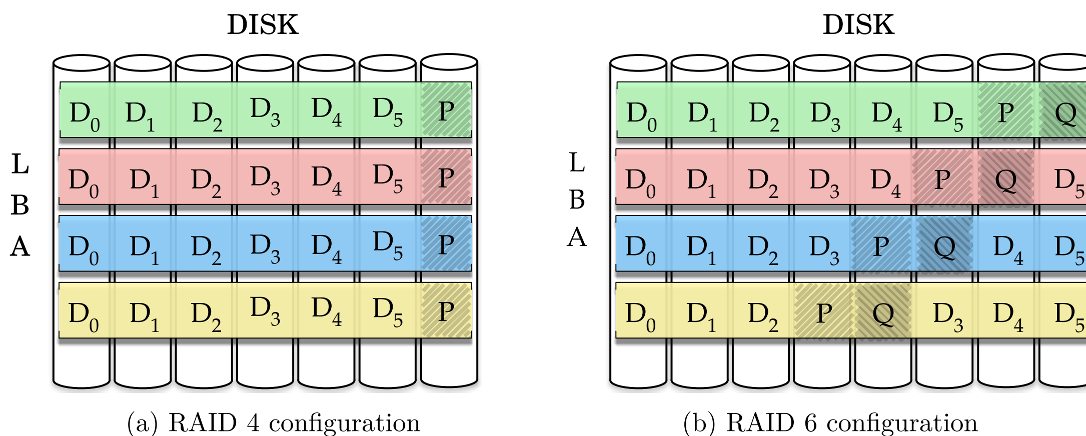
RAID1 mirrors data on all available drives. RAID1 array is able to operate as long as a single drive is operational, thus providing the best reliability. However, its capacity is limited to the capacity of the smallest drive in the array.

RAID4 array introduces block parity to achieve data reliability. Parity block, XOR of all blocks in the horizontal group, is stored on a dedicated drive, as shown in Figure 1-5(a).

RAID5 array performs block-level striping with one distributed parity. With the single parity, RAID5 array continues to be operational with one drive failure.

RAID6 is similar to RAID5, but it uses double parity, which increases fault tolerance to two drives.

The data/parity distribution scheme of RAID6 is shown in Figure 1-5(b). Two parity symbols, P and Q, are calculated, usually using a variant of the Reed-Solomon codes[1]. Parity symbols are interleaved with data, spreading them equally among all drives in the array.



(a) RAID 4 configuration

(b) RAID 6 configuration

Figure 1-5: Standard RAID configurations

Every modification of data causes the parity symbol to be updated, reducing the write performance to the RAID6 array. Furthermore, the write operation requires atomicity in updating data and party disks. Otherwise, a power failure or system crash can cause inconsistency between data and parity, referred to as *RAID write hole*, which would cause errors during recovery [2]. For this reason, hardware RAID controllers with write-back caches include a battery pack to preserve data until the system comes online again. Software implementations solve the problem by utilizing transaction journaling techniques.

Read error rates of hard drives are unlikely to decrease with increase of capacity, which means that the probability of read error during a RAID rebuild becomes a significant factor E.g., chances of read error during recovery of the RAID6 array consisting of

$8 \times$ HDD, with parameters as in Table 1-2, is 17 %. This problem has has long been recognized [65], which prompted storage vendors to stop recommending RAID5 solutions for business-critical data [21].

1.4 ZFS

Traditionally, file systems have not offered any data protection. Data integrity can be compromised in many ways due to many *byzantine faults* that can occur anywhere in the storage hardware and software stack. It has been proven that traditional parity-based RAID techniques are not sufficient protection against increasingly complex failures in both hardware and software components[60]. Such failures include silent data corruption[79][3][22], hard drive firmware errors[103], various software and driver errors[92], network errors, etc.

ZFS is a disk file system that integrates several management technologies that are traditionally not part of a file system. The Logical volume manager, LVM, is a base infrastructure of ZFS that provides software RAID and data integrity features. ZFS LVM supports assembling underlying storage into mirroring and RAID-Z data/parity distribution schemes, called vdev[5]. It is generally advised not to use hardware RAID controllers and to let ZFS manage drives itself. ZFS is built with different storage technologies in mind. As such, most accessed data is stored in the memory, the so-called ARC[6] cache. For the second level of caches, ZFS uses fast non-volatile storage solutions, most prominently SSDs. Each vdev can have optional dedicated read and write caches. The read cache, called L2ARC can maintain a considerably large amount of frequently accessed data. The write cache is called *Log Device*, and is used to absorb random synchronous writes. ZFS forms an *Intent Log* from data buffered in the Log device, which enables faster sequential writes to underlying hard drives.

ZFS uses a transactional copy on write (COW) model for data and metadata. All on-disk block pointers contain checksums of referenced data, and metadata, block. Then, the block pointer itself is checksummed and its checksum value is saved in its parent block pointer, continuing all the way to the top level.

This technique is called Merkle tree[69][70], and allows for efficient and secure data integrity verification. Checksums are always verified when a block is read from a storage medium, and if needed, the block is reconstructed from a replica or by using the RAID-Z erasure scheme. ZFS supports online integrity verification and data reconstruction. Capacity of hard drives has increased exponentially, whereas the throughput has not kept up. This means that reconstruction of failed disk drives can take several hours up to a

[5] **V**irtual **Dev**ices

[6] ARC uses a variant of Adaptive replacement cache algorithm

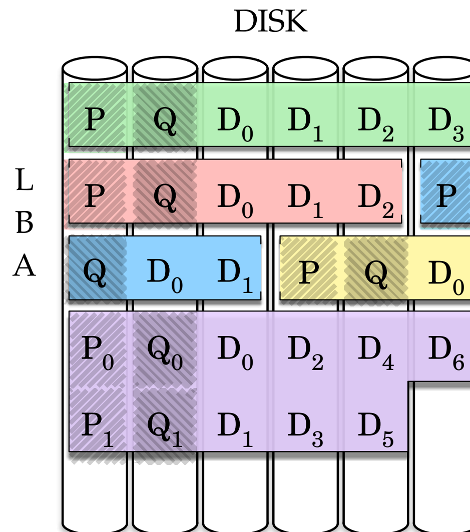


Figure 1-6: RAID-Z dynamic block layout

few days. Enabling this operation to happen while the file system is still available is of great significance for large storage systems installations[81].

RAID-Z is an advanced parity distribution scheme with the intention of providing software-defined RAID-like redundancy. In contrast to hardware solutions, RAID-Z uses dynamic stripe widths, which in combination with transactional COW semantics eliminates *write hole error*. The RAID-Z scheme is shown in Figure 1-6. There are three RAID-Z levels supported by ZFS. First, RAID-Z1, is similar to traditional RAID5, where the vdev can tolerate a single drive failure. RAID-Z2 is therefore akin to RAID6. Lastly, the RAID-Z3 erasure scheme supports recovery from three failures. Since the combined reliability of a redundant array ultimately depends on fast rebuilding, in this thesis we describe an implementation of vectorized methods for parity generation and data reconstruction for the ZFS file system.

1.5 Lustre

With the popularity of cluster computers came the need for distributed file systems. Scalability and capacity of *network attached storage* (NAS) systems quickly became limiting factors. Also, with advancements in network interconnects, parallel access to multiple storage systems became desirable. Lustre is an example of a parallel, POSIX-compliant, distributed file system[10] that fulfills many requirements of high performance computing (HPC). Currently it is the most popular open source file system in HPC and data center environments.

The main components of a typical Lustre installation are Metadata Servers (MDS), Object Storage Servers (OSS), and Lustre clients. The Metadata server uses the Metadata

Target (MDT) to store all file system metadata. Each OSS is connected to one or multiple Object Storage Targets (OST), which provide data storage for the file system. Lustre clients presents the entire file system's namespace using well-established POSIX semantics, while providing coherent and parallel access to the files. A diagram of Lustre components is shown in Figure 1-7.

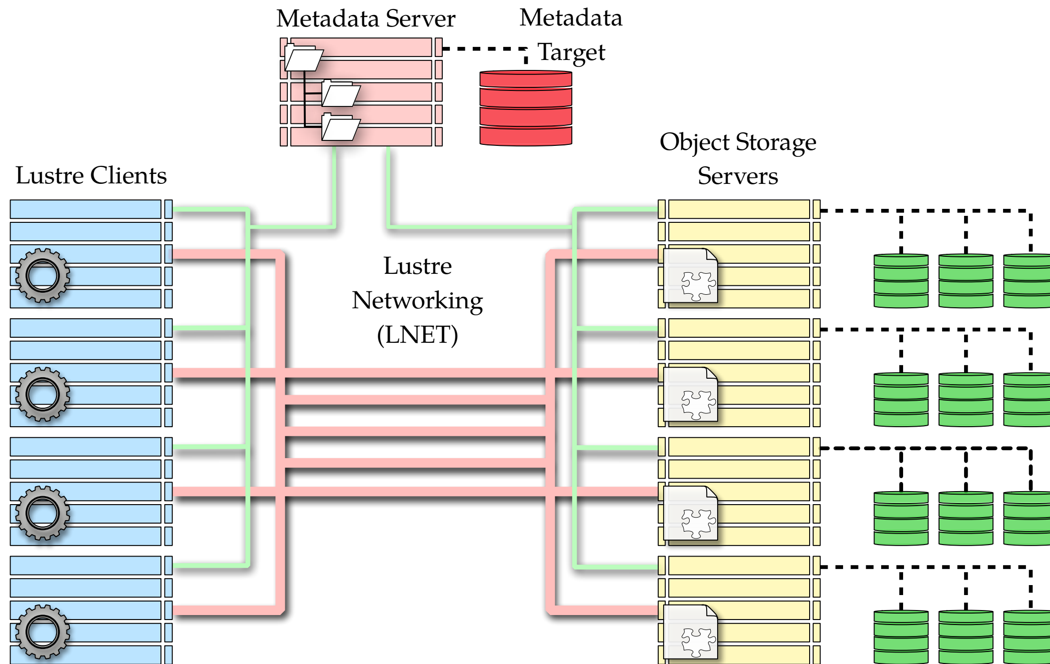


Figure 1-7: Components of Lustre file system installation

Lustre Networking (LNET) provides a connection for clients and servers of Lustre file system. It supports multiple networking technologies, such as Ethernet, InfiniBand, Omni-Path, and utilizes Remote Direct Memory Access (RDMA) if available. More complicated network installations are supported by LNET Router functionality. For availability, storage targets can be connected to multiple storage servers in active/active or active/passive failover configurations.

The Metadata server is only contacted for filename and permission checks, and is not involved in actual data operations. File inode[7] structures contain object layout which specifies object server and object number where file's data reside. Files in Lustre file system can be striped over multiple OSTs, similar to RAID0 scheme, to enables higher bandwidth for large files. Once a client has obtained the layout of desired file, all data read and write I/O operations are performed directly with corresponding OSTs. This enables total client I/O bandwidth to scale almost linearly with the number of OSTs in the system. To provide atomicity of the read/write operations, required by the POSIX

[7] An *inode* is data structure used to represent a file system object

semantics, Lustre uses distributed lock manager [107]. Locking is orchestrated by the MDS, and guaranties client cache coherency.

Data reliability is delegated to OSTs. Traditionally, a modified version of *ext3* file system, called ldiskfs, have been used on top of hardware RAID arrays, to provide object-based interface. As of the Lustre 2.4 version, the ZFS file system can be used instead. This change brings end-to-end data integrity, and online OST recovery and file system checking. Using ZFS to ensure OST reliability also removes the need for hardware RAID solutions. Instead, more cost-effective deployments using OSS with JBODs[8] are becoming more common. However, loss of an OST leads to an unrecoverable loss of data. Because the failed OST is used for storing parts of many files, the magnitude of unavailability or data loss is much greater. We propose a solution for this problem based on a flexible, file-level, OST de-clustered erasure scheme.

1.6 Summary

Reliability aspects of individual components are an important factor when estimating the reliability of complex systems. High capacity storage components, mainly hard disk drives, exhibit a set of failure modes which have to be understood. Many studies have concluded that specified reliability characteristics are not reflected in the real data. Standard RAID levels, as the industry de facto standard method for providing storage reliability, has been rendered unsuitable by disparity between capacity and reliability trends of HDDs. Furthermore, in order to deal with many types of failures, additional techniques, like checksumming and online data integrity checks, are required. Data replication can be used for increased reliability and availability, but requires significantly more storage capacity when compared to flexible erasure coding schemes.

Software defined reliability, provided by file systems or other layers of an operating system, requires less capital and operational costs. Removing hardware erasure coding accelerators, such as RAID controllers, requires an efficient, software-based, erasure scheme. Distributed file systems, while providing flexibility in usability and deployment, lack fine-grained and user-defined reliability. These problems provided motivation for an efficient, flexible and retargetable Reed-Solomon erasure scheme described in the following chapters.

[8] JBOD stand for "just a bunch of disks"

CHAPTER 2

Introduction to Error Correction Codes and Algebraic Signatures

Error correction codes are an integral part of digital data transmission and storage. They are used in cell phone networks, packet-based communication like internet, digital video broadcasts, deep space communication transmissions etc. They also have an important role in digital data storage and are found in devices like computer memories, compact discs, hard drives and solid state drives. While these examples are widely different in implementation and operation, they can be represented by a general model of communication, first formally introduced by C. Shannon in [104]. This model of message transmission over a noisy channel is shown in Figure 2-1. The original message is encoded to protect bits during transmission over channels that can introduce noise or distortion of data. The channel encoder achieves this by transforming the original message into alternate sequence containing redundancy, used to provide protection from

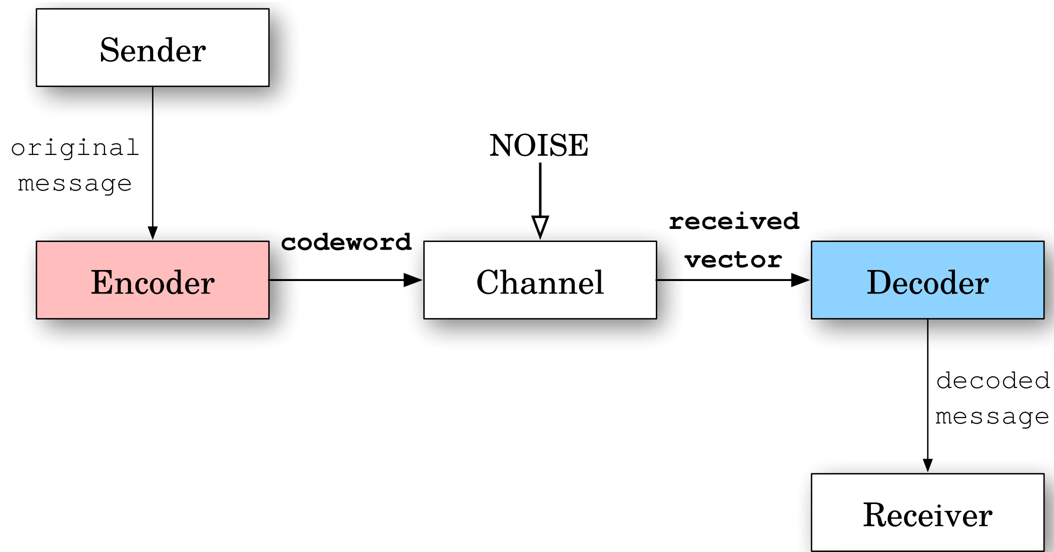


Figure 2-1: Communication system with a noisy channel

channel noise. The ratio of input and output data bits of the channel encoder is called the *code rate*, denoted R, with $0 < \mathsf{R} < 1$. For example, if a code rate of an encoder is 1/2, each codeword contains the same amount of redundant information as the message itself. The role of the channel decoder is to reconstruct the data received from the channel (received vector), and produce the original message data, using added redundancy to negate errors introduced by the channel. The *channel* is the transmission or storage medium used to convey the message. It is often prone to adding noise or interference from other signals. On the basis of this model, Shannon introduced a measure of how much information the channel can convey, called the *channel capacity* C. When C is expressed within the context of the channel code rate R, Shannon showed that arbitrary reliable codes exist provided that $\mathsf{R} < \mathsf{C}$ is satisfied. Conversely, if $\mathsf{R} > \mathsf{C}$, no code can provide reliable communication. Following we give an important mathematical concept needed for constructing error correction codes. Parts of this chapter are based on the presentation of the concepts in [111].

2.1 Mathematical concepts

Error-correcting codes and algebraic signatures described in this thesis are based on finite fields and the operations defined on them. This chapter outlines the basic algebra concepts used to construct such codes.

2.1.1 Groups

A group is an algebraic structure composed of a set with one binary operation defined on it.

Definition 2.1 A group is a set G together with a binary operation "$*$" defined on G such that the following axioms are satisfied:

1. The binary operation "$*$" is associative.
2. G contains an element e such that, for any element a of G,

   $$a * e = e * a = a$$

   Element e is called an *identity element* of G with respect to the "$*$" operation.
3. For any element a in G, there exists an element a' in G such that

   $$a * a' = a' * a = e$$

   The element a' is called an *inverse element* of a, and vice versa, with respect to the "$*$" operation.

Definition 2.2 A group is called *abelian group* (commutative group) if the binary operation "$*$" defined on it is also commutative.

Theorem 2.1 The identity element e of a group G is *unique*.

Proof. Suppose both e and e' are identity elements of G. Then

$$e * e' = e \text{ and } e * e' = e'.$$

This implies that e and e' are identical. Therefore, the identity of a group is unique. □

Theorem 2.2 The inverse of any element in a group G is *unique*.

Proof. Let a be an element of G. Suppose a has two inverses, a' and a''. Then

$$a'' = e * a'' = (a' * a) * a'' = a' * (a * a'') = a' * e = a'$$

This implies that a' and a'' are identical and therefore, a has a unique inverse. □

One example of a group we are familiar with is set of rational numbers, together with a real addition operation "$+$". The number 0 is the identity element of this group. The rational number of form a/b has an additive inverse of the rational form $-a/b$ (b is a nonzero integer) and vice versa. Another example of a commutative group is a set of all rational numbers excluding 0 under the multiplication operation "$\cdot$". In this case, integer 1 is the identity element, and the rational number a/b is the inverse of the rational number b/a (both a and b are nonzero integers) with respect to multiplication operation.

Finite Groups

Finite groups are of special practical importance, particularly in the field of coding theory. In the following, a special class of finite group is described. The binary operation of these groups is similar to real multiplication, hence these groups are also called the *multiplicative groups*.

Definition 2.3 (Finite group) Group G is called *finite* if it contains a finite number of elements. The number of elements in a finite group is called the *order* of the group.

Following describes definition of closed binary operation on finite fields whose order is a prime.

Let p be a prime number. Consider the set of $p-1$ integers less than p, $G = \{1, 2, ..., p-1\}$. Let "$\cdot$" denote real multiplication. Every integer element i of G is relatively prime to p. A binary operation "$\circ$" on G is defined as follows: for any two integers, i and j in G

$$i \circ j = r$$

where r is the reminder resulting from dividing $i \cdot j$ by the prime p. It follows that $i \cdot j$ is not divisible by p, and therefore r cannot be zero. As a result, $1 \leqslant r < p$ and r is an element in G. Therefore, the set $G = \{1, 2, ..., p-1\}$ is closed under the operation "$\circ$", which is called *modulo-p multiplication*. Set G together with modulo-p multiplication makes a finite group of the order $p-1$. The modulo-p multiplication "$\circ$" is in fact both associative and commutative. Element 1 is the identity element of G.

Definition 2.4 (Multiplicative Groups) Set of integers $G = \{1, 2, ..., p-1\}$, where p is a prime number, together with modulo-p multiplication constitutes a class of finite groups called *multiplicative group*.

Let i be an integer in G. Since i and p are relatively prime, according to the Bézout's identity, there exist two relatively prime integers a and b such that

$$a \cdot i + b \cdot p = 1$$

From the last identity, it follows that:

$$a \cdot i = -b \cdot p + 1$$

The last equation says that reminder of the product $a \cdot i$ divided by p is 1. Choosing a form G, that is $1 \leqslant a < p$, we have:

$$a \circ i = i \circ a = 1$$

Therefore, a is inverse of i and vice versa. When a is not an integer from G, a is divided by p which gives:

$$a = q \cdot p + r$$

Since r must be an integer between 1 and $p - 1$ it is an element of G. Finally, it follows that

$$r \cdot i = -(b + q \cdot i) \cdot p + 1$$

As $r \cdot i = i \cdot r$, from the definition of modulo-p multiplication, it follows that $r \circ i = i \circ r = 1$, and that r is the inverse of i with respect to the "$\circ$" operation. This completes the proof that set $G = \{1, 2, ..., p - 1\}$ with modulo-p multiplication is a *commutative group*.

Definition 2.5 *Powers* of element a from the multiplicative group $G = \{1, 2, ..., p - 1\}$ under the modulo-p multiplication "$\circ$" are defined as follows:

$$a^1 = a, \quad a^2 = a \circ a, ..., \quad a^i = \underbrace{a \circ a \circ a \circ \cdots \circ a}_{i \text{ factors}}$$

As the module-p multiplication is closed in group G, it follows that all powers are also elements of G.

Definition 2.6 A multiplicative group G is said to be *cyclic* if there exists an element a in G such that, for any b in G, there exists an integer i which satisfies $b = a^i$. Such element a is said to be a *generator* of the cyclic group, denoted as $G = \langle a \rangle$.

More than one generator may exist for a cyclic group. For every prime p, the multiplicative group $G = \{1, 2, ..., p - 1\}$ under modulo-p multiplication is cyclic.

2.1.2 Fields

Fields are useful in variety of practical areas. The field is an algebraic system with two binary operations. Fields with a finite number of elements, called *finite fields*, are widely used in constructing efficient error-correction codes. Some of the well-known error-correction codes that are based on finite fields are *BCH* (Bose–Chaudhuri–Hocquenghem) codes [9] and *RS* (Reed-Solomon) codes [95]. More recently, *LDPC* (low-density parity-check) codes [28] have been based on the finite fields as well.

Definition 2.7 Let F be a set of elements on which two binary operations, called *addition* "$+$" and *multiplication* "$\cdot$" are defined. F is a *filed* under these two operations, addition and multiplication, if the following axioms are satisfied:

1. F is a commutative group under addition "$+$". The identity element with respect to addition is called the *zero element* of F and is denoted by 0.

2. The set $F\backslash\{0\}$ of non-zero elements of F forms a commutative group under multiplication "$\cdot$". The identity element with respect to multiplication is called the *unit element* of F and is denoted by 1.
3. For any three elements a, b, and c in F,

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

i.e., multiplication is distributive over addition (*distributive law*).

The definition suggests that a field consists of two groups, one with an addition operation and the other with a multiplication operation. They are called the *additive group* and *multiplicative group* respectively. For any element a in F, additive inverse is denoted by $-a$ and multiplicative inverse by a^{-1}, provided $a \neq 0$. For the sake of convenience, two additional operations can be defined using inverse elements. The *Subtraction* operation $(a - b)$ can be defined as adding the additive inverse, the $-b$ to a, i.e.,

$$a - b \equiv a + (-b)$$

Division operation $(a \div b)$ is defined as multiplying a by the multiplicative inverse, b^{-1}, of b, i.e.,

$$a \div b \equiv a \cdot (b^{-1})$$

It is clear that $a - a = 0$ and $a \div a = 1$, resulted in $a \neq 0$. From this overview of the field definition, it follows that a field is simply an algebraic system in which we can perform addition, subtraction, multiplication and division without leaving the field.

One example of a field is the field of rational numbers $\mathbb{Q}$, consisting of numbers that can be written as fractions in the form of a/b, where a and b are integers, and $b \neq 0$. Field $\mathbb{Q}$ has an infinite number of elements. Its additive inverse is $-a/b$, and multiplicative inverse is b/a, when $a \neq 0$. Another example is a field of complex numbers, denoted by $\mathbb{Q}$ and a field of real numbers $\mathbb{R}$.

Table 2-1 shows important axioms for both additive and multiplicative groups of the field.

Finite Fields

Since the error-correction codes in this thesis are based on finite fields, the following section gives an overview of the class of finite fields constructed from prime numbers and their important algebraic properties. Finite fields are also referred to as *Galois Fields* by mathematician Évariste Galois, who introduced the concept of finite fields.

Definition 2.8 The number of elements in a field is called the *order of the field*. A field with a finite order is called a *finite field*.

Axiom	**Addition**	**Multiplication**
Associativity	$(a+b)+c = a+(b+c)$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$
Commutativity	$a+b = b+a$	$a \cdot b = b \cdot a$
Distributivity	$a \cdot (b+c) = a \cdot b + a \cdot c$	$(a+b) \cdot c = a \cdot c + b \cdot c$
Identity	$a+0 = a = 0+a$	$a \cdot 1 = a = 1 \cdot a$
Inverse	$a+(-a) = 0 = (-a)+a$	$a \cdot a^{-1} = 1 = a^{-1} \cdot a,\ a \neq 0$

Table 2-1: Summary of field axioms

Definition 2.9 Let F be a field and 1 be its multiplicative identity (also referred to as a *unity element*). The *characteristic* of F is defined as the smallest positive integer λ such that

$$\sum_{i=1}^{\lambda} 1 = \underbrace{1+1+\cdots+1}_{\lambda \text{ summands}} = 0 \tag{2.1}$$

where summation represents repeated application of addition + operator of the field. If no such integer exists, F is said to have a *zero characteristic*, i.e., $\lambda = 0$, and F is an infinite field.

Theorem 2.3 The characteristic λ of a finite filed is a prime.

Proof. Suppose that λ is not a prime, and that l and k are factors of λ, $1 < l$ and $k < \lambda$. From the distributive law, it follows that

$$\sum_{i=1}^{\lambda} 1 = \left(\sum_{i=1}^{l} 1\right)\left(\sum_{i=1}^{k} 1\right) = 0 \tag{2.2}$$

To satisfy this equation, either $\left(\sum_{i=1}^{l} 1\right) = 0$ or $\left(\sum_{i=1}^{k} 1\right) = 0$. Since $1 < l$ and $k < \lambda$, this is a contradiction. Therefore, λ must be a prime. □

In other words, Theorem 2.3 states that for every prime p, there exists a field GF(p). For any positive integer m, a Galois field GF(p^m) can be constructed using a root of an *irreducible polynomial* with coefficients from GF(p). The GF(p^m) has p^m elements and contains GF(p) as a subfield.

Definition 2.10 Let a be a non-zero element of a finite field GF(p^m). The smallest positive integer is n such that $a^n = 1$ is called the *order* of the non-zero field element a.

The rest of the Theorems are given without proofs. They are crucial for construction of finite fields GF(p^m). More detailed theoretical background can be found in [111].

Theorem 2.4 Let a be a non-zero element of the order n in a finite field $GF(p^m)$. The powers of a,

$$a^n = 1,\ a,\ a^2, .., \ a^{n-1}$$

form a cyclic subgroup of the multiplicative group of $GF(p^m)$.

Theorem 2.5 Let a be a non-zero element of a finite field $GF(p^m)$. Then $a^{p^m-1} = 1$.

Theorem 2.6 Let n be the order of a non-zero element a in $GF(p^m)$. Then n divides $p^m - 1$.

Definition 2.11 A non-zero element a of a finite field $GF(p^m)$ is called a *primitive element* if its order is $p^m - 1$. That is, $a^{p^m-1} = 1, a, a^2, ..., a^{p^m-2}$ form all the non-zero elements of $GF(p^m)$.

An important construct in coding theory are polynomials with coefficients from a finite field $GF(p^m)$. A polynomial with one variable X over $GF(p^m)$ has the following form:

$$a(X) = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$$

where n is a non-negative integer and the coefficients a_i, $0 \leqslant i \leqslant n$, are elements of $GF(p^m)$. The *degree* of a polynomial, denoted as $\deg(a(X))$, is the largest power of X with a non-zero coefficient. A polynomial is called *monic* if the coefficient of the highest power of X is 1. A polynomial with all zero coefficients is called a *zero polynomial* and is denoted by 0.

Addition and multiplication of polynomials over $GF(p^m)$ is carried out in the usual way. Let

$$a(X) = a_0 + a_1X + \cdots + a_nX^n \text{ , and } b(X) = b_0 + b_1X + \cdots + b_mX^m$$

be two polynomials over $GF(p^m)$ with degrees n and m, respectively. Assuming that $m \leqslant n$, adding $a(X)$ and $b(X)$ we get

$$\begin{aligned} a(X) + b(X) = {} & (a_0 + b_0) + (a_1 + b_1)X + \cdots + \\ & (a_m + b_m)X^m + a_{m+1}X^{m+1} + \cdots + \\ & a_nX^n \end{aligned}$$

where $a_i + b_i$ is carried out with the addition of $GF(p^m)$. Multiplication of $a(X)$ and $b(X)$ is defined as:

$$a(X) \cdot b(X) = c_0 + c_1X + \cdots + c_{n+m}X^{n+m}$$

where c_k is defined as

$$c_k = \sum_{\substack{i+j=k, \\ 0 \leqslant i \leqslant n, 0 \leqslant j \leqslant m}} a_i \cdot b_j$$

where $a_i \cdot b_j$ is carried out with the multiplication of GF(p^m).

When $a(X)$ is divided by $b(X)$, provided $b(X) \neq 0$, we obtain a *unique pair* of polynomials over GF(p^m), $q(X)$ (*quotient*) and $r(X)$ (*remainder*), such that

$$a(X) = q(X) \cdot b(X) + r(X)$$

where $0 \leqslant \deg(r(X)) < \deg(b(X))$. This formulation is known as *Euclid's division.* If $r(X) = 0$, $a(X)$ is said to be divisible by $b(X)$, or $b(X)$ divides $a(X)$.

Definition 2.12 A polynomial $p(X)$ of degree n over GF(q) is said to be *irreducible* over GF(q) if it is not divisible by any polynomial over GF(q) that has a degree less than n and is greater than 0.

In other words, a polynomial $p(X)$ is said to be irreducible if it is not the product of two polynomials of positive degree. A polynomial of positive degree that is not irreducible is called *reducible* polynomial.

Theorem 2.7 Any irreducible polynomial $p(X)$ over GF(q) of degree m divides $X^{q^m-1}-1$.

Definition 2.13 A monic irreducible polynomial $p(X)$ of degree m over GF(q) is said to be a *primitive polynomial* if the smallest positive integer n for witch $p(X)$ divides $X^n - 1$ is $n = q^m - 1$.

Vector Spaces over Finite Fields

Vector spaces defined over finite fields are very important in the algebraic system for definition of error-correcting codes. A vector space consists of a field F, a commutative group V and a multiplication operation defined between elements of F and elements of V. Elements of V are called *vectors*, and elements of F *scalars.* The following definitions are important for defining linear-algebraic error-correcting codes.

Definition 2.14 (Vector space) Let F be a field. Let V be a set of of elements on which an addition operation "+" is defined. A multiplication operation "·" is defined between the elements of F and the elements of V. The set V is called a *vector space* over the field F if it satisfies the following axioms:

1. V is a commutative group under addition + defined on V.
2. For any element a from F and any element $\boldsymbol{v}$ from V, $a \cdot \boldsymbol{v}$ is an element of V.
3. For any elements a and b from F and any element $\boldsymbol{v}$ from V, associative law is satisfied:
$$(a \cdot b) \cdot \boldsymbol{v} = a \cdot (b \cdot \boldsymbol{v})$$

4. For any element a from F and any element $\boldsymbol{v}$ and $\boldsymbol{u}$ from V, the distributive law of vector sums is satisfied:
$$a \cdot (\boldsymbol{v} + \boldsymbol{u}) = a \cdot \boldsymbol{v} + a \cdot \boldsymbol{u}$$
5. For any elements a and b from F and any element $\boldsymbol{v}$ from V, the distributive law of scalar sums is satisfied:
$$(a + b) \cdot \boldsymbol{v} = a \cdot \boldsymbol{v} + b \cdot \boldsymbol{v}$$
6. Let 1 be the unit element of F. Then, for any element $\boldsymbol{v}$ from V:
$$1 \cdot \boldsymbol{v} = \boldsymbol{v}$$

Definition 2.15 (Vector subspace) Let S be a non-empty subset of a vector space V over a field F. S is called a *subspace* of V if it satisfies all the vector space axioms given by Definition 2.14.

Definition 2.16 (Linear independence) A set of vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$ in a vector space V over a field F is said to be *linearly independent* over F if and only if, for all k scalars a_i from F, the relation

$$a_1 \cdot \boldsymbol{v}_1 + a_2 \cdot \boldsymbol{v}_2 + \cdots + a_k \cdot \boldsymbol{v}_k = 0$$

implies that

$$a_1 = a_2 = \cdots = a_k = 0$$

Vectors that are not linearly independent are said to be *linearly dependent*.

Definition 2.17 (Linear combination) Let $\boldsymbol{u}$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$ be vectors in a vector space V over a field F. Vector u is said to be *linearly dependant* on $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$ if it can be expressed as a linear combination of $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$ as:

$$\boldsymbol{u} = a_1 \cdot \boldsymbol{v}_1 + a_2 \cdot \boldsymbol{v}_2 + \cdots + a_k \cdot \boldsymbol{v}_k$$

where a_i are scalars from F, $1 \leqslant i \leqslant k$.

Definition 2.18 (Spanning vector set) A set of vectors is said to *span* a vector space V over a field F if every vector in the vector space V is equal to a linear combination of the vectors in the set.

Set B of linearly independent vectors that spans a vector space V is called a *basis*, or *base*, of V. The cardinality of set B is called the *dimension* of the vector space V. In case the basis of the vector space has a finite number n of linearly independent vectors, this is expressed as $\dim(V) = n$, otherwise $\dim(V) = \infty$. Let V be a vector space with a dimension of n. For $0 \leqslant k \leqslant n$, a set of k linearly independent vectors in V spans a k-*dimensional subspace* of V. The k-dimensional vector space over a finite field GF(q) has q^k elements.

Definition 2.19 (Inner product) Let $\mathbf{u} = (u_0, u_1, ..., u_k)$ and $\mathbf{v} = (v_0, v_1, ..., v_k)$ be vectors in a vector space V over a field F. The *vector product* $\mathbf{u} \cdot \mathbf{v}$ is defined as:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=0}^{k-1} u_i \cdot v_i$$

Construction of Galois Fields

To construct an extension field $GF(p^m)$, we begin with the prime field $GF(p)$ and a primitive polynomial of degree m over $GF(p)$,

$$p(X) = p_0 + p_1 X + \cdots + p_{m-1} X^{m-1} + X^m$$

Let α be a *root* of $p(X)$. Since $p(X)$ is irreducible over $GF(p)$, α must be an element of a larger field that contains $GF(p)$ as a *subfield*. Since α is a root of $p(X)$,

$$p(\alpha) = p_0 + p_1 \alpha + \cdots + p_{m-1} \alpha^{m-1} + \alpha^m = 0$$

As $p(X)$ divides $X^{p^m-1} - 1$ (Theorem 2.7), the following holds:

$$X^{p^m-1} - 1 = q(X) \cdot p(X)$$

After replacing X by α in the last equation, we obtain:

$$\alpha^{p^m-1} - 1 = q(\alpha) \cdot p(\alpha) = q(\alpha) \cdot 0 = 0$$

which means that α is also a root of $X^{p^m-1} - 1$. Finally, we have the following identity:

$$\alpha^{p^m-1} = 1$$

Let $\mathcal{F}$ be a set, $\mathcal{F} = \{0, 1, \alpha, ..., \alpha^{p^m-2}\}$. We define a multiplication "$\cdot$" operation on elements of $\mathcal{F}$ (0 and 1 are zero and unit elements) as follows:

$$\begin{aligned}
&0 \cdot 0 = 0, \\
&0 \cdot 1 = 1 \cdot 0 = 0, \\
&0 \cdot \alpha = \alpha \cdot 0 = 0, \\
\\
&1 \cdot 1 = 1, \\
&1 \cdot \alpha = \alpha \cdot 1 = \alpha,
\end{aligned}$$

$$\alpha^2 = \alpha \cdot \alpha,$$
$$\vdots$$
$$\alpha^j = \underbrace{\alpha \cdot \alpha \cdot \cdots \cdot \alpha}_{\text{j multiplicands}}$$

From the multiplication definition above it also follows that

$$0 \cdot \alpha^j = \alpha^j \cdot 0 = 0,$$
$$1 \cdot \alpha^j = \alpha^j \cdot 1 = \alpha^j,$$
$$\alpha^i \cdot \alpha^j = \alpha^j \cdot \alpha^i = \alpha^{i+j}$$

Next, the addition operation "+" is defined using a polynomial representation of the elements, as follows. For any α^i and α^j,

$$\begin{aligned}\alpha^i + \alpha^j &= (a_{i,0} + a_{i,1}\alpha + \cdots + a_{i,m-1}\alpha^{m-1}) + (a_{j,0} + a_{j,1}\alpha + \cdots + a_{j,m-1}\alpha^{m-1}) \\ &= (a_{i,0} + a_{j,0}) + (a_{i,1} + a_{j,1})\alpha + \cdots + (a_{i,m-1} + a_{j,m-1})\alpha^{m-1}\end{aligned}$$

where $a_{i,l} + a_{j,l}$ is carried out over the prime field GF(p). Hence the polynomial given by the last identity is a polynomial of α over GF(p), and thus represents an element of GF(p^m). It can be proven that the set $\mathcal{F}$ forms a field with p^m elements, denoted by GF(p^m). The characteristic of GF(p) is p and since GF(p) is a subfield of GF(p^m), the characteristic of GF(p^m) is also p.

Elements of GF(p^m) can be represented also in vector form. Let

$$a_{i,0} + a_{i,1}\alpha + \cdots + a_{i,m-1}\alpha^{m-1}$$

be the polynomial representation of element α^i. The same element in vector form is represented by m-tuple:

$$(a_{i,0}, a_{i,1}, ..., a_{i,m-1})$$

where the m components are coefficients of the polynomial representation of α^i. In the vector representation, to add two elements, we simply add their corresponding vector representations component-wise:

$$\begin{aligned}&(a_{i,0}, a_{i,1}, ..., a_{i,m-1}) + (a_{j,0}, a_{j,1}, ..., a_{j,m-1}) \\ &\qquad = (a_{i,0} + a_{j,0}, a_{i,1} + a_{j,1}, ..., a_{i,m-1} + a_{j,m-1})\end{aligned}$$

where addition $a_{i,l} + a_{j,l}$ is carried out over the ground field GF(p).

To demonstrate construction of an extended field GF(p^m), the example of GF(2^5) is used. First, we define the addition and multiplication operations in the ground field

+	**0**	**1**
0	0	1
1	1	0

Table 2-2: Modulo-2 addition

·	**0**	**1**
0	0	0
1	0	1

Table 2-3: Modulo-2 multiplication

GF(2) which has two elements $\{0, 1\}$. Modulo-2 addition and multiplication operations are defined in Tables 2-2 and 2-3.

The additive inverse of 0 and 1 in GF(2) are themselves. This implies that $1-1 = 1+1$, hence, over GF(2), addition and subtraction are the same. The multiplicative inverse of 1 is itself. The common name for GF(2) is *binary field*, and it is the simplest finite field. GF(2) is very important for implementation of error-correction codes because it uses the same binary alphabet as a digital computer. Furthermore, from Table 2-2 it is clear that the module-2 addition operation on GF(2) is defined exactly the same as the binary *exclusive or* (XOR) operation. The modulo-2 operation, shown in Table 2-3 follows the definition of the binary *and* (AND) operation. For this reason, many codes choose to use GF(2) as the base field.

The next step in constructing a GF(2^5) is the selection of a primitive polynomial $p(X)$ of degree 5 over GF(2). Let $p(X) = 1 + X^3 + X^5$. It follows that $p(\alpha) = 1 + \alpha^3 + \alpha^5 = 0$, which means that $\alpha^5 = 1 + \alpha^3$. Additionally, $\alpha^{2^5-1} = \alpha^{31} = 1$, which is used in multiplication. Table 2-4 shows elements of GF(2^5) in power, polynomial and vector representations.

Power representation	Polynomial representation	Vector representation
0	0	(00000)
1	1	(10000)
α	α	(01000)
α^2	α^2	(00100)
α^3	α^3	(00010)
α^4	α^4	(00001)
α^5	$1+\alpha^3$	(10010)
α^6	$\alpha+\alpha^4$	(01001)
α^7	$1+\alpha^2+\alpha^3$	(10110)
α^8	$\alpha+\alpha^3+\alpha^4$	(01011)
α^9	$1+\alpha^2+\alpha^3+\alpha^4$	(10111)
α^{10}	$1+\alpha+\alpha^4$	(11001)
α^{11}	$1+\alpha+\alpha^2+\alpha^3$	(11110)
α^{12}	$\alpha+\alpha^2+\alpha^3+\alpha^4$	(01111)
α^{13}	$1+\alpha^2+\alpha^4$	(10101)
α^{14}	$1+\alpha$	(11000)
α^{15}	$\alpha+\alpha^2$	(01100)
α^{16}	$\alpha^2+\alpha^3$	(00110)
α^{17}	$\alpha^3+\alpha^4$	(00011)
α^{18}	$1+\alpha^3+\alpha^4$	(10011)
α^{19}	$1+\alpha+\alpha^3+\alpha^4$	(11011)
α^{20}	$1+\alpha+\alpha^2+\alpha^3+\alpha^4$	(11111)
α^{21}	$1+\alpha+\alpha^2+\alpha^4$	(11101)
α^{22}	$1+\alpha+\alpha^2$	(11100)
α^{23}	$\alpha+\alpha^2+\alpha^3$	(01110)
α^{24}	$\alpha^2+\alpha^3+\alpha^4$	(00111)
α^{25}	$1+\alpha^4$	(10001)
α^{26}	$1+\alpha+\alpha^3$	(11010)
α^{27}	$\alpha+\alpha^2+\alpha^4$	(01101)
α^{28}	$1+\alpha^2$	(10100)
α^{29}	$\alpha+\alpha^3$	(01010)
α^{30}	$\alpha^2+\alpha^4$	(00101)
$\alpha^{31}=1$		

Table 2-4: Extended field $GF(2^5)$ generated using primitive polynomial $p(X)=1+X^3+X^5$ over GF(2)

2.2 Linear Block Codes

Two structurally different types of codes have been widely used in communication and storage systems. These are known as block and convolutional codes. The difference between the two is the result of encoding principle. Block codes process information bits in chunks and produce the redundancy bits. Examples of block codes are Reed-Solomon codes, Hamming codes [43] and Walsh-Hadamard codes [5]. Convolutional codes process the information bits to produce codeword sequentially. They are usually coupled with block codes, e.g. Reed-Solomon codes. Since errors can appear in bursts, data is usually interleaved before convolutional encoding, so that error bursts can be repaired by an outer block code. This approach is first described in [26] and is known as *concatenated code*. A diagram of such code scheme is shown in the Figure 2-2.

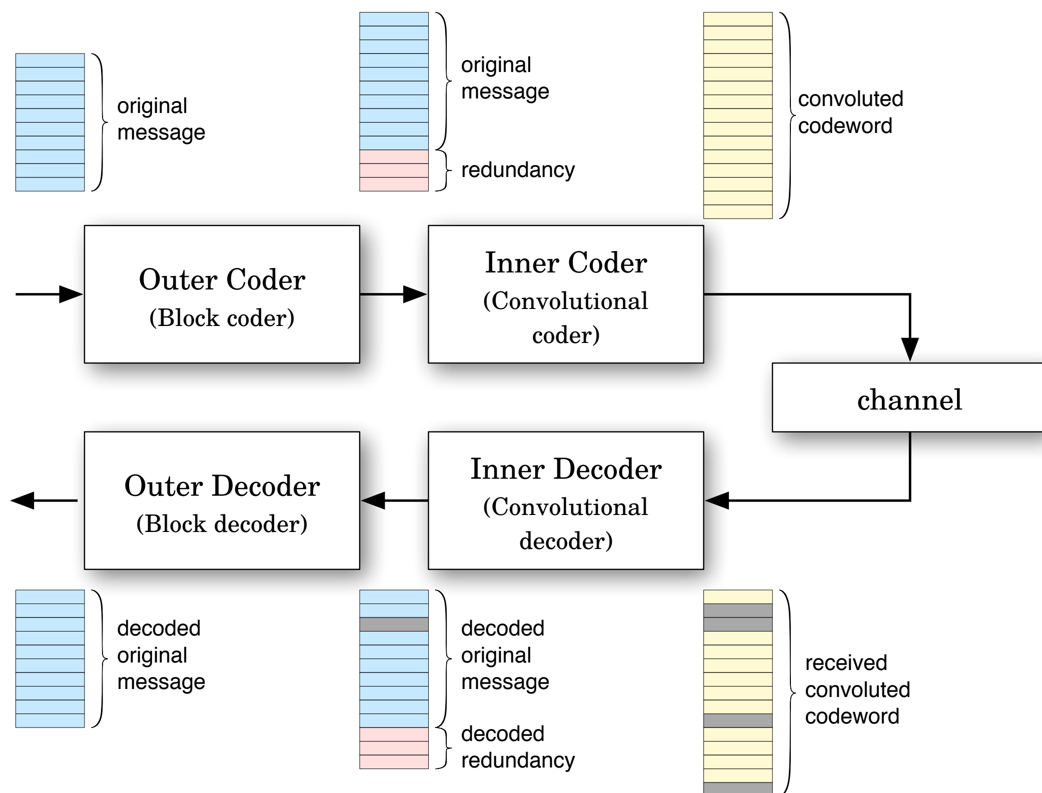


Figure 2-2: Block diagram of concatenated code

The original message is first encoded with an outer block code, which produces additional redundancy data. Output of outer code is then encoded with an inner convolutional code. The resulting codeword is then transmitted over a noise-prone channel, which adds errors in different size bursts, which are shown in grey. The inner convolutional decoder is able to correct short error bursts, while the long error bursts are corrected by the block decoder.

For the remainder of this chapter, linear block codes will be discussed in detail. The block codes are more suitable when a channel has the form of a physical medium for data storage, as opposed to transmission mediums. Another important characteristic of block codes is that they can be used for creating *erasure* coding schemes. Erasure is an error whose location in the codeword is known in advance. In the context of storage this can be a failed storage device or loss of connection of an attached network storage device.

2.2.1 Introduction to block codes

Input of linear block code is a continued sequence of binary symbols over GF(2). The binary block code then segments this binary symbol sequence into message blocks. Each block has a fixed length of k binary bits. This means there are 2^k distinct message blocks. Each input message of a block code encoder can thus be represented as a vector $\mathbf{d} = (d_0, d_1, ..., d_{k-1})$ over the binary field GF(2). This input message is encoded by the block code encoder into a longer sequence $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ of n bits, where $n > k$. The longer sequence $\mathbf{c}$ is called the *codeword* of the input message $\mathbf{d}$. Since each distinct input message is encoded to unique codewords, there are 2^k unique codewords, one for each unique input message. This set of 2^k codewords forms an (n, k) block code. The additional $n - k$ bits, added by the block code encoder, are called *redundant bits*, or *redundancy*. The redundancy, by definition, contains no new information. Its only purpose is to enable the detection and correction of transmission errors, caused by a noisy channel. Generally, the code rate R of a block code is $R = k/n$, which represents the amount of information bits carried by a single codeword bit.

One important aspect of practical usability of a block code is the definition of mapping between input message and corresponding codeword. Unless the code cannot be defined using some mathematical structures, the block code encoder and decoder would have to store a complete table of 2^k mappings between message and codewords. Therefore, codes with specific structural properties are used for practical implementations. An especially desirable property of block codes is *linearity*. Block codes with this property are called *linear* block codes. Linear block codes can be defined using formulations from linear algebra.

Definition 2.20 (Linear block code) A block code of length n over GF(2) with 2^k codewords is called an (n, k) *linear block code* if and only if its 2^k codewords form a k-dimensional subspace of the vector space of all the n-tuples over GF(2).

From Definition 2.20 it follows that there exists k *linearly independent codewords*, $\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{k-1}$, in a binary (n, k) block code $\mathcal{C}$, such that every codeword $\mathbf{v}$ in $\mathcal{C}$ is a *linear combination* of these k linearly independent codewords. Thus, the encoding procedure of a binary (n, k) linear block code can be defined as follows. Let $\mathbf{d} = (d_0, d_1, ..., d_{k-1})$ be the message to be encoded. The codeword for this message, $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$, is

given by the following linear combination of $\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{k-1}$, with k message bits of $\mathbf{d}$ as the coefficients:

$$\mathbf{c} = d_0\mathbf{g}_0 + d_1\mathbf{g}_1 + \cdots + d_{k-1}\mathbf{g}_{k-1} \tag{2.3}$$

2.2.2 Generator matrix

To create a matrix representation of the equation (2.3), the k linearly independent codewords, $\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{k-1}$, of the $\mathcal{C}$ can be arranged as rows of a $k \times n$ matrix over GF(2) as follows:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{2.4}$$

The codeword $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ for message $\mathbf{d} = (d_0, d_1, ..., d_{k-1})$, using equation (2.4) can be expressed as matrix product of vector $\mathbf{d}$ and matrix $\mathbf{G}$:

$$\mathbf{c} = \mathbf{d} \cdot \mathbf{G}$$

$$\mathbf{c} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{k-1} \end{bmatrix} \cdot \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{2.5}$$

The codeword $\mathbf{c}$ for message $\mathbf{d}$ is thus a linear combination of the rows of matrix $\mathbf{G}$ and the message bits of $\mathbf{d}$ as the coefficients. Matrix $\mathbf{G}$ is called a *generator matrix* of the (n, k) linear block code $\mathcal{C}$. A generator matrix of a given (n, k) linear block is not unique. Any choice of linearly independent codewords $\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{k-1}$ of $\mathcal{C}$ gives a generator matrix of $\mathcal{C}$. The rank of a generator matrix of a linear block code $\mathcal{C}$ is equal to the dimension of $\mathcal{C}$.

The *check* matrix $\mathbf{H}$ of a block code $\mathcal{C}$ is given by the following expression:

$$\mathbf{G} \cdot \mathbf{H}^T = \mathbf{O} \tag{2.6}$$

where $\mathbf{O}$ is a $k \times (n-k)$ zero matrix. This implies that the $\mathcal{C}$ is completely specified by an $\mathbf{H}$ matrix as:

$$\mathcal{C} = \{\mathbf{v} \in V \,:\, \mathbf{v} \cdot \mathbf{H}^T = \mathbf{0}\} \tag{2.7}$$

Usually, encoding of a linear block code is based on a generator matrix of the code, using equation (2.5), and decoding is based on a check matrix of the code.

A desirable property of linear block codes is to have a codeword with the format shown in Figure 2-3. The codeword is divided into two parts, the message part and the redundancy part. The message part contains k original information digits, and the redundancy part consists of $n-k$ parity digits. A linear block code which produces a codeword with this structure is called a *linear systematic code*.

A linear systematic code (n, k) block code is specified by a $k \times n$ generator matrix of the form:

$$\mathbf{G} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \left[\begin{array}{cccc} a_{0,0} & a_{0,1} & \cdots & a_{0,n-k-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k-1,0} & a_{k-1,1} & \cdots & a_{k-1,n-k-1} \end{array}\right.\underbrace{}_{\mathbf{A}\text{ matrix}} \left.\begin{array}{cccc} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{array}\right]\underbrace{}_{\mathbf{I}_k\text{ matrix}} \tag{2.8}$$

The generator matrix $\mathbf{G}$ of a linear systematic (n, k) block code consists of two submatrices. The left side forms a $k \times (n-k)$ submatrix $\mathbf{A}$ with elements over GF(2). The right side of the matrix $\mathbf{G}$ is a $k \times k$ identity matrix, $\mathbf{I}_k$. This relation can be written as $\mathbf{G} = [\mathbf{A} \quad \mathbf{I}_k]$. Identity matrix $\mathbf{I}_k$ in equation (2.8) guarantees that the rightmost k bits of codeword $\mathbf{c}$ are identical to k the message information bits of $\mathbf{d}$. The leftmost

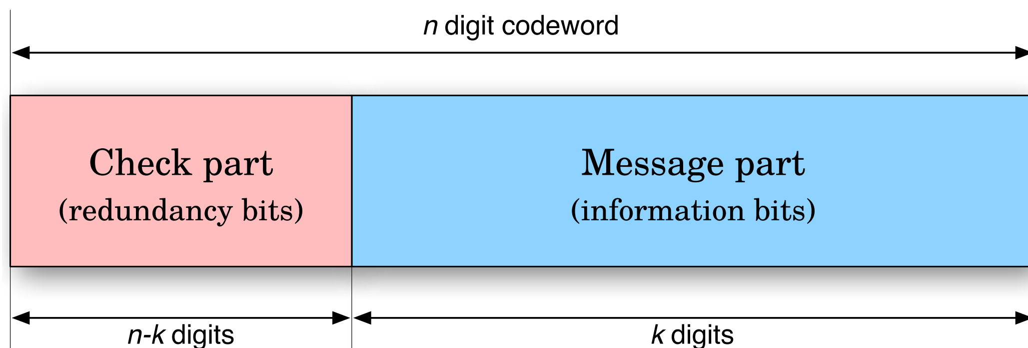


Figure 2-3: Systematic format of a codeword

$n-k$ codeword bits of $\mathbf{c}$ are a linear combination of information bits. These $n-k$ bits are commonly called *parity-check bits* or simply *parity bits*. It is apparent from equation (2.8), that the parity bits are completely defined by the $n-k$ columns of the $\mathbf{A}$ submatrix of the generator matrix $\mathbf{G}$. Evaluating the codeword $\mathbf{c}$ from equation (2.8) we get equations for calculating each bit of the codeword:

$$c_i = \begin{cases} d_0 a_{0,i} + d_1 a_{1,i} + \cdots + d_{k-1} a_{k-1,i} & \text{for } i = 0, 1, \ldots, n-k-1 \\ d_{i-(n-k)} & \text{for } i = n-k, \ldots, n-1 \end{cases} \tag{2.9}$$

The first $n-k$ equations from (2.9) generate parity bits and are thus called *parity equations* of the code $\mathcal{C}$. The $\mathbf{A}$ submatrix of $\mathbf{G}$ is called the *parity submatrix* of the generator matrix $\mathbf{G}$. This form of generator matrix, as given in (2.8), is called a *systematic* form. If a generator matrix $\mathbf{G}$ of an (n, k) linear block code $\mathcal{C}$ is given in the systematic form of (2.8), then the check matrix $\mathbf{H}$ of $\mathcal{C}$ is given as:

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}_{n-k} & \mathbf{A}^T \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & \cdots & 0 & a_{0,0} & a_{1,0} & \cdots & a_{k-1,0} \\ 0 & 1 & \cdots & 0 & a_{0,1} & a_{1,1} & \cdots & a_{k-1,1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a_{0,n-k-1} & a_{1,n-k-1} & \cdots & a_{k-1,n-k-1} \end{bmatrix} \tag{2.10}$$

2.2.3 Bounds of linear block codes

Definition 2.21 (Hamming weight) Let $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ be an n-tuple over GF(2). The *Hamming weight* of $\mathbf{V}$, denoted $\omega(\mathbf{v})$, is defined as the number of nonzero elements in $\mathbf{v}$.

The smallest weight of non-zero codeword in $\mathcal{C}$, denoted by $\omega_{\min}(\mathcal{C})$, is called the *minimum weight* of $\mathcal{C}$. The minimum weight of binary block code $\mathcal{C}$ is given by:

$$\omega_{\min}(\mathcal{C}) = \min\{\omega(\mathbf{u}) \,:\, \mathbf{u} \in \mathcal{C},\ \mathbf{u} \neq \mathbf{0}\} \tag{2.11}$$

Definition 2.22 (Hamming distance) Let $\mathbf{u} = (u_0, u_1, \ldots, u_{n-1})$ and $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ be two n-tuples over GF(2). The *Hamming distance* between $\mathbf{u}$ and $\mathbf{v}$, denoted as $d(\mathbf{u}, \mathbf{v})$, is defined as the number of elements where vectors $\mathbf{u}$ and $\mathbf{v}$ are different. Hamming distance function $d(\mathbf{u}, \mathbf{v})$ satisfies the triangle inequality. Let $\mathbf{w} = (w_0, w_1, \ldots, w_{n-1})$ be an additional n-tuple over GF(2). Then:

$$d(\mathbf{u}, \mathbf{v}) + d(\mathbf{v}, \mathbf{w}) \geqslant d(\mathbf{u}, \mathbf{w}) \tag{2.12}$$

Definition 2.23 (Distance of linear block code) The minimum Hamming distance of an (n, k) linear block code $\mathcal{C}$, denoted by $d_{\min}(\mathcal{C})$, is defined as the *smallest* Hamming distance between two codewords from $\mathcal{C}$, expressed as:

$$d_{\min}(\mathcal{C}) = \min\{d(\mathbf{u}, \mathbf{v}) \ : \ \mathbf{u}, \mathbf{v} \in \mathcal{C}, \ \mathbf{u} \neq \mathbf{v}\} \tag{2.13}$$

The Hamming distance between two n-tuples, $\mathbf{u}$ and $\mathbf{v}$, is equal to the Hamming weight of the vector sum of $\mathbf{u}$ and $\mathbf{v}$, i.e., $d(\mathbf{u}, \mathbf{v}) = \omega(\mathbf{u} + \mathbf{v})$. Using this fact, minimum distance $d_{\min}(\mathcal{C})$ of $\mathcal{C}$ can be written as:

$$\begin{aligned} d_{\min}(\mathcal{C}) &= \min\{d(\mathbf{u}, \mathbf{v}) \ : \ \mathbf{u}, \mathbf{v} \in \mathcal{C}, \ \mathbf{u} \neq \mathbf{v}\} \\ &= \min\{\omega(\mathbf{u} + \mathbf{v}) \ : \ \mathbf{u}, \mathbf{v} \in \mathcal{C}, \ \mathbf{u} \neq \mathbf{v}\} \\ &= \min\{\omega(\mathbf{x}) \ : \ \mathbf{x} \in \mathcal{C}, \ \mathbf{x} \neq \mathbf{0}\} \\ &= \omega_{\min}(\mathcal{C}) \end{aligned} \tag{2.14}$$

Definition 2.24 (Singleton bound) For any linear (n, k) block code $\mathcal{C}$, its distance $d_{\min}$ is bounded by

$$d_{\min} - 1 \leqslant n - k \tag{2.15}$$

A linear (n, k) block code $\mathcal{C}$ with a distance of $d_{\min}(\mathcal{C}) = n - k + 1$ is called the *maximum distance separable* (or MDS) code.

A block code $\mathcal{C}$ with minimum distance $d_{\min}$ is able to correct all error patterns of t or fewer bits, where

$$t = \left\lfloor \frac{(d_{\min} - 1)}{2} \right\rfloor$$

The parameter t is called the *random-error correcting capability* of a block code. A code with random-error correcting capability of t must have at least a distance given by

$$d_{\min} \geqslant 2t + 1$$

The error where the location of the error in codeword is known is called *erasure*. Let e be the number of errors and ϵ the number of erasures in a codeword. The distance of linear block code that is able to correct them is given by

$$d_{\min} \geqslant 2e + \epsilon + 1$$

2.2.4 Cyclic block codes

Cyclic codes are a special type of linear block codes. The cyclic property of these codes makes them easy to implement using simple shift registers with a feedback connection. Many cyclic codes have been constructed. Important properties of cyclic codes are listed below.

Definition 2.25 (Cyclic shift) Let $\mathbf{u} = (u_0, u_1, \ldots, u_{n-1})$ be an n-tuple over GF(2). If every component of $\mathbf{u}$ is shifted cyclically one place to the right, the following n-tuple is obtained:

$$\mathbf{u}^{(1)} = (u_{n-1}, u_0, \ldots, u_{n-2}) \tag{2.16}$$

which is called the *right cyclic-shift*, or simply the *cyclic-shift* of $\mathbf{u}$.

Definition 2.26 (Cyclic code) An (n, k) linear block code $\mathcal{C}$ is said to be *cyclic code* if the cyclic-shift of each codeword of $\mathcal{C}$ is also a codeword in $\mathcal{C}$.

An (n, k) cyclic code $\mathcal{C}$ consists of 2^k polynomials, called the *code polynomials*. There exists a unique non-zero code polynomial $\mathbf{g}(X)$ of degree $n - k$ of the following form:

$$\mathbf{g}(X) = 1 + g_1X + g_2X^2 + \cdots + g_{n-k-1}X^{n-k-1} + X^{n-k} \tag{2.17}$$

Every code polynomial in $\mathcal{C}$ is divisible by $\mathbf{g}(X)$, i.e., every code polynomial of $\mathcal{C}$ is a multiple of $\mathbf{g}(X)$. Therefore, to fully specify an (n, k) cyclic code $\mathcal{C}$, only the unique polynomial of the form given by (2.17) is needed. This polynomial is called the *generator polynomial* of the (n, k) cyclic code $\mathcal{C}$. The degree of $\mathbf{g}(X)$ determines the number of parity bits of such code.

Let $\mathbf{m}_i(X) = X^i$, $0 \leqslant i < k$, be the polynomial representing the message to be encoded (message polynomial). Dividing $X^{n-k}\mathbf{m}_i(X) = X^{n-k+i}$ by $\mathbf{g}(X)$, we get

$$X^{n-k+i} = \mathbf{a}_i(X)\mathbf{g}(X) + b_i(X) \tag{2.18}$$

where the remainder $b_i(X)$ has the following form:

$$\mathbf{b}_i(X) = b_{i,0} + b_{i,1}X + \cdots + b_{i,n-k-1}X^{n-k-1} \tag{2.19}$$

Polynomial $\mathbf{b}_i + X^{n-k+i}$ is divisible by $\mathbf{g}(X)$, which means that it is a code polynomial in $\mathcal{C}$. Arranging the n-tuple representation of the $n - k$ code polynomials, $\mathbf{b}_i + X^{n-k+i}$, $0 \leqslant i < k$, as the rows of $k \times n$ matrix over GF(2), the generator matrix of the (n, k) cyclic code $\mathcal{C}$ in systematic form is obtained:

$$\mathbf{G}_{\mathcal{C},\text{sys}} = \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-k-1} & 1 & 0 & \cdots & 0 \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-k-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{k-1,0} & b_{k-1,1} & \cdots & b_{k-1,n-k-1} & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.20}$$

2.3 Reed-Solomon Code

Reed-Solomon code is a linear block code with code symbols from GF(q), where q is a power of prime. Such codes are also called q-ary block codes, or block codes over GF(q). A message for a q-ary (n, k) block code consists of k information symbols from GF(q). A q-ary (n, k) block code has length n and contains q^k codewords.

Definition 2.27 (q-ary linear block code) A q-ary (n, k) block code over GF(q) of length n with q^k codewords is called a q-*ary* (n, k) *linear block code* if and only if its q^k codewords form a k-dimensional subspace of the vector space of all the q^n n-tuples over GF(q).

The version of q-ary codes most widely used are classed of Reed-Solomon codes [95].

Definition 2.28 (Reed-Solomon Code - *RS code*) Let α be a primitive polynomial of GF(q). For a positive integer t such that $2t < q$, the generator polynomial of a t-symbol-error-correcting RS code over GF(q) of length $q - 1$ is given by

$$\begin{aligned} g(X) &= (X - \alpha)(X - \alpha^2) \cdots (X - \alpha^{2t}) \\ &= g_0 + g_1 X + \cdots + g_{2t-1} X^{2t-1} + X^{2t} \end{aligned} \tag{2.21}$$

where $g_i \in$ GF(q).

Since $\alpha, \alpha^2, \ldots, \alpha^{2t}$ are roots of $X^{q-1} - 1$, generator polynomial $g(X)$ creates a cyclic RS code of length $q - 1$ with $2t$ parity symbols. The distance of RS code is determined by the weight of the generator polynomial $g(X)$. There are $2t + 1$ non-zero terms in $g(X)$, which means that the minimum distance of RS code is equal to $2t + 1$, which is 1 greater than the number of parity symbols. This puts the RS codes in class of *maximum-distance-separable* (MDS) codes. Table 2-5 summarises important characteristics of the Reed-Solomon code.

Code property	**Value**
Length	$q - 1$
Number of parity symbols	$2t$
Dimension	$q - 2t - 1$
Minimum distance	$2t + 1$

Table 2-5: Summary of properties of the (n, k) Reed-Solomon code over GF(q)

2.3.1 Vandermonde Reed-Solomon code

For use in data transmission over erasure channels a specialized form of Reed-Solomon codes are devised. Erasure channel is a communication channel that either transmits an information bit (or symbol) to the receiver or notifies the receiver that information is lost. The encoding procedure assumes that n codewords of (n, k) RS code are sent over the erasure channel. This class of RS codes uses Vandermonde matrix as the starting generator matrix for the code.

Definition 2.29 (Vandermonde matrix over GF(q)) Let $(a_0, a_1, \ldots, a_{n-1})$ be a n-tuple of elements from GF(q). *The Vandermonde matrix* is defined as:

$$\mathbf{V}_n = \begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^{n-1} \\ 1 & a_1 & a_1^2 & \cdots & a_1^{n-1} \\ 1 & a_2 & a_2^2 & \cdots & a_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{n-1} & a_{n-1}^2 & \cdots & a_{n-1}^{n-1} \end{bmatrix} \tag{2.22}$$

The direct formula for calculating the determinant of a square $n \times n$ Vandermonde matrix is given by [73]:

$$\det(V_n) = \prod_{0 \leqslant i \leqslant j \leqslant n} (a_i - a_j) \tag{2.23}$$

The inverse, $\mathbf{V}^{-1} = [b]_n$, of a square $n \times n$ Vandermonde matrix is specified as:

$$b_{i,j} = \frac{\displaystyle\sum_{\substack{0 \leqslant m_0 < \ldots < m_{n-i} \leqslant n \\ m_0, \ldots, m_{n-i} \neq j}} (-1)^{i-1} a_{m_0} \cdots a_{m_{n-i}}}{a_j \displaystyle\prod_{\substack{0 \leqslant m \leqslant n \\ m \neq j}} (a_m - a_j)} \tag{2.24}$$

From Equation 2.23 and Equation 2.24 it follows that the square Vandermonde matrix is invertible only if all elements a_i of constructing n-tuple are distinct. This property is exploited during the construction of the generator matrix for a (n, k) linear block code.

The $n \times k$ Vandermonde matrix $\mathbf{A}$ is constructed using an n-tuple of distinct elements $(0, 1, \ldots, n-1)$ over GF(q):

$$\mathbf{A}_{n \times k} = \begin{bmatrix} 0^0 & 0^1 & 0^2 & \cdots & 0^{k-1} \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{k-1} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (n-1)^0 & (n-1)^1 & (n-1)^2 & \cdots & (n-1)^{k-1} \end{bmatrix} \tag{2.25}$$

The Reed-Solomon (n, k) code is defined as:

$$\mathbf{c} = \mathbf{A} \cdot \mathbf{d} \tag{2.26}$$

where vector $\mathbf{c}$ represents the codeword symbols, vector $\mathbf{d}$ the information symbols, and matrix $\mathbf{A}$ is the generator matrix of the code. Removing exactly $n-k$ rows from matrix $\mathbf{A}$, we get a square $k \times k$ Vandermonde matrix. The resulting matrix is defined with k-tuple of distinct elements and is thus guaranteed to be non-singular. The generator matrix $\mathbf{A}$ has a non-systematic form, which makes the RS code defined by it not suitable for erasure coding. Matrix $\mathbf{A}$ can be transformed into a desirable systematic form applying a sequence of elementary matrix transformations until the upper part of the resulting matrix is an $k \times k$ identity matrix [88], as follows:

1. Column $\mathbf{C}_i$ and column $\mathbf{C}_j$ can be swapped, $i \neq j,\ i, j < k$,
2. Column $\mathbf{C}_i$ can be multiplied with scalar c, $c \neq 0,\ i < k$,
3. Column $\mathbf{C}_i$ can be replaced by $\mathbf{C}_i + c \cdot \mathbf{C}_j$, $c \neq 0,\ i \neq j,\ i, j < k$.

The derived matrix $\mathbf{A}^*_{sys}$ retains all properties of starting matrix $\mathbf{A}$, and is the generator matrix of systematic (n, k) Reed-Solomon code:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a^*_{k,0} & a^*_{k,1} & \cdots & a^*_{k,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a^*_{n-1,0} & a^*_{n-1,1} & \cdots & a^*_{n-1,k-1} \end{bmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{k-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{k-1} \\ r_0 \\ r_1 \\ \vdots \\ r_{n-k-1} \end{bmatrix}$$

$$\mathbf{c} = \mathbf{A}^*_{sys} \cdot \mathbf{d} = \begin{bmatrix} \mathbf{d} \\ \mathbf{r} \end{bmatrix} \tag{2.27}$$

where vector $\mathbf{r} = (r_0, r_1, \ldots, r_{n-k-1})$ is the parity part (redundancy) of systematic Reed-Solomon codeword.

Erasure decoding

In case of a symbol erasure in the message part of the codeword $\mathbf{c}$ of an (n, k) Vandermonde Reed-Solomon code, the corresponding rows of the generator matrix $\mathbf{A}^*$ and the codeword vector $\mathbf{c}$ are deleted. In case the number of erasures is lower than $n - k$, additional rows have to be deleted until the resulting square, $k \times k$, matrix $\tilde{\mathbf{A}}$, and vector $\tilde{\mathbf{c}}$ with

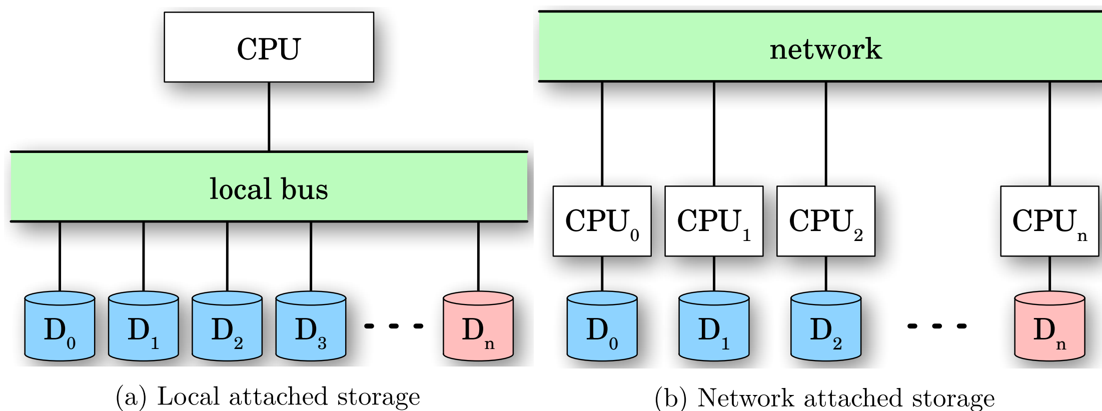
k elements, remain. The matrix $\tilde{\mathbf{A}}$ is guaranteed to be non-singular. By solving the resulting system for $\mathbf{d}$ we reconstruct the message part $\mathbf{d}$ of the codeword.

$$\begin{aligned} \tilde{\mathbf{A}} \cdot \mathbf{d} &= \tilde{\mathbf{c}} \\ \mathbf{d} &= \tilde{\mathbf{A}}^{-1} \cdot \tilde{\mathbf{A}} \cdot \mathbf{d} \\ \mathbf{d} &= \tilde{\mathbf{A}}^{-1} \cdot \tilde{\mathbf{c}} \end{aligned} \tag{2.28}$$

2.3.2 Distributed Reed-Solomon code

The encoding and decoding process of the systematic (n, k) Reed-Solomon code, described by Equation 2.27 and Equation 2.28 respectively, assumes that we have all codeword symbols locally. When the erasure code is used in a distributed storage environment, it is beneficial that the encoding/decoding scheme minimize network utilization. Figure 2-4 shows two common scenarios of erasure code usage in storage systems.

The system shown in Figure 2-4(a) is comprised of one processing element (CPU) that can encode and decode the RS code as well as the locally attached storage (Disks) that store the data. In this case, in order to perform encoding or decoding, the data has to be read, transferred to the CPU, and finally, the result has to be written back to the appropriate storage device. This is the typical organization of a RAID system. In case of the scenario described in Figure 2-4(b), each of the storage elements, together with a processing element, is connected to a network interconnect. This is a typical case of a distributed storage system. It is obvious that transferring all data blocks to a single processing element for encoding or decoding is inefficient. First, the network bandwidth is used for transferring data blocks to a node which will perform code calculation. After



(a) Local attached storage (b) Network attached storage

Figure 2-4: Local and distributed storage systems

the calculation is finished, results have to be transferred back to the appropriate storage device. Secondly, only one processing element is utilized in code calculation operation.

To better utilize resources and network bandwidth of distributed system the modified Vandermonde encoding scheme is used. The new scheme minimizes data movement by bringing the calculation to the processing elements where the data is stored. As a base for this scheme, systematic Vandermonde Reed-Solomon (n, k) code is used, as in Equation 2.27.

Each column of the code generator matrix $\mathbf{A}_{\text{sys}}$, denoted by $\mathbf{A}_i$, is multiplied by the corresponding element from the message data vector d_i, where $0 \leqslant i < k$, as follows:

$$\mathbf{c}_i = d_i \cdot \mathbf{A}_i = d_i \cdot \begin{bmatrix} a^*_{k,i} \\ a^*_{k+1,i} \\ \vdots \\ a^*_{n-1,i} \end{bmatrix} = \begin{bmatrix} d_i \cdot a^*_{k,i} \\ d_i \cdot a^*_{k+1,i} \\ \vdots \\ d_i \cdot a^*_{n-1,i} \end{bmatrix} \tag{2.29}$$

Since the vectors $\mathbf{c}_i$ are calculated at the local processing element, one that is closest to the data, no network bandwidth is utilized. By leaving out the upper identity matrix part of the $\mathbf{A}_{\text{sys}}$, the parity part of the codeword $\mathbf{r}$ is obtained by:

$$\mathbf{r} = \sum_{i=0}^{k} \mathbf{c}_i \tag{2.30}$$

Element of $\mathbf{c}_i$ vectors are transmitted to the processing element of the storage node that will store the corresponding parity block. Once k symbols is collected at the destination processing element, they are added together to form a resulting parity symbol r_j, $0 \leqslant j < n - k$. The example of the distributed Reed-Solomon coding scheme is shown in Figure 2-5.

The calculation of redundancy $\mathbf{r}$, Equation 2.30, can be carried out in $\log k$ steps if all data nodes participate in summation reduction. Thus, the amount of data transferred through the network during the encoding operation of (n, k) the Reed-Solomon code is

$$(n - k) \times S_B \times \log k \tag{2.31}$$

where S_B is the size of each data block. The time complexity of this method is discussed in [87]. Let R_+ be the rate of performing addition in GF(q), R_* be the rate of performing multiplication in the same field, and R_{NET} be the bandwidth of the network. The time complexity then can be expressed as follows:

$$(n - k) \times S_B \times \left(\frac{\log n}{R_*} + \frac{\log n}{R_+} + \frac{1}{R_{\text{NET}}} \right) \tag{2.32}$$

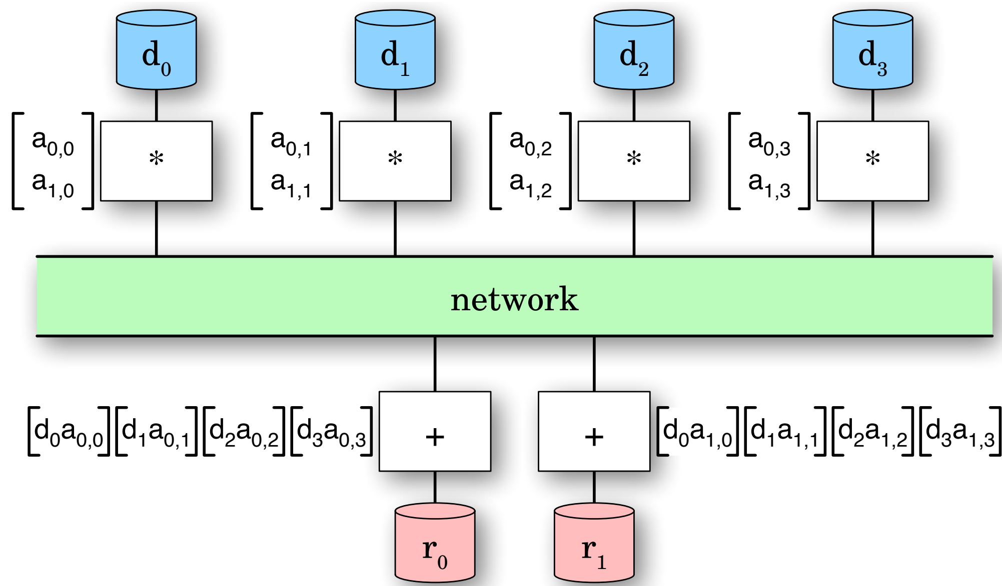


Figure 2-5: Example of distributed $(6, 4)$ Reed-Solomon code

2.4 Algebraic signatures

An erasure coding scheme, such as the Reed-Solomon erasure code, enables reconstruction of missing data or data that is known to be corrupted. However, a reliability coding scheme should also be able to detect and correct silent data corruption. This type of error, also known as *data rot*, is caused by the medium on which data is stored. Silent corruption cannot be detected by a pure erasure coding scheme. Modern enterprise grade hard disks claim an unrecoverable error rate of 1 bit per 10^{16} read bits. A study of data integrity from CERN [79] states the error rate to be 1 in 10^{7} read bits, but emphasizes different origins and patterns of errors. Their investigation of storage and the rest of the system, revealed a strong correlation between the errors of other components, the incompatibility issues between RAID controller for example, and the hard disk firmware. Moreover, the RAID controllers do not check for data corruption during data read. Instead, to check for data rot, RAID controllers implement a special *verify* command, which reads all data, compare the parity data, and correct the errors found. The verify command stresses the disks much more than a usual workload, which again can have a negative impact on read error rate and component lifetime. In a distributed environment an additional component of the system, such as network interconnect, can also introduce errors. Conclusion of the CERN study is that serious measures need to be taken in order to ensure end-to-end data reliability. One way to achieve this is by employing *checksums*.

The simplest form of checksum is the *parity bit*, or *check bit*. In this scheme, a single bit is added to the block of data to indicate whether the number of bits set to 1 is odd or even. Such a simple checksum scheme is very limited, and will detect

the error only if an odd number of bits have flipped. If an even number of bits in a message have changed state; the resulting parity bit will stay the same. To achieve better error-detection properties, more robust codes have been invented. Most widely used is the *cyclic redundancy check* (CRC) code, proposed by William Wesley Peterson in this 1961 paper [85]. The CRC code calculates a short *check* value for a block of data based on redundancy of a systematic cyclic code. Since their inception, CRC codes have been used in almost all areas of digital transmission and communication systems. In storage, CRC codes have been used in ANSI T10 standard [74] to provide end-to-end data reliability for reads and writes. It should be noted that checksums do not protect against intentional data modification, and thus should not be used for digital signature purposes.

Recently, a class of novel checksum functions called *Algebraic signatures* [66], based on the Karp-Rabin modular fingerprinting functions [93], have been introduced. An important property of algebraic signatures is their ability to guarantee the number of detectable changes in data based on the length of the signature.

Definition 2.30 (Algebraic signatures) Let $\mathbf{P} = (p_0, p_1, \ldots, p_{l-1})$ be a data vector of l symbols (elements) in GF(2^m), $l < 2^m - 1$. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ be a vector of n distinct, non-zero, elements in GF(2^m), called the n-symbol *signature base*, or simply the *base*. The n-*symbol P signature*, based on $\boldsymbol{\alpha}$, is a vector

$$\mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{P}) = \left(\mathrm{sig}_{\alpha_0}(\mathbf{P}),\ \mathrm{sig}_{\alpha_1}(\mathbf{P}),\ \ldots,\ \mathrm{sig}_{\alpha_{n-1}}(\mathbf{P})\right) \tag{2.33}$$

where each element of the vector is calculated as:

$$\mathrm{sig}_{\alpha_i}(\mathbf{P}) = \sum_{j=0}^{l-1} p_j \cdot \alpha_i^j \tag{2.34}$$

for $0 \leqslant i < n$.

Definition 2.31 (Signature collision) Let $\mathbf{P}$ and $\mathbf{Q}$ be two different data vectors of l symbols, and $\boldsymbol{\alpha}$ a signature base of the algebraic signature. *Signature collision* occurs when $\mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{P}) = \mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{Q})$.

When the signature base $\boldsymbol{\alpha}$ is derived using powers of a primitive element α from GF(2^m) as

$$\boldsymbol{\alpha} = (\alpha, \alpha^1, \alpha^2, \ldots, \alpha^{n-1}) \tag{2.35}$$

the probability of a signature collision with two pages of length l is 2^{-nm}, for $l < 2^m$. Table 2-6 gives an overview of collision probability for common combinations of the finite field and signature length selection. For small symbol sizes, e.g. symbols of GF(2^8), in order to keep collision probability sufficiently small, a larger number of concatenated signatures is needed.

Another significant advantage of algebraic signatures over other checksums is that they are based on the finite field arithmetic over GF(q). This allows for the interesting

	Collision probability of n-component signature Finite field GF(2^m), Message length l		
n	m=8 l=255 B	m=16 l=64 KiB	m=32 l=4 MiB
1	3.906×10^{-3}	1.526×10^{-5}	2.328×10^{-10}
2	1.526×10^{-5}	2.328×10^{-10}	5.421×10^{-20}
4	2.328×10^{-10}	5.421×10^{-20}	2.939×10^{-39}
8	5.421×10^{-20}	2.939×10^{-39}	8.636×10^{-78}

Table 2-6: Collision probability of n-component Algebraic signatures over GF(2^m)

possibility of coupling algebraic signatures with linear block erasure code. This property is presented in [101]. Algebraic signatures are a linear combination of a data symbol vector. As a consequence of linearity, the following equalities are satisfied:

$$\begin{aligned} \mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{X}+\mathbf{Y}) &= \mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{X}) + \mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{Y}) \\ \mathrm{sig}_{\boldsymbol{\alpha}}(c \cdot \mathbf{X}) &= c \cdot \mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{X}),\ c \in \mathrm{GF}(2^p) \end{aligned} \tag{2.36}$$

where addition and multiplication are operations over GF(2^m). Because of linearity properties (2.36) the following equation holds:

$$\mathcal{RS}\left(\mathrm{sig}_{\boldsymbol{\alpha}}(\mathbf{d})\right) = \mathrm{sig}_{\boldsymbol{\alpha}}\left(\mathcal{RS}(\mathbf{d})\right) \tag{2.37}$$

$$\mathcal{RS}\left(\mathrm{sig}_{\boldsymbol{\alpha}}(d_0), \mathrm{sig}_{\boldsymbol{\alpha}}(d_1), \ldots, \mathrm{sig}_{\boldsymbol{\alpha}}(d_{k-1})\right)\right) \quad = \quad \left(\mathrm{sig}_{\boldsymbol{\alpha}}(c_0), \mathrm{sig}_{\boldsymbol{\alpha}}(c_1), \ldots, \mathrm{sig}_{\boldsymbol{\alpha}}(c_{n-k-1})\right)$$

where vector $\mathbf{c}$ represents parity symbols calculated from the data vector $\mathbf{d}$ using systematic (n, k) Reed-Solomon erasure code is denoted as $\mathbf{c} = \mathcal{RS}(\mathbf{d})$.

The property Equation 2.37 is useful in the data verification process, especially in a distributed environment where each element of Reed-Solomon codeword is located on a separate server. Each of the storage nodes only has to calculate and transmit the signature of a stored codeword element. This speeds up the verification process significantly, as compared to the traditional approach where one node would have to collect all data blocks and perform full computation in order to verify consistency.

CHAPTER 3

On Implementing Reed-Solomon Codes

This chapter discusses implementation of the error correction codes and algebraic signatures used in the reliability middleware presented in this thesis. The implemented error correction codes are based on the Reed Solomon erasure code. This is a linear, block based, erasure coding scheme over a finite field. Described erasure codes use systematic Vandermonde formulation of a generator matrix. The generator matrices are dynamically crafted to provide desired erasure resiliency.

The main challenge in implementing fast and efficient Reed-Solomon error correction code is posed by the fact that all operations have to be carried out in a Galois Field. This problem has been previously addressed in [56] and [55]. Since the calculating of Reed-Solomon ECC can be represented by a matrix-vector multiplication, it is crucial that addition and multiplication operations in the Galois Field are implemented efficiently.

3.1 Galois Field multiplication

Multiplication of two elements of a binary finite field $GF(2^l)$ is defined as:

$$a \circ b = (a \times b) \mod p \tag{3.1}$$

The cross product "$\times$" is the carry-less multiplication of polynomial representations of elements in $GF(2^l)$. To obtain a result, polynomial reduction (modulo) operation is performed using a primitive field generator p of the Galois field. Since neither the carry-less multiplication nor the modulo operation is implemented in hardware, we need an efficient algorithm for both operations.

3.1.1 Carry-less multiplication

Carry-less multiplication is the operation of multiplying two elements without propagating carries. It is defined on elements in $GF(2^l)$ as follows:

Definition 3.1 (Carry-less multiplication) Let $a(X) = a_0 + a_1X + \cdots + a_{l-1}X^{l-1}$ and $b(X) = b_0 + b_1X + \cdots + b_{l-1}X^{l-1}$ be elements in $GF(2^l)$. Let $c(X) = c_0 + c_1 + \cdots + c_{2l-2}X^{2l-1}$ be a polynomial of degree $2l-1$ with coefficients in $GF(2)$. The *carry-less multiplication* is defined as follows:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} \text{ when } 0 \leqslant i \leqslant l-1 \tag{3.2}$$

$$c_i = \sum_{j=i-l+1}^{l-1} a_j b_{i-j} \text{ when } l \leqslant i \leqslant 2l-1 \tag{3.3}$$

where summation and multiplication are performed in the $GF(2)$.

Since the multiplication operation in $GF(2)$ equates to logical AND operation ($\bigodot$), and addition to logical XOR ($\bigotimes$), equations (3.2) and (3.3) can be written as:

$$c_i = \bigoplus_{j=0}^{i} a_j \odot b_{i-j} \text{ when } 0 \leqslant i \leqslant l-1 \tag{3.4}$$

$$c_i = \bigoplus_{j=i-l+1}^{l-1} a_j \odot b_{i-j} \text{ when } l \leqslant i \leqslant 2l-1 \tag{3.5}$$

Hereafter, the carry-less multiplication will be denoted by the "$\times$" symbol.

Algorithm 3-1 Carry-less multiplication of Galois Field elements. Inputs, a and b, are elements of GF(2^l). The largest possible length of a result is $2l$. Additional modulo operation is needed to obtain a proper GF(2^l) value.

```
1: function CARRY__LESS__MULTIPLICATION(a, b)
2:     r[2l − 1 : 0] ← 0                          ▷ Initialize accumulator
3:     for i ← 0; i < l; i ← i + 1 do
4:         if IsBitSet(a, i) then                 ▷ test i-th bit in a
5:             r ← r ⊕ LeftShift(b, i)
6:         end if
7:     end for
8:     return r
9: end function
```

Pseudo code that describes carry-less multiplication of two Galois field elements is described by Algorithm 3-1. Elements of GF(2^l) have at most l components in the vector representation.

Carry-less multiplication is somewhat similar to integer multiplication. In both operations, the second element is shifted as many times as the number of bits equal to 1 in the first elements. The difference is, that the carry-less multiplication uses carry-less addition (XOR operation), whereas the integer multiplication is using addition that generates and propagates carry.

3.1.2 Modulo operation

The result of the carry-less multiplication $c(X)$, described by equations (3.4) and (3.5), has at most $2l$ coefficients in GF(2), or $2l$ bits in the vector representation. In order to obtain a valid element of GF(2^l) after carry-less multiplication, modulo operation has to be performed using field generating polynomial $p(X)$. An efficient modulo reduction algorithm, based on the Barrett reduction algorithm [4] is described in [37]. From equation in polynomial representation:

$$\begin{aligned} a(X) \times b(X) =& c(X) \\ =& c_0 + c_1X + \cdots + c_{l-1}X^{l-1} + \\ & c_lX^l + c_{l+1}X^{l+1} + \cdots + c_{2l-1}X^{2l-1} \end{aligned} \tag{3.6}$$

it follows that:

$$\begin{aligned} a(X) \circ b(X) &= (a(X) \times b(X)) \mod p(X) \\ &= (c_0 + c_1X + \cdots + c_{l-1}X^{l-1}) \mod p(X) + \\ &\quad (c_lX^l + c_{l+1}X^{l+1} + \cdots + c_{2l-1}X^{2l-1}) \mod p(X) \\ &= c^*(X) + (c_lX^l + c_{l+1}X^{l+1} + \cdots + c_{2l-1}X^{2l-1}) \mod p(X) \\ &= c^*(X) + (c^\dagger(X) \times X^l) \mod p(X) \end{aligned} \tag{3.7}$$

where $c^*(X) = c(X) \mod X^l$, and $c^\dagger(X)$ is the quotient from the division of $c(X)$ with X^l. From equation (3.7) it follows that only modulo reduction of upper l elements is necessary, since the result can be obtained by XOR-ing of the lower part with the reduction of the upper part. Let $u(X)$ be the reduction of upper part of the carry-less multiplication result:

$$u(X) = (c^\dagger(X) \times X^l) \mod p(X) \tag{3.8}$$

Let the polynomial $q(X)$ be the quotient from the division of $c^\dagger(X) \times X^l$ with $p(X)$. Then, equation (3.8) can be expressed as:

$$c^\dagger(X) \times X^l = p(X) \times q(X) + u(X) \tag{3.9}$$

As the l least significant terms of the polynomial $c^\dagger(X) \times X^l$ are zero, that means that the l least significant terms of polynomials $p(X) \times q(X)$ and $u(X)$ must be equal (addition of the polynomial terms is performed in GF(2)). Using this fact, and the equation (3.9), we obtain following equality:

$$u(X) = p(X) \times q(X) \mod X^l \tag{3.10}$$

The module X^l is used to obtain only the l least significant terms, since the polynomial $u(X)$ is of degree $l - 1$.

Let $L^k(a)$ denote the coefficients of the k least significant terms of the polynomial a, and $M^k(a)$ denote k most significant terms of a. Now, the (3.10) can be expressed as:

$$u(X) = p(X) \times q(X) \mod X^l = L^l(p(X) \times q(X)) \tag{3.11}$$

Further, we define polynomial $p^\star(X)$ as the l least significant terms of $p(X)$ (primitive irreducible polynomial $p(X)$ of GF(2^l) is of degree l):

$$\begin{aligned} p(X) &= p^\star(X) + X^l \\ p^\star(X) &= p(X) + X^l \end{aligned} \tag{3.12}$$

Substituting the (3.12) into the equation (3.11) we have:

$$\begin{aligned} u(X) =& L^l(p^\star(X) \times q(X) + X^l \times q(X)) \\ =& L^l(p^\star(X) \times q(X)) + L^l(X^l \times q(X)) \end{aligned} \tag{3.13}$$

Since the l least significant terms of $X^l \times q(X)$ are zero, equation (3.13) is therefore:

$$u(X) = L^l(p^\star(X) \times q(X)) \tag{3.14}$$

From this result it follows that in order to obtain $u(X)$ we need to calculate quotient polynomial $q(X)$. This can be performed using the Barrett reduction algorithm. We start by multiplying the equation (3.9) by X^l.

$$c^\dagger(X) \times X^{2l} = p(X) \times q(X) \times X^l + u(X) \times X^l \tag{3.15}$$

Let $q^+(X)$ polynomial be the quotient of the division of X^{2l} with the polynomial $p(X)$:

$$X^{2l} = q^+(X) \times p(X) + r(X) \tag{3.16}$$

Substituting in (3.15) and applying the M^l function on the resulting equation we have:

$$\begin{aligned} M^l(c^\dagger(X) \times q^+(X) \times p(X)) + M^l(c^\dagger(X) \times r(X)) \\ = M^l(p(X) \times q(X) \times X^l) + M^l(u(X) \times X^l) \end{aligned} \tag{3.17}$$

First terms on both sides of the last equations are polynomials of degree $3l - 1$ while the second terms are of degree $2l - 1$. Thus, the equation (3.17) is not affected by polynomials $c^\dagger(X) \times r(X)$ and $u(X) \times X^l$:

$$M^l(c^\dagger(X) \times q^+(X) \times p(X)) = M^l(p(X) \times q(X) \times X^l) \tag{3.18}$$

Since we are only interested in the l most significant terms, we can introduce following substitution in the left side:

$$M^l(M^l(c^\dagger(X) \times q^+(X)) \times X^l \times p(X)) = M^l(p(X) \times q(X) \times X^l) \tag{3.19}$$

The equation is now satisfied when the quotient polynomial q is

$$q(X) = M^l(c^\dagger(X) \times q^+(X)) \tag{3.20}$$

Now, the polynomial $u(X)$ is defined [37] as:

$$u(X) = L^l(p^\star(X) \times M^l(c^\dagger(X) \times q^+(X))) \tag{3.21}$$

Modulo reduction algorithm, described by equations (3.21) and (3.7), can be summarised in the following steps:

Preprocessing Given the primitive irreducible polynomial $p(X)$ that generates $GF(2^l)$, we calculate polynomials $p^\star(X)$ and $q^+(X)$ as described by (3.12) and (3.16), respectively. The polynomial $p^\star(X)$ is at most a degree of $l - 1$ consisting of the least l significant coefficients of $p(X)$. The polynomial $q^+(X)$ is of degree l and equals the quotient of the division of X^{2l} with the polynomial $p(X)$.

Calculation of the reminder polynomial The algorithm consists of three steps:

Step 1 Most significant l bits of the input, $c^{\dagger}(X)$, are multiplied with $q^{+}(X)$, yielding a polynomial of degree $2l-1$, or at most $2l$ bits in a vector representation.

Step 2 The l most significant bits of the polynomial resulting from **Step 1** are multiplied with $p^{\star}(X)$. The result is a polynomial of degree $2l-2$.

Step 3 Result of the algorithm is the polynomial containing the l least significant coefficients of the polynomial resulting from **Step 2**.

The preprocessing step of the modulo reduction algorithm depends on the selection of irreducible polynomial $p(X)$ used to generate $GF(2^l)$. Algorithm 3-2 demonstrates modulo reduction algorithm for $GF(2^{128})$ generated with the polynomial $p(X) = 1 + X + X^2 + X^7 + X^{128}$. In this example, $l = 128$, $p^{\star}(X) = 1 + X + X^2 + X^7$, and $q^{+}(X) = p(X)$.

Algorithm 3-2 Modulo reduction operation for $GF(2^{128})$ generated with polynomial $p(X) = 1 + X + X^2 + X^7 + X^{128}$

```
1:  function REDUCTION_MODULO([X3 : X2 : X1 : X0])
2:      A ← RightShift(X3, 63)
3:      B ← RightShift(X3, 62)
4:      C ← RightShift(X3, 57)
5:
6:      D ← X2 ⊕ A ⊕ B ⊕ C
7:
8:      [E1 : E0] ← LeftShift([X3 : D], 1)
9:      [F1 : F0] ← LeftShift([X3 : D], 2)
10:     [G1 : G0] ← LeftShift([X3 : D], 7)
11:
12:     [H1 : H0] ← [X3 ⊕ E1 ⊕ F1 ⊕ G1 : D ⊕ E0 ⊕ F0 ⊕ G0]
13:
14:     [R1 : R0] ← [X1 ⊕ H1 : X0 ⊕ H0]
15:
16:     return [R1 : R0]
17: end function
```

The input operand of `reduction_modulo()` function is represented in vector form as a concatenation of four vectors X_i. Each of X_i vectors are 64 bit long, amounting to 256 bit in total, which is the expected length of the carry-less multiplication result of elements in $GF(2^{128})$.

3.2 Reed-Solomon Encoding

Because we use a systematic version of RS code, for data buffers D_i $(0 \leqslant i < k)$ we only produce the parity-check part of the codeword, denoted as C_j $(0 \leqslant j < m,\ m = n - k)$. S_B represents the size of the buffers in bytes, and it must contain a whole number of

symbols from $GF(2^l)$. In following, we will present implementation of Reed-Solomon codes with different sizes of symbols. For this reason, buffer size will be at least 512 B long.

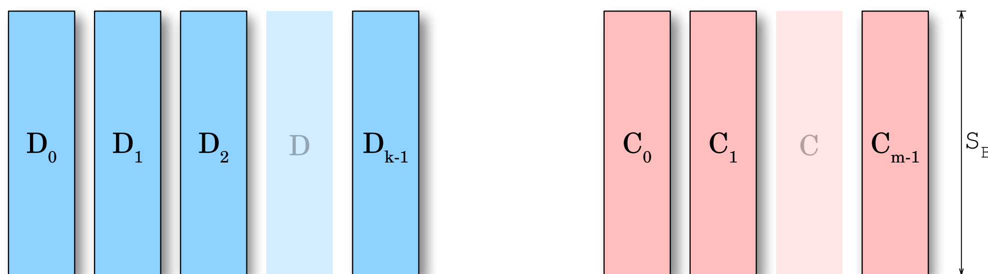


Figure 3-1: Buffer-based encoding and decoding with (n, k) Reed-Solomon code $(m = n - k)$

To implement an efficient algorithm for systematic (n, k) Reed-Solomon encoder we use Equation 2.27.

Let vector $\mathbf{c}$ of length m denote the parity-check (redundancy) part of the codeword, and $m \times k$ matrix $\mathbf{A}$ the lower part of the Vandermonde-based generator matrix of the code. Using the properties of $GF(2^l)$ multiplication given by Equation 3.1 we have:

$$\mathbf{A} \cdot \mathbf{d} = \mathbf{c} \tag{3.22}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,k-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,k-1} \end{bmatrix} \cdot \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{k-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} \tag{3.23}$$

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} \cdot d_0 & + & a_{0,1} \cdot d_1 & + & \cdots & + & a_{0,k-1} \cdot d_{k-1} \\ a_{1,0} \cdot d_0 & + & a_{1,1} \cdot d_1 & + & \cdots & + & a_{1,k-1} \cdot d_{k-1} \\ & & & \vdots & & & \\ a_{m-1,0} \cdot d_0 & + & a_{m-1,1} \cdot d_1 & + & \cdots & + & a_{m-1,k-1} \cdot d_{k-1} \end{bmatrix} \tag{3.24}$$

Operation "$\cdot$" is multiplication in $GF(2^l)$, so we can apply equation (3.1) on each of the $a_{i,j} \cdot d_j$ pairs as follows:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} (a_{0,0} \times d_0) & \bmod p & + & \cdots & + & (a_{0,k-1} \times d_{k-1}) & \bmod p \\ (a_{1,0} \times d_0) & \bmod p & + & \cdots & + & (a_{1,k-1} \times d_{k-1}) & \bmod p \\ & & & \vdots & & & \\ (a_{m-1,0} \times d_0) & \bmod p & + & \cdots & + & (a_{m-1,k-1} \times d_{k-1}) & \bmod p \end{bmatrix}$$

Finally, because the reduction modulo operation is distributive over addition in $GF(2^l)$, we obtain:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} (a_{0,0} \times d_0 & + & \cdots & + & a_{0,k-1} \times d_{k-1}) & \bmod p \\ (a_{1,0} \times d_0 & + & \cdots & + & a_{1,k-1} \times d_{k-1}) & \bmod p \\ & & \vdots & & & \\ (a_{m-1,0} \times d_0 & + & \cdots & + & a_{m-1,k-1} \times d_{k-1}) & \bmod p \end{bmatrix} \tag{3.25}$$

Equation 3.25 reduces the overhead of the vector-matrix multiplication operation. The reduction modulo operation is required only once per parity-check symbol, instead of k times. The only overhead is that intermediate results of carry-less operation "$\times$" now require $2l$ bits for storage, instead of l bits for elements of $GF(2^l)$. Since we will choose the value of l that fits in a register of a CPU, storing the result in two registers is not much overhead and can be easily mitigated. A flowchart diagram of the buffer block Reed-Solomon erasure encoding procedure is shown in Figure 3-2.

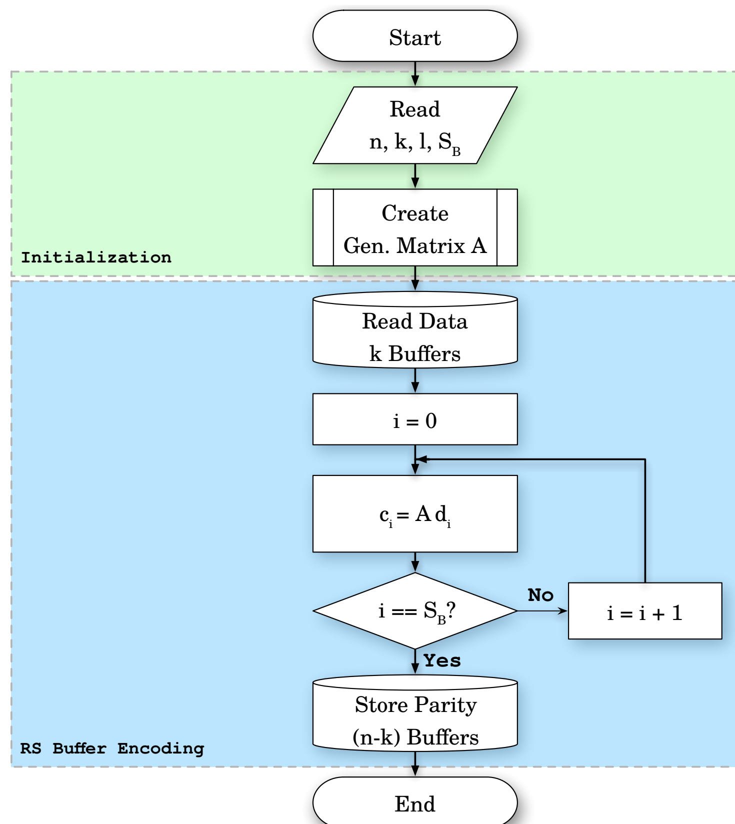


Figure 3-2: Flowchart of Reed-Solomon buffer-based encoding algorithm. Initialization step is performed once for desired configuration. The RS Encoding part can be repeated for multiple buffers by reusing the same RS generator matrix.

The algorithm is initialized by reading the parameters of the Reed-Solomon block, n, k, and l. In many applications, the block size is selected to match the underlying block size. For legacy hard drives, block size is usually $512\,\mathrm{B}$, whereas modern hard drives use the Advanced Format which extends block size to $4\,\mathrm{kB}$. If intended application of the Reed-Solomon code is in parallel file systems, usually the minimum block size is $1\,\mathrm{MB}$. The first step is the preprocessing step in which the code generator matrix is created according to selected parameters as described by Equation 3.25. Next, total of n buffers of required size (S_B) are allocated, and the data is read into the first k buffers. The code-generating phase is performed in the loop. Each loop iteration consumes at most l bits, where l corresponds to the bit-width of the native register size of the CPU (multiple of RS symbol size). Calculated parity symbols are stored in the dedicated m buffers at offset $il/8\,\mathrm{B}$. When the loop finishes, parity buffers can be stored at the dedicated parity location. Parity storage location is chosen in such way that a single erasure only affects one block, either the data block or the redundancy block.

In the following, several practical implementations of efficient, buffer-based, Reed-Solomon erasure codes are presented. Different application conditions allow for different kinds of optimizations. During this thesis the following implementation approaches have been explored:

Vectorization This approach exploits SIMD units of modern CPUs. By performing the described Galois Field operations on multiple elements in parallel, a significant speedup can be achieved. The main contribution of this thesis is in exploring the possible uses of supported SIMD operations on x86 and ARM platforms. A practical implementation of this work has contributed to the open source ZFS file system.

Just-in-Time compilation (JIT) The described methods for Reed-Solomon erasure codes computation contain intrinsically redundant operations that cannot be optimized with traditional coding methods. Here we use the JIT method for specifying, assembling and executing the code. We utilized LLVM [63] infrastructure which enables us to produce and run highly optimized and vectorized code for x86 and ARM platforms.

3.3 Summary

Reed-Solomon codes are ubiquitous in all areas of digital data storage and transmission. They are a class of non-binary cyclic linear error-correcting codes based on polynomials over finite fields. A systematic version of Reed-Solomon erasure codes is widely used for data reliability in a RAID-like storage solution. Data blocks are stored unmodified, avoiding read/write operations without computational overhead. Implementation of efficient Reed-Solomon codes hinges on ability to perform fast Galois field operations.

Addition operation is equivalent to *exclusive-or* (carry-less addition), and therefore can be efficiently performed. But, CPUs do not have Galois field multiplication realized in hardware, so this operation has to be emulated. A method that uses polynomial carry-less multiplication and reduction to obtain a multiplication result is presented. Galois field operations are then combined with a Vandermonde-based systematic Reed-Solomon generator matrix formulation to form the basis of codes discussed in the rest of the thesis. Lastly, a concept of buffer based erasure encoder, suitable for storage applications, is presented.

CHAPTER 4

Micro Benchmarks

In the previous chapter, we discussed requirements for the implementation of Reed-Solomon erasure codes. Practically, all necessary operations can be implemented using a limited set of arithmetic and logic operations. In this chapter we assess the relative performance of these operations through series of synthetic CPU micro-benchmarks. To implement a vectorized version of carry-less multiplication Algorithm 3-1 and polynomial module reduction Algorithm 3-2, only logical *SHIFT*, *exclusive or* (*XOR*), data *LOAD* and *STORE* operations are needed. In addition, the impact of CPU caching on memory throughput is investigated through a micro-benchmark designed to mimic erasure code requirements.

In the following chapter, we compare the performance of a server grade x86 CPU and a low-power ARM CPU, since the codes are constructed using a retargetable compiler. Both CPUs support vector instructions needed for vectorization of the erasure code. On a x86 platform, they are part of the SSE and AVX2 instruction set [51]. Similarly, on the ARM platform all required instructions are found in the NEON instruction set, introduced with ARMv6 architecture [14].

4.1 Arithmetic and logic operations

Figure 4-1 shows throughput of arithmetic and logic operations on our test-bed, Intel- and ARM-based, platforms[1]. The performed operations have no memory dependency, as they are performed only with one or two CPU registers. Throughput is a measure of processed data on a single CPU core. Obtained measurements show that vector instructions, when running on the same clock speed as scalar, are able to process multiples of scalar registers. The length of the scalar register of the Intel platform is 64 bit and the vector SSE register 128 bit, which corresponds to a twofold speedup for vector instructions. The ARM platform is a 32 bit ARMv7 CPU with NEON vector registers of 128 bit. Observed speedup is less than expected due to weaker instruction decoding and a scheduler unit of the low-powered CPU.

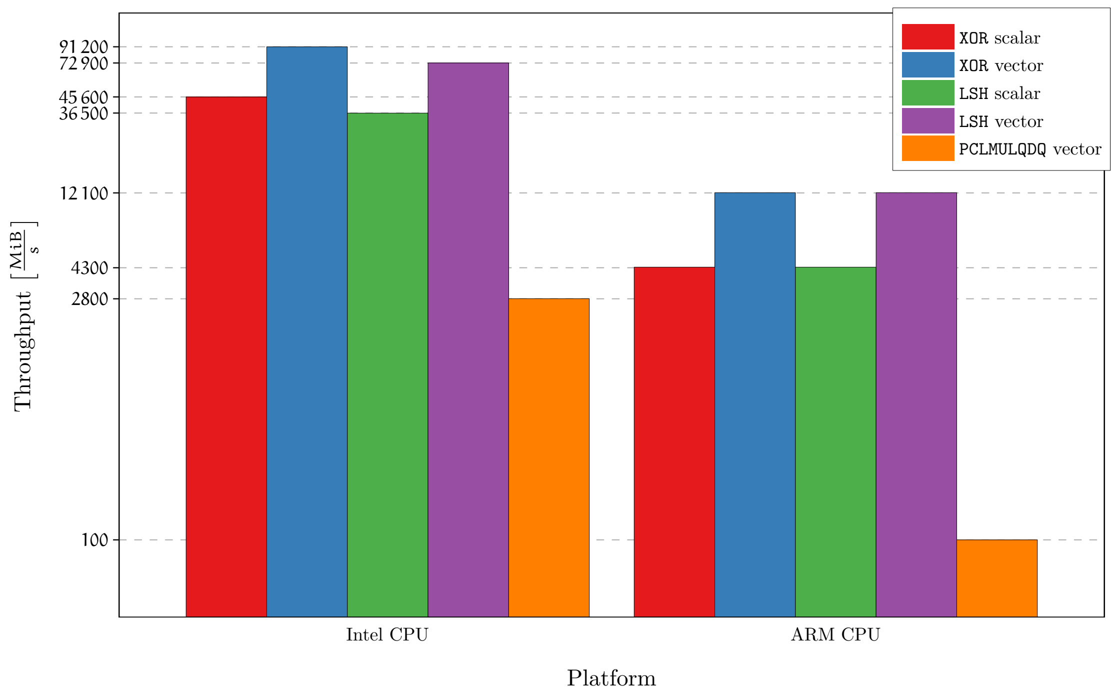


Figure 4-1: Throughput of arithmetic and logic operations used for Reed-Solomon implementation on Intel and arm systems

4.2 Memory load throughput

In order to efficiently fetch data from memory to CPU we first performed *load* micro-benchmarks. There are lot of factors that affect how fast data in RAM can be accessed, such as data alignment and allocation of memory in Non-Uniform Memory (NUMA)

[1] Detailed description of test-bed can be found in section A

systems. Also, several optimization techniques can be used to improve the throughput. We explored unrolling of the load instructions, as shown by Algorithm 4-1, and implicit data-prefetching techniques. With grouping of several load instructions of consecutive memory addresses we can consume the whole cache line, making sure all data elements of the cache line are used before the line is evicted from the cache. Since the data access pattern of the buffer based Reed-Solomon code is simple and known in advance, we can issue an explicit *prefetching* instruction to make sure the needed cache line is loaded into the L1 data cache just before it is needed.

Algorithm 4-1 Load benchmark, single and unrolled methods. $\mathbf{B}[]$ is the pre-allocated buffer. S_B is the size of the buffer, and S_V is the size of CPU vector registers.

```
 1: function LOAD_BENCHMARK_1X(B[], S_B, S_V)
 2:     for i ← 0; i < S_B; i ← i + S_V do
 3:         VECTOR_LOAD(B[i])
 4:     end for
 5: end function
 6:
 7: function LOAD_BENCHMARK_4X(B[], S_B, S_V)
 8:     for i ← 0; i < S_B; i ← i + 4 * S_V do
 9:         VECTOR_LOAD(B[i + 0 * S_V])        ▷ Consume whole cache line (64 B)
10:         VECTOR_LOAD(B[i + 1 * S_V])
11:         VECTOR_LOAD(B[i + 2 * S_V])
12:         VECTOR_LOAD(B[i + 3 * S_V])
13:     end for
14: end function
```

Figure 4-2 shows the results of *load* micro benchmarks for Intel and ARM platforms. The objective of this test is to measure memory load throughput from all CPU cache level as well as from the main memory. Tests were performed on a single CPU core. Results for our Intel platform showed good results for cached loads, up until the 25 MB mark which is the size of the L3 cache of the CPU. Since this is a NUMA system, we noted a significant difference in throughput when data was loaded from a remote NUMA node. Traditionally, desktop and server grade CPUs have good hardware prefetching logic that is able to recognize simple access patterns and preload needed data in a cache [71]. This eliminated the need for explicit prefetching instructions on this platform. To test this hypothesis, we performed the same measurement after disabling the automation prefetcher functionality. As predicted, results showed decreased throughput when reading from L3 and RAM.

However, obtaining optimal throughput on the ARM platform required manually employing several of the techniques described. Using just single vector load instruction to fetch data showed unexpectedly low throughput, around $400\,\mathrm{MiB\,s^{-1}}$. Unrolling the load instructions brought improvement only for L1 cache accesses, in fact doubling the throughput in the L1 cache region (blue line). The NEON pipeline of ARM CPU is located

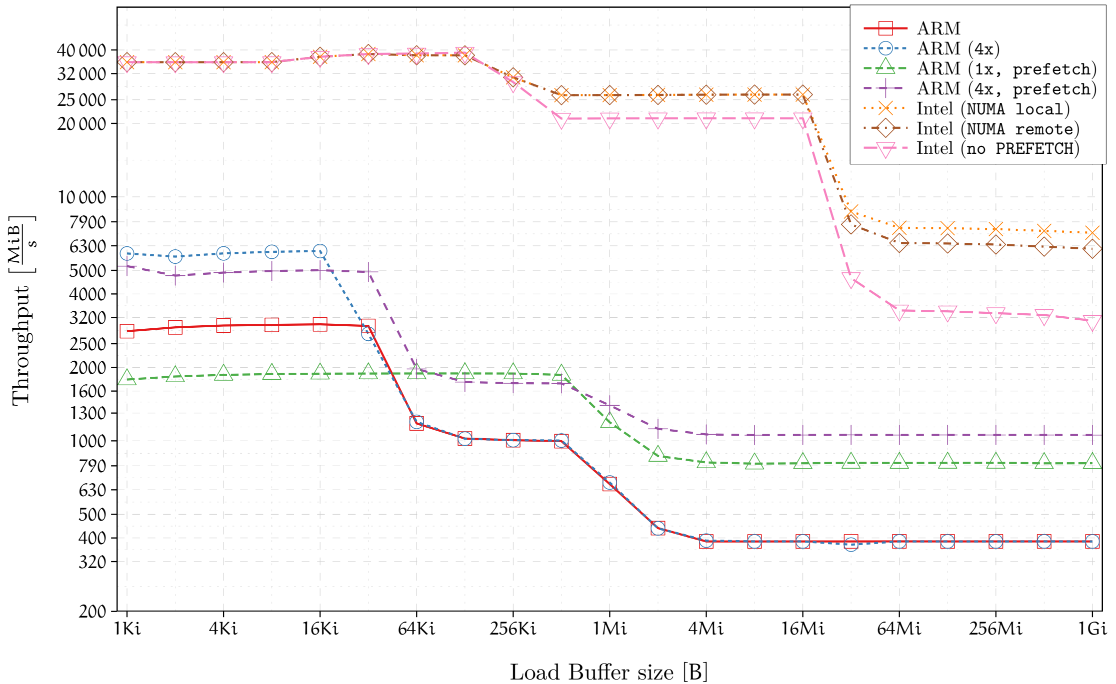


Figure 4-2: Throughput of the LOAD operation on Intel and ARM systems

behind the scalar integer pipeline, which introduces long stalls if vectorized and scalar code is mixed together. By unrolling load instructions, a group of load instructions is performed with less pipeline stalls, and thus the CPU is able to dispatch 2 load instructions per clock cycle. With the addition of explicit prefetch instruction[2], throughput from L2 and RAM significantly increased, while access to L1 showed worse performance. This indicated that CPU's hardware prefetching logic is not recognizing the access pattern correctly, and that no automatic prefetching has been performed. Upon more detailed investigation, we found that our specific CPU has a known errata [27] stating that the hardware prefetcher, if enabled, can corrupt data under some circumstances. As a consequence the hardware prefetcher has been disabled by the operating system, leading to the poor performance we observed. Finally, by combining the unrolled access and implicit prefetching we were able to obtain the best results on the ARM platform (purple line).

4.2.1 Multi stream memory throughput

The memory access pattern of the buffer-based block codes is known in advance. In each computation step, data is loaded from k data buffers, and after computing, the resulting parity symbols are stored in m code buffers. Since, in a typical erasure application the ECC k is several times larger than m, the algorithm will require much more `LOAD` than `STORE` bandwidth. An example of this benchmark is shown in Algorithm 4-2. The

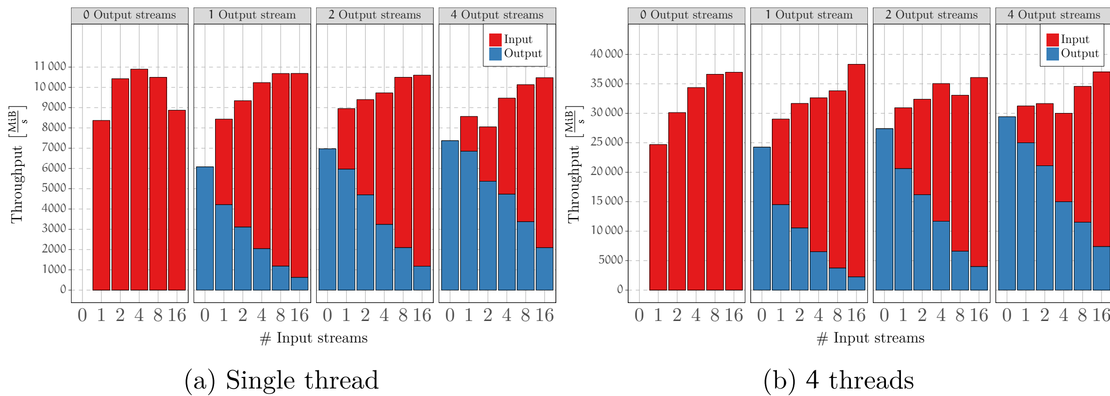
[2]PLD instruction, ARM NEON instruction set

Algorithm 4-2 Multi stream memory benchmark method with 4 input and 2 output streams. $\mathbf{B}_{\text{in}}[]$ and $\mathbf{B}_{\text{out}}[]$ are vectors of pointers to pre-allocated input and output buffers. S_B is the size of each buffer, and S_V is the size of CPU vector registers.

1: **function** STREAM_BENCHMARK_4IN_2OUT($\mathbf{B}_{\text{in}}[4][], \mathbf{B}_{\text{out}}[2][], S_B, S_V$)
2: **for** $i \leftarrow 0; i < S_B; i \leftarrow i + S_V$ **do**
3: VECTOR_LOAD($\mathbf{B}_{\text{in}}[0][i]$) ▷ Input stream 0
4: VECTOR_LOAD($\mathbf{B}_{\text{in}}[1][i]$) ▷ Input stream 1
5: VECTOR_LOAD($\mathbf{B}_{\text{in}}[2][i]$) ▷ Input stream 2
6: VECTOR_LOAD($\mathbf{B}_{\text{in}}[3][i]$) ▷ Input stream 3
7: VECTOR_STORE($\mathbf{B}_{\text{out}}[0][i]$) ▷ Output stream 0
8: VECTOR_STORE($\mathbf{B}_{\text{out}}[1][i]$) ▷ Output stream 1
9: **end for**
10: **end function**

algorithm shows a method used to benchmark a case with 4 input and 2 output stream buffers. This micro-benchmark measures only memory throughput, without any code (redundancy) computing overhead.

Figure 4-3 shows bi-directional bandwidth of a CPU memory bus on our Intel platform. From the Figure 4-3(a) it is evident that a single core cannot utilize full

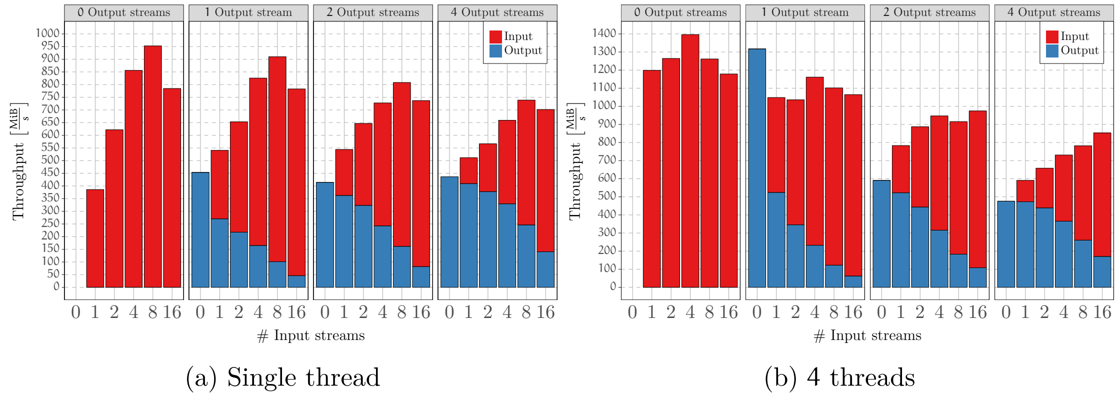


(a) Single thread (b) 4 threads

Figure 4-3: Streaming `LOAD/STORE` performance of Intel Haswell CPU

available bandwidth of the memory bus [112][20]. With the increase of the number of streams the decrease of load throughput occurs, while the bi-directional throughput remains at the same level. This is caused by exhaustion of *Line-fill Buffers* of the CPU code. Line-fill buffers are a limited physical resource used to track outstanding cache line transfers by the CPU core. Our test CPU has 10 line-fill buffers per core. If all line-fill buffers are used, all memory requests are sent directly to a unified L3 cache. Also, by accessing a multitude of memory addresses, hardware prefetcher is not providing full efficiency as with a simple linear access pattern. Throughput of multi-threaded tests is shown in Figure 4-3(b).

Figure 4-4 shows the result of the described test on the ARM platform. As expected, achieved throughput is much lower than on the Intel platform. In a single threaded test, Figure 4-4(a), the pipeline stall issue is noticeable once again. Good load performance is only achieved when issuing several load instructions at once, i.e. when using multiple streams. Also, a decrease in multi-threaded performance is noticeable with the increase of output streams, as shown in Figure 4-4(b).



(a) Single thread

(b) 4 threads

Figure 4-4: Streaming `LOAD/STORE` performance of ARM CPU

4.2.2 Memory load latency

CPU is able to hide RAM access latency with the help of automatic or manual data prefetching. To measure this effect, we utilized *lmbench3* [68] micro-benchmark suite. The micro-benchmark makes one single allocation of 256 MiB, so the entire data set does not fit into the CPU caches. Next, the test reads array elements from the end towards the beginning of the array, while creating data dependencies that cannot be resolved until the load has fully been performed. This serializes the access and prevents any data dependency optimization that the CPU might apply. Finally, the working set, *array size*, is varied so it fits into all cache levels, until it gets too big. By doing the serialized access and array size variation, it is guaranteed that memory access time is caused by maximum latency of the closest cache from which data is served. Micro-benchmark was repeated with changed access stride. This parameter controls the distance between accessed array members. By changing the stride, we investigate impact of prefetchers and TLB[3] misses on read latency. When a larger stride is used prefetchers are more likely to fail to get the data in time for when it is needed. Also, with a larger stride the benchmark performs less accesses per page and causing more TLB reloads which additionally decrease performance. Results of latency micro-benchmark for Intel Haswell CPU are shown in Figure 4-5.

The Figure shows the results of running `lat_mem_rd`[4] program from *lmbench3* [68] suite. Left y-axis shows latency in ns, while the CPU clock cycle count is shown on

[3]Translation Lookaside Buffer stores recent virtual to physical memory translations

[4]command: `lat_mem_rd 256m 64 128 256 512 1024`

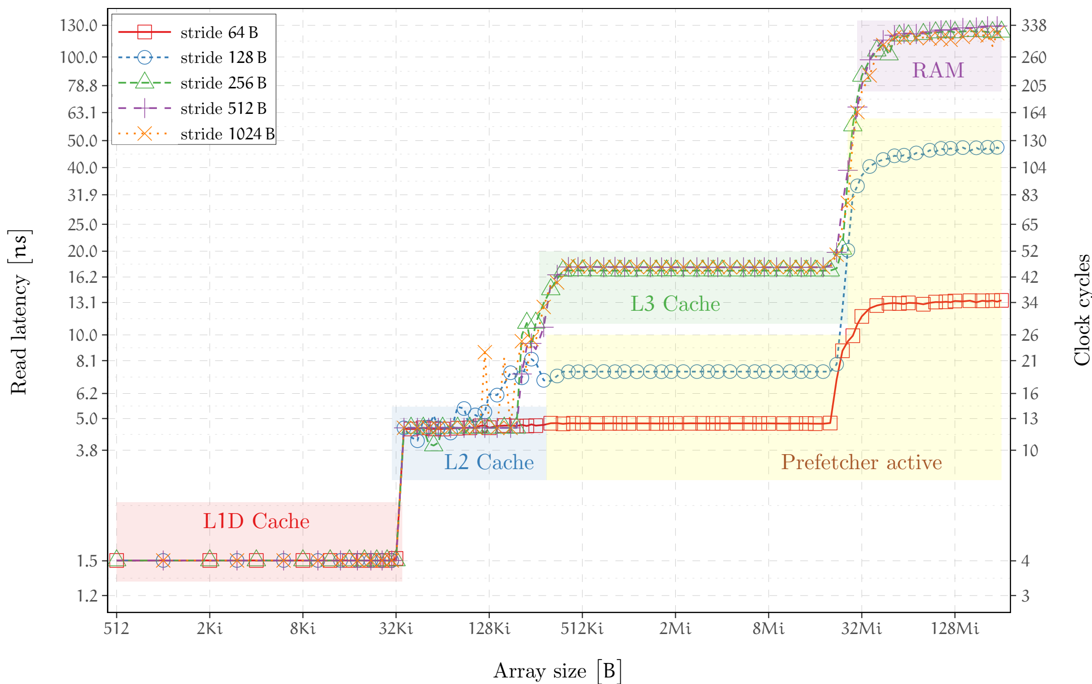


Figure 4-5: Read latency of Intel Haswell CPU with hardware prefetching

y-axis on the right. The graph shows several plateaus, each representative of a cache level. The L1 data cache is shown to have the fastest access latency, just 4 clock cycles. With an array size of 32 KiB, L2 cache shows minimal latency of 11 cycles. Variability in measured latencies around the 256 KiB array is likely due to the limited size of L1 TLB, which can only hold 64 entries for 4 KiB pages. The L2 cache TLB has 1024 entries shared for 4 KiB and 2 MiB pages, and its effects are not visible in the test. After the 288 KiB (L1 + L2 data cache) mark we expect to see L3 cache latency, which, for Haswell micro-architecture, is listed to be 34 clock cycles[5]. However, the graph shows 3 plateaus for different stride sizes. For the smallest stride size, 64 B, the read latency stays at L2 cache level, until the array size gets larger than L3 cache size. This is an indication that prefetchers are able to identify the access patterns of smaller strides correctly and preload required data. Haswell micro-architecture contains several hardware prefetchers [50], each dedicated to specific purpose:

The Data Cache Unit prefetcher (DCU) is responsible for loading the next line into the L1 data cache. It is triggered on ascending load memory accesses to recently accessed data.

The Spatial prefetcher monitor requests from L1 cache and prefetches another cache line to form a 128 B aligned chunk. Depending on which cache line is requested by the L1 cache, this prefetcher also requests the preceding or the following line.

[5] Actual latency will vary due to clock ratios between CPU core and uncore [50]

The Streamer prefetcher detects read requests for ascending and descending addresses. The streamer of Haswell architecture can detect up to 32 data load or store streams and prefetches next cache lines from memory. If there are not many outstanding memory requests per core, data is also prefetched to the L2 data cache. In case prefetched lines are far ahead, data is stored in the L3 cache only to avoid replacement of useful cache lines in L2.

Figure 4-5 does not show anything in the L1 cache region that would indicate that the DCU prefetcher is active. This is because the micro-benchmark traverses the array in descending order, thus avoiding a triggering of the DCU prefetcher. However, the spatial and streaming prefetcher are able to successfully recognize the access pattern and hide latency for smaller stride lengths. But for stride size 256 B, and larger, this is not the case. Actually, the spatial prefetcher in this case is contra-productive, since element access does not happen in 128 B blocks for larger stride sizes.

Figure 4-6 shows the results of repeated load latency micro-benchmark after disabling the CPU prefetchers[6]. The repeated results do not show any latency hiding performed by the prefetcher. After the array size increases beyond the L3 cache, around 25 MiB, the read latency increases to 120 ns.

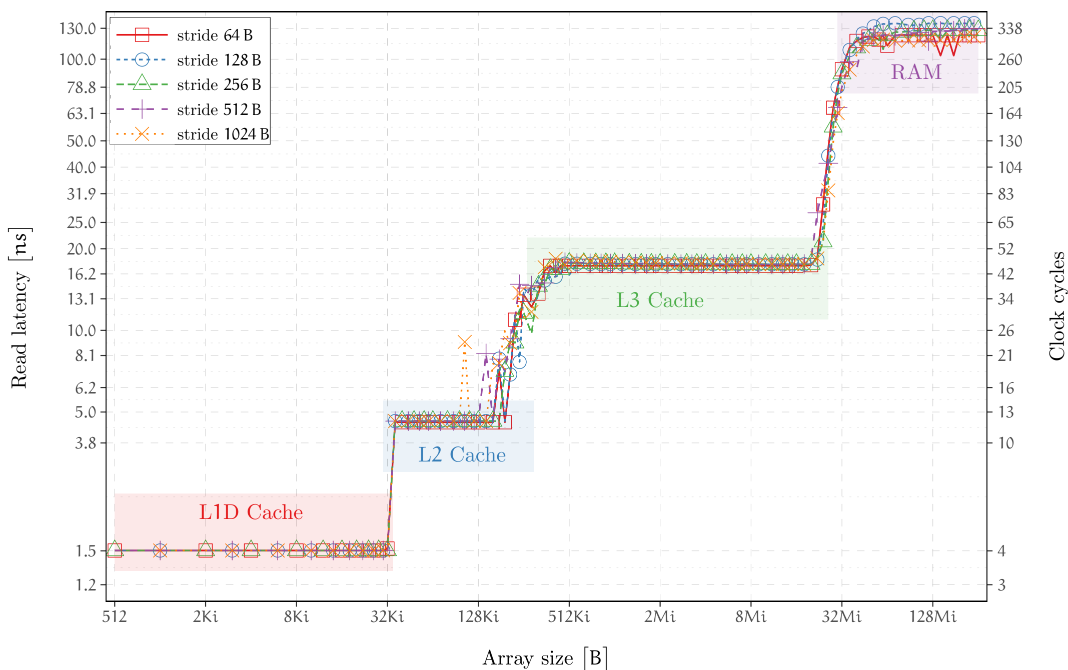


Figure 4-6: Read latency of an Intel Haswell CPU without hardware prefetching

Saturating a memory system with requests, as shown by graph, has an adverse effect on latency. Figure 4-6 illustrates the importance of prefetching. The same test is

[6]In runtime, using *Model-Specific Registers*

performed on the ARM platform, with results shown in Figure 4-7. The L1 data cache size of the ARM CPU core is 32 KiB and can be accessed in 4 clock cycles. The unified L2 cache is 1 MiB and is shared by all 4 CPU cores.

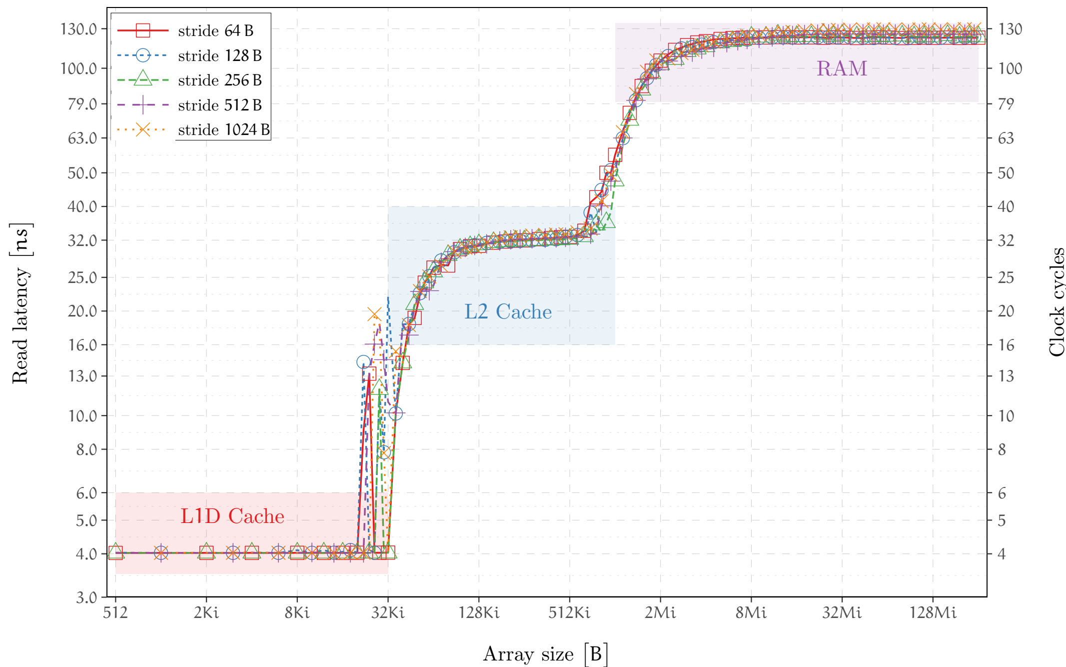


Figure 4-7: Read latency of ARM CPU

The graph shows the Latency of the L1 data cache to be 4 ns, or 4 clock cycles[7]. The L2 cache of the ARM platform has a much larger access latency than Intel CPU. We measured this to be 32 clock cycles. however, RAM access latency is comparable to Intel system, at around 130 ns. As discussed before, our platform has no automatic prefetcher logic that allow for more predictable latencies.

4.3 Conclusion

Even though DDR4 memory on the Intel platform operates at 2133 MHz and has peak bandwidth of 68 GiB s^{-1}, the *load to use* latency is still governed by underlying technology of DRAM. Latency of random DRAM read is dominated by the time it takes to access the desired row of storage cells and *Column Address Strobe* (CAS) latency to read data from it. Peak memory bandwidth has been increasing with every new generation of DIMMs[8], as the CAS latency has also increased. This can be explained by Moore's Law[80], which predicts a doubling of transistors per chip every 22-24 months approximately

[7] ARM CPU core clock operates at 1 GHz
[8] Dual In-line Memory Module (DIMM)

[97]. Bandwidth is improved with more transistors operating in parallel, but ultimately, the latency of DRAM memory is limited by the read and write access times of DRAM cells. Additionally, to be able to support high densities the *Registered memory modules*, RDIMM, are using a register in between the memory controller and the DRAM modules. This reduces electrical load on the memory controller, allowing stable operation with high DRAM densities. This in turn introduces higher latencies due to added components and longer signal propagation paths. In contrast, ARM platform has 2 GiB of unbuffered DDR3 memory, operating at 1066 MHz. While this configuration provides significantly lower peak memory bandwidth, latencies are comparable to the Intel system, as shown by Figure 4-7.

From the results of micro-benchmarks it is obvious that our algorithm must be designed with the memory hierarchy in mind. If a program has a pathological access pattern that defeats the purpose of the caches, we forego substantial performance improvements. Using wider registers, by means of SIMD CPU units, enables higher compute throughput and improves *instruction-per-clock* ratio. Similarly, caching mechanisms of the CPU can hide latency under certain conditions. Generally, a sequential access to memory, coupled with enough instruction in between enables prefetchers to effectively hide *load-to-use* latency completely.

Chapter 5

JIT Generation of Reed-Solomon Erasure Codes

This chapter presents methods for the implementation of the versatile Reed-Solomon erasure coding scheme targeted for modern CPUs. First, we discuss an implementation utilizing a specialized carry-less multiplication vector instruction. The second approach uses the JIT compiler technique to produce optimized erasure codes suitable for multiple CPU platforms. This approach allows construction of codes over arbitrary binary extended fields $GF(2^l)$. Since only a small set of CPU instructions is utilized in the code realization, the algorithm is naturally suitable for execution on the SIMD unit of a CPU. Code of erasure coding methods is produced in run-time, when all parameters of the coding scheme are known. This yields efficient, dense vectorized code free of control sequences. Throughput of vectorized erasure code configurations is compared when running on x86 and ARM platforms. In conclusion, we give an outlook for further research using low power data processing platforms, operating as fast, low power, erasure code computing offload engines.

5.1 Vectorization

Many modern processors support vector instructions as a means to improve data-intensive applications. In fact, according to Flynn's taxonomy [23], modern processors are *Single Instruction Multiple Data* (SIMD) multiprocessors. This data level parallelism is achieved using a vector arithmetic logic unit (vector ALU). Figure 5-1 illustrates a 128 bit vector arithmetic logic unit. The ALU depicted is able to perform operations simultaneously on two 64 bit, four 32 bit, or eight 16 bit operands. Operations are performed on all vector elements in parallel. Other than arithmetic and logic operations, CPU supports optimized instructions for loading and storing data into vector registers.

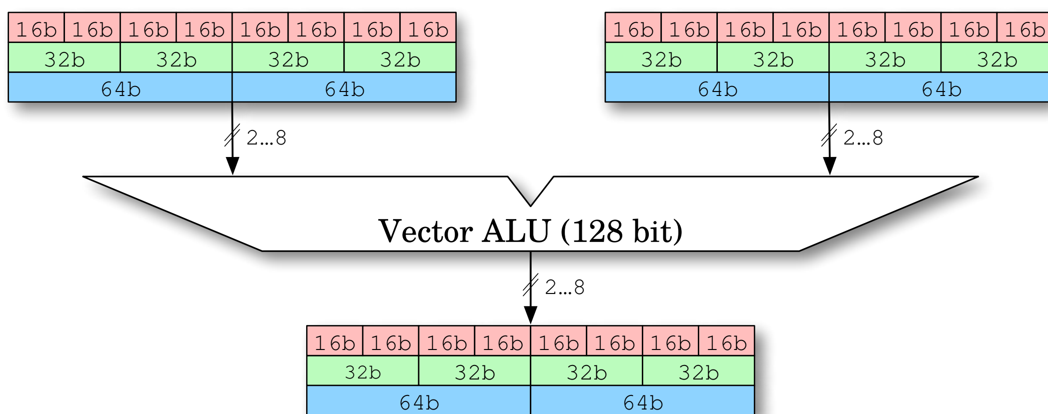


Figure 5-1: Illustration of a 128-bit Vector arithmetic logic unit

An algorithm can take best advantage of vector instruction if the same operation is performed on multiple data elements in sequence. There is usually no mechanism for flow control based on the value of vector register element. If such conditional execution is required, it can be achieved using masking of the results of operations. The mask vector is calculated based on vector elements and applied to selectively propagate the vector element. However, in the worst case this can lead to complete serialization of the execution if all vector elements diverge. Such algorithms that depend on masking operation, typically benefit less than straight-forward data-parallel algorithms.

To generate programs that exploit SIMD capabilities of processors, several different programming choices exist. The following approaches are most common:

Inline assembly This approach, when supported by high-level programming languages, is the most direct way to use vector instructions. However, developing larger algorithms using this approach is complex and error-prone, and should be avoided. The resulting source code is not portable, since compiler directives used for inlining assembly code are not part of high-level languages, such as C or C++.

Intrinsics Some compilers provide special functions that are used to explicitly implement vector instructions for high-level language that do not support vectorization natively. Vector intrinsic functions in C language have a form of typical C function, which operates on special vector types. The compiler maps intrinsic function to corresponding vector operation and parameters to vector registers. Use of intrinsic functions of one CPU with specific vector ALU, creates the code optimized only for that vector unit, which significantly increases porting efforts to the new vector unit.

Vector libraries In order to exploit benefits of vectorization using high-level programming languages, such as C++, libraries for explicit vectorization are needed. One example is the Vc library [59], which provides intuitive API and ensures portability across different compilers and vector unit architectures.

Auto-vectorization Modern compilers support an auto-vectorization feature, which means they are able to automatically exploit vector data-parallelism. The success of auto-vectorization is conditioned on data dependencies of the algorithm. Most basic techniques employed by compilers to vectorize scalar code are loop-level vectorization and block-level vectorization.

Support for special data annotations and native vector data types is another kind of compiler-assisted vectorization. Some C/C++ compilers support special attributes to data types, which should be vectorized. More recent compilers, such as LLVM clang [62], support vector literals natively. Clang adds support vector literals in C/C++ code, at the cost of portability. Internally, LLVM has vector types and permits almost all arithmetic operations. This will be the basis for implementing Reed-Solomon codes using the Just-In-Time (JIT) compilation technique.

The Block Reed-Solomon algorithm has no inter-data dependencies, so an efficient vectorized implementation is expected to bring significant performance over the scalar version. Because we use a buffer-based encoding algorithm, it is easy to employ *horizontal vectorization* in developing the algorithms. This technique combines several independent data symbols, processed in the same way, into a single vector register. After applying algorithm operations on vectors of sequential data symbols, resulting vector will contain the same number of parity-check symbols.

5.1.1 Carry-less multiplication

The need for efficient carry-less multiplication has long been recognized. It is also being used in the computations of several cryptographic systems and standards, such as Elliptic Curve Cryptography over binary fields [58], Galois Counter Mode [67], and Advanced Encryption Standard [16] algorithms. One of the best known algorithms, found in the OpenSSL library [76] [15], is able to emulate the 64 `bit` carry-less product in 100 cycles [67]. For this reason, Intel has introduced a new `PCLMULQDQ` vector instruction as a part

Listing 5-1 PCLMULQDQ instruction intrinsic

```
#include "wmmintrin.h"
__m128i _mm_clmulepi64_si128(__m128i a, __m128i b, const int sel);
```

of the AES-NI [98] instruction set. Intrinsic function of the `PCLMULQDQ` instruction is shown in Listing 5-1.

Operands `a` and `b` are standard 128 bit SSE vector registers. The `PCLMULQDQ` instruction multiplies one quadword (64 bit values) of `a` by one quadword of `b`. The value of the immediate parameter `sel` is used to determine which quadwords of `a` and `b` are used. The result of carry-less multiplication is returned as a double quadword value (128 bit). The pseudocode in Algorithm 5-1 shows operation of `PCLMULQDQ` instruction. The return value at line 14 is defined by Algorithm 3-1.

Algorithm 5-1 Pseudo code of the PCLMULQDQ instruction

```
 1: function PCLMULQDQ(a[127 : 0], b[127 : 0], sel[7 : 0])
 2:     aTemp[63 : 0] ← 0
 3:     bTemp[63 : 0] ← 0
 4:     if IsBitSet(sel, 0) then                    ▷ select quadword from a
 5:         aTemp[63 : 0] ← a[127 : 64]
 6:     else
 7:         aTemp[63 : 0] ← a[63 : 0]
 8:     end if
 9:     if IsBitSet(sel, 4) then                    ▷ select quadword from b
10:         bTemp[63 : 0] ← b[127 : 64]
11:     else
12:         bTemp[63 : 0] ← b[63 : 0]
13:     end if
14:     return CARRY_LESS_MULTIPLICATION(aTemp, bTemp)
15: end function
```

The `PCLMULQDQ` instruction works natively with 64 bit operands, in other words, it operates on elements of $GF(2^{64})$. Performance of this instruction is significantly lower than performance of other arithmetic instructions, as shown in Figure 4-1. This is caused by large latency and a high reciprocal throughput measure associated with the instruction. On Intel Haswell platform, `PCLMULQDQ` instruction has minimal latency of 7 clock cycles, and reciprocal throughput of 2 [24]. This means that new instruction can start to execute 2 cycles after a previous `PCLMULQDQ` instruction, but only if the operands of the new instruction are independent of previous instruction.

If smaller binary fields are required, e.g. $GF(2^{32})$, input operands have to be shuffled into lower parts of input registers. The implementation is trivial but effective throughput of the operation will be reduced. If, however, larger binary fields are required, the result

Listing 5-2 Carry-less multiplication of elements in GF(2^{128}) using PCLMULQDQ intrinsics

```
void clmul_128(__m128i a, __m128i b, __m128i *resU, __m128i *resL)
{
  __m128i c, d, e, f, ef, ef_l, ef_u;
  c = _mm_clmulepi64_si128(a, b, 0x00);
  d = _mm_clmulepi64_si128(a, b, 0x11);
  e = _mm_clmulepi64_si128(a, b, 0x10);
  f = _mm_clmulepi64_si128(a, b, 0x01);
  ef = _mm_xor_si128(e, f);
  ef_l = _mm_slli_si128(ef, 8);
  ef_u = _mm_srli_si128(ef, 8);
  *resU = _mm_xor_si128(ef_l, c);
  *resL = _mm_xor_si128(ef_u, d);
}
```

can be obtained by combining several carry-less multiplications and additions (XOR operations). This approach, for elements of GF(2^{128}) is shown in Listing 5-2. Larger binary fields are used more broadly in cryptography, whereas, specifically in storage applications, smaller binary fields are more commonly used. Both hardware and software RAID implementations, as well as the ZFS RAID-Z erasure scheme, use GF(2^8) as the base field. Because of this reason, PCLMULQDQ is not very well suited for use in storage applications. In the remainder of the thesis, we will describe other vectorized methods, suitable for specific implementations.

The SSE instruction set contains a collection of logic and arithmetic instructions that can be used for implementing carry-less multiplication on small fields like GF(2^8). Unfortunately, the SSE instruction set is not fully orthogonal. Instructions that operate on packed 8 bit integers are not supported, but the 16 bit instructions described in Listing 5-3 can be used. The result of the carry-less multiplication of GF(2^l) requires $2l$ bits for storage. To accommodate this, we split the result into 2 registers.

Listing 5-3 VEX encoded SSE instructions used for implementing GF(2^8) operations

```
vmovdqa m128, xmm       ; load 128 bit of data into xmm
vmovdqa xmm, m128       ; store 128 bit of data to memory
vpsllw  xmm, xmm, imm8 ; shift 16-bit integers left by imm8
vpsrlw  xmm, xmm, imm8 ; logically shift 16 bit integers right
vpand   xmm, xmm, xmm  ; compute bitwise AND
vpxor   xmm, xmm, xmm  ; compute bitwise XOR
```

Listing 5-4 shows an example of vectorized carry-less multiplication in GF(2^8). The lower part of the result is stored in xmm0, while the upper part is in xmm1. Following the carry-less algorithm described in Algorithm 3-1, we obtain the result by shifting the

input and adding these intermediate values to the resulting register. With the omission of the 8 bit `shift` instructions, we have to ensure that upper bits of the 8 bit operands do not cross over into the adjacent data location when using 16 bit `SHIFT` instructions. This is accomplished by applying `AND-masks` that clear upper or lower bits in the 8 bit operands.

Listing 5-4 Vectorized carry-less multiplication by $X^2 + 1$ using SSE vector instructions

```
vmovdqa ([src]), xmm2          ; load source into lower result reg.
vpand   ([mask1]), xmm2, xmm0 ; mask1 = 0x3F (vector)
vpsllw  $2, xmm0, xmm0         ; shift left by 2 bits
vpxor   xmm2, xmm0, xmm0       ; accumulate into lower result reg.
vpand   ([mask2]), xmm2, xmm1 ; mask2 = 0xC0 (vector)
vpsrlw  $6, xmm1, xmm1         ; logically shift right by 6 bits
```

The execution speed of the carry-less multiplication described here depends upon the number of non-zero polynomial coefficients of the multiplier. The total and maximal number of required CPU instructions is given in Table 5-1. It is obvious that the number

Instruction	Count	Maximum count
`vmovdqa`	$4\omega(b)$	32
`vpsllw`	$\omega(b)$	8
`vpsrlw`	$\omega(b)$	8
`vpand`	$2\omega(b)$	16
`vpxor`	$2\omega(b)$	16
Total	$10\omega(b)$	80

Table 5-1: Number of assembler instructions needed to perform carry-less multiplication $a \times b$ in $GF(2^8)$, where $\omega(\mathbf{b})$ is Hamming weight of multiplier b.

of instructions depends strictly on the Hamming weight of the multiplier. Since the carry-less multiplication has to be performed for each element of the Reed-Solomon encoding matrix, it would be beneficial to precondition the encoding matrix in such a way that the sum of the Hamming weights of all elements is minimal. That is, minimize the criterium given by Equation 5.1.

$$\underset{A \text{ is } \mathbf{RS} \text{ matrix}}{\arg\min} \sum_{i,j} \omega(a_{i,j}) \tag{5.1}$$

Strategies for finding the optimal Reed-Solomon generator matrices were described in [55]. Generally, these strategies work only for parity generation, where the preconditioned

generator matrix can be used to reduce the number of instructions. When reconstructing the missing data block, an inverse of the generator matrix is usually calculated as in Equation 2.28. The inversion operation undoes all optimization effort put into preconditioning of generator matrix.

5.1.2 Modulo operation

Vectorized modulo operation can be realized directly from the algorithm described in subsection 3.1.2. Let $p(X) = X^8 + X^4 + X^3 + X^2 + 1$ be the primitive polynomial that generates a field GF(2^8), and let $\mathbf{a}$ be a carry-less multiplication result that needs reduction. Following the modulo reduction algorithm, we have:

$$\begin{aligned} p^{\star}(X) &= X^4 + X^3 + X^2 + 1 \\ q^{+}(X) &= X^8 + X^4 + X^3 + X^2 \end{aligned} \tag{5.2}$$

The upper part of the carry-less multiplication, $c^{\dagger}(\mathbf{a})$ is then multiplied by $q^{+}(X)$ the polynomial. Since we are only interested in the upper 8 bit of the result, we can simplify the procedure. Let $t_1(X)$ be:

$$t_1(X) = M^l(c^{\dagger}(\mathbf{a}) \times q^{+}(X)) = M^l(c^{\dagger}(\mathbf{a}) \times \{X^8 + X^4 + X^3 + X^2\}) \tag{5.3}$$

Then, we define the polynomial $t_2(X)$ as follows:

$$t_2(X) = L^l(t_1(X) \times p^{\star}(X)) = L^l(t_1(X) \times \{X^4 + X^3 + X^2 + 1\}) \tag{5.4}$$

To obtain the modulo reduction, we are only interested in the lower 8 bit of the $t_2(X)$ polynomial.

Listing 5-5 shows implementation of the module reduction algorithm using SSE vector instructions. Algorithm depends strictly on the field generator polynomial $p(X)$, and thus has a constant runtime. The polynomial $t_1(X)$, as shown in Equation 5.3, is calculated in lines 2 through 11. From line 12 to 20, polynomial $t_2(X)$ is derived and added to the lower part of the carry-less multiplication result. This procedure yields the result in register `xmm0`.

Listing 5-5 Vectorized modulo reduction operation for GF(2^8) generated with $p(X) = X^8 + X^4 + X^3 + X^2 + 1$

```
1  vmovdqa ([a_low]), xmm0         ; load upper part of clmul result
2  vmovdqa ([a_up]), xmm1          ; init t1 accumulator
3  vpand   ([mask1]), xmm1, xmm2 ; mask1 = 16(0xF0)
4  vpsrlw  xmm2, xmm2, $4          ; logically shift right by 4 bits
5  vpxor   xmm2, xmm1, xmm1        ; accumulate into t1 result acc.
6  vpand   ([mask2]), xmm2, xmm2 ; mask2 = 16(0xFE)
7  vpsrlw  xmm2, xmm2, $1          ; logically shift right by 1 bits
8  vpxor   xmm2, xmm1, xmm1        ; accumulate into t1 result acc.
9  vpand   ([mask2]), xmm2, xmm2 ; mask2 = 16(0xFE)
10 vpsrlw  xmm2, xmm2, $1          ; logically shift right by 1 bits
11 vpxor   xmm2, xmm1, xmm1        ; accumulate into t1 result acc.
12 vpand   ([mask3]), xmm1, xmm2 ; mask3 = 16(0xFC)
13 vpsllw  xmm2, xmm2, $2          ; shift left by 2 bits
14 vpxor   xmm2, xmm0, xmm0        ; accumulate into result register
15 vpand   ([mask2]), xmm2, xmm2 ; mask2 = 16(0xFE)
16 vpsllw  xmm2, xmm2, $1          ; logically shift right by 1 bits
17 vpxor   xmm2, xmm0, xmm0        ; accumulate into result register
18 vpand   ([mask2]), xmm2, xmm2 ; mask2 = 16(0xFE)
19 vpsllw  xmm2, xmm2, $1          ; logically shift right by 1 bits
20 vpxor   xmm2, xmm0, xmm0        ; accumulate into result register
```

Algorithm	**Throughput** $[\frac{MiB}{s}]$	
	SSE	**AVX2**
`clmul`	2769.55	4806.23
`mod`	2812.40	4912.33
`mul`	1541.30	2863.98
`row mul naïve`	441.68	802.53
`row mul combined`	1025.59	1945.12

Table 5-2: Throughput of vectorized algorithms for carry-less multiplication and modulo reduction for GF(2^8) using SSE and AVX2 instruction set.

5.1.3 Evaluation

The described algorithms are implemented by writing the assembly code by hand. The evaluation is performed on the Intel Haswell platform (Table A-1), using a single CPU core. Table 5-2 summarizes throughput performance of the implemented carry-less multiplication, in Listing 5-4, and module reduction, in Listing 5-5.

Results represent throughput in $\mathsf{MiB\,s^{-1}}$ for carry-less multiplication (`clmul`), modulo operation (`mod`), and complete GF(2^8) multiplication (`mul`). Additionally, multiplication of 4 row elements of encoding matrix are performed, `row mul`. The naïve approach performs full multiplication of each individual element. However, when constructing the Reed-Solomon code, modulo reduction does not have to be performed for each element of the generator matrix, as shown by Equation 3.25. This fact greatly improves throughput of matrix row operations in Reed-Solomon block codes.

AVX2 results are obtained by rewriting the methods using an AVX2 instruction set. With doubling of the register width, from 16 B in SSE, to 32 B in AVX2, we can achieve close to twice as much throughput. One of the reasons we don not reach doubled performance of the SSE methods is the fact that current CPUs run AVX instructions at a lower clock frequency [52]. This is a necessary measure to maintain the thermal design power (TDP) of current CPU architectures. Also, reciprocal throughout of AVX2 instruction is usually lower when compared with the corresponding SSE instruction. E.g., our test CPU can retire 3 XOR vector instructions in a single clock cycle, but is only capable of performing 2 AVX2 XOR operations.

5.2 Just-In-Time compilation of Reed-Solomon codes

Just-In-Time (JIT) compilation is a technique used for compiling executable code during program runtime, often just before execution. This technique has found its use in various scenarios. One of the most used applications is in the translation of byte-code (virtual machine code) to native machine code. In this instance, JIT compilation is used to significantly speed up the execution of such programs (Java or .NET), because execution of native code is more efficient than interpretation of the byte-code. There are also situations where the required information to perform compilation before runtime is not present ahead of runtime. E.g., advanced regular expressions (regexp) implementation can utilize JIT compilation to translate a user-specified matching rule into native machine code and potentially increase string matching performance.

To perform the actual JIT compilation, many projects depend on optimizing compiler infrastructures, such as LLVM [63] and GCC [30], called JIT Compilers. By using the well-established codebases of these compilers, it is also possible to utilize their extensive optimization frameworks to generate faster and more efficient machine code. One of the important benefits is that the translated code will be able to run on many different CPU architectures supported by JIT compilers. If a compiler backend, traditionally called an *Assembler*, can output machine code for different CPU architectures, it is referred to as a retargetable assembler. This is a desired feature because it enables use of different CPU instruction set architectures, both currently existing and upcoming ones.

One of the drawbacks of JIT compilation is latency in execution of generated code. The more optimization passes the JIT compiler performs the more time it will take to

generate the machine code. If the generated code can be saved (*memoized*) and reused later it is worth spending more time on optimization to obtain more efficient machine code. Otherwise, a JIT compiler can make trade-offs between the compilation time and the quality of generated code.

A very similar technique to JIT compilation, is runtime code generation. The important distinction here is that, with code generation, there is no starting code representation. Instead, the running program (host program) outputs the machine code directly according to an algorithm. This is suitable for problems where starting code is not available at runtime (e.g. in case of regexp), or when the problem is well defined and the generated code would not benefit from further optimizations. An implementation of XOR- and ADD-only codes, based on Cauchy-Reed-Solomon codes, is described in [96], and available as QEnc library.

In the following, an implementation of Reed-Solomon block codes using the JIT compilation technique is presented. The presented implementation is able to produce highly optimized Reed-Solomon generating/reconstruction routines over a desired extended field $GF(2^l)$, for $l = \{8, 16, 32, 64\}$. For the actual JIT compilation the LLVM compiler is used, which provides us with the following benefits:

- Runtime code generation without having to provide initial algorithm representation (byte-code)
- Retargetability – support for different CPU platforms
- Performance improvements achieved through extensive optimization framework of LLVM
- Seamless support for SIMD code generation

The resulting, scalar and vectorized, code is evaluated on x86 and ARM platforms.

5.2.1 LLVM as a JIT compiler

The *LLVM* project started as a collection of toolchain components, including compilers, assemblers, debuggers, etc. One of the main design goals was that it be modular, reusable and extendable. The LLVM compiler infrastructure is shown in Figure 5-2. LLVM is designed as a three-phase compiler, consisting of frontend, optimizer, and backend phase.

The frontend parses and processes source code and builds a language agnostic code representation, called LLVM IR (Intermediate Representation). This design simplifies support for high-level programming languages (such as C/C++ or Fortran), as only a dedicated frontend is needed. LLVM IR can also be generated by an interpreter or a virtual machine aiming to optimize a suitable part of the interpreted program. This technique is commonly known as Just-In-Time (JIT) compilation. The optimizer is responsible for applying a variety of transformations and optimizations, with the goal of

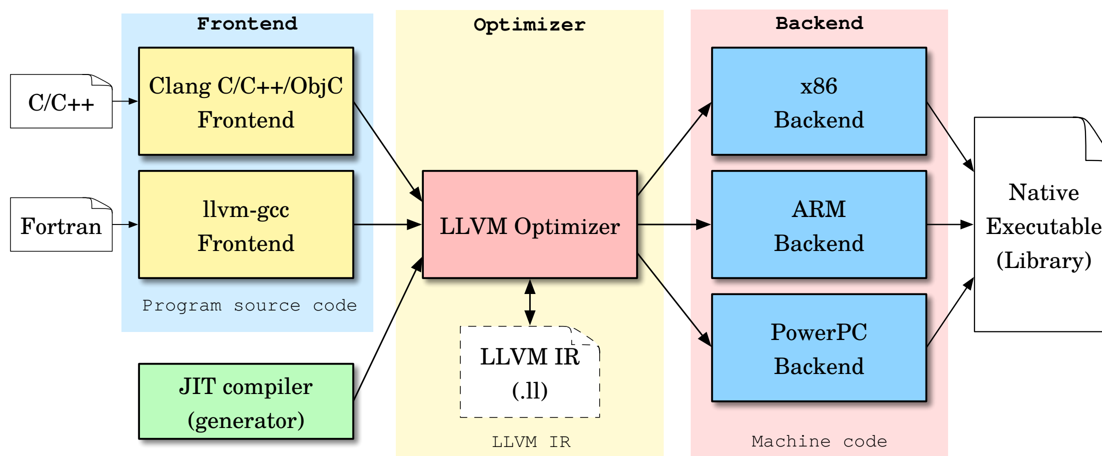


Figure 5-2: LLVM compiler infrastructure

improving program run-time. Most optimizations done by LLVM are independent of the language and target platform. This is accomplished by performing the optimization passes on a well-defined LLVM IR code. Only a small portion of optimization is performed during IR lowering to machine code. The backend is responsible for producing target machine code based on the optimized IR code. Common backend procedures are instruction selection, register allocation, and instruction scheduling. LLVM can run optimizations and produce scalars and vectors on both x86 and ARM CPU architectures.

LLVM IR representation

LLVM IR, sometimes also referred as LLVM assembly language, is an intermediate representation used in all phases of LLVM compilation. It provides type safety, flexibility, vector types, and is capable to represent programs written high-level languages. The Strict Static Single Assignment (SSA) form of the LLVM IR require all variables to be statically defined and assigned value only once. The SSA form simplifies and improves the results of compiler optimizations.

A minimal translation unit of the input program is called a `Module`. A module contains functions, global variables and symbol table entries, expressed in LLVM IR representation. Modules can be combined together by the LLVM linker. This process merges functions and global variables and resolves symbol table entries. It also enables the Link Time Optimization (LTO) passes to reduce and simplify program code. Minimal LLVM `Function` consists of a well-defined function interface, and contains at least single `Basic Block`. A basic block is a sequence of IR instructions, preceded and identified by a `label`. Basic blocks are useful for representing a typical sequence of instructions like loop body.

An excerpt from an LLVM Module is shown in Listing 5-6. The module contains a single function, `rs_gen`, which takes 3 strongly typed parameters. Parameters `d_ptrs`

and `c_ptrs` are addresses of data and parity buffers. On line 4, a local variable of the vector type is allocated on a function stack using the `alloca` instruction, followed by initialization with zero on the next line. The pointer to the first data buffer is loaded on line 7, followed by address offset calculation performed by `getelementptr` instruction. This instruction is used to calculate offset of a data element inside a structure of an array. Data symbols are finally loaded using the `load` instruction annotated with vector type on the line 9. Examples of vector arithmetic and logic instructions are given on lines 11 trough 15.

Listing 5-6 Example of LLVM IR code illustrating vector operations

```
; ModuleID = 'rs_jit_module_0'
define i32 @rs_gen(<16 x i8>** %d_ptrs, <16 x i8>** %c_ptrs, i64 %len) {
entrypoint:
  %C0 = alloca <16 x i8>
  store <16 x i8> zeroinitializer, <16 x i8>* %C0, align 16
  ...
  %D0_ptr = load <16 x i8>** %d_ptrs, align 8
  %D0_ptr_off = getelementptr <16 x i8>* %D0_ptr, i64 %off
  %D0 = load <16 x i8>* %D0_ptr_off, align 16
  ...
  %C0.1 = xor <16 x i8> %D0, %C0
  %D0.l2 = shl <16 x i8> %D0, <i8 2,i8 2,i8 2,i8 2,i8 2,i8 2,i8 2,i8 2,
                               i8 2,i8 2,i8 2,i8 2,i8 2,i8 2,i8 2,i8 2>
  %D0.r6 = lshr <16 x i8> %D0, <i8 6,i8 6,i8 6,i8 6,i8 6,i8 6,i8 6,i8 6,
                                i8 6,i8 6,i8 6,i8 6,i8 6,i8 6,i8 6,i8 6>
  ...
  ret i32 0
}
```

Instead of using the frontend binaries, we are utilizing the JIT helper libraries provided by LLVM. A class diagram of LLVM JIT library is shown in Figure 5-3. To achieve this, our host program links against parts of the LLVM toolchain. The IR Builder API of the JIT helper libraries is provided as a set of C++ classes. The top-level entity is an `LLVMContext` class that owns all modules, constant tables, etc. Upon instantiation of a `Module` object, which represents a compilation unit of our JIT code, we instantiate a new `Function` object. Finally, an instance of `IRBuilder` object is used to insert IR instructions of the function.

Most of the LLVM IR instructions are orthogonal with respect to scalar and vector types. The `IRBuilder` class provides additional operations for use with vector data types, such as element insertion/extraction, splat and shuffle operations. When all instructions and the function epilogue has been written, the JIT helper is used to optimize and create the machine instruction code. The first step in this process is selecting analysis and optimization passes to be run on the new IR module. As mentioned above, we depend heavily upon the compiler to optimize our code, since we have intentionally written redundant operations in the function body. Here we perform aliasing analysis, promoting

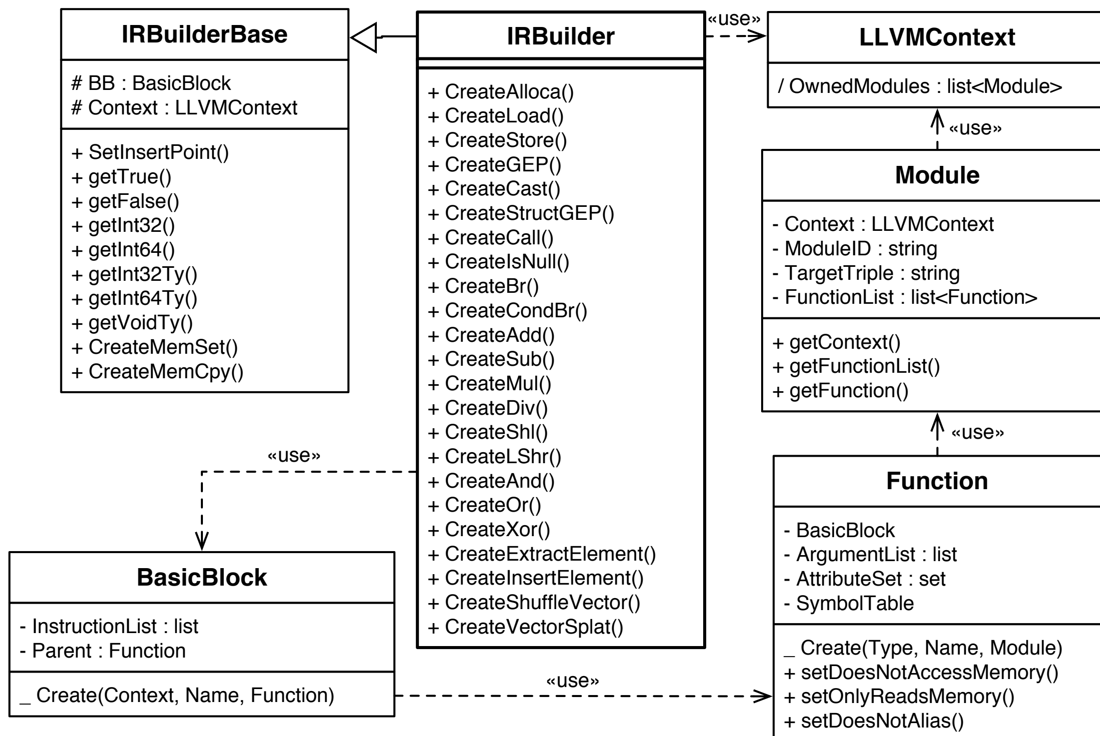


Figure 5-3: Class diagram of LLVM infrastructure for building JIT compiler

memory to registers, instruction combining, dead store elimination, and dead instruction elimination passes. Afterwards, the transformed module IR is finalized and the native machine code is written. Finally, we are given a pointer to the created function that can be called from the host program.

5.2.2 Reed-Solomon Encoding

Buffer-based Reed-Solomon parity generation is described by pseudocode in Algorithm 5-2. The algorithm starts with the creation of the generator matrix for systematic Reed-Solomon code, as described by Equation 2.27. The outer-most loop, at line 3, is used to keep track of current symbol offset in the codeword buffers. Next, local parity variables are cleared and algorithm proceeds to loading of the k data symbols, performed in the loop on line 8. For each data symbol D_{k_i}, partial parity symbols are calculated and updated (line 10.) Once all parity symbols of the current block are calculated, the parity symbols, computed in C_m, are stored into parity buffers (line 14.)

The described algorithm takes several parameters. Parameters m and k are used to parametrize the $\mathcal{RS}(n, k)$ code, where $n = m + k$. These parameters directly govern computational complexity of the solution. Parameters S_B (size of data and code buffers),

Algorithm 5-2 Pseudo code of Reed-Solomon buffer based parity generation algorithm.

```
1:  function RS_PARITY_GENERATION(m, k, S_B, d, c)
2:      A ← RS_CREATE_GENERATOR_MATRIX(m, k)             ▷ Initialize gen. matrix
3:      for off ← 0; off < S_B; off ← off + 1 do
4:          for all C_m do
5:              C_m ← 0                                   ▷ Initialize parity accumulators
6:          end for
7:          for k_i ← 0; k_i < k; k_i ← k_i + 1 do
8:              D_{k_i} ← d[k_i][off]                     ▷ Load data symbol
9:              for m_i ← 0; m_i < m; m_i ← m_i + 1 do
10:                 C_{m_i} ← C_{m_i} ⊕ D_{k_i} · A[m_i][k_i]   ▷ Accumulate parity
11:             end for
12:         end for
13:         for all C_m do
14:             c[m][off] ← C_m                           ▷ Store parity symbols
15:         end for
16:     end for
17: end function
```

$\mathbf{d}$ (data symbol buffers), and $\mathbf{c}$ (parity symbol buffers) do not contribute to algorithm complexity.

To achieve best performance when using JIT compilation, we have to make sure the generated code size is not too large. Generated machine code for the method in Algorithm 5-2 should fit in the CPU instruction cache. This is likely to be expected for reasonably small $m + k$, which are commonly used in storage applications.

By employing JIT compilation, we expect to benefit from the following:

Loop overhead reduction Every loop introduces the overhead by adding loop control instructions. These instructions are especially undesirable in the short loops bodies. Many optimizing compilers perform loop unrolling, where the loop is effectively replaced by repetition of loop body sequences. However, this is only possible when the loop count is known in advance, which is not the case in **RS** algorithm. All inner loops are parametrized exclusively by the m and k parameters, which are known at runtime. In our JIT implementation we completely unroll all inner loops, lines 4-15. Only the outer loop, which iterates over elements of buffers, is kept.

Code optimization By unrolling the inner loops the compiler is able to better optimize the procedure. After the LLVM optimization pass, we expect to see a large number of eliminated subexpressions. Most of eliminations should come from the carry-less multiplication procedure.

Vectorization Internal representation of LLVM supports vector instruction types. Moreover, LLVM allows usage of such types with most arithmetic and logic instructions.

Since we only depend on a limited set of instructions, we can simply select the desired vector type, and have LLVM generate SIMD instruction code for the targeted platform.

5.2.3 Carry-less multiplication

The number of `SHIFT` instructions executed by the CPU depends on the number of bits set to 1 in the first operand of the carry-less multiplication method, as described by Algorithm 3-1. The first operand is the element of the Reed-Solomon generator matrix. By choosing the elements of the Vandermonde matrix, we can minimize the number of operations while performing carry-less multiplication. Still, the algorithm, as described in Algorithm 3-1, contains a condition, on line 4. Branching instruction can reduce performance by introducing stalls in the CPU pipeline in case of a wrong branch prediction. Branch instructions can be replaced by masking operations in vectorized code, but they also contribute to slower throughput by increasing the total number of instructions.

The conditional instruction is performed on the first parameter of the carry-less multiplication method, which is an element of the generator matrix. This means, if we would unroll the loop between lines 3 and 7 in Algorithm 3-1, we would have a specialized carry-less multiplication method for each unique element of the generator matrix. An example of a specialized carry-less multiplication method by elements X^2+1 and X^2+X+1 (decimal 5 and 7) is shown by Algorithm 5-3. Here, the `LeftShift` instruction by 0 and 2 are performed by both specialized multiplication functions.

Algorithm 5-3 Specialization of carry-less multiplication of Galois Field elements

1: **function** CARRY_LESS_MULTIPLICATION_BY_5(a)
2: $r \leftarrow 0$ ▷ Initialize accumulator
3: $r \leftarrow r \oplus$ LEFTSHIFT$(a, 0)$ ▷ 010**1̲**
4: $r \leftarrow r \oplus$ LEFTSHIFT$(a, 2)$ ▷ 0**1̲**01
5: **return** r
6: **end function**
7: **function** CARRY_LESS_MULTIPLICATION_BY_7(a)
8: $r \leftarrow 0$ ▷ Initialize accumulator
9: $r \leftarrow r \oplus$ LEFTSHIFT$(a, 0)$ ▷ 011**1̲**
10: $r \leftarrow r \oplus$ LEFTSHIFT$(a, 1)$ ▷ 01**1̲**1
11: $r \leftarrow r \oplus$ LEFTSHIFT$(a, 2)$ ▷ 0**1̲**11
12: **return** r
13: **end function**

Every data symbol is multiplied by elements of column i of the generator matrix. Since the elements of the generator matrix are constant, we directly output the code for carry-less multiplication into the LLVM IR function body. In this process we will

potentially issue more than one `SHIFT` instruction with the same operands, because they are required by multiple multiplication steps. If these elements of the generator matrix share the subset of bits set to 1, the LLVM optimization pass can eliminate redundant `SHIFT` operations. In that case, we expect the optimizing compiler to reuse the result of the first `SHIFT` operation for all subsequent instances. This means that in the worst case scenario, there will be $l-1$ shift operations, performed per each data symbol, in $GF(2^l)$. Furthermore, we can define a more relaxed optimization criterion for finding optimal generator matrices, than the one previously given by Equation 5.1. A new criterion for finding optimal generator matrices is given by Equation 5.5.

$$\underset{0 \leqslant i < (n-k)}{\arg\min} \quad d_{max}(\mathrm{col}_i(a)) \tag{5.5}$$

The objective is to minimize column-wise Hamming distances of the generator matrix. The more elements share a non-zero bit position the less total `SHIFT` instruction will be needed per a single data element. The maximum number of `SHIFT` instructions needed to generate parity-check symbols of $\mathcal{RS}(n,k)$, defined over $GF(2^l)$, using JIT compilation is $l(n-k)$. In comparison, the worst case bound for traditional implementation is $lk(n-k)$.

5.3 Evaluation

The described method of JIT code allows for implementation of vectorized code with a different vector length, as well as different lengths of the Galois field elements. Multiple data symbols are simply packed into the vector data type of LLVM, which permits effortless horizontal vectorization. SIMD instructions operate on vectors of fixed length. To achieve the best results in our evaluation, we matched the bit size of the element of the Galois field with a length of the vector element. If the specified vector length is larger than the vector register width of the CPU, LLVM will generate code using a technique similar to that of loop unrolling. That is, a single IR vector variable will be represented by two native vector registers.

Not all instructions can be performed at an equal rate by the CPU. Complex, modern, CPU architectures are built to maximize instruction throughput by employing techniques like out-of-order execution, branch prediction, register renaming, data prefetching, etc. AVX (*Advanced Vector Extensions*) introduce several changes to existing 128 bit SSE programming environment. The width of vector registers is doubled, from 128 bit to 256 bit. New registers are renamed from XMM[0-15] to YMM[0-15]. Vector instructions can be encoded in the new `VEX` coding scheme to allow new operands and wider registers. VEX encoding also enables instructions to have three operand formats, where the destination register is different from the source registers. This new format produces more compact code, but the CPU still can perform various optimizations during execution.

Modern CPUs contain larger number of available registers, *Register File*, than it is specified by the architecture. The instruction scheduler is able to detect false data dependencies, arising from reusing the same registers, and exploit superscalar and out-of-order execution to improve performance. This technique is called register renaming, since the CPU actually uses different physical registers. A register allocation technique is used to eliminate register-to-register moves, or register initialization instructions. An example of this technique are an `XOR` operation with two identical registers, or an `SHIFT` operation by 0 places.

Table 5-3 summarizes latency and *reciprocal throughput* of some vector instructions most commonly used by JIT-compiled code. The reciprocal throughput is the reciprocal of the maximum instruction throughput when no data dependencies are present. The values are given for Intel Haswell CPU micro architecture. Each Haswell CPU core is designed for throughput of 4 instructions per cycle and contains a register file with 168 integer registers and 168 vector registers. It has 4 instruction decoders, each capable of generating up to 4 micro-operations[1] (μops) per clock cycle. After instructions have been decoded to their ops form, CPU executed them in a number of execution units. Haswell CPU has 8 execution ports, yielding a theoretical maximal throughput of 8 μops. Some execution units are duplicated, like the 4 integer/vector arithmetic, logic, and shift execution units. The latency of vector arithmetic operations is equal to the same operations on general purpose registers.

Instruction	**Latency**	**Reciprocal throughput**
`movdqa x, m128`	3	0.5
`movdqa m128, x`	3	1
`vmovdqa y, m256`	3	0.5
`vmovdqa m256, y`	4	1
`vpsllw y, y, i`	1	1
`vpxor y, y, y`	1	0.33
`vpxor y, y, m`	2	0.5

Table 5-3: Latency and reciprocal throughput of instructions on Intel Haswell micro architecture

A Haswell CPU core contains two memory ports for reading and one for writing. All memory ports are 256 bit wide. This means that the CPU can perform two full vector loads per clock cycle and one vector store, like indicated by `vmovdqa` entries in Table 5-3.

Internally, all vector execution units have 256 bit throughput, but data paths are divided into two lanes of 128 bit. All vector instructions that move data between these two lanes have a latency of 3 clock cycles, instead of 1. Another, more severe penalty, comes with mixing 256 bit VEX code with legacy, non-VEC encoded 128 bit code. The

[1]Low-level instruction used by the CPU to implement complex instructions

penalty is 70 clock cycles, but compilers are aware of this issue and are capable of avoiding it.

To evaluate the desired effect of JIT compilation, we first looked into generated code. Results of optimizations performed by the compiler can be seen while inspecting instructions of both LLVM IR and generated machine code. We expect to see elimination of the `SHIFT` instructions used in carry-less multiplication. The quality of JIT-generated code is confirmed by examining the LLVM IR produced and the machine code, for a range of Reed-Solomon block code configurations. We counted a number of `XOR` and `SHIFT` instructions. The results are shown in Figure 5-4.

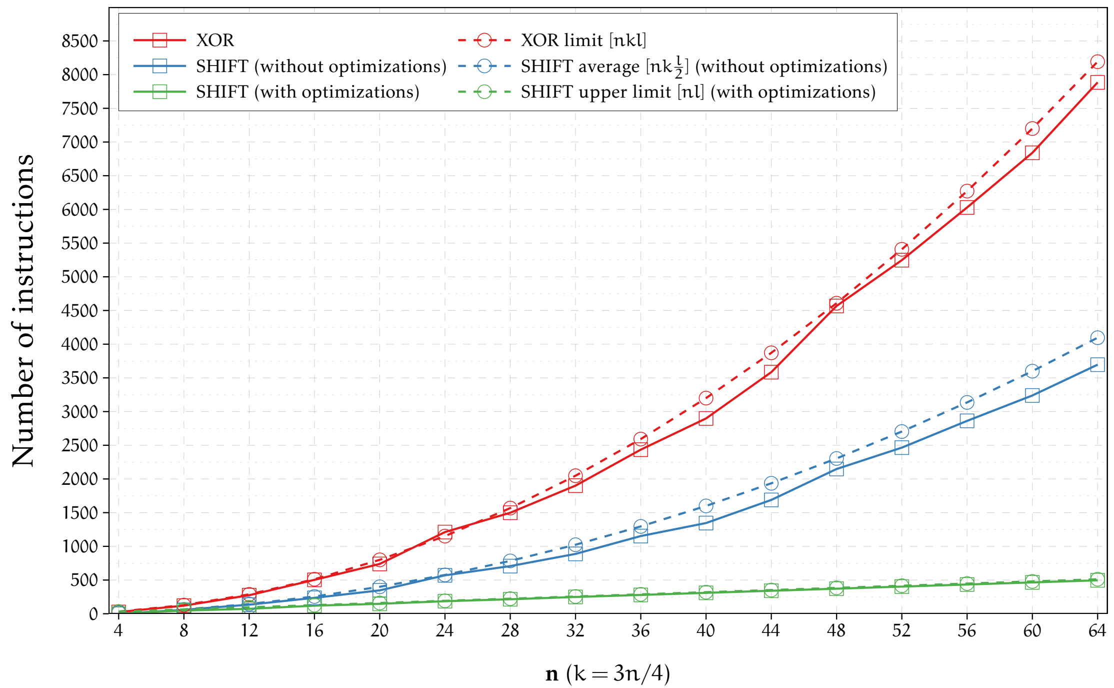


Figure 5-4: Number of instructions needed to implement Reed-Solomon(n,k) block codes over GF(2^8).

The Figure shows the number of instructions executed to produce $\mathcal{RS}(n,k)$ block code[2], for $4 \leqslant n \leqslant 64$, where $k = 3/4\ n$. Our first observation is that the number of `XOR` operations does not change significantly with enabling the JIT optimizations. In fact, the number of `XOR` operations matches the predicted limit (dashed red line). This is because the `XOR` operation is used to accumulate the results in output registers, thus they cannot be removed. On the other side, we see a significant reduction in the number of `SHIFT` instructions used to implement carry-less multiplication. The number of `SHIFT` instructions we initially generated into unoptimized LLVM IR directly corresponds to number of set bits in elements of the generator matrix of the code. Upon examination of the generator matrices, over GF(2^l), we found that one element of the generator matrix

[2]The code adds 25% redundancy overhead

has $l/2$ bits set to 1 on average. This means that an unoptimized $\mathcal{RS}(n,k)$ code, will need to execute $(nkl/2)$ `SHIFT` operations, (dashed blue line). We confirm this also by counting `SHIFT` instructions in the unoptimized LLVM IR representation of the code.

Finally, we counted `SHIFT` operations after applying instruction combining and other optimizations using the LLVM compiler. Ideally, we should see less than $n(l-1)$ `SHIFT` operations because one is not needed for multiplying by 1 in $GF(2^l)$. Our generated code also implements the modulo reduction algorithm, described in Listing 5-5, which introduces a number of `SHIFT` operations that increases linearly with the code size. Thus, the number of actual `SHIFT` instructions is $\approx 7.7n$ for $GF(2^8)$.

After confirming that the LLVM compiler can correctly identify and remove redundant operations, we can measure the real throughput of the generated code. The reduction of complexity, for required instructions, must directly translate to increased code performance. To establish a baseline, we first measured parity-symbol generation throughput of unoptimized Reed-Solomon codes constructed over two Galois fields, over range of block code configurations, and using code generated for scalar, SSE and AVX2 instruction sets. The results are shown in Figure 5-5.

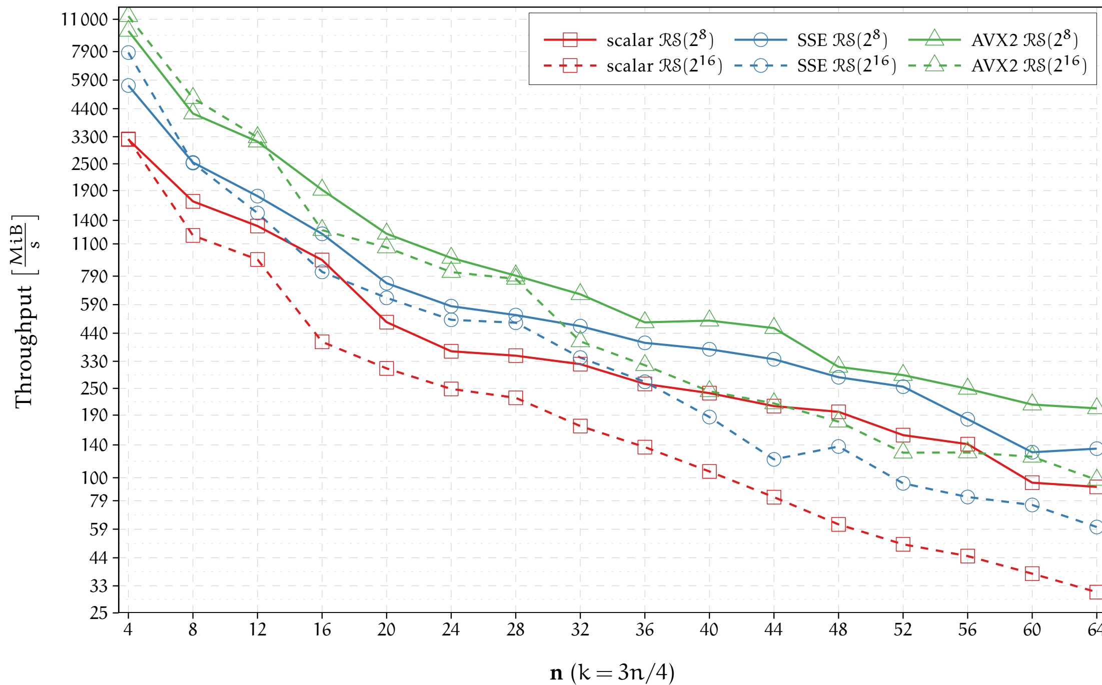


Figure 5-5: Throughput of unoptimized Reed-Solomon(n,k) block codes over $GF(2^8)$ and $GF(2^{16})$, on the Intel Haswell platform

All generated codes are measured on a single CPU core of an Intel Haswell CPU (Xeon E5-2660v3). The plot shows the degradation of performance due to matrix-vector multiplication that is the basis of all Reed-Solomon codes. The LLVM is used to produce

the code for scalar, SSE and AVX2 instruction sets. Since the underlying algorithm is exactly the same, the throughput curve behaves the same for all instruction sets. For small code configurations the encoding performance is limited by memory bandwidth, with encoding throughput approaching the streaming capabilities of the CPU shown in Figure 4-3 and Figure 4-4. For larger n, computational complexity is the limiting factor for throughput. The size of the code produced is also a limiting factor, since we execute a completely unrolled version of the generator matrix multiplication. The Haswell CPU has a 32 KiB Level 1 instruction cache, which is not large enough to hold the unoptimized code for n larger than 48. At this point, instructions in the encoding loop can no longer fit into the L1 instruction cache, causing a further performance decrease.

The results of codes produced by the optimizing JIT compiler are shown in Figure 5-6. The LLVM compiler was run with optimizations equivalent to O3 optimization level on the generated LLVM IR. The figure shows throughput of identical Reed-Solomon code configurations as in Figure 5-5.

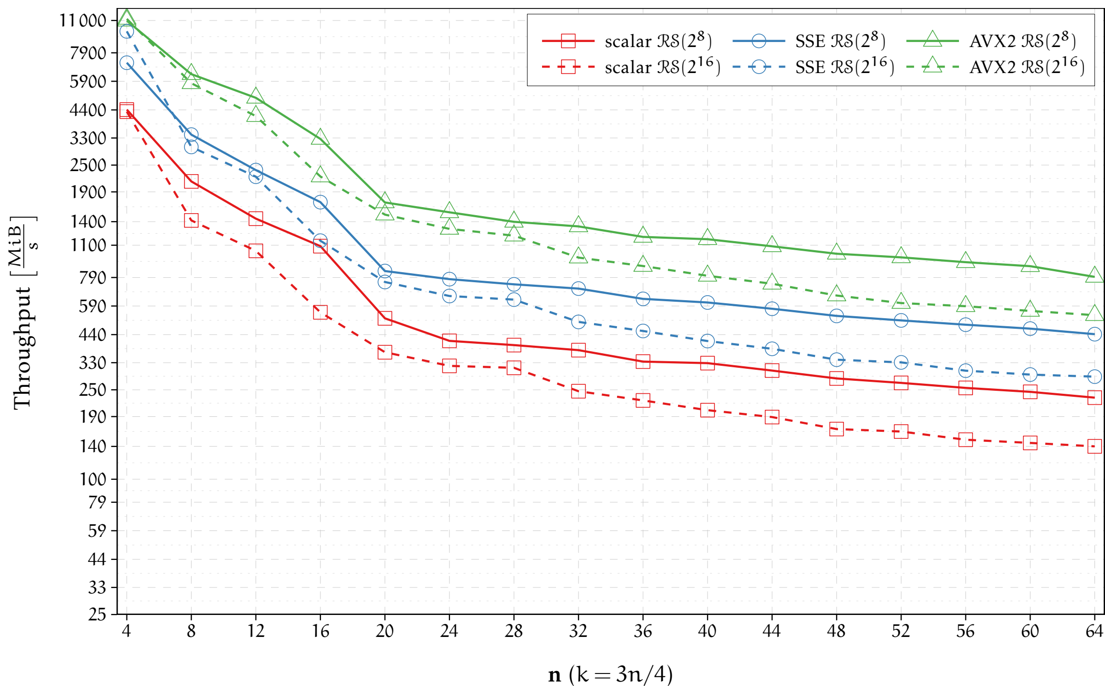


Figure 5-6: Throughput of optimized, JIT generated, Reed-Solomon(n,k) block codes over GF(2^8) and GF(2^{16}), on Intel Haswell platform.

The Figure shows two regimes, one for n smaller than 20 and other for code configurations with larger n. For smaller n and k the generator matrix has less rows, which means that less then l SHIFT operation are needed per data symbol. However, the number of SHIFT operations for each generator matrix row approaches l around $n = 20$. At that point, every increase of generator matrix (n) only adds a constant number of SHIFT instructions. This linear increase results in a less steep performance decline.

With the elimination of `SHIFT` instructions the complete encoding loop fits into the L1 instruction cache, avoiding an instruction decoding bottleneck. Codes implemented over GF(2^{16}), (dashed line) show lower performance because of larger upper bound of `SHIFT` instructions nl. However, GF(2^8) codes are more widely used in typical storage systems applications. LLVM is able to produce scalar code that uses 64 bit general purpose registers as vector registers in $<$ 8x8 bit $>$ configurations (shown by red line). For this reason, we observe doubling of the effective throughput when vectorized code is generated instead of scalar.

The NEON instruction set, general purpose SIMD extension available on ARM Cortex-A processors, offers a register file of 32 64 bit vector registers and dedicated execution pipeline. The same register file can be accessed as 16 128 bit vector registers. The LLVM ARM backend fully supports the NEON instruction set, thus we have evaluated the same Reed-Solomon code configurations on our ARM platform. Throughput of optimized codes compiled for ARM, scalar and NEON versions, is shown in Figure 5-7.

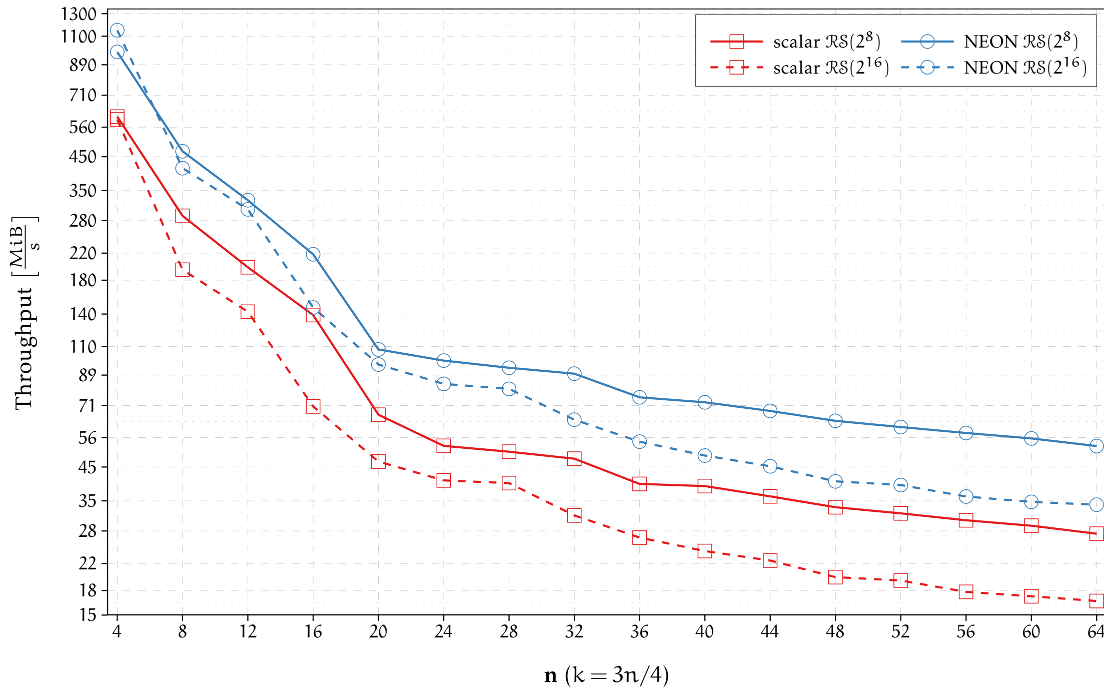


Figure 5-7: Throughput of optimized, JIT generated, Reed-Solomon(n,k) block codes over GF(2^8) and GF(2^{16}), on ARM platform.

The graph shows similar behavior to the one observed on the Intel platform. Since both codes are using exactly the same starting LLVM IR (and code generator matrices), the slope of the curve can be explained in the same way. For low computational complexity configurations, $n < 8$, code throughput approaches the memory bandwidth of the CPU, as shown by the multi-stream memory benchmark in Figure 4-4. For larger n and k,

code computational complexity becomes a limiting factor. Still, the results demonstrate the effectiveness of the CPU SIMD unit of the low power ARM platform.

5.4 Summary

Performance of symbol multiplication in the Galois field is a dominant factor in construction of fast Reed-Solomon erasure codes. We described a constant time vectorized multiplication algorithm in GF(2^{64}), which utilizes specialized SIMD instructions. The limitation of the filed length makes this approach unsuitable for some use cases, where other field lengths are required. To overcome these limitations, the use of JIT code generation was investigated. The proposed Reed-Solomon erasure codes use Vandermonde construction of arbitrary generator matrix sizes. Galois field operations are represented as simple, integer based, operations using LLVM IR language. This formulation allows for arbitrary Galois field lengths, supported by the arbitrary integer precision of LLVM IR. The Reed-Solomon encoder is constructed by emitting all IR methods necessary for performing generator matrix multiplication with the data symbol vector. Intrinsic redundancy in amalgamated operations for the matrix multiplication procedure is removed automatically by optimizing the compiler to produce codes with a lower instruction count per symbol. The data-level parallelism of Reed-Solomon codes is exploited by substituting LLVM IR integer types with vector types suitable for individual test platforms. This enabled seamless generation of efficient, high throughout, vectorized machine codes for both Intel and ARM platforms. The LLVM is able to generate code for SSE and AVX2 instruction sets for x86, and the NEON instruction set of the ARM CPU-based platform. Data-level parallelization reduces the number of instructions per code symbol to ≈ 2.2.

Evaluation shows maximum encoding throughput for small number of data and code symbols to be close to the memory bandwidth of the test platforms. However, the computational complexity of larger matrix multiplications is the dominant factor for large codes. The upcoming AVX-512 instruction set offers more parallelism and expanded register space. Equivalent code realized for this instruction set should have a factor of 2 larger throughput than AVX2 code and 4 times of SSE, for the same clock speed. Unfortunately, while the LLVM already supports the AVX-512 instruction set, Intel Skylake CPUs are not yet available.

Simplicity of operations involved in the realization of Reed-Solomon codes makes them suitable for DSP[3] class of hardware. The LLVM already supports Hexagon DSP [12] natively, which enables us to retarget the JIT generated Reed-Solomon routines for this special-purpose platform. The DSP supports vector instructions, and can execute instructions grouped in VLIW[4] packets. Packets can contain up to 4 instructions executing

[3] Digital Signal Processors
[4] Very Long Instruction Word

in parallel. An example of instruction packets, generated from a Reed-Solomon code method, is shown in the Appendix, section B. The example shows the grouping of `LOAD`, STORE, `SHIFT`, and XOR operations into a single, concurrently executed, wide instruction. By offloading erasure coding tasks from the CPU to a low power platform, such as DSP, significant power savings could be achieved.

Chapter 6

Vectorization of Reed-Solomon Codes used in ZFS File System

The ZFS filesystem has its roots in the Sun Solaris operating system. It started as a proprietary, closed-source project in 2001, which was finally released under a CDDL[1] open source license in 2005. After acquiring Sun Microsystems in 2010, Oracle has stopped contributing to open-source ZFS development, prompting the creation of the OpenZFS project. Open source ZFS quickly found adoption in many major operating systems, including Linux, FreeBSD, and OS X. A license disagreement between Linux kernel, licensed under GNU General Public License (GPL), and ZFS CDDL prohibits distribution of single derived work of both projects. However, it is possible to expose necessary Linux facilities trough a separate, GPL licensed kernel module. A stable version of a native Linux ZFS port, known under the name *ZFS on Linux*[2], was released in 2013 by Lawrence Livermore National Laboratory. It runs natively in a kernel-mode and

[1] Common Development and Distribution License
[2] `http://zfsonlinux.org`

uses and additional kernel module, called *Solaris Porting Layer* (SPL), that implements Solaris APIs on top of the Linux kernel. With the release of the Lustre version 2.4, ZFS on Linux was used as a backing filesystem on Object Storage Targets (OST). An OST is a filesystem that provides an object interface for the Lustre filesystem. With the capabilities of logical volume management and software-defined reliability (RAID-Z), ZFS was adopted for providing resiliency of large, multi-petabyte, file systems.

ZFS uses RAID-Z technology to implement data protection. In essence, this is a software scheme that utilizes error-correction erasure coding to minimize the cost of mirroring in terms of required disks. This is accomplished by joining several disks into a single N-disk RAID-Z volume. Depending on the selected scheme, RAID-Z1, RAID-Z2 or RAID-Z3, such volume is able to recover from 1, 2 or 3 failed disks. RAID-Z1 is equivalent to traditional RAID5, where parity is calculated and distributed together with data over all available disks. In the RAID-Z2 scheme, the same procedure is carried out but with the addition of another parity block, similar to RAID6. Finally, in RAID-Z3, yet another additional error correction block is added to the data. By choosing the number of disks and the level of protection it is possible to mitigate the cost of overhead disks that are used for storing parity instead of contributing to combined usable storage space. On the other side, a higher level of protection and larger RAID-Z volumes increase the computational complexity of erasure codes. Similarly, restoring from failed disks is more computationally intensive than parity generation. Because of this, we implemented efficient scalar and SIMD vectorized versions of the RAID-Z methods.

6.1 RAID-Z theoretical background

The RAID-Z is built upon systematic Reed-Solomon error-correction code, constructed using the Vandermonde method as described in subsection 2.3.1. RAID-Z uses elements of extended field $GF(2^8)$ constructed using primitive polynomial $p(X) = X^8+X^4+X^3+X^2+1$ over $GF(2)$. All elements of this field are listed in Appendix, section C. Generator matrix $\mathbf{A}_{\text{RAID-Z}}$ is obtained by concatenating the identity matrix $\mathbf{I}_k$ and code generating matrix $\mathbf{A}^*_{\text{RAID-Z}}$

$$\mathbf{A}_{\text{RAID-Z}} \cdot \mathbf{d} = \mathbf{c}_{\text{RAID-Z}} \tag{6.1}$$

$$\begin{bmatrix} \mathbf{I}_k \\ \mathbf{A}^*_{\text{RAID-Z}} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{d} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ \mathbf{c}^*_{\text{RAID-Z}} \end{bmatrix}$$

$$\mathbf{A}^*_{\text{RAID-Z}} = \begin{bmatrix} \mathbf{a}_P \\ \mathbf{a}_Q \\ \mathbf{a}_R \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ \alpha^{k-1} & \alpha^{k-2} & \cdots & \alpha & 1 \\ (\alpha^2)^{k-1} & (\alpha^2)^{k-2} & \cdots & (\alpha^2) & 1 \end{bmatrix} \tag{6.2}$$

$$\mathbf{c}^*_{\text{RAID-Z}} = \begin{bmatrix} d_0 + d_1 + \cdots + d_{k-2} + d_{k-1} \\ (((d_0\alpha + d_1)\alpha + \cdots)\alpha + d_{k-2})\alpha + d_{k-1} \\ (((d_0\alpha^2 + d_1)\alpha^2 + \cdots)\alpha^2 + d_{k-2})\alpha^2 + d_{k-1} \end{bmatrix} = \begin{bmatrix} c_P \\ c_Q \\ c_R \end{bmatrix} \tag{6.3}$$

where $\mathbf{A}^*_{\text{RAID-Z}}$ is a Vandermonde-type generator matrix of the code, and the polynomial $\alpha = X$. Code c_P is used when the RAID-Z1 scheme is in effect. From (6.2) it follows that c_P is equivalent to the parity code of RAID5, described in [87]. When RAID-Z2 is used, two code symbols are calculated, i.e. c_P and c_Q. Similarly, the calculation also uses the same principles as the RAID6 scheme [87]. For the highest level of protection offered by ZFS, RAID-Z3, a third code symbol c_R is calculated. There is no corresponding traditional RAID level with 3 code symbols.

The Generator matrix, $\mathbf{A}^*_{\text{RAID-Z}}$, of RAID-Z is constructed using a method described in [87]. The author of the original paper later issued a correction [88] to this method, stating that the generated matrices in some cases do not produce a maximum distance separable (MDS) code. Since the RAID-Z1 and RAID-Z2 schemes are equivalent to the standard RAID levels, they are already known to possess the MDS property. In the following, we give a proof that the RAID-Z3 code is also a MDS code.

Let x, y, and z, $0 \leqslant x < y < z < k$, signify an index of the missing data disk (erasure). Let d_x, d_y, and d_z be the data symbol of the corresponding missing data disk. Let c'_P, c'_Q and c'_R signify partial codes in the presence of erasures, i.e. the codes calculated with the 0 symbol used instead of the missing data symbols. Using this nomenclature, the Equation 6.3 can be written as:

$$\begin{aligned} c_P &= c'_P + d_x + d_y + d_z \\ c_Q &= c'_Q + \alpha^{k-1-x} \cdot d_x + \alpha^{k-1-y} \cdot d_y + \alpha^{k-1-z} \cdot d_z \\ c_R &= c'_R + (\alpha^2)^{k-1-x} \cdot d_x + (\alpha^2)^{k-1-y} \cdot d_y + (\alpha^2)^{k-1-z} \cdot d_z \end{aligned} \tag{6.4}$$

For simplicity, we will introduce the following substitutions:

$$\begin{aligned} a &= k - 1 - x \\ b &= k - 1 - y \\ c &= k - 1 - z \end{aligned} \tag{6.5}$$

After substitutions, the equation (6.4) becomes:

$$\begin{aligned} c_P &= c'_P + d_x + d_y + d_z \\ c_Q &= c'_Q + \alpha^a \cdot d_x + \alpha^b \cdot d_y + \alpha^c \cdot d_z \\ c_R &= c'_R + \alpha^{2a} \cdot d_x + \alpha^{2b} \cdot d_y + \alpha^{2c} \cdot d_z \end{aligned} \tag{6.6}$$

Solving the system in (6.6) for d_x, d_y, and d_z we obtain the following:

$$
\begin{aligned}
d_x &= \frac{(\alpha^{2b+c}+\alpha^{b+2c})\cdot(c_P+c'_P)+(\alpha^{2b}+\alpha^{2c})\cdot(c_Q+c'_Q)+(\alpha^{b}+\alpha^{c})\cdot(c_R+c'_R)}{(\alpha^a+\alpha^b)\cdot(\alpha^a+\alpha^c)\cdot(\alpha^b+\alpha^c)}\\
d_y &= \frac{(\alpha^{2a+c}+\alpha^{a+2c})\cdot(c_P+c'_P)+(\alpha^{2a}+\alpha^{2c})\cdot(c_Q+c'_Q)+(\alpha^{a}+\alpha^{c})\cdot(c_R+c'_R)}{(\alpha^a+\alpha^b)\cdot(\alpha^a+\alpha^c)\cdot(\alpha^b+\alpha^c)}\\
d_z &= \frac{(\alpha^{2a+b}+\alpha^{a+2b})\cdot(c_P+c'_P)+(\alpha^{2a}+\alpha^{2b})\cdot(c_Q+c'_Q)+(\alpha^{a}+\alpha^{b})\cdot(c_R+c'_R)}{(\alpha^a+\alpha^b)\cdot(\alpha^a+\alpha^c)\cdot(\alpha^b+\alpha^c)}
\end{aligned}
\tag{6.7}
$$

In order to obtain a single unique solution from (6.7), it is necessary to satisfy the following:

$$
\begin{aligned}
&\alpha^{2b+c}+\alpha^{b+2c} && \neq 0\\
&\alpha^{2a+c}+\alpha^{a+2c} && \neq 0\\
&\alpha^{2a+b}+\alpha^{a+2b} && \neq 0\\
&\alpha^{2b}+\alpha^{2c} && \neq 0\\
&(\alpha^a+\alpha^b)\cdot(\alpha^a+\alpha^c)\cdot(\alpha^b+\alpha^c) && \neq 0
\end{aligned}
\tag{6.8}
$$

For a result of the addition in $GF(2^8)$ to be a zero element, operands have to be equal. After substitution (6.5), we have a unique a, b and c, such that $0 \leqslant c < b < a \leqslant k$. Since α is a primitive element of the extended field $GF(2^8)$, any expression of form α^i, where i is $0 \leqslant i < 2^8$, will yield a unique, non zero element of the said field. Factoring the equations from condition (6.8) we obtain:

$$
\begin{aligned}
&\alpha^{b+c} \neq 0 \wedge \alpha^b+\alpha^c \neq 0\\
&\alpha^{a+c} \neq 0 \wedge \alpha^a+\alpha^c \neq 0\\
&\alpha^{a+b} \neq 0 \wedge \alpha^a+\alpha^b \neq 0\\
&\alpha^{2b}+\alpha^{2c} \neq 0\\
&\alpha^a+\alpha^b \neq 0 \wedge \alpha^a+\alpha^c \neq 0 \wedge \alpha^b+\alpha^c \neq 0
\end{aligned}
\tag{6.9}
$$

Since all factors are guaranteed to be non-zero, we conclude that set of equations (6.4) has unique solutions for d_x, d_y, and d_z. □

6.2 Implementation

The RAID-Z originally used a look-up table method for Galois Field multiplication. All erasure coding is implemented over the $GF2^8$ binary extended field. An exponentiation table, `Exp[]`, and a logarithm table, `Log[]`, are constructed using the field generator

$p(X) = X^8 + X^4 + X^3 + X^2 + 1$. Multiplication of two symbols, method `mul(a,b)`, can be carried out as:

$$\mathtt{mul}(a, b) = \mathtt{Exp}[\ (\mathtt{Log}[a]\ +\ \mathtt{Log}[b])\ \%\ 255\ ] \tag{6.10}$$

and inverse, `inv(a)`, is obtained using the equation:

$$\mathtt{inv}(a) = \mathtt{Exp}[\ 255\ -\ \mathtt{Log}[a]\ ] \tag{6.11}$$

This approach has several disadvantages. Each of the tables has 2^l elements, and combined they require $2 \cdot 2^l \cdot \frac{l}{8}$ B of memory. Accessing these large tables frequently, within the inner loop of the block code calculation code, can impact performance severely. The growing disparity between the execution speed of CPU and the bandwidth of random access memory (RAM) means that CPU can execute hundreds of instructions during a data fetch from RAM. This effect is called *memory wall* [112], and to minimize its effects on scalability, we should eliminate access to large memory structures within the code. The multiplication is performed with 3 table lookup operations, each requiring an, essentially random, memory access. I.e., only 40% of total memory bandwidth is available for application encoding throughput. Another disadvantage of this method is in the intrinsic serialization of the computation. The lookup tables only support computation of one individual symbol at a time, in this case a single byte. Considering that current processors natively operate with 64 bit-wide registers this means that the algorithm is effectively utilizing only 1/8 of the theoretical scalar operation throughput. For the RAID-Z parity generation, only multiplication by an $\alpha = X$ (decimal {02}) element is required, since the generator matrix is evaluated using a method given by Equation 6.3. This method eliminated the need for multiplication by all powers of α. Since the value for α has only one bit set to 1, the multiplication can be performed without the help of lookup tables. An efficient method for scalar and vector implementation is given in [1]. Listing 6-1 shows implementation of multiply-by {02} methods for a single element and parallel, on 8 elements.

Listing 6-1 Optimized multiplication by 2 in $GF(2^8)$ used in RAID-Z parity generation [114]

```c
uint8_t VDEV_RAID-Z_MUL_2(uint8_t x) {
        return (x << 1) ^ ((x & 0x80) ? 0x1d : 0);
}

uint64_t VDEV_RAID-Z_64MUL_2(uint64_t x) {
        uint64_t mask = x & 0x8080808080808080ULL;
        mask = (mask << 1) - (mask >> 7);
        x = (x << 1) & 0xfefefefefefefefeULL;
        return x ^ (mask & 0x1d1d1d1d1d1d1d1dULL);
}
```

The constant {80} is used to check for the overflow bit on data symbols. In case the most significant bit of a symbol is 1, the result of carry-less multiplication, obtained by left shift operation ($x \ll 1$), needs to be reduced using a *xor* operation with the constant $\{1d\} = X^4 + X^3 + X^2 + 1$. The second method calculates the multiplication by {02} in parallel on 8 elements packed in the scalar 64 bit register. Variable `mask` is used to mark the symbols with overflow bit. The corresponding value of these elements will save the {FF} value in the `mask` variable. On line 8, carry-less multiplication is performed, followed by clearing the least significant bit of the result. This is necessary because the scalar shift operation does not respect byte boundaries of packed elements. Also, since each element has only shifted left by 1, it is expected that each byte symbol will have 0 at its lowest position. Finally, a modulo reduction is applied to the result using a `mask` variable to select elements with overflow bit only.

The method described does not require any additional memory bandwidth and is able to perform packed element multiplication by {02} efficiently. To generate the last parity symbol of RAID-Z3, c_R, multiplication by the $\{02\}^2 = \{04\}$ element is required. This can be achieved by simply repeating the multiplication by {02}, as described previously. An approach for vectorization of this algorithm is described in [1].

6.2.1 RAID-Z parity generation

Since the erasure code used in the RAID-Z data/parity scheme is a fixed, (n, k) linear block code (where $n - 3 \leqslant k < n$), we can provide specialized methods for all parity generation cases. That is, a parity generation method for each of RAID-Z1, RAID-Z2, and RAID-Z3. Also, the new code should support scalar, as well as vector SIMD implementations, such as SSE, AVX2, and ARM NEON instruction sets. Lastly, the implementation should minimize code duplication as much as possible. The underlying algorithm for all methods is the same, so the code should support addition of new architecture implementation with as little effort as possible.

Another consideration is the execution environment of the code. Since our new implementation is contributed to *ZFS on Linux* project, the code will execute inside the Linux kernel context (kernel mode). The consequences of this fact are as follows:

No memory protection The kernel mode code executes with no memory protection. This means that error in the code can corrupt and crash the whole system.

Explicit SIMD code Most of the Linux kernel does not use any floating point or SIMD instructions. Thus, to save time on kernel threads´ context switches, the kernel does not save values of floating point and vector registers. However, if execution of such code is desirable, the code must be enclosed between `kernel_fpu_begin()` and `kernel_fpu_end()` functions.

Stack limit Kernel threads usually execute within limited stacks, usually 8 KiB or 16 KiB on an x86_64 platform. This means that the code should use stack space conservatively, and avoid deep or unbounded recursions.

Portability The code must compile and run on all x86 32 bit and 64 bit platforms. This also means that we have to adhere to strict coding guidelines and the C89 standard. Also, the use of the advanced JIT techniques described in the previous chapter is not possible.

Having considered all of the limitations and requirements listed, final code is implemented in the C programming language and is organized as follows:

Algorithm The Algorithm file contains all specialized methods for parity generation and data reconstruction. These methods depend on a set of data types and macros that are implemented by each specific implementation separately.

Implementation Each new implementation will define the required data-type structures to match native register sizes and implement a set of basic operations on these types. Each implementation is contained in a separate translation unit. An implementation file will then include an algorithm file, completing the specialization of the algorithm methods. Specialized methods are exported through a collection of function pointers.

Selection Each implementation has to provide a method of testing whether it is supported on a currently executing CPU. All supported implementations are benchmarked when the ZFS filesystem is initialized, and the fastest one is selected to be used in the runtime. The user is given an option to change the used implementation using kernel module parameters.

An example of the algorithm method for RAID-Z3 parity generation is shown by Algorithm 6-1. The method `GEN_PQR_PARITY()` takes an array of input data buffers (`d[][]`) and the size of buffers (S_B) and produces parity codes in buffers $\mathbf{c_P}[]$, $\mathbf{c_Q}[]$ and $\mathbf{c_R}[]$.

The method described is only provided as a template. An implementation must provide all macros and variables underlined in Algorithm 6-1 to fully specialize the algorithm method. Macro `DECLARE_LOCAL_VAR`() is used to declare all variables that will be used by this method. In the case of vectorized implementation this declaration is unnecessary since architecture registers are used explicitly, but it is required for the scalar implementation. An implementation must also provide a method for loading data from memory into a designated variable or register. This is accomplished by providing a `LOAD`() macro that performs a suitable memory load. The specialized macros `MUL2`() and `MUL4`() are used because special cases of multiplication by {02} and {04} in the Galois Field are usually much faster and easier to implement than generalized multiplication. Finally, `XOR`() and `STORE`() macros are used to perform addition and to store calculated parity symbols to the memory. All described macros can be implemented as variadic macros in the C programming language, enabling an implementation to fully utilize available registers. E.g., SSE and AVX2 implementations are computing each of the parity using 4 registers at once, thus reducing the loop count to 1/4.

Algorithm 6-1 Method for generating RAID-Z3 parity

```
1: function GEN_PQR_PARITY(d[][], cP[], cQ[], cR[], SB)
2:     DECLARE_LOCAL_VAR(D, P, Q, R)                 ▷ Declare local variables
3:     for off ← 0; off < SB; off ← off + 1 do
4:         P ← Q ← R ← LOAD(d[0][off])               ▷ Load first data symbol
5:         for b ← 1; b < |d|; b ← b + 1 do
6:             Q ← MUL2(Q)                           ▷ Multiply by {02}
7:             R ← MUL4(R)                           ▷ Multiply by {04}
8:             D ← LOAD(d[b][off])                   ▷ Load current data symbol
9:             P ← XOR(P, D)                         ▷ Add data symbol to P
10:            Q ← XOR(Q, D)                         ▷ Add data symbol to Q
11:            R ← XOR(R, D)                         ▷ Add data symbol to R
12:        end for
13:        STORE(cP[off], P)                         ▷ Store cP parity
14:        STORE(cQ[off], Q)                         ▷ Store cQ parity
15:        STORE(cR[off], R)                         ▷ Store cR parity
16:    end for
17: end function
```

6.2.2 RAID-Z data reconstruction

Reconstruction of erasures in the complete RAID-Z scheme can be implemented in seven specialized methods. The same considerations as with parity generation apply. The only substantial difference is that now we need an efficient way to perform multiplication in the Galois Field by an arbitrary constant. The original code used a `Log/Exp` lookup table method, described by Equation 6.10. As the multiplier, `b` is always known in advance if the logarithm is provided directly, which saves one lookup `Log[b]` operation in runtime. Still, both `Log[]` and `Exp[]` lookup tables are used in a random access pattern. This method requires 2 lookup table operations per symbol to perform multiplication by a constant. For a 64 bit scalar implementation, this amounts to 16 lookup operations in order to multiply all elements packed in a scalar register, not counting byte extracting operations for each byte element.

In order to perform erasure reconstruction, each method computes the respective syndromes first, and then reconstructs the missing data, as described by Equation 6.7. As it turns out, only a few multiplication factors are actually used by each reconstruction method, as shown in Table 6-1.

With this observation, we can make a space-time trade-off by providing specialized lookup tables for each multiplicative element (`Mul[][]`). This strategy increases total lookup tables to 64 KiB, but the access pattern is limited to only a few of 256 B long tables at once. Numbers of operations and size of accessed lookup tables for two scalar multiplication methods is shown in Table 6-1. The table shows that the number of operations is halved with the new approach and that the total table size only increased in

Method	Log[]/Exp[]		Mul[][]		AVX2 shuffle	
	#ops	size	#ops	size	#ops	size
Reconstruct using P	0	0 B	0	0 B	0	0 B
Reconstruct using Q	2	512 B	1	256 B	0.125	64 B
Reconstruct using PQ	4	512 B	2	512 B	0.25	128 B
Reconstruct using R	2	512 B	1	256 B	0.125	64 B
Reconstruct using PR	4	512 B	2	512 B	0.25	128 B
Reconstruct using QR	8	512 B	4	1024 B	0.5	256 B
Reconstruct using PQR	12	512 B	6	1536 B	0.75	384 B

Table 6-1: Number of lookup table operations per symbol and total size of lookup tables required by each of erasure reconstruction methods of RAID-Z scheme.

two of the reconstruction methods (QR and PQR). Furthermore, for reconstructing only a single erasure, which is a common real-world scenario, the size of lookup tables required to perform reconstruction is halved, when compared to the original implementation. This brings speedup of 2.4 times over the original implementation.

A vectorized multiplication algorithm suited for the $GF(2^8)$ Galois Field is described in [89]. The described algorithm calculates full SIMD vector multiplication in constant time using precomputed multiplication tables and the vector element's *shuffle* instructions. This is accomplished by computing two lookup tables, called "right-left" tables as described in [34]. Authors have provided an implementation library, *GF-Complete* [90], implementing the methods. Unfortunately, a company has made claims that the method infringes on its patented technology[3]. Authors have subsequently removed the implementation library.

Thus, the multiplication method we used here relies on methods described in previous chapters. To improve the throughput of our method, we employed techniques that are based on composite field methods [34] [77] [78]. The Galois Field $GF(2^8)$ is an isomorphic to field $GF((2^4)^2)$ generated with an irreducible polynomial $p(X)$ with a degree of 4 with coefficients in $GF(2^4)$. This property enables us to rewrite the starting carry-less multiplication in Galois Field given by Equation 3.2 andEquation 3.3 in the following way.

Let b^u be the quotient and b^r be the remainder from the polynomial division of $b(X)/X^4$, that is $b(X) = b^u \times X^4 + b^r$. Then:

$$\begin{aligned} a \times b &= a \times (b^u \times X^4 + b^r) \\ &= [a \times b^r] + [X^4 \times a \times b^u] \end{aligned} \tag{6.12}$$

[3]StreamScale Inc, Accelerated erasure coding system and method - United States Patent No. 8,683,296

Since both b^u and b^r are polynomials of degree 4, the previous equation can be implemented using vector *shuffle* instruction. The instruction is part of the SSSE3 and the AVX2 instruction sets [42] on x86 platform, `PSHUFB` and `VPSHUFB`. On the ARM platform, similar instruction, `VTBL`, is available in NEON instruction set [31]. The vector shuffle instruction is capable of performing 16 parallel, byte-wise, table lookups in vector registers. From the last equation it follows that the result of carry-less multiplication for a fixed b can be obtained as a sum of two table lookups, by b^u and b^r. Since the result of carry-less multiplication is $2l$ bits long (16 in this case), the upper bits will be implicitly calculated in the modulo reduction step.

Let $u(X)$ be a polynomial of a degree up to 16; we define a polynomial $M^8[u(X)]$ as a quotient, and $L^8[u(X)]$ as the remainder of the polynomial division $u(X)/X^8$. Then, Equation 6.12 can be rewritten as follows:

$$\begin{aligned} a \times b = & L^8\Big[a \times b^r\Big] + L^8\Big[X^4 \times a \times b^u\Big] \\ & + X^8 \times M^8\Big[a \times b^r\Big] + X^8 \times M^8\Big[X^4 \times a \times b^u\Big] \end{aligned} \tag{6.13}$$

Terms with $L^8[]$ describe a formula to calculate carry-less multiplication using 16 bit-wise vector lookup operations. Terms with $M^8[]$ are the upper $8\,\mathrm{bit}$ results of carry-less multiplication, denoted by $c^\dagger(X)$ in Equation 3.8.

To obtain the modulo reduction component, we follow the algorithm described in [36] as outlined in the subsection 3.1.2. Using the RAID-Z field Galois Field generator polynomial $p(X) = X^8 + X^4 + X^3 + X^2 + 1$, using the Barrett reduction algorithm, we obtain the polynomials $p^\star(X)$ and $q^+(X)$ as follows:

$$\begin{aligned} p^\star(X) &= X^4 + X^3 + X^2 + 1 \\ q^+(X) &= X^8 + X^4 + X^3 + X^2 \end{aligned} \tag{6.14}$$

Using Equation 3.21, we get the expression for the module reduction of the upper part of the carry-less multiplication $u(X)$, as follows:

$$\begin{aligned} u(X) &= L^8\Big[p^\star(X) \times M^8\Big[c^\dagger(X) \times q^+(X)\Big]\Big] \\ &= L^8\Big[p^\star(X) \times M^8\Big[\Big(X^4 \times M^4[c^\dagger(X)] + L^4[c^\dagger(X)]\Big) \times q^+(X)\Big]\Big] \\ &= L^8\Big[p^\star(X) \times M^8\Big[X^4 \times q^+(X) \times M^4[c^\dagger(X)] + q^+(X) \times L^4[c^\dagger(X)]\Big]\Big] \end{aligned}$$

Finally, we obtain a formula for the modulo operation that is compatible with the vector shuffle operation:

$$\begin{aligned} u(X) = L^8\left[p^\star(X) \times M^8\left[X^4 \times q^+(X) \times M^4[c^\dagger(X)]\right]\right] \\ + L^8\left[p^\star(X) + M^8\left[q^+(X) \times L^4[c^\dagger(X)]\right]\right] \end{aligned} \tag{6.15}$$

where $c^\dagger(X)$ follows from the Equation 6.13:

$$c^\dagger(X) = M^8\left[a \times b^r\right] + M^8\left[X^4 \times a \times b^u\right] \tag{6.16}$$

In order to implement the vectorized multiplication algorithm, the necessary shuffle lookup tables are precomputed as shown by Equation 6.12 and Equation 6.15. Algorithm 6-2 shows this procedure. Operation "×" is carry-less multiplication as defined by Algorithm 3-1.

Algorithm 6-2 Calculation of carry-less multiplication and modulo reduction lookup tables for parallel, 16 byte-wise, multiplication using vector shuffle operation.

```
 1: LT_CLMUL_UPPER[256][16] ← {0}
 2: LT_CLMUL_LOWER[256][16] ← {0}
 3: LT_MOD_UPPER[256][16] ← {0}
 4: LT_MOD_LOWER[256][16] ← {0}
 5:
 6: function GEN_SIMD_MUL_TABLES( )
 7:     for b ← 0; b < 256; b ← b + 1 do
 8:         for k ← 0; k < 16; k ← k + 1 do
 9:             LT_CLMUL_UPPER[b][k] ← L^8[X^4 × k × b]
10:             LT_CLMUL_LOWER[b][k] ← L^8[k × b]
11:             LT_MOD_UPPER[b][k] ← L^8[p*(X) × M^8[M^8[X^4 × k × b]]]
12:             LT_MOD_LOWER[b][k] ← L^8[p*(X) × M^8[M^8[k × b]]]
13:         end for
14:     end for
15: end function
```

Even though the complete set of shuffle lookup tables takes 16 KiB of memory space, only 64 B of the lookup tables are required to perform parallel multiplication by a constant. Using the SSSE3 vector shuffle instruction, the vector multiplication algorithm can perform 16 parallel multiplications by a constant, while the AVX2 version can perform the same operation on 32 elements at once. Precomputed shuffle lookup

Algorithm 6-3 Parallel multiplication method using SIMD shuffle instruction and precomputed lookup tables.

1: **function** SIMD_GF_MUL($\langle a \rangle$, b)
2: $\langle a_l \rangle \leftarrow \langle a \rangle$ AND <0x0f>
3: $\langle a_u \rangle \leftarrow \langle a \rangle \gg 4$
4: $\langle r \rangle \leftarrow$ SHUFFLE($\langle a_u \rangle$, LT_CLMUL_UPPER[b])
5: $\langle r \rangle \leftarrow \langle r \rangle$ XOR SHUFFLE($\langle a_l \rangle$, LT_CLMUL_LOWER[b])
6: $\langle r \rangle \leftarrow \langle r \rangle$ XOR SHUFFLE($\langle a_u \rangle$, LT_MOD_UPPER[b])
7: $\langle r \rangle \leftarrow \langle r \rangle$ XOR SHUFFLE($\langle a_l \rangle$, LT_MOD_LOWER[b])
8: **return** $\langle r \rangle$
9: **end function**

tables are not accessed randomly, but instead only 4 16 B values are used for parallel multiplication function, as shown by Algorithm 6-3.

Naïve construction of lookup tables places each 16 B segment in separate arrays which wastes memory bandwidth, because CPU performs memory transfers in cache line chunks. The actual implementation interleaves 4 lookup tables so that all elements, used by a single multiplication method, are placed in consecutive memory locations. Also, the whole lookup table is aligned to the CPU cache line size. As the typical cache line size of a cache line is 64 B [20], a multiplication method is guaranteed to use only a single cache line for lookup table entries. This optimization allows greater throughput by reducing memory fetches from 4 cache lines to just one. Table 6-1 shows the number of operations and lookup table size per reconstructed symbol using original, and new scalar and vector implementations. A parallel multiplier, using an SIMD AVX2 instruction set, is capable of greatly reducing the number of lookup instructions when compared to the best scalar method. Besides an 8 fold increase in throughput performance, the SIMD implementation reduces lookup table size to 1/4 that of the scalar implementation.

6.3 Evaluation

To aid the process of evaluation of the RAID-Z erasure coding methods, we implemented a user-space tool to benchmark the code. The benchmark tool is capable of executing methods of the new implementation as well as the original RAID-Z code. All benchmarks are performed on an Intel Haswell based test platform that supports both SSE and AVX2 instruction sets.

To determine the code throughput limitations, we first performed measurements of RAID-Z2 methods using 8 data disks, i.e. using linear block code configuration $\mathcal{RS}(10, 8)$. Throughput of RAID-Z2 parity generation and data reconstruction methods, running on a single CPU core, is shown in Figure 6-1.

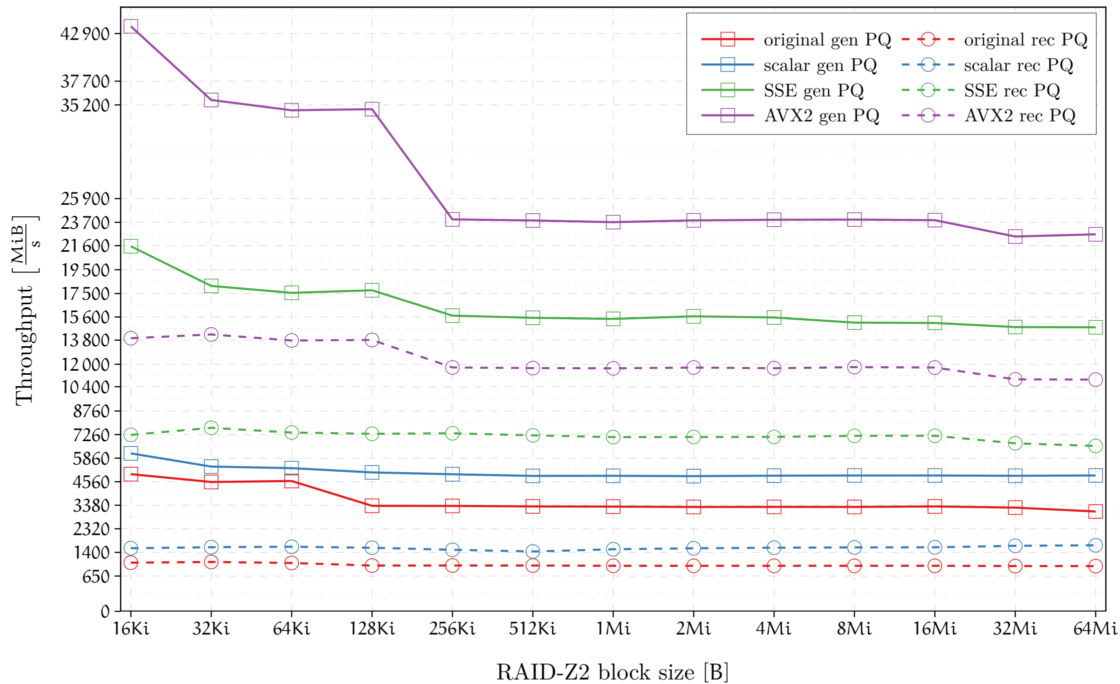


Figure 6-1: Combined throughput of RAID-Z2 parity operations on a pool consisting of 8 data and 2 parity disks

RAID-Z block size takes values between 16 KiB and 64 MiB[4] so that data does not fit into the CPU caches. The plot shows original and newly implemented scalar, SSE and AVX2 RAID-Z methods. Effects of CPU data caching are immediately noticeable, which suggest that the throughput is limited by a *memory wall* rather than by computational complexity. This effect is more pronounced with less computationally intensive parity generation methods, and with wider register sizes. CPU of the test platform has 32 KiB L1 and 256 KiB L2 data cache per CPU core, and 25 MiB of unified L3 data cache, shared between all CPU cores. Measured methods show drops in throughput exactly when block data becomes larger than the L1 and L2 cache, but throughput of large block remains on the level of L3 cache, around 23 000 MiB s^{-1}, as seen on Figure 4-2. For this to be true, accessed data has to be prefetched ahead of time by a CPU automatic hardware prefetcher.

The effectiveness of these prefetchers in this case is confirmed by utilizing `Linux perf` utility to monitor CPU performance counters. The L3 cache miss rate is only 3.67% for RAID-Z blocks larger than 4 MiB. We also repeated the same test after disabling automatic prefetchers of the CPU. Results are shown in Figure 6-2.

The new graph shows a similar performance curve as the previous measurement, until data becomes larger than the L3 cache. At that point, each new memory access

[4]Default value of RAID-Z block size is 128 KiB, and has only recently been expanded to 1 MiB. Maximal supported size is 16 MiB.

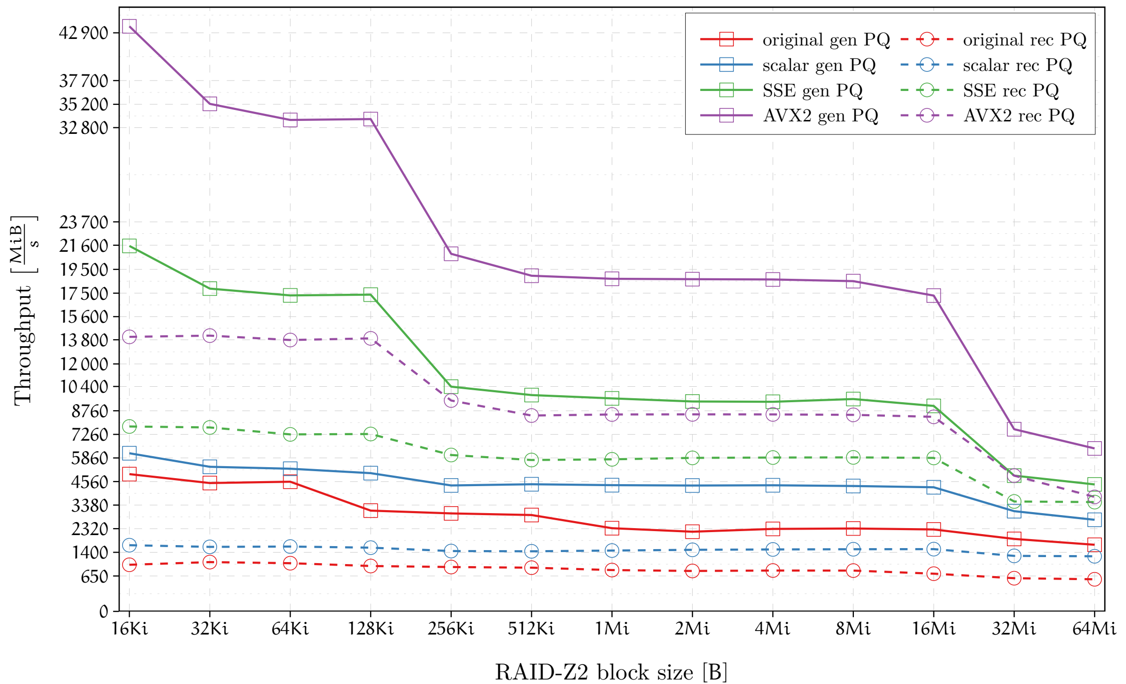


Figure 6-2: Combined throughput of RAID-Z2 parity operations running on Intel Haswell CPU with disabled hardware prefetching.

has to be served from RAM, thus incurring full transfer latency, as seen in Figure 4-6. This confirms that automatic prefetcher is able to hide RAM latency by prefetching data before it is needed. Our re-implementation of scalar methods for RAID-Z2 shows better throughput mainly because of new scalar multiplication table method. Table 6-2 shows speedup of new RAID-Z implementations relative to the original.

RAID-Z operation	**scalar**	**SSE**	**AVX2**
Generate P	2.2	2.4	2.6
Reconstruct using P	1.4	2.0	2.2
Generate PQ	1.5	4.1	4.3
Reconstruct using Q	1.5	7.2	8.8
Reconstruct using PQ	1.6	4.7	7.1
Generate PQR	1.4	5.6	8.8
Reconstruct using R	4.8	20.7	32.3
Reconstruct using PR	8.5	43.0	69.1
Reconstruct using QR	8.7	35.5	60.2
Reconstruct using PQR	9.4	55.1	95.8

Table 6-2: RAID-Z operation speed-up relative to the original RAID-Z methods

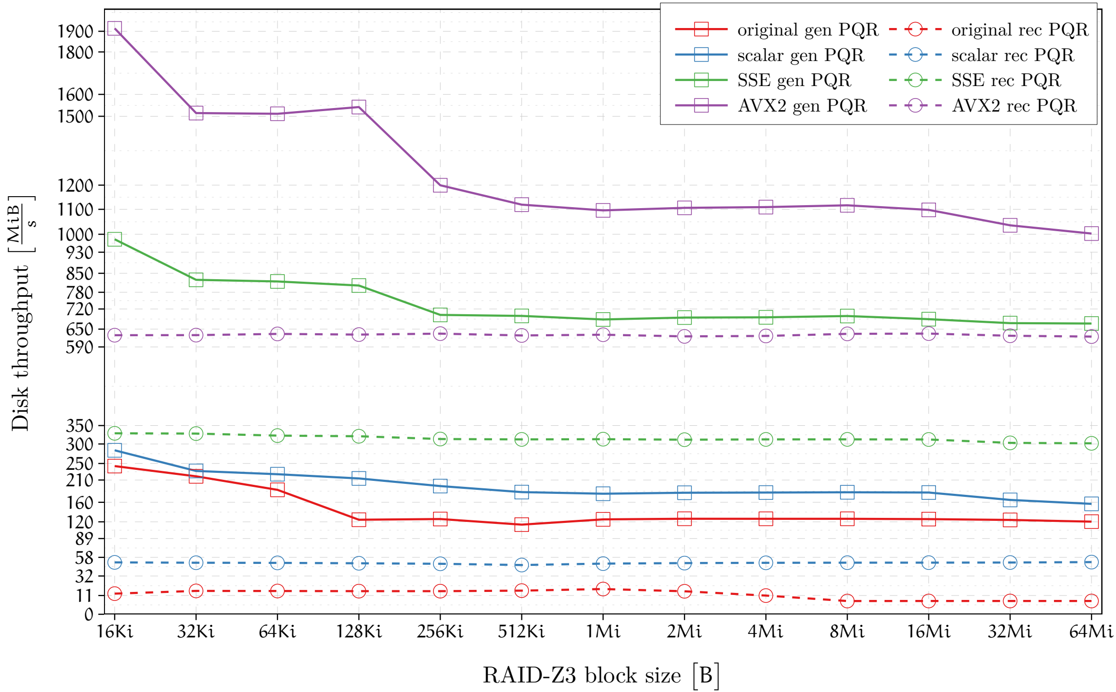


Figure 6-3: Per disk throughput in RAID-Z3 pool with 8 data and 3 code disks with data already present in CPU caches

For RAID-Z2, methods including c_Q parity, we observe a significant increase in throughput, especially for vectorized reconstruction methods. This is a consequence of the ability to perform parallel Galois Field multiplication, as opposed to byte-wise serial access of scalar multiplication. Similarly, Figure 6-3 shows performance of RAID-Z3 methods, with 8 data and 3 parity disks, i.e. in $\mathcal{RS}(11, 8)$ erasure code configuration.

The log-log plot shows *per disk throughput* of original, and new scalar, SSE and AVX2 methods of RAID-Z3. This per-disk metric shows an important aspect of the code efficiency. Being able to saturate available bandwidth of all HDD in the RAID-Z pool using just a single CPU core brings significant increase in scalability and efficiency of the storage system. With a more computationally complex RAID-Z3 algorithm, we see greater disparity between original and new scalar erasure reconstruct methods. Primarily, this is caused by lack of specialized reconstruction methods that utilize c_R parity in the original implementation. The AVX2 implementation shows throughput of $1000\,\mathrm{MiB\,s^{-1}}$ per disk for computing 3 parity symbols (AVX2 gen PQR line), resulting in total encoding throughput of $11\,\mathrm{GiB\,s^{-1}}$. This is the memory bandwidth limit of the single CPU core. The last section in Table 6-2 shows relative speedup of new RAID-Z3 implementations, compared to original code.

6.4 Summary

The ZFS file system is being increasingly used in high performance storage environments for its software-defined reliability features. Recent versions of the Lustre parallel system use ZFS for object storage targets, to provide resiliency for multi-petabyte file systems. Having no dependencies on hardware RAID solutions, ZFS simplifies hardware infrastructure, while relying on CPU for data redundancy and integrity calculations. Typically, a single server controls several ZFS pools, putting even greater stress on the CPU and I/O sub-system for parity operations.

ZFS supports the erasure coding scheme for the storage pool, RAID-Z, which can add resiliency for up to 3 failures. Fast recovery from disk failures is a major concern for data resiliency of RAID-like storage systems, as shown by reliability models in the chapter 1. Original RAID-Z parity computation code provides specialized scalar methods only for 2 parity modes, RAID-Z1 and RAID-Z2. Recovering form failures using third parity block, in RAID-Z3 scheme, was not handled as efficiently. Throughput of parity operations can be greatly improved by implementing specialized methods for all parity operations, and by performing computationally intensive parity operations using SIMD units of modern CPUs.

We provided a proof that systematic code used to construct RAID-Z3 is indeed an MDS erasure code. Next, we described a generic framework for RAID-Z parity operations, which supports implementation for multiple instruction sets with minimal overhead. Using this foundation, we describe how new implementations, using scalar, SSE, and AVX2 instruction sets, are realized. Finally we discuss benchmark results, showing significant improvement in parity generation and data reconstruction throughput. Implemented routines were made available, and are now part of *ZFS on Linux* project.

Chapter 7

ECCFS: A Middleware for Parallel File System Reliability

The traditional way of providing reliability in a data center is by using replication of data. However, ever increasing data volumes has motivated research in using more storage-efficient alternatives, such as using erasure codes. We propose a new erasure code strategy that is both performance and storage-efficient, while enabling flexibility in configuration and operation. The erasure codes are calculated on a per-file basis, and stored in specifically created files within the same Parallel File System (PFS). This chapter describes a generic middleware for data reliability for use with High Performance PFS, called ECCFS. Our solution imposes no changes to existing PFS source code (e.g. Lustre), nor does it require special deployment considerations. The following section describes the motivation behind this work.

7.1 Motivation

Parallel file systems are complex distributed systems, comprised of many components. To support the ever increasing demand for capacity and throughput, the number of individual components must also increase. As a result, the storage systems must be able to cope with more frequent failures, while still providing guaranties of data reliability and durability. This implies that no data is lost or silently corrupted, regardless of the size and the nature of the failure. Given the complexity of individual PFS components, failures of differing natures are to be expected. The object storage server (OSS) is typically comprised of a computer node equipped with high capacity storage solutions. The storage backend exposes an object-based interface to the upper layers of the PFS, hence they are called Object Storage Target (OST). Failure of any OSS component usually renders the entire OSS unavailable, and possibly causes an unrecoverable loss of data. Another potential source of failures of PFS is the network interconnect. To accommodate increasing amount of data, PFSs are designed to scale horizontally. That means, instead of making the individual OSTs larger, the systems provide increase of storage by adding additional OSTs. This also entails adding more interconnect and server equipment, shifting the reliability concerns from OSTs storage technologies to the other components of the system. In this work, we present an application of erasure codes aimed to solve the problem of reliability of OSS, OST and interconnect of a typical PFS.

Usually, it is accepted for file access performance to be degraded during presence or recovery from a failure. Storage targets are built using a hardware or software type of RAID systems, where a fraction of available bandwidth is used for rebuilding. But without the presence of failures, performance must not suffer.

To design a failure-proof reliability system, we first need to understand underlying system characteristics and classify the failure domains of the whole system. In following, we use a typical Lustre installation as a prototype of a PFS. An overview of a typical Lustre installation is shown on Figure 1-7. Lustre stores files in a striped manner over OSTs, similar to RAID0 scheme. When files are created, the Metadata Server (MDS) creates a new file layout, consisting of: *stripe count*, *stripe size*, and list of pairs *<OST:OBJID>*[1]. The stripe count tells over how many OSTs the file will be distributed, and the stripe size indicates the distribution block size granularity. The client can now perform file I/O operations with assigned OSTs, but they must do so through OSS servers. This interaction pattern defines failure domains of the system:

Object Storage Target Even though they are built with reliability in mind, in this scheme, they are the single point of failure where data durability is concerned. Due to file striping, downtimes caused by component failures can leave a large number of user files unavailable.

[1]This map uniquely identifies a physical data location

Object Storage Server OSS typically controls multiple storage targets, it defines a larger failure domain. But, since OSS does not store data, it is only important for availability consideration.

Metadata Server Data integrity of Metadata Target (MTD) is of utmost importance. The size of PFS metadata allows for snapshot backup strategies, so that even in case of catastrophic failure, the system can be restored to some previously working state.

With this in mind, we propose a distributed, erasure-code based, redundancy distribution scheme, as illustrated by Figure 7-1. For each file, signified by different colors on multiple OSTs, we produced the desired amount of redundancy (striped blocks) using Reed-Solomon erasure coding. To satisfy reliability and availability constraints, we placed the redundancy block in isolated failure domains. I.e., sets of OSTs used for file data and redundancy must be disjoint. Furthermore, to improve scalability and decrease performance impact on primary MDS, reliability blocks can be stored on OSTs of a completely separate file system.

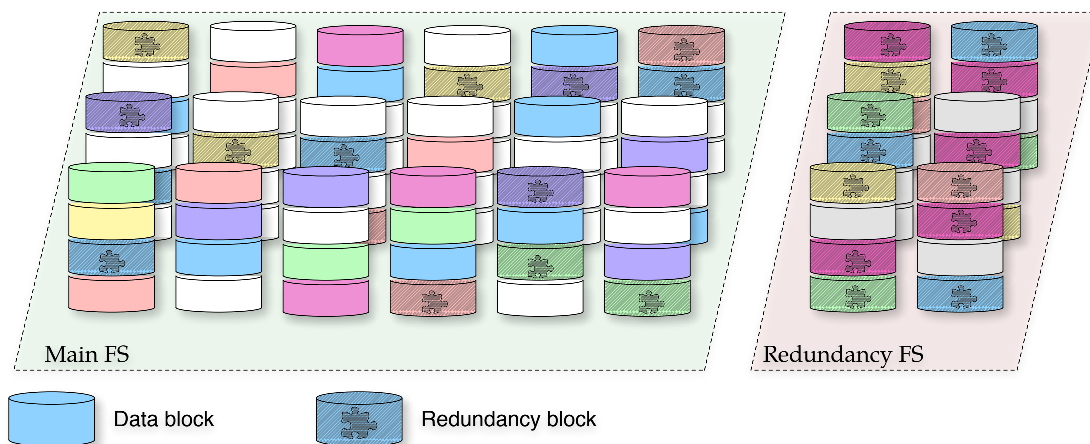


Figure 7-1: Desclustered redundancy placement

In the following, we discuss motivation and requirements for the proposed reliability scheme:

Optimal storage utilization Simple data replication requires much higher costs in storage as compared to same reliability erasure coding. Most parallel file systems utilize multiple storage servers to store data for a single file, thus reducing the reliability of the replication scheme in case of multiple failures. Using maximum distance separable (MDS) codes, like the Reed-Solomon erasure code, we ensure minimum storage utilization while providing optimal reliability, even in the case of the multiple simultaneous OST failures.

Minimizing ECC calculation overheads Newly produced data in a data center is likely to be accessed, changed and even deleted in the near future. Examples of this

are: intermediate data in batch analysis jobs, process logs, compilation auxiliary files, etc. With that in mind, we propose the introduction of an adjustable period, during which we delay calculation of ECC blocks for newly introduced files. This prevents wasting time, power and bandwidth on ECC computation for short-lived and frequently updated files.

Flexible and automatic management An important aspect of data center operation is the flexibility of the data reliability solution. With typical RAID systems, the number of parity disks is fixed which ensures the same reliability for all data. However, not all data is equally important, e.g. data produced by an experiment is more important than simulation results which can be recomputed. Thus, the ability to specify a desired level of redundancy brings added value. This process must be automated, while allowing user to specify redundancy policies for different criteria, such as: file name, file type, file owner, etc. Once in place, the automatic redundancy management system shall continuously provide the desired level of redundancy.

7.2 Design of ECCFS

In this section we describe the design of the proposed ECC distribution scheme for parallel file systems (ECCFS middleware). To minimize code calculation overhead and decrease impact on performance, we employ a delayed asynchronous method for ECC calculation. The advantages of this method are:

Temporal locality ECC calculation does not need to happen every time a new file is created or when the existing file is modified. If application overwrites the same area of the file multiple times in a short amount of time, or, if the file is temporary, ECC calculation is not performed at all.

Spatial locality We do not calculate ECC calculation on the client side, so that user application performance does not suffer. Once a file is not changed for a predefined amount of time, the ECCFS middleware calculates ECC only for the modified stripes of the file. With the knowledge of file's layout, the ECCFS can utilize only object servers that are actually holding data.

For the storage of redundancy we reuse the existing PFS. For each stripe of *data file* we calculate redundancy blocks using Reed-Solomon(n,k) block erasure coding. Corresponding redundancy blocks are then stored as separate files, in a predefined location, as shown in Figure 7-9. Layout and placement of redundancy files have to satisfy following criteria:

1. Redundancy files must have a stripe count equal to 1 to ensure that each one will be placed exactly on a single OST.

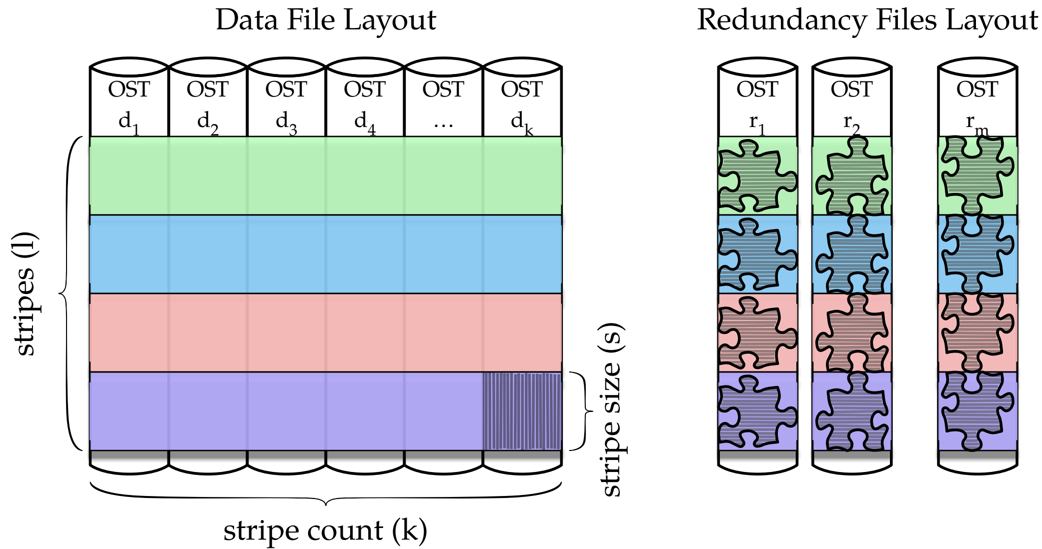


Figure 7-2: Layouts of data and redundancy files in ECCFS

2. Stripe size of the redundancy file must match that of the data file. This ensures best storage utilization.
3. Selection of OSTs for redundancy files should be on different failure domains from data file: $\{d_1, d_2, d_3, \ldots, d_k\} \cap \{r_1, r_2, \ldots, r_m\} = \emptyset$

By doing so, we do not have to keep any long-term metadata for stable and consistent data to redundancy mappings. We achieve this by implicit name mapping between data file names and redundancy file names.

Let "`/pfs/example/data.file`" be the absolute path of a data file, where "`/pfs/`" is the mount point of PFS, and "`example/data.file`" the relative path of a file in PFS global namespace. We define a directory in the top-level of PFS's namespace, "`/pfs/.eccfs/`", where only redundancy data will be stored, called *global redundancy repository*. This is enough to create bijective mapping, `DataToRedundancy`(), defined as follows:

$$\begin{aligned} \texttt{RedundancyFile}(\textit{relative_filename},\ i) = \\ \texttt{PathConcatenate}(&\textit{eccfs_mount_point}, \\ &\textit{global_eccfs_repository}, \\ &\textit{relative_filename},\ \text{“}\texttt{eccfs.}\text{“},\ i) \end{aligned} \tag{7.1}$$

In other words, we define a namespace which will have a shadow directory for each data file. Inside this directory, we place files with redundancy blocks. In the example above, directory "`/pfs/.eccfs/example/data.file/`" will host m, $m = n - k$, redundancy files: "`eccfs.1`", ..., "`eccfs.m`".

This mapping enables a great level of deployment and operational flexibility. Component `eccfs_mount_point` can be set to point to an entirely different PFS installation, enabling storage of redundancy on separate hardware from original data. Storing each redundancy block in separate files allows complete flexibility in reliability throughout the lifetime of the data file. The number of redundancy files can be increased, providing a greater level of reliability, or decreased, e.g., in case storage capacity is needed for data files.

Overview of the ECCFS middleware and underlying PFS (Lustre) is shown in Figure 7-3. For clarity, interactions between PFS components are omitted. The ECCFS

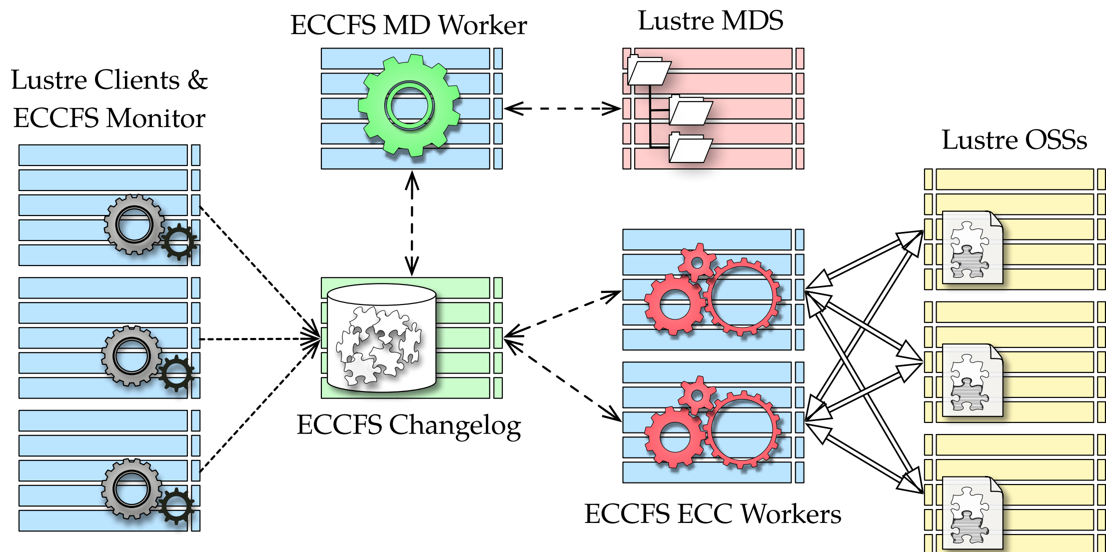


Figure 7-3: Components of ECCFS middleware in relation to Lustre file system components

middleware consists of following components: *Monitor*, *Changelog*, *MD Worker*, and *ECC Worker*. Components encapsulate specific functionality according to the *separation of concerns principle* [61][18]. Except for the MD Worker component, the entire ECCFS middleware is independent of underlying PFS, allowing simultaneous coupling with other parallel file systems that support minimum of requirements. Component separation enables greater scalability, since performance-critical components can be deployed in more than one instance. The high level of horizontal scalability of the system maximizes utilization of computation and bandwidth resources. In the following, responsibility of each component is described.

7.2.1 ECCFS Monitor

The main challenge of this ECCFS is in ensuring consistency between data and corresponding redundancy blocks. To maintain this consistency in the most efficient way,

the ECCFS middleware requires information about created and changed files. Since the variance in file sizes is large, for efficiency, we define minimum granularity to be equal to file's stripe size of the PFS. Using modern OST storage systems and fast interconnects, stripe size is usually between 1 MiB and 4 MiB. Obviously, maximal granularity is equal to file size.

The ECCFS Monitor component is responsible for collecting information about changed or created data within the PFS, using minimal update granularity. The modify update is represented by a tuple (`file_name`, `offset`, `length`), which uniquely represents the position of every change the user makes during the creation or modification of a file. For the purpose of ensuring consistency between data and redundancy namespace, the ECCFS monitors the file path-modifying operations, such as rename, move and delete. All updates are sent to the ECCFS Changelog component.

7.2.2 ECCFS Changelog

The ECCFS Changelog component serves as central database for collecting file modify and metadata updates sent from the ECCFS Monitor. This information is kept until redundancy for modified data is recalculated, or until all operations concerning redundancy repository namespace are performed. Once the system is back in a consistent state, the Changelog will be empty again. Information is kept indexed by a data file name. To reduce the data memory requirement, the Changelog component can perform compactization of information received from ECCFS Monitor if overlapping intervals are present. Also, if an update sequence for a file ends with a delete operation, all previous updates can be ignored.

7.2.3 ECCFS MD Worker

ECCFS MD Worker is in charge of metadata operations and redundancy calculation scheduling. The design of the ECCFS enables multiple instances of the MD Worker to operate simultaneously, providing higher scalability of metadata operations. The MD Worker component has the following responsibilities:

Redundancy placement The ECCFS middleware is designed to reuse the existing PFS for storage of redundancy. MD Worker must choose an adequate location for storing redundancy files, according to the set of rules we outlined previously. Each new ECC block is stored as a separate file, and it is the MD Worker's responsibility to place this file on a suitable OSS. This is needed in order to keep optimal redundancy guaranties of the erasure encoding scheme.

PFS interface In order to support multiple parallel file systems without changing their source code, the MD Worker relies on already provided facilities for low-level file manipulation. Minimum functionality needed for operation of the ECCFS middleware is retrieval of the file's layout information and ability to create a new file with specified layout information on desired OST. Additionally, information about the failed components of the PFS can be used for automation of restore operation. This is the only non-generic part of the ECCFS middleware.

ECC calculation Once a predefined amount of time since modification of a file has passed, MD Worker starts ECC calculation jobs for the file. It first retrieves data file layout from the PFS Metadata server. Next, it ensures that required redundancy files are created, and then issues the ECC calculation jobs for changed regions of the file. Since erasure code is a block code, the ECC block is mapped to full stripe of the data file. This entails that, for single data change anywhere within a file stripe, the ECC is going to be recalculated for the whole stripe. Each ECC calculation job runs either on a single, or on multiple consecutive stripes.

Housekeeping of Data-Redundancy namespace Once a data file is deleted, the MD Worker must delete all redundancy files used for the file. Similarly, when a data file is moved, all redundancy files are moved to a new location to reflect this change.

7.2.4 ECCFS ECC Worker

The ECCFS ECC Worker listens on job queue and performs the actual calculation of the error correction codes. The ECC calculation job, created by the MD Worker instance, contains file layout, desired redundancy policy, index of start stripe and number of stripes to calculate ECCs. Using this information, the ECC Worker maps the file's data to erasure block and performs the ECC calculation. Once calculated, redundancy blocks are stored into previously created redundancy files. Multiple instances of the ECC Worker component can run simultaneously, sharing code calculation jobs between them. This provides high scalability, enabling the ECCFS middleware to reach the consistent state between data and ECCs faster. Restoring files based on ECCs is also done by the ECC Worker component, in which case reconstructed data is written to a specified file. The ECC Worker instance must request only data that is known to be available during the reconstruction process. The reconstruction job, besides the file layout, also contains a map of failed OSTs, which is used to request only file stripe blocks that are available. To enable this, PFS must continue to deliver data from the OSTs not affected by the failure.

7.3 Implementation

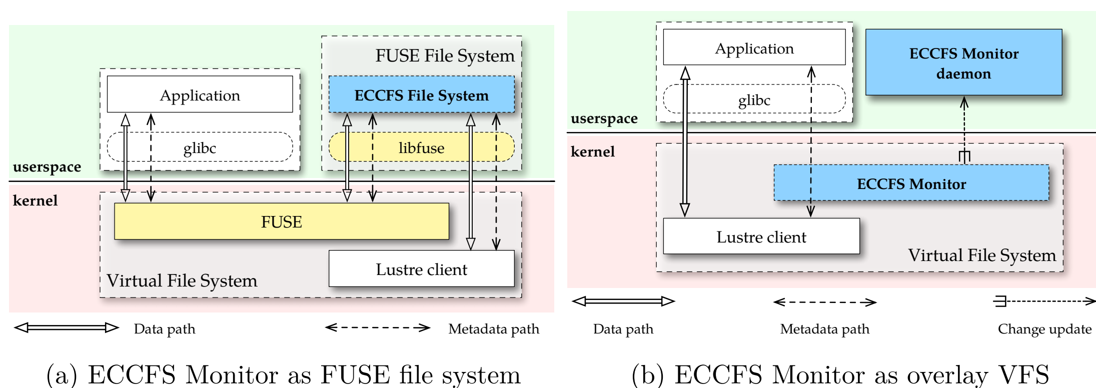
During the process of design and implementation of the ECCFS middleware we had to honor multiple constraints that come from the fact that we do not have control over

pre-existing components of the data center. ECCFS can be added to already deployed PFS, without any modification of software or hardware infrastructure that was already deployed. Another important consideration is that the ECCFS middleware must add as little overhead as possible. Here, we have to consider the amount of computational and network bandwidth resources used, as well as the impact on the data and metadata performance of PFS. The following section describes implementation of each component of the ECCFS middleware.

7.3.1 ECCFS Monitor

ECCFS Monitor is the only component that needs to be installed on each client of the PFS. As such, it is the most performance-critical component of the ECCFS middleware. The ECCFS Monitor has to intercept file system calls that applications make, and extract the changelog information. It is important not to impose any new software libraries or changes to the user application code. This means that monitoring has to happen between the application and the PFS client module. Several approaches to interpose on system calls between application and kernel exist [53][38][29], but all of them require modification of the application environment. Furthermore, enforcing the use of such a solution is a problem, because if an application circumvents the interposing solution and accesses the PFS directly, it will cause inconsistent data-redundancy mappings.

Because of this, we explored options to place the ECCFS Monitor inside the kernel, preventing undetectable access to the PFS data. This can be accomplished in two ways: by implementing the file system in user space using the FUSE [106] infrastructure, or by implementing an overlay file system layer inside the Linux kernel, similar to eCryptfs [41] or WrapFS [113]. Both approaches are shown in Figure 7-4. The benefit of the FUSE



(a) ECCFS Monitor as FUSE file system (b) ECCFS Monitor as overlay VFS

Figure 7-4: Implementation option for ECCFS Monitor component

filesystem approach, shown in Figure 7-4(a), is that no new code is added to the Linux kernel. Instead, a file system mimicking functionality is implemented in the userspace application which, in conjunction with the FUSE kernel module, presents a virtual file

system. However, this approach involves significant performance overhead due to the required kernel/userspace context switching and data copying, as noted in [7]. Since all data and metadata operations have to be performed by a Lustre kernel client module, data is copied multiple times between the application, the fuse kernel module, fuse userspace application, and the Lustre kernel module. Thus, we decided to implement the ECCFS Monitor as an overlay file system within the Linux kernel itself, as shown in Figure 7-4(b). To add as little modifications to the kernel as possible, we split the ECCFS Monitor component into two parts: a kernel file system interposing module, and a userspace *daemon*[2]. The kernel module inspects metadata operations and transmits them to the daemon via the character device interface "`/dev/eccfs`". The functionality is performed asynchronously, so that application system calls are not stalled while the update is propagated to the userspace daemon. All I/O-intensive operations are performed directly by the Lustre client module, without any overhead. The userspace daemon collects updates and transmits them periodically to Changelog. This implementation enables us to offer an unaltered file system interface with the least performance overhead. In following, we describe handling of file system operations relevant for operation of the ECCFS middleware:

`write()` This operation modifies data, and the ECCFS middleware has to intercept every invocation of this operation in the PFS. During the time PFS services a `write()` call, the ECCFS Monitor sends a *modify update* to the ECCFS Changelog component. The modify update contains a file name, start position, and length of the modified data. This process is shown in Figure 7-5. Operations of the ECCFS middleware are prefixed with ECCFS, while other operations are performed by the PFS.

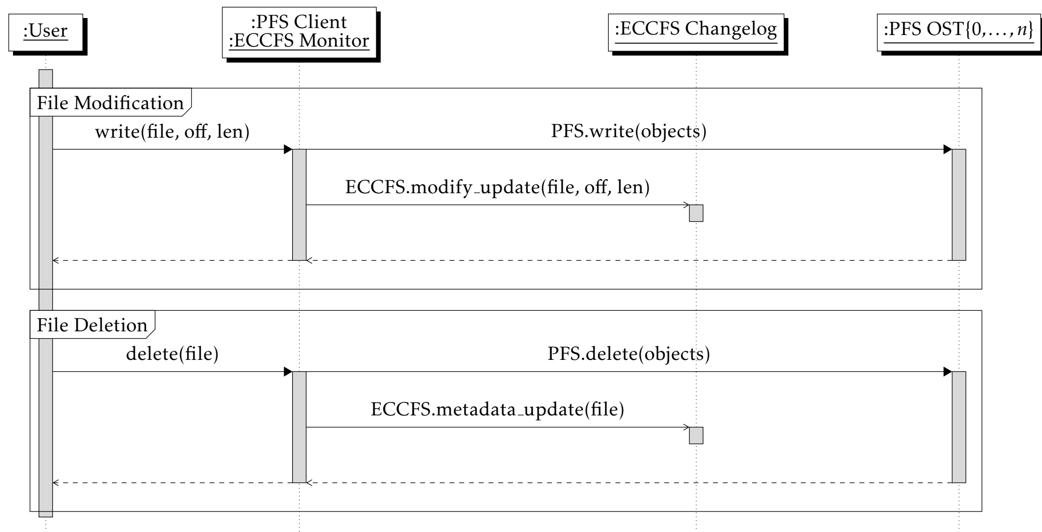


Figure 7-5: Sequence diagram of ECCFS Monitor component

[2]Background process

`move()/delete()` Moving files within the PFS namespace does not involve copying data, so the ECCFS middleware has to intercept these operations in order to ensure consistent mapping between data and the ECC files. We call these *metadata updates*. Using them, the MD Worker instance can perform housekeeping operations in the redundancy repository, and ensure consistency without having to recalculate ECC blocks.

7.3.2 ECCFS Changelog

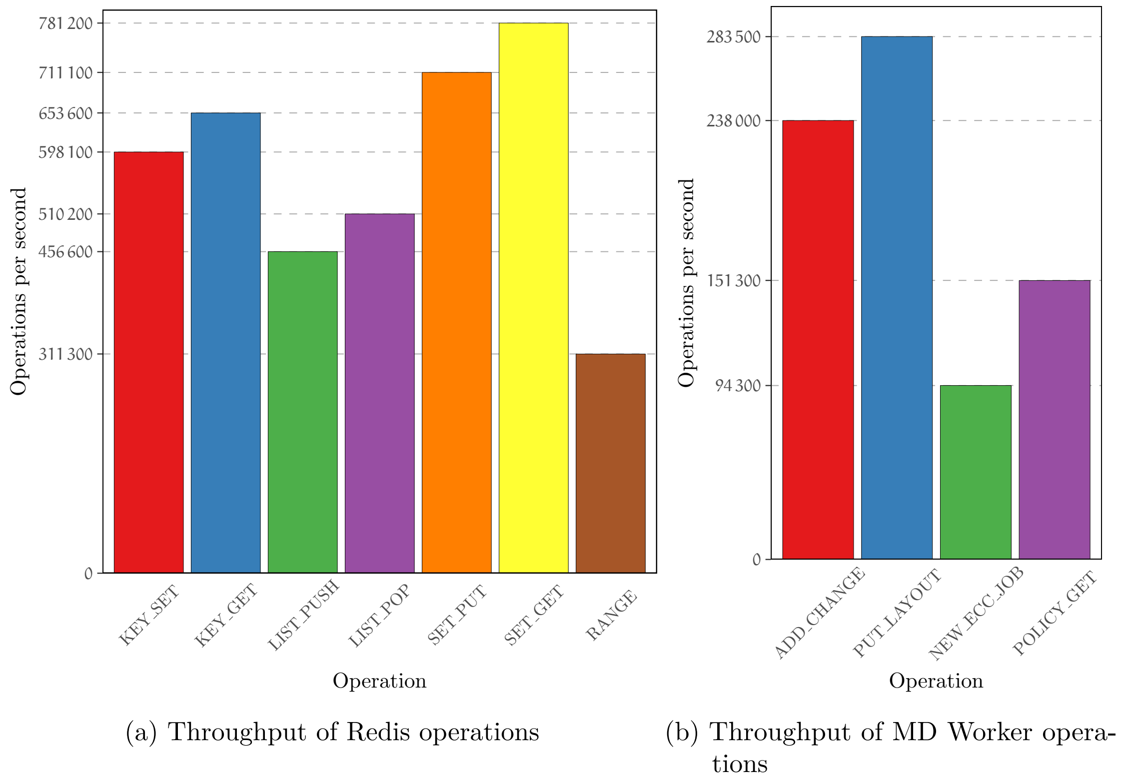
The main goal of the ECCFS Changelog component is to keep updates sent from the ECCFS Monitor, until other components finish required operations in the redundancy repository. During the implementation phase, we decided to use the component to also store configuration and redundancy policies for the system. Since the main functionality is to store change logs, a database solution, with enough scalability, can be used. We considered multiple solutions, including several SQL databases, MySQL [72], PostgreSQL [91]; and NoSQL[3] databases, like: Redis [44] and MongoDB [8].

We chose the Redis NoSQL database for the following reasons:

1. Support for a high number of simultaneous client connections [11], over 100 000, ensures scalability. The Changelog accepts a connection from every compute node that mounts a PFS and an ECCFS Monitor, and all other instances of MD Worker, and ECC Worker components.
2. Efficiency and versatility of data handling. Redis keeps its entire data set in the main memory, where each key can be an instance of: list, set, hash, or other supported data structure. This enables efficient data queries and modifications to take place on the server itself. Figure 7-6(a) shows throughput of basic Redis database operations.
3. Durability of the data set is maintained by saving snapshots of a data set to the disk, or alternatively, by master-slave replication.
4. Scripting support. Redis functionality can be extended by embedding key-values modifying procedures written in Lua programming language [49]. Scripts are compiled to native code using JIT compilation methods. This functionality enables creation of more complicated data manipulation procedures, which execute atomically on the server.

Scripting support is used extensively during implementation of the middleware, providing functionality of the *remote procedure call* (RPC) communication model. Each component of the ECCFS middleware executes procedures atomically, greatly simplifying the design of the system. Performance of the redis script procedures is shown in Figure 7-6(b). All messages are exchanged in the *JSON*[4] format. Besides the main functionality, the Changelog also implements synchronization and job queue functionality for ECCFS MD Worker and ECC Worker instances.

[3] A non-relational database engine
[4] JavaScript Object Notation



(a) Throughput of Redis operations

(b) Throughput of MD Worker operations

Figure 7-6: Performance of individual Redis operations and MD Worker operations

7.3.3 ECCFS MD Worker

Implementation of the ECCFS MD Worker component is straightforward. Following the responsibility-encapsulation approach, the MD Worker is responsible solely for metadata operations within the ECCFS middleware. Since there are no standard methods for querying and setting file layouts in PFS, the MD Worker must adapt to each underlying PFS. In our implementation, which is targeted for use with the Lustre file system, all required PFS operations can be performed using `lfs`[5] command. The MD Worker component performs two main functions:

Ensuring consistency Modify and delete updates are processed after a predefined timeout. A sequence diagram of the processing delete metadata update is shown in Figure 7-7. After an MD Worker instance receives a delete update from the ECCFS Changelog, the information is confirmed with the PFS. This usually only entails contacting the metadata server of the PFS. After confirmation, the MD Worker removes all redundancy files previously associated with the deleted data file.

Starting ECC calculations Once a predefined amount of time passes, the MD Worker component starts the process of ECC calculation for modified files. As the ECCFS

[5]Specifically, `getstripe` and `setstripe` subcommands

middleware uses a linear block erasure scheme, modify updates from Changelog have to be transformed into the information needed for ECC calculation. To do this, the MD Worker gets the file layout information from the PFS Metadata server. Once the layout of the file is known, the MD Worker ensures that appropriate redundancy files exist, or creates them if needed.

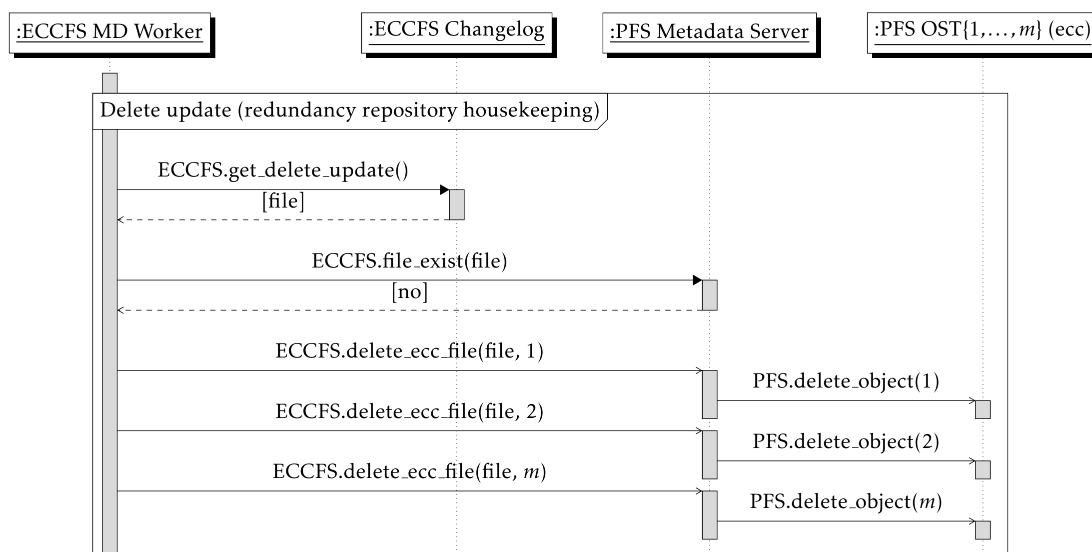


Figure 7-7: Cleaning of redundancy files performed by the ECCFS MD Worker component during processing of delete update.

7.3.4 ECCFS ECC Worker

The job of this component is to perform erasure coding operations. Once an instance of MD Worker posts a redundancy job, an idle instance of ECC Worker is notified. In order to achieve better I/O throughput, a single ECC job can be defined on a single or multiple consecutive file stripes. To allow parallel ECC operations on large files, a maximum limit of stripes can be configured. The ECCFS middleware does not dictate the usage of specific erasure code implementation. Instead, we defined a simple interface that allows different erasure code implementations to be used. The only requirement is that used code is an MDS block-based linear erasure code. ECC calculation is not implemented within the ECC Worker itself, but rather as a standalone application. Calculation of redundancy, for a single stripe of modified file is shown in Figure 7-8.

Once the MD Worker queries all necessary information from PFS, it posts a new ECC job, using functionality implemented in the Changelog. An idle ECC Worker instance receives information about the file layout, region of the file for which ECC is calculated, and parameters of the erasure scheme. To produce ECC, required file data is read using standard POSIX file system interface, and ECC is calculated. Finally, each

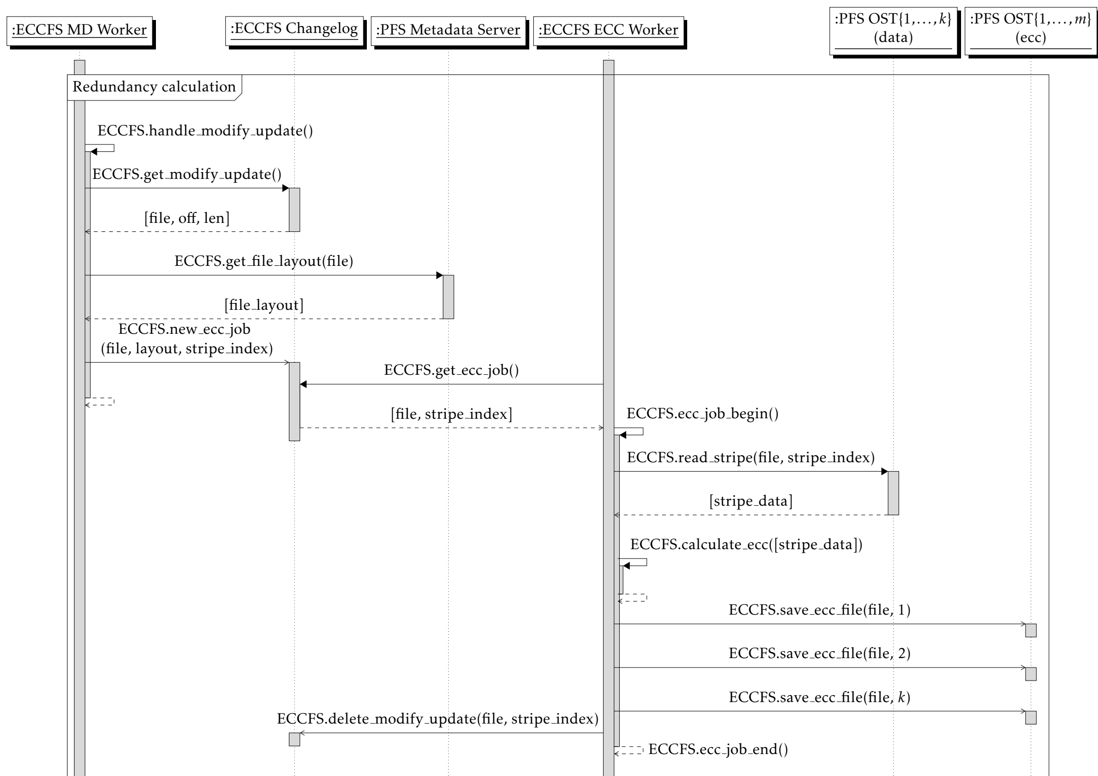


Figure 7-8: ECC calculation performed by the ECCFS ECC Worker component

of the redundancy files is updated, and the job is deleted in the Changelog, marking completion of the operation. The ECCFS allows flexible redundancies policies to be defined by the file system administrators and ordinary users. An redundancy policy defines the number of redundancy blocks and priority, and can be applied per user or by matching against the file path. Also, the time before parity calculation starts after modification is performed can be configured separately, adapting to the use-case scenario.

To provide additional data protection, the ECC Worker also calculates and stores algebraic signatures for data and redundancy blocks. The benefits of this kind of checksumming are described in section 2.4. Signatures for all data blocks and a signature of the ECC block are saved to all corresponding ECC files. The structure of the redundancy files is shown in Figure 7-9. The header of each file is updated on each new redundancy update job. Information stored in headers is used during reconstruction of data allowing original file information to be correctly restored. Algebraic signatures can be used to verify a file´s reconstructed data, but also to check whether available redundancy is consistent with data, using the property given by Equation 2.37. This prevents redundancy-consistency errors, commonly known as *RAID write hole*, from causing invalid data on reconstruction.

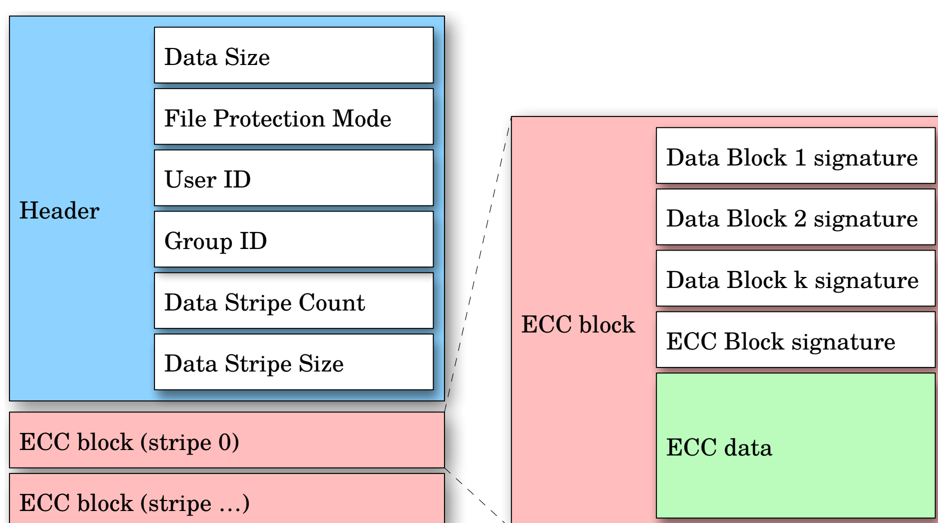


Figure 7-9: Structure of ECC files

7.3.5 Deployment and Administration

Noncommercial data centers, as the ones found in academia and research environments, usually operate several generations of compute and storage clusters. The ECCFS middleware was designed with deployment and operation in such complex environments in mind. As previously noted, ECCFS can be deployed on already existing PFS. Redundancy is first calculated for pre-existing files, after which time ECCFS starts its normal operation. A single ECCFS deployment infrastructure can be virtualized to service multiple parallel filesystems. This feature decreases capital and operational costs for system deployment and administration. Each of the sub-instances have separate configuration and reliability parameters. Finally, a deployment using two or more file systems is possible, where a parity distribution scheme is applied over all available OSTs. This scheme can be useful in environments with storage systems of different generations, reliability features, and failure domains. E.g., a file system with older and less reliable hardware can be used for redundancy repository of other file systems. Even though reliability of individual OSTs is lower, appropriately configured erasure code can still provide an additional level of data durability to a combined system.

ECCFS middleware provides an administration tool for performing various operations. The tool can be used to recalculate file redundancy or restore a file that has been affected by a failure. Additionally, the administrator can schedule a data *scrub* operation, during which ECCFS uses checksums and redundancy blocks to check for and repair silent data corruption. An administrator command is required to perform rebuilding of all affected files in case of failure or prolonged unavailability of an OST, as this operation can lead to excessive storage and computing resource utilization. The rate at which the ECC Worker consumes I/O bandwidth can be adjusted so that application throughput is not affected.

Redundancy policies can be adjusted at a later time, either to increase or decrease the level of reliability. This can be beneficial if reliability criteria change or if more free space is required in the PFS. In case of decreasing, redundancy files with the highest index are deleted, while the remaining files still make a valid erasure scheme. The redundancy policy can specify any number of ECC files, including 0, in which case all matching files are ignored by the ECCFS. If a data file is not striped over multiple OSTs, i.e. its stripe count is 1, ECCFS will use the replication strategy, since the erasure coding scheme requires at least two data blocks.

7.4 Summary

Parallel file systems, commonly used in high performance computing environments, lack flexibility in defining data resiliency. Individual object storage targets, created using hardware or software RAID solutions, define rigid reliability domains. File striping patterns, used by PFS to provide better I/O performance, significantly decrease data reliability while expanding the magnitude of data loss caused by a failure of a single storage OST. In this chapter, we described the design and prototype implementation of an adaptable erasure scheme, tailored for use in conjunction with existing parallel file systems. The proposed method, the ECCFS middleware, is designed with complex deployment and operational environments in mind. The middleware imposes no requirements or modifications to existing applications or infrastructure, allowing for the highest level of compatibility. The utilized block erasure code scheme maps to the PFS striping pattern, offering optimal storage, computation, and network resource utilization, while providing a high level of reliability.

The distributed nature of ECCFS middleware enables a high level of scalability and performance. Erasure code operations are orchestrated and performed concurrently by the desired number of worker components. The erasure code implementation, implemented for SIMD execution units of modern CPUs, described in chapter 5 is capable of producing redundancy at a high data rate. Upon installation of the ECCFS middleware on a file system, redundancy calculation of pre-existing files is shared between all instances of the ECC Worker component, maximizing available I/O bandwidth. Similarly, reconstructing all files that are affected by an OST failure is performed in parallel, improving availability of files.

Erasure coding, in some form, is supported by several parallel filesystems. GlusterFS [45] supports a non-systematic form of erasure codes to build storage volumes. Ceph [109] supports erasure coding for pool resiliency, but a default method is replication. It allows creation of fixed erasure code during storage pool creation, and does not support changing the configuration afterwards. Recently, RozoFS [84] parallel filesystems was introduced, utilizing a new class of non-MDS erasure codes built upon Mojette transform [35]. However, all of these implementations lack flexibility in data resiliency policies.

Once the number of redundancy blocks is set, all data is encoded with the same policy. In contrast, the ECCFS support flexible redundancy policies adaptable to different user or application use-cases.

We envision many possibilities for further enhancements of the presented middleware. Many scientific data environments operate on the so-called, "write once read many" datasets. The ECCFS can be extended to handle such files more efficiently, since they are guaranteed not to change in the future. For this purpose, we could provide an special class of resiliency policy, one that would perform more frequent data scrubs. Another approach would be to check if files are marked "read-only" or "immutable". In a multi-user data center environment, there is always a possibility that some data will be duplicated, intentionally or not. ECCFS cannot perform de-duplication of data files, but it can detect it and store the redundancy of such files only once. To achieve this, a database of all data blocks would have to be maintained. However, savings on storage space might not be significant in general cases.

After the file modification has been made, the ECCFS delays computation of redundancy for a configurable amount of time. Accessing modified data at the same time as the application is not advisable due to cache coherency and access serialization models of the underlying PFS. If the PFS does not provide consistent, POSIX like, read/write serialization and coherent caching across all clients, an ECC Worker instance could read stale data. An example of such a PFS is the BeeGFS[6] [6] file system. On the other side, if PFS does provide full cache coherency, concurrent access would cause significant performance degradation, due to locking and forced cache flushing overheads. However, by integrating more deeply with the PFS Metadata server, the ECCFS could receive more timely information when a client releases file locks and performs a cache write-back operation. This would shorten the vulnerable time, during which data and redundancy blocks are inconsistent. Also, the possibility to update, or rewrite, file layouts would greatly benefit file reconstruction speed and efficiency. Presently, to reconstruct a file using an erasure (n, k) configuration, ECCFS has to read k blocks, reconstruct missing data parts, and write the k data block into a new file. This is suboptimal, because only failed stripe sections need to be saved, since other stripe blocks are not affected. To support this, MDS would need to allocate a new object on available OST, and update the file layout replacing the failed OST. Then, the ECCFS would only need to populate the missing stripe block during the reconstruction process, greatly reducing write bandwidth utilization to the rest of PFS.

To provide an even greater level of data integrity and availability, the ECCFS middleware could be integrated with job scheduling platforms. In operation without failures, before a new job is scheduled, the ECCFS could perform a data integrity check on all files required by a job execution. Otherwise, all unavailable files could be reconstructed in time for job execution. Finally, more options for reliability repository could be added

[6]Formerly known as Fraunhofer GFS (FhGFS)

to enable an even greater level of customization. Currently, the ECCFS framework stores redundancy in ordinary files of PFS, with implicit naming scheme. This functionality could be expanded to support object storage, such as OpenStack swift [105]. Also, storing redundancy of read-only files on the MAID[7] type of nearline storage would enable greater storage density and lower power and cooling requirements.

[7] Massive Array of Idle Drives

Chapter 8

Summary

Reliability aspects of individual components are important factors when estimating reliability of a complex system. High capacity storage components, mainly hard disk drives, exhibit a set of failure modes which have to be well understood. Disparity between capacity and reliability of hard disk drives is making standard RAID levels unsuitable for modern high capacity storage solutions. This prompted the use of more resilient erasure coding. Furthermore, in order to prevent data corruption, checksumming and online data integrity verification is required. Software-defined reliability requires less capital and operational costs, when compared to specialized hardware reliability solutions. Parallel distributed file systems, used for efficient data handling at petabyte scales, lack fine-grained and user defined reliability. These problems provided motivation for an efficient, flexible and retargetable Reed-Solomon erasure scheme.

Reed-Solomon codes are ubiquitous for all areas of digital data storage and transmission. Implementation of efficient Reed-Solomon codes hinges on the ability to perform fast Galois field operations. The Addition operation is equivalent to *exclusive-or* (carry-less addition), and therefore can be performed efficiently. But CPUs do not have Galois field multiplication realized in hardware, so this operation has to be emulated. A method that uses polynomial multiplication and reduction to obtain a multiplication result is presented. Galois field operations are then combined with a Vandermonde-based, systematic

Reed-Solomon generator matrix formulation to form the basis of codes discussed in the rest of this thesis. The proposed implementation of Reed-Solomon erasure codes utilizes the JIT code generation technique.

Galois field operations are represented as simple, integer based, operations using LLVM IR language. This formulation allows for arbitrary Galois field lengths, supported by arbitrary integer precision of LLVM IR. Reed-Solomon is constructed by emitting all operations of matrix multiplication to a single, control-sequence free, block of instructions. Intrinsic redundancy in amalgamated operations is removed automatically by an optimizing compiler. Data level parallelism of Reed-Solomon codes is exploited by substituting LLVM IR integer types with vector types suitable for an individual test platform. This enabled seamless generation of efficient, high throughput, vectorized machine codes for both x86 and ARM platforms. The LLVM is able to generate code for SSE and AVX2 instruction sets for x86, and the NEON instruction set of the ARM CPU-based platform. Evaluation shows maximum encoding throughput, for a smaller number of data and code symbols, to be close to the memory bandwidth of test platforms. However, computational complexity of larger matrix multiplication is a dominant factor for large codes.

The ZFS file system is being increasingly used in high performance storage environments for its software-defined reliability features. Having no dependencies on hardware RAID solutions, ZFS simplifies hardware infrastructure, while relying on CPU for data redundancy and integrity calculations. Typically, a single server controls several ZFS pools, putting even greater stress on the CPU and I/O sub-system for parity operations. ZFS supports the erasure coding scheme for the storage pool, RAID-Z, which can add resiliency for up to 3 failures. Throughput of parity operations can be greatly improved by implementing specialized methods for all parity operations, and by performing computationally intensive parity operations using SIMD units of modern CPUs. A generic framework for RAID-Z parity operations, which supports implementation for multiple instruction sets with minimal overhead, is presented. Using this foundation, the new RAID-Z implementations using scalar, SSE, and AVX2 instruction sets, are realized. Optimized SIMD code is showing significant improvement in parity generation and data reconstruction throughput (up to 95x). Implemented routines were made available, and are now part of the *ZFS on Linux* project.

Parallel file systems, commonly used in high performance computing environments, lack flexibility in defining data resiliency. Individual object storage targets, created using hardware or software RAID solutions, define rigid reliability domains. Erasure coding, in some form, is supported by several parallel filesystems. However, all of these implementations lack flexibility in data resiliency policies.

This thesis presents a design and prototype implementation of an adaptable erasure scheme tailored for use in conjunction with existing parallel file systems. The proposed system, the ECCFS middleware, is designed with complex deployment and operational environments in mind, imposing no changes or requirements to the hardware or software

stacks. The block erasure code scheme enables optimal storage, computation, and network resource utilization, while providing a high-level of data resiliency. The distributed nature of ECCFS middleware enables a high level of scalability. Erasure code operations are orchestrated and performed concurrently, efficiently utilizing available computation resources and I/O bandwidth. Randomization of redundancy block placement allows parallel reconstruction of all files that are affected by a storage target failure. The ECCFS support flexible redundancy policies, adaptable to different users or application use-cases.

We envision many possibilities for further enhancements of the middleware presented. ECCFS can be extended to handle "write once read many" datasets more efficiently, and to provide stronger assurance against silent data corruption. To avoid a decrease in client I/O and metadata performance, due to distributed file locking and cache coherency of PFS, the ECCFS implements delayed parity synchronization. In order to decrease the time where modified files are vulnerable to failures, we would need to integrate deeper with PFS. Additionally, ability to update, or rewrite, file layouts would greatly benefit file reconstruction speed and efficiency. The Metadata server of PFS would only have to allocate a new object on the available OST, and update the file layout replacing the failed OST. Then, ECCFS would only need to populate the missing stripe block during the reconstruction process, greatly reducing write bandwidth utilization. To provide an even greater level of data integrity and availability, the ECCFS middleware could be integrated with job scheduling platforms. In operation without failures, before a new job is scheduled, ECCFS could perform a data integrity check on all files required by a job execution. Otherwise, all unavailable files could be reconstructed in time for job execution, or alternatively, the job scheduler would be notified about unavailable files so that a new execution plan can be generated. Finally, more options for reliability repository could be added to enable an even greater level of customization. The latest roadmap [19] for the Lustre[1] parallel file system indicates that a set of reliability features, similar to the ECCFS middleware, is being investigated.

[1] Lustre version 2.11+

References

[1] H Peter Anvin. *The mathematics of RAID-6*. 2007.

[2] Raja Appuswamy, David C van Moolenbroek, and Andrew S Tanenbaum. "Block-level RAID Is Dead." In: *HotStorage*. 2010.

[3] Lakshmi N Bairavasundaram et al. "An analysis of data corruption in the storage stack." In: *ACM Transactions on Storage (TOS)* 4.3 (2008), p. 8.

[4] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor." In: *Crypto*. Vol. 86. Springer. 1986, pp. 311–323.

[5] Kenneth G Beauchamp. *Applications of Walsh and related functions, with an introduction to sequency theory*. Vol. 2. Academic Pr, 1984.

[6] *BeeGFS, The Parallel Cluster File System*. 2016. URL: `http://www.beegfs.com`.

[7] John Bent et al. "PLFS: a checkpoint filesystem for parallel applications." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 21.

[8] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. "MongoDB vs Oracle-Database Comparison." In: *EIDWT*. 2012, pp. 330–335.

[9] R.C. Bose and D.K. Ray-Chaudhuri. *On A Class of Error Correcting Binary Group Codes*. 1960. DOI: `10.1016/S0019-9958(60)90870-6`.

[10] Peter J Braam et al. "The Lustre storage architecture." In: (2004).

[11] Rick Cattell. "Scalable SQL and NoSQL data stores." In: *Acm Sigmod Record* 39.4 (2011), pp. 12–27.

[12] Lucian Codrescu et al. "Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications." In: *Hot Chips*. Vol. 25. 2013.

[13] John Cook, Robert Primmer, and Ab de Kwant. "Comparing cost and performance of replication and erasure coding." In: *arXiv preprint arXiv:1308.1887* (2013).

[14] ARM Cortex. "A8 technical reference manual." In: *Revision: r3p2, May* (2010).

[15] Mark Cox et al. *The openssl project*. 2002.

[16] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael." In: (1999).

[17] Veera Deenadhayalan et al. "Matrix methods for lost data reconstruction in erasure codes." In: *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*. 2005, pp. 183–196.

[18] Edsger W Dijkstra. "On the role of scientific thought." In: *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[19] Andreas Dilger. "Lustre 2.9 and Beyond." 2016.

[20] Ulrich Drepper. "What every programmer should know about memory." In: *Red Hat, Inc* 11 (2007), p. 2007.

[21] Dell Storage Engineering. *Dell PS Series Storage: Choosing a Member RAID Policy*. Tech. rep. Jan. 2016. URL: `http://en.community.dell.com/dell-groups/dtcmedia/m/mediagallery/19861480/download.aspx`.

[22] David Fiala et al. "Detection and correction of silent data corruption for large-scale high-performance computing." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 78.

[23] Michael J Flynn. "Some computer organizations and their effectiveness." In: *Computers, IEEE Transactions on* 100.9 (1972), pp. 948–960.

[24] Agner Fog. "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs." In: *Copenhagen University College of Engineering* (2016).

[25] Daniel Ford et al. "Availability in Globally Distributed Storage Systems." In: *OSDI*. 2010, pp. 61–74.

[26] G David Forney and G David Forney. *Concatenated codes*. Vol. 11. Citeseer, 1966.

[27] Inc. Freescale Semiconductor. "Chip Errata for the i.MX 6Dual/6Quad and i.MX 6DualPlus/6QuadPlus." In: (2014).

[28] Robert G Gallager. "Low-density parity-check codes." In: *Information Theory, IRE Transactions on* 8.1 (1962), pp. 21–28.

[29] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. "Ostia: A Delegating Architecture for Secure System Call Interposition." In: *NDSS*. 2004.

[30] *GCC:GNU Compiler Collection*. 2016. URL: `https://gcc.gnu.org`.

[31] John Goodacre and Andrew N Sloss. "Parallelism and the ARM instruction set architecture." In: *Computer* 38.7 (2005), pp. 42–50.

[32] Jim Gray and Catharine Van Ingen. "Empirical measurements of disk failure rates and error rates." In: *arXiv preprint cs/0701166* (2007).

[33] Kevin M Greenan. *Reliability and power-efficiency in erasure-coded storage systems*. Citeseer, 2011.

[34] Kevin M Greenan, Ethan L Miller, and Thomas JE Schwarz. "Optimizing Galois Field arithmetic for diverse processor architectures and applications." In: *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–10.

[35] Jeanpierre V Guedon and Nicolas Normand. "Mojette transform: applications for image analysis and coding." In: *Electronic Imaging'97*. International Society for Optics and Photonics. 1997, pp. 873–884.

[36] Michael E. Kounavis(Intel) Gueron, Shay(Intel). *Intel ® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. Tech. rep. Intel, 2010, pp. 1–72.

[37] Shay Gueron and Michael E Kounavis. "Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode." In: *Intel white paper (September 2012)* (2010).

[38] Philip J Guo and Dawson R Engler. "CDE: Using System Call Interposition to Automatically Create Portable Software Packages." In: *USENIX Annual Technical Conference*. 2011.

[39] Preeti Gupta et al. "An economic perspective of disk vs. flash media in archival storage." In: *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*. IEEE. 2014, pp. 249–254.

[40] James Lee Hafner and K Rao. "Notes on reliability models for non-MDS erasure codes." In: *IBM Res. rep. RJ10391* (2006).

[41] Michael Austin Halcrow. "eCryptfs: An enterprise-class encrypted filesystem for linux." In: *Proceedings of the 2005 Linux Symposium*. Vol. 1. 2005, pp. 201–218.

[42] Per Hammarlund et al. "Haswell: The fourth-generation intel core processor." In: *IEEE Micro* 2 (2014), pp. 6–20.

[43] Richard W Hamming. "Error detecting and error correcting codes." In: *Bell System technical journal* 29.2 (1950), pp. 147–160.

[44] Jing Han et al. "Survey on NoSQL database." In: *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE. 2011, pp. 363–366.

[45] Red Hat. *GlusterFS*. 2012.

[46] *HGST Hard Drives*. 2016. URL: `https://www.hgst.com/products/hard-drives`.

[47] Martin Hilbert and Priscila López. "The world's technological capacity to store, communicate, and compute information." In: *science* 332.6025 (2011), pp. 60–65.

[48] Cheng Huang et al. "Erasure coding in windows azure storage." In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 15–26.

[49] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. "Lua-an extensible extension language." In: *Softw., Pract. Exper.* 26.6 (1996), pp. 635–652.

[50] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-032. Jan. 2016.

[51] Intel Intel. "and IA-32 Architectures Software Developer's Manual." In: *Volume 3A: System Programming Guide, Part* 1 (2010).

[52] Intel Intel. "Optimizing Performance with Intel® Advanced Vector Extensions." In: (2014).

[53] Kapil Jain and R Sekar. "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement." In: *NDSS*. 2000.

[54] Weihang Jiang et al. "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics." In: *ACM Transactions on Storage (TOS)* 4.3 (2008), p. 7.

[55] Sebastian Kalcher. "An erasure-resilient and compute-efficient coding scheme for storage applications." PhD thesis. Goethe University Frankfurt am Main, 2013. URL: `http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/32295`.

[56] Sebastian Kalcher and Volker Lindenstruth. "Accelerating Galois Field Arithmetic for Reed-Solomon Erasure Codes in Storage Applications." In: *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. 2011, pp. 290–298. DOI: `10.1109/CLUSTER.2011.40`. URL: `http://dx.doi.org/10.1109/CLUSTER.2011.40`.

[57] Sooyong Kang et al. "Performance trade-offs in using nvram write buffer for flash memory-based storage devices." In: *Computers, IEEE Transactions on* 58.6 (2009), pp. 744–758.

[58] Neal Koblitz. "Elliptic curve cryptosystems." In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[59] Matthias Kretz and Volker Lindenstruth. "Vc: A C++ library for explicit vectorization." In: *Softw., Pract. Exper.* 42.11 (2012), pp. 1409–1430. DOI: `10.1002/spe.1149`. URL: `http://dx.doi.org/10.1002/spe.1149`.

[60] Andrew Krioukov et al. "Parity Lost and Parity Regained." In: *FAST*. Vol. 8. 2008, pp. 127–141.

[61] Philip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

[62] Chris Lattner. "LLVM and Clang: Next generation compiler technology." In: *The BSD Conference*. 2008, pp. 1–2.

[63] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[64] Adam Leventhal. "Flash storage memory." In: *Communications of the ACM* 51.7 (2008), pp. 47–51.

[65] Adam Leventhal. "Triple-parity RAID and beyond." In: *Queue* 7.11 (2009), p. 30.

[66] Witold Litwin and Thomas Schwarz. "Algebraic signatures for scalable distributed data structures." In: *Data Engineering, 2004. Proceedings. 20th International Conference on*. IEEE. 2004, pp. 412–423.

[67] David McGrew and John Viega. "The Galois/counter mode of operation (GCM)." In: *Submission to NIST* (2004).

[68] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.

[69] Ralph C Merkle. "A digital signature based on a conventional encryption function." In: *Advances in Cryptology—CRYPTO'87*. Springer. 1987, pp. 369–378.

[70] Ralph C Merkle. *Method of providing digital signatures*. US Patent 4,309,569. Jan. 1982.

[71] Daniel Molka et al. "Memory performance and cache coherency effects on an intel nehalem multiprocessor system." In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE. 2009, pp. 261–270.

[72] *MySQL Satabase Server*. 2016. URL: http://www.mysql.com.

[73] A. Spitzbart N. Macon. "Inverses of Vandermonde Matrices." In: *The American Mathematical Monthly* 65.2 (1958), pp. 95–100. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2308881.

[74] David Nagle et al. "The ANSI T10 object-based storage standard and current implementations." In: *IBM Journal of Research and Development* 52.4.5 (2008), pp. 401–411.

[75] *One Billion Drive Hours and Counting: Q1 2016 Hard Drive Stats*. 2016. URL: www.backblaze.com/blog/hard-drive-reliability-stats-q1-2016/.

[76] *OpenSSL:Cryptography and SSL/TLS Toolkit*. 2016. URL: https://www.openssl.org.

[77] Christof Paar. "A new architecture for a parallel finite field multiplier with low complexity based on composite fields." In: *Computers, IEEE Transactions on* 45.7 (1996), pp. 856–861.

[78] Christof Paar, Peter Fleischmann, and P Roeise. “Efficient multiplier architectures for Galois Fields GF (2 4n).” In: *Computers, IEEE Transactions on* 47.2 (1998), pp. 162–170.

[79] Bernd Panzer-Steindel. “Data integrity.” In: *CERN/IT* (2007).

[80] David A Patterson. “Latency lags bandwith.” In: *Communications of the ACM* 47.10 (2004), pp. 71–75.

[81] David A Patterson et al. “A Simple Way to Estimate the Cost of Downtime.” In: *LISA*. Vol. 2. 2002, pp. 185–188.

[82] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*. Vol. 17. 3. ACM, 1988.

[83] Tony Pearson. “Correct use of the term Nearline.” In: *IBM Developerworks, Inside System Storage* (Oct. 2010).

[84] Dimitri Pertin et al. “Distributed File System Based on Erasure Coding for I/O-Intensive Applications.” In: *CLOSER*. 2014, pp. 451–456.

[85] William Wesley Peterson and Daniel T Brown. “Cyclic codes for error detection.” In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[86] Eduardo Pinheiro, Wolf-dietrich Weber, and Luiz Andr. “Failure Trends in a Large Disk Drive Population.” In: February (2007).

[87] J. S. Plank. *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems*. Tech. rep. CS-96-332. University of Tennessee, July 1996.

[88] J. S. Plank and Y. Ding. *Note: Correction to the 1997 Tutorial on Reed-Solomon Coding*. Tech. rep. CS-03-504. University of Tennessee, Apr. 2003.

[89] James Plank, Kevin Greenan, and Ethan L. Miller. “Screaming Fast Galois Field Arithmetic Using Intel SIMD Extensions.” In: *Proceedings of the 11th Conference on File and Storage Systems (FAST 2013)*. Feb. 2013.

[90] JS Plank et al. “GF-Complete: A comprehensive open source library for Galois Field arithmetic.” In: *University of Tennessee, Tech. Rep. UT-CS-13-703* (2013).

[91] *PostgreSQL*. 2016. URL: http://www.postgresql.org.

[92] Vijayan Prabhakaran et al. *IRON file systems*. Vol. 39. 5. ACM, 2005.

[93] Michael O Rabin et al. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[94] KK Rao, James Lee Hafner, and Richard A Golding. “Reliability for networked storage nodes.” In: *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE. 2006, pp. 237–248.

[95] Irving S Reed and Gustave Solomon. “Polynomial codes over certain finite fields.” In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[96] David Rohr. "On development, feasibility, and limits of highly efficient CPU and GPU programs in several fields - fast parallel SIMDized GPU-accelerated reed-solomon encoding, heterogeneous linpack benchmark, and event reconstruction for the ALICE experiment." PhD thesis. Universität Frankfurt a. M., 2013. URL: http://publikationen.ub.uni-frankfurt.de/files/34377/dissertation_david_rohr.pdf.

[97] Philip E Ross. "5 Commandments [technology laws and rules of thumb]." In: *Spectrum, IEEE* 40.12 (2003), pp. 30–35.

[98] Jeffrey Rott. *Intel advanced encryption standard instructions (aes-ni)*. Tech. rep. Technical report, Intel, 2010.

[99] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.2.0)*. http://www.sagemath.org. 2016.

[100] B. Schroeder and G.A. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *IEEE Trans. Dependable Secur. Comput.* 7 (2010). ISSN: 1545-5971. DOI: 10.1109/TDSC.2009.4.

[101] Thomas JE Schwarz. "Verification of Parity Data in Large Scale Storage Systems." In: *PDPTA*. 2004, pp. 508–514.

[102] *Seagate Internal Drives*. 2016. URL: http://www.seagate.com/internal-hard-drives/.

[103] Sandeep Shah and Jon G Elerath. "Reliability analysis of disk drive failure mechanisms." In: *Proceedings of the Annual Symposium on Reliability and Maintainability*. Citeseer. 2005, pp. 226–231.

[104] Claude Elwood Shannon. "A mathematical theory of communication." In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), pp. 3–55.

[105] *SWIFT Object Storage*. 2016. URL: https://www.openstack.org.

[106] Miklos Szeredi et al. "Fuse: Filesystem in userspace." In: *Accessed on* (2010).

[107] Feiyi Wang et al. "Understanding lustre filesystem internals." In: *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* (2009).

[108] *WD Internal Hard Drive Overview*. 2016. URL: http://www.wdc.com/en/products/internal/.

[109] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[110] Willis Whittington. *Desktop, nearline and enterprise disk drives,-delta by design*. 2007.

[111] Ryan William and Lin Shu. *Channel Codes: classical and modern*. Cambridge University Press, 2009, p. 708. ISBN: 978-0-521-84868-8.

[112] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious." In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[113] Erez Zadok and Ion Badulescu. "A stackable file system interface for Linux." In: *LinuxExpo Conference Proceedings*. Vol. 94. 1999, p. 10.

[114] ZFS on Linux. *ZFS on Linux*. Version 0.6.5. Apr. 19, 2016. URL: `https://github.com/zfsonlinux/zfs`.

Chapter A

Appendix

A Test-bed platforms

Component	Specification	
CPU		
Xeon E5-2660v3	micro architecture	Haswell
	clock speed	2.6 GHz
	# of cores	10
	# of threads	20
	L1 data cache	32 KiB
	L2 data cache	256 KiB
	L3 shared cache	25 MiB
	Max Memory Bandwidth	68 GiB s^{-1}
	instruction sets	x86_64, SSE, AVX2
	TDP	105 W
Main memory		
16× SAMSUNG M386A4G40DM0-CPB	type	DDR4 + ECC
	capacity	512 GiB
	clock speed	2133 MHz
	bus width	64 B
	CAS latency	15
Software		
	Linux Kernel	3.16.7
	gcc	4.8.3
	LLVM	3.4
	go	1.6
	redis	3.0

Table A-1: Intel CPU platform testbed

Component	Specification	
CPU		
NXP i.MX6 Quad	micro architecture	Cortex-A9
	clock speed	1 GHz
	# of cores	4
	L1 data cache	32 KiB
	L2 shared cache	1 MiB
	instruction sets	32 bit arm, NEON
	TDP	3 W
Main memory		
	type	DDR3
	capacity	2 GiB
	clock speed	1066 MHz
	bus width	64 B
Software		
	Linux Kernel	3.10.17
	gcc	4.6.1
	LLVM	3.4

Table A-2: ARM CPU platform testbed

B Example of Hexagon DSP VLIW code

Listing A-1 Example of VLIW packets of Hexagon DSP generated from JIT realized Reed-Solomon erasure codes, using LLVM compiler. Instructions of each packet, surrounded by curly braces, execute in parallel on 4 execution units of the DSP.

```
1  {
2          r5 ^= lsr(r26, #8)
3          r2 ^= lsr(r27, #8)
4  }
5  {
6          r6 ^= lsr(r25, #8)
7          r4 ^= lsr(r17, #8)
8  }
9  {
10         r5 ^= lsr(r26, #7)
11         r4 ^= lsr(r17, #7)
12 }
13 {
14         r19 ^= lsr(r4, #1)
15         r18 ^= lsr(r9, #1)
16         r24 = r9
17 }
18 {
19         r16 ^= asl(r0, #5)
20         memh(r5+#4) = r28
21         r9 ^= asl(r23, #12)
22         memh(r5+#0) = r1
23 }
24 {
25         memh(r0+#6) = r16
26         r16 ^= lsr(r17, #11)
27         r5 ^= lsr(r15, #13)
28         r17 = and(r18, r7)
29 }
```

C Galois Field used in RAID-Z

Table A-3: Binary extended binary field GF(2^8), generated with $p(X) = X^8 + X^4 + X^3 + X^2 + 1$, used in ZFS RAID-Z erasure coding.

Representations of binary Galois Field elements			
Power	**Polynomial**	**Decimal**	**Vector**
α^{0}	1	{1}	(00000001)
α^{1}	X^1	{2}	(00000010)
α^{2}	X^2	{4}	(00000100)
α^{3}	X^3	{8}	(00001000)
α^{4}	X^4	{16}	(00010000)
α^{5}	X^5	{32}	(00100000)
α^{6}	X^6	{64}	(01000000)
α^{7}	X^7	{128}	(10000000)
α^{8}	$X^4 + X^3 + X^2 + 1$	{29}	(00011101)
α^{9}	$X^5 + X^4 + X^3 + X^1$	{58}	(00111010)
α^{10}	$X^6 + X^5 + X^4 + X^2$	{116}	(01110100)
α^{11}	$X^7 + X^6 + X^5 + X^3$	{232}	(11101000)
α^{12}	$X^7 + X^6 + X^3 + X^2 + 1$	{205}	(11001101)
α^{13}	$X^7 + X^2 + X^1 + 1$	{135}	(10000111)
α^{14}	$X^4 + X^1 + 1$	{19}	(00010011)
α^{15}	$X^5 + X^2 + X^1$	{38}	(00100110)
α^{16}	$X^6 + X^3 + X^2$	{76}	(01001100)
α^{17}	$X^7 + X^4 + X^3$	{152}	(10011000)
α^{18}	$X^5 + X^3 + X^2 + 1$	{45}	(00101101)
α^{19}	$X^6 + X^4 + X^3 + X^1$	{90}	(01011010)
α^{20}	$X^7 + X^5 + X^4 + X^2$	{180}	(10110100)
α^{21}	$X^6 + X^5 + X^4 + X^2 + 1$	{117}	(01110101)
α^{22}	$X^7 + X^6 + X^5 + X^3 + X^1$	{234}	(11101010)
α^{23}	$X^7 + X^6 + X^3 + 1$	{201}	(11001001)
α^{24}	$X^7 + X^3 + X^2 + X^1 + 1$	{143}	(10001111)
α^{25}	$X^1 + 1$	{3}	(00000011)
α^{26}	$X^2 + X^1$	{6}	(00000110)
α^{27}	$X^3 + X^2$	{12}	(00001100)
α^{28}	$X^4 + X^3$	{24}	(00011000)
α^{29}	$X^5 + X^4$	{48}	(00110000)
α^{30}	$X^6 + X^5$	{96}	(01100000)
α^{31}	$X^7 + X^6$	{192}	(11000000)
α^{32}	$X^7 + X^4 + X^3 + X^2 + 1$	{157}	(10011101)
α^{33}	$X^5 + X^2 + X^1 + 1$	{39}	(00100111)
α^{34}	$X^6 + X^3 + X^2 + X^1$	{78}	(01001110)
α^{35}	$X^7 + X^4 + X^3 + X^2$	{156}	(10011100)
α^{36}	$X^5 + X^2 + 1$	{37}	(00100101)
α^{37}	$X^6 + X^3 + X^1$	{74}	(01001010)
α^{38}	$X^7 + X^4 + X^2$	{148}	(10010100)
α^{39}	$X^5 + X^4 + X^2 + 1$	{53}	(00110101)
α^{40}	$X^6 + X^5 + X^3 + X^1$	{106}	(01101010)
α^{41}	$X^7 + X^6 + X^4 + X^2$	{212}	(11010100)
α^{42}	$X^7 + X^5 + X^4 + X^2 + 1$	{181}	(10110101)
α^{43}	$X^6 + X^5 + X^4 + X^2 + X^1 + 1$	{119}	(01110111)
α^{44}	$X^7 + X^6 + X^5 + X^3 + X^2 + X^1$	{238}	(11101110)
α^{45}	$X^7 + X^6 + 1$	{193}	(11000001)
α^{46}	$X^7 + X^4 + X^3 + X^2 + X^1 + 1$	{159}	(10011111)
α^{47}	$X^5 + X^1 + 1$	{35}	(00100011)
α^{48}	$X^6 + X^2 + X^1$	{70}	(01000110)
α^{49}	$X^7 + X^3 + X^2$	{140}	(10001100)
α^{50}	$X^2 + 1$	{5}	(00000101)
α^{51}	$X^3 + X^1$	{10}	(00001010)
α^{52}	$X^4 + X^2$	{20}	(00010100)
α^{53}	$X^5 + X^3$	{40}	(00101000)
α^{54}	$X^6 + X^4$	{80}	(01010000)
α^{55}	$X^7 + X^5$	{160}	(10100000)
α^{56}	$X^6 + X^4 + X^3 + X^2 + 1$	{93}	(01011101)
α^{57}	$X^7 + X^5 + X^4 + X^3 + X^1$	{186}	(10111010)
α^{58}	$X^6 + X^5 + X^3 + 1$	{105}	(01101001)
α^{59}	$X^7 + X^6 + X^4 + X^1$	{210}	(11010010)
α^{60}	$X^7 + X^5 + X^4 + X^3 + 1$	{185}	(10111001)
α^{61}	$X^6 + X^5 + X^3 + X^2 + X^1 + 1$	{111}	(01101111)
α^{62}	$X^7 + X^6 + X^4 + X^3 + X^2 + X^1$	{222}	(11011110)
α^{63}	$X^7 + X^5 + 1$	{161}	(10100001)
α^{64}	$X^6 + X^4 + X^3 + X^2 + X^1 + 1$	{95}	(01011111)
α^{65}	$X^7 + X^5 + X^4 + X^3 + X^2 + X^1$	{190}	(10111110)

α^{66}	$X^6 + X^5 + 1$	{97}	(01100001)
α^{67}	$X^7 + X^6 + X^1$	{194}	(11000010)
α^{68}	$X^7 + X^4 + X^3 + 1$	{153}	(10011001)
α^{69}	$X^5 + X^3 + X^2 + X^1 + 1$	{47}	(00101111)
α^{70}	$X^6 + X^4 + X^3 + X^2 + X^1$	{94}	(01011110)
α^{71}	$X^7 + X^5 + X^4 + X^3 + X^2$	{188}	(10111100)
α^{72}	$X^6 + X^5 + X^2 + 1$	{101}	(01100101)
α^{73}	$X^7 + X^6 + X^3 + X^1$	{202}	(11001010)
α^{74}	$X^7 + X^3 + 1$	{137}	(10001001)
α^{75}	$X^3 + X^2 + X^1 + 1$	{15}	(00001111)
α^{76}	$X^4 + X^3 + X^2 + X^1$	{30}	(00011110)
α^{77}	$X^5 + X^4 + X^3 + X^2$	{60}	(00111100)
α^{78}	$X^6 + X^5 + X^4 + X^3$	{120}	(01111000)
α^{79}	$X^7 + X^6 + X^5 + X^4$	{240}	(11110000)
α^{80}	$X^7 + X^6 + X^5 + X^4 + X^3 + X^2 + 1$	{253}	(11111101)
α^{81}	$X^7 + X^6 + X^5 + X^2 + X^1 + 1$	{231}	(11100111)
α^{82}	$X^7 + X^6 + X^4 + X^1 + 1$	{211}	(11010011)
α^{83}	$X^7 + X^5 + X^4 + X^3 + X^1 + 1$	{187}	(10111011)
α^{84}	$X^6 + X^5 + X^3 + X^1 + 1$	{107}	(01101011)
α^{85}	$X^7 + X^6 + X^4 + X^2 + X^1$	{214}	(11010110)
α^{86}	$X^7 + X^5 + X^4 + 1$	{177}	(10110001)
α^{87}	$X^6 + X^5 + X^4 + X^3 + X^2 + X^1 + 1$	{127}	(01111111)
α^{88}	$X^7 + X^6 + X^5 + X^4 + X^3 + X^2 + X^1$	{254}	(11111110)
α^{89}	$X^7 + X^6 + X^5 + 1$	{225}	(11100001)
α^{90}	$X^7 + X^6 + X^4 + X^3 + X^2 + X^1 + 1$	{223}	(11011111)
α^{91}	$X^7 + X^5 + X^1 + 1$	{163}	(10100011)
α^{92}	$X^6 + X^4 + X^3 + X^1 + 1$	{91}	(01011011)
α^{93}	$X^7 + X^5 + X^4 + X^2 + X^1$	{182}	(10110110)
α^{94}	$X^6 + X^5 + X^4 + 1$	{113}	(01110001)
α^{95}	$X^7 + X^6 + X^5 + X^1$	{226}	(11100010)
α^{96}	$X^7 + X^6 + X^4 + X^3 + 1$	{217}	(11011001)
α^{97}	$X^7 + X^5 + X^3 + X^2 + X^1 + 1$	{175}	(10101111)
α^{98}	$X^6 + X^1 + 1$	{67}	(01000011)
α^{99}	$X^7 + X^2 + X^1$	{134}	(10000110)
α^{100}	$X^4 + 1$	{17}	(00010001)
α^{101}	$X^5 + X^1$	{34}	(00100010)
α^{102}	$X^6 + X^2$	{68}	(01000100)
α^{103}	$X^7 + X^3$	{136}	(10001000)
α^{104}	$X^3 + X^2 + 1$	{13}	(00001101)
α^{105}	$X^4 + X^3 + X^1$	{26}	(00011010)
α^{106}	$X^5 + X^4 + X^2$	{52}	(00110100)
α^{107}	$X^6 + X^5 + X^3$	{104}	(01101000)
α^{108}	$X^7 + X^6 + X^4$	{208}	(11010000)
α^{109}	$X^7 + X^5 + X^4 + X^3 + X^2 + 1$	{189}	(10111101)
α^{110}	$X^6 + X^5 + X^2 + X^1 + 1$	{103}	(01100111)
α^{111}	$X^7 + X^6 + X^3 + X^2 + X^1$	{206}	(11001110)
α^{112}	$X^7 + 1$	{129}	(10000001)
α^{113}	$X^4 + X^3 + X^2 + X^1 + 1$	{31}	(00011111)
α^{114}	$X^5 + X^4 + X^3 + X^2 + X^1$	{62}	(00111110)
α^{115}	$X^6 + X^5 + X^4 + X^3 + X^2$	{124}	(01111100)
α^{116}	$X^7 + X^6 + X^5 + X^4 + X^3$	{248}	(11111000)
α^{117}	$X^7 + X^6 + X^5 + X^3 + X^2 + 1$	{237}	(11101101)
α^{118}	$X^7 + X^6 + X^2 + X^1 + 1$	{199}	(11000111)
α^{119}	$X^7 + X^4 + X^1 + 1$	{147}	(10010011)
α^{120}	$X^5 + X^4 + X^3 + X^1 + 1$	{59}	(00111011)
α^{121}	$X^6 + X^5 + X^4 + X^2 + X^1$	{118}	(01110110)
α^{122}	$X^7 + X^6 + X^5 + X^3 + X^2$	{236}	(11101100)
α^{123}	$X^7 + X^6 + X^2 + 1$	{197}	(11000101)
α^{124}	$X^7 + X^4 + X^2 + X^1 + 1$	{151}	(10010111)
α^{125}	$X^5 + X^4 + X^1 + 1$	{51}	(00110011)
α^{126}	$X^6 + X^5 + X^2 + X^1$	{102}	(01100110)
α^{127}	$X^7 + X^6 + X^3 + X^2$	{204}	(11001100)
α^{128}	$X^7 + X^2 + 1$	{133}	(10000101)
α^{129}	$X^4 + X^2 + X^1 + 1$	{23}	(00010111)
α^{130}	$X^5 + X^3 + X^2 + X^1$	{46}	(00101110)
α^{131}	$X^6 + X^4 + X^3 + X^2$	{92}	(01011100)
α^{132}	$X^7 + X^5 + X^4 + X^3$	{184}	(10111000)
α^{133}	$X^6 + X^5 + X^3 + X^2 + 1$	{109}	(01101101)
α^{134}	$X^7 + X^6 + X^4 + X^3 + X^1$	{218}	(11011010)
α^{135}	$X^7 + X^5 + X^3 + 1$	{169}	(10101001)
α^{136}	$X^6 + X^3 + X^2 + X^1 + 1$	{79}	(01001111)
α^{137}	$X^7 + X^4 + X^3 + X^2 + X^1$	{158}	(10011110)
α^{138}	$X^5 + 1$	{33}	(00100001)
α^{139}	$X^6 + X^1$	{66}	(01000010)
α^{140}	$X^7 + X^2$	{132}	(10000100)
α^{141}	$X^4 + X^2 + 1$	{21}	(00010101)
α^{142}	$X^5 + X^3 + X^1$	{42}	(00101010)

α^{143}	$X^6 + X^4 + X^2$	{84}	(01010100)
α^{144}	$X^7 + X^5 + X^3$	{168}	(10101000)
α^{145}	$X^6 + X^3 + X^2 + 1$	{77}	(01001101)
α^{146}	$X^7 + X^4 + X^3 + X^1$	{154}	(10011010)
α^{147}	$X^5 + X^3 + 1$	{41}	(00101001)
α^{148}	$X^6 + X^4 + X^1$	{82}	(01010010)
α^{149}	$X^7 + X^5 + X^2$	{164}	(10100100)
α^{150}	$X^6 + X^4 + X^2 + 1$	{85}	(01010101)
α^{151}	$X^7 + X^5 + X^3 + X^1$	{170}	(10101010)
α^{152}	$X^6 + X^3 + 1$	{73}	(01001001)
α^{153}	$X^7 + X^4 + X^1$	{146}	(10010010)
α^{154}	$X^5 + X^4 + X^3 + 1$	{57}	(00111001)
α^{155}	$X^6 + X^5 + X^4 + X^1$	{114}	(01110010)
α^{156}	$X^7 + X^6 + X^5 + X^2$	{228}	(11100100)
α^{157}	$X^7 + X^6 + X^4 + X^2 + 1$	{213}	(11010101)
α^{158}	$X^7 + X^5 + X^4 + X^2 + X^1 + 1$	{183}	(10110111)
α^{159}	$X^6 + X^5 + X^4 + X^1 + 1$	{115}	(01110011)
α^{160}	$X^7 + X^6 + X^5 + X^2 + X^1$	{230}	(11100110)
α^{161}	$X^7 + X^6 + X^4 + 1$	{209}	(11010001)
α^{162}	$X^7 + X^5 + X^4 + X^3 + X^2 + X^1 + 1$	{191}	(10111111)
α^{163}	$X^6 + X^5 + X^1 + 1$	{99}	(01100011)
α^{164}	$X^7 + X^6 + X^2 + X^1$	{198}	(11000110)
α^{165}	$X^7 + X^4 + 1$	{145}	(10010001)
α^{166}	$X^5 + X^4 + X^3 + X^2 + X^1 + 1$	{63}	(00111111)
α^{167}	$X^6 + X^5 + X^4 + X^3 + X^2 + X^1$	{126}	(01111110)
α^{168}	$X^7 + X^6 + X^5 + X^4 + X^3 + X^2$	{252}	(11111100)
α^{169}	$X^7 + X^6 + X^5 + X^2 + 1$	{229}	(11100101)
α^{170}	$X^7 + X^6 + X^4 + X^2 + X^1 + 1$	{215}	(11010111)
α^{171}	$X^7 + X^5 + X^4 + X^1 + 1$	{179}	(10110011)
α^{172}	$X^6 + X^5 + X^4 + X^3 + X^1 + 1$	{123}	(01111011)
α^{173}	$X^7 + X^6 + X^5 + X^4 + X^2 + X^1$	{246}	(11110110)
α^{174}	$X^7 + X^6 + X^5 + X^4 + 1$	{241}	(11110001)
α^{175}	$X^7 + X^6 + X^5 + X^4 + X^3 + X^2 + X^1 + 1$	{255}	(11111111)
α^{176}	$X^7 + X^6 + X^5 + X^1 + 1$	{227}	(11100011)
α^{177}	$X^7 + X^6 + X^4 + X^3 + X^1 + 1$	{219}	(11011011)
α^{178}	$X^7 + X^5 + X^3 + X^1 + 1$	{171}	(10101011)
α^{179}	$X^6 + X^3 + X^1 + 1$	{75}	(01001011)
α^{180}	$X^7 + X^4 + X^2 + X^1$	{150}	(10010110)
α^{181}	$X^5 + X^4 + 1$	{49}	(00110001)
α^{182}	$X^6 + X^5 + X^1$	{98}	(01100010)
α^{183}	$X^7 + X^6 + X^2$	{196}	(11000100)
α^{184}	$X^7 + X^4 + X^2 + 1$	{149}	(10010101)
α^{185}	$X^5 + X^4 + X^2 + X^1 + 1$	{55}	(00110111)
α^{186}	$X^6 + X^5 + X^3 + X^2 + X^1$	{110}	(01101110)
α^{187}	$X^7 + X^6 + X^4 + X^3 + X^2$	{220}	(11011100)
α^{188}	$X^7 + X^5 + X^2 + 1$	{165}	(10100101)
α^{189}	$X^6 + X^4 + X^2 + X^1 + 1$	{87}	(01010111)
α^{190}	$X^7 + X^5 + X^3 + X^2 + X^1$	{174}	(10101110)
α^{191}	$X^6 + 1$	{65}	(01000001)
α^{192}	$X^7 + X^1$	{130}	(10000010)
α^{193}	$X^4 + X^3 + 1$	{25}	(00011001)
α^{194}	$X^5 + X^4 + X^1$	{50}	(00110010)
α^{195}	$X^6 + X^5 + X^2$	{100}	(01100100)
α^{196}	$X^7 + X^6 + X^3$	{200}	(11001000)
α^{197}	$X^7 + X^3 + X^2 + 1$	{141}	(10001101)
α^{198}	$X^2 + X^1 + 1$	{7}	(00000111)
α^{199}	$X^3 + X^2 + X^1$	{14}	(00001110)
α^{200}	$X^4 + X^3 + X^2$	{28}	(00011100)
α^{201}	$X^5 + X^4 + X^3$	{56}	(00111000)
α^{202}	$X^6 + X^5 + X^4$	{112}	(01110000)
α^{203}	$X^7 + X^6 + X^5$	{224}	(11100000)
α^{204}	$X^7 + X^6 + X^4 + X^3 + X^2 + 1$	{221}	(11011101)
α^{205}	$X^7 + X^5 + X^2 + X^1 + 1$	{167}	(10100111)
α^{206}	$X^6 + X^4 + X^1 + 1$	{83}	(01010011)
α^{207}	$X^7 + X^5 + X^2 + X^1$	{166}	(10100110)
α^{208}	$X^6 + X^4 + 1$	{81}	(01010001)
α^{209}	$X^7 + X^5 + X^1$	{162}	(10100010)
α^{210}	$X^6 + X^4 + X^3 + 1$	{89}	(01011001)
α^{211}	$X^7 + X^5 + X^4 + X^1$	{178}	(10110010)
α^{212}	$X^6 + X^5 + X^4 + X^3 + 1$	{121}	(01111001)
α^{213}	$X^7 + X^6 + X^5 + X^4 + X^1$	{242}	(11110010)
α^{214}	$X^7 + X^6 + X^5 + X^4 + X^3 + 1$	{249}	(11111001)
α^{215}	$X^7 + X^6 + X^5 + X^3 + X^2 + X^1 + 1$	{239}	(11101111)
α^{216}	$X^7 + X^6 + X^1 + 1$	{195}	(11000011)
α^{217}	$X^7 + X^4 + X^3 + X^1 + 1$	{155}	(10011011)
α^{218}	$X^5 + X^3 + X^1 + 1$	{43}	(00101011)
α^{219}	$X^6 + X^4 + X^2 + X^1$	{86}	(01010110)

α^{220}	$X^7+X^5+X^3+X^2$	{172}	(10101100)
α^{221}	X^6+X^2+1	{69}	(01000101)
α^{222}	$X^7+X^3+X^1$	{138}	(10001010)
α^{223}	X^3+1	{9}	(00001001)
α^{224}	X^4+X^1	{18}	(00010010)
α^{225}	X^5+X^2	{36}	(00100100)
α^{226}	X^6+X^3	{72}	(01001000)
α^{227}	X^7+X^4	{144}	(10010000)
α^{228}	$X^5+X^4+X^3+X^2+1$	{61}	(00111101)
α^{229}	$X^6+X^5+X^4+X^3+X^1$	{122}	(01111010)
α^{230}	$X^7+X^6+X^5+X^4+X^2$	{244}	(11110100)
α^{231}	$X^7+X^6+X^5+X^4+X^2+1$	{245}	(11110101)
α^{232}	$X^7+X^6+X^5+X^4+X^2+X^1+1$	{247}	(11110111)
α^{233}	$X^7+X^6+X^5+X^4+X^1+1$	{243}	(11110011)
α^{234}	$X^7+X^6+X^5+X^4+X^3+X^1+1$	{251}	(11111011)
α^{235}	$X^7+X^6+X^5+X^3+X^1+1$	{235}	(11101011)
α^{236}	$X^7+X^6+X^3+X^1+1$	{203}	(11001011)
α^{237}	$X^7+X^3+X^1+1$	{139}	(10001011)
α^{238}	X^3+X^1+1	{11}	(00001011)
α^{239}	$X^4+X^2+X^1$	{22}	(00010110)
α^{240}	$X^5+X^3+X^2$	{44}	(00101100)
α^{241}	$X^6+X^4+X^3$	{88}	(01011000)
α^{242}	$X^7+X^5+X^4$	{176}	(10110000)
α^{243}	$X^6+X^5+X^4+X^3+X^2+1$	{125}	(01111101)
α^{244}	$X^7+X^6+X^5+X^4+X^3+X^1$	{250}	(11111010)
α^{245}	$X^7+X^6+X^5+X^3+1$	{233}	(11101001)
α^{246}	$X^7+X^6+X^3+X^2+X^1+1$	{207}	(11001111)
α^{247}	X^7+X^1+1	{131}	(10000011)
α^{248}	$X^4+X^3+X^1+1$	{27}	(00011011)
α^{249}	$X^5+X^4+X^2+X^1$	{54}	(00110110)
α^{250}	$X^6+X^5+X^3+X^2$	{108}	(01101100)
α^{251}	$X^7+X^6+X^4+X^3$	{216}	(11011000)
α^{252}	$X^7+X^5+X^3+X^2+1$	{173}	(10101101)
α^{253}	$X^6+X^2+X^1+1$	{71}	(01000111)
α^{254}	$X^7+X^3+X^2+X^1$	{142}	(10001110)
$\alpha^{255}=\alpha^0$	1	{1}	(00000001)

List of Tables

1-1 Comparison of hard disks (mid 2016) [110][108][102][46] 3
1-2 Parameters of the δ-erasure resilient Markov models 6

2-1 Summary of field axioms 21
2-2 Modulo-2 addition 27
2-3 Modulo-2 multiplication 27
2-4 Example of extended field $GF(2^5)$ 28
2-5 Summary of properties of the (n, k) Reed-Solomon code over $GF(q)$ . . . 36
2-6 Collision probability of n-component Algebraic signatures over $GF(2^m)$. 43

5-1 Number of assembler instructions needed to perform carry-less multiplication $a \times b$ in $GF(2^8)$, where $\omega(b)$ is Hamming weight of multiplier b. 70
5-2 Throughput of vectorized algorithms for carry-less multiplication and modulo reduction for $GF(2^8)$ using SSE and AVX2 instruction set. 72
5-3 Latency and reciprocal throughput of instructions on Intel Haswell micro architecture 81

6-1 Number of lookup table operations per symbol and total size of lookup tables required by each of erasure reconstruction methods of RAID-Z scheme. 97
6-2 RAID-Z operation speed-up relative to the original RAID-Z methods . . . 102

A-1 Intel CPU platform testbed 136
A-2 ARM CPU platform testbed 136
A-3 Binary extended binary field $GF(2^8)$, generated with $p(X) = X^8 + X^4 + X^3 + X^2 + 1$, used in ZFS RAID-Z erasure coding. 138

List of Figures

1-1 Transition rate diagram of parallel rebuilding of δ-erasure tolerant code without data checksum validation. 4
1-2 A revised transition rate diagram of the parallel rebuilding of an δ-erasure tolerant code with data integrity validation 6
1-3 Mean Time To Data Corruption of δ-erasure resilient code; with and without checksumming during reconstruction 7
1-4 MTTDC as a function of redundancy overhead for replication and δ-resilient erasure codes 8
1-5 Standard RAID configurations 9
1-6 RAID-Z dynamic block layout 11
1-7 Components of Lustre file system installation 12

2-1 Communication system with a noisy channel 16
2-2 Block diagram of concatenated code 29
2-3 Systematic format of a codeword 32
2-4 Local and distributed storage systems 39
2-5 Example of distributed $(6, 4)$ Reed-Solomon code 41

3-1 Buffer-based encoding and decoding with (n, k) Reed-Solomon code ($m = n - k$) 51
3-2 Flowchart of Reed-Solomon buffer-based encoding algorithm. Initialization step is performed once for desired configuration. The RS Encoding part can be repeated for multiple buffers by reusing the same RS generator matrix. 52

4-1 Throughput of arithmetic and logic operations used for Reed-Solomon implementation on Intel and arm systems 56
4-2 Throughput of the LOAD operation on Intel and ARM systems 58
4-3 Streaming `LOAD/STORE` performance of Intel Haswell CPU 59
4-4 Streaming `LOAD/STORE` performance of ARM CPU 60
4-5 Read latency of Intel Haswell CPU with hardware prefetching 61
4-6 Read latency of an Intel Haswell CPU without hardware prefetching . . . 62

4-7 Read latency of ARM CPU 63

5-1 Illustration of a 128-bit Vector arithmetic logic unit 66
5-2 LLVM compiler infrastructure 75
5-3 Class diagram of LLVM infrastructure for building JIT compiler 77
5-4 Number of instructions needed to implement Reed-Solomon(n,k) block codes over GF(2^8). 82
5-5 Throughput of unoptimized Reed-Solomon(n,k) block codes over GF(2^8) and GF(2^{16}), on the Intel Haswell platform 83
5-6 Throughput of optimized, JIT generated, Reed-Solomon(n,k) block codes over GF(2^8) and GF(2^{16}), on Intel Haswell platform. 84
5-7 Throughput of optimized, JIT generated, Reed-Solomon(n,k) block codes over GF(2^8) and GF(2^{16}), on ARM platform. 85

6-1 Combined throughput of RAID-Z2 parity operations on a pool consisting of 8 data and 2 parity disks 101
6-2 Combined throughput of RAID-Z2 parity operations running on Intel Haswell CPU with disabled hardware prefetching. 102
6-3 Per disk throughput in RAID-Z3 pool with 8 data and 3 code disks with data already present in CPU caches 103

7-1 Desclustered redundancy placement 107
7-2 Layouts of data and redundancy files in ECCFS 109
7-3 Components of ECCFS middleware in relation to Lustre file system components 110
7-4 Implementation option for ECCFS Monitor component 113
7-5 Sequence diagram of ECCFS Monitor component 114
7-6 Performance of individual Redis operations and MD Worker operations . 116
7-7 Cleaning of redundancy files performed by the ECCFS MD Worker component during processing of delete update. 117
7-8 ECC calculation performed by the ECCFS ECC Worker component . . . 118
7-9 Structure of ECC files 119

List of Algorithms

3-1 Carry-less multiplication of Galois Field elements. Inputs, a and b, are elements of $GF(2^l)$. The largest possible length of a result is $2l$. Additional modulo operation is needed to obtain a proper $GF(2^l)$ value. 47

3-2 Modulo reduction operation for $GF(2^{128})$ generated with polynomial $p(X) = 1 + X + X^2 + X^7 + X^{128}$ 50

4-1 Load benchmark, single and unrolled methods. $\mathbf{B}[]$ is the pre-allocated buffer. S_B is the size of the buffer, and S_V is the size of CPU vector registers. 57

4-2 Multi stream memory benchmark method with 4 input and 2 output streams. $\mathbf{B}_{in}[]$ and $\mathbf{B}_{out}[]$ are vectors of pointers to pre-allocated input and output buffers. S_B is the size of each buffer, and S_V is the size of CPU vector registers. 59

5-1 Pseudo code of the PCLMULQDQ instruction 68

5-2 Pseudo code of Reed-Solomon buffer based parity generation algorithm. . 78

5-3 Specialization of carry-less multiplication of Galois Field elements 79

6-1 Method for generating RAID-Z3 parity 96

6-2 Calculation of carry-less multiplication and modulo reduction lookup tables for parallel, 16 byte-wise, multiplication using vector shuffle operation. . . 99

6-3 Parallel multiplication method using SIMD shuffle instruction and pre-computed lookup tables. 100

List of Listings

5-1 PCLMULQDQ instruction intrinsic 68
5-2 Carry-less multiplication of elements in GF(2^{128}) using `PCLMULQDQ` intrinsics 69
5-3 VEX encoded SSE instructions used for implementing GF(2^8) operations 69
5-4 Vectorized carry-less multiplication by $X^2 + 1$ using SSE vector instructions 70
5-5 Vectorized modulo reduction operation for GF(2^8) generated with $p(X) = X^8 + X^4 + X^3 + X^2 + 1$ 72
5-6 Example of LLVM IR code illustrating vector operations 76

6-1 Optimized multiplication by 2 in GF(2^8) used in RAID-Z parity generation [114] 93

A-1 Hexagon DSP VLIW packets 137

Zusammenfassung

Wir leben im Zeitalter des Datenüberflusses. Selbst konservative Schätzungen prognostizieren ein exponentielles Wachstum an produzierten, übermittelten und gespeicherten Daten. Im Rahmen der Optimierung von Geschäftsprozessen sowie der wissenschaftlichen Forschung ist das Durchforsten und Verabeiten massiver Datenmengen, deren explosionsartiger Anstieg meist die Fähigkeiten konventioneller digitaler Speichersysteme übersteigt, von zunehmender Bedeutung. Wissenschaftliches Hochleistungsrechnen setzt zur Bewältigung dieses Problems auf große, verteilte Speichersysteme, die mittels Datenredundanz hohe Zuverlässigkeit und Langlebigkeit garantieren. Die einfachste Art und Weise Datenredundanz zu erreichen, ist das Replizieren der Daten auf mehreren physikalischen Geräten. Allerdings können modernere Ansätze, wie z.B. Löschcodierungsverfahren eine im Vergleich höhere Datensicherheit bei weniger eingesetztem Speicherplatz gewährleisten. In jüngster Zeit werden traditionelle, hardware-basierte Lösungen zur Ausfallsicherheit immer stärker von Software-Lösungen verdrängt. Komplexe Ausfallmodi von Speichersystemkomponenten erfordern Prüfsummen, um eine langfristige Datenintegrität zu gewährleisten. Um mit ständig wachsenden Datenmengen umzugehen, ist eine flexible und effiziente Softwareimplementierung von Fehlerkorrekturverfahren von großer Bedeutung.

Diese Arbeit stellt eine Methode zur Realisierung eines flexiblen Reed-Solomon-Fehlerkorrekturverfahrens unter Verwendung der Just-in-time-Kompilierungstechnik vor. Durch das Entfernen arithmetischer Redundanz im Algorithmus und die Nutzung moderner Optimierungs-Compiler erhalten wir eine durchsatzeffiziente Löschcodierungs-Implementierung. Zusätzlich wird die Datenparallelität ausgenutzt, indem der Compiler angewiesen wird, einen SIMD-Code für die gewünschte Ausführungsplattform zu erzeugen Wir zeigen die Ergebnisse von Codes, die mit SSE- und AVX2-SIMD-Befehlssätzen für x86- und NEON-Befehlssatz für ARM-Plattformen implementiert wurden. Als nächstes stellen wir ein Framework für effiziente, vektorisierte RAID-Z-Redundanzoperationen des ZFS-Dateisystems vor. Traditionelle, tabellenbasierte Galoiskörper-Multiplikationsalgorithmen werden durch spezialisierte SSE- und AVX2-Parallelmethoden ersetzt, die wesentlich schnellere und effizientere Paritätsoperationen

ermöglichen. Schließlich stellen wir ein neues Löschcodierungs-Schema vor, das auf bestehenden, leistungsstarken, parallelen Dateisystemen verwendet werden kann. Die beschriebene Middleware (ECCFS) ermöglicht die Definition von flexiblen, dateibasierten Zuverlässigkeitsrichtlinien und die Anpassung an spezifische Benutzerbedürfnisse. Durch die Verwendung des Blocklöschcodeverfahrens erreicht ECCFS eine optimale Speicher-, Berechnungs- und Netzwerkressourcenauslastung und bietet gleichzeitig ein hohes Maß an Zuverlässigkeit. Die verteilte Struktur der Middleware ermöglicht eine größere Skalierbarkeit und effizientere Nutzung von Speicher- und Netzwerkressourcen, um die Verfügbarkeit des Systems zu verbessern.

Zuverlässigkeit von Massenspeichern

Während die Festplatten-Kapazität ungebremst wächst können andere Merkmale nicht Schritt halten. Der lineare Lese-/Schreibdurchsatz wird durch die physikalische Drehzahl der Platten begrenzt. Auch die Zuverlässigkeit hat sich, wenn überhaupt, nicht wesentlich verbessert. Aktuelle Festplatten der Server-Klasse[1] bieten Kapazitäten von 1 TB bis 8 TB mit einer anhaltenden Datenübertragungsrate von $200\,\mathrm{MB\,s^{-1}}$. Solche Festplatten haben laut Angaben eine *mittlere Btreibsdauer zwischen Ausfällen*[2] (MTBF) von 2 000 000 h und eine *Bitfehlerrate* (nicht behebbare Lesefehler) von 10^{-15}. Allerdings belegen Zuverlässigkeitsstudien [75], dass realistische Zahlen in der Regel deutlich schlechter sind.

Von großem Interesse in Verbindung mit der Erhöhung der Festplatten-Kapazitäten sind *nicht behebbare Lesefehler* (URE). Diese Fehler können transient, verursacht durch Fehler in der magnetischen Flussdichte-Decodierung, oder dauerhaft, basierend auf physikalischen Fehlern des magnetischen Mediums, sein. Dies stellt vor allem bei herkömmlichen RAID-Systemen Probleme dar, da hier die Daten-Integrität nicht mit Prüfsummen sichergestellt wird. Mit steigenden HDD-Kapazitäten erhöht sich die Wahrscheinlichkeit für einen URE während eines RAID-Wiederherstellungs-Vorgangs deutlich. Ohne spezifischen Schutz wird der auftretende Fehler in die rekonstruierten Blöcke übertragen[17].

Herkömmliche Dateisysteme bieten keinen Schutz der Daten. Die Integrität der Daten kann allerdings in vielerlei Hinsicht aufgrund schwer zu erfassender Fehler, die überall in der Speicher-Hardware oder in den Software Ebenen auftreten können beeinträchtigt werden Solche Fehler beinhalten z.B. stille Datenverfälschung[79][3][22], Festplatten-Firmware-Fehler[103], verschiedene Software- und Treiberfehler[92] sowie

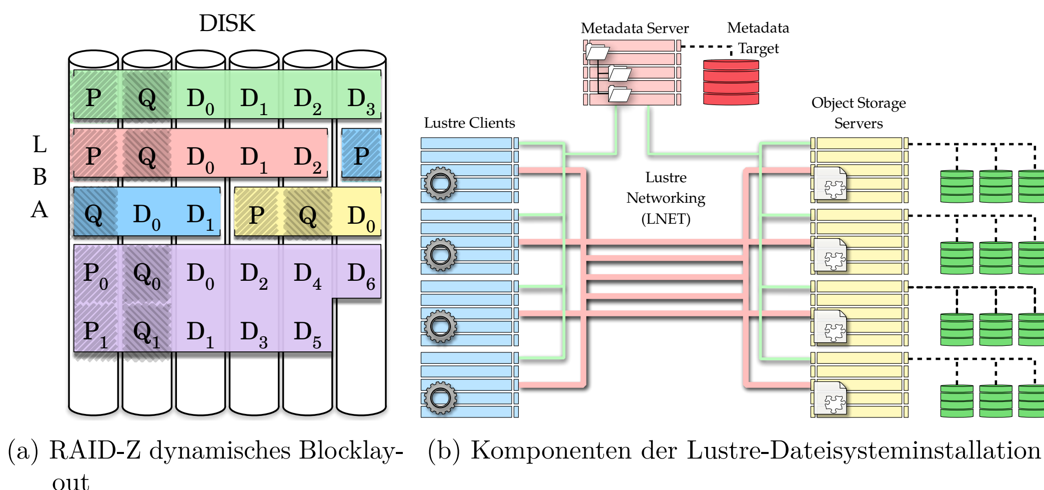
[1] Basierend auf WD Gold™ Festplatten Spezifikation (Modell WD6002FRYZ, 2016)

[2] Da HDDs typischerweise nicht reparierbar sind, ist dies gleichzeitig ein Maß für die mittlere Lebensdauer (MTTF)

Netzwerkfehler. Es wurde bereits gezeigt, dass traditionelle paritätsbasierte RAID-Techniken nicht genügend Schutz gegen zunehmend komplexe Fehler in Hardware- und Softwarekomponenten bieten[60].

ZFS und Lustre

ZFS ist ein Festplatten-Dateisystem, das mehrere Management-Technologien integriert, die traditionell nicht Teil eines Dateisystems sind. Der *Logical Volume Manager* (LVM) ist eine Basis-Infrastruktur von ZFS, die Software-RAID- und Datenintegritätsfunktionen bereitstellt. Der ZFS-LVM unterstützt das Zusammenfassen des zugrunde liegenden Speichers in Spiegelungen oder RAID-Z Daten-Paritäts-Verteilungsschemata, genannt vdev[3]. RAID-Z ist ein fortschrittliches Paritätsverteilungsschema, das eine softwaredefinierte RAID-ähnliche Redundanz bietet (Abbildung 1(a)). Das RAID-Z3-Löschschema unterstützt die Wiederherstellung von drei Festplattenfehlern. Da die kombinierte Zuverlässigkeit eines redundanten Arrays letztlich vom der schnellen Wiederherstellung abhängt, beschreiben wir in dieser Arbeit eine Implementierung von vektorisierten Methoden zur Paritätserzeugung und Datenrekonstruktion für das ZFS-Dateisystem.



(a) RAID-Z dynamisches Blocklayout

(b) Komponenten der Lustre-Dateisysteminstallation

Abbildung 1: Komponenten der ZFS und Lustre-Dateisysteminstallation

Mit der Popularität von Cluster-Computern ging die Notwendigkeit für verteilte Dateisysteme einher. Skalierbarkeit und Kapazität von NAS-Systemen (*Network Attached Storage*) erwiesen sich rasch als limitierende Faktoren. Durch Fortschritte in Netzwerk-Technologie wurde zudem ein paralleler Zugriff auf mehrere Speichersysteme wünschenswert. Ein Beispiel eines parallelen, POSIX-kompatiblen, verteilten

[3] **V**irtual **Dev**ices

Dateisystem[10], das viele Anforderungen an Hochleistungsrechner (HPC) erfüllt ist Lustre. Es ist derzeit das beliebteste Open-Source-Dateisystem in HPC- und Rechenzentrumsumgebungen.

Die Hauptkomponenten einer typischen Lustre-Installation bestehen aus einem Metadaten-Server (MDS), einem Objekt-Storage-Server (OSS) und Lustre-Klienten (Abbildung 1(b)). Jeder OSS ist mit einem oder mehreren Object-Storage-Targets (OST) verbunden, die den Datenspeicher für das Dateisystem bereitstellen. Die Datenzuverlässigkeit wird an die OSTs delegiert. Der Verlust eines OST führt daher zu einem unwiederbringlichen Datenverlust. Da ein OST Teilstücke vieler Dateien speichert ist im Falle eines Fehlers die Nichtverfügbarkeit und der Datenverlust deutlich größer als die Speicherkapazität des OST selbst. Im Folgenden beschreiben wir ein flexibles, auf Dateiebene ansetzendes, Multi-OST-Löschschema zur Lösung dieses Problems

Fehlerkorrekturverfahren

Fehlerkorrekturverfahren (englisch *Error Correction Codes, ECC*) sind ein integraler Bestandteil der digitalen Datenübertragung und -speicherung, die z.B. in Mobilfunknetzen, paketbasierter Kommunikation wie dem Internet, digitalen Videoübertragungen, Weltraumkommunikation, usw. verwendet werden. Ein allgemeines Kommunikationsmodell für Nachrichtenübertragung über einen nicht rauschfreien Kanal, das erstmals formal durch C. Shannon in [104] eingeführt wurde ist in Abbildung 2 dargestellt.

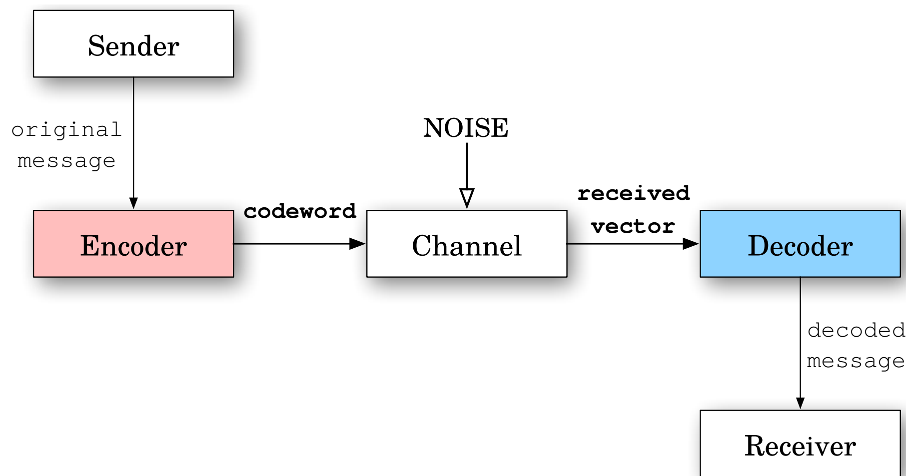


Abbildung 2: Kommunikationssystem mit einem verrauschten Kanal

Um eine Nachricht während der Übertragung über einen verlustbehafteten Kanal vor Datenverlust zu schützen kann sie codiert werden. Ein Kanal-Codierer wandelt

dazu die ursprünglichen Nachricht in eine neue Sequenz mit Redundanz um, die im Fehlerfall verwendet werden kann um die ursprüngliche Nachricht wieder herzustellen. Das Verhältnis von Eingangs- und Ausgangs-datenbits des Kanal-Codierers wird als *Coderate* bezeichnet. Der Kanal ist das Übertragungs- oder Speichermedium, das zur Übermittlung der Nachricht verwendet wird. Die Rolle des Kanal-Decodierers besteht darin, die vom Kanal empfangenen Daten zu übernehmen und die ursprünglichen Nachrichtendaten wiederherzustellen.

Fehlerkorrigierende Block-Codierung

In Kommunikations- und Speichersystemen sind zwei strukturell unterschiedliche Arten von Kanalcodierungen weit verbreitet. Dies sind Block- und Faltungscodes. Blockcodes verarbeiten Informationsbits in festen Blockgrößen und erzeugen daraus die Redundanzbits. Beispiele für Blockcodes sind Reed-Solomon-Codes, Hamming-Codes [43] und Walsh-Hadamard-Codes [5]. Blockcodes eignen sich insbesondere, wenn ein Kanal die Form eines physikalischen Mediums für die Datenspeicherung hat. Ein weiteres wichtiges Merkmal von Blockcodes ist, dass sie für *Löschcodierungsverfahren* verwendet werden können. Eine *Löschung* bezeichnet in diesem Fall ein Fehler, dessen Position im Codewort im Voraus bekannt ist. Im Kontext von Datenspeichen kann dies z.B. ein ausgefallenes Speichermedium oder ein Verbindungsverlust eines angeschlossenen Netzwerkspeichers sein.

Reed-Solomon-Code

Ein Reed-Solomon-Code ist ein linearer Blockcode mit Codesymbolen aus dem Körper GF(q), wobei q eine Potenz einer Primzahl ist. Solche Codes werden auch q-ary Blockcode oder Blockcode über GF(q) genannt. Ein q-ary (n, k) Blockcode hat die Länge n und enthält q^k Codeworte. Nachrichten eines solchen Blockcodes besteht aus k Informationssymbolen aus GF(q).

Für die Verwendung in der Datenübertragung über *Löschkanäle* wurde eine spezielle Form von Reed-Solomon-Codes entwickelt. Ein Löschkanal ist ein Kommunikationskanal, der entweder ein Informationsbit (oder Symbol) an den Empfänger überträgt oder dem Empfänger mitteilt, dass Informationen verloren gegangen sind. Die Codierungsprozedur geht davon aus, dass n Codewörter eines (n, k) Reed-Solomon-Codes über den Löschkanal gesendet werden. Diese Klasse von Reed-Solomon-Codes verwendet die Vandermonde-Matrix als Startgeneratormatrix für den Code.

Da wir eine systematische Version des Reed-Solomon-Codes verwenden produzieren wir für die Datenpuffer D_i ($0 \leqslant i < k$) wie in Abbildung 3 gezeigt nur die mit

C_j $(0 \leqslant j < m,\ m = n - k)$ bezeichneten Paritäts-Teile des Codewortes. S_B stellt die Größe der Puffer in Bytes dar und muss eine ganze Zahl an Symbolen aus $GF(2^l)$ enthalten. In dieser Arbeit präsentieren wir die Umsetzung von Reed-Solomon-Codes mit unterschiedlichen Symbolgrößen. Aus diesem Grund wird die Puffergröße als mindestens $512\,B$ lang angenommen.

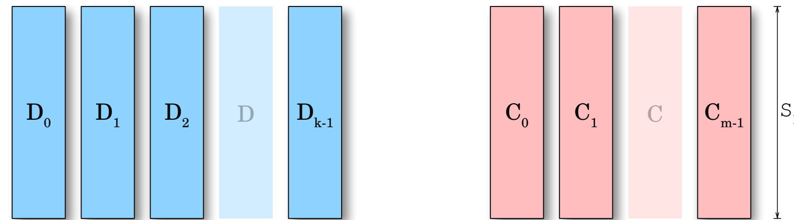


Abbildung 3: Puffer-basierte Codierung und Decodierung mit (n, k) Reed-Solomon-Code $(m = n - k)$

Die größte Herausforderung bei der Implementierung eines schnellen und effizienten Reed-Solomon-Fehlerkorrekturverfahrens ist die Tatsache, dass alle Operationen in einem Galoiskörper durchgeführt werden müssen. Da die Berechnung des Reed-Solomon-Codes durch eine Matrix-Vektor-Multiplikation repräsentiert werden kann, ist es entscheidend, dass Additions- und Multiplikationsoperationen im Galoiskörper effizient umgesetzt werden.

Die Multiplikation von zwei Elementen eines binären endlichen Körpers $GF(2^l)$ ist definiert als:

$$a \circ b = (a \times b) \mod p$$

Das Kreuzprodukt "$\times$" ist die übertragsfreie Multiplikation der Polynomdarstellungen von Elementen in $GF(2^l)$. Das Endergebnis erhält man durch eine Polynomreduktion (Modulo) unter Verwendung eines primitiven Körpergenerators p des Galoiskörpers. Da weder die übertragsfreie Multiplikation noch die Modulo-Operation in Hardware implementiert sind, benötigen wir einen effizienten Algorithmus für beide Operationen. Die übertragsfreie Multiplikation ähnelt der Ganzzahl-Multiplikation. In beiden Operationen wird das zweite Element entsprechend der Anzahl an gesetzten Bits im ersten Element verschoben. Der Unterschied besteht darin, dass die übertragsfreie Multiplikation eine übertragsfreie Addition (XOR-Operation) verwendet, während die Ganzzahl-Multiplikation eine gewöhnliche Addition verwendet, die einen Übertrag erzeugt und propagiert. Ein effizienter Modulo-Reduktionsalgorithmus, basierend auf dem Barrett-Reduktionsalgorithmus [4] ist im [37] beschrieben.

Wir präsentieren mehrere praktische Implementierungen von effizienten, pufferbasierten Reed-Solomon-Löschcodierungsverfahren. Unterschiedliche Anwendungsbedingungen erlauben verschiedene Arten von Optimierungen. In dieser Arbeit zeigen wir Implementierungen mit SIMD-Anweisungen und Just-in-time-Kompilierungstechnik.

Reed-Solomon-Code Vektorisierung

Die SIMD-Implementierung nutzt die SIMD-Einheiten moderner CPUs. Durch die parallele Ausführung der beschriebenen Galoiskörper-Operationen auf mehreren Elementen kann eine signifikante Beschleunigung erreichet werden. Der Hauptbeitrag dieser Arbeit ist die Erforschung der Einsatzmöglichkeiten zur Verfügung stehender SIMD-Operationen auf x86- und ARM-Plattformen. Eine praktische Umsetzung dieser Arbeit hat bereits zum quelloffenen Dateisystem ZFS beigetragen. Die beschriebenen Methoden für Reed-Solomon-Codeberechnungen enthalten intrinsisch redundante Operationen, die nicht mit herkömmlichen Software-Erstellungsmethoden optimiert werden können. Um dieses Problem zu lösen, verwenden wir die Just-in-time-Kompilierungstechnik zum Spezifizieren, Zusammensetzen und Ausführen des Codes. Wir haben dabei die LLVM-Infrastruktur [63] verwendet, die es ermöglicht, hoch optimierte und vektorisierte Code für x86- und ARM-Plattformen zu erstellen und auszuführen.

Viele moderne Prozessoren unterstützen Vektoranweisungen zur Verbesserung datenintensiver Anwendungen. Diese Parallelität auf Datenebene wird durch Verwendung einer vektorisierte arithmetisch-logischen Einheit (englisch *Arithmetic Logic Unit, ALU*) erreicht. Abbildung 4 veranschaulicht eine 128-Bit Vektor-ALU. Die dargestellte ALU ist in der Lage, Operationen auf zwei 64-Bit-, vier 32-Bit- oder acht 16-Bit-Operanden gleichzeitig auszuführen. Die Operationen werden auf allen Vektorelementen parallel durchgeführt.

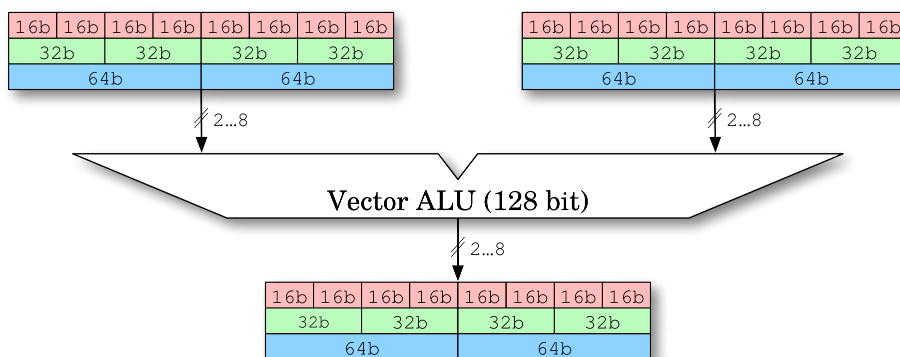


Abbildung 4: Illustration eines 128-Bit-Vektor-Arithmetisch-logische Einheit

Der Block-Reed-Solomon-Algorithmus hat keine Inter-Data-Abhängigkeiten, weshalb für eine effiziente vektorisierte Implementierung eine signifikante Leistungssteigerung gegenüber der skalaren Version zu erwarten ist. Weil wir einen Puffer-basierten Codierungsalgorithmus verwenden, lässt sich recht einfach eine *horizontale Vektorisierung* bei der Entwicklung der Algorithmen verwenden. Diese Technik kombiniert

mehrere unabhängige Datensymbole, die in gleicher Weise verarbeitet werden, in ein einziges Vektorregister. Nach dem Anwenden der Algorithmusoperationen auf Vektoren sequentieller Datensymbolen enthält der resultierende Vektor dieselbe Anzahl an Paritätsprüfsymbolen.

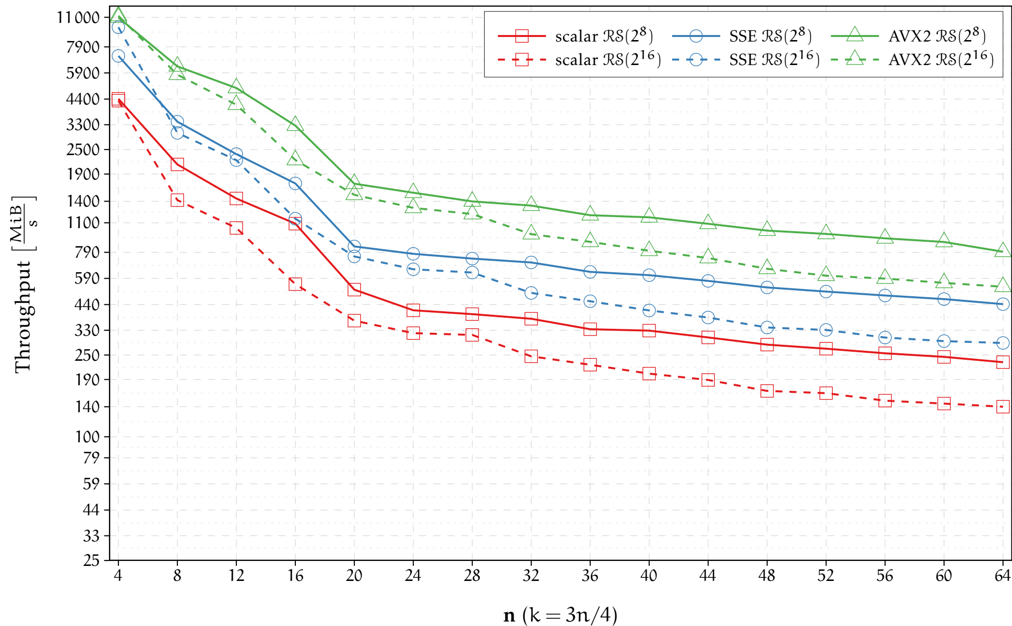


Abbildung 5: Durchsatz von optimierten, JIT generierten Reed-Solomon (n, k) Blockcodes über $\mathsf{GF}(2^8)$ und $\mathsf{GF}(2^{16})$, auf Intel Haswell CPU.

Um die besten Ergebnisse zu erzielen, haben wir die Bitgröße des Elements des Galoiskörper auf die Länge der Vektorelemente angepasst. Die Auswertung (Abbildung 5) zeigt, dass der maximalen Codierungsdurchsatz für eine kleine Anzahl von Daten und Codesymbolen nahezu der Speicherbandbreite der Testplattformen entspricht. Allerdings ist die rechnerische Komplexität größerer Matrixmultiplikationen der dominierende Faktor für große Codes.

ZFS: RAID-Z Vektorisierung

Das RAID-Z-Schema des ZFS-Dateisystems basiert auf einem systematischen Reed-Solomon-Code, der nach der Vandermonde-Methode konstruiert wird. RAID-Z verwendet Elemente des erweiterten Körpers $GF(2^8)$, die unter Verwendung des primitiven Polynoms $p(X) = X^8 + X^4 + X^3 + X^2 + 1$ über $GF(2)$ kostruiert werden. Die Generatormatrix $\mathbf{A}_{\text{RAID-Z}}$ erhält man durch Verkettung der Identitätsmatrix $\mathbf{I}_k$ und der Code generierenden Matrix $\mathbf{A}^*_{\text{RAID-Z}}$

$$\mathbf{A}_{\text{RAID-Z}} \cdot \mathbf{d} = \mathbf{c}_{\text{RAID-Z}}$$

$$\begin{bmatrix} \mathbf{I}_k \\ \mathbf{A}^*_{\text{RAID-Z}} \end{bmatrix} \cdot \mathbf{d} = \begin{bmatrix} \mathbf{d} \\ \mathbf{c}^*_{\text{RAID-Z}} \end{bmatrix}$$

Die ursprüngliche RAID-Z-Implementierung verwendet Umsetzungstabellen für die Galoiskörper-Multiplikation, wobei die Exponetialtafel und die Logarithmentafel zum Einsatz kommen. Dieser Ansatz hat mehrere Nachteile. Der häufige Zugriff auf diese großen Tabellen innerhalb der inneren Schleife der Blockcode-Berechnungsroutine, kann die Leistung stark negativ beeinträchtigen. Die Multiplikation erfordert 3 Tabellen-Nachschlage-Operationen, die jeweils einen, im wesentlichen zufälligen Speicherzugriff erfordern. Ein weiterer Nachteil dieser Methode ist die intrinsische Serialisierung der Berechnung. Die Umsetzungstabellen unterstützen nur die Berechnung eines einzelnen Symbols zu einer Zeit, in diesem Fall eines einzelnen Bytes. In Anbetracht der Tatsache, dass die derzeitigen Prozessoren nativ mit 64 bit breiten Registern arbeiten, nutzt der Algorithmus effektiv nur 1/8 des theoretischen skalaren Operationsdurchsatzes.

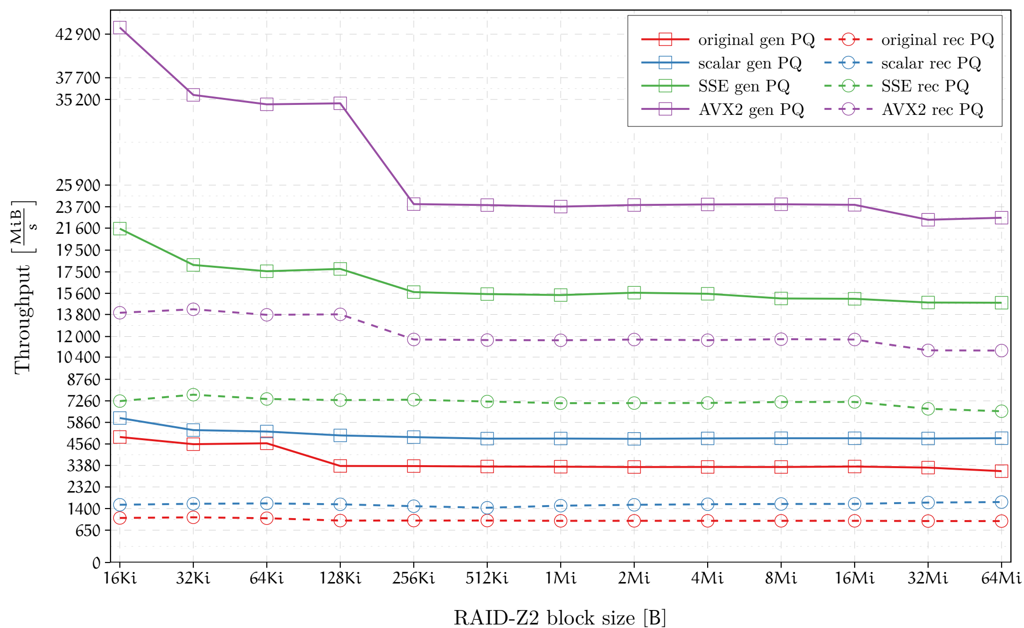


Abbildung 6: Kombinierter Durchsatz von der RAID-Z Paritätsoperationen auf einem Pool bestehend aus 8 Daten und 2 Paritätsplatten

Wir haben vektorisierte Methoden für die RAID-Z-Codierung und Decodierung angewandt. Die Ergebnisse sind in Abbildung 6 dargestellt. Die RAID-Z-Blockgröße umfasst Werte zwischen 16 KiB und 64 MiB. Die unterstützte Maximalgröße ist 16 MiB, so dass Daten nicht in die CPU-Caches passen. Die Auswirkungen des CPU-Daten-Caches sind deutlich erkennbar, was nahelegt, dass der Durchsatz durch

die Speicherbandbreite und nicht aufgrund der Rechenkomplexität limitiert ist. Dieser Effekt ist mit weniger rechenintensiven Paritätserzeugungsmethoden und mit breiteren Registergrößen stärker ausgeprägt. Die CPU der Testplattform hat 32 KiB L1- und 256 KiB L2-Daten-Cache pro CPU-Kern sowie 25 MiB gemeinsamen L3-Daten-Cache, der zwischen allen CPU-Kernen geteilt wird. Die Messungen zeigen einen Rückgang im Durchsatz exakt dann, wenn die Blockdaten nicht mehr vollständig in die Caches passen. Der Durchsatz größerer Blocks bleibt mit etwa $23\,000\,\mathrm{MiB\,s^{-1}}$ auf hohem Niveau.

ECCFS-Middleware

In dieser Arbeit stellen wir eine Anwendung von Löschcodierungsverfahren (*ECCFS-Middleware*) vor, die darauf abzielen, das Problem der Zuverlässigkeit von OSS, OST und der Kommunikationsverbindung eines typischen parallelen Dateisystems (PFS) zu lösen. Die Motivation und die Anforderungen an das vorgeschlagene Zuverlässigkeitsschema beinhalten: Optimale Festspeicher-Ausnutzung, Minimierung des ECC-Berechnungs-Mehraufwands und ein flexibles und automatisches Management. Eine einfache Datenreplikation erfordert viel höhere Speicher-Kosten im Vergleich zu Löschcodierungen mit gleichen Zuverlässigkeit. Die meisten parallelen Dateisysteme nutzen mehrere Speicherserver, um Daten einer einzelnen Datei zu speichern, wodurch die Zuverlässigkeit des Replikationsschemas bei mehreren Ausfällen verringert wird. Mit Hilfe von Maximum-Distanz-Codes, wie dem Reed-Solomon-Löschcode, sorgen wir für eine minimale Festspeicher-Bedarf bei gleichzeitiger optimaler Zuverlässigkeit, selbst im Falle mehrfacher gleichzeitiger OST-Ausfälle.

Um den ECC-Berechnungs-Mehraufwand zu minimieren, verschieben wir die Berechnung von ECC-Bausteinen für neu eingeführte Dateien. Dies verhindert das Verschwenden von Rechenleistung und Bandbreite bei der ECC-Berechnung für kurzlebige und häufig aktualisierte Dateien. Bei typischen RAID-Systemen ist die Anzahl der Paritätsdatenträger festgelegt, was gleiche Zuverlässigkeit für alle Daten gewährleistet. Jedoch sind nicht alle Daten gleichermaßen wichtig, z.B. Daten, die durch ein Experiment produziert werden, sind wichtiger als Simulationsergebnisse, die neu berechnet werden können. Die Möglichkeit, ein gewünschtes Redundanzniveau festzulegen, bringt also einen Mehrwert.

Eine Übersicht über die ECCFS-Middleware und das zugrundeliegende PFS (Lustre) ist in Abbildung 7 dargestellt.

Die ECCFS-Middleware besteht aus folgenden Komponenten: *Monitor* (Änderungsüberwachung), *Changelog* (Änderungsprotokoll), *MD-Worker* (Metadaten-Arbeiter), und *ECC-Worker* (ECC-Arbeiter). Die ECCFS-Monitor-Komponente ist

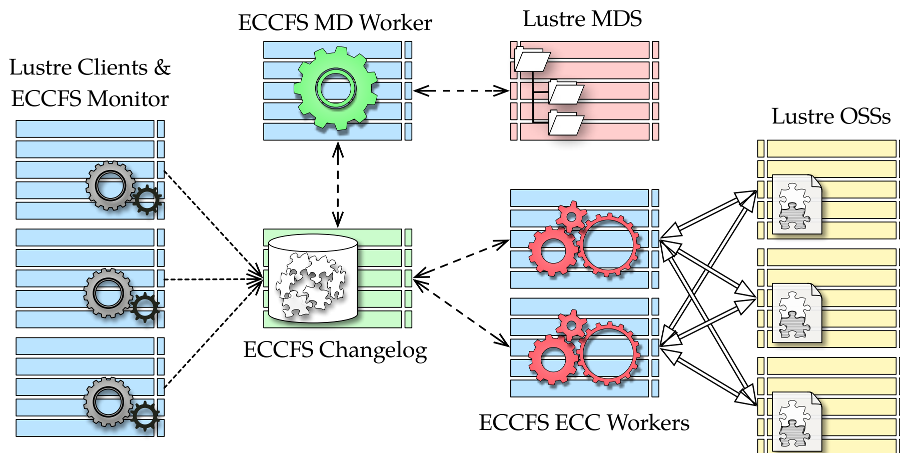


Abbildung 7: Komponenten der ECCFS-Middleware in Bezug auf Lustre-Dateisystemkomponenten

für das Sammeln von Informationen über geänderte oder erstellte Daten innerhalb des PFS verantwortlich. Ein Änderungsupdate wird durch das Tupel (`file_name`, `offset`, `length`) dargestellt, das die Position jeder Veränderung, die der Benutzer macht, eindeutig darstellt. Die ECCFS-Changelog-Komponente dient als zentrale Datenbank zum Sammeln von Datei- und Metadaten-Updates, die vom ECCFS-Monitor gesendet werden.

Diese Informationen werden beibehalten, bis die Redundanz für geänderte Daten neu berechnet wurde oder bis alle Operationen, die den Redundanz-Repository betreffen durchgeführt wurden. Der ECCFS-MD-Worker ist verantwortlich für Metadatenoperationen und Redundanzberechnungsplanung. Das Design des ECCFS ermöglicht es, mehrere Instanzen des MD-Worker gleichzeitig zu betreiben, was eine höhere Skalierbarkeit von Metadatenoperationen ermöglicht. Der ECCFS-ECC-Worker bedient die Arbeits-Warteschlange und führt die eigentlichen Berechnungen zur Fehlerkorrektur durch. Einmal berechnet werden die Redundanzblocks in zuvor erstellten Redundanz-Dateien gespeichert.

Mit Ausnahme der MD-Worker-Komponente ist die gesamte ECCFS-Middleware unabhängig vom zugrunde liegenden PFS und ermöglicht die gleichzeitige Kopplung mit anderen PFSs, die ein Minimum an Anforderungen unterstützen. Die Trennung der Komponenten ermöglicht eine größere Skalierbarkeit, da leistungskritische Komponenten in mehr als einer Instanz eingesetzt werden können. Die hohe horizontale Skalierbarkeit des Systems maximiert die Nutzung von Berechnungs- und Bandbreitenressourcen.

In den Redundanzrichtlinie kann eine beliebige Anzahl von ECC-Dateien spezifiziert werden, einschließlich `0`, wobei in diesem Fall alle übereinstimmenden Dateien

vom ECCFS ignoriert werden. Die Redundanzrichtlinien können jederzeit angepasst werden, um die Zuverlässigkeit zu erhöhen oder zu verringern. Dies kann von Vorteil sein, wenn sich die Zuverlässigkeitskriterien ändern oder auf dem PFS mehr Platz benötigt wird. Im Falle einer Reduzierung werden die Redundanzdateien mit dem höchsten Index gelöscht, während die verbleibenden Dateien weiter ein gültiges Löschschema darstellen.

Um eine noch höhere Datenintegrität und Verfügbarkeit zu gewährleisten, könnte die ECCFS-Middleware in die eingesetzten Resourcenmanager integriert werden. Bevor ein neuer Auftrag ausgeführt wird könnte ECCFS im laufenden Betrieb eine Datenintegritätsprüfung aller vom Auftrag benötigten Dateien durchführen. Im Fehlerfalle könnten alle nicht verfügbaren Dateien rechtzeitig zur Auftrags-Ausführung rekonstruiert werden.

Fazit

Reed-Solomon-Codes sind im Bereiche der digitalen Datenverarbeitung und Speicherung allgegenwärtig. Die Implementierung effizienter Reed-Solomon-Fehlerkorrekturverfahren hängt maßgeblich von der Fähigkeit ab schnelle Multiplikationen in Galois-Körpern durchführen zu können. Diese Multiplikation steht in den meisten gewöhnlichen CPUs jedoch nicht als Hardware-Operation zur Verfügung und muss daher emuliert werden. Zur Lösung dieses Problems haben wir eine Methode präsentiert, die die nötige Multiplikation mittels Polynommultiplikation und -Reduktion löst. Wir haben gezeigt, dass sich die dafür benötigten Berechnungen durch die Anwendung von Just-in-time-Kompilierungstechniken entscheidend optimiert lassen. Die vorgestellte Implementation dieses Verfahrens bietet eine erhebliche Verbesserung der Galois-Körper-Emulation für Reed-Solomon-Codes. Darüber hinaus wurde dargestellt wie sich existierende Compiler-Infrastruktur nutzen lässt, um Code automatisch für x86 und ARM Plattformen zu vektorisieren.

Das ZFS-Dateisystem erfreut sich im Umfeld von Hochleistungs-Speichersystemen aufgrund seiner software-basierten Ausfallsicherheistmechanismen an wachsender Beliebtheit. Anstelle von hardware-basierten RAID-Lösungen setzt ZFS auf die CPU zum berechnen von Redundanz- und Integritätsinformationen und vereinfacht so die Hardware Infrastruktur. ZFS unterstützt RAID-Z, eine Löschcodierungsverfahren für Massenspeicher-Pools, welches Ausfallsicherheit bei bis zu 3 Fehlern bietet. Der Durchsatz der benötigten Paritäts-Operationen kann durch den Einsatz von SIMD-Einheiten moderner CPUs erheblich gesteigert werden. Wir haben die Implementierung eines generischen Frameworks zur Berechnung von RAID-Z-Paritäts-Operationen vorgestellt, welches eine Vielzahl an SIMD-Befehlssätzen unterstützt. Mit Hilfe dieses

Frameworks wurden RAID-Z-Implementierungen für den Skalaren-, SSE- und AVX2-Befehlssatz erstellt. Die optimierte SIMD-Implementation weist eine bis zu 95-fache Durchsatz-Verbesserungen bei der Berechnung von RAID-Z-Paritätsinformationen und der Rekonstruktion von Daten auf. Die implementierten Routinen wurden erfolgreich ins ***ZFS on Linux*** Projekt integriert und finden dort seit **Version** 0.70 aktive Verwendung.