An Erasure Coding Performance Metric for Windows 8

Sarah Edge Mann, Michael Anderson, and Marek Rychlik

Abstract

Electronic data itself may be ethereal, but the devices that store it and transport it are physical. As such, when electronic data travels in space and time over physical devices, it is subject to noise and deterioration. The standard solution to eliminate this noise is Reed-Solomon coding, typically implemented in hardware and used today in every data storage device. We present a measurement of Reed-Solomon coding in a new application: a Windows 8 storage device driver. This software application brings the benefits of noise elimination much closer to the producer or consumer of the data, and has surprisingly good performance, even for very strong encoding. Our results challenge the common assumption that standard Reed-Solomon codes, realized with Galois Field multiply operations, are too computationally expensive to be effectively deployed in software. We also include a brief general discussion of Reed-Solomon codes and of the computational costs of ECC.

1 Introduction

Reed-Solomon codes are used throughout the communication and storage industries to protect data from noise. All non-volatile storage devices, most server DRAMs and all disk arrays use Reed-Solomon or Reed-Solomon-like codes to detect and correct data errors. These codes are often called Erasure Codes or Error Correcting Codes (ECC), both of which have the same mathematical basis.

Noise that affects electronic data has many sources, including cosmic rays, reported and unreported device or communication failures, human error, power problems, physical media degradation and many others. As distance, time, and data volumes increase, so does this noise and deterioration. Unless a methodology is applied to compensate for this noise, electronic data will be lost in the noise, rendering it useless.

To understand the noise cancellation and regenerative effects of ECC in real applications, we suggest that it is important to consider two separate metrics. The first metric is the distance in space and time between the creation of the data and the protection of the data. As long as the data remains unprotected, it can be corrupted in an undetectable manner. The second metric is the strength of the codeword. The stronger the codeword, the more noise can be eliminated.

An ideal data communication or storage system would apply ECC protection as close as possible in space and time to the creator of the data, remove ECC protection as close as possible to the data consumer, and would support very strong codewords. This combination of features would best protect the data from the inevitable noise it encounters in its lifecycle.

For application data, the creator or consumer of the data is an application program, executing on an application processor. In the past, application processors have been too expensive to apply to anything but application execution. However, in modern processors with multiple cores, these resources are often underutilized. This is primarily because legacy applications were designed for a single processor, and most processors sold now are multi-core.

From a space and time perspective, a core in an application processor is an ideal location for ECC-based noise elimination. The distance in space and time between the application data creation and the ECC logic is nearly zero. The physical distance may actually be zero, since the same processor that the application used to generate the data and initiate the transfer could be used to encode or decode the ECC.

In Windows 8, we accomplished this near-zero distance noise elimination solution with a software device driver implemented within a Windows 8 device driver model called *StorPort virtual miniport* [7]. Our implementation is an autonomous piece of software apart from using the Microsoft driver framework. The driver acts on behalf of the application that produces the data, and encodes this data with ECC prior to storing it on a local or remote memory system. When an application reads the data in the future, this same driver applies ECC logic to eliminate any noise that was introduced as the data travelled through space and time.

For Windows 8 applications that use this ECC driver, the ECC encoding and decoding provides strong data protection and noise immunity. The ECC logic ensures that all data communication paths and memory devices between from the application processor through the storage are fully protected from both reported and unreported data errors, regardless of their source.

For example, if a cosmic ray from deep space caused a soft failure of the computer's DRAM, the ECC logic would detect it and correct it, without requiring any additional hardware. Or, if a flash device media degraded to the point that a sector was unreadable, the driver would use ECC to recover the data. With a thoughtful arrangement of the codewords and devices, even full device failures can be considered "noise", and data can be recovered in their absence. With strong codewords, multi-device failures could be tolerated without requiring immediate service. This is very strong application data protection, especially compared to typical Windows 8 desktop systems without any DRAM ECC.

This paper provides measurements of and discussions about such a Windows 8 ECC device driver as well as a discussion of the coding theory on which the driver is built. This driver implements ECC-protected memory that can be benchmarked with standard tools such as the ATTO Disk Benchmark and Intel's IOMeter [1, 8]. By varying the strength of the ECC codewords and the number of cores used by the driver, we present a series of measurements that show Windows 8 can provide very high noise immunity for application data using standard Reed-Solomon codes based on Galois Field multiply operations.

2 Our Windows 8 Software ECC

In the current secton we discuss a set of ideas and ECC technologies which we call jointly *Virtual ECC*. Subsequently, we introduce our software implementation of *Virtual ECC* for the new Windows 8 operating system from Microsoft.

The current section adequately describes our main result for those readers who are familiar with Reed-Solomon codes and their application to RAID. However, an interested reader will find the background material organized in several additional sections of this paper. Here, occasionally we make a (forward) reference to this material, hopefully with minimal distraction to an expert reader.

2.1 Virtual ECC

The concept of *Virtual ECC* is quite simple: we mean an approach to data protection in which extra bits are added to the data as early as possible after data inception, and stored as its integral part forever. The virtuality of this approach alludes to the fact that the manner in which these extra bits are chosen is not fixed by a single physical hardware implementation but may be adjusted to suit the varying needs of different applications. Strong encoding may be used in some cases, and weak encoding in others, all on the same hardware.

This contrasts with the typical hardware-based approach. Hardware-assisted ECC typically works as follows:

1. Dedicated hardware encodes the input data by adding a fixed number of redundant bits on the fly when the data reaches the hardware device.
2. The extra bits are removed when the data is shipped off, typically to another hardware device which in turn may add ECC bits of its own.
3. When the data travels between devices through some interconnect fabric, other bits, usually much weaker than ECC, may or may not be added or checked depending on the technology or vendor.

Thus, understanding the actual data protection as data moves through a system may be very complex. As a result, the probability of data loss is dominated by the weakest link, which likely will remain unknown until after a failure occurs.

By comparison, *Virtual ECC* deals with these unknowns in advance, with a scheme that matches the intended use and environment of the data. For data stored on very large systems, commonly called cloud infrastructures, very strong codewords can be applied to data, and then spread out between devices, device racks, facilities, even continents. For small systems, like home storage for multimedia libraries, weaker codewords would suffice.

Virtual ECC has the power to transform the lowest reliability devices into high reliability devices, and to eliminate another important problem that occurs frequently in large systems — silent data corruption. Since the protection bits are generated early in the life of the data, and stay attached to the data throughout its lifecycle, they guarantee the correctness of the data. Generating and checking ECC near the application eliminates all practical possibility of silent errors in any device that the data may encounter.

For example, here is a minimal list of devices that data will encounter on a typical computer: CPU, DRAM, PCI, Local Storage Controller, Remote Storage Controller, Flash, Tape or Disk media, and finally Interconnects (SATA, SAS, FC, IB). Strikingly, most computers use DRAM with no ECC protection. However, by using *Virtual ECC*, they are transformed into devices with much higher reliability than computers with hardware ECC. With *Virtual ECC*, data is protected from the moment the message is transformed into a codeword, and remains protected, with a known codeword strength, until it is used again. This protection covers every device the data encounters during its lifecycle. Corrupted codewords, even those silently corrupted, are easy to detect and correct regardless of the correctness of the devices used to store or transport the data.

We can also point to the following advantages of *Virtual ECC* over current ECC schemes:

1. Strong codes are millions of times more reliable for the same overhead or cost.
2. The ability to upgrade the reliability of existing hardware as well as future hardware at a software-only cost.
3. High configurability — ECC protection can be matched to the data and the environment.
4. Device lifetimes can be extended — ECC can be increased as devices deteriorate.

2.2 Our *Virtual ECC* Benchmarks

Virtual ECC is arguably the best solution for comprehensive data protection, and our implementation of it under Windows, in addition to the general benefits of *Virtual ECC*, offers additional advantages. They are rooted in the modern OS and hardware technology:

- A new, "software friendly" ECC algorithm.
- Leveraging of multi-core architecture.
- Faster processing than hardware-based alternatives.

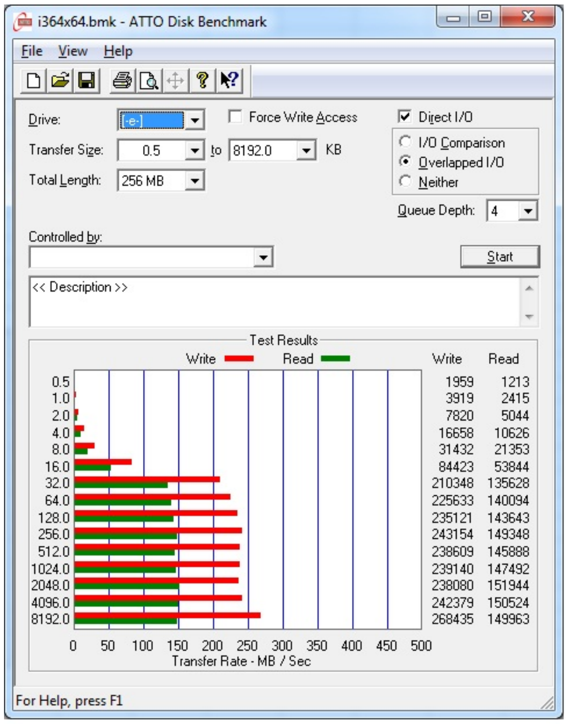
We emphasize the light load on the CPU, which should become even ligther with new generations of processors. Thus, we have achieved unsurpassed flexibility without taxing hardware resources. The algorithm used by us and its computational complexity are covered in Section 4.

By writing the driver, we have demonstrated that the software implementation of ECC is *in practice* faster than a conceivable hardware implementation. Our ECC leverages the fastest, superbly tested, and most computationally advanced component of the system, the CPU ASIC. This is the most highly optimized part of a computer system and most capable for implementing advanced mathematical algorithms. Thus, by utilizing the CPU we are able to cash in on the tremendous resources which went into its development.

In contrast, the typical peripheral hardware design is subject to many compromises, and as a result ECC runs at a fraction of the speed possible with the CPU. By its nature, the CPU development enforces much higher quality assurance than a typical peripheral would.

Since in our *Virtual ECC* approach ECC is an integral part of the data, it can be easily implemented utilizing new infrastructures, based for instance on cloud storage. We may either replace or add to the existing ECC. Given that the reliability of the underlying storage system is available or can be estimated, we may estimate how much extra data protection we need, to balance the extra cost involved against our data protection need. RAID failure modeling may be used by the designer of the RAID system to determine the major parameters of the data protection, such as the ratio of check to data drives. Further information on RAID failure modeling is provided in Section 5.

In Table 1 and Table 2 we present performance measurements of the StorPort virtual miniport driver on several hardware configurations. The measurements come from the industry-standard Disk Benchmark software from ATTO Technology, Inc [1]. We tested write and read operations with a varying number of bytes until sustained throughput was achieved. We also took steps to ensure that we measure the cost of ECC only, in isolation from other factors, such as the drive speed. This isolation was achieved by interposing a write-back cache in front of the real hard drive cache. The cache was sufficiently large to ensure the cache was not flushed before our measurements are performed. Although other costs are captured in this measurement, such as the cost of the DRAM to CPU data movement, they should be considered negligible as compared to the cost of performing ECC operations.



i364x64.bmk - ATTO Disk Benchmark

Drive: [-e-]
[ ] Force Write Access
[✓] Direct I/O
Transfer Size: 0.5 to 8192.0 KB
Total Length: 256 MB
( ) I/O Comparison
[✓ Overlapped I/O]
( ) Neither
Queue Depth: 4
Controlled by:
<< Description >>

	Write	Read
0.5	1959	1213
1.0	3919	2415
2.0	7820	5044
4.0	16658	10626
8.0	31432	21353
16.0	84423	53844
32.0	210348	135628
64.0	225633	140094
128.0	235121	143643
256.0	243154	149348
512.0	238609	145888
1024.0	239140	147492
2048.0	238080	151944
4096.0	242379	150524
8192.0	268435	149963

Figure 1: Screen capture of the GUI of the ATTO benchmarking software with sample measurements of the StorPort virtual miniport driver.

When configured with an extremely strong codeword, 64 data bytes + 64 check bytes, we achieved write speed exceeding 600MB/sec and read speed of over 350MB/sec. With a weaker codeword, 64 data bytes + 10 check bytes, the throughput is in excess of 3GB/sec. In all experiments, we utilized 2–4 cores[1]. As usual, the performance analysis of a real computer system can only be known approximately. Thus, the sustained throughput rate for 10 and 64 check drives do differ *roughly* by a factor of 5. From the point of view of reliability (see Section 5 for more details on this subject), the difference still may be justified in some demanding applications. The difference in codeword strength between 64 and 10 check bits is several orders of magnitude in probability of data loss, and may be well worth it when preservation of data is paramount. The reader may also note somewhat slower read performance, which is not accounted for by theoretical operation counts.

Table 1: The driver write/read performance: 64 data bytes, 64 check bytes.

	Hardware configuration				
Volume of Data	Celeron 1.6Ghz	Pentium 2.2Ghz	i3 2.4Ghz	i5 2.7Ghz	i7 2.2Ghz
Write Performance (transfer rate in kiB/sec)					
512b	1445	1955	1959	2740	2540
1k	2774	3957	3919	5562	5132
2k	5508	8070	7820	11377	10541
4k	11959	16181	16658	24035	22415
8k	24576	33557	31432	55158	51447
16k	59076	79897	84423	150226	144340
32k	116260	161729	210348	467021	453199
64k	121972	173399	225633	482527	495103
128k	125136	181594	235121	498789	496325
256k	130043	171095	243154	662886	597922
512k	125144	172442	238609	666092	721753
1024k	127522	178956	239140	664444	708103
2048k	129366	180024	238080	661171	669713
4096k	134892	216480	242379	651542	651542
8192k	125144	173557	268435	651542	614268
Read Performance (transfer rate in kiB/sec)					
512b	914	1270	1213	1798	1714
1k	1808	2528	2415	3648	3479
2k	3624	5190	5044	7492	7211
4k	7430	10702	10626	15906	15321
8k	15641	21609	21353	36883	35703
16k	37321	52461	53844	104162	104667
32k	76249	110461	135628	300452	296023
64k	80669	116260	140094	312125	312125
128k	81035	122819	143643	321900	325825
256k	90697	112649	149340	418961	367121
512k	81344	114227	145000	418496	448460
1024k	80249	116711	147492	420368	443171
2048k	85353	122016	151944	417566	426088
4096k	95520	112081	150524	414800	405841
8192k	82090	114961	149963	415718	387166

Table 2: The driver write/read performance: 64 data bytes, 10 check bytes.

	Hardware configuration				
Volume of Data	Celeron 1.6Ghz	Pentium 2.2Ghz	i3 2.4Ghz	i5 2.7Ghz	i7 2.2Ghz
Write Performance (transfer rate in kiB/sec)					
512b	6632	8832	6415	10544	9631
1k	13789	17408	12579	21089	18130
2k	27068	32768	26295	41867	37376
4k	47300	66228	53115	83284	77596
8k	93499	120356	110235	164526	169387
16k	189665	272531	268435	342879	376504
32k	356554	523182	648269	897754	930968
64k	438645	621217	754581	1576887	1823763
128k	487893	675162	891806	2274442	2191108
256k	507391	691666	995109	3200232	2796029
512k	511305	689904	1022611	3314017	3730102
1024k	522502	713324	1010580	3338749	3768675
2048k	522502	691519	984482	3277737	3587998
4096k	511305	703045	986895	3146250	3347075
8192k	510091	685102	1115746	3079309	3271974
Read Performance (transfer rate in kiB/sec)					
512b	5409	7132	5081	8151	6843
1k	11008	13721	10290	16262	13789
2k	21961	26800	20127	32524	28089
4k	38814	49192	41261	66774	58658
8k	78223	107531	83284	135251	138313
16k	159695	231204	210457	327419	350177
32k	308623	457708	496325	758978	1034229
64k	358454	515051	590595	1514190	1597065
128k	392254	555745	678416	1879048	1830975
256k	402653	565640	745924	2577017	2329764
512k	407602	588451	759722	2625285	3004874
1024k	418496	586388	747384	2664371	2960685
2048k	407602	580749	743931	2631720	2850055
4096k	399655	600974	745654	2591332	2763306
8192k	403229	560538	745385	2464543	2730981

2.3 Operation Count vs. Real Performance

Let us focus on the 64 data + 64 check bytes configuration, achieving write speed of 600MB/sec. Based on our knowledge of Reed-Solomon coding, we find that the number of Galois field operations required to generate a codeword from a message is equal to the number of check bytes per each byte of the input message. In Section 4 we provide the theoretical basis for this claim. Since in Read-Solomon coding of RAID check bytes map 1:1 to check drives, this number is also equal to the number of check drives. This means that each of the 64 bytes of inputs contributes 64 Galois field operations to the cost of creating the 128 byte codeword. 600MB/s translates into 6×10^8 bytes of input per second, or $6 \times 10^8 \times 64 = 384 \times 10^8$ Galois operations per second. We feel that this is a very important metric of system performance and justifies giving it a name; we propose GalOPS ($\underline{\text{G}}$alois $\underline{\text{OP}}$erations per $\underline{\text{S}}$econd). We may think of GalOPS as analogous to FLOPS which characterizes floating point performance.

[1]The precise count of resources such as cores and hardware threads is complicated by such technologies as hyperthreading, where some of the computational resources are shared; thus, we would rather think of the capabilities of a system as a whole than the particulars of the CPU architecture.

In Section 4 we included further discussion of this metric.

In our example the performance of our driver suggests that the performance of our system is 38 GigaGalOPS. Based on the clock speed of 2.4GHz, we perform about 16 Galois field operations per clock cycle, or 4 Galois operations per clock cycle per processor, given a quadcore system. These estimates are imprecise, but it is certain that we must be able to perform *at least* 4 Galois operations in one clock cycle in each processor to justify the measured data. Using another method based on observing hardware performance counters, we indeed confirmed that this simple estimate of the actual performance is accurate (see Section 2.4). It is clear that the resulting performance is due to both multithreading and utilization of the pipelined architecture of the modern CPU.

When we use the codeword of 64 data bytes with 10 check bytes (still very strong protection), the I/O scales up by a factor of 5 but the number of Galois operations is 6.4 times smaller, resulting in approximately the same GalOPS measurement. This suggests that GalOPS indeed are a measure of the performance of the system, and not of the particulars of the algorithm used.

2.4 Measurement Methodology Details

For those readers who would like to more closely follow what our process, in this section we include additional details of our approach. They can be skipped without affecting the ability to understand the rest of the paper.

As we mentioned, Windows supplies a driver model *StorPort virtual miniport*, as well as examples to aid programmers who develop drivers [7]. The StorPort virtual miniport model itself was used to develop our *Virtual ECC* driver, but we did not use the Microsoft example code.

Inside this driver model, Windows also provides a set of measurement tools that can gather real time events at high speed and with low overhead. These tools are called *Event Tracing for Windows (ETW)*. We used these tools to gather real time performance information about the driver so we could understand how it behaved in a real operating system environment, and verify actual performance against modeled results.

Using these tools, we gathered detailed measurements of each IO that the benchmark generated as it was executed by the driver, and saved these measurements for later analysis. To bound the scope of our measurements, we focused on a particular processor and transfer size, though this same technique could have been applied to any processor or transfer size.

For each IO, we recorded the total execution time. In addition, we recorded three important events to understand performance at a more detailed level:

1. The time that the cache was searched and locked;
2. The time that the data was transferred to or from the benchmark to the cache;
3. The time it took to compute the ECC bytes for the request.

A typical 32k write request on an i7, for example, took 1 microsecond or less to search and lock the cache, about 6 microseconds to move the data from the ATTO benchmark memory to the driver cache memory, and about 400 microseconds to compute the ECC. A typical read request had a similar cache search and lock time, but a larger time (about 600 microseconds) to compute the ECC. The cause for this difference should be further investigated.

We included two charts, Figures 2 and 3, which show the ECC computation time in a 64 data bytes + 64 check bytes codeword configuration for a 32k IO. The first chart shows the time required to regenerate lost data using ECC, the second chart shows the time required to generate the parity bytes for a write. Though it is clear from these measurements that generating parity is faster than regenerating data, the cause should also be further investigated.

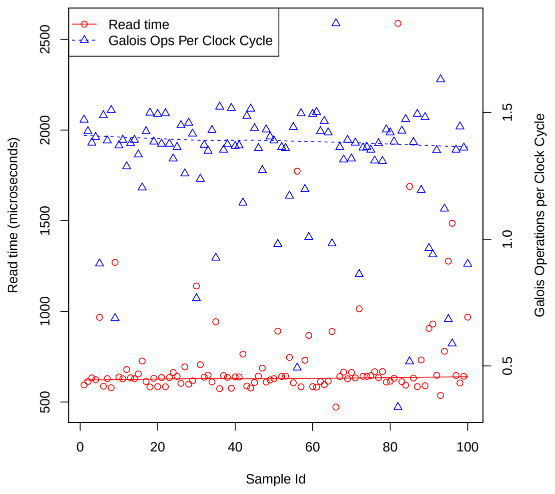


Figure 2: Measurements of the first 100 read operations @32k, 64+64 codeword. The lines are smoothed data obtained using the LOWESS method.

3 Review of Coding Theory

In this section we discuss the essentials of coding theory that allow for ECC in computing systems. Here, we will limit our discussion to linear block codes over finite fields focusing on Reed-Solomon-like codes in particular. These codes are particularly well suited for data protection in

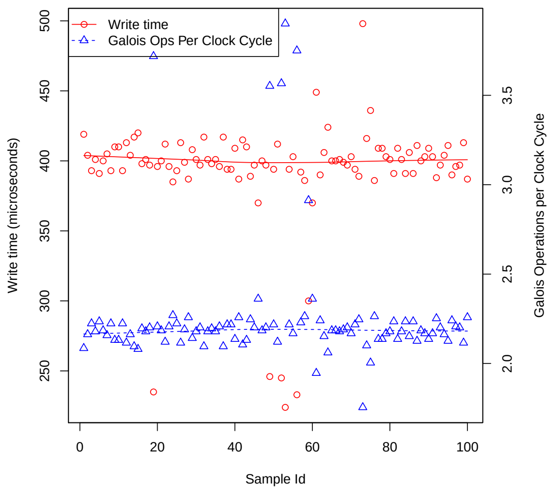


Figure 3: Measurements of the first 100 write operations @32k, 64+64 codeword. The lines are smoothed data obtained using the LOWESS method.

computer systems. A more detailed discussion of coding theory, linear block codes, and Reed Solomon codes can be found in standard texts on this subject [5, 12, 16].

In Reed-Solomon ECC, data originates as a string of bits which are first grouped into *characters* consisting of l bits of data; $l = 8$ is typical and corresponds to byte-sized characters. These characters are then grouped into k character *messages*. By adding an additional $m = n-k$ carefully chosen characters to a message, a group of n characters (the *codeword*) is created. Thus codewords are somewhat redundant, containing more characters than the message. The set of all possible codewords constitutes the *code*; this set is dependent on the manner in which the redundant characters are chosen.

The codeword is then transmitted or stored for later use and in the process is subjected to random errors. We refer to this codeword post storage or transmission as the *received word*. Due to the redundancy, the codeword and thus the original message may be recovered from the received word if the number of errors is not too great. The *rate* $R = \frac{k}{n}$ of the code measures the degree of redundancy in the codeword. $R = 1$ indicates there is no redundancy and no error correction capacity, and R near zero indicates there is a lot of redundancy and thus error correction capacity, but little information content.

3.1 Data Encoding

The essence of the problem of ECC is how to choose the redundant characters such that a message may be recovered from a received word in the presence of a large number of errors in a computationally efficient manner. We accomplish this using a Reed-Solomon-like encoding scheme reliant on linear algebra. We view each character as an element of the finite (Galois) field GF(2^l) and thus each message as a vector in GF$(2^l)^k$. All arithmetic is performed over this finite field.

We call a matrix $F \in$GF$(2^l)^{n\times k}$ an *information dispersal matrix* if any $k \times k$ submatrix of F is invertible. F is *systematic* if the first k rows of F form the $k \times k$ identity matrix. Let $\vec{d} \in$ GF$(2^l)^k$ be a message and $F \in$GF$(2^l)^{n\times k}$ be a systematic information dispersal matrix. Then $\vec{c} = F\vec{d}$ is a *codeword*. Since F is systematic, the first k characters of $\vec{c}$ are the k characters of $\vec{d}$. The last m characters of $\vec{c}$ are redundant or *check* data. There are a number of ways to construct a systematic information dispersal matrix. See [13] and [14] for one one example. Such a matrix may be constructed with as much redundancy as desired subject to the constraint $n \leq 2^l$, and this matrix uniquely defines the code.

3.2 Error Detection

It is easy to verify the integrity of data in this scheme. Given a length n received word, interpret the first k characters as a message. Use this message to recalculate the check data, as described above. If the computed check data matches the check data portion of the received word, then the received word is a codeword and, with high probability, the first k characters constitute the original message. However, if the check data does not match, an error has been detected.

3.3 Erasure Decoding

In some cases, a codeword may be corrupted by a simple deletion of a few characters in the word at known locations. These deletions are called *erasures* and recovery from erasures is straightforward when there are no other errors present. If characters $e_1, \ldots, e_f$ have been erased, with $f \leq m$, then we can recover the intended message in the following manner. Let $\hat{\vec{c}}$ represent the first k non-erased characters from the received word. Form $\hat{F}$ by removing rows $e_1, \ldots, e_f$ from F; also remove the last $m-s$ rows. $\hat{F}$ is a $k \times k$ submatrix of F and is thus invertible. Therefore, $\vec{d} = \hat{F}^{-1}\hat{\vec{c}}$.

3.4 Code Distance and Error Decoding

Recovering a message from a received word when the locations of the errors are unknown is more difficult and requires greater redundancy. Here, we will only discuss the conditions under which it is possible to recover from such *silent errors*, and leave the details of how to perform the decoding to other sources.

The *Hamming distance*, or simply distance, between two words of the same length is the number of positions at which the two words differ [4]. The *distance d* of a code is the minimum distance between any two codewords in the code. This is the minimum number of errors that must occur in order to change one valid codeword into another.

Notice from the discussion of erasure decoding in Section 3.3, that any k correct characters from a codeword are sufficient to recover the message. Therefore, any two codewords must differ from each other in at least $m+1$ positions. For one codeword to be transformed into another, at least $m+1$ errors must have accumulated. Thus $d = m+1$ for this class of codes.

Without a known bias in the types of errors that accumulate in a codeword, the standard approach is to decode a received word to the codeword that is nearest in Hamming distance. If a codeword has accumulated t errors during storage or transmission, then the received word is a distance t from the codeword. If $t < \frac{d}{2}$ then no other codeword is closer to the received word than the original codeword. The received word will be correctly decoded to the original codeword. On the other hand, if $t \geq \frac{d}{2}$ then some other codeword may be closer to or as close to the received word as the original codeword. If so, the decoding will be incorrect. For our codes, if $t < \frac{d}{2} = \frac{m+1}{2}$ silent errors have accumulated in a given received word, then we can always correctly decode to the original codeword. $t < \frac{m+1}{2}$ is the *error bound* for this code.

For t small, e.g. $t = 1$ or $t = 2$, it is reasonably efficient to simply check whether each word that is distance t from the received word is a codeword. If one is, decode to that codeword. For larger t, a more sophisticated algorithm is required to decode efficiently. The Welch-Berlekamp algorithm is suggested [17, 2].

3.5 Simultaneous Erasure and Silent Error Decoding

These codes can also correct simultaneous erasures and silent errors, for m sufficiently large. If the received word contains f erasures and t silent errors then it may always be decoded correctly for $2t + f < m+1$. Exhaustive search of nearby words is recommended for t small, and the Welch-Berlekamp algorithm for t large.

3.6 ECC for Data Storage

A primary use of ECC in computer systems is to protect stored data from corruption or device failure. RAID systems are often used for this task. A RAID (Redundant Array of Independent Disks) is a collection of total $T = M+N$ storage devices with the capacity to hold N devices worth of data. As with general ECC, the stored data is broken into l bit characters, and the l bit characters are grouped into length $k = N$ messages. Each character in a message is stored on a different one of the N data devices. $m = M$ characters of check data are generated per Section 3.1; these characters are stored across the remaining M check devices. Together with the Reed-Solomon coding scheme, we refer to this setup as RS(T,M) by analogy with the simpler schemes such as RAID0–6.

The failure of a device in the RAID corresponds to an erasure. So long as no more than M devices fail simultaneously, full data recovery is possible using the technique outlined in Section 3.3. Undetected sector failures and erroneous device read or write operations might contribute silent errors to a codeword. These may also be detected and corrected per Sections 3.2, 3.4, and 3.5.

4 The Computational Cost of ECC

ECC is not free but comes with certain computational costs. In this section we will analyze these costs for various ECC-related tasks.

In a computer system, we usually must handle a large amount of data all at once, particularly when preparing data for RAID storage. Let $D \in GF(2^l)^{k \times p}$ be a matrix of such data. Each column of D corresponds to a message $\vec{d}$; p is the number of messages to be handled; and the ith row of D consists of all characters in the ith message position in all of the messages. $C = FD$ is the matrix of all codewords, including both message and check data. The first k rows of C are identical to D, and the last m rows of C are check data.

4.1 Generating Check Data

The check characters in the codewords C are computed by multiplying D by the last m rows of F. This requires mkp multiplications and $m(k-1)p$ additions over GF(2^l). This is essentially m multiplications and additions for each character of message data.

4.2 Recovery from Erasures

Recovery from data erasures will be of particular interest in the application of ECC to storage systems, as discussed in Section 3.6. In this case, a failed storage device corresponds to the erasure of a row of data from the matrix C. We now give a detailed account of the computation needed to recover from such erasures.

Notice that since F is systematic, it has the form

$$F = \begin{bmatrix} I_k \\ \star \end{bmatrix},$$

where I_k is the $k \times k$ identity matrix, and $\star$ some matrix. Assume there have been f erasures of rows of data from the first k rows of C corresponding to message data , and thus $s = k - f$ message characters survive in each codeword. So long as no more than $m - f$ of the check characters are erased, we can recover the erased message data. We need only k characters total from the codewords to recover this data, so we may discard extra check characters if fewer than m total characters were erased.

Construct $\hat{C}$ as k characters (rows) of C that were not erased, including all available message characters. If Q is obtained from the $n \times n$ identity matrix by removing the m rows corresponding to erased (or ignored) characters, then $\hat{C} = QC$. Construct $\hat{D}$ as a permutation of D so that good data drives appear first in D and erased or ignored data appears last. We may denote this by $\hat{D} = PD = \begin{bmatrix} X \\ Y \end{bmatrix}$ where P is the appropriate permutation matrix, X is the good data and Y is the lost data. If we then set $\hat{F} = QFP^T$ (noting that $P^T = P^{-1}$) we have the reduced system $\hat{F}\hat{D} = \hat{C}$. $\hat{F}$ has the additional block structure

$$\hat{F} = \begin{bmatrix} I_k & 0 \\ A & B \end{bmatrix},$$

so we have the system of equations

$$\overset{\hat{F}}{\begin{bmatrix} I_k & 0 \\ A & B \end{bmatrix}} \overset{\hat{D}}{\begin{bmatrix} X \\ Y \end{bmatrix}} = \overset{\hat{C}}{\begin{bmatrix} X \\ W \end{bmatrix}}$$

where W is the matrix of remaining check data. This can be rewritten as $AX + BY = W$. Y is the unknown lost data, and all other variables are known, so we simply solve this system for Y: $Y = B^{-1}(W - AX)$.

It is the cost of computing Y via this formula in which we are interested. We will assume that the operation cost associated with constructing B, W, A, and X are negligible, as through clever referencing of F and C they need not be explicitly constructed. There are three steps in computing Y with operations counts as follows: (1) Computation of AX requires sfp Galois Field multiplies and $(s-1)fp$ GF additions. (2) Subsequent computation of $W - AX$ requires fp additions. (3) Solving $BY = W - AX$ for Y via *LU* factorization of B may be further broken into three steps:

1. Computing the *LU* factorization of B without pivoting requires $\frac{1}{3}(f^3 - f)$ additions, $\frac{1}{3}(f^3 - f)$ multiplies, and $\frac{1}{2}f(f-1)$ divides [15].

2. Solving $LZ = W - AX$ for Z using forward substitution requires $(f-1)p + \frac{1}{2}(f^2 - 3f + 1)p$ additions, and $\frac{1}{2}(f^2 - 3f + 1)p$ multiplies.

3. Solving $UY = Z$ for Y using backward substitution requires $(f-1)p + \frac{1}{2}(f^2 - 3f + 1)p$ additions, $\frac{1}{2}(f^2 - fm + 1)p$ multiplications, and fp divisions.

The overall operation count is:

- Additions (including subtractions): $fp(f + s - 1) - l + \frac{1}{3}(f^3 - f))$;

- Multiplications: $fp(f + s - 3) + l + \frac{1}{3}(f^3 - f)$;

- Divides: $fp + \frac{1}{2}f(f-1)$.

RAID systems are generally measured by their throughput, that is, how many bytes (characters) can be accepted or delivered within a fixed time interval. With this in mind, the most useful perspective on the cost of computation is with regard to each data byte that is accepted or delivered, *i.e.* the number of original data bytes in the system which is the number of data bytes in D, Nl. We can think of the operation counts listed above as a cost per original byte plus some overhead associated with computing the *LU* factorization of B. Table 3 summarizes these costs, also using the relationship $k = f + s$ to further simplify expressions.

Count	
+, -	$fkp - fp - p + \frac{1}{3}(f^3 - f))$
$\cdot$	$fkl - 3fp + p + \frac{1}{3}(f^3 - f)$
$\div$	$fp + \frac{1}{2}f(f-1)$
Count per byte of original data	
+, -	$f - \frac{f+1}{k}$
$\cdot$	$f - \frac{3f-1}{k}$
$\div$	$\frac{f}{N}$
Overhead	
+, -	$\frac{1}{3}(f^3 - f))$
$\cdot$	$\frac{1}{3}(f^3 - f))$
$\div$	$\frac{1}{2}f(f-1)$

Table 3: Summary of the count of operations required to recover lost data.

In summary, if f data devices are lost, then it costs about f additions and multiplies and < 1 divide per byte of original data requested to read data plus an additional f^3 additions and multiplies and f^2 divides of overhead costs associated with computing the *LU* factorization. When $f \ll p$ as is typically the case, the cost of f additions and multiplies per original data byte is the dominating cost of data delivery. As discussed in Section 4.1, it costs about m additions and multiplies per original data byte to generate (or regenerate) the check drives. Thus the cost of preparation for the failure of m drives is essentially the same as the cost of recovering from the failure of m drives: m additions and m multiplications.

4.3 System performance and GalOPS

Computational devices such as the CPU are capable of performing a certain number of Galois field operations per second. This determines the speed of Reed-Solomon style ECC. In a general purpose CPU the circuitry that performs integer arithmetic (modulo a power of 2, typically 2^k, where $k = 8$, 16, 32 and 64) is used to simulate the Galois field operations. GF(2^l) for small l is a finite set of cardinality small enough for a lookup table approach to multiplication and division. Additions may be directly performed by the existing integer arithmetic logic. Today's CPU's lack a dedicated hardware implementation of Galois field operations, but this situation may change. To make reasonable choices of the parameters of the system, ECC system designers face the problem of estimating how much computer resources the main algorithm would consume. This includes the silicon and CPU cycles. The absolute operation counts presented in this section can be a starting point of evaluating system performance. However, due to pipelining and branch prediction, more than 1 Galois field operation will be performed per clock cycle. In fact, the number can easily be as high as 16–32 on a real computer system, as evidenced by the data presented in Section 2.

We propose rating computer systems according to the number of Galois field operations per second that the system can perform. Necessarily, the measurement would need to be performed using some standardized benchmarking software.

In the past, similar synthetic measures have been devised for floating-point arithmetic, and the most popular measure in the High-Performance Community is the number of FLOPS, <u>FLO</u>ating point <u>OP</u>erations per <u>S</u>econd, that a processor can perform. For a 2.4GHz processor, it is not uncommon to achieve 25 GigaFLOPS, an order of magnitude difference between the clock speed and the FLOPS count, on standard Linear Algebra benchmarks (LINPACK).

A similar measure for the I/O subsystem only is called IOPS and is used in the IOMeter software originally from the Intel Corporation [8].

We propose a similar measure expressed in GalOPS, <u>Gal</u>ois <u>OP</u>erations per <u>S</u>econd (pronounced *G-ah-lops*), to evaluate the performance of a computer system from the point of view of its capability to perform Galois field arithmetic. Since the essence of the erasure coding algorithm we presented is linear algebra over the finite field GL(2^l) or in the vector space GL$(2^l)^k$, we could use the Reed-Solomon code implementation introduced in Section 2 as one such measure. Of course, the measure would depend on parameters, such as l and k.

In the era of virtual design, GalOP ratings of computer systems could not only be obtained for existing, but also future hardware (using existing simulators), and help in controlling costs before large systems supporting ECC, such as cloud infrastructures, are even built or deployed.

5 Design and Reliability of RAID

The engineering design of RAID systems is about increasing *reliability* by using *redundancy*, and yet it involves compromises, such as keeping the design cost down by using the smallest number of devices which satisfying the reliability objectives. We must have quantitative measures of reliability and, short of conducting massive, costly experiments, we must rely upon mathematical models to predict, for instance, what fraction of deployed RAID systems would fail in, say, 5 years.

A simple quantitative model of reliability of a RS(T, M) system should represent reliability as function of its fundamental parameters: the number of data drives N, the number of check drives M, the total number of drives $T = M + N$, and the reliability of a single device, represented by the *failure rate* λ, which is assumed constant. Modeling various scenarios leading to RAID failure is a subject of a significant number of publications [10, 11, 3, 6]. A popular reliability measure is the *mean time to data loss* (MTTDL). In the case of an isolated RAID system which can only fail due to random loss of devices happening with rate λ and which is never repaired, an exact formula for MTTDL exists [6]:

$$\text{MTTDL} \;=\; \frac{1}{\lambda}\sum_{k=0}^{M}\frac{1}{1-k/T}\,\frac{1}{T} \approx \frac{1}{\lambda}\int_0^{M/T}\frac{1}{1-u}\,du.$$

It is not hard to see that for large T the following approximation can be made:

$$\boxed{\text{MTTDL} \approx \lambda \log\frac{1}{R}.} \qquad (1)$$

where $R = \frac{N}{T}$ is the *rate*, the fraction of data devices in the RAID. Solving for R we obtain:

$$\boxed{R = e^{-\frac{\text{MTTDL}}{\lambda}}} \qquad (2)$$

This yields a simple, prescriptive rule which tells us what portion of the devices in our system should be data devices vs. check devices to achieve a desired MTTDL by determining R via the formula above. λ, the failure rate of an individual device is typically provided by the device manufacturer. Notice that λ is the inverse of the MTTDL for a single device. RAID failure modeling literature provides more rules of this sort, taking into account common sources of failure, such as sector errors, maintenance schedules and human error [6].

5.1 Benefits of $M > 2$

Note that 2 check devices are used in the familiar RAID 6 configuration. The main objection to systems with this few check devices is their vulnerability to not only device failure but also silent data corruption. We can easily evaluate the problem remembering the inequality $2t + f < M + 1$ from Section 3.5, which must hold in order to recover from errors. Thus, RAID with $M = 1$ cannot recover from just a single silent data error (unnoticed corruption of 1 byte of $M + N$ of the codeword). When $M = 2$, 1 failed device and 1 silent data error cause the system to lose data. Since it is expected that every RAID6 will have at least one failure in its lifetime, RAID6 is not resilient to silent errors. In fact, a RAID6 that encounters a silent error during reconstruction will silently corrupt additional data. To be resilient in the face of silent errors, which have been carefully documented in large systems [9], a minimum of M=3 check drives are required.

5.2 Large M solves the Big Data problem

The main benefits of a design that supports a higher check device count are lowered cost and increased reliability for large (Big Data) systems. By matching M with N we may build optimal storage systems for arbitrarily large data and arbitrarily strong reliability requirements, and thus solve the Big Data problem.

For example, no modern RAID vendor supports a RAID 6 configuration of 50 drives or more. They recognize that the likelihood of data loss is too high. Instead, they deploy less efficient, more costly configurations such as 10+2 (i.e. $M = 10$ and $N = 2$), replicated 5 times.

Using more check drives allows the deployment of larger configurations, for example, 50+5, that require fewer total components and need less frequent service. The service requirement can be delayed until the risk of data loss warrants a service call.

Larger systems with more check drives lower the acquisition, operation and service costs, while simultaneously increasing the protection from data loss.

6 Conclusions

The world is filled with sophisticated CPU devices. These range from those in our cell phones and digital cameras to our automobiles, airplanes, communication processors and business equipment. Today, it is normal for data to remain unprotected as it travels around these systems, and as such, is lost. Who has not seen "high-definition" video disintegrating to a collection of large, colorful squares[2], or heard audio dropouts, or experienced corrupted files? This state of affairs is no longer necessary to tolerate. Data need not be lost again[3]. With the addition of a simple software ECC algorithm, data can be protected from its "birth", whether it originates in a camera, or as the result of a business calculation, or as part of the research that will affect our future health and security.

We call our variant of this solution *Virtual ECC*. We identified how, utilizing the ubiquity of CPU's around us, to eliminate many sources of noise, both identified and silent, and provided hard measurements of real processors computing very reliable codewords. We described a practical implementation as a kernel module in the forthcoming Windows 8 platform. We presented benchmarks showing that the approach is not only viable, but it provides excellent performance and reliability characteristics, exceeding those of typical hardware ECC.

References

[1] ATTO Technology, Inc. Disk Benchmark—ATTO. http://www.attotech.com/products/product.php?sku=Disk_Benchmark, 2012. Software.

[2] Peter Gemmell and Madhu Sudan. Highly resilient correctors for polynomials. *Information Processing Letters*, 43(4):169–174, September 1992.

[3] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *HotStorage '10: 2nd Workshop on Hot Topics in Storage and File Systems*, Boston, June 2010.

[4] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–160, April 1950.

[5] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[6] Sarah Edge Mann, Michael Anderson, and Marek Rychlik. On the Reliability of RAID Systems: An Argument for More Check Drives. *arXiv.org*, cs.PF, February 2012.

[7] Microsoft. http://code.msdn.microsoft.com/windowshardware/WDKStorPortVirtualMiniport-973650f6.

[2] The proper term is "pixelation". There are other causes of pixelation besides data loss. Those cannot be addressed with ECC because they are artifacts of "normal" lossy codec operation, which gives priority to maintaining fixed bitrate over image quality.

[3] At least on time scales so long that an average person will *never* see important data loss during their lifetime.

[8] Intel Corporation (original version of 1998). Iometer. http://www.iometer.org/, 1998–2012. Software.

[9] Bernd Panzer-Steindel. Data integrity. Technical report, CERN/IT, April 2007.

[10] Jehan-François Paris, Ahmed Amer, Darrell D. E. Long, and Thomas J. E. Schwarz. Evaluating the Impact of Irrecoverable Read Errors on Disk Array Reliability. In *Proceedings of the IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC09)*, November 2009.

[11] Jehan-François Paris, Thomas J. E. Schwarz, Darrel D. E. Long, and Ahmed Amer. When MTTDLs Are Not Good Enough: Providing Better Estimates of Disk Array Reliability. In *Proceedings of the 7th International Information and Telecommunication Technologies Symposium (I2TS '08)*, December 2008.

[12] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. The MIT Press, Cambridge, Massachusetts, second edition, 1972.

[13] James S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[14] James S. Plank and Y. Ding. Note: Correction to the 1997 Tutorial on Reed-Solomon Coding. *Software – Practice & Experience*, 35(2):189–194, February 2005.

[15] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, June 1997.

[16] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.

[17] Lloyd R. Welch and Elwyn R Berlekamp. Error Correction for Algebraic Block Codes, December 1986. United States Patent. Patent number: 4,633,470.