Parallel coding for storage systems - An OpenMP and OpenCL capable framework

Peter Sobe
Faculty of Mathematics and Computer Engineering
Dresden University of Applied Sciences
Dresden, Germany
sobe@htw-dresden.de

Abstract: Parallel storage systems distribute data onto several devices. This allows high access bandwidth that is needed for parallel computing systems. It also improves the storage reliability, provided erasure-tolerant coding is applied and the coding is fast enough.
In this paper we assume storage systems that apply data distribution and coding in a combined way. We describe, how coding can be done parallel on multicore and GPU systems in order to keep track with the high storage access bandwidth. A framework is introduced that calculates coding equations from parameters and translates them into OpenMP- and OpenCL-based coding modules. These modules do the encoding for data that is written to the storage system, and do the decoding in case of failures of storage devices. We report on the performance of the coding modules and identify factors that influence the coding performance.

1 Introduction

Parallel and distributed storage systems are susceptible against faults due to their higher number of storage devices that all can fail or can become unaccessible temporarily. Thus, a combination with fault-tolerant coding, particularly erasure-tolerant coding is often applied. Codes are applied to calculate redundant data that is distributed in addition to the original data onto several failure-independent devices. That redundant data serves for the recalculation of 'erased' data that can not be read when devices fail or get disconnected.
There is a number of simple solutions, e.g. duplication of every data unit in a distributed system to another storage node. This introduces a high overhead in terms of storage capacity and a high write access load. Another simple solution is a parity code across all units that are distributed. The parity data is a kind of shared redundancy and can be applied to recalculate any data pice in case of a single failure. Erasure-tolerant codes are a generalization of the shared redundancy principle and are capable to tolerate a higher number of failures. Generally, codes base on a distribution of original data across k devices and a number of redundant data blocks that are placed on m additional devices (see Figure 1). It must be known which devices failed in order to decode the original data successfully.

This assumption is typically fulfilled within storage systems and differentiates the applied codes from general error-correction codes, e.g. codes for channel coding.

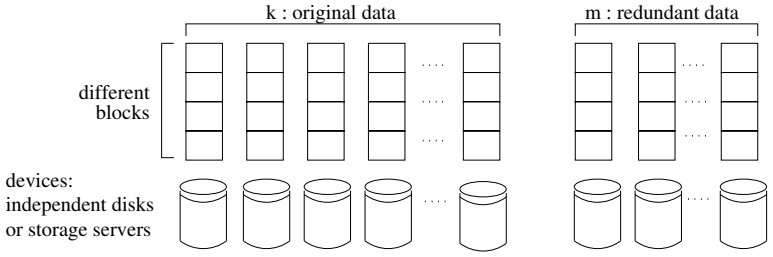

Figure 1: Data block distribution and redundancy used in parallel and distributed storage systems.

Some erasure-tolerant codes are optimal in terms of tolerated failures and storage overhead by allowing to tolerate every combination of up to m failed devices among these $k+m$ devices in total. The coding community investigated much research effort to find codes that show this optimal property for a large range of parameters k and m. Another criterion is the number of operations for encoding and decoding that should be as low as possible.
We already introduced an equation-oriented approach to erasure-tolerant coding in [SP08] that applies the Cauchy Reed/Solomon code arithmetics. Equations that calculate redundant data units by XORing original data units in an appropriate way define the functionality of the storage system. Initially, we provided these equations in data files in order to parameterize the en- and decoder of the storage system. The contribution of this paper is a proof of the concept that equations can be translated into programming language code directly. This code is enriched with expressions that control parallel processing, either in terms of data-parallel OpenCL kernel code, or in terms of OpenMP directives. These expressions are generated automatically.
The paper is organized as follows. Related work is surveyed in Section 2. The principle of equation-oriented en- end decoding is explained in Section 3 and in Section 4 we describe the translation to OpenCL and OpenMP code. A performance evaluation of our implementation can be found in Section 5. We conclude with a summary.

2 Related Work

Parallel storage systems that employ several storage devices and coding for fault tolerance first have been introduced with RAID systems [KGP89] in the context of several host-attached disks. This general idea later got adopted to networked storage. Later a variety of different codes were explored and applied for different

types of systems, e.g. networked storage, distributed memory systems or memories for sensor networks.

The Reed/Solomon code [IR60] (R/S) is a very flexible code that allows to construct coding systems for different distribution factors (k) and different amount of redundant data (m). R/S provides specific coding and decoding rules for any k and m, following a linear equation system approach. Originally, R/S requires Galois Field arithmetics and therefore needs more instructions and processing time on general purpose processors, compared to XOR-based codes that can directly use the processors XOR instruction. An XOR-based variant of R/S was introduced by Blomer et al. [BKK+95] and got later known as the so called Cauchy-Reed/Solomon code (CRS). This code divides each of the $k+m$ storage resources into ω different units (ω is chosen such that $2^\omega > k+m$ holds) that are individually referenced by XOR-based calculations. In our previous work on the NetRAID [Sob03, SP06] system an equation-based description of encoding and decoding was developed and allows a flexible use of different codes.

Equation-based coding strongly relates to the matrix-based coding technique that is supported by the jerasure library for erasure-tolerant codes [Pla07]. A binary code generator matrix selects Bits of the original data word to be XORed to the redundant Bits. Optimizations of the encoding algorithms and the creation of decoding algorithms are a result of matrix operations. The main objective is to find efficient codes with optimal failure-correction capabilities and minimal computation cost. In our tools we apply matrix-based techniques as well, but provide a textual description of coding algorithms that consists of equations over different Bits.

In an environment with parallel processes and parallel storage devices, it is necessary to exploit parallelism as well for storage coding to reach reasonable high coding throughput that keeps track with the desired high speed of the storage system. To use multi core processors is obvious. In addition, R/S and CRS have been offloaded to FPGA [HKS+02],[HSM08], GPU using NVidia CUDA [CSWB08] and other hardware [SPB10]. In [CSWB08] a GPU was evaluated for encoding a k=3, m=3 R/S code. It could be shown that the GPU's encoding rate is higher than the RAID level 0 aggregated write rate to the disks and coding keeps track with the pure disk system performance. The wide availability of multicore processors and OpenMP (Open Multi Processor) motivated further steps to run the coder as a multithreaded system.

Besides data parallelism as a straightforward way, further functional parallelism can be exploited in storage system coding. The functional parallelism is represented by the different equations for different redundant data units. For CRS, a number of $\omega \cdot m$ different redundant units can be calculated independently using individual XOR calculations which allows equation-based functional parallelism. A comparison between equation-oriented coding and data-parallel coding in [Sob10] revealed that equation-parallel coding improves the locality of data access for input and output data. Nevertheless, equation-oriented parallelism does not always produce an evenly balanced workload and requires a special choice of parameters to create evenly distributed encode equations.

3 Coding by Equations

The concept to describe encoding and decoding by XOR equations has been introduced in [SP08]. The equations are provided by a tool that includes all the CRS arithmetics and delivers the equation set for a storage system.

The naming of the units and the placement of the units on the storage resources is defined as follows. We place units $0, 1, \ldots \omega$-1 consecutively on the first original storage device, units ω to $2 \cdot \omega$-1 on the second device and so on. Each unit is denoted by the character 'u' and a number, e.g. $u0$ for the first unit in the system. The code calculations have to reference these units properly in the XOR equations. For the example with $k = 5$ and $m = 2$, the number of equations is 6. There is an individual equation for each of the 6 units. These 6 units are placed on two redundant storage devices (see Listing 1).

```
u15 = XOR(u2,u3,u4,u5,u7,u9,u11,u12)
u16 = XOR(u0,u2,u3,u7,u8,u9,u10,u11,u13)
u17 = XOR(u1,u3,u4,u6,u8,u10,u11,u14)
u18 = XOR(u0,u2,u4,u6,u7,u8,u11,u12,u13)
u19 = XOR(u0,u1,u2,u4,u5,u6,u9,u11,u14)
u20 = XOR(u1,u2,u3,u5,u6,u7,u10,u12)
```

Listing 1: Example for a coding scheme ($k = 5$, $m = 2$, $\omega = 3$).

The equations above allow to calculate every redundant unit independently from the other ones. Such a coding naively supports parallel processing, but contains redundant calculations, e.g. $XOR(u2, u3)$ is calculated 3 times. We call this the *direct coding style*. Another style of coding is called the *iterative coding style* that exploits previously calculated elements when possible. In that way, redundant calculations can be eliminated, e.g. $XOR(u2, u3)$ is stored in a temporary unit $t0$ and then referenced 3 times. Replacing all common subexpressions reduces significantly the number of XOR operations. For the $k = 5$, $m = 2$ system a reduction from 45 to 33 XOR operations occurred. For this example, the equations are given in Listing 2 with temporary units denoted with 't' and their number. The iterative equations can be formed from the equations given in the direct style using an automated preprocessing step.

Our approach is to translate the equations in a further processing step directly to OpenCL kernel code, or alternatively to OpenMP code. Both variants use extension of the C programming language. The generated code can be compiled to storage system components during system runtime. Particularly, at the time when a new failure situation is detected, the framework calculates new decoding equations to recalculate the missing data units from the other ones that are still available. A new decoder code can be generated from the decoding equations, translated to C code with parallel OpenCL or OpenMP extensions and then compiled to runtime components of the system.

```
u15 = XOR(t1,t3,t4)
u16 = XOR(t4,t6,t7)
u17 = XOR(u3,u4,u8,t6,t9)
u18 = XOR(u2,u4,u6,u7,t3,t7)
u19 = XOR(u0,u2,u9,u11,t1,t9)
u20 = XOR(u5,u7,u10,u12,t0,t8)
t0 = XOR(u2,u3)
t1 = XOR(u4,u5)
t2 = XOR(u7,u9)
t3 = XOR(u11,u12)
t4 = XOR(t0,t2)
t5 = XOR(u0,u8)
t6 = XOR(u10,u11)
t7 = XOR(u13,t5)
t8 = XOR(u1,u6)
t9 = XOR(u14,t8)
```

Listing 2: Coding scheme ($k = 5$, $m = 2$, $\omega = 3$)

4 Translation to parallel code

Our tool that generates the equations is capable to generate OpenCL kernel code and OpenMP program code as well. To do that, the user solely has to specify a few optional parameters, e.g. the file for code output and the unit length that is needed for addressing within the data arrays. This can be seen in the following command line of the tool:

```
./cauchyrs -k=5 -m=2 --ocl_encoder --ocl_file=crs5+2.cl
                     --ocl_unit_len=2048 --ocl_encstyle=iterative
```

The OpenCL code, generated from a CRS code with $k = 5$, $m = 2$ is listed in Listing 3. The unit numbers got translated into index values in order to address the data that relate to the unit. For instance $u0$ got translated to $n[0+i]$, $u4$ to $n[8192+i]$ and $u15$ to $r[0+i]$ by comparing the OpenCL code with the equations given in Listing 2. The constant offset in the index is calculated from the unit number and the length of the units, e.g the 5th unit with the index 4 points to 4×2048, when 2048 is the unit length. The redundant units with numbers $n : k \times \omega \leq n < (k+m) \times \omega$ are translated into elements within the r array (r denotes redundant data). The constant part is derived by $((n-k) \times \omega) \times$ unitlength. Every index contains a variable part i that addresses each individual byte in the unit. The XOR (operator symbol ^) causes that corresponding bytes of the different units in the C statements are XORed bitwise. Finally, a XOR operation on corresponding bits within the units takes place. A processing along different i values is done by the GPU that invokes the kernel code when the function

```
clEnqueueNDRangeKernel(...,ckKernel,1,NULL,&GlobalSize,&LocalSize,...)
```

```
// Parameters: k=5, m=2, w=3, OCL_UNIT_LEN=2048
__kernel void crs(__global const char *n, __global char *r)
{
  unsigned int i = get_global_id(0);
  char t21 = n[4096+i]^ n[6144+i];
  char t22 = n[8192+i]^ n[10240+i];
  char t23 = n[14336+i]^ n[18432+i];
  char t24 = n[22528+i]^ n[24576+i];
  char t25 = t21^ t23;
  char t26 = n[0+i]^ n[16384+i];
  char t27 = n[20480+i]^ n[22528+i];
  char t28 = n[26624+i]^ t26;
  char t29 = n[2048+i]^ n[12288+i];
  char t30 = n[28672+i]^ t29;

  r[0+i] = t22^ t24^ t25;
  r[2048+i] = t25^ t27^ t28;
  r[4096+i] = n[6144+i]^ n[8192+i]^ n[16384+i]^ t27^ t30;
  r[6144+i] = n[4096+i]^ n[8192+i]^ n[12288+i]^ n[14336+i]^ t24^ t28;
  r[8192+i] = n[0+i]^ n[4096+i]^ n[18432+i]^ n[22528+i]^ t22^ t30;
  r[10240+i] = n[10240+i]^ n[14336+i]^ n[20480+i]^ n[24576+i]^ t21^ t29;
}

// another kernel that works with a word len of 16 bytes
// and reaches slightly better performance
__kernel void crs16(__global const char16 *n, __global char16 *r)
```

Listing 3: Automatically generated OpenCL kernel.

is called by the host program. A number of *LocalSize* threads are started on the GPU multiprocessors and are supported by the SIMD-like data parallel execution technique. The GPUs used, allowed to start 512 threads (NVidia Quadro FX880M) and 1024 threads (NVidia Quadro 600). The parameter *GlobalSize* can express a higher thread number, that are run in a batch mode in groups of *LocalSize* threads.

In the same style like the OpenCL code generation we support OpenMP (Open Multi Processor language) programs for coding. OpenMP allows to create multithreaded processes from a sequential code by adding directives to the program. Typically, the workload of for-loops is distributed to several threads. OpenMP threads run on a shared memory and do not need to transfer any data before and after the multithreaded execution.
In the example, a C-program is written to the file omp5+2.c.

```
./cauchyrs -k=5 -m=2 --omp_encoder --omp_file=omp5+2.c
                     --omp_unit_len=2048 --omp_encstyle=iterative
```

The C program (see Listing 4) contains array index values instead of unit numbers. For OpenCL we generated macro code for the index. This allows to read the code like equations and find the unit numbers. The C preprocessor replaces the macro symbols with the macro expressions. At compile time, the constant part

of each index is calculated. The variable part i of an index is controlled by the for-loop during the runtime of the encoder. Where a common C program would run all the iterations from i=0 to i=unitlength-1, OpenMP delegates the loop to several threads that cooperatively run different index ranges. The OpenMP directive (#pragma omp ...) is generated automatically in the same way as the program code.
Variables that are in local use for each iteration have to be declared as private. For our application this applies to the character variables for the temporary units.

```
// Parameters: k=5, m=2, w=3, OCL_UNIT_LEN=2048
#define UNIT_LEN 2048
#define RUNIT(a) a*UNIT_LEN+i
#define OUNIT(a) a*UNIT_LEN+i

void calc(const char *n, char *r)
{
    int i;
    char t21, t22, t23, t24, t25, t26, t27, t28, t29, t30;

    #pragma omp parallel for private(t21, t22, t23, t24, t25, t26, t27,
        t28, t29, t30)
    for (i = 0; i < UNIT_LEN; i++)
    {
        t21 = n[OUNIT(2)]^ n[OUNIT(3)];
        t22 = n[OUNIT(4)]^ n[OUNIT(5)];
        t23 = n[OUNIT(7)]^ n[OUNIT(9)];
        t24 = n[OUNIT(11)]^ n[OUNIT(12)];
        t25 = t21^ t23;
        t26 = n[OUNIT(0)]^ n[OUNIT(8)];
        t27 = n[OUNIT(10)]^ n[OUNIT(11)];
        t28 = n[OUNIT(13)]^ t26;
        t29 = n[OUNIT(1)]^ n[OUNIT(6)];
        t30 = n[OUNIT(14)]^ t29;

        r[RUNIT(0)] = t22^ t24^ t25;
        r[RUNIT(1)] = t25^ t27^ t28;
        r[RUNIT(2)] = n[OUNIT(3)]^ n[OUNIT(4)]^ n[OUNIT(8)]^ t27^ t30;
        r[RUNIT(3)] = n[OUNIT(2)]^ n[OUNIT(4)]^ n[OUNIT(6)]^ n[OUNIT(7)]^
            t24^ t28;
        r[RUNIT(4)] = n[OUNIT(0)]^ n[OUNIT(2)]^ n[OUNIT(9)]^ n[OUNIT(11)]^
             t22^ t30;
        r[RUNIT(5)] = n[OUNIT(5)]^ n[OUNIT(7)]^ n[OUNIT(10)]^
                        n[OUNIT(12)]^ t21^ t29;
    }
}
```

Listing 4: Automatically generated OpenMP program.

5 Performance Evaluation

OpenCL always uses 'just in time' compilation of the GPU code. This means that during the operation of the storage system processes a new kernel code can be compiled and executed. Typically, when the storage system processes are started, the encoding algorithm is compiled into the run time components once. Later on, for encoding the data is transferred to the GPU, the kernel is invoked and the data is moved back to the host memory. An OpenCL process runs through the following phases:
(1) platform exploration and connecting to the GPU, (2) buffer management i.e. allocating GPU memory, (3) kernel compilation, (4) input data transfer, (6) kernel invocation and (7) result data transfer. We measured the time of these phases by a host program (see Listing 5).

```
OpenCL  K=3 M=2 w=3 UNIT_LEN=7168 wordlen=16
0.088277 seconds for platform exploration
0.000014 seconds for buffer management
0.189414 seconds for kernel compilation
0.000048 seconds for data transfer
0.000573 seconds for kernel execution and result return
Data rate incl. transfer: 99.058733 MiB/s
Data rate w/o. transfer: 107.341098 MiB/s
```

Listing 5: Example for an OpenCL kernel execution and the measured times.

The times measured for OpenCL phases of different runs are depicted in Figure 2 for a small system (code parameters $k = 1$, $m = 1$, $\omega = 1$, unitlen= 16 byte) that moves 1×16 Byte to the GPU, copies the 16 Bytes to the array of redundant bytes and moves $1\times$ 16 Byte back to he host memory. The comparison for a system with a higher distribution factor, more redundancy and a larger block size is given in Figure 3 (code parameters $k = 4$, $m = 2$, ω=3, unitlen= 32kByte). This coding scenario transfers $4 \times 3 \times 32kiB = 384kiB$ to the GPU memory and $2 \times 3 \times 32kiB = 192kiB$ The kernel executes 33 XOR operations for every 12 Bytes input and 8 Bytes output data. These are 1081344 Byte XOR operations in total. Both measurements were taken on a NVidia Quadro FX880M. The individual times for specific phases show that a bigger equation system causes a longer kernel compilation time. The other time values are dependent from the size of processed data.

A direct comparison of sequential processing, OpenMP processing with 4 threads and OpenCL processing on two different GPUs (GPU-1: NVidia Quadro FX 880M, GPU-2: NVidia Quadro 600) is given by the data throughput rates in Table 1. The execution times for sequential and OpenMP processing were measured on two different processors (CPU-1: Intel Core i5 M520, 2.4 GHz, CPU-2: AMD Phenom II X4, 840, 3.2 GHz). We calculated the redundancy for a $k = 4$, $m = 2$, $\omega = 3$ code with a unit length of 32 kiB and 64 kiB. The values are average times taken from 10 runs. The measurements for the direct encoding style were done with equations

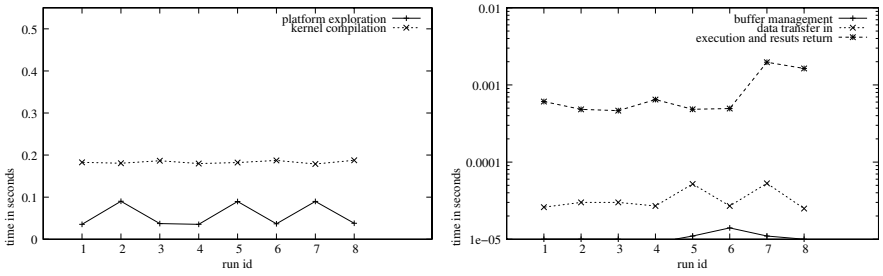

Figure 2: Minimal Code: Time consumption of phases for OpenCL processing on a GPU.

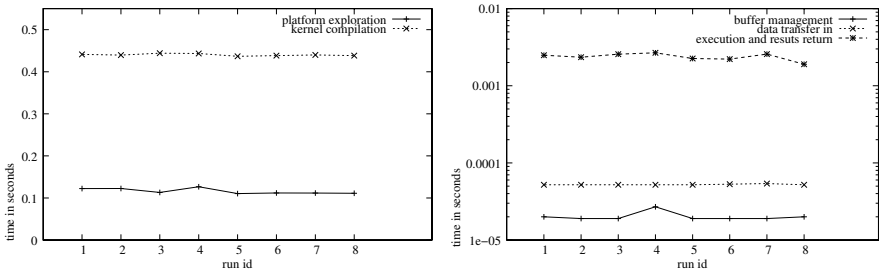

Figure 3: Bigger Code: Time consumption of phases for OpenCL processing on a GPU.

that still contained redundant calculations and are clearly disadvantageous for sequential processing.
OpenMP shows a moderate performance improvement. CPU-1 is a dual core processor that supports 4 threads. The measured speedup on that dual core system is 1.6 and 1.8. CPU-2 is a real 4-core system and the speedup factor is approximately 4 on that system. Besides of the different speedup, both CPUs reach nearly the same absolute performance when OpenMP processing is applied. This can be explained with the bigger cache of CPU-1 that improves the performance of each thread for this data-intense coding application.
The GPU performance is significantly better than sequential processing on the CPU, and better than multithreaded execution on the CPUs as well. However, it does not reach the theoretical performance of the GPU by far. This is caused by the data transfer between the host memory and the GPU memory.
When doing coding with a higher distribution factor and more redundant devices, the computational cost increases. Accordingly, the ratio between transferred data and computations is moved in direction of a bigger computational part. For sequential processing the data rates sink due to the higher computation cost. For GPU computing the disadvantageous cost of data transfer is assumed to diminish with increasing distribution and redundancy factors, due to the higher computational load that can be coped with by the highly multithreaded architecture.

unit length	encoding style	CPU-1		CPU-2		GPU-1	GPU-2
		sequential	OpenMP	sequential	OpenMP	OpenCL	
32 kiB	direct	124.3	189.7 (× 1.5)	44.9	179.3 (× 4)	224.4	299.9
	iterative	168.9	269.8 (× 1.6)	72.0	256.1 (× 3.5)	222.2	407.1
64 kiB	direct	111.3	181.2 (× 1.6)	54.6	181.4 (× 3.3)	273.2	318.6
	iterative	252.6	464.1 (× 1.8)	70.7	257.3 (× 3.6)	263.6	410.3

Table 1: Data throughput of encoding on different platforms in MiB/s.

6 Summary

OpenCL and OpenMP are appropriate platforms to implement software-based erasure-tolerant coding for storage systems. Because the erasure-tolerant codes strictly follow mathematical principles, particularly linear equation systems in case of the Cauchy Reed/Solomon code, the kernels of the coding programs can be generated in an automated way. We showed that an equation-oriented description of the codes can be easily translated to OpenCL and to OpenMP code. Fortunately, all the expressions to control parallel execution could be generated automatically as well.

OpenCL supports just in time compilation of GPU code which can be applied for a storage system to exchange coding modules during runtime. This is needed to insert new decoding algorithms in case of failures. The decoding algorithm adapts to the specific failure situation without requiring to run through control flow instructions. Because OpenCL is capable to run code on several platforms, especially on a multicore CPU device as well, it is a preferable platform compared to OpenMP. The performance evaluation revealed a moderate speedup for GPU processing using OpenCL and for multicore processing using OpenMP. We expect that GPU computing using OpenCL can reach to I/O bound (transfer bandwidth to and from GPU via the system interface) when all optimizations are applied.

References

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based Erasure–resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[CSWB08] M. L. Curry, A. Skejellum, H. L. Ward, and R. Brightwell. Acellerating Reed-Solomon Coding in RAID Systems with GPUs. In *Proceedings of the 22nd IEEE Int. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2008.

[HKS+02] A. Haase, C. Kretzschmar, R. Siegmund, D. Müller, J. Schneider, M. Boden, and M. Langer. Design of a Reed Solomon Decoder Using Partial Dynamic Reconfiguration of Xilinx Virtex FPGSs - A Case Study. 2002.

[HSM08] V. Hampel, P. Sobe, and E. Maehle. Experiences with a FPGA-based Reed/Solomon-encoding coprocessor. *Microprocessors and Microsystems*, 32(5-6):313–320, August 2008.

[IR60] G. Solomon I. Reed. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics [SIAM J.]*, 8:300–304, 1960.

[KGP89] R. Katz, G. Gibson, and D. Patterson. Disk System Architectures for High Performance Computing. In *Proceedings of the IEEE*, pages 1842–1858. IEEE Computer Society, December 1989.

[Pla07] J. S. Plank. Jerasure: A Library in C/C++ Fasciliating Erasure Coding to Storage Applications. Technical Report CS-07-603, University of Tennessee, September 2007.

[Sob03] P. Sobe. Data Consistent Up- and Downstreaming in a Distributed Storage System. In *Proc. of Int. Workshop on Storage Network Architecture and Parallel I/Os*, pages 19–26. IEEE Computer Society, 2003.

[Sob10] P. Sobe. Parallel Reed/Solomon Coding on Multicore Processors. In *6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*. IEEE Computer Society, 2010.

[SP06] P. Sobe and K. Peter. Comparison of Redundancy Schemes for Distributed Storage Systems. In *5th IEEE International Symposium on Network Computing and Applications*, pages 196–203. IEEE Computer Society, 2006.

[SP08] P. Sobe and K. Peter. Flexible Parameterization of XOR based Codes for Distributed Storage. In *7th IEEE International Symposium on Network Computing and Applications*. IEEE Computer Society, 2008.

[SPB10] T. Steinke, K. Peter, and S. Borchert. Efficiency Considerations of Cauchy Reed-Solomon Implementations on Accelerator and Multi-Core Platforms. In *Symposium on Application Accelerators in High Performance Computing 2010*, Knoxville, TN, July 2010.