2010 International Workshop on Storage Network Architecture and Parallel I/Os

Parallel Reed/Solomon Coding on Multicore Processors

Peter Sobe
Institute of Computer Engineering
University of Luebeck
Luebeck, Germany
Email: sobe@iti.uni-luebeck.de

***Abstract*—Cauchy Reed/Solomon is an XOR-based erasure-tolerant coding scheme, applied for reliable distributed storage, fault-tolerant memory and reconstruction of content from widely distributed data. The encoding and decoding is based on XOR operations and already well supported by microprocessors. On multicore processors, the coding procedures should also exploit parallelism to speed up coding.**

In this paper we derive coding procedures from code parameters (e.g. the number of tolerated failures) and propose their transformation into parallel coding schedules that are mapped on multicore processors. We (i) compare functionally decomposed coding procedures with data-parallel coding of different blocks, and (ii) specify the method to derive these schedules.

***Keywords*-Parallel Storage; Dependable Storage; Erasure-tolerant Coding**

I. Introduction

Erasure-tolerant coding is mainly applied for storage systems that spread data across several devices and tolerate device failures. Coding is applied in combination with a distribution of data elements across different devices, which provides the independence of failures and is the basis of the failure correction capability. When storage resources - mostly disks, but also solid-state based devices and memory - fail, the lost data elements are calculated using the remaining data elements and the redundant elements. The application field of such erasure-tolerant codes is wide and ranges from disk controllers to software-based distributed systems. For instance, enterprise server systems use specialized storage elements that implement erasure-tolerant coding on disk controllers using microprocessors or specific ASICs to do that task. Most of these systems are rather static, i.e. all the data is handled equally. In addition, relatively simple codes are applied, e.g. replication or pure parity codes with a single parity. It is hard to provide appropriate levels of failure tolerance for different classes of data with different importance and lifetime within a single storage system.

A software-based coding that relies on standard multicore processors tends to be more flexible. For example, a system can apply different strength of failure tolerance, depending on which filesystem directories are accessed an how critical the stored data is. Until the present days, microprocessors were not powerful enough to run application, operating system and do software-based coding together. This changed with multicore processors and continues in the manycore era. Particularly inhomogeneous manycore processors with a few complex cores and many of small and simple cores will be powerful enough to do data en- and decoding.

The contribution of the paper is an analysis of different parallelization approaches for Cauchy-Reed/Solomon codes. These codes are described by equations which are transformed into an iterative schedule and mapped onto a set of CPU cores.

Beyond, coding by interpretation of equations allows to structure a system into several components. Some components are dedicated to deliver the equations, and other components solely interprete equations. With respect to multicore and manycore systems, many cores can be used to interprete different equations and process data elements according to them - without relying on code-specific strategies.

The paper is organized as follows. Related areas - erasure tolerant coding, parallel storage and parallel code computations - are shortly revisited in Section II. In Section III we explain the principles of Cauchy-Reed/Solomon erasure-tolerant codes and their description by equations. Parallel coding is described in Section IV. We conclude with a summary.

II. Erasure-tolerant Coding

Erasure-tolerant codes are a special case of error correcting codes. Error correcting codes detect and correct a certain number of erroneous elements in a code word. When it is known, which elements are erroneous, an erasure-tolerant code can be applied. The error is treated as an erasure of data and the erased content is recalculated by the decoding procedure. Compared to error correcting codes, erasure-tolerant codes are a better choice in terms of storage overhead and calculation cost.

The correction capability relies on a limitation of the number of erroneous elements. Thus, data has to be spread across different data storage resources that do not fail simultaneously. In that way, k storage resources are used for original data elements. Redundancy is added and placed onto m additional storage resources.

A. *State of the Art*

For a low storage overhead, MDS (maximum distance separable) codes are preferred which are optimal with respect to their number of tolerable faults with a certain number of additional storage units. By MDS codes up to m failures can be tolerated among $n = k + m$ resources. The most flexible MDS codes are Reed/Solomon [16] codes that can be applied to any number of data and redundancy storage resources. A Reed/Solomon (R/S) implementation for disk arrays is described by Plank et al. [12], [14] using Galois field (GF) arithmetics. Cauchy-Reed/Solomon codes [2] allow to use XOR operations for en- and decoding. Cauchy-R/S combines the usage of a Cauchy matrix with an interpretation of data elements and matrix factors as Bit-vectors and Bit-matrices respectively. A proper partitioning of the input data in units, combined with the distribution onto several resources allows the use of bit-parallel XOR operations with the width of typically 64 or 128 bit. Cauchy-Reed/Solomon codes are widely applied, e.g. in the archival layer of the global storage system Oceanstore [11] and also in UpStore [8] with an extension for cryptography. These codes allow a relatively high encoding and decoding performance, but are still not optimal im terms of coding cost. Only a few particularly designed codes, e.g. EVENODD [1], the similar Double parity code and the Star code [7], show both low computation cost for encoding and the least possible storage overhead (the MDS property). The original en- and decoding algorithms for these codes are known as near-optimal. These codes work solely for a particular number of additional storage resources ($m \in \{2, 3\}$) and a restricted choice of k - and are not considered in this paper. A wide variety of XOR-based codes are implemented in the software-based distributed storage system NetRAID [17], [19]. In that system we implemented the equation-based description of encoding and decoding in order to allow a flexible use of different codes. Besides working with hardware accelerators [18], we currently are porting the system to multicore processors. Another direction towards generalized solutions is matrix-based encoding and decoding. An example is the jerasure[13] library that implements general matrix-based codes with optimizations for en- and decoding performance. The code can be flexibly parameterized by specifying a bit-level generator matrix. The decoding algorithm is calculated using the generator matrix for each failure case inside the library. This approach can be directly translated to the equations which are used in this paper.

B. *Coding Algorithms*

Throughout the paper we use the term *code* to specify the set of code words and the function to calculate code words from the original data words. *Encoding* is applied to calculate code words from the original data words. *Decoding* is the algorithm to recalculate the original data words from code words that are incomplete due to failures. Encoding and decoding can both be described by XOR equations, as shown in [20]. This requires a (1) data element placement on the storage resources and (2) to reference data elements properly in the XOR equations. The code type, execution time and resource requirements of coding procedures depend on these equations. For instance, encoding can be optimized to consume the least number of XOR operations on a single processor core, i.e. to consume as little computational resources as possible. In contrast, a fast encoder may run on several cores and does not rely on the least XOR operation count necessarily. We call this *coding style* - the way like the code calculations are structured. In the iterative coding style, the required elements are calculated serially, using previously calculated elements when possible. In contrast, the direct coding style allows to calculate each required element independently from other elements. Direct coding naively supports parallel coding, but contains redundant calculations. In contrast, a fully iterative coding algorithm eliminates all redundancy, but introduces data dependencies and communication.

C. *Parallel Storage and Parallel Coding*

Single storage devices are relatively slow, with a transfer rate of a few 100 MByte/s. This has to be compared to the memory bandwidth in the range of tens of GByte/s. Higher performance of storage systems mainly raises from parallel operation of several storage devices. Parallel storage devices first have been introduced with RAID systems [9] in the context of several attached disks and later adopted to networked storage in many different implementation variants, e.g. the Parallel virtual File system [4]. Even though devices are arranged to work in parallel, the accessing instances often operate in a sequential fashion. An exception are hardware-based accelerators that contain parallel structures. The software functions for erasure-tolerant en- and decoding are often still sequential procedures. Most research has been directed to find efficient codes with a low number of XOR operations and reasonable high failure-tolerance. Therefore, an obvious task is to exploit parallelism for accessing distributed data and for parallel coding on multicore processors. A recently developed driver [10] already implements Reed/Solomon coding on multicore systems according to the block level parallelism which is also described in this paper. Another recent field of research is on using graphic processing units (GPU) for error-tolerant coding. In [5] a GPU was evaluated for encoding a k=3, m=3 Reed/Solomon 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. In [3] a Linux block device is combined with a GPU for Reed/Solomon coding and provides considerable coding speedup related to a CPU-based implementation.

The coding principles in this paper have been implemented within a Linux block device driver [6], yet operating sequentially. This device driver is currently extended for parallel coding, based on equation-oriented parallel coding.

III. CAUCHY REED/SOLOMON

The original Reed/Solomon code and the XOR-based variant Cauchy-Reed/Solomon are flexible codes. that can handle any number of data storage resources (k) together with any number of redundant resources (m). At every resource, ω different units are required to express the code with equations ($2^\omega > n + m$).

The encoding equations are formed by interpreting the relation between data units and redundant units as a linear equations system. When the equation system is properly constructed, it can be used directly to decode erased data units, but also to provide equations to recalculate the erased units. This complies with the common way to formulate a code by a code generator matrix G. To obtain a code word a, the original data word a^* is multiplied (left sided) with G.

$$a = G \cdot a^*$$

Fig. 1 shows a k=5, m=2 code which is the example used to demonstrate en- and decoding. The generator matrix $G = \left\{ \begin{array}{c} I \\ C \end{array} \right\}$ consists of a 5×5 identity matrix I and $C = \left\{ \begin{array}{ccccc} 2 & 7 & 4 & 3 & 1 \\ 3 & 4 & 7 & 2 & 5 \end{array} \right\}$. For the example C is specifically selected and has to be a Cauchy matrix for a Cauchy-Reed/Solomon code. The arithmetics used correspond to a Galois field GF(2^3) using the modular polynomial $M(x) = x^3 + x + 1$. To apply this code with bit level arithmetics (logical AND and XOR), each element must be arranged as a block of 3 bits according to Tab. I. For instance, resource $r1$ is split into three units (3, 4 and 5) that are individual elements for the coding algorithm. In practice, a resource stores data in three separated parts, e.g. three partitions on a disk.

Correspondingly, every factor in C is mapped to a 3×3 bit matrix, according to a rule described in [2]. For the depicted example, encoding is completely covered by 6 equations that require a total number of 45 XOR operations. The required number of XOR operations depends from the modular polynomial and the generator matrix. By constructing proper generator matrices that contain a low number of 1-bits, but still provide a set of linearly independent equations, the computational cost of Cauchy-R/S is two to three times above the theoretical optimal cost[15].

	data					parities	
resource	$r0$	$r1$	$r2$	$r3$	$r4$	$r5$	$r6$
units	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14	15 16 17	18 19 20

Table I
UNIT ASSIGNMENT FOR A (5,2) CAUCHY R/S CODE

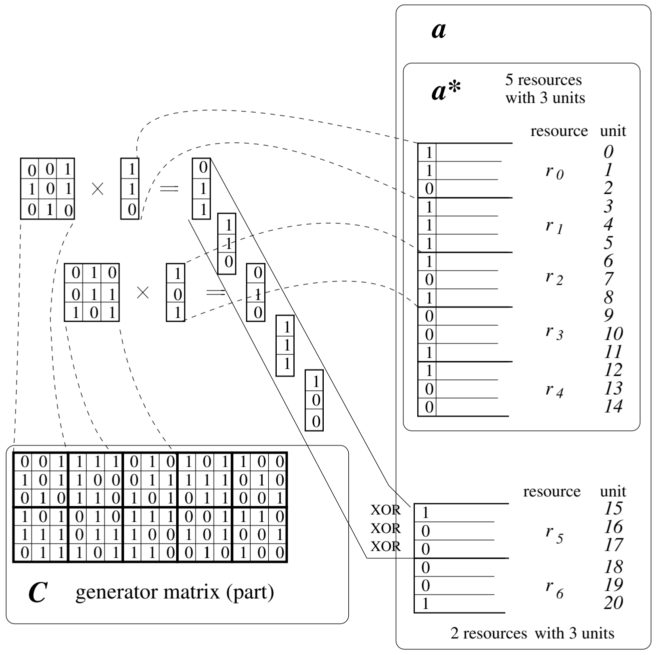

Figure 1. Cauchy Reed/Solomon Encoding

The following equations result from C and express a direct encoding style. It this style, each redundant unit is calculated independently from other redundant units. As input, solely units are used that represent original data content.

In order to express equations in a compact notation, we combine units to be xor-ed in an argument list for a XOR operator, i.e. XOR(3,5,8) is the expression for unit3 $\oplus$ unit5 $\oplus$ unit8. The equations refer the units by their number, i.e. the number 1 within the XOR arguments should be read as unit 1. The XOR operation is applied on the data onto the referred units.

```
(1) 15 = XOR(2,3,4,5,7,9,11,12)
(2) 16 = XOR(0,2,3,7,8,9,10,11,13)
(3) 17 = XOR(1,3,4,6,8,10,11,14)
(4) 18 = XOR(0,2,4,6,7,8,11,12,13)
(5) 19 = XOR(0,1,2,4,5,6,9,11,14)
(6) 20 = XOR(1,2,3,5,6,7,10,12)
```

Every Cauchy-R/S code contains $\omega \times m$ redundant units which each are calculated using an equation. In that way, the encoding calculations can be mapped to $\omega \times m$ processor cores, 6 cores for the example code.

During encoding a couple of XOR operations are executed redundantly, for instance XOR(4,9,11) is contained in equations (1) and (5). This is the motivation for another coding style, the so-called iterative style. By the iterative coding style, redundant elements are calculated using other redundant elements and temporary elements. This eliminates redundant calculations and reduces the total number of operations, which often leads to a faster parallel coding, compared to the direct coding style. The iterative encoding equations for the example code are derived as follows, using the symbols $A \ldots H$ for temporary elements.

```
(1) 15 = XOR(B,C,D)            (7) A = XOR(2,3)
(2) 16 = XOR(D,E,F)            (8) B = XOR(4,5)
(3) 17 = XOR(3,4,8,E,H)        (9) C = XOR(11,12)
(4) 18 = XOR(2,4,6,7,C,F)      (10) D = XOR(7,9,A)
(5) 19 = XOR(0,2,9,11,B,H)     (11) E = XOR(10,11)
(6) 20 = XOR(5,7,10,12,A,G)    (12) F = XOR(0,8,13)
                               (13) G = XOR(1,6)
                               (14) H = XOR(14,G)
```

This iterative encoding needs 33 XOR operations in total. Iterative equations can be automatically generated from the direct equations by identifying common subexpressions. Because of the lower number of XOR operations, sequential en- and decoders benefit from the iterative coding strategy. A possible order of equation interpretation is depicted in Fig. 2. Parallel coding also profits from iterative equations in terms of less XOR operations in total, but data transfer cost is introduced.

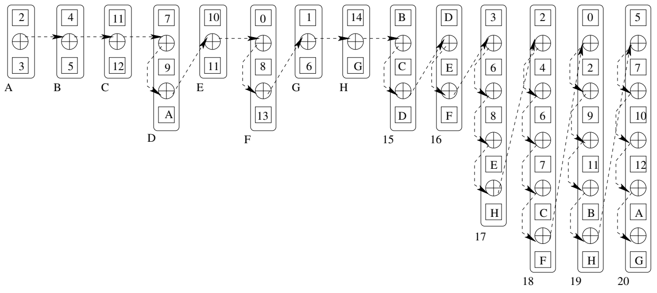

Figure 2. One possible sequential execution path through iterative encoding equations

Decoding equations are derived from the inverse of a sub matrix of $\left\{ \begin{array}{c} I \\ C \end{array} \right\}$, without rows that correspond to failed devices. While this is done within the Galois field interpretation, the inverse matrix is subsequently transformed to a bit-level matrix and then provides the equations for decoding. In that way, decoding is also an interpretation of XOR equations that can either follow the direct or the iterative style after a eliminating redundant computations. This allows to include solely a XOR equation interpreter into the storage system that is responsible for the en- and decoding of data. This part possibly operates for long time on a huge amount of data that is written to and read from the storage system. It has to be optimized for performance and therefore should exploit parallelism.

The *equation generator* is another component which does the matrix-based calculations and prepares the equations. The system structure which results from this separation of equation generation and equation-based coding is depicted in Fig. 3.

IV. Parallel Coding

With the presence of multicore and manycore systems, a parallel execution of the coding functions is strongly

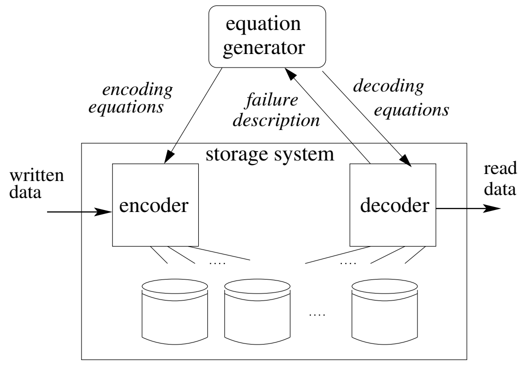

Figure 3. Separated equation generation and coding

motivated. It should be considered for performance reasons, but also to mitigate the lower clock frequency and stagnating performance of a single core. The speedup of the coding functions is mainly determined by a balanced separation of the computational workload across the cores. A division of workload should also divide the access to input and output data into smaller independent parts, because the performance is also limited by data access (access to storage resources, memory access and cache effects).

Parallel coding can be organized either by

- ***block level parallelism*** - calculating parts of redundant data independently from corresponding blocks of the input data, or
- by ***equation-oriented coding*** - assigning whole equations to different cores.

How good the workload can be balanced and how the references can be held locally an small data regions is studied in the following for several variants of parallel coding. For explanation the example with k=5, m=2 from Section III will be used.

A. Block Level Parallelism

This variant operates on several tiles of input data in a data-parallel fashion. Corresponding data blocks from data storage resources are collected and coded sequentially, as shown in Fig. 4. The iterative coding style should be preferred, due to less XOR operations for coding.

Conceptually, block parallel coding ideally exploits parallelism of the coding procedure. When p cores are used, each core covers $1/p$-th of the computational workload and accesses $1/p$-th of the input and output data amount. The size of the tiles can be scaled between a few bytes to very large blocks, depending on the access granularity. Nevertheless, these tiles always include all units of a stripe and accesses are always directed to all storage resources and to all partitions on a resource. Another observation is the average number of references to a specific unit when all iterative equations are calculated: surprisingly every unit is accessed almost two times (exactly two times in the example).

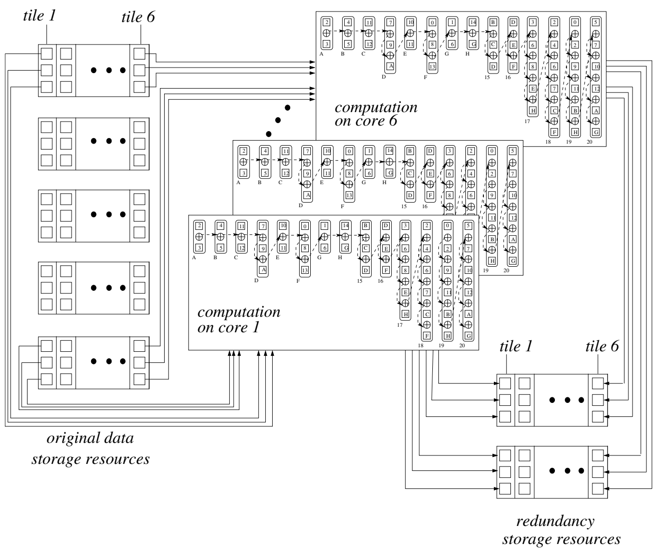

Figure 4. Block level parallelism

B. Equation-Oriented Parallelism

Alternative variants of parallel coding assign equations to individual cores, instead of dividing the data into blocks. For instance, the direct encoding equations from section III can be calculated on six cores independently. For iterative coding, every core is responsible for a number of related equations, as illustrated in Fig. 5. Exemplarily, lines connect cores with the accessed input data units and accessed non-local temporary units. A more detailed view on a possible schedule of equations on six cores is given in Fig. 6.

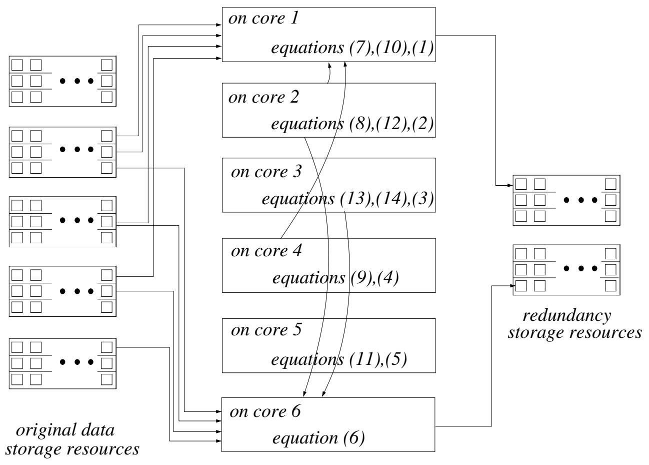

Figure 5. Equation-oriented parallelism

An equation-oriented coding causes a different access locality to the units compared to a block parallel coding. In Tab. II measures for several aspects of locality are defined with their relation to code parameters.

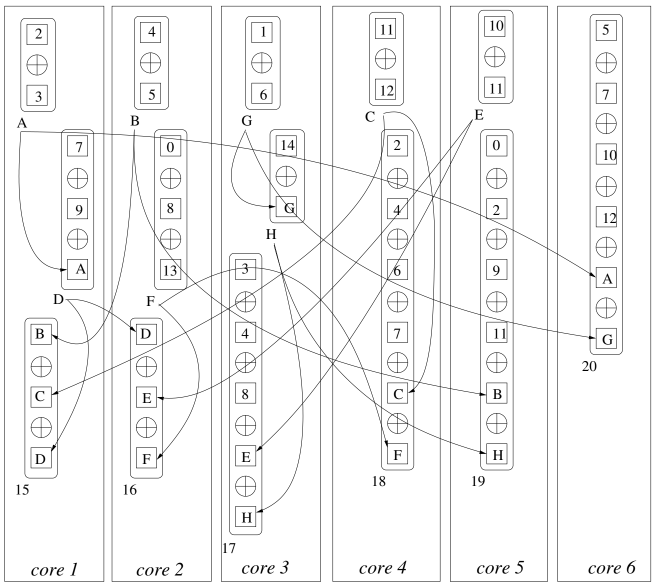

Figure 6. A possible schedule of iterative equations on six cores

locality measure	symbol	block-level	equation-oriented direct	equation-oriented iterative
accessed data (input and output)	a	$\frac{1}{p}$	$> \frac{1}{p}$	$> \frac{1}{p}$
number accessed resources (input)	r_{inp}	k	k‘ $k‘ \leq k$	k“ $k“ \leq k'$
number accessed resources (output)	r_{out}	m	1	1
average number of accessed units per resource[1]	u	all ω	$< \omega$	$< \omega$
average multiplicity of accesses[2]	m	> 1 higher	1 1	> 1 less
average number of accessed distant temporary units	d	0	0	> 1

Table II
LOCALITY MEASURES DERIVED PER CORE

A quantitative comparison of the equation-oriented coding variants to block parallel coding can be found in Tab. III and IV, specifically for the example code (Cauchy-Reed/Solomon, k=5, m=2). For comparison, the measures for block parallel coding are $a = 0.1\overline{6}$, $r_{inp} = 5$, $r_{out} = 2$, $u = 3$, $m = 2$, $d = 0$. For both variants of equation-oriented parallelism, the cores access a bigger share of the input data (a) compared to block level parallelism, but in turn the degree of multiple usage of a unit is reduced (m). Among the variants of equation-based coding, iterative coding takes fewer units for input (u) and concentrates accesses on fewer ressources (r_{inp}).

equation-oriented direct coding								
	core						average	relation to block level par.
	1	2	3	4	5	6		
a	$0.5\overline{3}$	0.6	$0.5\overline{3}$	0.6	0.6	$0.5\overline{3}$	$0.5\overline{6}$	$\times$ 3.4
r_{inp}	5	5	5	5	5	5	5	equal
r_{out}	1	1	1	1	1	1	1	$\times$ 0.5
u	1.6	1.8	1.6	1.8	1.8	1.6	1.7	$\times$ $0.5\overline{6}$
m	1						1	$\times$ 0.5
d	0						0	equal

Table III
PROPERTIES OF EQUATION-ORIENTED DIRECT CODING (k=5,m=2)

equation-oriented iterative coding								
	core						average	relation to block level par.
	1	2	3	4	5	6		
a	$0.2\overline{6}$	$0.\overline{3}$	0.4	0.4	0.4	$0.2\overline{6}$	$0.3\overline{4}$	$\times$ $2.0\overline{6}$
r_{inp}	4	4	4	4	3	4	$3.8\overline{3}$	$\times$ $0.7\overline{6}$
r_{out}	1	1	1	1	1	1	1	$\times$ 0.5
u	0.8	1	1.2	1.2	1	0.8	1	$\times$ $0.\overline{3}$
m	1	1	1	1	1.2	1	$1.0\overline{3}$	$\times$ 0.52
d	2	2	1	1	2	2	$1.\overline{6}$	+ $1.\overline{6}$

Table IV
PROPERTIES OF EQUATION-ORIENTED ITERATIVE CODING (k=5,m=2)

In the following, we propose a method to derive the parallel execution schedules from the equation set. It is used to prepare encoding schedules and decoding schedules as well.

C. *Schedules Derived from Direct Equation Sets*

Using direct equations, each equation can be assigned to a single core and all XOR operations of an equation are performed sequentially on a core. We call this equation-to-core mapping. The number of equations for encoding is $m \times \omega$ and is six for the example with m=2 and ω=3.
We illustrate a schedule by a number of horizontal time lines, a row for every core in the scheme which will be used for preparing the schedule.
Severals time lines in a vertical arrangement express multiple cores as shown in Fig. 7. For illustration, XOR operations are projected to time steps along a horizontal axis. The first XOR operation of an equation is assigned to the earliest step which is left-sided, the following operations to the time steps in right direction. The length of a schedule is determined by the longest equation.
The order how units are referenced in the slots is primarily not restricted, but certainly will be an issue for reference locality and cache behavior.

[1]Only the accessed ressources referenced units are included in the count of unit references.
[2]Only referenced units are included in the multiplicity average.

cores	steps							
	1	2	3	4	5	6	7	8
1	2 $\oplus$ 3	$\oplus$ 4	$\oplus$ 5	$\oplus$ 7	$\oplus$ 9	$\oplus$ 11	$\oplus$ 12	
2	0 $\oplus$ 2	$\oplus$ 3	$\oplus$ 7	$\oplus$ 8	$\oplus$ 9	$\oplus$ 10	$\oplus$ 11	$\oplus$ 13
3	1 $\oplus$ 3	$\oplus$ 4	$\oplus$ 6	$\oplus$ 8	$\oplus$ 10	$\oplus$ 11	$\oplus$ 14	
4	0 $\oplus$ 2	$\oplus$ 4	$\oplus$ 6	$\oplus$ 7	$\oplus$ 8	$\oplus$ 11	$\oplus$ 12	$\oplus$ 13
5	0 $\oplus$ 1	$\oplus$ 2	$\oplus$ 4	$\oplus$ 5	$\oplus$ 6	$\oplus$ 9	$\oplus$ 11	$\oplus$ 14
6	1 $\oplus$ 2	$\oplus$ 3	$\oplus$ 5	$\oplus$ 6	$\oplus$ 7	$\oplus$ 10	$\oplus$ 12	

Figure 7. Schedule for direct encoding

The locality of references to input and output units can be a reason for a faster multicore coding algorithm, as quantified in Tab. III.
The example also points out a possible problem of equation-based schedules - uneven utilization of cores. Cores 1, 3 and 6 perform one XOR operation less than cores 2, 4 and 5. This effect is present in spite of an optimal schedule, because 45 XOR operations on 6 cores lead to 7 XOR operations per core and three additional XOR operations. Solutions for this problem are the following. An even utilization can be reached by the selection of proper modular polynomials and Cauchy matrices, i.e. such ones that produce equations with equal computational effort (see IV-E). A necessary condition is that the number of XOR operations can be evenly divided by the number of cores. Another solution is to compensate the different computational effort on the cores by regularly changing the cores/equation assignment after coding a number of stripes.

D. *Iterative Equations Sets*

The mapping of an iterative equation set is directed to $m \times \omega$ different cores. For description, equations are divided in two classes: *terminal* and *temporary* equations. Terminal equations are those that produce a redundant unit as result. Temporary equations calculate temporary units, needed for two or more equations. The mapping to cores is a stepwise stacking of equations into the time lines on the cores. For preparation of the schedules a scheme is used that contains the core's time lines in the upper part. The bottom part of the scheme is used to list reference times to temporary units and the step when a unit becomes available. During schedule preparation, reference times may change, when earlier temporary equations are included and new references are added. The output of the schedule preparation is the upper part of the scheme which can be directly interpreted by the encoder and decoder procedures at the cores.
The preparation consist of the following phases: (1) equation preparation, (2) terminal equation mapping, (3) a check for unresolved temporary units, (4) temporary equation mapping and (3) a check of the schedule whether all temporary units are available prior references to them. Steps (4) and (3) are repeated until all required temporary equations are mapped. Finally equations are (5) backfilled in the scheme. The phases are described in detail in the following.

Equation preparation: Temporary unit references are placed on the rightmost positions in all equations.

Terminal Equation Mapping: Every terminal equation is assigned to a distinct core, similarly to equation–to–core mapping. During the preparation phase, shorter equations are shifted in right direction the end together with the longest equation. The intermediate schedule after the terminal equation mapping step is shown in Fig. 8.

cores	steps										
		1		2		3		4		5	
1							B	⊕	C	⊕	D
2							F	⊕	E	⊕	D
3			3	⊕	4	⊕	8	⊕	E	⊕	H
4	2	⊕	4	⊕	6	⊕	7	⊕	C	⊕	F
5	0	⊕	2	⊕	9	⊕	11	⊕	B	⊕	H
6	7	⊕	5	⊕	10	⊕	12	⊕	A	⊕	G
temporary units required								B,A, F,C, E		D,H, G	
temporary units available											

Figure 8. Schedule for iterative encoding after mapping of terminal equations

Check: For the required temporary units it is checked, if and in which time step these units are available. If a required unit is not yet calculated by a temporary equation, the corresponding temporary equation is mapped. If a temporary unit gets available at a later time than referenced, one of the constraints used for mapping is violated and the equation is mapped on earlier time steps.

Temporary equation mapping: All other equations used for calculation of temporary units are stacked left-sided to the terminal equations. For selection of temporary equations, a couple of choices and constraints apply:

- *choice A*: long temporary equations should be mapped first and mapped to the cores that offer the most unused time steps.
- *choice B*: Temporary equations which refer to other temporary units should be mapped earlier. The result is a placement at later timesteps in the schedule.
- *constraint A*: When the latest time step of an inserted equation is $t_{insert}(u_t)$ and the temporary unit u_t is calculated, then u_t may not be referenced before $t_{insert}(u_t) + 1$. If this constraint is violated, the temporary equation has to be scheduled for an earlier time period. This is managed either by selecting another core for mapping, or by choosing another equation for the next mapping step.
- *constraint B*: Data dependency - A reference to a temporary unit u_{tmp} requires that this unit has been already calculated by an equation in an earlier time step. If this is not the case, this is detected during insertion, by existence of the temporary equation for u_{tmp} that ends on a later time step. A way to cope with the detected problem is to mark the dependency and undo all mappings until to the mapping of equation for u_{tmp}. Then the mapping is redone considering the dependency by mapping the referencing equation first.

Mapping and checking steps are repeated for all remaining temporary equations until all equations are mapped. In the example, the first temporary equation is equation 10 (D=XOR(7,9,A)) that is mapped to core 1. This equation is selected first, because it is one of the longest equation and contains another temporary element. The second equation selected for mapping is (12) (F=XOR(0,8,13)), because of its length. The intermediate schedule is shown in Figure 9.

cores	steps										
		1		2		3		4		5	
1							B	⊕	C	⊕	D
			7	⊕	9	⊕	A				
2							F	⊕	E	⊕	D
			0	⊕	8	⊕	13				
3			3	⊕	4	⊕	8	⊕	E	⊕	H
4	2	⊕	4	⊕	6	⊕	7	⊕	C	⊕	F
5	0	⊕	2	⊕	9	⊕	11	⊕	B	⊕	H
6	7	⊕	5	⊕	10	⊕	12	⊕	A	⊕	G
temporary units required						A		B,~~A~~, F,C, E		D,H, G	
temporary units available	D,F										

Figure 9. Schedule for iterative encoding after mapping of temporary equations (10) and (12)

The next insertion is equation (14) H=XOR(14,G) on core 3. Core 3 is selected, because H is referenced by equation (3) is already assigned to core 3. Further insertions follow: equations (7) (A=XOR(2,3)) on core 1, (8) B=XOR(4,5) on core 2, (9) C=XOR(11,12) on core 5, (11) E=XOR(10,11) on core 4 and (13) G=XOR(1,6) on core 3.

Backfilling - When all equations are mapped, equations are shifted leftwards. This causes that each equation is executed as soon as possible. Shifting is allowed only when all references to temporary variables do not violate data dependencies. Concretely, all equations must be rechecked for a backfilling possibility as long temporary equations are shifted. The backfilling terminates, when there is no equation that can be shifted anymore. In the example, the equations in slot 1, 2 and 6 can be shifted to earlier time steps.

E. Improving Equations Sets

Even for iterative coding with well designed schedules, uneven utilization of cores may occur. Because of that, one should vary the Cauchy matrix and the modular polynomial to find equation sets that can be evenly distributed on a given number of cores.

For k=5, m=2 and six cores we found a couple of Cauchy

cores						steps							
		1		2		3		4		5		6	
1									B	⊕	C	⊕	D
					7	⊕	9	⊕	A				
			2	⊕	3								
2									F	⊕	E	⊕	D
					0	⊕	8	⊕	13				
			4	⊕	5								
3					3	⊕	4	⊕	8	⊕	E	⊕	H
			14	⊕	G								
	1	⊕	6										
4			2	⊕	4	⊕	6	⊕	7	⊕	C	⊕	F
	10	⊕	11										
5			0	⊕	2	⊕	9	⊕	11	⊕	B	⊕	H
	11	⊕	12										
6			7	⊕	5	⊕	10	⊕	12	⊕	A	⊕	G
						temporary units required							
				G				A		B,~~A~~, F,C, E		D,H, ~~G~~	
						temporary units available							
			G,C, E		A,B, H				D,F				

Figure 10. Schedule for iterative encoding after mapping of all temporary equations

cores						steps							
		1		2		3		4		5		6	
1							B	⊕	C	⊕	D		
			7	⊕	9	⊕	A						
	2	⊕	3										
2							F	⊕	E	⊕	D		
			0	⊕	8	⊕	13						
	4	⊕	5										
3					3	⊕	4	⊕	8	⊕	E	⊕	H
			14	⊕	G								
	1	⊕	6										
4			2	⊕	4	⊕	6	⊕	7	⊕	C	⊕	F
	10	⊕	11										
5			0	⊕	2	⊕	9	⊕	11	⊕	B	⊕	H
	11	⊕	12										
6	7	⊕	5	⊕	10	⊕	12	⊕	A	⊕	G		
						temporary units required							
				G		A		B,~~A~~, F,C		D,~~G~~, E		H	
						temporary units available							
			A,B, C,G,E		H		D,F						

Figure 11. Schedule after backfilling

matrices and modular polynomials that led to better iterative equations than in the example. Particularly equation sets with a number of XOR operation that can be divided by the number of cores without rest are good candidates. When using the polynomial $M(x) = x^3 + x^2 + 1$ and the Cauchy matrix $\left\{ \begin{array}{ccccc} 4 & 5 & 1 & 6 & 1 \\ 5 & 4 & 7 & 2 & 6 \end{array} \right\}$ we obtain equations with 30 XOR operations in total:

```
15 = XOR(D, E)                  A = XOR(1,2)
16 = XOR(2, 5, F, G)            B = XOR(3, 4)
17 = XOR(9, 13, B, D, H)        C = XOR(5, 6)
18 = XOR(0, 13, A, C, G)        D = XOR(14, A)
19 = XOR(7, 8, C, D, F)         E = XOR(10, B, C)
20 = XOR(1, 11, 12, E, H)       F = XOR(9, 12)
                                G = XOR(4,7,11)
                                H = XOR(0, 8)
```

These equation can be mapped onto 6 cores as shown in Fig. 12 with 5 XORs on every core.

cores					steps						
		1		2		3		4		5	
1									D	⊕	E
					10	⊕	B	⊕	C		
	4	⊕	7	⊕	11						
2					2	⊕	5	⊕	F	⊕	G
			14	⊕	A						
	1	⊕	2								
3			9	⊕	13	⊕	B	⊕	D	⊕	H
	3	⊕	4								
4			0	⊕	13	⊕	A	⊕	C	⊕	G
	5	⊕	6								
5			7	⊕	8	⊕	C	⊕	D	⊕	F
	0	⊕	8								
6			1	⊕	11	⊕	12	⊕	E	⊕	H
	9	⊕	12								

Figure 12. Schedule for an improved code

The improved equation set shows better properties even when used for block parallel coding. The unit accesses could be reduced, which is expressed by $m = 1.73$ (unit access multiplicity), compared to $m = 2.0$ in the basis variant. The rest of the measures (a, r_{inp}, r_{out}, u, d) do not change (see IV-A). The improvement of the equation-oriented iterative schedule is expressed by the measures given in Tab. V. Compared to Tab. IV, the amount of accessed data (a) is reduced and less resources need to be accessed by a core (r_{inp}). Whereby these two measures could be decreased, other measures are only slightly increased. The multiplicity of accesses increases from $m = 1.03$ to $m = 1.13$, and the non-local references increase (d). Taking the better utilization of the cores by the improved equations into account, the improvements prevail.

equation-oriented iterative coding (improved eqn.)								
	core						average	relation to block level par.
	1	2	3	4	5	6		
a	$0.2\overline{6}$	0.3	$0.2\overline{6}$	$0.2\overline{6}$	0.2	$0.2\overline{6}$	$0.2\overline{6}$	$\times 1.\overline{6}$
r_{inp}	4	3	3	4	2	3	$3.1\overline{6}$	$\times 0.6\overline{3}$
r_{out}	1	1	1	1	1	1	1	$\times 0.5$
u	$1.\overline{3}$	$1.\overline{3}$	$1.\overline{3}$	1	1.5	$1.\overline{3}$	1.3	$\times 0.4\overline{3}$
m	1	1.25	1	1	$1.\overline{3}$	1.2	1.13	$\times 0.65$
d	2	2	2	2	3	2	$2.1\overline{6}$	$+2.1\overline{6}$

Table V
IMPROVED EQUATION-ORIENTED ITERATIVE CODING (k=5,m=2)

F. Comparison

The quality a parallel coding is characterized by two aspects. The first characterizes how uniform the workload can be distributed. The other aspect is the access locality to input and output data. These aspects are used for a comparison of block level parallelism and equation-oriented parallelism with iterative equations. The explanation is based on the values given in Tab. III. We exclude the direct equation variant because of the high redundancy in calculations which also influences data access in a disadvantageous way.

Workload distribution:

- ***block parallel coding*** - As long as large amounts of input data are processed, the workload is distributed evenly. Assuming a workload unit of a single core of *chunksize*, the input data size divided by the number of cores must be bigger than chunksize. An uneven distribution may occur only for small accesses or for offcuts.
- ***equation-oriented iterative coding*** - An uneven distribution is often the case when an arbitrary Reed/Solomon code is applied. To reach an even distribution of workload, the equation set must be specifically constructed. This can be done for encoding by choosing the specific code by the modular polynomial and one out of many possible Cauchy matrices. For decoding, the code is fixed and a comparable choice is not available. When the number of faulty devices f is less than m, one can choose to use k from k+m-f devices for decoding. This gives several variants for decoding that differ in computation effort and also differ in terms of workload distribution. It is worth to vary the devices used for decoding to find a proper decoding equation set that allows even distribution.

Access locality:

- ***block parallel coding*** - Every core accesses a part of the input and output data which is the total data size divided by the number of cores. In terms of size this is the best possible result. However, the accessed data by a single core is scattered over all devices and all units on them. In addition, data units are typically accessed several times by different equations, in average every unit is accessed almost two times. This requires buffer space or several subsequent accesses, which relativizes the access size aspect.
- ***equation-oriented iterative coding*** - The data size that is accessed by a core is about two times larger, compared to block parallel coding ($1.\overline{6}$ for the improved schedule). But this relates to the multiple accesses to a unit when block parallel coding is applied. Equation-oriented coding accesses data units mostly one time (in average $1.0\overline{3}$ for the example code). This results in roughly the same accessed data amount for both variants when there is no local buffering of input data. A clear advantage of equation-oriented coding is that only a small number of storage devices is accessed from a core. If a device is accessed, then also less units are used for coding on a core compared to block parallel coding.

It is worth to invest further work to equation-oriented coding. A proper choice of the code and a preparation of the equations for an even workload distribution is necessary.

V. Summary

In this paper variants of parallel coding for reliable storage systems have been described. We started from a Cauchy-Reed/Solomon code and decomposed the coding into several parts, described by equations. A distribution of these equations on a number of processor cores is one variant of parallel coding. When the code is chosen properly, the (i) workload of equations can be evenly distributed and (ii) access to input and output data for coding is held local on specific data units for every processor core. Compared to block level parallelism, equation-oriented parallelism concentrates the data accesses of a core on specific regions of data that is associated with partitions or logical devices. This will be an advantage for software-based coding performance on multicore processors.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures . *IEEE Transactions on Computers*, 44(2), February 1995.

[2] 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.

[3] A. Brinkmann and D. Eschweiler. A Microdriver Architecture for Error Correcting Codes inside the Linux Kernel. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009*. ACM, 2009.

[4] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[5] 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.

[6] O. J. Frahm and P. Sobe. A Block Device Driver for Parallel and Fault-tolerant Storage. In *PARS Workshop 2010: ARCS '10 Workshop Proceedings*. VDE Verlag, Berlin, Offenbach, 2010.

[7] C. Huang and L. Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[8] T. Jing, Z. Yang, and Y. Dai. SEC: A Practical Secure Erasure Coding Scheme for Peer-to-Peer Storage Systems. In *14th Symposium on Storage System and Technology*, 2006.

[9] 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.

[10] H. Klein and J. Keller. Storage Architecture with Integrity, Redundancy and Encryption. In *Proceedings of the 23rd IEEE Int. Parallel and Distributed Processing Symposium, DPDNS Workshop*. IEEE Computer Society, 2009.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*. ACM, Nov. 2000.

[12] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *SOFTWARE - PRACTICE AND EXPERIENCE*, 27(9):995–1012, September 1997.

[13] 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.

[14] J. S. Plank and Y. Ding. Note: Correction to the 1997 Tutorial on Reed-Solomon Coding. Technical Report CS-03-504, University of Tennessee, April 2003.

[15] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2006.

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

[17] 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.

[18] P. Sobe and V. Hampel. FPGA-Accelerated Deletion-tolerant Coding for Reliable Distributed Storage. In *ARCS 2007 Proceedings*, pages 14–27. LNCS, Spinger Berlin Heidelberg, 2007.

[19] 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.

[20] 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.