Coding for Reliable Data Storage on Different Hardware Platforms

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

Abstract

When systems are designed to tolerate faulty components, application data must be protected against loss. This is reached by a distribution of data together with addition of redundant elements according to an erasure-tolerant code. In this paper, we elaborate architectures for such a fault-tolerant data storage. The concepts are originated from distributed systems and mostly implemented by software. We extend these concepts for usage in the scope of system on chip architectures. On the one hand, systems on chips, and multi core systems are employed as a platform for code calculation - on the other hand, such architectures include these techniques to fulfill their own functionality. We explain how data coding is mapped to (i) multi core CPU structures and (ii) implemented in a specialized design on a FPGA. We compare the time for coding on these architectures for a Cauchy-Reed/Solomon and a classical Reed/Solomon code.

1. Introduction

Coding of data with erasure-tolerant codes protects against data loss in systems composed of several components that fail independently. Such systems range from networks of computers to system-on-chip architectures (SoC). For coding, data is split into elements and distributed across k regions. Additional redundant elements are added by the code and distributed across additional m regions. Well designed codes allow to tolerate up to m failures, i.e. regions that are faulty and indentified by a diagnosis system. A typical application field are distributed mass storage systems that can tolerate failures of storage resources. These resources are mostly hard disks, but also solid-state-based storage devices. The loss of data is tolerated by decoding the remaining data and redundancy pieces to the original data. As well, data at several locations within a system on chip (SoC) can be the subject of erasure-tolerant data coding. Therefore, we use the term *storage locations* for the sake of universality, which includes memory as well. Particularly, future systems contain several independent functional units and storage locations that are connected by an on-chip network.

The demand for such a failure-tolerant data storage (and memory) originates from the trend of shrinking semiconductor feature sizes and high clock frequencies. This causes unreliable operation that manifests in failures of functional units and data corruption as well. In such a situation, the faulty functional units need to get deactivated. When a unit acts as a storage location, the contained data is lost. In addition, chip regions can be deactivated by a management component, due to overheating effects or power saving reasons. All these phenomena require the system to deal with partial failures and to provide fault-tolerance techniques.

However, erasure-tolerant coding was often considered as too time consuming and too resource-consuming. This changed with the availability of multi-core processors, multi-core SoCs and with the integration of FPGAs into computing systems. These architectures offer a sufficiently high number of resources (processor cores, logic cells, etc.) to implement data encoding and decoding fast enough. Moreover, the calculations can be efficiently mapped to the structures for parallel operation.

In this paper, we show how to express codes by equations. These equations can be mapped to several architectures. Their interpretation covers the core calculations for coding that are tailored for the specific data layout and computing platform.

The paper is organized as follows. Section 2 introduces into erasure-tolerant coding. How these codes are implemented in different hardware platforms is described in Section 3. In Section 4 the coding cost on a single CPU system, a multi core system and a FPGA-based system is compared.

2. Erasure-tolerant data coding

Codes are typically specified using xor operations that ara applied across selected bits of the original data. This results in a number of redundant bits. Most erasure-tolerant codes are so-called systematic codes, that store original data bits and redundancy bits at different locations. A wide variety of xor-based codes can be described in this way. Common examples are the parity code, the Hamming code[4], Evenodd[1], or Cauchy-Reed/Solomon[2]. For a distributed storage system, we developed a xor-based description of the encoding and decoding calculations[10]. Such a description not only allows to parameterize which code is applied, it also defines the style of coding. This style defines either iterative calculations of redundant elements or calculations of redundant elements independently from each other.

All codes described by xor-based equations can be categorized by the horizontal-vertical scheme (HoVer)[3]. It is a common reference scheme that unifies most xor-based codes by defining the element placement pattern and the calculation algorithm for redundant elements. Data elements are arranged in a two-dimensional array spanned by (i) different storage locations and (ii) several elements on each location. This allows to define codes that combine selected elements from different locations, the so called two-dimensional codes. Specifically, original data is arranged in a $k \times \omega$ array in this scheme, and the horizontal redundancy in a $m \times \omega$ array. The parameter ω specifies how many different bits of a code word are placed at a storage location. For larger data sets, multiple code words are mapped onto storage locations in a repet-

itive way. When $\omega = 1$, i.e. only one dimension is required, the codes are called one-dimensional ones. Examples for one-dimensional codes are the parity code and Hamming codes. Two dimensional codes are for instance Evenodd and the Cauchy-Reed/Solomon code (CRS). HoVer specifications can be easily transformed to our equation-based code description.

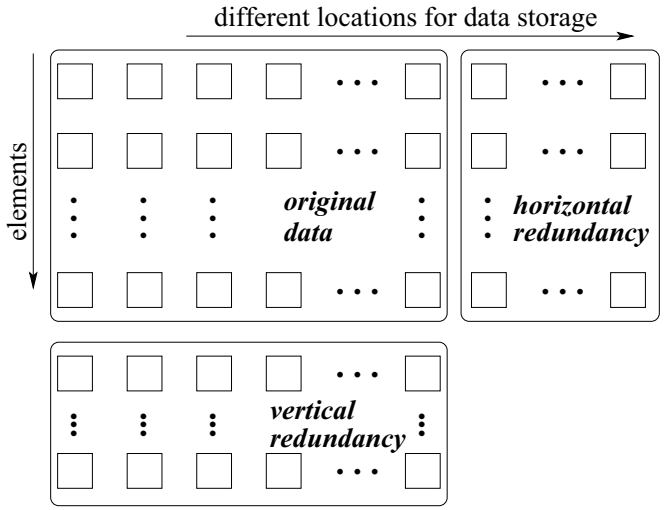

Figure 1. Horizontal-vertical scheme

The xor equations are provided by a dedicated component, which either is a service used by a storage system, or a dedicated functional unit on a chip. The equation generation can be a complex task, compared to the raw interpretation of the xor equations.

3. Architectures supporting coding

It will be explained how to map the equation-based code description to several hardware platforms. A Cauchy-Reed/Solomon code (CRS) is used - classified as a 2-dimensional xor-based code.
In 3.1, coding on a multi-core CPU is explained. Another platform, a FPGA-based hybrid computing system, is focused in 3.2. Both architectures contain computing elements that are used for calculation according to equations. Storage locations are another type of elements, e.g. internal memory in a SoC or external memory and storage. Possibly, computing elements and storage locations can be placed together on a common unit, e.g. on a processor core with local memory.

3.1 Case study: Multi-core CPU

Typical multi core CPUs combine several processor cores on a chip. Today this is a number between 2 and 8 cores. Mostly, the cores are connected by a hierarchy of split and shared caches and share an interface to external memory. A few systems also contain local memory for the processors, for instance the CellBE chip. Multi-core CPUs allow to use the cores for the execution of different coding equations in parallel. When there is local memory, storage locations can be included in the system, such as assumed for our design.

A particular case for data storage within a multi core CPU will be explained for a CRS code with $k = 5$ data storage locations and two locations for redundant elements ($m = 2$). The data elements have to be arranged in a 5×3 array with $\omega = 3$ elements on each storage location. There are six redundant elements that are correspondingly distributed across two additional storage locations. We refer the elements with numbers: elements 0 to 14 are the data elements, the elements 15 to 20 are the redundant elements. According to an algorithm described in [2, 8] and particularly chosen generator parameters, encoding equations are found as follows:

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

These equations are independent and six cores can be used to calculate each redundant element independently from another. This direct encoding requires 45 xor operations in total with 8 operations for the most costly equation. Compared to encoding on a single core, a speedup of 5.6 is reached on 6 cores, with an efficiency of 0.93.
Direct encoding includes redundant operations. These can be eliminated by changing to a so-called iterative coding style. In this case, the redundant elements are calculated using other redundant elements and temporary elements. This eliminates redundant calculations, but also introduces dependencies and limits the parallel execution of calculations. On the other hand, the total number of operations is reduced, which often leads to a faster coding in result. The iterative encoding equations for the same code (CRS, $k = 5$, $m = 2$) are derived as follows, using the symbols $A \ldots H$ for temporary elements.

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

eq. 7: A = XOR(2,3)
eq. 8: B = XOR(4,5)
eq. 9: C = XOR(11,12)
eq.10: D = XOR(7,9,A)
eq:11: E = XOR(10,11)
eq:12: F = XOR(0,8,13)
eq:13: G = XOR(1,6)
eq:14: H = XOR(14,G)
```

This iterative encoding requires 33 xor operations in total. The longest path consists of 7 xor operations. The iterative style provides a faster and more efficient calculation of the redundant elements. Figures 2 and 3 show the regions for data storage, regions for code calculation and the data communication path for the two coding styles.
The equations that are prepared for a system can be tailored specifically for an available number of cores.

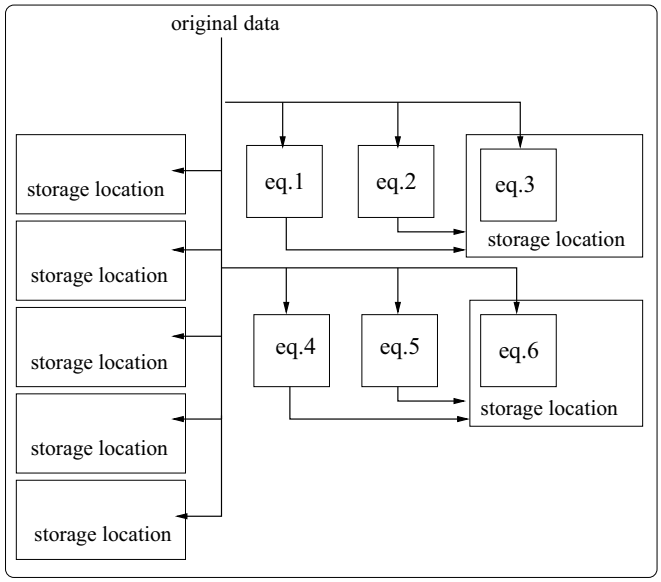

Figure 2. Direct coding on a multi core CPU with different storage locations

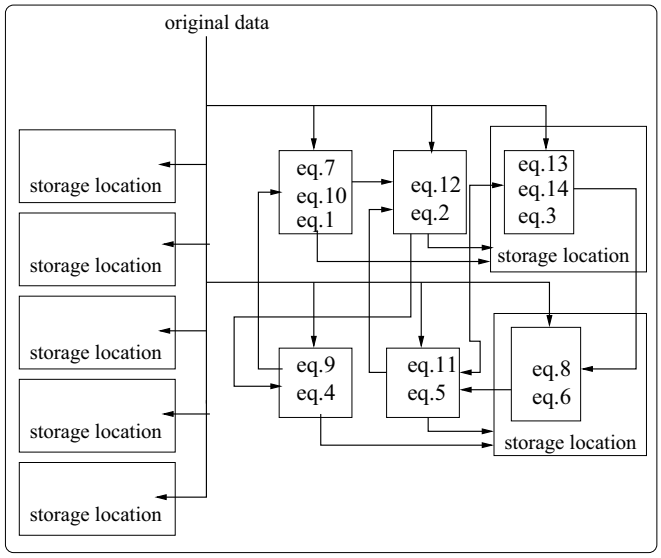

Figure 3. Iterative coding on a multi core CPU with different storage locations

Decoding of data in case of a unit failure works with xor-based equations as well. A dedicated component provides the decoding equation for the particular failure case, similarly to the encoding equations. These decoding equations are either provided on demand, i.e. at the time when the failures are detected, or are selected from a pre-calculated set of equations. Similarly to encoding, a selection of a (de-)coding style and a mapping of equations to functional units is necessary.

3.2 Case study: FPGA-based Hybrid Computing System

A hybrid computing system consists of several differently structured computing resources. For example, these can be field programmable gate arrays (FPGAs) that are coupled with multi-core CPUs. Today, a couple of commercial high performance computing systems are designed as a combination of CPUs and FPGAs, for instance the Cray XD-1 system, or SGI RASC-Brick (Reconfigurable Application Specific Computer). We use the FPGA as platform for a coding coprocessor.

The redundant elements of Reed/Solomon-coded data are calculated by several multiplication units and summators. The encoding can be described on the level of single bits, where multiplication maps to logical 'and' and the sum is an xor operation. Such an architecture follows CRS as a variant of the Reed/Solomon code. The direct encoding style is used. For these calculations, the FPGA is configured to a set of XOR-calculation engines that are connected properly, according to the equation-based description of the code.

Alternatively, coding can also be implemented by a classical Reed/Solomon code [5, 7] by combining several words (groups of ω bits that are placed continuously on a particular storage location) using Galois Field arithmetics. The latter approach has been experimentally evaluated in [9].

For a system with 5 data storage location (elements e_1, e_2, e_3, e_4, e_5) and two locations for redundant data (elements e_6, e_7), the equations are specified as follows:

$$e_6 = g_{1,1} \cdot e_1 + g_{2,1} \cdot e_2 + g_{3,1} \cdot e_3 + g_{4,1} \cdot e_4 + g_{5,1} \cdot e_5$$
$$e_7 = g_{1,2} \cdot e_1 + g_{2,2} \cdot e_2 + g_{3,2} \cdot e_3 + g_{4,2} \cdot e_4 + g_{5,2} \cdot e_5$$

For the elements, the notation e_i is used instead of the element number, to distinguish elements from constants. The constants $g_{i,j}$ form a part of the generator matrix that is loaded into the coprocessor. The coprocessor implements structures and a data path that are directly adopted from the data flow for Reed/Solomon calculations, according to [7]. For the multiplication we use a specialized multiplier architecture [6], instead of lookup tables. This multiplier is able to generate one product in a single clock, when included into a pipelined architecture. The rest of the coprocessor structures are designed as follows:

- The multiplications are done in parallel by a 2-dimensional array of Galois field multipliers. On a Xilinx Virtex4-LX160 FPGA we could place 8×8 multipliers together with the other components on the system. This allows to calculate 8 redundant elements in parallel for $k = 8$ data elements.
- The products are summed up by XOR gates that combine corresponding bits of all products in a row. Every single row produces a redundant element.
- The constant factors for multiplication are taken from a matrix memory that is loaded once at the beginning of the systems operation.

Fig. 4 depicts the encoding circuit that has been implemented on a FPGA. A similar coding architecture can be used for decoding too. Then, the decoding equations are transformed to a matrix and placed in the code matrix memory. In case of failures, with the remaining data and redundancy elements as an

input, the erased data elements are obtained. In the next section, we provide a comparison of computation cost for single-core-based, multi-core-based and FPGA-based coding.

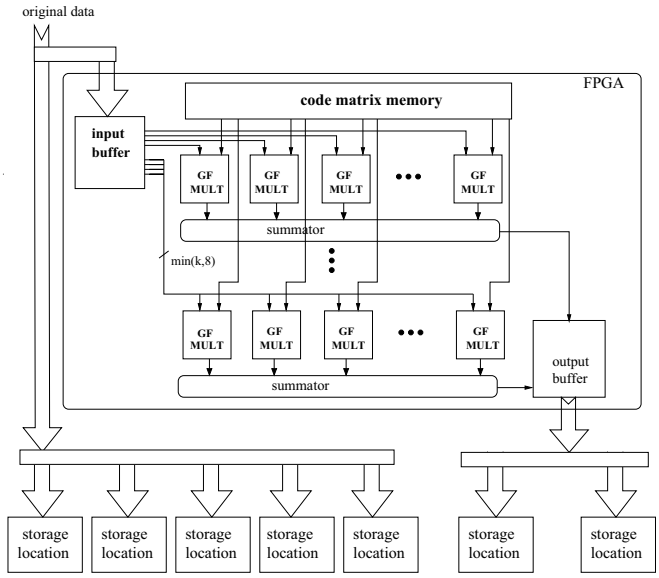

Figure 4. FPGA-based coprocessor for R/S coding

4. Comparison

The computation cost, measured in the number of time steps, is derived for different platforms. It covers CRS on a single core, CRS on a multi core execution platform, classical R/S on a FPGA, and for comparison for the optimum cost for a MDS code on a single core system. The reference unit is the time for a single xor operation. All costs are justified to that base.

Optimum for sequential code calculation:
In the optimal case, k bits (one bit on every data storage location) have to be combined for a single bit on a redundant storage location. This requires $k-1$ xor operations for every redundant bit, and $k-1 \times m$ xor operations for all redundant bits.

$$C_{opt} = (k-1) \times m$$

This is a lower bound that holds for MDS codes, which actually tolerate m faulty storage locations. It also expresses the optimal calculation time on a sequential computation platform, which is reached only by a few codes for special parameters (k,m).

Cauchy–Reed/Solomon (CRS):
Computations are mapped to xor operations, which allows a comparison to the optimum of MDS coding. However, more than a single bit per storage location is included in the xor calculations for a single redundant bit. This can be quantified by a factor between 2 and 3 (see [8]), depending on the parameters k,m and the construction of the code generator matrix.

$$\begin{aligned} C_{CRS} &= (F_{CRS} \times k) - 1) \times m \\ &\text{using } \; 2 \leq F_{CRS} \leq 3 \end{aligned}$$

This factor F_{CRS} corresponds to a comparison made in [1] for $\omega = 8$. A single Galois field multiplication costs 1.3 time units additionally to the cost of xor-ing the products (+1). For a Reed/Solomon code, the factor 2.3 over the optimum is obtained. It is notable that the cost of a classical Reed/Solomon code is practically much higher, due to the need for table lookup of products that are more costly than xor operations.

CRS on a multi core execution platform:
The cost C_{CRS} can be reduced by using multiple cores for the calculation of different equations. In a system with p processor cores, a number of $min(p, m \times \omega)$ cores can be actually taken for code calculations. For example in 3.1, a number of 6 cores has been used, because of $m = 2$ and $\omega = 3$.
In the best case, the cost is reduced by a fraction of $min(p, m \cdot \omega)$, compared to the original cost. This reduction has to be weighted with the efficiency of parallel execution E, which is 0.93 for direct coding and 1.07 for iterative coding. The latter efficiency includes the reduction of xor operations by reusing intermediate results.

$$C_{CRS-multi} = C_{CRS} \cdot \frac{1}{min(p, m \cdot \omega) \cdot E} \qquad (1)$$

The value E is strongly dependent on the code used; we use the values above for the comparison.

FPGA-accelerated Reed/Solomon-Code:
The FPGA-based coprocessor combines parallel multiplication and pipelining of the stages multiplication and summation. As long as the multiplier array is large enough for the required number of storage locations, redundant data can be produced each single clock period. This corresponds to a time of one xor operation for a set of m redundant bits, i.e. C_{FPGA} would be 1.
For a fair comparison we take the lower clock frequency of a FPGA into account and introduced a factor $T = \frac{f_{CPU}}{f_{FPGA}}$. For example, this factor is 18 for a design that runs on 124 MHz and have to compete with a CPU running with 2.2 GHz clock speed.
Another limitation is the available space for multipliers and summators. When using 64 multipliers and 8 summators, the calculation requires $\lceil \frac{k}{8} \rceil \times \lceil \frac{m}{8} \rceil$ time units, solely for the multiplication. When $k > 8$, the partial sums have to be combined sequentially on another xor unit, which requires $(\lceil \frac{k}{8} \rceil - 1) \times m$ xor operations. The cost is derived as follows:

$$C_{FPGA} = \lceil \tfrac{k}{8} \rceil \times \lceil \tfrac{m}{8} \rceil \times T + (\lceil \tfrac{k}{8} \rceil - 1) \times m.$$

The cost function for the different platforms are depicted by plots in Figure 5 by varying k. The different plots are displayed for different numbers of redundant storage locations $m = 2$, $m = 4$ and $m = 6$. A factor $F_{CRS} = 2.5$ is used. The results must be interpreted related to the word size of elementary operations, e.g. 32-bit wide xor operations of a processor core.

The analysis reveals that both the multi core system and the FPGA based design are faster than the optimum coding on a single core system. But this is reached with different resource consumption. The multi core system reaches the fastest speed by using $\omega \cdot m$ cores.

The FPGA-based design requires more time steps for coding, but is still faster than a single core CPU and optimal coding. This speed is reached with a relatively high number of units (e.g. multipliers), which are far less complex than a CPU core. It could be shown that multiple computation units within a SoC - and also special designs for code calculations - can be effectively used to speed up code calculations. This can help to provide a reliable, an fast memory (or storage) for applications executed on a SoC.

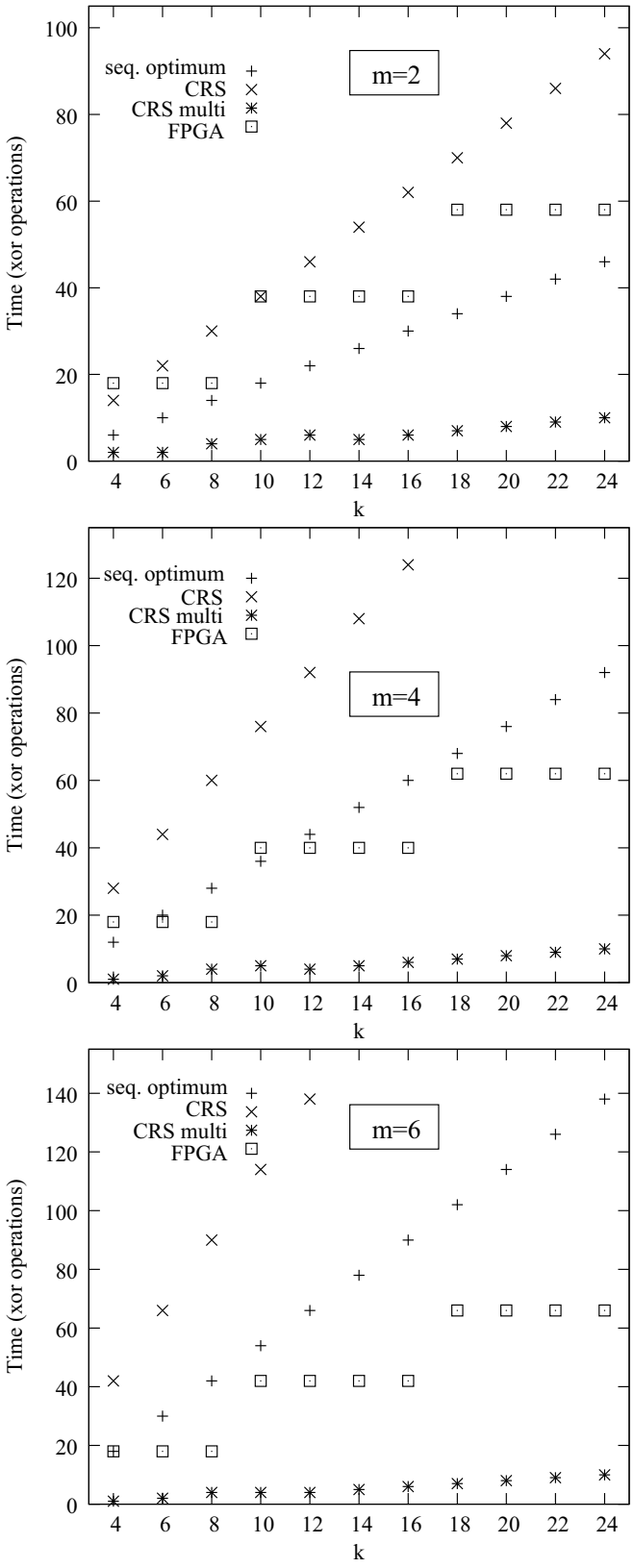

Figure 5. Cost comparison of coding on different platforms.

5. Summary

The concepts of erasure-tolerant codes for reliable data storage have been analyzed for reliable memory and storage within SoCs. The en- and decoding calculations, expressed by equations, have to be mapped to several computation units. On the one hand, this provides fast memory/storage access, on the other hand it utilizes the available units.
It is a complex task to select a proper code and to map them efficiently to the computation units. We demonstrated it using examples and derived the coding cost for a multi core CPU and a specialized design for en- and decoding. The analysis shows that one can speed up coding significantly by using - or wasting - a high number of processor cores for that purpose.

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–resilent Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[3] J. L. Hafner. HoVer Erasure Codes for Disk Arrays. In *Proceeding of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*. IEEE Computer Society, 2006.

[4] R. W. Hamming. *Coding and Information Theory*. Prentice Hall Inc., 1986.

[5] 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.

[6] C. Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. *IEEE Transactions on Computers*, 45(7):856–861, 1996.

[7] 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.

[8] 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.

[9] 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.

[10] 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.