An Erasure-Resilient and Compute-Efficient Coding Scheme for Storage Applications

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Sebastian Kalcher

aus Lahn-Gießen

Frankfurt 2013

(D30)

vom Fachbereich Informatik und Mathematik der

Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Thorsten Theobald
Gutachter:	Prof. Dr. Volker Lindenstruth Prof. Dr. Udo Kebschull
Datum der Disputation:	31.10.2013

Abstract

Driven by rapid technological advancements, the amount of data that is created, captured, communicated, and stored worldwide has grown exponentially over the past decades. Along with this development it has become critical for many disciplines of science and business to being able to gather and analyze large amounts of data. The sheer volume of the data often exceeds the capabilities of classical storage systems, with the result that current large-scale storage systems are highly distributed and are comprised of a high number of individual storage components. As with any other electronic device, the reliability of storage hardware is governed by certain probability distributions, which in turn are influenced by the physical processes utilized to store the information. The traditional way to deal with the inherent unreliability of combined storage systems is to replicate the data several times. Another popular approach to achieve failure tolerance is to calculate the block-wise parity in one or more dimensions. With better understanding of the different failure modes of storage components, it has become evident that sophisticated high-level error detection and correction techniques are indispensable for the ever-growing distributed systems. The utilization of powerful cyclic error-correcting codes, however, comes with a high computational penalty, since the required operations over finite fields do not map very well onto current commodity processors. This thesis introduces a versatile coding scheme with fully adjustable fault-tolerance that is tailored specifically to modern processor architectures. To reduce stress on the memory subsystem the conventional table-based algorithm for multiplication over finite fields has been replaced with a polynomial version. This arithmetically intense algorithm is better suited to the wide SIMD units of the currently available general purpose processors, but also displays significant benefits when used with modern many-core accelerator devices (for instance the popular general purpose graphics processing units). A CPU implementation using SSE and a GPU version using CUDA are presented. The performance of the multiplication depends on the distribution of the polynomial coefficients in the finite field elements. This property has been used to create suitable matrices that generate a linear systematic erasure-correcting code which shows a significantly increased multiplication performance for the relevant matrix elements. Several approaches to obtain the optimized generator matrices are elaborated and their implications are discussed. A Monte-Carlo-based construction method allows it to influence the specific shape of the generator matrices and thus to adapt them to special storage and archiving workloads. Extensive benchmarks on CPU and GPU demonstrate the superior performance and the future application scenarios of this novel erasure-resilient coding scheme.

Zusammenfassung

Die sogenannte *digitale Revolution*, beginnend am Ende des 20. Jahrhunderts, hat unseren Zugang zu Informationen umfassend verändert. Viele Bereiche des gesellschaftlichen Lebens, der Wirtschaft und der Wissenschaft sind mittlerweile stark auf Informationstechnologie angewiesen. Zusammen mit der rasanten Entwicklung der Mikroprozessoren war die Evolution der magnetischen Speichertechnologien einer der Schlüsselfaktoren für den Übergang in eine Informationsgesellschaft. Kostengünstige Datenspeicher mit hoher Kapazität bilden heute die Basis für die große Familie von allgegenwärtigen Geräten, die in die Klasse der *personal computing devices* eingeordnet werden, beispielsweise klassische Arbeitsplatzrechner, tragbare Laptops, Mobiltelefone, Tablet-Computer, sowie Foto- und Videokameras. Besonders der Durchbruch der Halbleiter-Speicher hat die Entwicklung immer kleinerer und robusterer Geräte aus dieser Klasse begünstigt. Aber auch die Entwicklung der professionellen Hochleistungscomputer und deren Speichersysteme wird durch diese Schlüsseltechnologien angetrieben. Besonders diese Art von Computern hat enorme Bedeutung für den wissenschaftlichen Erkenntnisgewinn, aber auch ein breites wirtschaftliches Anwendungsspektrum. Das gewaltige Wachstum der Menge an Informationen, die weltweit erzeugt (oder gewonnen), übertragen und gespeichert wird ist daher wenig überraschend. Schätzungen zu Folge stieg die Gesamtmenge aller gespeicherten Informationen (auf allen gängigen analogen und digitalen Medien) von 2,6 Exabytes[1] in 1986 auf 295 Exabytes in 2007 [1]. Bemerkenswerterweise trugen die digitalen Speichertechnologien erst ab dem Jahr 2000 signifikant zur gesamten Speicherkapazität bei. Nur drei Jahre später wurden geschätzte 90% aller neuen Daten auf magnetischen Medien gespeichert [2] und im Jahr 2007 überholten die digitalen Speicher die analogen endgültig: 52% aller existenten Informationen waren auf Festplatten, 28% auf optischen Medien und 11% auf magnetischen Bändern gespeichert. Neuere Studien aus dem Jahr 2011 quantifizierten die Gesamtmenge aller vom Menschen jemals erzeugten und gespeicherten Informationen auf gewaltige 1,8 Zettabytes [3][4]. Die datenintensiven Wissenschaften gelten neben dem Experiment, der Theorie und der Simulation mittlerweile als vierte essentielle wissenschaftliche Disziplin. Das prominenteste Beispiel kommt aus Gebiet der Teilchenphysik: Der Large Hadron Collider (LHC) der Europäische Organisation für Kernforschung (CERN) in Genf [5] beheimatet sechs unterschiedlich große Experimente die zusammen einige zehn Petabyte an zu speichernden Daten pro Jahr aufnehmen. Aber nicht nur derartige Großforschungsprojekte stellen hohe Anforderungen an die Datenverarbeitungs- und Speichersysteme: Die Entwicklung bildgebenden Sensoren mit Auflösungen Gigapixel-Bereich und immer höher auflösenden Teleskopen stellt die astronomische Forschung vor ähnliche Herausforderungen. Sogar die relativ kompakten DNA-Sequenzierungsgeräte aktueller Bauart produzieren bereits Rohdatenmengen im Bereich einiger Terabyte pro Tag. Mit den immer größer

[1]Ein Exabyte steht für 10^{18} Bytes.

werdenden Datenmengen müssen natürlich auch die Hochleistungsrechner mit immer größeren Speichersystemen ausgestattet werden, um die Analyse der gewonnen Informationen zu ermöglichen. Doch die Verarbeitung von großen Datenmengen hat nicht nur im wissenschaftlichen Umfeld enorm an Bedeutung gewonnen. Besonders das Geschäftsfeld der Internet Suchmaschinen wäre ohne die Fähigkeit enorme Datenmengen zu sammeln, zu analysieren und zu speichern heute undenkbar. Weiter Beispiele sind elektronische Handelsplattformen und weltumspannende soziale Netzwerke.

Der größte Teil all dieser Daten wird zum Online-Zugriff auf großen Sammlungen von Festplatten vorgehalten, deren Verwaltung und Pflege die Besitzer vor große Herausforderungen stellt: Obwohl die Zuverlässigkeit einer einzelnen Festplatte durchaus beachtlich ist, ergeben sich bei Systemen aus Zehntausenden Festplatten hohe Raten an Fehlerereignissen. Große wissenschaftliche Experimente können aus Kostengründen oft nicht beliebig wiederholt werden und schon eine kurze Unerreichbarkeit im Internethandel kann enorme Verluste erzeugen. Typischerweise werden daher Kopien der Daten erstellt und diese auf (geographisch) verschiedene Rechenzentren verteilt. Da diese Strategien der mehrfachen Replikation relativ kostenintensiv sind, wurden effizientere Verfahren entwickelt. In ihrem berühmten Artikel aus dem Jahre 1988 beschreiben Patterson, Gibson und Katz [6], wie man die Zuverlässigkeit von Sammlungen von Festplatten mit verschiedenen Techniken erhöhen kann. Die Berechnung redundanter horizontaler Paritätsinformationen zur Erzeugung einer einfachen Fehlertoleranz hat sich daraus zum Standard für die folgenden Jahrzehnte entwickelt (RAID). Später kamen diverse Erweiterungen hinzu, die sogar den Ausfall von zwei Festplatten tolerieren können. All diese Verfahren stellen jedoch nur die Spezialfälle der umfassenden mathematischen Theorie der linearen fehlerkorrigierenden Codes dar. Im Rahmen dieser Theorie nutzt man die Werkzeuge der linearen Algebra um das Kodieren und das Dekodieren zu formalisieren. Als besonders leistungsstark haben sich die zyklischen linearen Codes über endliche Körper erwiesen. Die weitverbreiteten Reed-Solomon-Codes [7] erlauben es, zu einer Menge an n Nachrichtensymbolen eine frei wählbare Menge an k Redundanzsymbolen zu berechnen. Die original Nachricht kann auch dann noch rekonstruiert werden, wenn bis zu k Symbole verloren gegangen sind. Eine Variante dieser Codes eignet sich besonders für den Einsatz in Speichersystemen [8]. Für die n Nachrichtensymbole $d = d_0, d_1, \dots, d_{n-1}$, müssen k Redundanzsymbole $c = c_1, c_2, \dots, c_{k-1}$ berechnet werden. Mit Hilfe einer speziellen Generatormatrix $G = (g_{i,j})$, die von einer Vandermonde-Matrix abgeleitet wird, lässt sich die Kodierung durch eine Matrix-Vektor-Multiplikation darstellen:

$$G \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ c_0 \\ c_1 \\ \vdots \\ c_{k-1} \end{pmatrix} = \begin{pmatrix} d \\ c \end{pmatrix}.$$

Nachrichten- und Redundanzsymbole werden nun auf $n+k$ unterschiedliche Speicherkomponenten (zum Beispiel Festplatten) verteilt. Im Falle des Defekts einer Komponente (und dem einhergehenden Verlusten des Nachrichtensymbols d_i) wird nun die i-te Zeile der Generatormatrix G und des Ergebnisvektors $(d,c)^T$ gestrichen. Nach k Streichungen erhält man G^* und $(d^*,c)^T$. Das verbleibende Gleichungssystem

$$G^* \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix} = \begin{pmatrix} d^* \\ c \end{pmatrix} \tag{1}$$

kann nun gelöst werden. Die Dekodierung entspricht also einer Multiplikation der invertierten reduzierten Matrix mit dem Vektor der noch intakten Symbole:

$$(G^*)^{-1} \begin{pmatrix} d^* \\ c \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{pmatrix} = d \tag{2}$$

Diese elegante Kodierungsschema hat jedoch einen entscheidenden Nachteil. Bei allen arithmetischen Operationen handelt sich um Operationen auf endlichen Körpern (auch Galois-Körper genannt). Die Symbole sind Elemente des Körpers und repräsentieren Polynome. Alle arithmetischen Operationen sind daher Operationen auf Polynomen. Besonders die Multiplikation zweier Elemente ist deshalb sehr aufwendig und lässt sich schlecht auf die Ausführungseinheiten moderner Prozessoren abbilden. Üblicherweise werden daher die Ergebnisse (oder bestimmte Zwischenergebnisse) der arithmetischen Operationen im Voraus berechnet und in Lookup-Tabellen im RAM gespeichert. Diese spezielle Implementierung hat sich im Laufe der Zeit jedoch als immer problematischer erwiesen. Besonders der Hauptspeicher hat sich zu einem Flaschenhals in modernen Computern entwickelt, begründet durch die ungleiche Entwicklung der Prozessorleistung und der Zugriffs-Latenzen des Speichers. Der häufige Zugriff auf große Datenstrukturen im Hauptspeicher kann daher die Leistung eines Programms stark beeinträchtigen. Für effiziente Algorithmen kann es daher durchaus vorteilhaft sein, bestimmte Daten on-the-fly neu zu berechnen, anstatt diese aus dem RAM anzufordern. Immerhin können moderne Prozessoren in der Zeit, die für den Speicherzugriff benötigt wird (falls das entsprechende Wort nicht im Prozessor-Cache vorhanden ist), einige hundert Instruktionen ausführen [9].

Kern dieser Arbeit ist ein Kodierungsverfahren, das speziell auf moderne Mehrkern-Prozessoren mit breiten Vektoreinheiten und auf aktuelle Beschleuniger-Architekturen mit Hunderten von Kernen angepasst ist. Die Basis ist ein fehlerkorrigierender Code über einem endlichen Körper (ähnlich den Reed-Solomon-Codes). Die Lookup-Tabellen für die Multiplikation sind durch einen polynomiellen Algorithmus (Abbildung 1) ersetzt. Sowohl eine auf den *Streaming SIMD Extensions (SSE)* basierte Vektor-Implementierung für Haupt-Prozessoren, als auch eine Implementierung, die moderne Grafikprozessoren als Co-Prozessoren verwendet, werden vorgestellt. Hierbei werden lediglich komponentenweise UND und Exklusiv-ODER, sowie Schiebeoperationen und Vergleiche benötigt. Die Elemente

```
procedure GFMULT_POLY($a, b$)
    $p \leftarrow 0$
    while $a \neq 0$ AND b $\neq$ 0 do
        if $LSB(b) = 1$ then
            $p \leftarrow p \oplus a$
        end if
        $msb\_set \leftarrow MSB(a)$
        $a \leftarrow LeftShiftByOne(a)$
        if $msb\_set = 1$ then
            $a \leftarrow a \oplus PP$
        end if
        $b \leftarrow RightShiftByOne(b)$
    end while
    return $p$
end procedure
```

Abbildung 1.: Polynomielle Multiplikation über einem endlichen Körper. LSB(x) gibt das *least significant bit* von x zurück, MSB(x) liefert das *most significant bit* von x. PP bezeichnet das primitive Polynom mit dem der endliche Körper erzeugt wurde.

der Flusskontrolle lassen sich dabei gut auf die Gegebenheit der jeweiligen Architektur abbilden. Eine genauere Betrachtung des polynomiellen Algorithmus zeigt, dass die Multiplikation bei einer vorteilhaften Verteilung der polynomiellen Koeffizienten (in zumindest einem der Faktoren) stark beschleunigt werden kann. Da im Bezug auf die zu kodierenden Daten keine Einschränkungen gemacht werden sollten, bieten sich nur die Elemente der Generatormatrix an, um die beschleunigte Multiplikation nutzbar zu machen. Eine eigehende Untersuchung aller primitiven Polynome des gewählten Körpers zeigt, dass sich mit der ursprünglichen algebraischen Konstruktion der Generatormatrix allein kein zufriedenstellendes Ergebnis erzielen lässt. Daher werden verschiedene Heuristiken vorgestellt, mit denen sich Generatormatrizen von besonderer Güte erzeugen lassen. Besonders vielseitig zeigt sich hierbei ein Monte-Carlo Verfahren. Damit lassen sich nicht nur Matrizen erzeugen, bei denen der Rechenaufwand pro Element besonders gut balanciert ist, sondern auch solche, bei denen sich die Verteilung der polynomiellen Koeffizienten zwischen verschiedenen Zeilen oder Spalten stark unterscheidet. Es können also Generatormatrizen mit bestimmten Strukturen erzeugt werden. Ist zum Beispiel eine besonders schnelle Berechnung des ersten Redundanzsymbols gewünscht, so kann der Rechenaufwand für dieses auf Kosten des Rechenaufwands für die weiteren Symbole reduziert werden (tatsächlich kann die Berechnung sogar auf die einfache Parität zurückgeführt werden). Diese begünstigt eine verzögerten Berechnung der höhergradigen Fehlertoleranzsymbole, entweder in Zeiten geringerer Last oder erst nachdem entschieden wurde, dass eine höherer Grad an Ausfallsicherheit für die jeweiligen Daten gewünscht ist. Damit besitzt das Verfahren eine bisher unbekannte Flexibilität und erlaubt eine anwendungsspezifische Optimierung. Abbildung 2 zeigt verschiedene Beispiele. Da es sich um einen systematisch Code handelt, findet sich in den ersten n Zeilen die Identitätsmatrix.

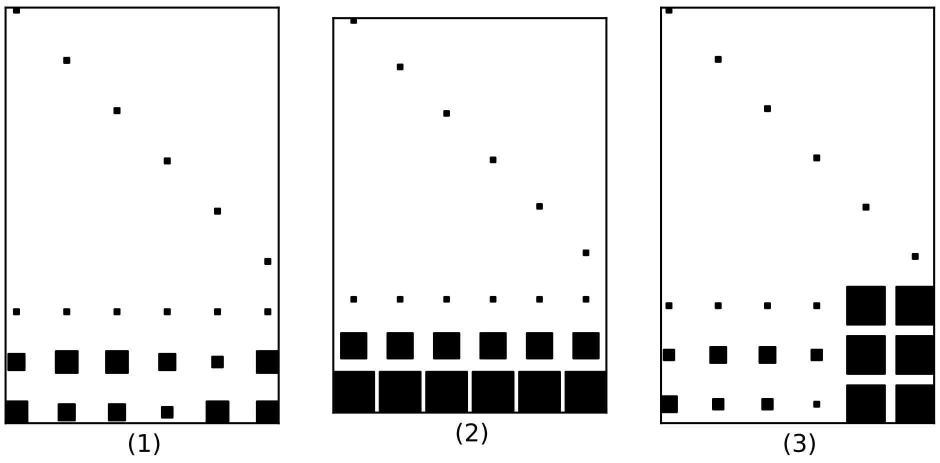


Abbildung 2.: Illustration der anwendungsspezifischen Generatormatrizen. Die Größe der gefüllten Quadrate zeigt den Rechenaufwand für das jeweilige Matrixelement an.

Umfangreiche Benchmark-Tests belegen die Leistungsfähigkeit dieses Kodierungsverfahrens. Mit Hilfe der vektorisierten Implementierung der Multiplikation ist es möglich die Kodierung und Dekodierung auf handelsüblichen Prozessoren mit ausreichender Leistung durchzuführen. Die erforderlichen Vektor-Instruktionen sind über eine weite Bandbreite von Prozessoren unterschiedlicher Hersteller und Preisklassen verfügbar. Zudem belegen die Leistungsmessungen der GPU-Implementierung, dass sich moderne Grafikkarten als leistungsstarke Co-Prozessoren für Kodierungsaufgaben anbieten. Für einige Konfigurationen ist deren Kodierungsleistung sogar nur durch die limitierte Bandbreite des PCI Express-Busses beschränkt. Damit lassen sich Durchsätze erreichen, die vergleichbar mit denen von modernen Hochgeschwindigkeitsnetzwerken wie zum Beispiel InfiniBand oder 10GbE/40GbE sind. Die Kombination aus einem arithmetisch intensiven Multiplikationsalgorithmus mit darauf optimierten Generatormatrizen zeigt sich demnach als besonders geeignet für die verschiedenen aktuellen Prozessorarchitekturen. Im Hinblick auf die zukünftige Entwicklungen in Richtung breiterer Vektoreinheiten und einer höheren Anzahl an einfachen Ausführungseinheiten zeigen erste Tests, dass das Verfahren davon stark profitieren kann. Untersuchungen zur tatsächlichen Integration des Kodierungsverfahrens in großskalige Massenspeichersysteme (zum Beispiel Lustre oder FraunhoferFS) haben bereits begonnen.

Contents

1. Introduction

The so-called *digital revolution* in the past few decades has profoundly changed the access to information. Rapid developments in digital computing and communication have marked the beginning of the information age and today many parts of modern economy and society rely deeply on information technology. The rise of electronic commerce, the global development of the communication and finance sectors, but also the influences on education, health, and even aspects of social life are consequences of this remarkable transition. Together with rapid advancements in microprocessor technologies, the innovation in magnetic disk storage has been among the key factors for this development. Inexpensive, high capacity data storage devices have been the technological basis for both, the ubiquitous personal computing devices, but also for large-scale professional computing and storage systems. More recently, mass production of solid state storage devices has further enabled portable computing and communication equipment, as well as high-capacity digital photo and video cameras, and various portable entertainment devices. Unsurprisingly, the amount of information that is created or captured, communicated, and stored worldwide has tremendously grown over the past decades. An overview of the different prefixes that are used subsequently to quantify information is shown in Table 1.1. It is

Table 1.1.: Decimal and binary prefixes for bits and bytes.

Decimal prefix	Value	Binary prefix	Value
kilo	10^3	kibi	2^{10}
mega	10^6	mebi	2^{20}
giga	10^9	gibi	2^{30}
tera	10^{12}	tebi	2^{40}
peta	10^{15}	pebi	2^{50}
exa	10^{18}	exbi	2^{60}
zetta	10^{21}	zebi	2^{70}
yotta	10^{24}	yobi	2^{80}

estimated that the total amount of (optimally compressed) data which is stored worldwide by the most widely used analog and digital storage technologies has increased from 2.6 exabytes in 1986 to 295 exabytes in 2007 [1]. Remarkably, the digital storage technologies have only contributed notably to the overall capacity from the year 2000 on. Only three years later it was estimated that more than 90% of all new information was stored on magnetic media [2] and by 2007 digital storage had become the dominant storage technology with 52% of all information stored on hard disks, 28% on optical media, and 11% on digital tape. More recent studies estimate that in 2011 the total amount of information created and replicated surpassed the astonishing mark of 1.8 zettabytes [3][4]. Extrapolating with these growth rates, the amount of information will soon become comparable to the approximately 10^{23} bits that are stored in the DNA of an adult human [1]. Clearly, both numbers are dwarfed by the estimated 10^{90} bits of storage capacity of the entire observable universe [10].

The enormous growth of information is not only based on personal data. Data-intensive science is today seen as the fourth major scientific discipline, aiming to connect experimental science, theoretical science, and simulation [11]. In this context data-intensive science is an umbrella term covering many sub-disciplines including data capturing and curation, as well as data analysis and visualization. Prominent examples are found in various fields of science: The Large Hadron Collider at the European Organization for Nuclear Research CERN hosts six detector experiments of different sizes: ALICE, ATLAS, CMS, TOTEM, LHCb, and LHCf [5]. The main task of the ALICE experiment, for example, is the study of lead ion collisions. The peak read-out rate into the front-end electronics of this experiment is about 6 terabytes/s, which is reduced through various selection steps (the so-called triggers [12]) to around 1 gigabyte/s. Similar amounts are produced by the other three large experiments, with the result that the combined amount of recorded data at CERN in 2010 was 13 petabytes [13]. While the data generation rates in high-energy physics are certainly remarkable, they are not exceptional. The development of multi-gigapixel imaging sensors and the increasing size of telescopes also requires the field of astronomy to manage large data collections in the order of tens of petabytes per year [14]. Modern high-throughput DNA sequencing systems enable genome analysis at unprecedented time scales, delivering terabytes of raw data per day from relatively compact machines. This has allowed for cataloging and processing the genome of several thousands of individuals, followed by widespread distribution for further scientific analysis [15]. Due to the ever-increasing resolution and complexity of sensors and detectors, many future projects are expected to fit the category of data intensive problems. The same is true for the increasing HPC computing capabilities: The transition to exascale science is anticipated for 2018, with estimated storage requirements between 500 and 1000 petabytes per supercomputer installation [16]. An added challenge after the immediate capturing and analysis of data is the long-term storage and preservation of the data. Many guidelines for good scientific practice demand the secure and durable storage of primary data sets for at at least ten years after publication.

The ability to gather and analyze large amounts of data is not only crucial for science, today various business models rely heavily on the ability to handle the so-called *big data*[1]. The most prominent example is internet search. The amount of data that is processed to maintain and update an index of the web is so large, that it cannot be handled by classical database and storage systems. Google claims that their indexing system stores several tens of petabytes per day, while handling billions of updates [17]. In 2011 it was estimated[2] that Google was using around 900,000 individual servers in their global network of data centers [18]. Many of these search-related problems have been the driving factor for

[1]Big data is a loosely defined term, covering, generally speaking, all data sets, that are so huge that they require massively parallel software and hardware to be handled.

[2]Unfortunately, many companies consider the size and the architecture of their IT infrastructure a trade secret. The numbers are usually estimated from secondary information (such as power consumption, number of customers, or number of public IP addresses), nonetheless they are useful to illustrate the size of the data sets that are handled by companies today.

novel techniques and paradigms for distributed processing of big data which have been quickly applied to other problems in in business and science. Another example is the electronic commerce company Amazon. Initially focussed on online retail, the company has become one of the biggest providers for computing services. They operate a global cloud computing platform using around 450,000 servers (estimation for 2012 [19]). An essential part of this platform is a fully redundant storage service which was estimated to store around 566 petabytes of data in 2011 [20]. Several secondary cloud storage vendors for consumers as well as for enterprises rely on the Amazon platform as underlying storage system. A last example for big data in the business context is the social network Facebook. They reported 955 million of monthly active users in June 2012, requiring more than 100 petabytes storage, mainly for the pictures[3] and videos of the users [22].

A large fraction of the data of all these applications is stored on huge collections of magnetic disks for on-line access. Data management at this scale has become a tremendous challenge. For many of the large scientific experiments a loss of primary data is a catastrophic event, since the repetition of the experiment is often simply to expensive. Even unavailability of data for short periods of time can have significant economic consequences (for example due to lost sales or advertisement). Typically these dangers are addressed by creating multiple copies, possibly distributed across different geographical locations. While simple replication is relatively inexpensive in terms of implementation and computational overhead, it clearly is very expensive regarding the additionally required storage capacity. With increasing size of storage systems, a sophisticated way to deal with failures (beyond replication) becomes crucial. In their famous paper from 1988 on redundant arrays of inexpensive disks (RAID), Patterson, Gibson, and Katz described several techniques to increase the reliability of collections of disks beyond replication [6]. Using the horizontal parity over all involved disk has become the standard for achieving single disk failure tolerance in the following decades. Several two failure resilient schemes have been proposed as extension to the horizontal parity. As it turns out, all these specialized approaches can be seen as the edge cases of the very refined theory of linear error-correcting codes. In this theory, the generation of codes, as well as the encoding of messages and the decoding of code words are formalized in the language of linear algebra. Hence many familiar techniques and tools can be applied (this is discussed in depth in Chapter 4). Within the framework of coding theory, sophisticated error and erasure correcting codes with the capability to tolerate an arbitrary, adjustable number of failures can be constructed. These codes can be designed such that they require only the theoretical minimum of additional storage space. The price for this flexibility is a higher computational overhead. Several codes (among them the ubiquitous Reed-Solomon codes [7]) operate on finite fields and their symbols are seen as coefficients of polynomials over those finite fields. Since polynomial arithmetic does not map very well onto modern processor architectures, the results of certain arithmetic operations have traditionally been pre-calculated and stored in lookup tables. The strategy to store results of arithmetic operations in memory to avoid recalculation has been successful for the last decades. However, the memory subsystem has become one of the main bottlenecks in commodity systems today. Frequently accessing large in-memory data structures from inner loop code can severely impact the overall performance. Due to the ever increasing disparity between the instruction throughput of processors

[3]In December 2011 Facebook published that, on average, 250 million photos are uploaded per day [21].

and the access times of memories, it is worthwhile to examine whether it is faster to calculate a certain arithmetic result just-in-time, instead of fetching the pre-calculated result from memory. After all, modern processors are able to issue in the order of hundred instructions during the time it requires to load an uncached word from memory [9]. Decoupling the arithmetic code from cached data structures can also be beneficial for scalability in terms of utilization of vector units and utilization of multiple processor cores.

In this thesis a novel coding scheme is presented which is designed for modern multi-core commodity processors with wide vector units, as well as for modern many-core devices (such as GPUs) that are used in a co-processor fashion. The table-based multiplication algorithm (for finite fields) has been replaced by a (vectorized) polynomial multiplication with modular reduction. Depending on the distribution of the polynomial coefficients in the factors, the multiplication can be vastly accelerated. Since a favorable distribution of the polynomial coefficients cannot be assumed for arbitrary data[4], the focus of this thesis has been the distribution of polynomial coefficients in the elements of the generator matrices of the erasure correcting code. Sine the classical algebraic construction methods of the generator matrices do not produce matrices with elements that are optimal in terms of the distribution of polynomial coefficients, several heuristics have been developed to find better generator matrices. The most versatile method follows a Monte-Carlo approach and, therefore, allows to influence the specific shape of the generator matrix. In this way it is possible to create special purpose generator matrices which have distinct advantages for special use cases (for example different computational costs for different levels of fault tolerance). The combination of polynomial multiplication and suitable generator matrices proves to be very beneficial for both execution on modern CPUs as well as on many-core accelerator devices. A helpful addition to the coding scheme is an algebraic signature that also operates on finite fields. These signatures can be used as checksums for larger data blocks and they retain their validity through the encoding step. In conclusion, the coding scheme presented in this thesis addresses the challenge of efficient execution of the coding operations on modern off-the-shelf components. Hence it provides a means to economically increase reliability, availability, and integrity of data in modern large-scale storage systems.

1.1. Organization of the thesis

The thesis is organized as follows. In Chapter 2 a short review of secondary storage technologies is presented. Chapter 3 gives an overview about approaches to achieve fault-tolerance and reliability in current storage systems. An introduction into the theory of error- and erasure-correcting codes is given in Chapter 4. This is the foundation for a detailed discussion of a novel compute-efficient coding scheme presented in Chapter 5. Chapter 6 shows the performance characteristics of the coding scheme for the two different usage scenarios, the vectorized implementation for execution on a modern SIMD-enabled CPU, and a many-core version which is run on a modern GPU-based co-processor. The final chapter gives a conclusion and provides an outlook for further research.

[4]In fact, most high-entropy data storage formats display a uniform distribution of the polynomial coefficients.

2. Storage Technologies

This chapter gives a short overview of the important (secondary) storage technologies and their characteristics. The first sections reviews the hierarchy of storage technologies in relation to access time, capacity and relative cost. The following sections describe the essentials of magnetic disk and solid state storage. A micro-benchmark for hard disks is presented subsequently, illustrating the characteristics of a system composed of rotating disks. In the final section the reliability models of disks and secondary storage systems composed of disks are discussed.

2.1. Storage hierarchy

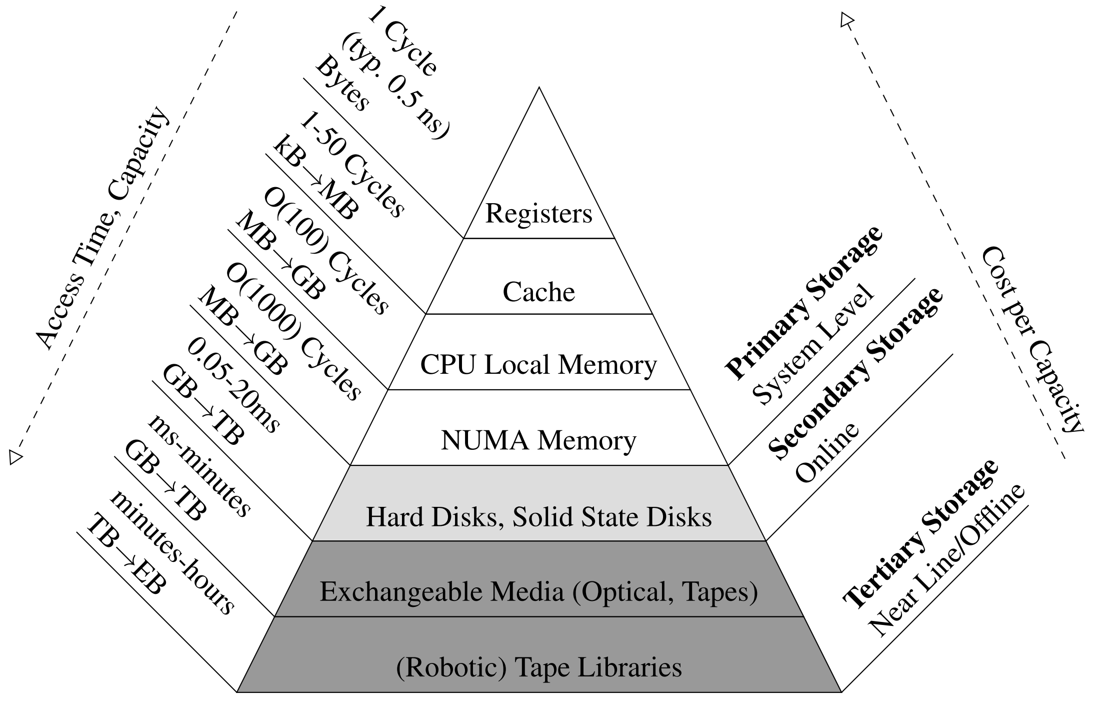


Figure 2.1.: Storage hierarchy [23].

An overview of the hierarchy of storage technologies is shown in Figure 2.1. Access times and available capacities are increasing while going from the top towards the base, whereas the opposite is true for cost

Table 2.1.: Capacities and prices for top-of-line storage devices commercially available in 2012

Device	Capacity	Typical price
CPU registers[1]	~4 KiB	2000 EUR/CPU
CPU cache	24MiB	2000 EUR/CPU
CPU local RAM	256 GiB	3.200 EUR
Multi CPU NUMA RAM	1 TiB	12.800 EUR
Hard disks	4 TB	300 EUR
Solid state disks	512 GB	1000 EUR
Optical media (Blue Ray)	50 GB	10 EUR
Tapes	5 TB	300 EUR
Full tape library [24]	1 EB	N/A

per capacity. Capacities and typical prices for several top-of-line products are presented for reference in Table 2.1. The *primary storage* devices are used at system level and include mainly volatile memory system that are rather close to the CPU. Registers and caches are typically a part of the processor and reside therefore on the same die. On modern multiprocessor systems a fraction of the total random access memory is usually directly attached to each CPU. These configurations are called *non-uniform memory architecture (NUMA)*. From the view of the single processor the local memory has lower access latencies and can be accessed directly, while access to the remote memories requires transfers over some intermediate system I/O network. The following layer of *secondary storage* includes persistent online storage devices such as *hard disk drives (HDDs)* and *solid state drives (SSDs)*. Due to their performance characteristics SSDs are often used in a caching layer on top of large scale HDD-based storage systems. All devices that use exchangeable media, such as optical disks and magnetic tapes and larger archival libraries that are built thereof, form the layer of *tertiary storage* devices. Tertiary storage can be characterized as offline storage, as random access is not always possible and explicit media change is often necessary. While reliability is evidently a critical issue for all storage components, the following sections are dedicated to the middle layer of secondary storage devices.

Today there are mainly two technologies for secondary storage: Rotating magnetic disks and solid state storage. While non-volatile solid state storage devices have been available from the early 1970s [26], the utilization of solid state devices for (mass market) persistent secondary storage is a relatively new development. Around the year 2004 several vendors started offering devices with SATA, SAS, and ATA interfaces on a larger scale, however, capacities were several orders of magnitudes lower than for hard drives and prices were several orders of magnitudes higher. The origins of magnetic disk storage on the contrary are in the 1950s [27]. Growing demand for online storage with the possibility for random access drove the transition from punchcards and magnetic tapes to rotating magnetic disks.

[1]Counting the 16 general purpose 64-bit registers and the 16 256-bit vector registers. The actual number of registers can be larger, since not all registers are exposed.

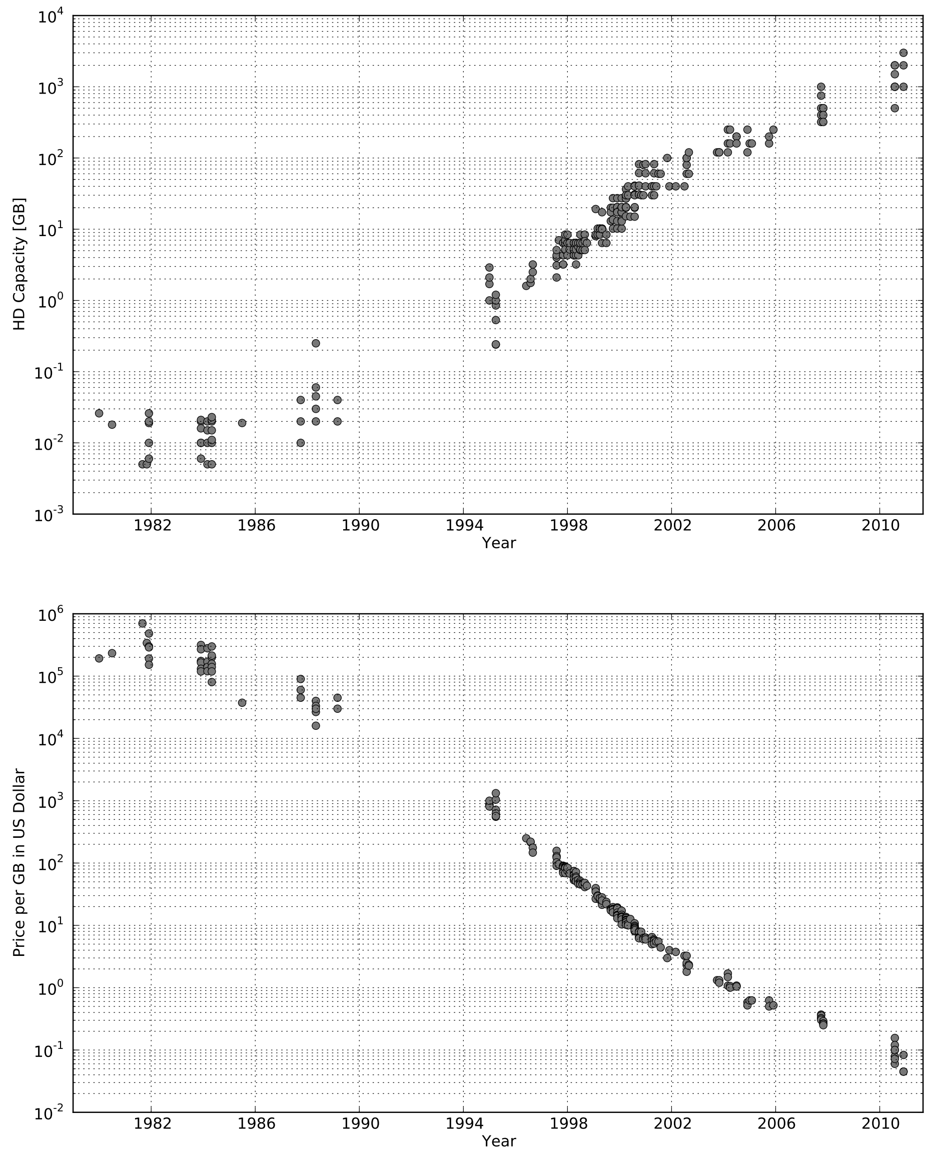


Figure 2.2.: Development of hard disk capacities and relative costs [25].

The first patent describing a magnetic data storage machine based on a set of rotating disks was granted in 1964 [28]. Several decades of active research and continuous development and innovation have lead to a remarkable role of magnetic disk storage today. In 2003 it was estimated that 92 % of all new information is stored on magnetic media, primarily hard disks [2], and the total amount of hard disk storage worldwide at the end of 2008 was estimated at roughly 200 exabytes [29]. The development of hard disk capacities and relative prices is shown in Figure 2.2. This plot of historical data illustrates one of the reasons for the success of magnetic disks: the aerial storage density doubles approximately every 12 to 18 months, in analogy to the well known law for the doubling of the transistor count by Gordon Moore, this is known as Kryder's law [30, 31]. The relative price for magnetic disk storage on the other hand has been decreasing with the same pace.

2.2. Magnetic disk storage

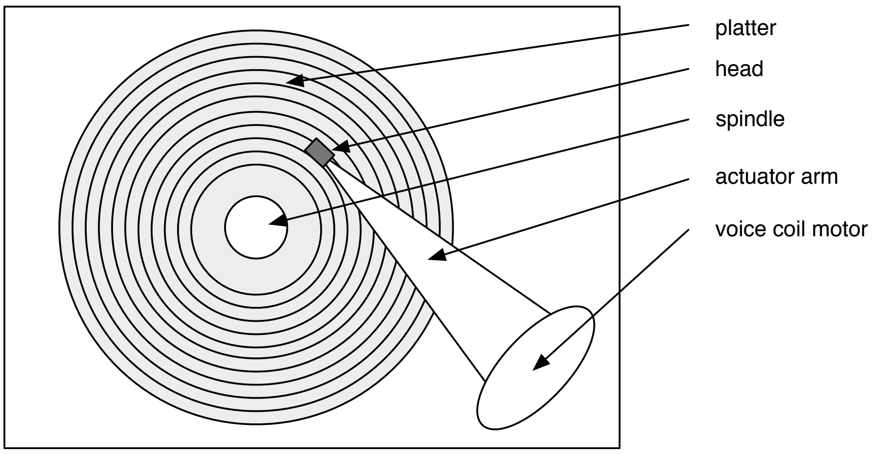


Figure 2.3.: Top view of a hard disk

A generic view of a hard disk drive is depicted in Figures 2.3 and 2.4. In general it consists of a collection of platters which rotate on a spindle at a fixed number of revolutions per minute [32]. The platters themselves are built of three layers: The rigid substrate (based on metal, glass or ceramic), a layer of magnetic recording material, and a protective overcoat layer [33, 34]. The substrate must be extremely uniform, free of material defects, and unsusceptible to thermal expansion. The magnetic recording material consists of several thin film layers [35] (for example containing Cobalt, Chromium, and Platinum) with different properties. The thin overcoat layer protects against microscopic particles, dust, and vapors. Multiple platters are arranged on top of each other with enough space between them to allow both sides of the platter to be accessed by an actuator arm. In principle it is possible to stack as many platters as needed, however, current disks contain only up to five platters to limit the number of mechanical components and control the vibration and power consumption. The read and write heads are positioned at the tip of the actuator arm and the distance between the head and the platter is only several nanometers in modern disks. The heads are responsible for storing and reading back data on the magnetic layer. Individual bits are stored in sub-micrometer-sized regions containing a small number

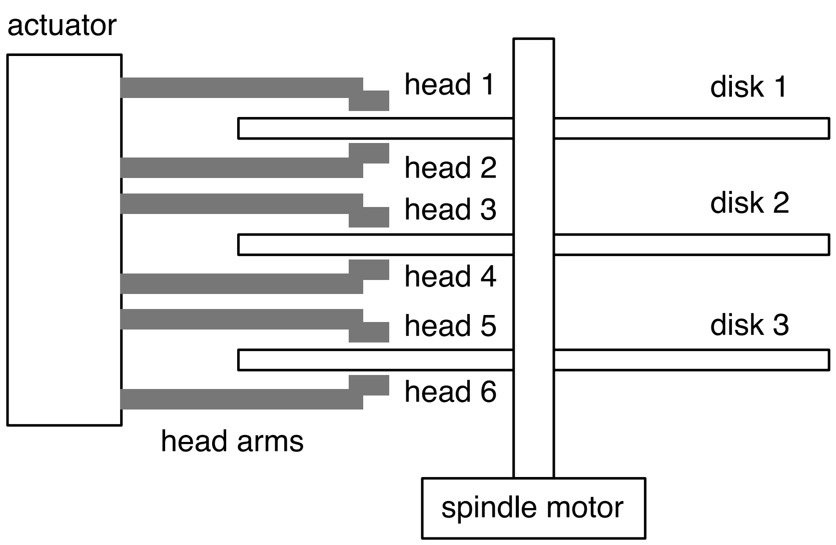


Figure 2.4.: Hard disk side view.

of magnetic grains. Originally, longitudinal magnetization (parallel to the plane of the disk) was used. However, starting in 2005 manufacturers switched to perpendicular magnetization in order to achieve higher storage densities. In 2012 the highest magnetic storage density achieved was one terabit per square inch [36]. Consecutive bits are stored in the circumferential direction of the platter, forming the so-called *track*. Tracks are subdivided into *sectors* which are the smallest accessible unit for data storage. Sectors contain header information, the actual data and a field containing information for error-correction. Traditionally sectors contained space for 512 bytes of user accessible data, however, starting in 2009 a transition to a capacity of 4096 bytes per sector has begun. This allowed for better storage efficiency and it was possible to introduce stronger error-correcting capabilities. The concepts of tracks and sectors are illustrated in Figure 2.5.

All tracks with equal radius on different disks form a *cylinder*, as shown in Figure 2.6. The write head consists of a micro-fabricated electromagnet which induces a magnetization in the region of an individual bit. The read head, on the other hand, uses the *giant magnetoresistance (GMR)* or *tunnel magnetoresistance (TMR)* effects to detect the stray field at the bit region borders. Both are quantum mechanical effects which manifest as a significant change in the electrical resistance depending on the parallel or anti-parallel orientation of the magnetization of neighboring ferromagnetic layers [37]. To achieve precise positioning of the read and write heads, the movements of the actuator arm are controlled by voice coil motors. The platters on the spindle are directly connected to the spindle motor, which has to maintain a constant speed with minimal vibration. This is achieved by using a servo-controlled closed loop which reads servo information embedded on the magnetic surface between the actual data elements. The final component is the disk controller hardware and its associated firmware. The controller is responsible for actually performing the conversion from the digital information to the magnetic signal on the media using sophisticated signal amplification and processing. Furthermore, it

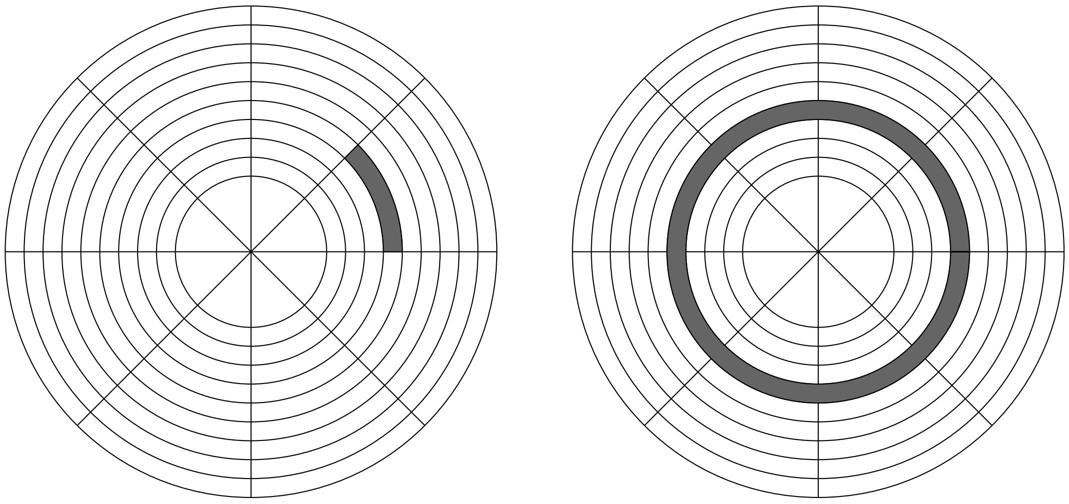


Figure 2.5.: Sector (left) and track (right).

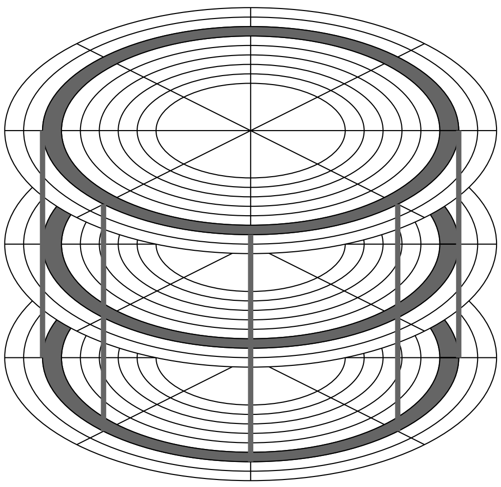


Figure 2.6.: Cylinder.

manages the communication with the host CPU, performs command queueing and scheduling, provides error-correction and data integrity checks, manages buffering, and also controls the servo loop.

The market for hard disks is divided into three major categories: Desktop, nearline and enterprise disk drives [38]. These categories reflect different usage models with very different requirements in terms of robustness, reliability, performance, features, and cost. Desktop (and laptop) drives are designed for non-contiguous operation and a low duty cycle with a strong focus on power consumption, capacity, and low cost. Nearline devices provide high storage capacities with increased reliability. They are designed for continuous availability with a moderately higher workload as desktop drives. Nearline hard drives are used in scenarios where large amounts of data need to be available at all times without interruption (bulk storage), but without the highest demands on performance and throughput, such as

Table 2.2.: Comparison of desktop, nearline, and enterprise disks [38, 40, 41].

	Desktop	Nearline	Enterprise
Capacity (mid 2012)	3 TB	2TB	900 GB
Cost (mid 2012)	∼150EUR	∼300EUR	∼700EUR
Power Consumption (rel)	1x	1.2x	1.5x
Reliability (MTTF)	600kh	1200kh	1600kh
Spindle Speed (RPM)	5.4k-7.2k	7.2k-10k	10-15k
Duty cycle	<10%	<20%	100%
Power-on hours	2400h/a	8760h/a	8760h/a
Performance (rel)	1x	1x	1.4x - 2.5x
T10 data integrity	no	no	yes
Unrecoverable error rate	10^{-14} bits^{-1}	10^{-15} bits^{-1}	10^{-16} bits^{-1}

backup and reference data sets, cloud and scale-out storage. Enterprise disks aim at applications which require highest operational availability and workloads. Their platters are usually smaller in diameter to reduce rotational vibration, head arms and voice coils are designed for 100% duty cycles, and controller electronics and software are generally more sophisticated (containing for instance dual CPUs and better error resilience). They provide the highest performance together with the highest reliability and the lowest uncorrectable error rate. In addition to the error-correcting codes in the disk itself, enterprise disks provide end-to-end data integrity mechanisms from the user perspective: Supplementary 8 byte of information are added to every sector to store user or operating system generated checksums and reference information [39]. With these boundary conditions, enterprise drives are generally more expensive while providing less storage capacity. An overview of different properties of desktop, nearline, and enterprise disks is shown in Table 2.2.

The *super-paramagnetic effect* sets a lower limit for the volume of the magnetic grains that can be used in magnetic storage devices [42]. At this limit the magnetization can be spontaneously flipped due to thermal fluctuation which leads to information loss. To compensate one can increase the magnetic anisotropy of the medium, which at the same time requires an increased magnetic field for which larger heads are needed. Because of this trade-off, for a long time the predicted maximum areal density was several hundreds of gigabits per square inch. However, the transition from longitudinal to perpendicular magnetic recording already allowed for much higher densities of up to 1 terabit per square inch. The road maps for upcoming magnetic recording technologies include several approaches that promise several order of magnitudes higher areal storage densities beyond the super-paramagnetic limit: *Bit-patterned magnetic recording (BPMR)* uses magnetic nano-islands with well defined positions to store information, which are assembled with nano-lithographic processes. *Heat-assisted magnetic recording (HAMR)* employs an additional laser in the read/write head to locally raise the temperature above the Curie temperature and to then change the magnetization with a relatively small magnetic field. Related is the *microwave assisted magnetic recording (MAMR)*, where the write head contains a nano microwave oscillator instead of the more complex optical system. With *shingled magnetic recording (SMR)* neighboring tracks overlap heavily (more than two thirds of the track are covered by the adjacent track). This technique has no disadvantages in case of reading, but for writing it requires an extended read-modify-write cycle, since direct write access to an inner overlapped track would damage the neighbor tracks. SMR can be combined with *two dimensional magnetic recording (TDMR)* where a 2D-image is built up from multiple adjacent tracks and sophisticated image detection and decoding techniques are employed to reconstruct the original data. As a result the problem of destructive interference of shingled tracks can be resolved. All these proposed technologies indicate that magnetic disc storage will remain a core component of secondary storage systems in the next decades, even though alternative non-volatile memory technologies are reaching maturity [31].

2.3. Solid state storage

The success of solid state storage has been driven by the demand for low-power, light, and robust non-volatile storage for mobile applications. Starting with memory cards for digital cameras, USB thumb drives, digital audio players, and mobile phones, flash-based storage is today increasingly used for high performance secondary storage in the form of solid state drives (SSDs). The basic building block of a flash cell is a floating gate MOS transistor which can be arranged in a NOR-type or NAND-type configuration to obtain the two fundamental flash architectures [43, 44, 45]. NOR flash delivers a high read performance, but suffers from low write performance. It can be randomly accessed (similar to DRAM) and application code can be executed from it. Available capacities are rather low and it is therefore often used to store the boot code or firmware for computers or embedded systems. NAND flash, on the other hand, enables much higher cell densities and allow fast erase and write operations, but it cannot be directly addressed and requires complex controller hardware. It offers larger capacities and lower cost and is, therefore, better suited for bulk data storage applications. Since almost all solid state block storage devices are based on NAND-type flash, the architecture and properties of these memories are reviewed in the following paragraphs.

Figure 2.7 on the facing page shows a NAND flash floating gate transistor and the architecture of a NAND flash memory. A single level cell (SLC) transistor stores one bit of data, whereas a multi level cell (MLC) is able to store multiple bits. The value of the bits is determined by the amount of charge stored in the floating gate. Since the floating gate is surrounded by insulating material, the charge is transported by a quantum mechanical tunnel mechanism: It is injected into the floating gate during writing and persistently stored there (in this case the bit value is 0). During erase operations the charge is released from he floating gate, reverting the bit value to 1. To form a memory device, the floating gate transistors are assembled into an array: Several of them are connected in series via a bit line, whereas their control gates are connected to individual word lines. All transistors sharing the same word line form a *page*, and all pages make up a *block* (Figure 2.7 on the next page, right). In order to read, all but one word line are pulled up above the threshold voltage of the transistors,

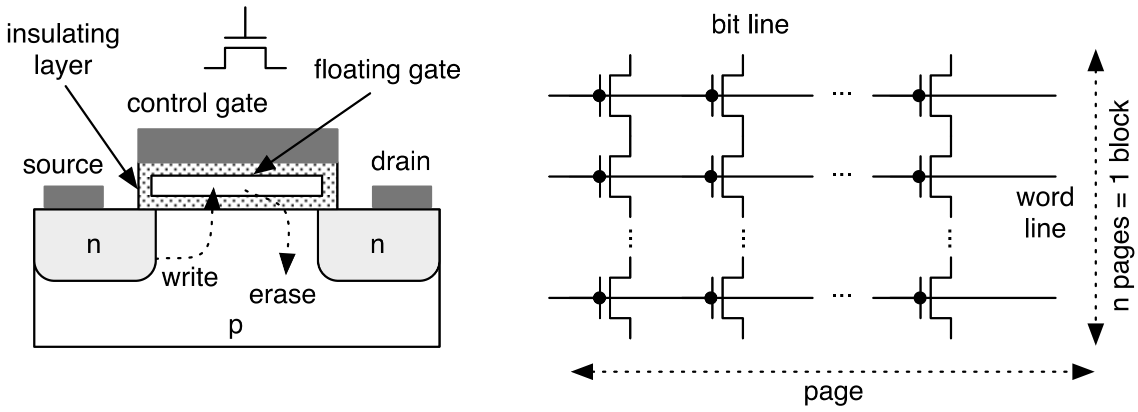


Figure 2.7.: NAND Flash transistor cell (left) and NAND flash memory architecture (right) [43].

the bit line is then conductive or non-conductive depending on state of the singled out transistor. For writing, a high positive voltage is applied to the control gate, which allows tunneling of electrons in to the floating gate. Therefore, a page is the smallest amount of data that must be read or written at a time. Since erasure requires the eviction of electrons from the floating gate by applying a high voltage to the substrate, it can only be done for the entire block (bulk erasing). In effect, an individual page in a NAND flash block cannot be directly overwritten, but the write request has to be redirected to another block by a transparent management layer. If it is clear that all pages in certain block are not used anymore, the whole block can be erased and used for future write requests. The number of write or erase cycles, however, is finite due to two wear-out effects: Electrons can be trapped in the insulating layer around the floating gate or the insulting layer can break down all together. As a result electrons cannot tunnel anymore between the floating gate and the substrate. The page can then not be used anymore and has to be added to a list of bad pages (write endurance for flash devices is in the order of $10^5 - 10^6$ cycles). To avoid the breakdown of pages that are frequently written, write counters and address mapping mechanisms are used to distribute the writes evenly over different blocks (wear-leveling). All these characteristics of NAND flash require an intermediate software or firmware layer which organizes the logical to physical mapping, the garbage collection, the wear leveling, the bad block handling, and the error-correction. Since file systems usually only remove the directory entry of a file upon deletion, the TRIM command has been implemented to let the management layer know that the page is not used anymore. This decreases the reorganization of pages that do not contain valid data anymore and alleviates the performance degradation of background garbage collection. Typical values for a current enterprise level NAND flash storage device are shown in Table 2.3 on the following page. While offering significantly higher random access bandwidth and I/O operations per second than hard disks, capacities and cost differ by at least one order of magnitude. Remarkably, the values for the mean time to failure and the unrecoverable error rate are at par with those of an enterprise disk drive (although the SSD does not contain an equally complex mechanical system). In addition to the problem of limited write endurance due to degradation of the insulation layer, flash storage reliability is affected by several effects [48, 49]: Especially NAND flash is prone to bit flipping in neighboring

Table 2.3.: Characteristics of a top-of-line NAND flash SSD in mid 2012 [46, 47].

	Typical value
Interface	SAS 6 Gb/s
Cell type	MLC
Capacity	400 - 800 GB
Page size	4-8 kB
Block size	$\geq$ 256 pages
Random read IOPS (4K)	180k IOPS
Random write IOPS (4K)	75k IOPS
Random read bandwidth	2000 MB/s
Random read bandwidth	1000 MB/s
Unrecoverable error rate	10^{-16} bits^{-1}
MTTF	2.000.000 h
Write endurance	7-14 PB
Prize	$\sim$ 5 US/GB

cells during read and write operations. Furthermore, the charge in the floating gate can slowly leak and lead to a wrong interpretation of the bit value. The ability to retain data is also influenced by the number of experienced erase or write cycles. Flash devices can contain up to 5 % of initial bad blocks due to production yield constraints, as well as the bad blocks accumulated during operation. Failure to detect and manage both types can lead to data loss. To increase reliability it is essential to use strong error-correcting codes (BCH, Reed-Solomon) on a page level, similar to the sector protection scheme of hard disks.

Solid state storage is often seen complementary to hard disk storage as the top layer in tiered storage scenarios. While hard disks fulfill the requirements for high capacity and low cost, SSD provide higher bandwidths and drastically higher rates of I/O operations per second. However, projections show that as storage densities of flash devices increase, properties such as performance, endurance, energy efficiency and data retention time could be adversely affected [50].

2.4. Disk micro-benchmark

Listing 2.1: Definition of the `sg_io_hdr_t` data structure in the linux kernel

```
typedef struct sg_io_hdr
{
    int interface_id;           /* [i] \'S\' for SCSI generic (required) */
    int dxfer_direction;        /* [i] data transfer direction  */
    unsigned char cmd_len;      /* [i] SCSI command length ( <= 16 bytes) */
    unsigned char mx_sb_len;    /* [i] max length to write to sbp */
    unsigned short iovec_count; /* [i] 0 implies no scatter gather */
    unsigned int dxfer_len;     /* [i] byte count of data transfer */
    void __user *dxferp;        /* [i], [*io] points to data transfer memory
                                              or scatter gather list */
    unsigned char __user *cmdp; /* [i], [*i] points to command to perform */
    void __user *sbp;           /* [i], [*o] points to sense_buffer memory */
    unsigned int timeout;       /* [i] MAX_UINT->no timeout (unit: millisec) */
    unsigned int flags;         /* [i] 0 -> default, see SG_FLAG... */
    int pack_id;                /* [i->o] unused internally (normally) */
    void __user * usr_ptr;      /* [i->o] unused internally */
    unsigned char status;       /* [o] scsi status */
    unsigned char masked_status;/* [o] shifted, masked scsi status */
    unsigned char msg_status;   /* [o] messaging level data (optional) */
    unsigned char sb_len_wr;    /* [o] byte count actually written to sbp */
    unsigned short host_status; /* [o] errors from host adapter */
    unsigned short driver_status;/* [o] errors from software driver */
    int resid;                  /* [o] dxfer_len - actual_transferred */
    unsigned int duration;      /* [o] time taken by cmd (unit: millisec) */
    unsigned int info;          /* [o] auxiliary information */
} sg_io_hdr_t;  /* 64 bytes long (on i386) */
```

To illustrate the characteristics of direct disk operations a small micro-benchmark was developed for this thesis. The benchmark utilizes the SCSI Generic (sg) driver found in the linux kernel [51]. The drivers allows user applications to directly send SCSI commands to the device. Therefore, an object of type `sg_io_hdr_t` (defined in Listing 2.1) is prepared and handed to an `ioctl()`[1]. The `sg_io_hdr_t` data structure itself contains a low level description of a SCSI request using various input and return parameters needed to perform the operation. The `ioctl()` is blocking until the prepared SCSI command is finished. This behavior can be used to probe the internal structure of a disk by measuring the repeated execution time for different SCSI commands. With this low-level access many layers of the operation system can be bypassed and the disturbing effect of caching and queueing systems can be avoided. All benchmarks were performed with an off-the-shelf IBM enterprise-class SCSI disk[2]. For this disk an excellent and comprehensive specification and documentation is available [52], such that the measured properties can be verified. All relevant characteristics are summarized in Table 2.4. To reduce noise during the measurement the disk was installed as secondary (non-OS) disk and used as a pure block device (that is, a file system was not installed on the disk). For all measurements the disk cache was disabled.

2.4.1. Data zones

The disk characteristic to validate the easiest is the *Zone Bit Recording (ZBR)*. Since the track length (or circumference) increases with the distance from the center of the disk, it is possible to fit more

[1] `ioctl()` is short for input/output control and is special system call for non-generic input/output operations specific to a certain device.

[2] Model IC35L018UWD210-0

Table 2.4.: Drive characteristics of the IBM IC35L018UWD210-0 disk [52].

Property	Value
label capacity	18.35 GB
number of heads	3
number of disks	2
number of data bytes	18,351,959,040
number of 512-byte sectors	35,843,670
rotational speed	10,000 RPM
data buffer size	4096 KiB
data zones	17
- sector density in zone 0	750 sectors/track
- sector density in zone 16	390 sectors/track
average seek time (read)	4.9 ms
full stroke seek time (read)	10.5 ms
cylinder skew	0.88 ms
head skew	0.72 ms

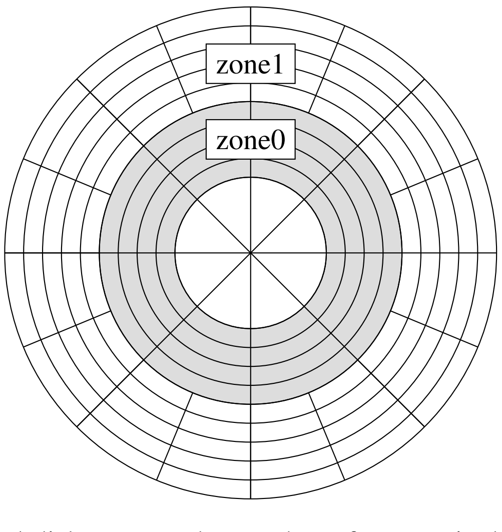


Figure 2.8.: Illustration of hard disk zones. The number of sectors is dependent on the length of the tracks. Hence outer tracks hold more sectors. As a trade-off tracks are grouped into zones.

sectors on the outer tracks. Disks are therefore partitioned into a certain number of zones with constant number of sectors per track as illustrated in Figure 2.8. Since during a full revolution the amount of data read in the outermost zone is larger than in the inner zones, the zoning schema can be directly seen by observing the data transfer rate of a small number of consecutive sectors in the respective zone. Figure 2.9 shows the measured data transfer time depending on the sector number and hence depending on the position of the disk. The arrows indicate the zone borders according to the specification of the used disk.

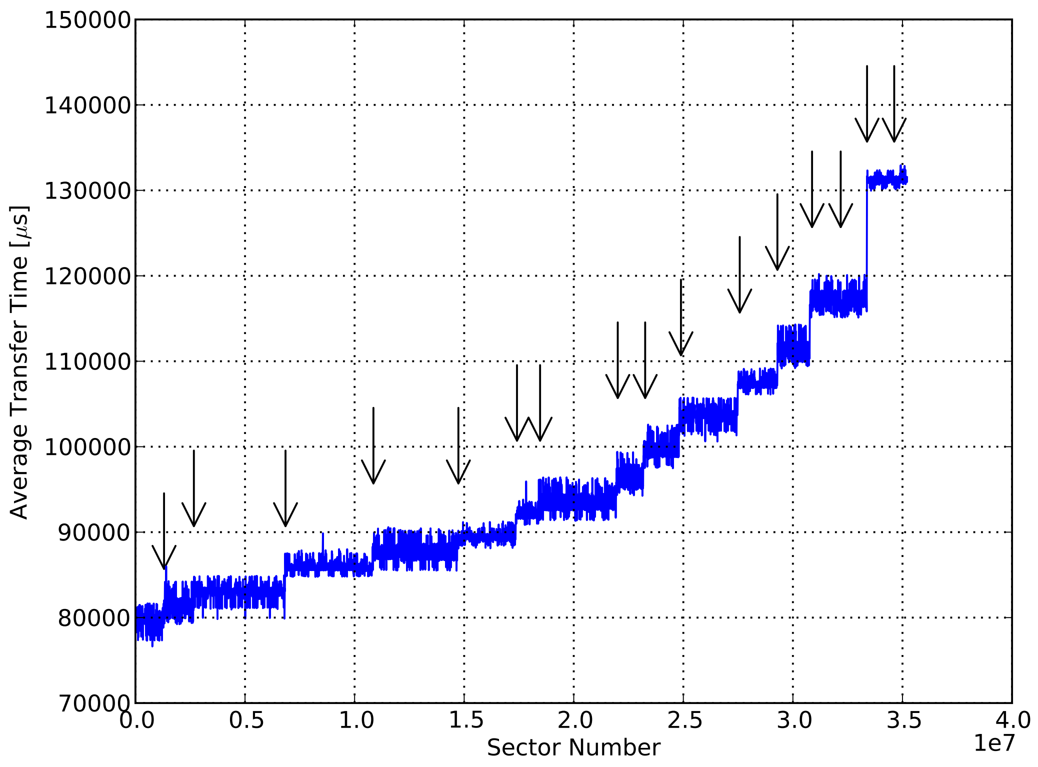


Figure 2.9.: Hard disk zones become visible when measuring the transfer times for a fixed-sized chunk with different staring positions on the disk. The arrows indicate the zone borders according to the specification of the used disk.

2.4.2. Seek time

The seek time is measured from the start of the actuator movement to the start of a reliable read or write operation. The average seek time $\overline{T}$ is specified in the data sheet [52] as weighed average of all possible seek combinations[3]:

$$\overline{T} = \frac{1}{s_{\max}(s_{\max}+1)} \sum_{n=1}^{s_{\max}} (s_{\max}+1-n)(T_n^{in}+T_n^{out}), \tag{2.1}$$

where $s_{\max}$ is the maximum seek length, T_n^{in} is the inward measured seek time to track n, and T_n^{out} is the outward measured seek time to track n. Figure 2.10 shows the seek times for all destination sectors.

[3]The definition of the average seek time varies considerably between hard drive manufacturers. Often the average seek time is calculated as the sum of all possible seek times divided by the number of seeks. It can be shown that this corresponds to one-third of the full stroke seek time.

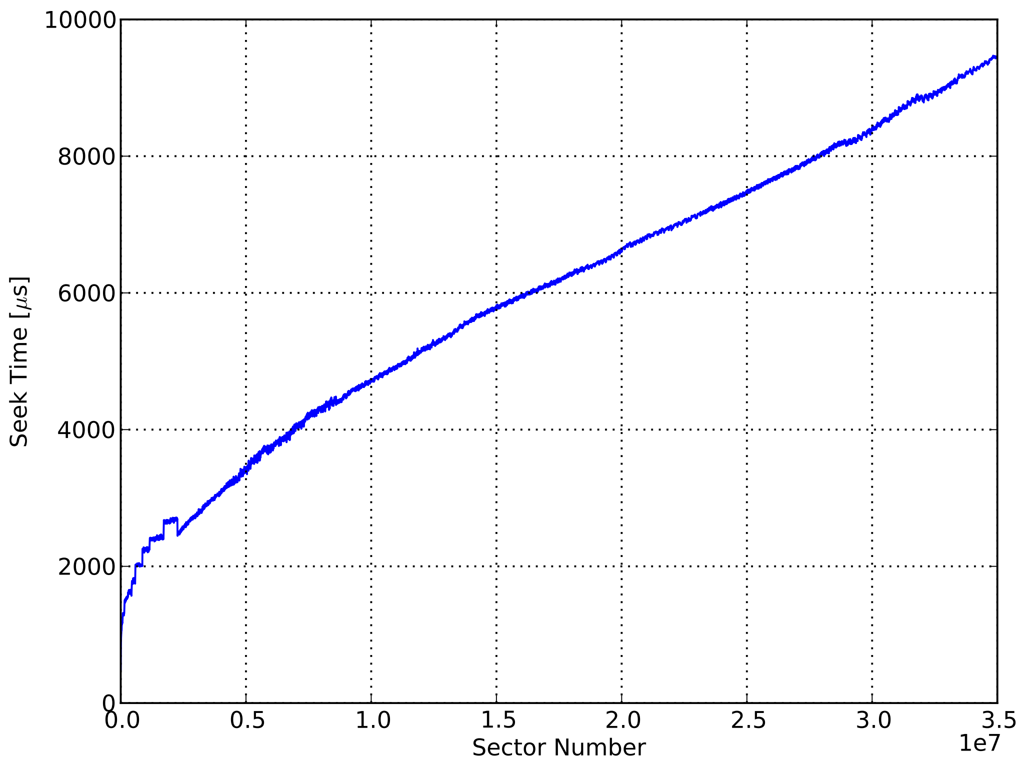


Figure 2.10.: Seek times. The patterns for small sector numbers indicate that the hard disk firmware employs different strategies for shorter movements of the actuator arm.

2.4.3. Jump time

In this particular test the time to read a particular sector directly after reading sector 0 is measured (with two consecutive but separate read commands). The head not only has to seek to the correct track, but also has to wait for the correct sector to be in reach. Figure 2.11 shows the result for the first 7000 sectors. The execution time increases at first with increasing sector number, since the requested sector is too close to sector 0. After the time needed to process the second read request, the requested sector has already passed the head. Therefore almost another full rotational delay is required to access the second sector. When the distance between the two sectors becomes comparable to the distance corresponding to the request processing time, the execution time drops drastically and is effectively reduced to the seek time. This pattern repeats periodically as the second read request moves through the upward tracks. The occasional gaps in the slope correspond to the times required to switch heads inside a cylinder (head skew) and the times to switch to the next cylinder (cylinder skew).

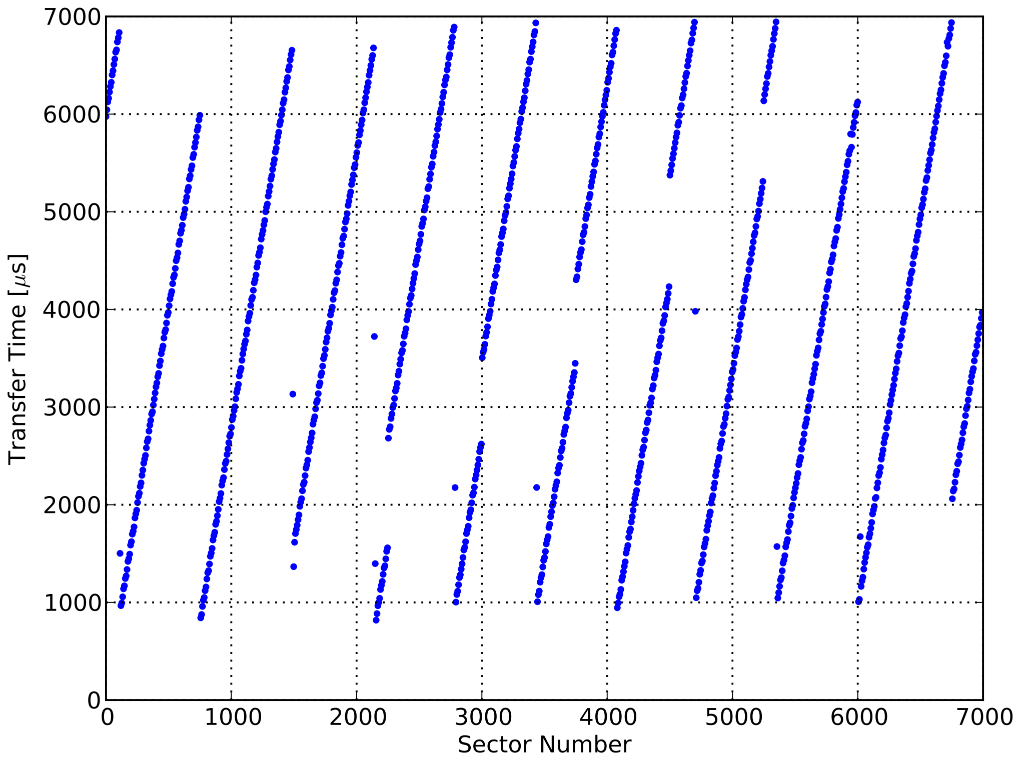


Figure 2.11.: Jump time: Read a particular sector directly after reading sector 0 (with two consecutive but separate read commands).

2.4.4. Window read

The window read benchmark exposes the most information about the structure of the hard disk. The time to repeatedly read a small chunk of consecutive sectors is measured. If the chunk lies in the middle of a track, the measured time reflects the time needed to perform a full rotation (the chunk size is negligible compared to the number of sectors on an outer track). However, if the chunk crosses a disk border, that is, the next sector still lies on the same cylinder but on another platter, the head skew time becomes visible. Furthermore, if the chunk crosses a cylinder border one can observe the cylinder skew. These two penalties show up as distinct peaks in Figure 2.12 and can be used to estimate the size of tracks and cylinders. The mean time needed for one full rotation is measured as $5999 \pm 31 \mu s$, therefore the number of rotations per minute can be estimated as

$$RPM = \frac{60s}{5.999 \cdot 10^{-3}s} = 10.001 \pm 52, \tag{2.2}$$

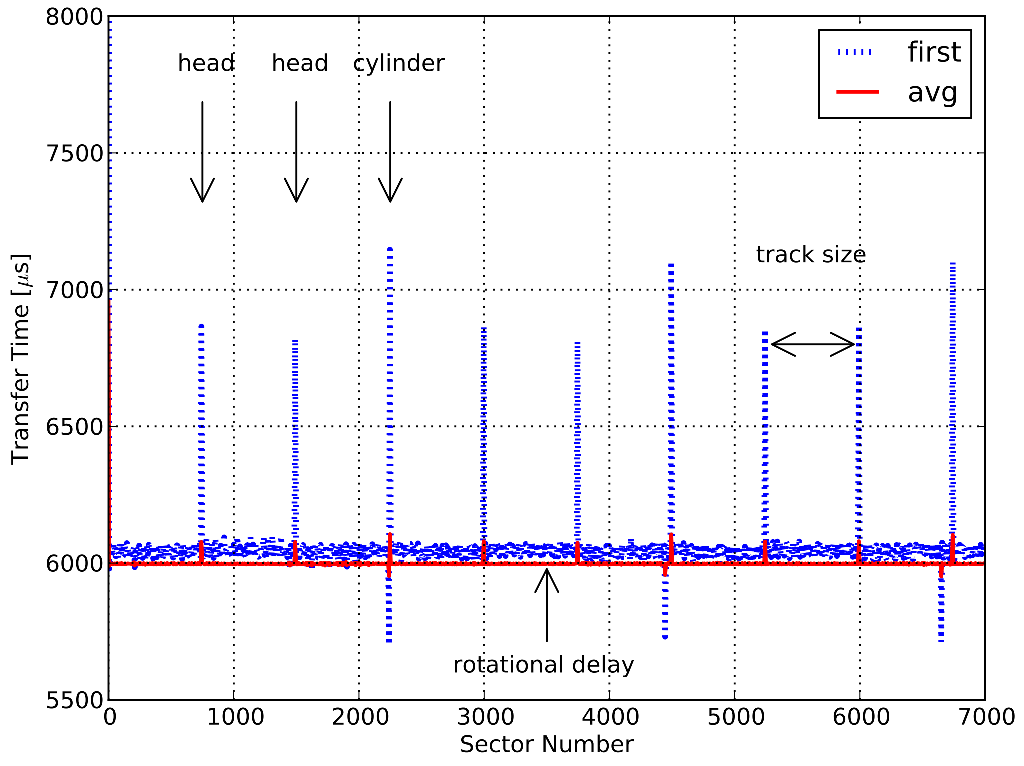


Figure 2.12.: Window read times for chunks of 10 consecutive sectors. The dotted curve indicates the first read operation, whereas the solid curve shows the average. Head and cylinder skew, as well as the rotational delay and the track size in zone 0, can be directly estimated.

which is in remarkable accordance with the data sheet. The peaks in execution time indicate head (platter) switches at sector 750 and 1500, and cylinder switches at sector 2250.

Figure 2.12 shows the average execution time as well as the time needed for the very first access. The times for the first access are significantly higher, especially when a platter or cylinder switch is necessary. This indicates an intrinsic caching mechanism which cannot be switched off with the disk cache.

2.5. Reliability modeling and failure modes

This section provides a short survey of reliability and availability theory, particularly with respect to storage devices. A comprehensive overview can be found in [53] and [54], on which several parts of this introduction are based.

2.5.1. Introduction

The failure rate of hardware components is often associated with the so-called *bathtub curve*, illustrated in Figure 2.13. In this plot the failure rate is shown over the operational time. During the first period of operation poorly manufactured or weak components contribute greatly to the high failure rate until most problematic components have been sorted out and the failure rate stabilizes. This first phase is often called *infant mortality period*. The period with a constant intrinsic failure rate is considered the *normal life period* of the component. At the end of the normal life period the components enter the *wear out period*, associated with a greatly increasing failure rate. Manufacturers often try to reduce the early failure rate visible to the customer by applying burn-in tests, where the component is exposed to increased stress for a short period of time.

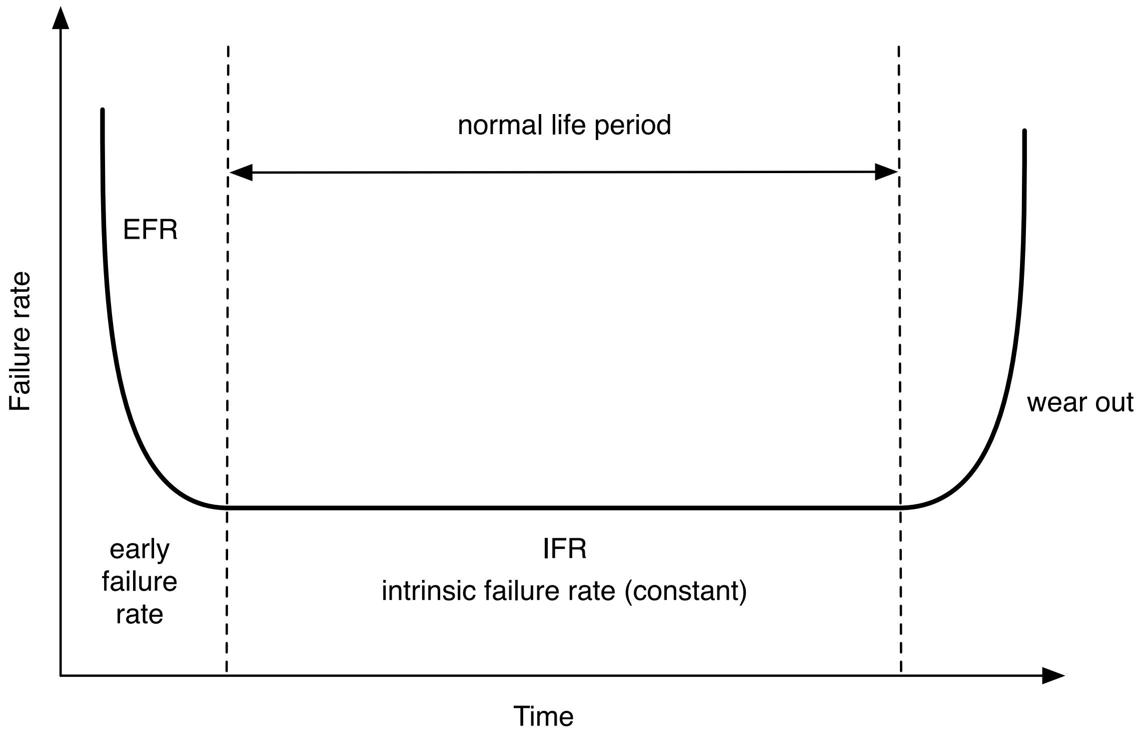


Figure 2.13.: Bathtub curve

In the context of failure rates different mean times are often used: The *Mean Time To Failure (MTTF)* is the arithmetic mean of the times from start of the component to the failure. The *Mean Time To Repair (MTTR)* is the average time until the component repair is completed. Finally, the *Mean Time Between Failure (MTBF)* is the average time between two failures, or simply the sum of MTTF and MTTR. An illustration of MTTF, MTBF, MTTR is shown in Figure 2.14 on the following page. The availability A of a repairable system is defined as the probability that the system is operational. With the definitions

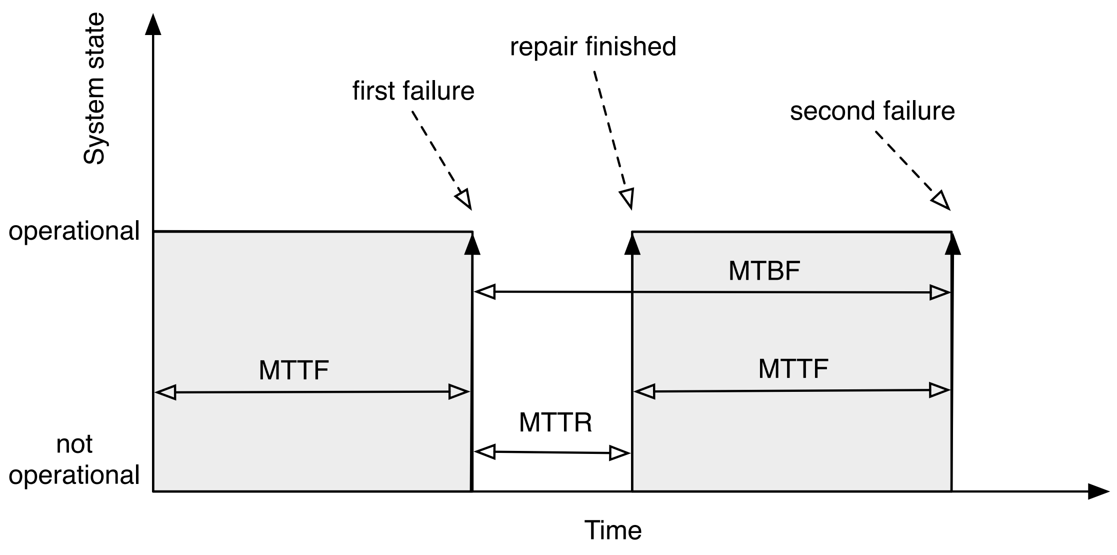


Figure 2.14.: Mean time to failure (MTTF), mean time between failure (MTBF), and mean time to repair (MTTR).

from above the availability can be written as

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}. \tag{2.3}$$

Highly available systems are often grouped into availability classes, depending on the number of nines in the decimal places of the availability: For example $A = 0.99999$, referred to as *five nines*, corresponds to a combined down time of 5.26 minutes per year of operation.

2.5.2. Reliability theory

The reliability $R(t)$ is the probability that a system is operating correctly until the time t. Let T be a random variable denoting the time of a failure. The reliability is then the probability that T is greater than t

$$R(t) = P(T > t), \quad t \geq 0. \tag{2.4}$$

Conversely, the failure probability $F(t)$ is

$$F(t) = P(T \leq t), \quad t \geq 0. \tag{2.5}$$

$F(t)$ is often called the *cumulative failure distribution function (CFDF)* and is related to the reliability function by

$$R(t) = 1 - F(t). \tag{2.6}$$

The derivative of $F(t)$ is called the *failure density function*,

$$f(t) = \frac{\mathrm{d}F(t)}{\mathrm{d}t} = -\frac{\mathrm{d}R(t)}{\mathrm{d}t}. \tag{2.7}$$

Integration of Equation 2.7 leads to

$$F(t) = \int_0^t f(\tau)\mathrm{d}\tau \tag{2.8}$$

and

$$R(t) = 1 - \int_0^t f(\tau)\mathrm{d}\tau = \int_t^\infty f(\tau)\mathrm{d}\tau. \tag{2.9}$$

The relationship can be interpreted as follows: The failure density function gives the probability for a failure at a given point in time. Integration over the complete time must give 1. Therefore, the failure probability by time t is the sum of all failure probabilities from time 0 to t. On the other hand, the survival probability by time t is the remaining part of the integral from time t to infinity.

The expected value of a real valued random variable X with a probability density function $f(x)$ is defined as

$$E(X) = \int_{-\infty}^{\infty} x f(x)\mathrm{d}x. \tag{2.10}$$

Therefore the *Mean time to failure (MTTF)* is defined as[4]

$$MTTF = \int_0^\infty t f(t)\mathrm{d}t, \tag{2.11}$$

and it can be shown (details can be found in [53]) that with the above definitions the MTTF can be expressed as

$$MTTF = \int_0^\infty R(t)\mathrm{d}t. \tag{2.12}$$

The *hazard function* (or instantaneous failure rate function) can then be defined as

$$h(t) = \frac{f(t)}{R(t)}. \tag{2.13}$$

In many cases an exponential failure distribution is chosen where the failure density function is given by

$$f(t) = \begin{cases} \lambda e^{-\lambda t} & t > 0 \\ 0 & t \leq 0, \end{cases} \tag{2.14}$$

where λ is constant. This results in the reliability function

$$R(t) = e^{-\lambda t}, \tag{2.15}$$

[4]Only $t \geq 0$ is relevant for failure analysis.

and the failure density function

$$F(t) = 1 - e^{-\lambda t}. \tag{2.16}$$

The hazard function

$$h(t) = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \tag{2.17}$$

is therefore constant and its inverse is the mean time to failure

$$MTTF = \frac{1}{\lambda}. \tag{2.18}$$

Inherent to the exponential distribution is the property of memorylessness, meaning that the failure rate is independent of the history of failures. However, this is not always an appropriate assumption. The three-parameter *Weibull distribution* provides a generalization of the exponential distribution. The shape of the distribution depends significantly on the specific values of the parameters and it is particularly suitable to model the different phases of the bathtub curve. The failure density function is given by

$$f(t,\beta,\theta) = \begin{cases} \frac{\beta(t-\gamma)^{\beta-1}}{\theta^\beta} e^{-\left(\frac{t-\gamma}{\theta}\right)^\beta} & t > \gamma \\ 0 & t \leq \gamma \end{cases}, \tag{2.19}$$

where β is called the shape parameter, θ is called the scale parameter, and γ is called the location parameter. Since γ is used to specify a guaranteed failure free period from $t = 0$ to $t = \gamma$, in the following $\gamma = 0$ is assumed.

The reliability function is then

$$R(t) = e^{-\left(\frac{t}{\theta}\right)^\beta} \quad \text{for } t > 0,\ \beta > 0,\ \theta > 0, \tag{2.20}$$

and the corresponding hazard function

$$h(t) = \frac{\beta t^{\beta-1}}{\theta^\beta} \quad \text{for } t > 0,\ \beta > 0,\ \theta > 0. \tag{2.21}$$

When $\beta = 1$ the Weibull distribution is equivalent to the exponential distribution, since then

$$f(t) = \frac{1}{\theta} e^{-\left(\frac{t}{\theta}\right)} \tag{2.22}$$

and

$$h(t) = \frac{1}{\theta}. \tag{2.23}$$

For $\beta = 2$ the failure density function is reduced to a function which is also known as Raleigh distribution:

$$f(t) = \frac{2t}{\theta^2} e^{-\left(\frac{t}{\theta}\right)^2}, \tag{2.24}$$

and

$$h(t) = \frac{2t}{\theta^2}. \tag{2.25}$$

In general the shape parameter influences the time dependence of the hazard rate and the interpretation is as follows:

- $\beta < 1$: The hazard rate is decreasing over time. This is used to model the burn-in period.
- $\beta = 1$: The hazard rate is constant and as for the exponential distribution. This behavior models the normal life period.
- $\beta > 1$: The hazard rate is increasing over time, which represents the wear-out period.

The hazard rate of the Weibull distribution for different values of β is depicted in Figure 2.15. The combination of the individual hazard rates can be used to model the characteristic shape of the bathtub curve.

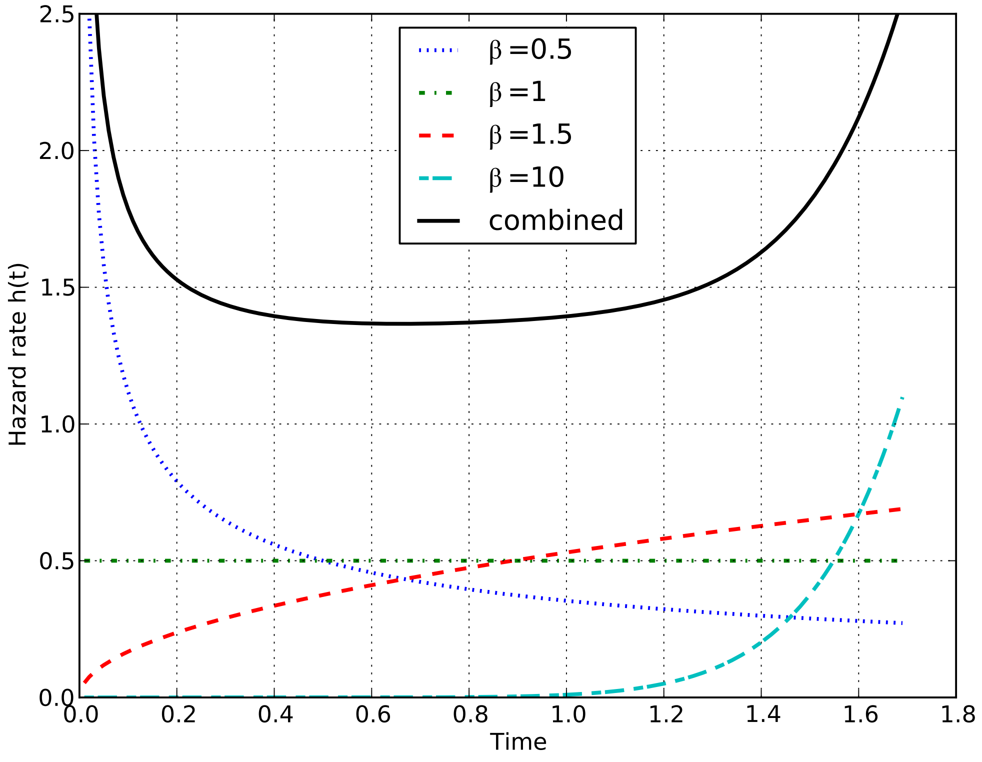


Figure 2.15.: Weibull hazard rates and their additive combination to model the bathtub curve.

2.5.3. RAID reliability

In the original RAID publication [6] the MTTF of a collection of disks, assuming exponentially distributed time to failure and constant failure rate, is given by

$$MTTF = \frac{MTTF_{\text{single disk}}}{N}, \tag{2.26}$$

where N is the number of disks in the collection. With the 1-error tolerance of the original RAID systems, the *Mean Time To Data Loss (MTTDL)* of a RAID array is

$$MTTDL_{\text{RAID}} = \frac{MTTF^2_{\text{single disk}}}{N \cdot (G-1) \cdot MTTR}, \tag{2.27}$$

where N denotes the total number of disks, and G is the number of disks in a (parity) group. $MTTR$ is the mean time to repair or replace the failed disk and to reconstruct the lost data. Later, this has been extended for 2-error tolerant RAID-6 arrays [55]:

$$MTTDL_{\text{RAID6}} = \frac{MTTF^3_{\text{single disk}}}{N \cdot (G-1) \cdot (G-2) \cdot MTTR^2}. \tag{2.28}$$

The generalization to a δ-error tolerant array is straight-forward

$$MTTDL_{\delta} = \frac{MTTF^{\delta+1}_{\text{single disk}}}{N \cdot \prod_{i=1}^{\delta}(G-i) \cdot MTTR^{\delta}}. \tag{2.29}$$

Equation 2.29 can be rewritten to make the benefit of additional error-tolerance in this model more apparent:

$$MTTDL_{\delta} = \frac{MTTF_{\text{single disk}}}{N \cdot \prod_{i=1}^{\delta}(G-i)} \cdot \left(\frac{MTTF_{\text{single disk}}}{MTTR}\right)^{\delta}. \tag{2.30}$$

The $MTTDL$ for the system is multiplied by the ratio of $MTTF_{\text{single disk}}$ and $MTTR$ for every additional error the system can withstand. Clearly $MTTR \ll MTTF_{\text{single disk}}$ holds, since realistically the $MTTR$ is in the order of several ten hours (assuming a 2 TiB disk with a moderate reconstruction rate of 50 MiB/s), whereas the order of the $MTTF_{\text{single disk}}$ of a modern disks ranges from 10^5 to 10^6 hours. Therefore, the repeated division by the size of the parity group is small against the additional power of the ratio.

2.5.4. Markov models

Markov models can be used to model the random behavior of systems if the process is stationary (the behavior must be the same at all points in time independent from the actual point being considered). Again, the occurrence of failures and the failure recovery must therefore be characterized by constant failure and recovery rates. System operations, failures and repair efforts are represented by states

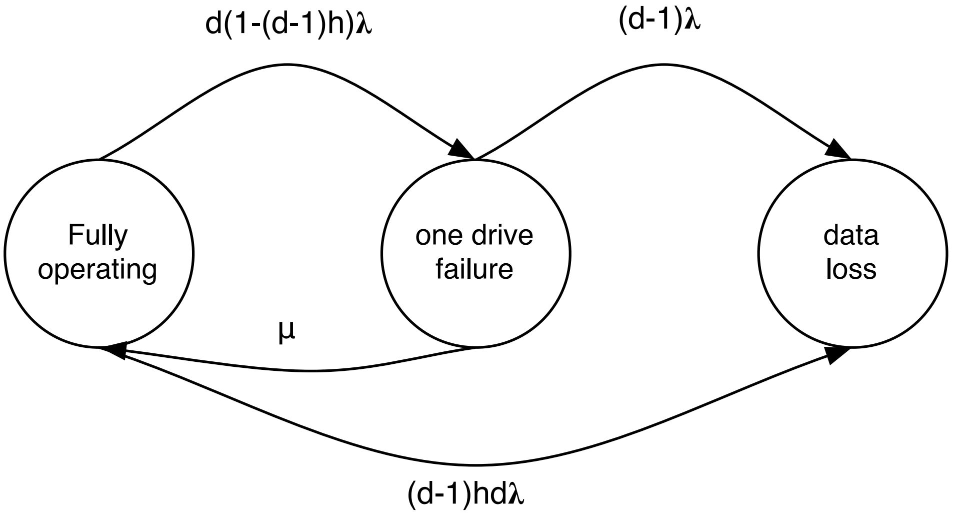


Figure 2.16.: Markov model for a 1-error tolerant array [56].

in a state transition diagram together with events that initiate transitions from one state to another (represented by the corresponding failure and repair rates). To determine how much time the system spends in each of the states, a set of equations is constructed: For the steady state the sum of input and output for each state must vanish. Hence, for every state an equation can be written and together they can be solved for the probabilities P_i for being in state i (all P_i must sum up to one). Markov models can also be used if they contain absorbing states, that is, states that are impossible to leave. Figure 2.16 shows the Markov model for an 1-error tolerant RAID system with unrecoverable read errors as presented in [56]. The parameters used in the state diagram are:

d - Number of disks in the array,

λ - Drive failure rate, $MTTF_{\text{single disk}}^{-1}$,

μ - Drive repair rate, $MTTR_{\text{single disk}}^{-1}$,

h - Probability of an uncorrectable error during rebuild for a single drive.

With the assumption that $\mu \gg \lambda$ the Markov model can be solved for the $MTTDL$ [57]:

$$MTTDL = \frac{(2d-1-dh)\lambda + \mu}{d(d-1)\lambda^2 + d\lambda\mu h} \approx \frac{\mu}{d(d-1)\lambda^2 + d(d-1)\lambda\mu h} \tag{2.31}$$

This model introduces the concept of an unrecoverable read error rate as an additional failure source. In this case the disk does not break completely, but cannot read a typically small amount of sectors. If this happens during recovery from a previous drive failure, data are lost. By neglecting this error probability (by assuming $h = 0$) it is easy to show that Equation 2.31 is equivalent to Equation 2.27. Since both models take an exponentially distributed time to failure as a basis, the similarity is not surprising. Figure 2.17 shows the state diagram for a 2-error tolerant array with the same parameters

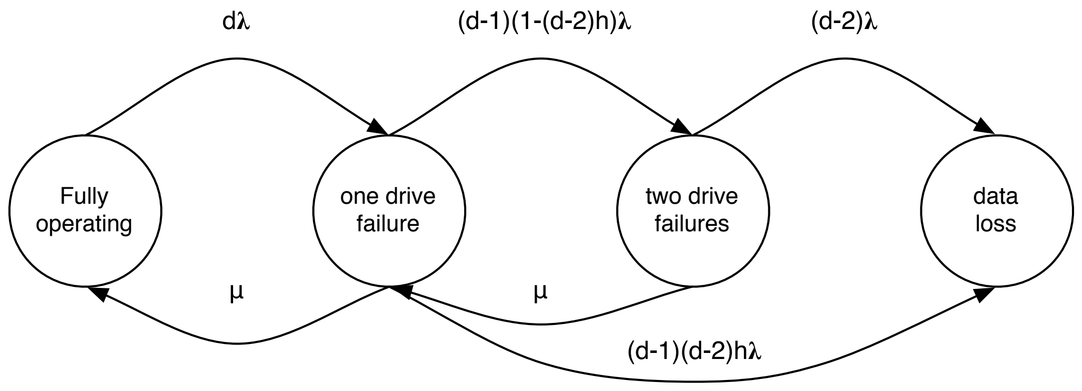


Figure 2.17.: Markov model for a 2-error tolerant array [56].

as above. The solution of the Markov model gives

$$MTTDL \approx \frac{\mu^2}{d(d-1)(d-2)\lambda^3 + d(d-1)(d-2)\lambda^2\mu h} \tag{2.32}$$

The generalization to a δ-error tolerant array (Figure 2.18) was shown in [57].

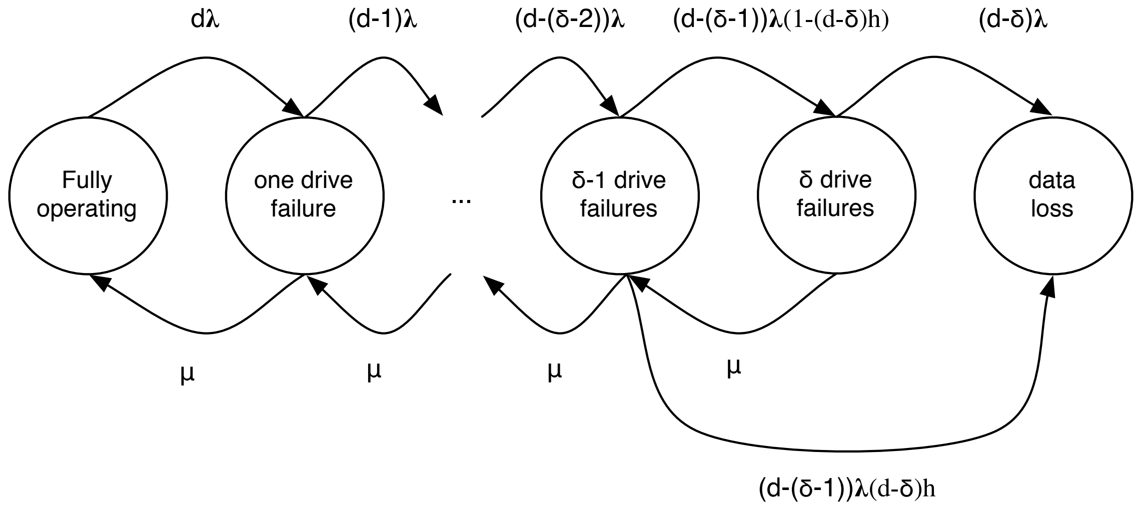


Figure 2.18.: Markov model for a δ-error tolerant array [57].

With the assumptions $\mu \gg \lambda$ and that h is many magnitudes smaller than d, the authors specify the approximate $MTTDL$ as

$$MTTDL_\delta \approx \frac{\mu^\delta}{\prod_{i=0}^{\delta}(d-i)\lambda^\delta(\lambda + h\mu)}. \tag{2.33}$$

Table 2.5 shows some typical values [58] for the parameters in the δ-error tolerant Markov model. The dependence of $MTTDL_\delta$ on d is depicted in Figure 2.19. The probability of an uncorrectable error during rebuild for a single drive is given as $h = C \cdot UER$, the product of capacity C and the unrecoverable bit error rate UER. For simplicity the array sizes are limited, such that $d \cdot h < 1$.

Table 2.5.: Typical parameters for the δ-error tolerant Markov model.

Parameter	Value
λ	$10^{-6}\frac{1}{\text{h}}$
μ	$\frac{1}{24}\frac{1}{\text{h}}$
C	1 TB
UER	$10^{-15}\frac{1}{\text{bits}}$
h	0.0086

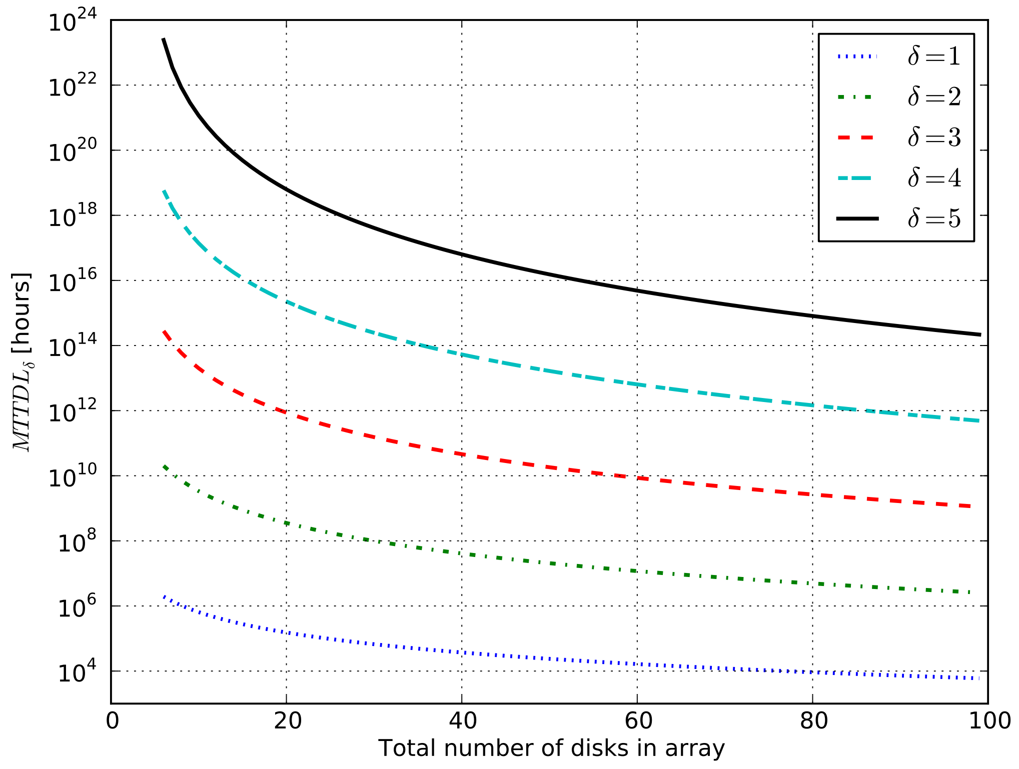


Figure 2.19.: δ-error tolerant Markov model with realistic parameters.

2.5.5. Model checking analysis

The formal method of *model checking* [59] can be utilized to analyze the design of various protection schemes. Model checking provides a means to automatically test whether a given model of a system meets an abstract specification of the system. In the context of storage systems it is used to identify execution sequences and events in case of a single error that can lead to data loss or data corruption [60]. In contrast to the failure of the entire disk described in the previous sections, the authors examine an extended set of storage errors and model their behavior:

- Latent sector errors: Data cannot be read reliably from the medium. This error results in an explicit error code reported to the system.
- Corruptions: Data stored in a disk block are corrupted by an element of the storage stack.
- Torn writes: Only a portion of the sectors of a given request is actually written to the disk (for instance when a power cycle occurs during the write).
- Lost writes: Due to firmware bugs, a success code is reported to the system although data have, in fact, not been written to the disk.
- Misdirected writes: Also due to buggy firmware, data are written to a wrong location (either within a disk or even to another disk). This type of error has the combined effect of lost writes and data corruptions.

The occurrence of these storage errors (or combinations thereof) can have three outcomes:

- The protection scheme in place is successful and data can be recovered.
- The error can be detected, but it is not possible to recover the data.
- The error cannot be detected. Corrupt data are returned to the user.

Subsequently, the following protection schemes and their combinations are evaluated when errors are injected: The classical *RAID* scheme protects only against latent sector errors that are reported by the disk drive. All other errors lead to corrupt data or parity pollution (in this case incorrect data are propagated to the parity disk). When *data scrubbing* is added to the RAID scheme, all blocks of a stripe are read and reconstructed if an error is detected. Additionally, during scrubbing the parity is recomputed and compared with the stored redundancy. If an inconsistency is detected, the parity information is recalculated and updated. While this basically adds an error-detection capability with the intention to reduce the chances of double errors, it has no effect on the single error. In the case of an incorrect data block, the inconsistency is detected during scrubbing and the parity information is subsequently polluted. In order to detect data corruption, three different *checksumming* techniques can be added: Checksums can be calculated for every disk sector, for every RAID block, or in form of a parental checksum, where the checksum is stored in a structure that is read first during user reads (for instance in the inode of a file). *Sector checksums* detect corrupt sectors but cannot protect against torn, lost, or misdirected writes. *Block checksums* consider a whole RAID block as a consistent unit and can therefore protect additionally against torn writes. However, lost or misdirected writes still go unnoticed in this scenario, since the old data (which was supposed to be overwritten) are still in a consistent state. Since *parental checksums* are only verified during user access, parity pollution during scrubbing can still occur. The authors conclude that block checksumming is most beneficial to prevent corruption. In order to detect lost writes, *write verification* can be employed: Immediately after writing, the data are read back and compared to the originals still in memory. While detecting lost and torn writes, write verification has two drawbacks: It does not protect against misdirected writes (a misdirected write is interpreted as a lost write, that is, the original write is simply reissued, and the victim of the misdirected write is overwritten with wrong data but a consistent checksum) and the read-after-write access pattern imposes a severe performance penalty. Instead of verification, different identifying characteristics can be stored along with the data to alleviate the impact on performance. A *physical identity* in form of disk and sector number or a *logical identity* in form of a inode number and file offset are the typical approaches. The physical identity can (as the sector checksum) not detect against lost writes and corrupt data can be returned to the user, whereas the logical identity works similar to parental checksums, that is, parity pollution can occur, but data loss is detected and corrupt data are not returned to the user. At last, to protect against parity pollution after a lost write, a technique called *version mirroring* can be utilized: Every RAID block in a stripe contains a version number, which is incremented with every write to that particular block. The version number of all blocks are mirrored in the parity block and with every read of a data block the corresponding version numbers are compared. In case of a version mismatch, the block with the more recent version can be used to recover from the lost write. All these techniques can now be combined to decrease the chances of data loss or corruption compared to the bare-bone RAID. The authors identify a combination of version mirroring, physical and logical identity, block checksums and RAID as the best scheme for a realistic range of disk errors. However, the study has also shown that even for the single-error case a variety of problems and corner cases can lead to error propagation and parity pollution and subsequently to data loss or, sometimes almost worse, undetected data corruption. In this context data scrubbing can even have a deteriorating effect, since it spreads originally isolated errors.

2.5.6. Non-homogeneous Poisson process (NHPP) models

The initial reliability models for RAID systems were created under that assumption that catastrophic disk failures were the dominant factor. Media degradation, latent sector errors, firmware bugs, or errors within the storage stack were considered negligible compared to full disk failures. Therefore, these models are characterized by exponential failure density functions and constant failure rates. Comparison with field data has increasingly shown that the error modes mentioned above have a significant impact on reliability and that failure rates are not constant in time [61, 62, 63]. As with the initial MTTDL models, the Markov models presented in Section 2.5.4 also assume that failures follow a homogeneous Poisson process and that failure and repair rates constant. Consequently, advanced models have been developed, that correct errors associated with the above assumption [64, 65]. The new models account for latent defects and allow rates for failures, repair, latent defects and scrubbing to take

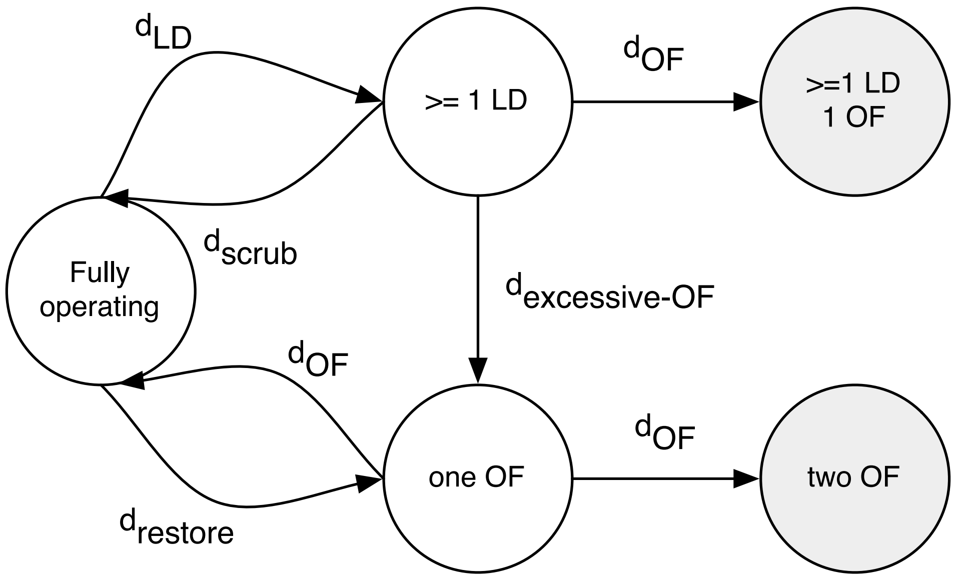


Figure 2.20.: NHPP-latent defect model for a 1-error tolerant array [64]. The grey-filled states indicate a system failure (data loss).

on any distribution. For these models hard disk failure modes and mechanisms are grouped into categories: *Operational failures* and *latent defects*. Operational failures include all failures where data cannot be found, such as bad servo-tracks, bad electronics, bad read heads, inability to stay on a track or the exceedance of certain SMART limits. Latent defects include all failures where data are missing. Errors can occur during the actual process of writing: The inherent bit error rate of the combination of all the electrical, mechanical, magnetic, and firmware systems also accounts for write errors. Magnetic media can be scratched by loose, hard particles that are stuck between the head and the surface, whereas softer particles can be smeared over the surface. Particles that were embedded during manufacturing and later on dislodged, can leave pits and voids. All these defects can be the cause for poorly written data. However, data can also be destroyed after it was successfully written to the media: Short pulses of extreme heat caused by contact of head and surface can thermally erase data. And again, soft and hard particles that are stuck between head and surface can cause scratches and smears. Together with corrosion these effects can render previously written data unreadable.

The NHPP-latent defect model proposed by Elerath and Pecht [64] is depicted in Figure 2.20 and requires four probability distributions:

- The *Time to operational failure* is modeled with a Weibull distribution d_{OF} with slightly increasing failure rate, that is, a shape parameter $\beta > 1$.
- The *Time to restore an operational failure* also uses a Weibull distribution $d_{restore}$ with a location parameter $\gamma > 0$, to take into account that an restoration requires a minimum time to complete given by capacity and bandwidth of the disk.

- The *Time to latent defect* is modeled with a exponential distribution d_{LD}.
- The *Time to scrub* again uses a Weibull distribution d_{scrub} with a non-zero location parameter, since disk scrubbing also requires a minimum time to finish.

The distribution for $d_{excessive-OF}$ is not explicit, but included in d_{OF}. The transition corresponds to a sudden burst of media defects rendering the disk inoperative. The model is evaluated using sequential Monte-Carlo techniques and appropriate parameters for all distributions, which are derived from field data. The authors compare the results in terms of data loss due to *double disk failures (DDFs)* to the results of the simple MTTDL model (Section 2.5.3) and to field data of over 7000 RAID groups. For arrays without scrubbing the predicted number of DDFs is more than 2500 times higher in the first year than in the old model. For a mission of ten years the predicted factor is 4000. The ratio decreases with decreasing characteristic scrub times, as expected. The model shows excellent correlation the actual field data and the authors conclude that the NHPP model is more suitable for modeling RAID systems and that disk scrubbing is imperative to RAID systems. However, as scrubbing is typically done in the background, higher I/O loads leave less time for scrubbing. Furthermore, increasing disks capacities also lead to higher characteristic scrubbing times.

2.6. Real world data

Reliability specifications from hard drive vendors vary considerably and are sometimes difficult to interpret. The manufacturer Seagate estimates the MTTF for a drive as the number of power-on hours (POH) per year divided by the first year *Annualized Failure Rate (AFR)* [66] . The AFR is derived from a reliability test conducted in ovens with an elevated ambient temperature of 43 °C and the highest possible duty cycle, with a run time of 28 days. For a typical data sheet MTTF of 10^6 hours and uninterrupted operation the Annualized Failure Rate is

$$AFR = \frac{8760 \text{ h}}{10^6 \text{ h}} = 0.0088. \tag{2.34}$$

To understand how useful these specified parameters (based on relatively short accelerated failure tests) actually are, one has to examine large scale field data. Several reliability studies have been published in recent years presenting field data from very large disk populations. In the following paragraphs some important results are highlighted.

A study of disks within the Google computing infrastructure by Pinheiro et al. [63] examines the influence of temperature and utilization on the failure rates. Data are collected from more than 100,000 consumer-grade disk drives with different interfaces, rotational speeds, and sizes. Their analysis gives the following results:

- AFRs vary significantly between 1.7% in the first year and over 8.6% in the 3-year old population. The infant mortality phenomenon is noticeable.

- Only for very young and very old age groups a high utilization also leads to significantly higher failure rates. Overall the correlation between failure rates and utilization is much weaker than previously suspected.
- Higher failure rates are only correlated with higher temperatures for older drives and especially high temperatures ($> 40°C$). Surprisingly, for drives younger than 3 years, lower temperatures are associated with higher AFRs.
- Some of the SMART parameters are correlated with higher AFRs, however, a large fraction of the failing drives do not show any SMART error. The failure prediction accuracy for the individual drive is therefore very limited.

Schroeder and Gibson [62] analyze more than 70,000 disk from four different vendors, deployed in compute clusters at large internet service providers and high performance computing data centers. They come to a set of conclusions that lead to a better understanding of disks failures in the field:

- Field failure rates of disk drives are significantly larger than indicated by the data sheet MTTF.
- Even younger drives (lifetime < 5 years) exhibit failure rates that are by a factor of 2-12 higher. For 5-8 year old drives the factor can be as much as 30.
- Failure rates are not constant, they increase continuously starting in year two of operation.
- The early onset of wear-out has a stronger impact on overall failure rates than infant mortality.
- The assumption of exponentially distributed time to failures can be rejected with high confidence. Instead, the Weibull distribution is more suitable.
- There is strong evidence for correlation between failures with a long range dependence.

Another study by Jiang et al. [67] investigates which components of storage systems are actually the dominant contributor for storage failures. Instead of focussing solely and the storage component, they include the whole storage subsystem. They examine about 1,800,000 disk in about 155,000 storage shelf enclosures. Some of their major findings are:

- In addition to actual disk failures, physical interconnect failures account for a similar fraction of system failures.
- AFRs for disks and full storage systems do not increase with disk size.
- Storage subsystem failures are not independent. After one failure, the probability of additional failures is higher.

An analysis of latent sector errors in a population of 1.53 million disks is presented in [61]. Interesting observations are:

- 3.45 % of the disks developed one or more latent sector errors over 32 months.
- Over 80 % of the erroneous disks show less than 50 latent sector errors.

- Near-line disks are more likely to develop latent sector errors than enterprise class disks (8.5% versus 1.9%).
- With increasing disk sizes, the fraction of disks that develop latent sector errors also increases.
- Latent sector errors are not independent. A disks with an existing latent sector error is more likely to develop additional latent sector errors.
- Latent sector error display a high locality with regard to logical sector addresses.
- Latent sector errors display a high temporal locality.
- Scrubbing detects a large percentage of latent sector errors.

An extended statistical study of the above data was done by [68]. They confirm that nearly all latent sector errors that are experienced by a drive in its lifetime are in the same 2-week window and conclude that these errors are caused by the same damaging event rather than by continuous wear-out. The authors claim that both models for latent sector errors are unrealistic: The workload dependent bit error rate, as well as a Poisson governed error arrival process. Instead they propose the Pareto distribution.

2.7. Conclusions

Individual storage components (and systems constructed from them) face a set of possible failure modes and several models of different complexity have been developed to predict their reliability. Comparison with field data shows, that many simplifying assumptions are not justified. Nearly all studies come to the conclusion, that the ability to tolerate one error is not nearly sufficient do deal with full disk failures in large scale storage systems. Partial disk failures such as latent sector errors, or a buggy disk firmware have a much higher impact on overall reliability than previously thought. In order to handle them appropriately, additional protecting techniques such as checksumming, scrubbing, versioning and identities have to be utilized to complement the RAID redundancy mechanism. All these results provide the motivation for a versatile compute-efficient coding scheme that is presented in one of the following chapters. If combined with a checksumming method this erasure-resilient scheme protects against an extended set of storage failure events.

3. Fault-Tolerance and Reliability in Current Storage Systems

In the following sections mechanisms for fault tolerance in storage systems are reviewed. At first the traditional RAID concept is shown. Subsequently, a selection of distributed large-scale storage systems and their approaches to deal with component failures is presented. Due to the high number of storage systems available (commercial and academic), only a selection of the different implementations is shown. Most of these systems can be grouped in classes with respect to their architecture and techniques to deal with component failures. Therefore, the most prominent representatives of the different classes are discussed in this chapter.

3.1. RAID (block storage)

The acronym *RAID* stands for *Redundant Array of Inexpensive Disks* and was introduced in 1988 [6]. It establishes a taxonomy of five different organizations of disk arrays (the so-called levels) with different performance and reliability characteristics. Today it is the prevailing concept for grouping disk into fault-tolerant arrays and various extensions to the original levels have been proposed. Common to all levels is an array of disks onto which data and redundant information are placed following different strategies.

3.1.1. Level 0

The RAID level 0 (shown in Figure 3.1) is strictly not a redundant array and therefore not part of the original proposition, but for completeness it is mentioned here. In RAID level 0 the data are divided into blocks (or stripes) and then distributed over the whole array in an interleaved fashion. Consequently, the blocks can be accessed in parallel which leads to a significant performance improvement. However, RAID level 0 provides no fault-tolerance.

3.1.2. Level 1

The RAID level 1 uses replication to improve reliability. Every write operation to the data disk is also replicated to one or more check disks. Level 1 provides a significantly increased reliability while

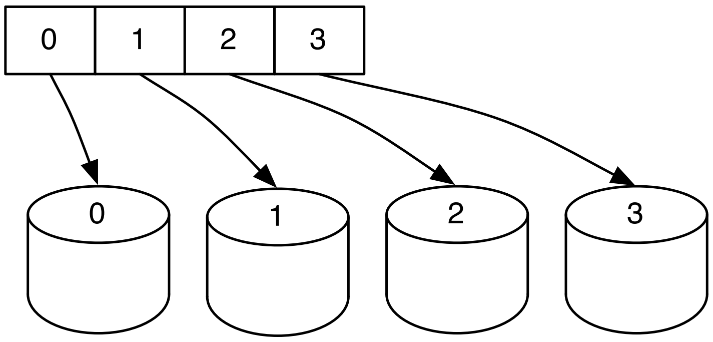


Figure 3.1.: RAID level 0: Data are divided into blocks and distributed over the array.

displaying the highest overhead costs of all RAID levels (the usable storage capacity is 50% at best). Read performance is on average slightly improved, since data can always be retrieved from either disk, preferably from the on with shorter seek and rotational delays for accessing the requested sector. Figure 3.2 shows the duplication of every data block onto two disks.

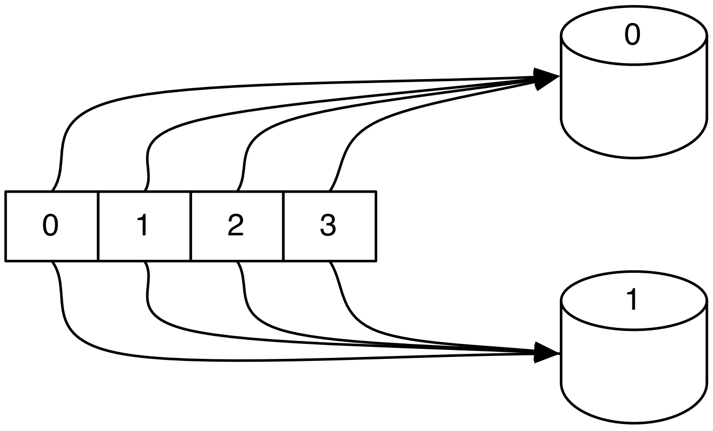


Figure 3.2.: RAID level 1: Data are replicated to at least one more disk.

3.1.3. Level 2 and Level 3

The RAID levels 2 and 3 have no practical relevance today. Data are distributed bitwise or bytewise over the array and fault-tolerance is achieved by either applying single-error-correcting Hamming codes (level 2) or by using a parity approach. Level 2 storage efficiency is better than for level 1 (the exact numbers depend on the used Hamming code and the configuration) but worse than for level 3, since exactly one more disk for the parity information is needed in every configuration. The use of bit-interleaved disks has a severe impact on performance in case of small operations, since they always

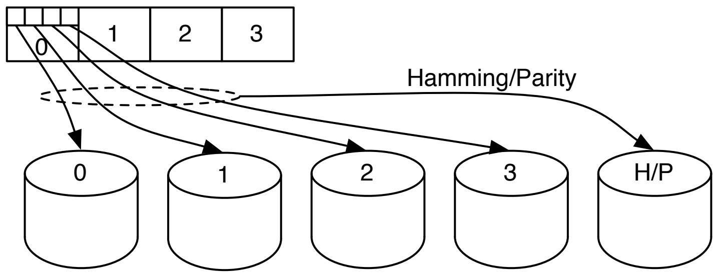


Figure 3.3.: RAID level 2 and 3. Data are distributed bitwise over the array together with Hamming check bits (level 2) or parity bits (level 3).

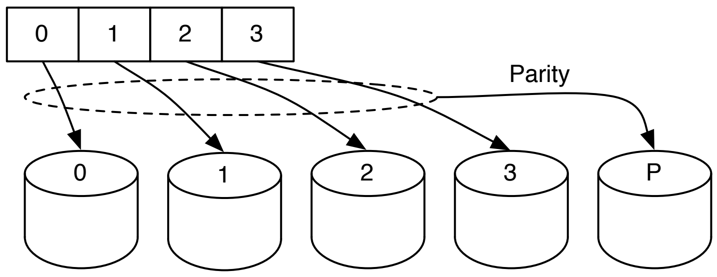


Figure 3.4.: RAID level 4. Consecutive data are divided into blocks which are striped horizontally over the disk array. One parity block is calculated per horizontal stripe.

involve read or writes of the full sector. A level 3 RAID using only two disks is equivalent to a level 1 replication.

3.1.4. Level 4

In level 4 RAID all operations are on larger blocks (ideally multiples of full hard disk sectors). A dedicated disk stores the horizontal parity of all blocks with the same (vertical) index. The parity information is calculated as the bitwise XOR of all blocks in the same horizontal group. Data are typically striped over the whole array to exploit the bandwidth of all disks for large transfers (Figure 3.4). For small write operations where only one block is overwritten, a differential update mechanism can be used to avoid reading all remaining blocks of the horizontal ensemble:

$$p_{\text{new}} = (d_{old} \oplus d_{new}) \oplus p_{old}, \tag{3.1}$$

were $p_{old/new}$ denotes the new and old parity blocks and $d_{old/new}$ the old and new data blocks, respectively. The disadvantage of this level is that the parity disk is involved in all write operations, such that the effective throughput of the array is limited by the throughput of the parity disk. Furthermore, depending on the ratio between small and large operations the parity disks experiences a higher load than the data disks, which puts it at risk of an increased failure probability. In terms of storage efficiency it is ideal, since only one additional disk is required to provide tolerance of one disk failure.

3.1.5. Level 5

To alleviate the bottleneck of a dedicated parity disk, the RAID level 5 introduces a block-interleaved scheme, while still calculating one parity block per horizontal stripe ensemble (Figure 3.5). As a result the I/O load is distributed more evenly over the whole array. Since Equation 3.1 also holds for RAID level 5, the performance for multiple small writes is improved, while conserving the good performance for large transfers. Due to its excellent storage efficiency (which is directly correlated to cost efficiency) of

$$E = \frac{N-1}{N}, \tag{3.2}$$

where N is the number of equal sized disks in the array, RAID level 5 has been the preferred mode of achieving fault tolerance in disk arrays for several years. RAID level 5 is often complemented by additional hot spare disks, in order to keep the window of vulnerability after a disk failure as small as possible. However, with growing array sizes and disk capacities the risk of data loss through latent sector errors during reconstruction has also increased, which stipulated the use of multi-error tolerant schemes.

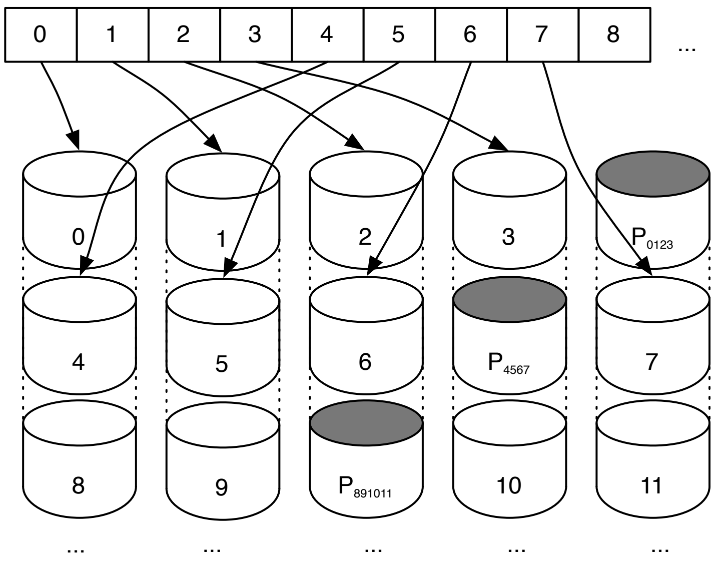


Figure 3.5.: RAID level 5. Similar to level 4, consecutive data are divided into blocks and striped over the array. Parity blocks are interleaved with the data blocks, such that the I/O load is distributed more evenly.

3.1.6. Level 6

RAID level 6 is the umbrella term for any form of RAID that can tolerate two simultaneous failures of disks, while still being able to continue operation[1] [69]. Instead of calculating a single parity block which is interleaved with the data blocks, several methods for efficiently calculating two redundant blocks have been devised: Horizontal parity can be complemented by some form of diagonal parity [70, 71, 72], such that no two elements of the diagonal ensemble reside in the same horizontal group. Another approach is to start with a more general coding scheme, such as Reed-Solomon codes, and to derive a compute efficient version for the case of 2-error tolerance. With a suitable generator polynomial the costly finite field multiplications can be transformed into a combination of XOR, AND and shift operations [73]. In practice, similar to levels 4 and 5, data are divided into blocks and striped horizontally over the array. In addition to the parity P a second redundant block Q is calculated and interleaved with P and the data blocks. The scheme is depicted in Figure 3.6

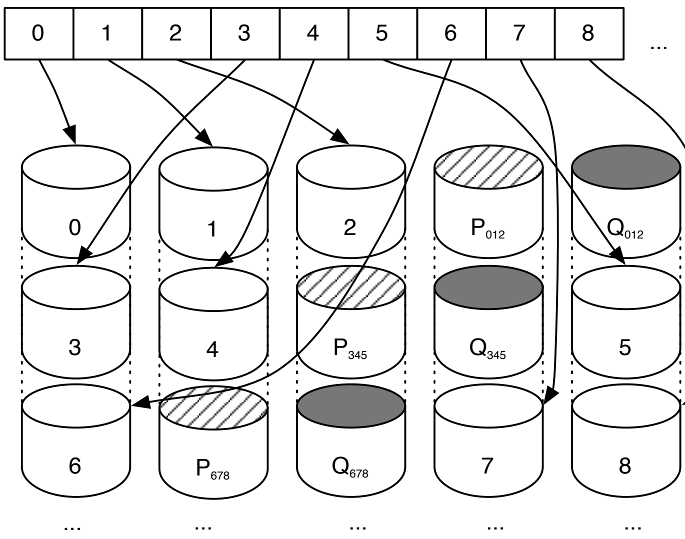


Figure 3.6.: RAID level 6.

3.1.7. Composite levels and variations

Many hybrid versions of the RAID levels exists. The most commonly used ones are combinations of striping and the higher levels: RAID01 uses a striped array as a basis which is then replicated. RAID10 follows the opposite approach where data are striped over a set of mirrored disks. In a similar fashion RAID50 stripes data over several level 5 arrays and RAID51 replicates a level 5 array. Double parity is sometimes not interleaved but stored on dedicated disks similar to RAID level 4. Some storage systems

[1] In the sense of still being able to accept read and write requests to the exported virtual block device.

include the hot spare disk into the array for load leveling and perform a reorganization of the array upon single disk failure.

3.2. ZFS

As an example from the class of local file systems *ZFS* has been selected, since it incorporates several advanced fault-tolerance and reliability mechanisms. Originally designed by Sun Microsystems, it is today developed by the Oracle Corporation and has been released as open-source software [74]. ZFS is an integrated file system and logical volume manager with a special focus on data integrity trough end-to-end checksumming. Data are always written following the copy-on-write paradigm, that is, new data are always written to a new block. The actual file system is decoupled from the physical storage devices by utilizing the concept of virtual storage pools. As the first file system ZFS uses an 128-bit address space, eliminating any practical limits to scalability which have become problematic for traditional file systems[2]. Remarkably, several fault-tolerance mechanisms have been integrated into the file system: ZFS provides a RAID5, a RAID6 and even a triple parity protection scheme, the latter one is called RAID-Z3 [75]. Although such a high degree of fault-tolerance is rarely found in current file systems, the developers have recognized the need for a high-performance generalized $n+k$ coding scheme. Another example for an advanced local file system is *btrfs (B-tree FS)* [76]. It shares many progressive features with ZFS. A file striping feature with single and dual parity protection schemes is planned.

3.3. Lustre

The name Lustre is an association of the terms linux and cluster. It is a shared and distributed file system for clusters with a POSIX interface [77]. Initially designed for the use of future object-based disks[3], it has become the most popular storage architecture for clusters and is today widely deployed in high performance computing environments and data centers. It is used as site-wide high-capacity and high-throughput file system in many supercomputers, but also as general purpose backend file system for business applications. Lustre uses a performance enhanced version of the ext4 file system to store data and metadata. In addition to standard TCP/IP networking, it supports a variety of high-performance network technologies that offer low latency and high throughput (such as InfiniBand or Myrinet) and enables simultaneous operation through routing. High availability is achieved through active/active and active/passive failover using shared and replicated storage. Lustre offers fine-grained file and metadata locking mechanisms, in order to allow any client to operate on the same file or its properties. The distribution of files and directories can be adapted to the application requirements,

[2]The maximum size of a single file in ZFS is 16 exabytes and the maximum size of one of the virtual storage pools is 256 zettabytes.

[3]A storage device similar to disks that, instead of providing a block interface, offers a flexible object interface.

starting from assignment of files or directories to certain storage locations to a RAID level 0 type striping of files. Lustre also provides export interfaces using NFS or CIFS to enable access from non-Linux clients. A Lustre cluster contains three main functional units (Figure 3.7):

- File system clients: The clients access the Lustre file system through a POSIX-compliant interface.
- Object Storage Servers (OSS): The OSS store the actual data in one or more attached Object Storage Targets (OSTs).
- Metadata Servers (MDS): The MDS manages the namespace, it handles file and directory operations and stores the metadata of the file system in a Metadata Target (MDT).

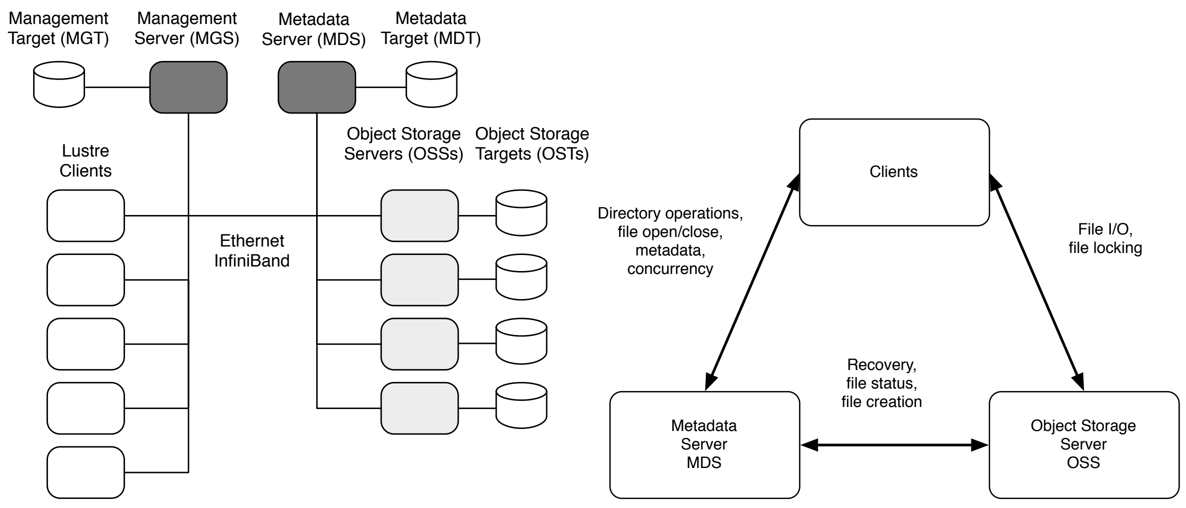

Management Target (MGT)
Management Server (MGS)
Metadata Server (MDS)
Metadata Target (MDT)
Lustre Clients
Object Storage Servers (OSSs)
Object Storage Targets (OSTs)
Ethernet InfiniBand
Clients
Directory operations, file open/close, metadata, concurrency
File I/O, file locking
Recovery, file status, file creation
Metadata Server MDS
Object Storage Server OSS

Figure 3.7.: Basic Lustre setup (left) and interactions of the Lustre components (right) [78].

The clients are usually some kind of compute node and their total number can be scaled up to 100.000. When a client opens a file it contacts the MDS for the file metadata, in particular the position of the file or its individual stripes on the various OSTs in the cluster. The intended file I/O is subsequently performed by communicating directly with the involved OSSs. A Lustre cluster has only one active MDS (at least in the current implementation), but can have several MDS on standby seeing the same MDT in order to take over in case of failure of the active MDS. The metadata typically constitute 1-2% of the total file system capacity. Every OSS can manage up to 8 OSTs and the number os OSS can practically be scaled up to 500. While Lustre offers several mechanisms for high-availability and fault tolerance, it lacks a high level approach for performance reasons. Lustre relies on shared and replicated storage at the target level, data distribution is only possible in RAID level 0 striping mode.

3.4. Ceph

Ceph is a scalable, high-performance distributed file system that advances the separation between data and metadata management [79]. Similar to Lustre, the basic architecture of Ceph includes clients, (multiple) metadata servers and object storage servers. Metadata operations are collectively managed by a cluster of metadata servers. Instead of keeping allocation lists as metadata, data and replica distribution is calculated using a pseudo-random mapping function. As a result lookups in distributed lists are replaced with computations which can be performed independently by any involved agent. Responsibility for managing parts of the file system hierarchy are assigned dynamically to the individual servers in the metadata cluster using Dynamic Subtree Partitioning. In this way Ceph adapts intelligently to varying metadata workloads and utilizes the available MDS resources much better. Another advantage of this approach is an almost linear scaling in the number of MDSs. While Lustre relies on sufficiently reliable object storage devices, Ceph recognizes the inherent unreliability of a large number of individual storage components. It is designed such that it can handle frequent failures, as well as continuos addition of new storage devices and decommissioning of older hardware. Responsibility for failure detection and recovery, data replication and migration is therefore delegated to the cluster of object storage servers, the so-called Reliable Autonomic Distributed Object Store (RADOS). Below of RADOS a POSIX-compliant file system with extended attributes is used. To the clients it offers block, file, and block storage interfaces.

3.5. GoogleFS/HDFS

The Google file system and the Hadoop Distributed File System (HDFS) represent a different class of storage systems [80, 81]. GoogleFS has been developed in conjunction with the MapReduce distributed computing framework. The framework allows the distributed processing (for instance indexing, searching, or sorting) of large data sets and has become extremely successful for a broad range of data intensive applications. Hadoop is a free and open source implementation of the framework, derived from information that was published about MapReduce. HDFS is the open source counterpart to the Google file system. Booth systems are designed for running on a large number of commodity components. Therefore, a high rate of component failures is expected and error detection and fault tolerance mechanisms are key aspects of the design. In terms of workload, the systems are tailored to the MapReduce/Hadoop requirements. The file systems are optimized for very large files (typically larger than 100 MB) which are read often. Write operations are large and sequential and mostly append data to files. Moreover, high bandwidth is favored over low latency. The architecture of both file systems is depicted in Figure 3.8. A single master (or namenode) maintains all file system metadata. Files are split up into fixed-sized chunks (64MB) and then distributed to the chunkservers (datanodes). For reliability each chunk is replicated at least three times (taking actual network and datacenter layout into account). Accordingly, metadata also include the file to chunk mapping and the chunk locations and replication

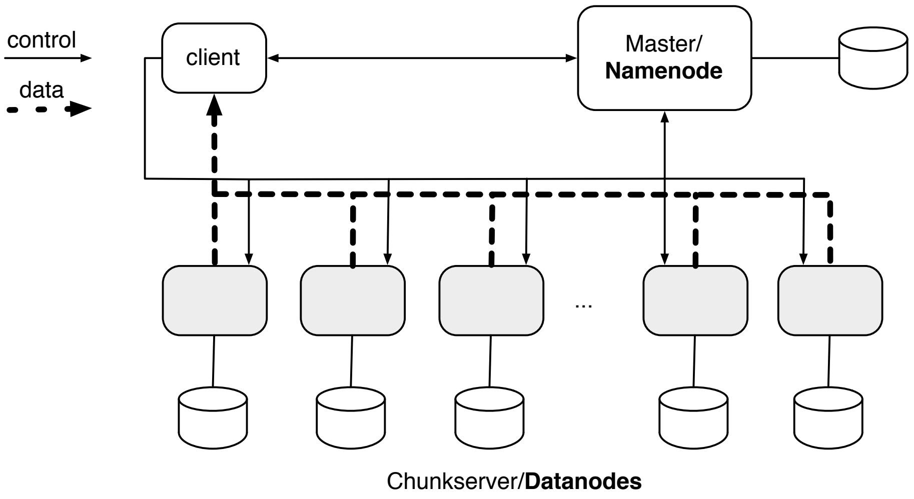


Figure 3.8.: Schematic view of the Google file system and the related HDFS [80, 81]. Names printed in bold face follow the naming convention of HDFS, regular print indicates the terms used by GoogleFS.

status. Clients use this information to directly interact with the chunkservers. The file systems do not offer POSIX-compliant interfaces, access is handled by the application framework.

3.6. Summary

Reliability and fault-tolerance are key issues for small and large scale storage systems. RAID is the standard concept for grouping disk into arrays which are able to tolerate up to two disk failures. Many large scale distributed storage systems built of commodity components address the inherent unreliability of their components by data replication. While this is the most straight-forward approach, it requires a lot more storage capacity and is not as failure-tolerant as a versatile erasure-resilient coding scheme. In conjunction with the anticipated exponential growth of digital data, this gives the motivation for a flexible coding scheme which delivers a high performance on commodity hardware.

4. Error-Correcting Codes

Whenever digital data have to be transmitted reliably over inherently unreliable channels, techniques for error detection and correction are indispensable. The theory of error control coding covers all aspects of detecting and correcting errors introduced into data during transmission through a communication channel. Unsurprisingly error-correcting codes are ubiquitous when it comes to moving, transforming, or storing any kind of information. An easy example of how error-correcting codes are used in everyday life is the *International Standard Book Number (ISBN)*. The ISBN-13 is a unique commercial identifier for books and publications. It consists of three to five elements of varying length [82] and has 13 digits in total, for example:

$$978 - 0 - 824 - 70465 - 0 \tag{4.1}$$

- A three digit prefix element (either 978 or 979).
- A group element that identifies country, region or language area (one to five digits).
- A publisher element (up to seven digits).
- A title element (up to 6 digits).
- A single check digit.

The check digit is calculated with a modulus division after summing up the digits with alternating weights of one and three:

$$d_{13} = \left(10 - \left(\sum_{i=1}^{12} d_i \cdot 3^{(i+1) \mod 2}\right) \mod 10\right) \mod 10 \tag{4.2}$$

The check digit helps to detect typographical errors or transposed digits. It has, however, a particular weakness: If adjacent digits differ by five the check bit is identical when the digits are transposed.

Apart from this simple example, error-correcting codes play an important role in various areas. They are used in practically every digital transmission such as in the package-based internet communication, cell phone networks, digital video broadcast, or communication with deep space vessels. They are also extensively used in data storage technologies, such as hard drives, compact disks, and digital versatile disks.

In the following chapter a short introduction to the theory of (linear) error- and erasure-correcting codes is presented, beginning with the basic concepts and ending with the specific ideas that were used for a novel coding scheme. Parts of this chapter are based on the presentation of the concepts in [83, 84, 85].

4.1. Introduction

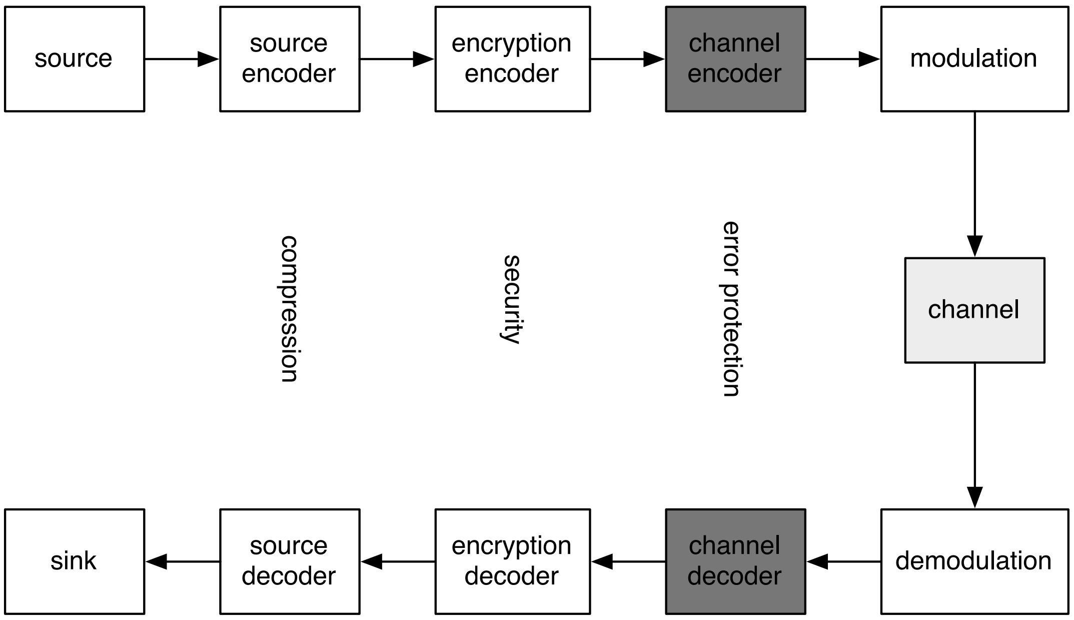


Figure 4.1.: General model of a digital communication system

An overview of a generalized digital communication system is shown in Figure 4.1. Data coming from a source pass the three encoding stages and are modulated for communication over a channel. At the receiving end of the channel the data are demodulated, decoded, and then delivered to the sink. The source contains the actual data to be communicated (basically a stream of numbers that is governed by a certain probability distribution). Each of the three encoders provides a distinct coding technique: The source encoder removes redundancy from the source, thus performing a lossless data compression. After compression the data are encrypted, that is they are transformed such that eavesdroppers cannot determine the transmitted information content. The channel encoder is responsible for adding redundant symbols to the stream for enabling error detection and correction. Not every encoder is present in every communication system, but all of them can be classified as a certain type of *code*. In the scope of this thesis only channel encoding is considered. After the stream is supplemented with redundant symbols, the message sequence is converted into signals by the modulator and send over the noisy channel. At the receiving end, the signals are demodulated and converted back in to a stream of symbols. The channel decoder then checks for introduced errors and attempts to correct them. The sequence of symbols is subsequently decrypted and decompressed. At last the data are delivered to the final destination, the information sink. In the context of storage systems the inherently unreliable storage hardware can be seen as the noisy channel. The task of a storage channel encoder is then to create redundant information that are stored alongside the actual data.

4.2. Basic definitions

Information can be transmitted by a sequence of zeros and ones[1]. While an individual 0 or 1 is called a (binary) digit, a sequence of digits is called a (binary) word. The length of a word corresponds to the number of digits in it. A binary code is a set C of words. The words that belong to C are called *code words*.

Definition 1 (Block code)
A code C with all code words having the same length is called a block code. The length of a block code is equal to the length of its code words and the number of code words in a code C is denoted by $|C|$ (size of the code).

For example, the code

$$C_1 = \{111, 000, 100, 110\}$$

is a block code of length 3 and $|C_1| = 4$.

Definition 2 (Hamming weight)
The Hamming weight w_H of a word v is the number of nonzero digits of v.

For example, the Hamming weight of the word $v = 10110$ is

$$w_H(v) = 3.$$

Definition 3 (Hamming distance)
The Hamming distance between two words $u = u_1u_2\dots u_n$ and $v = v_1v_2\dots v_n$ is the number of digits that they differ:

$$d_H(u,v) = \sum_i^n [u_i \neq v_i], \tag{4.3}$$

where

$$[u_i \neq v_i] = \begin{cases} 1 & \text{if } u_i \neq v_i \\ 0 & \text{if } u_i = v_i. \end{cases}$$

For example, the Hamming distance between the words $u = 10110$ and $v = 00111$ is

$$d_H(u,v) = 2.$$

[1]In the following a binary symmetric channel is assumed, that is, only zeroes and ones are transmitted. The probability for an error is p and the probability for a correct transmission is $1 - p$. The error probability is assumed to be independent from the value of the original bit.

With $K = \{0,1\}$ the set of the binary digits, we call K^n the set of all words of the length n. Addition of two words of K^n is defined as component-wise exclusive OR and (inner) multiplication as component-wise AND. Multiplication of a scalar element of K with an element of K^n is also defined component-wise. Together with the zero word with the 0 digit in all components, it can be shown that K^n is a vector space.

Definition 4 (Field)
A field $\mathbb{F}$ is a set of objects together with the addition operation $+$ and the multiplication operation $\cdot$ that satisfy the following conditions:

- *$\mathbb{F}$ is closed under addition and multiplication.*
- *An identity element exists for both operations.*
- *Inverse elements exists for both operations.*
- *Associativity of addition and multiplication.*
- *Commutativity of addition and multiplication.*
- *Distributivity of multiplication over addition.*

A finite field with q elements is denoted by $\mathbb{F}_q$.

Codes are not limited to binary digits as symbols. In general the alphabet for the code words is a finite field $\mathbb{F}_q$.

Definition 5 (Vector space)
Let $\mathbb{F}$ be a field of elements called scalars and V a set of elements called vectors. Let there be defined an addition operation $+$ between vectors and a scalar multiplication $\cdot$ between a scalar $a \in \mathbb{F}$ and a vector $u \in V$, such that $a \cdot u \in V$. V is called a vector space over $\mathbb{F}$ if $+$ and $\cdot$ satisfy the following conditions:

- *V forms a commutative group under $+$.*
- *For all scalars $a \in \mathbb{F}$ and vectors $v \in V$, $a \cdot v \in V$.*
- *The operations $+$ and $\cdot$ are distributive.*
- *The multiplication $\cdot$ is associative.*

Let $W \subset V$ a vector space, i.e. for any $u_1, u_2 \in W$ and any scalars $a_1, a_2 \in \mathbb{F}$ and , $a_1u_1 + a_2u_2$ is also in W. W is then called a subspace.

Definition 6 (Linear combination)
A vector u in a vector space V is called a linear combination of the vectors $v_1, \ldots, v_k \in V$ if scalars $a_1, \ldots, a_k$ exist, such that

$$u = a_1v_1 + \ldots + a_kv_k. \tag{4.4}$$

Definition 7 (Linear dependence)
A set of vectors $S = \{u_1, \dots, u_k\}$ is called linearly dependent if there exists a set of scalars $\{a_1, \dots, a_k\}$ such that

$$a_1 u_1 + \ldots + a_k u_k = 0, \tag{4.5}$$

where not all $a_i = 0$. If S is not linearly dependent, it is called linearly independent.

Definition 8 (Spanning set and linear span)
Given a vector space V and set of vectors $S = \{u_1, \dots, u_k\}$ with $u_i \in V$, the set S is called a spanning set for V if every vector $v \in V$ can be represented as a linear combination of vectors in S.

The set formed from all possible linear combinations of vectors in S is called the linear span of S, $\text{span}(S)$. *It can be shown that* $\text{span}(S)$ *is a subspace of V.*

Definition 9 (Basis and dimension)
Let V be a vector space. A spanning set for V with the smallest possible number of elements is called a basis for V. The number of vectors in the basis of V is called the dimension of V.

It can be shown that a k-dimensional vector space V over a finite field $\mathbb{F}_q$ has exactly q^k elements. In the following only vector spaces of finite dimension are considered.

Definition 10 (Inner product)
Let V be a vector space. The inner product of two vectors $u = (u_0, \dots, u_{n-1})$ and $v = (v_0, \dots, v_{n-1})$ in V, where $u_i, v_i \in \mathbb{F}$, is denoted by $u \cdot v$ and is defined as

$$u \cdot v = \sum_{i=0}^{n-1} u_i \cdot v_i. \tag{4.6}$$

Definition 11 (Orthogonality)
Two vectors u and v are called orthogonal*, if $u \cdot v = 0$. A vector u is called orthogonal to a set of vectors S, if $u \cdot v = 0$ for all elements $v \in S$.*

The set of all orthogonal vectors to a set S is called the orthogonal complement of S. It can be shown that for any subset S of a vector space V, the orthogonal complement of S is also a subspace of V.

Definition 12 (Ring)
A ring R is a set of objects together with the addition operation $+$ and the multiplication operation $\cdot$ that satisfy the following conditions:

- *R is closed under addition.*
- *An identity element exists for addition.*
- *Inverse elements exist for addition.*
- *Associativity of addition.*
- *Commutativity of addition.*

- *Distributivity of multiplication over addition.*

An expression of the form

$$f(x) = \sum_{i=0}^{n} a_i x^i, \tag{4.7}$$

with $a_n \neq 0$ is called *polynomial* of degree n. The symbol x is called the indeterminate and the constants a_i are called the coefficients of the polynomial.

Definition 13 (Polynomial ring)
Let R be a ring. The set of all polynomials with coefficients $a_i \in R$, together with the usual operations for polynomial addition and multiplication, is called the polynomial ring $R[x]$.

4.3. Linear codes

Every code can be represented by a complete list of message words and their corresponding code words. However, with growing size of the code the cost for storing the list and for performing decoding operations also increases. Using mathematical structure to describe codes can reduce the overhead considerably. One of the most important properties of codes is linearity which makes it possible to apply tools and techniques from linear algebra.

Definition 14 (Distance of a code)
The distance d of a code C is the smallest Hamming distance between any two code words,

$$d = \min_{c_i, c_j \in C, c_i \neq c_j} d_H(c_i, c_j). \tag{4.8}$$

A code C is called *linear code* if the linear combination of any two words in C is again a word in C. A more formal definition of a linear code is:

Definition 15 (Linear code)
A block code C over a finite field $\mathbb{F}_q$ of distance d, length n, and size q^k is called a linear (d,n,k) code, if and only if its code words form a k-dimensional subspace of the vector space $\mathbb{F}_q^n$ of all the n-tuples of elements of $\mathbb{F}_q$. The number k is called the dimension of the code.

With the properties of a vector space every code word c of a linear (d,n,k) code C can be represented as a linear combination of the k basis vectors $g_0, \ldots, g_{k-1}$ of the code:

$$c = m_0 g_0 + \ldots + m_{k-1} g_{k-1}, \tag{4.9}$$

where $m_i \in \mathbb{F}_q$. Interpreting the scalars $m_0, \ldots, m_{k-1}$ as a k-tuple message word $m = (m_0 \ldots m_{k-1})$, Equation 4.9 describes a method to encode message words into code words of C.

The matrix G obtained from packing the basis vectors row-wise into the $k \times n$ matrix is called the *generator matrix* of C:

$$G = \begin{bmatrix} g_0 \\ \vdots \\ g_{k-1} \end{bmatrix}. \tag{4.10}$$

The encoding operation in Equation 4.9 can then be written as

$$c = mG. \tag{4.11}$$

Instead of storing all q^k code words, a linear code is therefore completely represented by its k basis vectors of length n (or the $k \times n$ generator matrix, respectively). However, this representation of a code is not unique. A different generator matrix for the same code can for instance be obtained by performing elementary row operations on the generator matrix.

Definition 16 (Systematic code)
A code generated by a $k \times n$ matrix G whose first k columns form the $k \times k$ identity matrix $\mathbb{I}_k$ is called a systematic code,

$$G = \left[\mathbb{I}_k, X\right]. \tag{4.12}$$

Code words $c = mG$ of a systematic code consist of the original message word m in the first k digits.

An arbitrary generator matrix can always be transformed into a systematic generator matrix with Gaussian elimination.

Definition 17 (Dual code)
The orthogonal complement to a linear (d,n,k) code C is called dual code $C^\perp$. The dimension of $C^\perp$ is $n-k$.

Since $C^\perp$ is also a vector space there exists a basis with $n-k$ vectors $\{h_0, \ldots, h_{n-k-1}\}$. The matrix H formed by stacking the basis vectors as rows is called *parity check matrix*:

$$H = \begin{bmatrix} h_0 \\ \vdots \\ h_{n-k-1} \end{bmatrix}. \tag{4.13}$$

The parity check matrix H is a generator matrix for the dual code $C^\perp$ and satisfies the following:

$$GH^T = 0. \tag{4.14}$$

A linear (d,n,k) code C over $\mathbb{F}_q$ and the corresponding parity check matrix H have the following useful property: A vector $u \in \mathbb{F}_q^n$ of length n is a code word of C if and only if

$$uH^T = 0. \tag{4.15}$$

If the generator matrix of C is given in systematic form, then the parity check matrix is determined as

$$H = \left[\mathbb{I}_{n-k}, -X^T\right]. \tag{4.16}$$

4.4. Bounds for codes

Definition 18 (Singleton bound)
For any linear (d,n,k) code the distance is bounded by

$$d - 1 \leq n - k. \tag{4.17}$$

A linear (d,n,k) code with $d = n-k+1$ is called *maximum distance separable* (or MDS).

Code words and non-code words can be interpreted geometrically: For every code word of a q-ary[2] code of length n there are in general

$$p = (q-1)^l \binom{n}{l} \tag{4.18}$$

vectors at the hamming distance l from the code word. All vectors at Hamming distance $d \leq t$ constitute the so-called *Hamming sphere* of radius t. The number of vectors $V_q(n,t)$ inside an Hamming sphere of radius t for a q-ary code of length n is

$$V_q(n,t) = \sum_{j=0}^{t} (q-1)^j \binom{n}{j}. \tag{4.19}$$

Any non-code word inside a sphere of radius

$$t = \lfloor \frac{d-1}{2} \rfloor \tag{4.20}$$

around a code word is decoded to that code word[3]. In the same manner a code with error-correction ability of t must have at least a distance of

$$d \geq 2t + 1. \tag{4.21}$$

Definition 19 (Hamming bound)
For a q-ary code C of length n and distance d such that

$$t = \lfloor \frac{d-1}{2} \rfloor, \tag{4.22}$$

[2]A q-ary code is a set of code words, where each code word is a sequence of symbols which are chosen from q distinct elements.

[3]In the sense that the code word with the smallest distance to the non-code word is the most probable candidate.

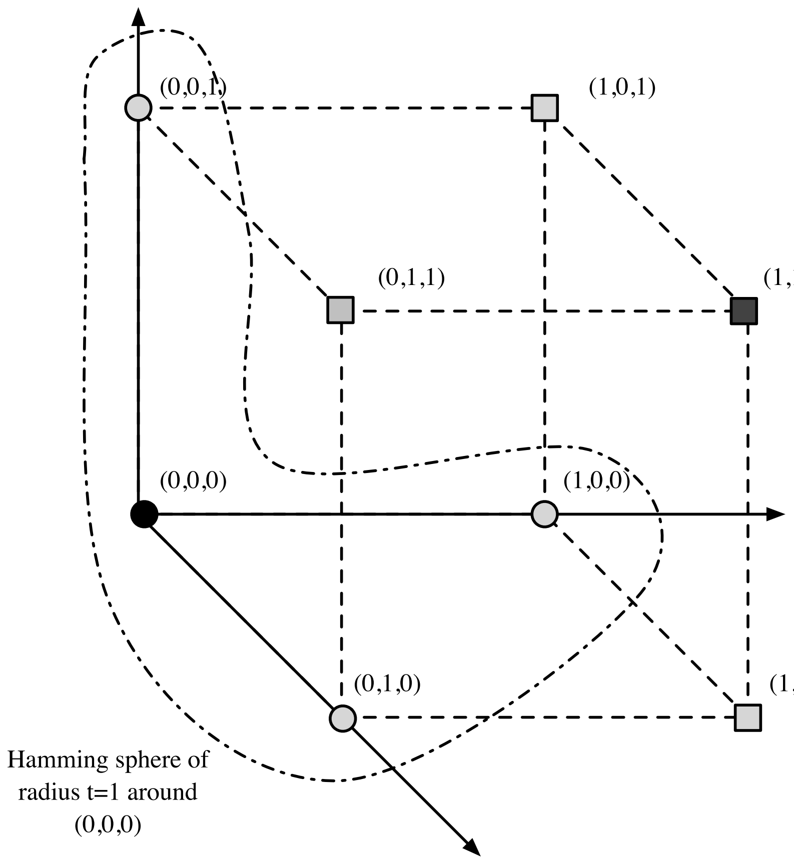


Figure 4.2.: Code words and Hamming spheres can be interpreted geometrically.

the size of C is bounded by

$$|C| \leq \frac{q^n}{\sum_{j=0}^{t}(q-1)^j\binom{n}{j}} = \frac{q^n}{V_q(n,t)}. \tag{4.23}$$

A code that attains the Hamming bound is called a *perfect code*.

4.5. Error detection and correction

Given a vector $r \in \mathbb{F}_q^n$ and the parity check matrix H for a linear (d,n,k) code C, the vector

$$s = rH^T \tag{4.24}$$

is called the *syndrome* of r. Since the syndrome is the zero vector if and only if $r \in C$, it can be used to detect errors. With the code C over $\mathbb{F}_q$, the code word $c \in C$, and the vector $r \in \mathbb{F}_q^n$, the transmission over a noisy channel can be written as the addition

$$r = c + e. \tag{4.25}$$

If the code word c is transmitted and subsequently the vector r is received, the vector e is called the *error pattern* that is introduced during transmission. The syndrome is then

$$s = rH^T = (c+e)H^T = eH^T \tag{4.26}$$

A syndrome $s \neq 0$ only contains the information that an error pattern was introduced, finding the correct code word is in general done using *maximum likelihood decoding*. For a received vector r the code word c is selected such that the Hamming distance to r is minimal:

$$c = \arg \min_{a \in C} d_H(a, r) \tag{4.27}$$

Definition 20 (Complete and bounded distance decoders)
A complete error-correcting decoder selects for every received vector r the code word c such that $d_H(c,r)$ is minimal.

A t-error-correcting bounded distance decoder selects the code word c such that $d_H(c,r) \leq t$. If no such c exists, decoding has failed.

A powerful decoding scheme is characterized by an efficient mechanism to perform the maximum likelihood decoding[4].

4.6. Erasure codes

An error where the location of the error is known is called *erasure*. In general, in the presence of e errors and ε erasures a code must have at least a distance of

$$d \geq 2e + \varepsilon + 1 \tag{4.28}$$

in order to be able to correct them. Erasure coding can be useful in various applications where code words can be interleaved. A sequence of code words $c_1, \ldots, c_L$ of length n are written horizontally in the rows of a matrix as shown in Figure 4.3. The symbols are then read out vertically and stored for example as blocks of length L on disjunct storage devices or transmitted as packets of length L over a noisy channel.

[4]One example is to pre-calculate the syndromes and the associated error patterns, instead of keeping all code words in a table and performing a minimum distance search for every received non-code word.

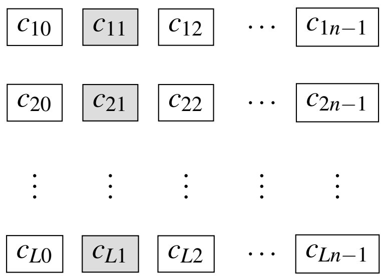

c_{10} c_{11} c_{12} $\cdots$ c_{1n-1}
c_{20} c_{21} c_{22} $\cdots$ c_{2n-1}
c_{L0} c_{L1} c_{L2} $\cdots$ c_{Ln-1}

Figure 4.3.: Code words are interleaved an written into rows of a matrix. The symbols are then used vertically (grey).

If a block or a packet is lost, an entire column of data in the matrix is erased. Due to the interleaving this corresponds to only one erased symbol in each of the L individual code words, which can be corrected.

4.7. Cyclic linear codes

For a vector $c = (c_o, \dots, c_{n-2}, c_{n-1}) \in \mathbb{F}_q^n$ we call the vector

$$\pi(c) = (c_{n-1}, c_o, \dots, c_{n-2}) \tag{4.29}$$

the *cylic shift* of c.

Definition 21 (Cyclic linear code)
A linear (d,n,k) code C is called cyclic linear code *if for every code word $c \in C$, the cyclic shift $\pi(c)$ is also in C.*

Code words and their cyclic shifts can be conveniently represented by polynomials. The code word $c = (c_0, c_1, \dots, c_{n-1})$ corresponds to the polynomial

$$c(x) = c_0 + c_1 x + \dots + c_{n-1} x^{n-1.} \tag{4.30}$$

The cyclic shift of $c(x)$ corresponds to

$$\pi(c(x)) = xc(x) \pmod{x^n - 1} \tag{4.31}$$

using the usual division process for polynomials with remainder. For polynomials $p(x)$ and $d(x)$, there exist polynomials $q(x)$ and $r(x)$ such that $p(x) = q(x)d(x) + r(x)$, where $0 \le deg(r(x)) < deg(d(x))$.

For every (d,n,k) cyclic code C there exists a unique generator polynomial $g(x)$ of degree $n-k$

$$g(x) = g_0 + g_1 x + \dots + g_{n-k} x^{n-k}. \tag{4.32}$$

Every code word in C represented as polynomial $c(x)$ can be expressed as product of a message polynomial $m(x)$ with the generator polynomial $g(x)$

$$c(x) = m(x)g(x) \tag{4.33}$$

with $deg(m(x)) < k$ and $deg(c(x)) < n$. The generator matrix G for the cyclic code is created with the generator polynomial and its first $k-1$ cyclic shifts

$$G = \begin{bmatrix} g(x) \\ \pi(g(x)) \\ \pi^2(g(x)) \\ \vdots \\ \pi^{k-1}(g(x)) \end{bmatrix}, \tag{4.34}$$

or

$$G = \begin{bmatrix} g_0 & g_1 & \cdots & g_{n-k} & & & & & \\ & g_0 & g_1 & \cdots & g_{n-k} & & & & \\ & & g_0 & g_1 & \cdots & g_{n-k} & & & \\ & & & \ddots & \ddots & & \ddots & & \\ & & & & g_0 & g_1 & \cdots & g_{n-k} & \\ & & & & & g_0 & g_1 & \cdots & g_{n-k} \end{bmatrix} \tag{4.35}$$

where all empty elements are zero. Correspondingly, there exists a *parity check polynomial* $h(x)$ with $deg(h(x)) = k$ that satisfies

$$h(x)g(x) = x^n - 1. \tag{4.36}$$

The polynomial $j(x)$ is a code word of C, if and only if

$$j(x)h(x) \mod (x^n - 1) = 0 \tag{4.37}$$

The polynomial $s(x) = j(x)h(x) \mod (x^n - 1)$ is called the *syndrome polynomial*. The $(n-k) \times n$ parity check matrix H is constructed from $h(x) = h_0 + h_1x + \ldots + h_kx^k$ as

$$H = \begin{bmatrix} h_k & h_{k-1} & \ldots & h_0 & & & & & \\ & h_k & h_{k-1} & \ldots & h_0 & & & & \\ & & h_k & h_{k-1} & \ldots & h_0 & & & \\ & & & \ddots & \ddots & & \ddots & & \\ & & & & h_k & h_{k-1} & \ldots & h_0 & \\ & & & & & h_k & h_{k-1} & \ldots & h_0 \end{bmatrix}, \tag{4.38}$$

where, again, all empty elements are zero.

4.8. Galois fields

Since the codes in the subsequent sections utilize specific properties of finite fields, the mathematical concepts are quickly reviewed in this section. In honor of the French mathematician Évariste Galois, finite fields are often referred to as *Galois fields*. A finite field containing q elements $\mathbb{F}_q$ is therefore also denoted by $GF(q)$. The order q of a Galois field must be of the form $q = p^m$, where p is a prime number and m is a positive integer. For any given q, the field $GF(q)$ is unique up to isomorphisms and therefore completely described by its size. The elements of a Galois field $GF(q = p^m)$ can be represented by polynomials of maximum degree $m-1$ and coefficients in $GF(p)$. Addition of elements is simply defined according to the conventional rules for polynomial addition. Multiplication, in order to satisfy all the requirements of a field, requires the additional step of computing the remainder modulo some special polynomial:

Definition 22 (Irreducible polynomial)
Let $f(x)$, $h(x)$, $g(x) \in R[x]$ with the degree of $h(x)$ and $g(x)$ being less than the degree of $f(x)$. The polynomial $h(x)$ is a divisor *of $f(x)$ if $f(x) = g(x)h(x)$. The polynomials $h(x) = 1$ and $h(x) = f(x)$ are called* trivial divisors.

The non-constant polynomial $f(x)$ is called irreducible over the Ring R, if it has only trivial divisors in $R[x]$.

With this, the multiplication of elements of a Galois field is defined as the usual multiplication of polynomials, followed by a modular reduction with an irreducible polynomial. A further requirement for the irreducible polynomial allows for an elegant construction of the Galois field:

Let α be an element of $GF(q)$. The *order* of α is the smallest positive integer m for which $\alpha^m = 1$.

Definition 23 (Primitive element)
$\alpha \in GF(q)$ is called primitive element, if it has order (q-1).

It can be shown that every Galois field contains at least one primitive element. A polynomial is called *monic* if the coefficient of the term of highest degree is equal to 1.

Definition 24 (Primitive polynomial)
Let $f(x)$ be a monic polynomial of degree m with coefficients in $GF(p)$. If $f(x)$ has a primitive element $\alpha \in GF(p^m)$ as one of its roots, $f(x)$ is called primitive polynomial.

If a primitive polynomial $h(x)$ is chosen for the modular reduction and $\alpha \in GF(p^m)$ is a root of $h(x)$, then the $p^m - 1$ consecutive powers of α,

$$\{1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\}, \tag{4.39}$$

are distinct and they are the $p^m - 1$ nonzero elements of $GF(p^m)$. This is the so-called *power representation* of the Galois field. Since the powers of α are unique, a nonzero element can be represented by the exponent, often referred to as the *logarithm* of the element. The multiplication of two elements α^a and α^b can thus be described as

$$\alpha^a \cdot \alpha^b = \alpha^{(a+b)} \quad \text{if} \quad (a+b) \leq (p^m - 2) \tag{4.40}$$

If $(a+b) > p^m - 1$, the order of α can be used for the modular reduction:

$$\alpha^{p^m-1} = 1 \Rightarrow \alpha^a \cdot \alpha^b = \alpha^{(a+b)} = \alpha^{p^m-1} \cdot \alpha^{a+b-(p^m-1)}. \tag{4.41}$$

In summary, the multiplication can be written as

$$\alpha^a \cdot \alpha^b = \alpha^{(a+b) \mod (p^m-1)}. \tag{4.42}$$

From the exponential representation of the nonzero elements of $GF(p^m)$, the polynomial representation can be obtained by reduction modulo the primitive polynomial. The polynomial coefficients can also be conveniently transformed into binary and decimal vector representations. Due to the byte-based nature of memory, $p = 2$ is of particular interest. The field $GF(2^4)$ for the primitive polynomial $h(\alpha) = 1 + \alpha + \alpha^4$ and all discussed representations are shown in Table 4.1.

Since for polynomials in $GF(2^m)[x]$ the coefficients are elements of $GF(2)$, the polynomial addition can be performed modulo 2. For most practical applications this corresponds to the bitwise XOR of the binary representation. The description of the multiplication using the powers of a root of the primitive polynomial gives a second practical implementation: Instead of a full polynomial multiplication modulo the primitive polynomial, the mapping between the logarithm and the vector representation can be stored in two tables (for both directions). To multiply two polynomials in vector representation the table is used to identify their logarithms, and the sum of them (including the reduction) is calculated according to Equation 4.42. With the reverse table, the vector representation of the result is then found. For small m the result of all possible multiplications in $GF(2^m)$ can also be stored in a table. The practical usefulness of the table approach, however, depends highly on the hardware architecture. As will be shown later, a fast polynomial multiplication algorithm which avoids table structures can be faster and deliver the basis for an optimized coding scheme.

4.9. BCH codes

A class of commonly used cyclic multiple error-correcting codes are the BCH codes, named after Bose, Ray-Chaudhuri, and Hocquenghem [86, 87]. As any cyclic code the BCH codes can be specified by a generator polynomial.

Definition 25 (Minimal polynomial)
The minimal polynomial *of an element $\beta \in GF(p^m)$ with respect to $GF(p)$ is the minimum-degree, nonzero, monic polynomial $f(x) \in GF(p)[x]$, such that $f(\beta) = 0$.*

Decimal	Binary	Polynomial	Power of α	Logarithm
0	0000	0	-	-
1	0001	1	α^0	0
2	0010	α	α^1	1
4	0100	α^2	α^2	2
8	1000	α^3	α^3	3
3	0011	$\alpha+1$	α^4	4
6	0110	$\alpha^2+\alpha$	α^5	5
12	1100	$\alpha^3+\alpha^2$	α^6	6
11	1011	$\alpha^2+\alpha+1$	α^7	7
5	0101	α^2+1	α^8	8
10	1010	$\alpha^3+\alpha$	α^9	9
7	0111	$\alpha^2+\alpha+1$	α^{10}	10
14	1110	$\alpha^3+\alpha^2+\alpha$	α^{11}	11
15	1111	$\alpha^3+\alpha^2+\alpha+1$	α^{12}	12
13	1101	$\alpha^3+\alpha^2+1$	α^{13}	13
9	1001	α^3+1	α^{14}	14

Table 4.1.: The elements of $GF(2^4)$ and their different representations, constructed using $h(\alpha) = 1+\alpha+\alpha^4$.

Definition 26 (Root of unity)
An element $\beta \in GF(p^m)$ with $\beta \neq 1$ is called nth root of unity *if $\beta^n = 1$. If n is the smallest integer for which $\beta^n = 1$, then β is called a* primitive *nth root of unity.*

The coefficients of the generator polynomial (the digits of the code word) are in $GF(p)$, whereas the roots of the polynomial are in $GF(p^m)$. A t-error-correcting BCH code of length n can be constructed as follows:

1. Find the minimum m, such that $GF(p^m)$ has a primitive nth root of unity β.
2. Choose a nonnegative integer b.
3. Create a sequence of the $2t$ consecutive powers of β,
$$\beta^b, \beta^{b+1}, \dots, \beta^{b+2t-1},$$
and determine the minimum polynomial for each of the powers of β with respect to $GF(p)$.
4. The generator polynomial $g(x)$ is given by the least common multiple of all these minimal polynomials.

The dimension of the code is $n - deg(g(x))$ and the distance satisfies $d \geq 2t+1$. Let

$$c(x) = m(x)g(x) \tag{4.43}$$

a code polynomial. Since all $2t$ selected powers of β are chosen to be the roots of $g(x)$,

$$c(\beta^i) = m(\beta^i)g(\beta^i) = 0, \tag{4.44}$$

or,

$$c_0 + c_1\beta^i + \ldots + c_{n-1}(\beta^i)^{(n-1)} = 0, \tag{4.45}$$

for $i = b, b+1, \ldots, b+2t-1$.

With $s = 2t+1$, the parity-check matrix H is obtained from the set of equations in 4.45 as

$$H = \begin{bmatrix} 1 & \beta^b & \beta^{2b} & \ldots & \beta^{(n-1)b} \\ 1 & \beta^{b+1} & \beta^{2(b+1)} & \ldots & \beta^{(n-1)(b+1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \beta^{b+s-3} & \beta^{2(b+s-3)} & \ldots & \beta^{(n-1)(b+s-3)} \\ 1 & \beta^{b+s-2} & \beta^{2(b+s-2)} & \ldots & \beta^{(n-1)(b+s-2)} \end{bmatrix}. \tag{4.46}$$

Let the received polynomial $r(x)$ be the sum of the code polynomial $c(x)$ and the error pattern $e(x)$:

$$r(x) = c(x) + e(x). \tag{4.47}$$

With Equation 4.45 the syndromes S_j with $j = 1, 2, \ldots, 2t$ (assuming $b = 1$) are obtained by

$$S_j = r(\beta^j) = 0 + e(\beta^j) = \sum_{k=0}^{n-1} e_k \beta^{jk}. \tag{4.48}$$

4.10. Reed-Solomon codes

Reed-Solomon codes have been developed by I. Reed and G. Solomon in 1960 [7]. They are related to the BCH codes, however, the digits in the code words are no longer elements of the base field $GF(p)$, but elements of $GF(p^m)$.

The original construction method creates a polynomial from the message vector and evaluates this polynomial at all nonzero elements of $GF(p^m)$ (given by the powers of one primitive element α):

Definition 27 (Reed-Solomon codes I)
Let $m = (m_0, m_1, \ldots, m_{k-1}) \in GF(p^m)^k$ be a message vector with components in $GF(p^m)$. The associated polynomial is $m(x) = m_0 + m_1x + \ldots + m_{k-1}x^{k-1} \in GF(p^m)[x]$. With some primitive element $\alpha \in GF(p^m)$, $n = p^m - 1$ and the code vector $c = (c_0, c_1, \ldots, c_{n-1})$, the encoding is defined by:

$$c = (c_0, c_1, \ldots, c_{n-1}) = (m(1), m(\alpha), m(\alpha^2), \ldots, m(\alpha^{n-1})) \tag{4.49}$$

A complete set of code words is obtained by evaluating all p^{mk} possible message vectors. The Reed-Solomon code is a $(n-k+1,n,k)$ linear MDS code.

Accordingly to the BCH codes the Reed-Solomon codes can also be formulated with the help of a generator polynomial. While both formulations are equivalent, the following definition is most commonly used.

Definition 28 (Reed-Solomon codes II)
For the t-error-correcting Reed-Solomon code of length (p^m-1) with symbols in $GF(p^m)$ and a non-negative integer b, the generator polynomial is constructed such that it has the $2t$ consecutive powers of some primitive element $\alpha \in GF(p^m)$ as roots.

$$g(x) = \prod_{j=b}^{b+2t-1} (x-\alpha^j) \tag{4.50}$$

It can be shown that $deg(g(x)) = 2t$ and therefore the dimension $k = n-2t$. Since any of the k “digits” in the message vector can be any element of $GF(p^m)$, the size of the code is p^{mk}.

Similar to equation 4.33, the encoding is done by polynomial multiplication,

$$c(x) = m(x)g(x). \tag{4.51}$$

Equation 4.50 can be expanded into

$$g(x) = g_0 + g_1x + \ldots + g_{2t-1}x^{2t-1} + x^{2t}, \tag{4.52}$$

where all coefficients $g_i \in GF(p^m)$ and a generator matrix can be obtained analogously to Equation 4.35.

4.11. Decoding of BCH and Reed-Solomon codes

The general outline for algebraic decoding of the BCH and the Reed-Solomon codes is as follows:

- Syndrome computation: Using the fact that the $2t$ consecutive powers of some field element β are the roots of the generator polynomial, the $2t$ syndromes can be calculated.
- Calculation of the *error locator polynomial*.
- Determination of the roots of the error locator polynomial. The roots indicate the location of the error.
- For non-binary codes the *error values* have to be determined in order to being able to correct the errors.

The relationship between syndromes and error pattern has been shown in Equation 4.45. Assuming that v errors at locations $i_1, i_2, \ldots, i_v$ have occurred, the syndromes can be rewritten such that

$$S_j = \sum_{k=0}^{n-1} e_k \beta^{jk} = \sum_{l=1}^{v} e_{i_l} (\beta^j)^{i_l} = \sum_{l=1}^{v} e_{i_l} (\beta^{i_l})^j. \tag{4.53}$$

With the so-called *error locators* $X_l = \beta^{i_l}$ and the *error values* e_{i_l} one gets $2t$ equations:

$$S_j = \sum_{l=1}^{v} e_{i_l} X_l^j \qquad j = 1, 2, \ldots, 2t, \tag{4.54}$$

or,

$$\begin{bmatrix} X_1 & X_2 & \ldots & X_v \\ X_1^2 & X_2^2 & \ldots & X_v^2 \\ \vdots & \vdots & & \vdots \\ X_1^{2t} & X_2^{2t} & \ldots & X_v^{2t} \end{bmatrix} \begin{bmatrix} e_{i_1} \\ e_{i_2} \\ \vdots \\ e_{i_v} \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{2t} \end{bmatrix} \tag{4.55}$$

In order to find the error locators the *error locator polynomial* $\Lambda(x)$ is defined:

$$\Lambda(x) = \prod_{l=1}^{v} (1 - xX_l) = 1 + \Lambda_1 x + \ldots + \Lambda_{v-1} x^{v-1} + \Lambda_v x^v \tag{4.56}$$

If $x = \frac{1}{X_l}$, then $\Lambda(x) = 0$. That is, the roots of this polynomial are the reciprocals of the error locators. It can be shown that the syndromes and the coefficients Λ_i are related by

$$S_k + \Lambda_1 S_{k-1} + \ldots + \Lambda_{k-1} S_1 + k\Lambda_k = 0 \quad 1 \leq k \leq v \tag{4.57}$$

$$S_k + \Lambda_1 S_{k-1} + \ldots + \Lambda_{v-1} S_{k-v+1} + \Lambda_v S_{k-v} = 0 \quad k > v. \tag{4.58}$$

From this relationship a system of linear equations can be obtained and it can be solved for the coefficients of the error locator polynomial:

$$\begin{bmatrix} S_1 & S_2 & \ldots & S_v \\ S_2 & S_3 & \ldots & S_{v+1} \\ S_3 & S_4 & \ldots & S_{v+2} \\ \vdots & \vdots & & \vdots \\ S_v & S_{v+1} & \ldots & S_{2v-1} \end{bmatrix} \begin{bmatrix} \Lambda_v \\ \Lambda_{v-1} \\ \Lambda_{v-2} \\ \vdots \\ \Lambda_1 \end{bmatrix} = - \begin{bmatrix} S_{v+1} \\ S_{v+2} \\ \vdots \\ S_{2v} \end{bmatrix} \tag{4.59}$$

Once the coefficients are determined, the roots of Equation 4.56 can be found, for instance through exhaustive search. With these roots the error locators are easily obtained. Now Equation 4.55 can be solved for the actual error values e_{i_l} and the original message can be reconstructed by subtracting the error pattern from the received polynomial.

Since the algebraic decoding method is computationally relatively complex, several algorithms have been developed to perform certain steps more efficiently: *Chien search* is a fast algorithm for finding roots of polynomials over finite fields without evaluating every field element [88]. *Forney's algorithm* [89] uses Lagrange interpolation to find the error values, instead of solving the system of linear equations in 4.55. An iterative procedure for finding the error locator polynomial (especially when the number of errors is not known) is given by the *Berlekamp-Massey algorithm* [90, 91]

By design, Reed-Solomon codes are able to correct

$$t = \lfloor (n-k)/2 \rfloor \tag{4.60}$$

errors. Provided that the error locators X_l are already known, that is, an erasure has occurred, Equation 4.55 can be solved directly for the error values. In case of a mixture of e errors and ε erasures, a Reed-Solomon code of distance d is capable of decoding a message vector correctly, if

$$2e + \varepsilon \leq d - 1. \tag{4.61}$$

When dealing with erasures only, the maximum number of tolerable erasures τ is related to the dimension and the length of the code by

$$\tau = d - 1 = n - k. \tag{4.62}$$

4.12. Vandermonde-based Reed-Solomon erasure codes

For an n-tuple $(x_1, x_2, \cdots, x_n)$ of elements of a field the Vandermonde matrix is defined as

$$V(x_1, x_2, \dots, x_n) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}. \tag{4.63}$$

The determinant of a square Vandermonde matrix can be determined by [92]:

$$\det(V(x_1, x_2, \dots, x_n)) = \prod_{1 \leq k < j \leq n} (x_j - x_k) \tag{4.64}$$

With Equation 4.64 it is easy to see that the square Vandermonde matrix is invertible if and only if the x_i are pairwise distinct. This property can now be used to construct a generator matrix for a linear code:

The Vandermonde matrix is chosen such that

$$V = (v_{i,j}) = i^j, \tag{4.65}$$

and therefore the $n \times m$ Matrix ($m < n$)

$$V_{i,j} = \begin{pmatrix} 0^0 & 0^1 & \cdots & 0^{m-1} \\ 1^0 & 1^1 & \cdots & 1^{m-1} \\ \vdots & \vdots & & \vdots \\ (n-1)^0 & (n-1)^1 & \cdots & (n-1)^{m-1} \end{pmatrix}. \tag{4.66}$$

is obtained (with the definition of $0^0 = 1$). It is evident that this matrix (and any other Vandermonde matrix) is non-singular if any $n-m$ rows are removed: The remaining square matrix contains still rows with pairwise distinct x_i and is therefore still invertible. In order to use this matrix for generating a systematic linear code it has to be transformed into an information dispersal matrix where the $m \times m$ matrix in the first n rows is the identity matrix. This can be achieved by applying a sequence of elementary matrix transformations [93]:

- Any column C_i may be swapped with column C_j.
- Any column C_i may be multiplied with non-zero a
- To any column C_i the multiple of another column may be added: $C_i \leftarrow C_i + a \cdot C_j$, where $i \neq j$ and $a \neq 0$.

Since elementary matrix transformations do not change the rank of the matrix, the derived matrix maintains the properties of the Vandermonde matrix and has the following form:

$$V^* = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ v^*_{m,0} & v^*_{m,1} & \cdots & v^*_{m,m-1} \\ \vdots & \vdots & & \vdots \\ v^*_{n-1,0} & v^*_{n-1,1} & \cdots & v^*_{n-1,m-1} \end{pmatrix}. \tag{4.67}$$

V^* is the generator of the systematic linear Vandermonde-based Reed-Solomon code. Encoding is performed as usual by multiplying the generator with the vector of data words $d = (d_0, d_1, \cdots, d_{m-1})$, where $d_i \in GF(2^p)$ and $2^p > n$. The results consist of the original data words in the first $m-1$ rows and the check words $c = (c_0, c_1, \cdots, c_{n-m-1})$:

$$V^*d = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ v^*_{m,0} & v^*_{m,1} & \cdots & v^*_{m,m-1} \\ \vdots & \vdots & & \vdots \\ v^*_{n-1,0} & v^*_{n-1,1} & \cdots & v^*_{n-1,m-1} \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \\ c_0 \\ c_1 \\ \vdots \\ c_{n-m-1} \end{pmatrix} = \begin{pmatrix} d \\ c \end{pmatrix} \tag{4.68}$$

In case of an erasure of a data word d_i, the i-th row of V^* and the i-th component of the resulting vector have to be deleted. After exactly $n-m$ data rows are removed, a non-singular square matrix $V^\dagger$ remains, together with resulting vector of the surviving data and check words $(d^\dagger, c)^T$:

$$V^\dagger d = V^\dagger \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \end{pmatrix} = \begin{pmatrix} d^\dagger \\ c \end{pmatrix} \tag{4.69}$$

The matrix $V^\dagger$ is guaranteed to be invertible and the original vector can be reconstructed as:

$$(V^\dagger)^{-1} V^\dagger d = (V^\dagger)^{-1} \begin{pmatrix} d^\dagger \\ c \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \end{pmatrix} \tag{4.70}$$

In case of erasures of the check symbols c_i, the encoding procedure in Equation 4.68 can simply be repeated.

When one of the data words d_j is changed into d'_j all check words have to be updated:

$$V^* d = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_j \\ \vdots \\ d_{m-1} \\ c_0 \\ c_1 \\ \vdots \\ c_{n-m-1} \end{pmatrix} \quad \Rightarrow \quad V^* \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d'_j \\ \vdots \\ d_{m-1} \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d' j \\ \vdots \\ d_{m-1} \\ c'_0 \\ c'_1 \\ \vdots \\ c'_i \\ \vdots \\ c'_{n-m-1} \end{pmatrix}. \tag{4.71}$$

Subtraction of both systems gives

$$V^* \begin{pmatrix} 0 \\ \vdots \\ 0 \\ d_j - d'_j \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ d_j - d'_j \\ 0 \\ \vdots \\ 0 \\ c_0 - c'_0 \\ c_1 - c'_1 \\ \vdots \\ c_{n-m-1} - c'_{n-m-1} \end{pmatrix}. \tag{4.72}$$

Since the first m rows of V^* are the identity matrix only the last $(n-m)$ check words are of interest and give the following relationship between changed data words and corresponding updates of the check words:

$$v^*_{i,j}(d_j - d'_j) = (c_i - c'_i) \tag{4.73}$$

or

$$c'_i = c_i - v^*_{i,j}(d_j - d'_j) \tag{4.74}$$

4.13. Cauchy Reed-Solomon codes

Cauchy Reed-Solomon codes [94] are very similar to the Vandermonde-based codes but they come with two important modifications. Instead of utilizing the Vandermonde matrix as a starting point to generate the code, they use a *Cauchy matrix* with similar properties. Conveniently, square $n \times n$ Cauchy matrices may be inverted with $O(n^2)$ Galois Field operations [95].

Definition 29 (Cauchy matrix)
Let $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$ be two sets of elements of a field $\mathbb{F}$ such that

a) $\forall i \in 1, \dots, m$ and $\forall j \in 1, \dots, n$: $x_i + y_j \neq 0$.

b) $\forall i, j \in 1, \dots, m, i \neq j$: $x_i \neq x_j$ and $\forall i, j \in 1, \dots, n, i \neq j$: $y_i \neq y_j$.

The $(m \times n)$ Cauchy matrix over $\mathbb{F}$ is defined as:

$$C = \begin{pmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_{m-1}+y_1} & \frac{1}{x_{m-1}+y_2} & \cdots & \frac{1}{x_{m-1}+y_n} \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \cdots & \frac{1}{x_m+y_n} \end{pmatrix} \tag{4.75}$$

The determinant of a square Cauchy matrix can be determined by

$$det(C) = \frac{\prod_{i<j}(x_i - x_j)\prod_{i<j}(y_i - y_j)}{\prod_{i,j=1}^{n}(x_i + y_j)}, \tag{4.76}$$

and it can be shown that every sub-matrix of a Cauchy matrix is again a Cauchy matrix and that every square sub-matrix of a Cauchy matrix is non-singular [92]. The matrix formed by the identity matrix $\mathbb{I}$ in the first n rows and the a Cauchy matrix C over $GF(2^p)$ in the remaining m rows,

$$G = (g_{i,j}) = (\mathbb{I}, C), \tag{4.77}$$

is the generator of a systematic code with word size p, where $n + m \leq 2^p$. The second modification is to use a special field isomorphism to convert operations over $GF(2^p)$ into XOR operations over $GF(2)[x]/(h(x))$, the field of polynomials $GF(2)[x]$ modulo an irreducible polynomial $h(x)$ of degree p:

- Elements α in $GF(2^p)$ can be identified with polynomials $\alpha_0 + \alpha_1 x + \cdots + \alpha_{p-1}x^{p-1}$ where $\alpha_i \in GF(2)$. It can also be represented by a column vector $(\alpha_0, \alpha_1, \cdots, \alpha_{p-1})^T$ in $GF(2)^p$.
- An element can also be represented by an $(p \times p)$ bit matrix $\tau(\alpha)$, where there i^{th} column is the column vector of $\alpha x^{i-1} \mod h(x)$.

With these projections the generator matrix of the Cauchy Reed-Solomon codes is simply

$$G^* = (\tau(g_{ij})). \tag{4.78}$$

All arithmetic is subsequently done over $GF(2)$, that is, multiplication can be performed using bitwise AND operations and addition with XOR operations. Encoding and decoding is performed as usual, after the data words have been transformed into their bit representation. The costly polynomial arithmetic has been replaced with a (typically) higher number of modulo 2 operations.

4.14. Summary

The concepts of linear error and erasure correcting codes have been reviewed briefly in this chapter. Starting with basic block codes, the property of linearity quickly leads to a description of codes in terms of generator matrices. Cyclic codes allow an even more compact description with the help of a generator polynomial. The introduction of Galois fields enables the construction of the cyclic BCH codes with powerful correction capabilities. Finite fields are also required by the Reed-Solomon codes that are the foundation for several erasure correcting codes suitable for distributed storage systems. The concepts of non-binary codes over finite fields are essential for the following chapter, where a linear MDS code specifically tailored to modern processor architectures is constructed.

5. A SIMD-optimized Coding Scheme

In this chapter a novel coding scheme is introduced. The scheme is based on a linear non-binary erasure correcting code over a finite field (similar to Reed-Solomon codes). For the essential multiplication operation a polynomial algorithm is presented, which displays a high degree of arithmetic intensity and is therefore particularly suitable for execution on the vector units of general purpose processors and on modern many-core accelerator devices. In contrast to previous schemes the generator matrices are specifically tailored to exploit the features of the processor architectures and avoid their bottlenecks. This is achieved by aiming for a specific distribution of the polynomial coefficients that represent the elements of the generator matrix. Several approaches to create suitable generator matrices are elaborated and their implications are discussed. A Monte-Carlo based construction makes it possible to influence the specific shape of the matrices and to hereby adapt them to special usage scenarios. Parts of this chapter have been previously published in [96].

5.1. Introduction

The Galois fields of the form $GF(2^r)$ are naturally of particular interest in storage applications. The efficiency and performance of these applications depend considerably on the implementation of Galois field arithmetic. While the addition of two elements of $GF(2^r)$ can be carried out by a comparatively inexpensive bitwise XOR of the binary representation of the elements, multiplication of two elements is performed by multiplying two polynomials which represent the elements. To avoid the computational cost of a polynomial multiplication, implementations usually use pre-calculated lookup tables for the logarithm and its inverse. The use of 2-dimensional tables that store the multiplication of any two elements is also common. However, on modern computer architectures the table-based strategy has serious shortcomings: Due to the growing disparity between the speed of the CPU and the dynamic access random memory (DRAM) latencies, also known as the *memory wall* [97], modern processing units are able to execute several hundreds of instructions during the time required to service an uncached memory load [9]. The situation gets worse for modern multiprocessors systems commonly employing cache coherent non-uniform memory architectures (NUMA). In contrast to the traditional front-side-bus (FSB) approach, memory is directly connected to a particular CPU in NUMA systems (Figure 5.1). This increases the totally available memory bandwidth, but also increases the latency when the CPU is accessing a remote memory region. In this case DRAM latencies can be in the order of a thousand processor instructions [98]. As a consequence, modern operating systems are equipped with various

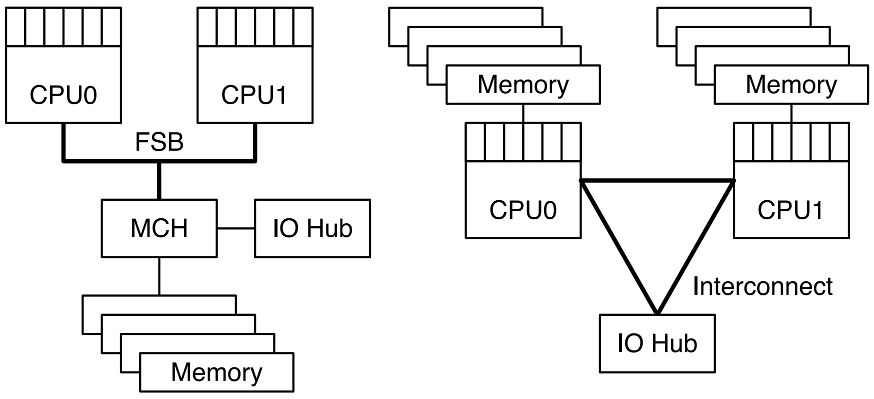


Figure 5.1.: Memory organizations of multiprocessor systems. The traditional frontside bus model is depicted on the left, the NUMA model is shown on the right [100].

mechanisms to ensure optimal process placement and memory allocation and even allow for migration of pages between memory domains to minimize NUMA latencies [99].

Another problem arises from the increasing number of individual cores in multiprocessors paired with the stagnation of processor clock frequencies. The architecture of the memory subsystem, including memory controllers and caches, necessarily becomes more complex. This introduces serious performance limitations and the implications have to be well understood in order to achieve reasonable efficiency. In addition to the scaling in terms of processor cores, modern processors allow for the vectorization of applications by extending previously implemented scalar instructions [101] according to the *Single Instruction, Multiple Data (SIMD)* [102] paradigm. In particular applications processing large bit streams can benefit from using such SIMD instructions, while certain classes of (flow-control heavy) algorithms cannot be efficiently vectorized. A trend of growing vector units (in terms of elements that can be processed in parallel) in commodity processors can be foreseen for the next years [103]. Another approach to achieve high performance in a high-throughput computing task, is the use of an external accelerator. In particular graphics processing units capable of performing general purpose computations (GPGPUs) have been established as versatile co-processors in the recent years. GPGPUs are massively parallel many-core devices (with typically hundreds or even thousands of small cores) that are connected to a host system through PCI Express. GPGPUs have their own private memory and thus operate in a separate address space. In a typical co-processor model, data have to be transferred from the host memory to the GPGPU memory, and are then processed on the device. Subsequently the results have to be transferred back. Due to exceptionally high numbers of execution units (and a correspondingly high theoretical peak performance), versatile programming models, and relatively low prices, GPGPUs have become a popular accelerator platform.

With these boundary conditions in mind, a novel coding scheme for storage applications is presented in the following sections. The scheme replaces the costly table-lookups for Galois multiplications with arithmetic instructions in order to limit memory references to a minimum. Furthermore, the scheme exploits the fact that the execution time of the multiplication is determined by the distribution of the polynomial coefficients in the multiplicands. The distribution of the corresponding bits in the generator matrix of a linear block code thus has an impact on the overall performance of the encoding operation. Therefore, several methods for creating performance-optimized generator matrices with different characteristics have been evaluated.

5.2. Table-based multiplication

It has been shown in Chapter 4 that a Galois field can be constructed by taking the powers of some root α of the primitive polynomial that generates the field (α is called a *primitive element* of the field). Every element of the field is itself a polynomial and the coefficients can be used to give a binary or decimal representation of the element. Another representation is given by the powers of the primitive element: every field element corresponds to a unique power of α. Multiplication of two elements in $GF(2^r)$ can thus be implemented in several ways:

- Multiplication of the polynomials modulo a primitive polynomial.
- Using a full pre-computed multiplication table requiring space for $(2^r)^2$ elements.
- Using pre-computed tables for the logarithm and its inverse of the primitive element. This requires space for $2 \cdot 2^r$ elements. The tables provide the mapping between an element of the field and its corresponding power of α. Multiplication of two elements $a = \alpha^i$ and $b = \alpha^j$ is then

  $$\alpha^i \alpha^j = \alpha^{\left[log_\alpha(\alpha^i)+log_\alpha(\alpha^j)\right] \mod (2^r-1)} = \alpha^{[i+j] \mod (2^r-1)}. \tag{5.1}$$

  The pseudocode for this multiplication is shown in Figure 5.2.

Prior implementations have regarded the polynomial multiplication as too costly and the lookup tables were the preferred method. While this was a successful strategy for several generations of microprocessors, today the memory subsystem has become one of the main bottlenecks in commodity systems. Accessing large in-memory data structures from inner loop code can thus severely impact the overall performance and deteriorate scalability. Evidently, the table sizes are highly dependent on the size of the underlying Galois field. Resulting memory requirements for both table approaches for different Galois fields are shown in Table 5.1. Because of the byte-based nature of the system memory the comparison is limited to the standard word sizes of 8, 16 and 32 bit. Due to the increasing limitations of the memory subsystem, the lookup strategy can only be efficient if a large fraction of the tables can be kept in the processor cache. With current L2/L3 cache sizes in the order of several MiB, it is apparent that this approach is limited to symbol sizes of 8 bit or 16 bit which is only a small fraction of the native symbol size of modern processors (typically 64 bit).

```
procedure GFMULT($a, b$)
    if $a = 0$ OR $b = 0$ then
        return 0
    end if
    $sum \leftarrow log[a] + log[b]$
    if $sum \geq 2^l - 1$ then
        $sum \leftarrow sum - 2^l - 1$
    end if
    return $inv\_log[sum]$
end procedure
```

Figure 5.2.: Multiplication in a Galois field using pre-computed lookup tables. The modulo operation is implemented as subtraction.

Table 5.1.: Sizes of the lookup tables for logarithm and inverse or the full multiplication depending on the size of the underlying Galois fields.

$GF(2^r)$	Log/Log^{-1}	Full
$r =$	(KiB)	(KiB)
8	0.5	64
16	256	$8.4 \cdot 10^6$
32	$33.6 \cdot 10^6$	$7.2 \cdot 10^{16}$

5.3. Polynomial implementation

The foundation for the implementation of the polynomial multiplication is the well known *Russian peasant multiplication* algorithm [104] (variations are also known as *Egyptian multiplication*): To multiply two integers, they are written next to each other into two columns. The left-hand column is successively halved (rounding down), while simultaneously the right-hand column is doubled, until 1 is reached in the left-hand column. Now each line with even numbers in the left-hand column is crossed out and the remaining rows of the right-hand column are added up. During the process numbers are reordered in groups of the left-hand number. The lines with the odd numbers represent the remainders during the halving step. The sum of the remainders and the final number then corresponds to the product of the two integers. Fig. 5.3 and Fig. 5.4 illustrate the manual multiplication of two integers by only using integer doubling, halving, and addition. The classical multiplication tables for numbers from one to ten are not required.

This concept can be extended to multiplication of two polynomials in $GF(2^r)$ [105]. The algorithm is shown in Figure 5.5. Multiplying or dividing by two can be easily performed using left and right

17	32
8	64
4	128
2	256
1	512

17	32	+
~~8~~	~~64~~	
~~4~~	~~128~~	
~~2~~	~~256~~	
1	512	=
	544	

Figure 5.3.: Russian peasant multiplication. An integer multiplication algorithm using only doubling, halving and addition.

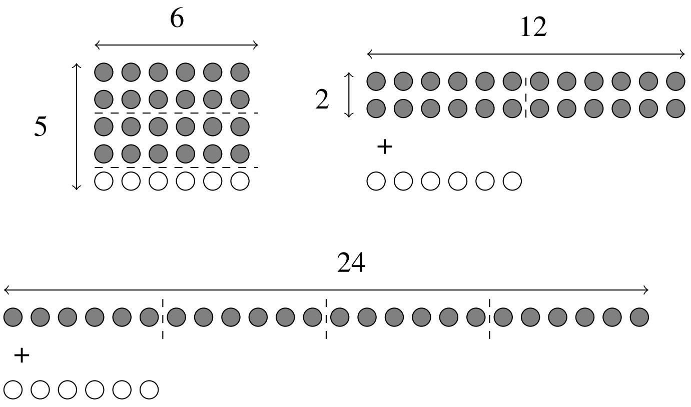


Figure 5.4.: Graphical illustration of a Russian peasant multiplication for $5 \cdot 6$: groups of 6 are reordered by halving and doubling. If the halving step leaves a remainder (empty dots), it has to be summed up with all other remainders to obtain the final result.

shift operators. Halving and doubling is correspondingly implemented with single left (Line 8) and right shifts (Line 12). If the halving step leaves a remainder (which is the case if the lowest order bit is set before the right shift is performed), it has to be added to the final sum. The conditional addition of the remainders is done using a bitwise XOR in Line 5. Since the polynomials represent elements of a finite field, one has to take care of arithmetic operations which result in non-field elements. This is in particular the case if the highest order bit is set before a left shift is performed, that is, if the doubling operation overflows. At this case a modular reduction by the generator polynomial (Line 10) of the Galois field ensures that the result will again be a polynomial of degree less than r. The rationale for the elegant implementation using only a single XOR for the modular reduction[1] requires some additional explanation: The left shift operation corresponds to a multiplication by the polynomial α. The degree of the resulting polynomial $q(\alpha)$ is hence only raised by one and it can thus be represented

[1]One would expect a costly polynomial division with remainder.

```
procedure GFMULT_POLY($a, b$)
    $p \leftarrow 0$
    while $a \neq 0$ AND $b \neq 0$ do
        if $LSB(b) = 1$ then
            $p \leftarrow p \oplus a$
        end if
        $msb\_set \leftarrow MSB(a)$
        $a \leftarrow LeftShiftByOne(a)$
        if $msb\_set = 1$ then
            $a \leftarrow a \oplus PP$
        end if
        $b \leftarrow RightShiftByOne(b)$
    end while
    return $p$
end procedure
```

Figure 5.5.: Modified Russian peasant algorithm for polynomial multiplication in a Galois Field. LSB(x) gives the least significant bit of x, whereas MSB(x) returns the most significant bit of x. PP denotes the primitive polynomial that generates the Galois field.

as the sum of the high-order term plus some remainder $R_q(\alpha) \in GF(2^r)$,

$$q(\alpha) = \alpha^r + R_q(\alpha). \tag{5.2}$$

The primitive polynomial $h(\alpha)$, on the other hand, is also of degree r and can be represented similarly:

$$h(\alpha) = \alpha^r + R_h(\alpha). \tag{5.3}$$

By definition α has been chosen to be a root of h, and therefore

$$0 = \alpha^r + R_h(\alpha), \tag{5.4}$$

or

$$\alpha^r = R_h(\alpha). \tag{5.5}$$

Substitution of Equation 5.5 into Equation 5.2 gives

$$q(\alpha) = R_h(\alpha) + R_q(\alpha). \tag{5.6}$$

This reduces the modular reduction to a simple XOR of both remainders (which have both degree less or equal than $r - 1$).

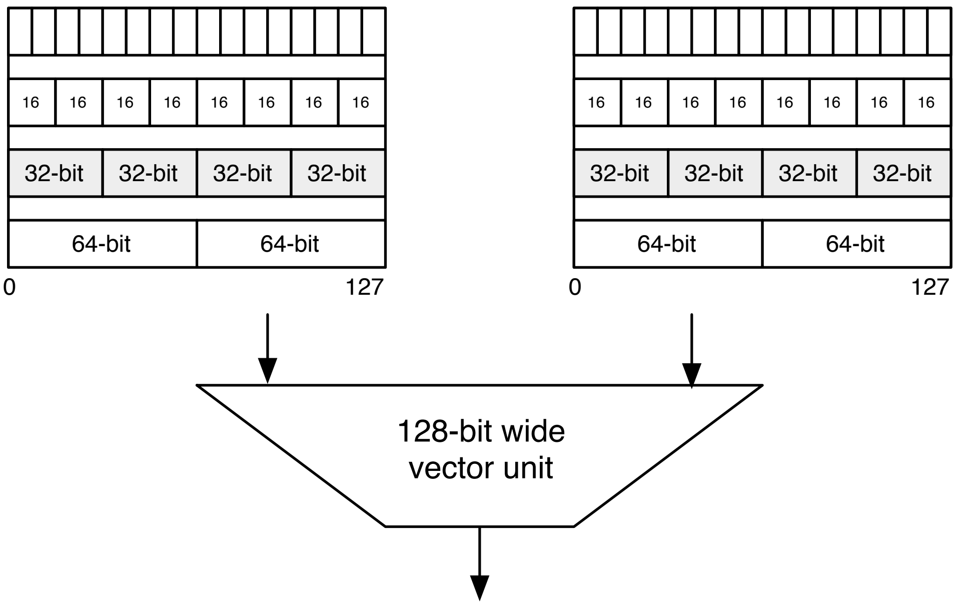


Figure 5.6.: Illustration of a 128-bit wide vector arithmetic logic unit.

5.4. Vectorization

In the instruction sets of modern processors vector extensions to many scalar instructions can be found. Originally designed for data-intensive multimedia applications, these vector instructions can significantly improve performance also for general-purpose computations. Processing follows the *Single Instruction Multiple Data* (SIMD) paradigm, where a single instruction is executed for several data elements at the same time. Figure 5.6 illustrates a 128-bit wide vector arithmetic unit, where the input vectors may contain either two 64-bit double, four 32-bit float, eight 16-bit integer, or sixteen 8-bit integer operands. The vector instructions are performed in parallel on all elements of the vector operands. In addition to the standard arithmetic vector instructions for floating point and integer numbers, logical and shift instructions, instructions for loading and storing of vectors, and comparing of vector components, several specialized instructions are available. However, not all vector instructions support all input vector types. The SIMD model works well, if in fact the same instruction sequence is executed for all vector elements. There exist no direct mechanisms for flow control inside the vectors. However, conditional execution for different elements within the same vector can be emulated by using masks. This effectively leads back to a serialization of the execution. In the worst case of divergent execution paths for all vector elements no performance gain can be achieved, on the contrary, the masking generates an additional overhead. Since only XOR, AND and shift operations are used, the multiplication algorithm can be vectorized with the integer part of the *Streaming SIMD Extensions* (SSE) [101]. The most direct way to use SIMD instructions is to inline assembly instructions into high-level language source code. However, this task is complex and error-prone. Many compilers offer API extensions for the different vector instruction sets that allow for an easier utilization of SIMD instructions. These extensions are called *intrinsics* or *intrinsic functions*. They use a syntax of C function calls and variables instead of assembly language and registers [106]. The presented implementation uses intrinsics of the SSE vector instruction set. The core multiplication code uses only integer SIMD instructions available in SSE version 2, but some instructions from SSE version 3 and 4 (shuffle, min/max, extract) can be beneficial for reducing the number of loop cycles. The full implementation therefore mixes intrinsics of various SSE versions with inlined scalar assembly. The selection of an appropriate size of the Galois field strongly depends on the available SIMD instructions for the corresponding element size. As a result the field $GF(2^{16})$ was chosen, such that eight 16-bit field elements can be processed in parallel inside the 128-bit wide SSE registers. Since the check of the break condition after every loop cycle imposes a severe branch overhead, it is more efficient to simply execute the while loop 16 times (which is the worst case for a minimal value of a and a maximum value of b). For the vectorization approach, it would be beneficial to choose an even smaller field size (for instance $GF(2^8)$). This way the maximum number of loop cycles could be further reduced. However, current SSE versions do not support the required shift instructions for less than 16-bit wide integers and emulation of the missing shift instruction is not economical. Figure 5.7 shows an illustration of the vectorized implementation of the inner loop of the polynomial multiplication algorithm (lines 4 to 12 in the scalar algorithm). The notation is as follows: The actual data elements in the vector registers are depicted in the white boxes. SSE instructions are represented by the hatched boxes and take either one or two vector registers as operands. The result of the instruction is stored in another register or directly fed into another instruction (both indicated by the solid dashes). Register contents that are used as masks are colored gray. The dashed arrows indicate the subsequent reuse of the contents of the vector register. In the boxes to the right an explanation of the vector instruction sequence is presented. With the exception of the mask operations, the individual steps match the ones in the scalar implementation in Figure 5.5. For a full encoding and decoding operation (basically a sequence of multiplications and additions over the finite field) the vectorized implementation can be easily chained.

Before putting the code into application, it is worthwhile to examine whether the overall number of instructions can be reduced. Since the result of the multiplication is determined as soon as either of the factors is zero, the first approach is to determine the number of loop cycles dynamically and in advance. Since one factor is shifted right and the other factor is shifted left, one can either determine the position of the highest order bit or of the lowest order bit to find after how many shifts the argument reaches the value of zero. For the scalar version this can be achieved with the scalar *bsr (Bit Scan Reverse)* or *bsf (Bit Scan Forward)* instructions. Instead of applying this instruction sequentially to every element in the vector, the process can be shortened using vector instructions. Figure 5.8 illustrates the vectorized search for the maximum loop cycles $L_{\max}$ for two vectors a_i and b_i where $i \in \{0,\ldots,7\}$: Due to the commutativity of the multiplication it is possible to interchange multiplier and multiplicand, such that the vector b_i holds the smaller of both elements and the vector a_i holds the larger. Now the number of loops that have to be calculated depends on the highest set bit of any $b_i = b_i^{15} b_i^{14} \ldots b_i^{0}$:

$$L_{\max} = \max_{i\in\{0,\ldots,7\}} \Big(\max_{m\in\{0,\ldots,15\}} (m \cdot b_i^m) \Big) \tag{5.7}$$

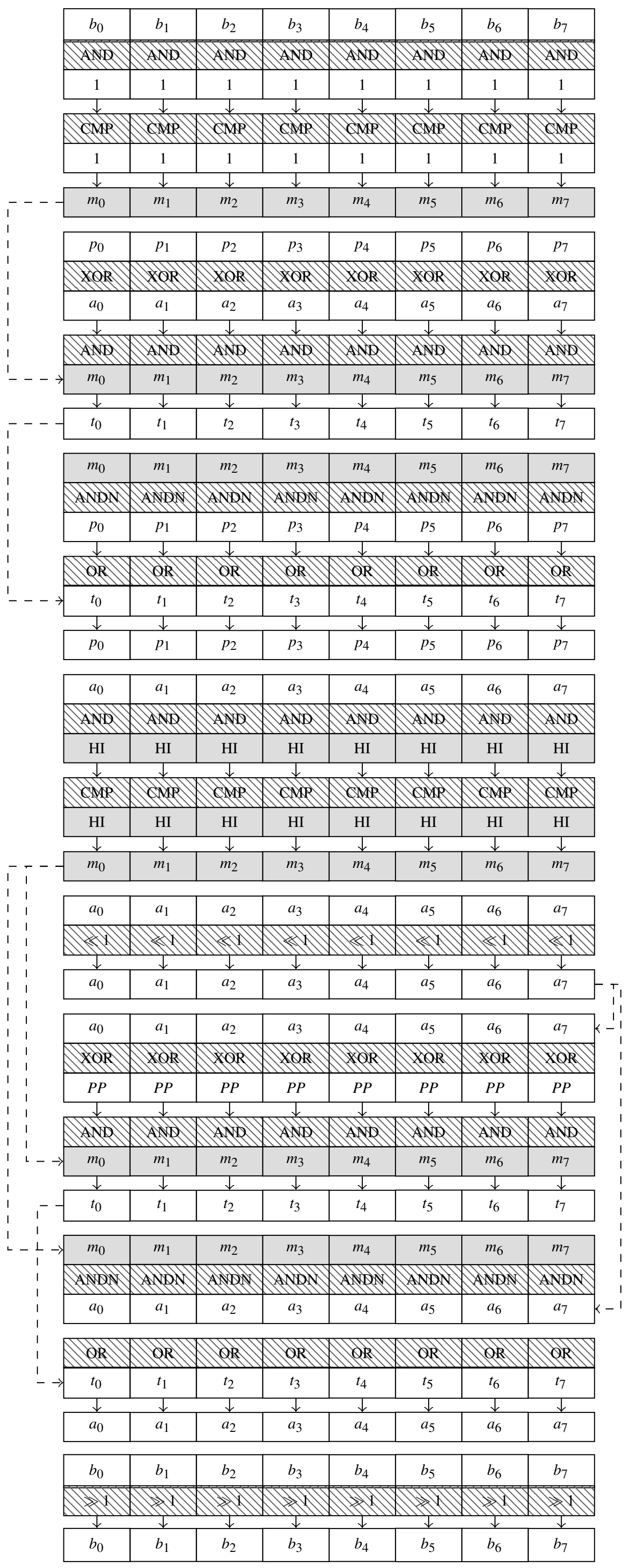
Table 5.2.: Typical values of $\overline{MSSB}_{16}$ for different types of data. At least 1GiB of the stated data type has been examined.

data format	$\overline{MSSB}$	Zeroes (%)
bz2	13.99	0.03
gz	14.00	0.02
jpeg	14.03	0.03
mp3	13.93	0.4
Mach-O 64-bit executable	12.49	29.57
man pages	13.81	0.01

To find this bit, all elements in b_i are subsequently folded into each other by using the OR and shuffle vector instructions. At last, the position of the *most significant set bit (MSSB)* is determined with the scalar bsr instruction. With this mechanism the number of required loop cycles can be dynamically reduced if at least one of the factors stored in the individual vector components is sufficiently small (in terms of its decimal representation). How can this help for the general coding and decoding task? The mean position of the most significant set bit in all w-bit words excluding the zero word is calculated as

$$\overline{MSSB}_w = \frac{1}{2^w - 1} \sum_{i=0}^{w-1} i \cdot 2^i. \tag{5.8}$$

For uniformly distributed 16-bit words (without the zero word) the mean MSSB position is therefore equal to $\overline{MSSB}_{16} \approx 14$. In general, many file storage formats display a high entropy (for instance through compression or encryption). A quick examination of standard file formats confirms this assumption (Table 5.2). As will be shown later it is thus more promising to focus on the generator matrix elements, instead of relying on adequate MSSB distributions of the actual data.


b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
AND
1
CMP
m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
XOR
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7
ANDN
OR
HI
$\ll 1$
PP
$\gg 1$

Check for the least significant bit of b_i and create a mask m_i.

$$m_i \leftarrow \begin{cases} \texttt{0x0000} \text{ if } LSB(b_i) = 0 \\ \texttt{0xFFFF} \text{ if } LSB(b_i) = 1 \end{cases}$$

Calculate $p_i \quad \oplus \quad a_i$ and use m_i to mask out all results t_i for all i with $LSB(b_i) = 0$:

$$t_i \leftarrow (p_i \oplus a_i) \cdot m_i.$$

Conserve the masked out values of p_i and fill in the updated values:

$$p_i \leftarrow (\overline{m_i} \cdot p_i) + t_i.$$

Check for the most significant bit of a_i using a HighBit mask (0x8000) and create a mask m_i.

$$m_i \leftarrow \begin{cases} \texttt{0x0000} \text{ if } MSB(a_i) = 0 \\ \texttt{0xFFFF} \text{ if } MSB(a_i) = 1 \end{cases}$$

Left-shift a_i by one:

$$a_i \leftarrow a_i \ll 1.$$

Calculate $(a_i \oplus PP)$ with PP being the primitive polynomial of the used Galois Field and use m_i to mask out all results t_i for all i with $MSB(a_i) = 0$:

$$t_i \leftarrow (a_i \oplus PP) \cdot m_i.$$

Conserve the masked out values of a_i and fill in the updated values:

$$a_i \leftarrow (\overline{m_i} \cdot a_i) + t_i.$$

Right-shift b_i by one:

$$b_i \leftarrow b_i \gg 1.$$

Figure 5.7.: SSE adaption of the loop body of algorithm 5.5 using 128-bit wide XMM registers.

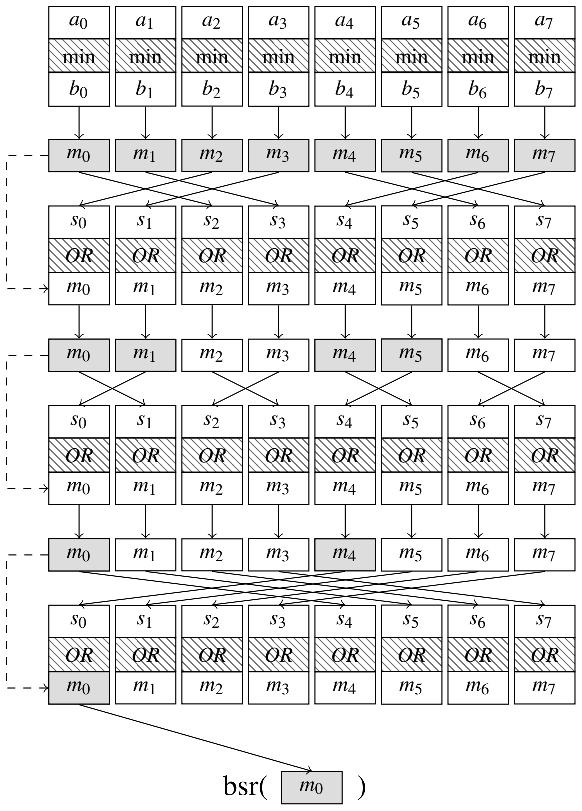

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
min
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7
OR
bsr(m_0)

Figure 5.8.: Illustration of the instruction sequence to find the required number if loop cycles. White boxes represent the values in the vector registers, the gray boxes indicate which remaining elements need to be ORed and the hatched boxes represent the bitwise SIMD instructions. The arrows show the order of the SIMD shuffle instructions. Dashed arrows indicate the subsequent reuse of the contents of the vector register.

5.5. GPU as co-processor

In addition to the SIMD units of commodity processors, the use of graphics processing units for non-graphics algorithms has evolved rapidly (graphics cards that are used in this context are often referred to GPGPUs). Originally, GPUs could only be used for general purpose computation by exploiting pure graphics APIs such as OpenGL and DirectX with severe constraints on flexibility and portability [107]. With the introduction of the Compute Unified Device Architecture (CUDA) [108] by NIVIDA in 2006, a comprehensive programing model together with development tools and compilers became available. In 2009 the Open Computing Language (OpenCL) [109] was standardized, defining a framework for executing programs across heterogeneous systems containing CPUs, GPGPUs and other kinds of processors (e.g. DSPs and FPGAs). Today, GPGPUs display a much higher performance per price ratio than commodity processors, with typically hundreds or thousands of execution units. Many high-performance computing systems are therefore cost-effectively complemented with GPGPUs. In general, applications with a high degree of data level parallelism can achieve high performance speedups on these particular many-core architectures. However, the cost of the data transfers between host and GPU, the memory and cache hierarchies inside the GPU and their limitations, as well as the hardware architecture itself, and finally the execution model have to be well understood in order to develop efficient parallel algorithms. Figure 5.9 depicts the architecture of the NVIDA Kepler Streaming Multiprocessor (SM). Every multiprocessor consists of 192 CUDA cores (indicated in the figure with C), each equipped with a fully pipelined integer and floating point unit [110]. Both single and double precision floating point arithmetic are supported. Each SM contains 32 load/store units (LD/ST) and the same number of special function unit (SFU) responsible for execution of transcendental instructions, such as sine, cosine, as well as reciprocal and square root functions. The Streaming Multiprocessors are the basic building blocks of a GPU. Depending on the intended performance, multiple of these units form a GPU, together with the required amount of on-board memory and further supporting graphics hardware (for instance the texture units). From a developer's perspective the GPU appears as a highly multi-threaded compute device which executes threads in a single-instruction, multiple-thread fashion (SIMT) [111]. Threads are mapped to the scalar cores and execute the same sequence of instructions, the so called *kernel*, independently. Kernels are designed in the C/C++ language (with minimal extensions) and compiled into the instruction set architecture of the GPU. A kernel is invoked with the help of a runtime system which transfers it and the associated data to the GPU where the kernel is then executed as a set of parallel threads. The programmer organizes these threads beforehand into blocks and grids of thread blocks, which are then mapped onto the individual multiprocessors. Every thread within a block represents an instance of the kernel. The threads are uniquely identified by a combination of an ID within the thread block and the ID of the block itself. Threads within the same block can cooperate through synchronization and exchange data through a small, but fast, shared memory. Data are read from and written to the (onboard) global memory, which is the only way to exchange data between thread blocks. In principle GPUs also suffer from the high access latencies of their onboard memory, but they pursue various strategies to overcome this limitation and to enable a massively parallel throughput: GPGPUs typically use large register files which are shared between the active threads

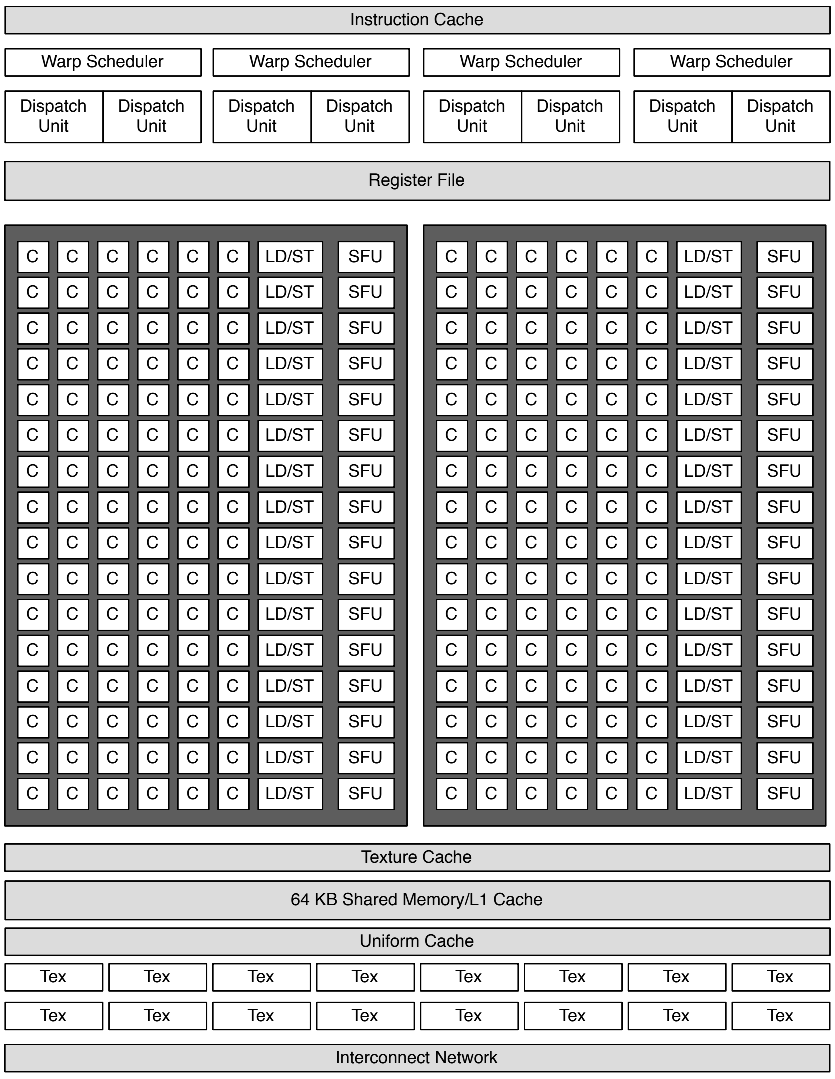


Figure 5.9.: Architecture of the Kepler Streaming Multiprocessor (SM) [110].

and additionally offer explicitly managed caches to keep data local to the computation. Ideally, the execution units are overcommitted to a high degree. In conjunction with the ability to perform very fast thread context switches, this enables an effective hiding of memory access latencies. A prerequisite is the abundance of arithmetic operations that can be quickly switched to, if a previous load or store instruction incurred a long access latency. In principle, the polynomial multiplication algorithm in Figure 5.5 is therefore ideal for the use with a GPGPU accelerator. Unlike the lookup based algorithm, it uses only arithmetic operations which help hiding the necessary latencies of the load operations of the factors as well as the store operation of the result.

For the use as an accelerator for storage applications the performance of the data transfer to the GPU and back to the host is crucial. The bandwidth is limited by the maximum bandwidth of the PCIe interface which strongly depends on the maximum packet payload size that is supported by the PCIe controller. With typical efficiencies between 80% and 85% [112] the theoretically achievable PCIe 2.0 bidirectional bandwidth of 8 GB/s is already significantly reduced. Fig. 5.10 shows the PCIe bandwidth of a NVIDIA GTX680 GPU depending on the size of the transferred block using page-locked memory.

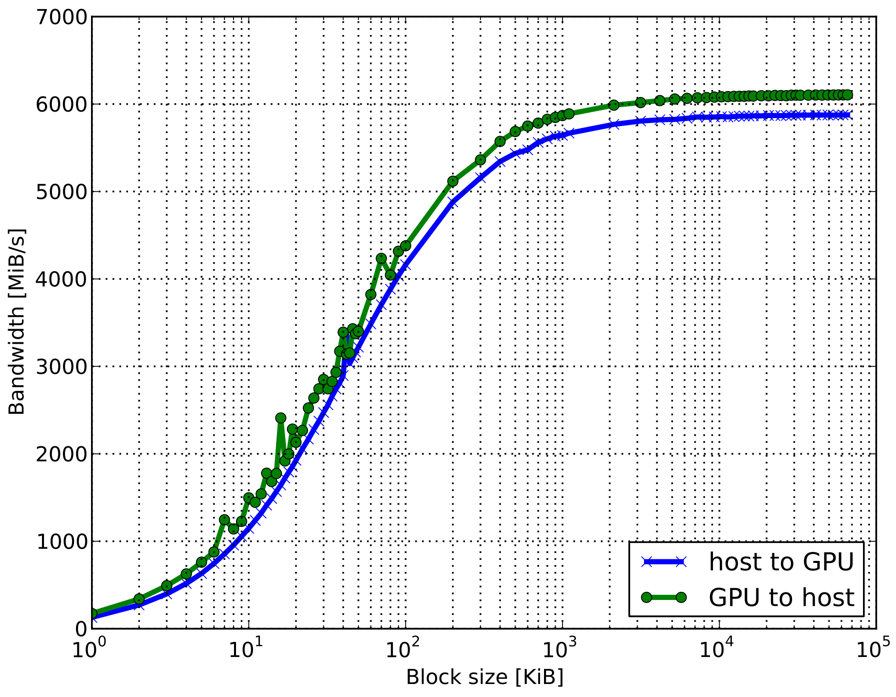


Figure 5.10.: PCI express bandwidth in both directions (host to GPU and GPU to host) using page-locked memory.

This bandwidth sets the upper theoretical limit for differential encoding, for full encoding or decoding significantly more data has to be transferred from the host to the GPU than back and the overall performance is reduced accordingly. The next generation PCI Express 3.0 standard effectively delivers twice the bandwidth compared to the previous version, achieved by increasing transfer rate and reducing the overhead for symbol encoding at the same time. To hide PCIe latencies kernel execution and transfer in CUDA can be overlapped by defining multiple streams. Figure 5.11 shows the asynchronous execution of a multiplication kernel with four overlapping streams. In this case two third of the communication can be hidden behind the kernel execution.

A short kernel performing the scalar polynomial multiplication is shown in Listing 5.1. In this case every threads performs exactly one multiplication of two factors. The factors *a* and *b* are stored as consecutive buffers in the GPU global memory. Additionally, one result buffer has been set up. Pointers to these buffers are passed as parameters to the kernel. Since possibly thousands of instances of the kernel are executed, the first step involves the identification of the correct input data. Therefore, the thread and block IDs are used to create a globally unique offset. This offset determines the address of the input factors and the address within the result buffer for the multiplication results. Otherwise the kernel resembles the scalar implementation of the multiplication. Since a single piece of code is executed on several multiprocessor at the same time, it is difficult to reduce the number of loop cycles in reasonable way. Inspecting the factors for the MSSB within an entire thread block and communicating the result would require a significant amount of synchronization. Simply executing the full number of loops cycles is therefore much more performant. If a general limit for the MSSB of one of the factors can be specified, the thread synchronization can be avoided altogether.

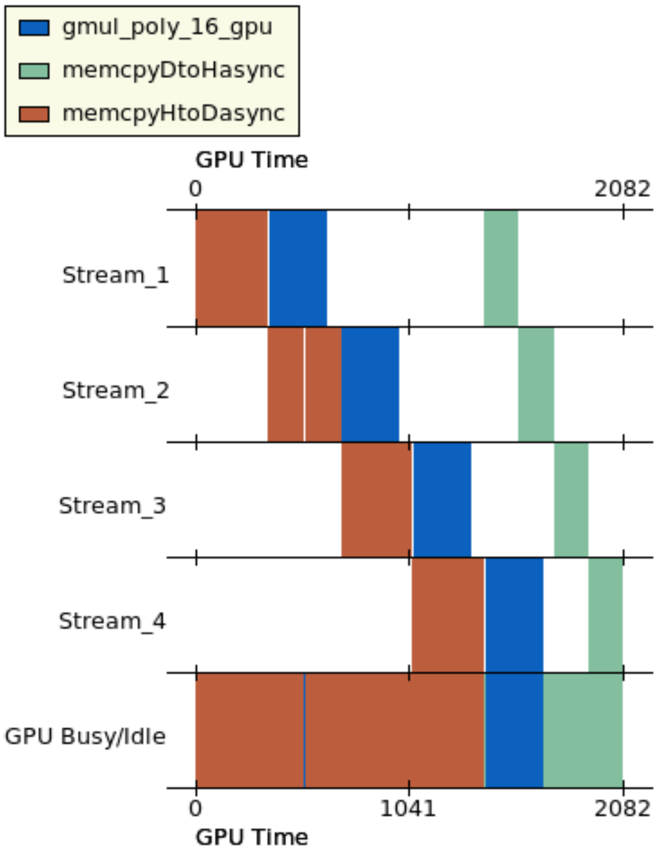


Figure 5.11.: Asynchronous execution of a multiplication kernel with four streams. The plot was created with the CUDA visual profiler.

Listing 5.1: CUDA multiplication kernel

```
__global__ void gmul_poly_16_gpu(ushort *a, ushort *b, ushort *res)
{
  unsigned int prim_poly_16 = 0210013;
  int tx   = threadIdx.x;
  int bx   = blockIdx.x;
  int gx   = tx + blockDim.x * bx;

  ushort counter;
  ushort p=0;
  ushort hi_bit_set;

  #pragma unroll

  for(counter = 0; counter < 16; counter++) {
      if((b[gx] & 1) == 1)
         p ^= a[gx];
      hi_bit_set = (a[gx] & 0x8000);
      a[gx] <<= 1;
      if(hi_bit_set == 0x8000)
         a[gx] ^= prim_poly_16;
      b[gx] >>= 1;
    }

  res[gx]=p;
}
```

5.6. Low MSSB generator matrices

The previous sections have shown that the polynomial multiplication can be significantly accelerated if at least one of the factors displays a low MSSB. Since no assumptions about the data that is to be encoded can be made, the focus shifts to the generator matrix of the code. A variant of the ubiquitous Reed-Solomon codes is generated by a matrix which is obtained by starting with a Vandermonde matrix

$$V = (v_{i,j}) = i^j, \tag{5.9}$$

and by applying a series of elementary transformations to it until the matrix has a systematic form. In the following, n denotes the number of the message symbols and k represents the number of the check symbols. The resulting matrix is called *information dispersal matrix* and its specific form can be different for different combinations of n and k. An illustration of the MSSB distribution for a Vandermonde matrix and the derived information dispersal matrix is shown in Figure 5.12. Larger boxes reflect a higher MSSB of the matrix element. Empty spaces indicate that the matrix element is zero (no bit is set). After the transformation the first n rows become the identity matrix and the actual coding part contains elements of various size. In order to benefit from the MSSB dependent multiplication

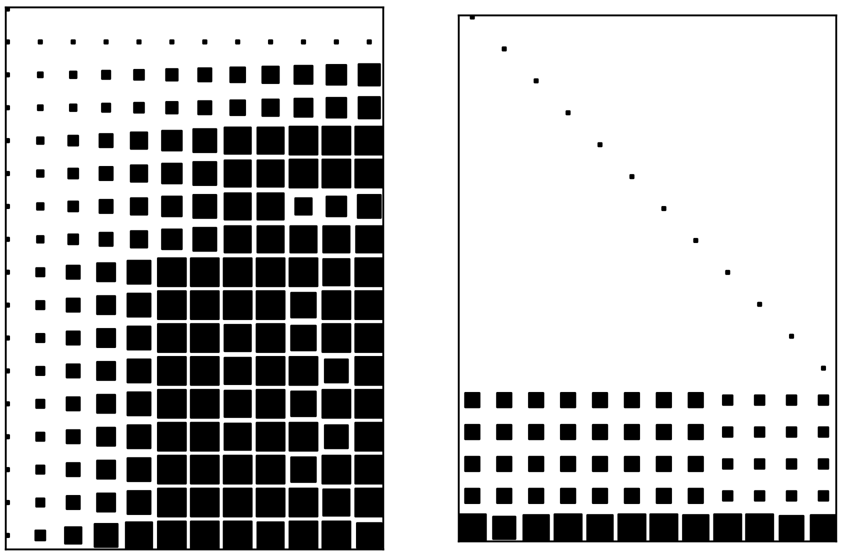


Figure 5.12.: Visualization of the MSSB distribution of the Vandermonde matrix (left) and the corresponding information dispersal matrix (right). Larger sized boxes indicate a higher MSSB.

algorithm, it is important to use generator matrices that display a low average MSSB in the systematic part. At first, it is of interest how the MSSB distribution of the information dispersal matrix varies depending on the underlying Vandermonde matrix. Figure 5.13 shows the MSSB distribution of the non-identity matrix elements for different values of n and k in the generator Matrix for a Vandermonde-based Reed-Solomon code. The code was constructed for the Galois field generated by the polynomial $h(\alpha) = 1 + \alpha^4 + \alpha^{13} + \alpha^{15} + \alpha^{16}$. The MSSB distribution of the matrix elements shows a remarkable structure. The figure shows the maximum, the minimum, as well as the average MSSB. Certain combinations of n and k lead to matrices with particular small elements. With growing k the average MSSB also increases and the minima are not as distinct anymore.

To answer the question whether the chosen primitive polynomial for the underlying Galois field can also have an influence on the distribution, one has to examine all primitive polynomials. It can be shown [84] that the number of irreducible polynomials of degree m (denoted by I_m) is related to the order of the field q^r by

$$q^r = \sum_{m|r} m \cdot I_m, \tag{5.10}$$

Table 5.3.: Number of irreducible (I_r) and primitive (P_r) polynomials for the Galois Fields $GF(2^r)$, $r \leq 16$. See also [114]

r	Order	I_r	P_r
1	2	1	1
2	4	1	1
3	8	2	2
4	16	3	2
5	32	6	6
6	64	9	6
7	128	18	18
8	256	30	16
9	512	56	48
10	1024	99	60
11	2048	186	176
12	4096	335	144
13	8192	630	630
14	16384	1161	756
15	32768	2182	1800
16	65536	4080	2048

where $m|r$ denotes all m which divide r (including 1 and r). Equation (5.10) can be used to iteratively determine the number of irreducible polynomials. Furthermore, the number of primitive polynomials is given by

$$P_r = \frac{\phi(q^r - 1)}{r}, \tag{5.11}$$

where ϕ is Euler's totient function [113]. Table 5.3 shows I_r and the corresponding number of primitive polynomials P_r for the fields with $q = 2$ and $1 \leq r \leq 16$.

A full list of the primitive polynomials (in decimal representation) of $GF(2^{16})$ is shown for reference in Appendix A.1.

Figure 5.14 shows the average MSSB of the systematic part of the information dispersal matrix for all primitive polynomials in $GF(2^{16})$ with a fixed value of $n = 20$ and variable k. A decimal representation of the primitive polynomial is used for the abscissa. The plot shows that the average MSSB can be slightly lower for selected polynomials. However, this selection is not consistent with changing k. Furthermore, the window of the average MSSB positions is relatively narrow (which means, that the potential gain for the multiplication through selection of an appropriate primitive polynomial is relatively low). However, this plot is only a snapshot for a fixed value of n. To confirm that this observation is also true for a broad range of n and k, Figure 5.15 combines both perspectives into a single plot. It shows bands of average MSSB positions for all 2048 primitive polynomials in $GF(2^{16})$ for different

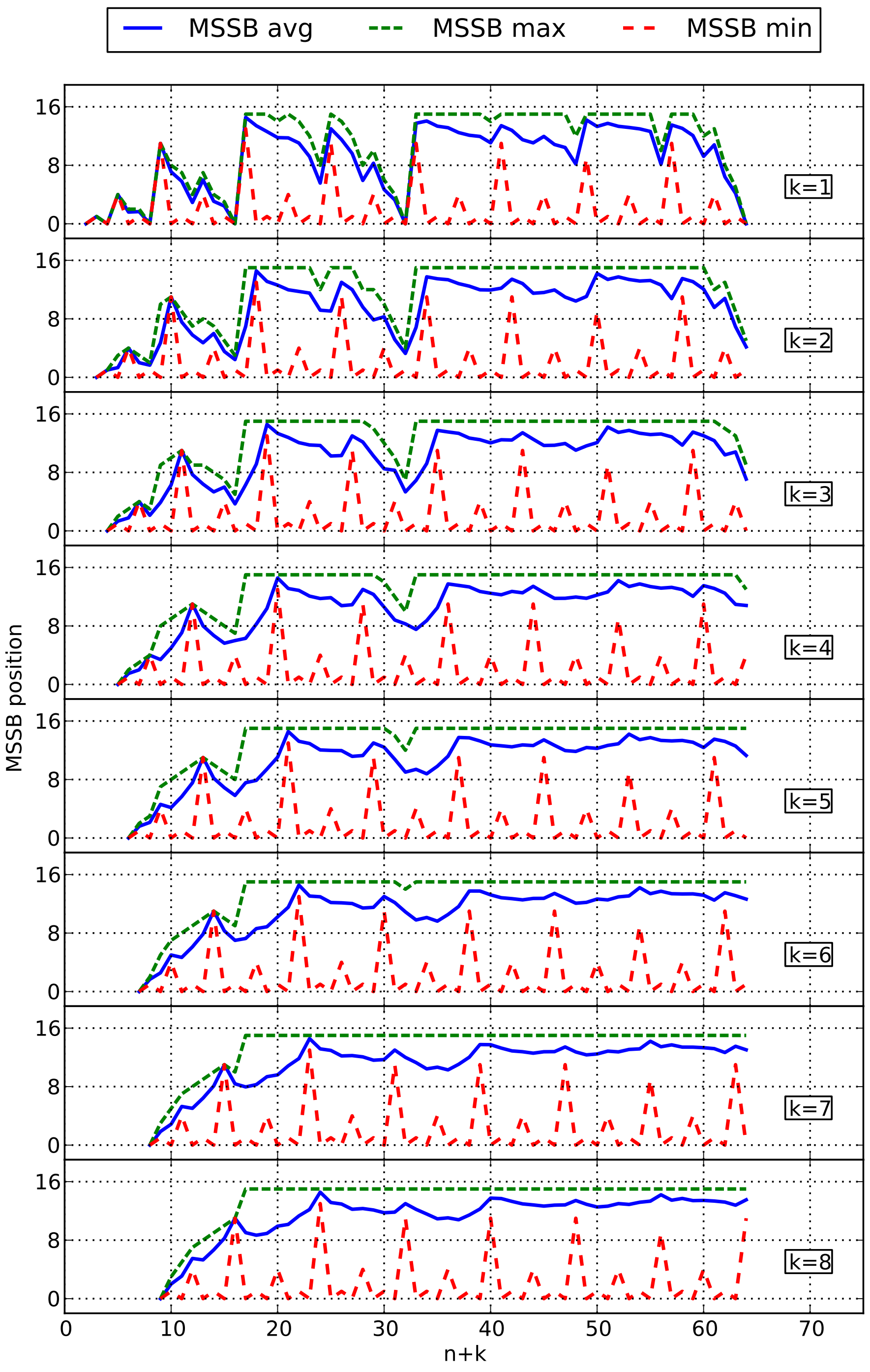


Figure 5.13.: MSSB distribution in the non-identity elements of a Vandermonde-based Reed-Solomon generator matrix for given n and k in $GF(2^{16})$. The field was constructed with the primitive polynomial $h(\alpha) = 1 + \alpha^4 + \alpha^{13} + \alpha^{15} + \alpha^{16}$.

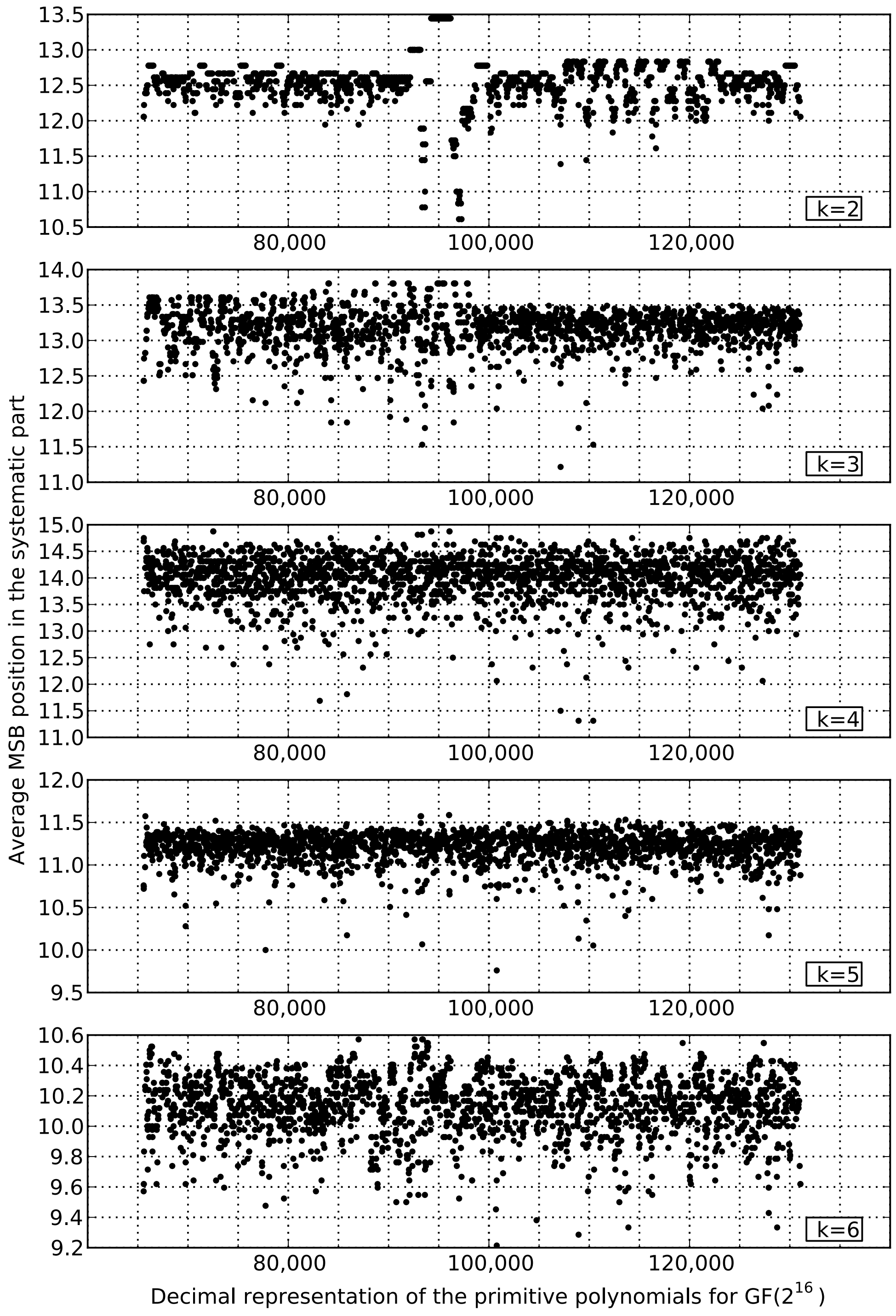


Figure 5.14.: Average MSSB in the systematic part of the Vandermonde-based generator matrices for all primitive polynomials in $GF(2^{16})$, $n = 20$ and variable k.

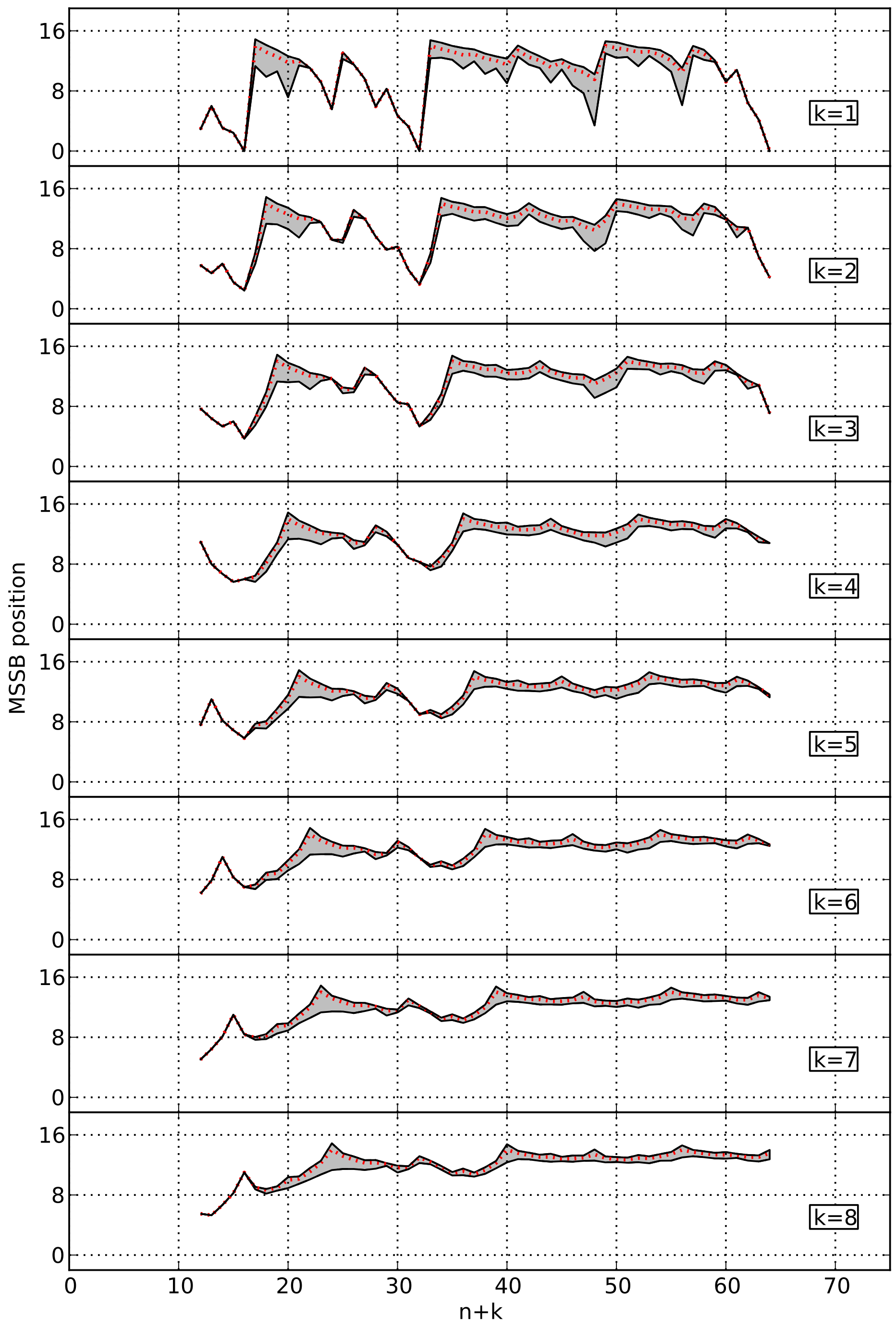


Figure 5.15.: Range of average MSSB positions of the Vandermonde-based generator matrices for all 2048 primitive generator polynomials of $GF(2^{16})$. The dotted lines indicate the mean values over all primitive polynomials.

values of $n+k$ and k [2]. The dotted line inside each band indicates the average over all primitive polynomials at this point. Especially for larger k and larger $n+k$ the band are relatively narrow and centered around high values for the MSSB. With this plot it becomes apparent that the Vandermonde-based information dispersal matrices are not the best choice in order to take full advantage of the polynomial multiplication.

5.7. Generating low average MSSB matrices

An exhaustive enumeration of all matrices that could be qualified as generators for a Reed-Solomon-type linear block code is more than impractical due to the combinatorial explosion. In the following paragraph two heuristic methods for finding suitable matrices for polynomial multiplication are presented:

- A known generator matrix is transformed in a certain way in order to reduce the average MSSB.
- A matrix is randomly created and it is examined whether it can be used as generator. If so, the properties of this matrix serve as upper limit for a next round of random sampling.

The reason for beginning with a Vandermonde matrix of the form $(v_{i,j}) = i^j$ is the following: Since all i are naturally pairwise distinct, the determinant for a square matrix of this form is guaranteed to vanish. This property is preserved under elementary matrix transformations. Extending the square matrix by k additional rows (which are also pairwise distinct) gives a rectangular matrix which cannot be inverted. However, when k arbitrary rows are removed, the matrix is again a square Vandermonde matrix. Since all rows are pairwise distinct, there are no restrictions which k rows can be deleted. Therefore, it is an ideal foundation for the generator matrix of a linear systematic block code.

The first approach for generating a more suitable generator matrix is to start with the information dispersal matrix derived from a Vandermonde matrix. An element in the systematic part is chosen and it is manipulated in a suitable way to reduce the average MSSB. The key question is then, whether the new matrix is still invertible if any k rows are removed. Therefore, one has to check all $\binom{n+k}{k}$ square matrices that are obtained by deleting k rows from the modified rectangular matrix. One way to check for invertibility is to examine the determinant of the matrix, which can be efficiently computed using LU decomposition [115]: If the matrix A can be factorized into two matrices

$$A = L \cdot U, \tag{5.12}$$

where L is in lower triangular form[3] and U is in upper triangular form, it can be shown that the determinant is equivalent to the product of the diagonal elements of U:

$$det(A) = \prod_j u_{jj}. \tag{5.13}$$

[2]A version with for even higher values of $n+k$ is shown in the Appendix in Figure A.1

[3] It can be shown that L can be always chosen with all diagonal elements equal to one.

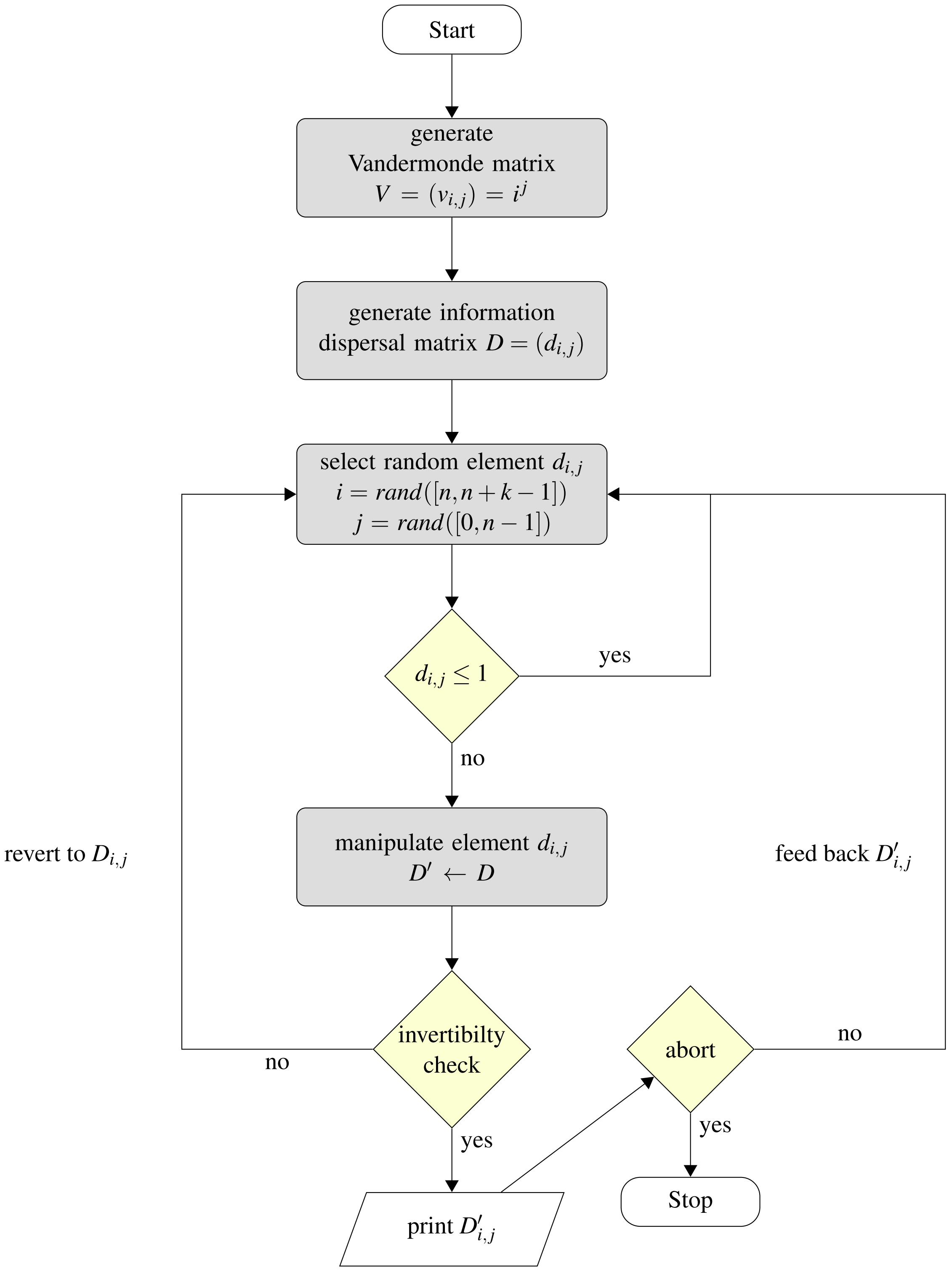

Start
generate Vandermonde matrix $V = (v_{i,j}) = i^j$
generate information dispersal matrix $D = (d_{i,j})$
select random element $d_{i,j}$ $i = rand([n, n+k-1])$ $j = rand([0, n-1])$
$d_{i,j} \leq 1$
yes
no
manipulate element $d_{i,j}$ $D' \leftarrow D$
revert to $D_{i,j}$
feed back $D'_{i,j}$
invertibilty check
no
yes
abort
no
yes
print $D'_{i,j}$
Stop

Figure 5.16.: Flowchart of the Vandermonde-based matrix finding algorithm.

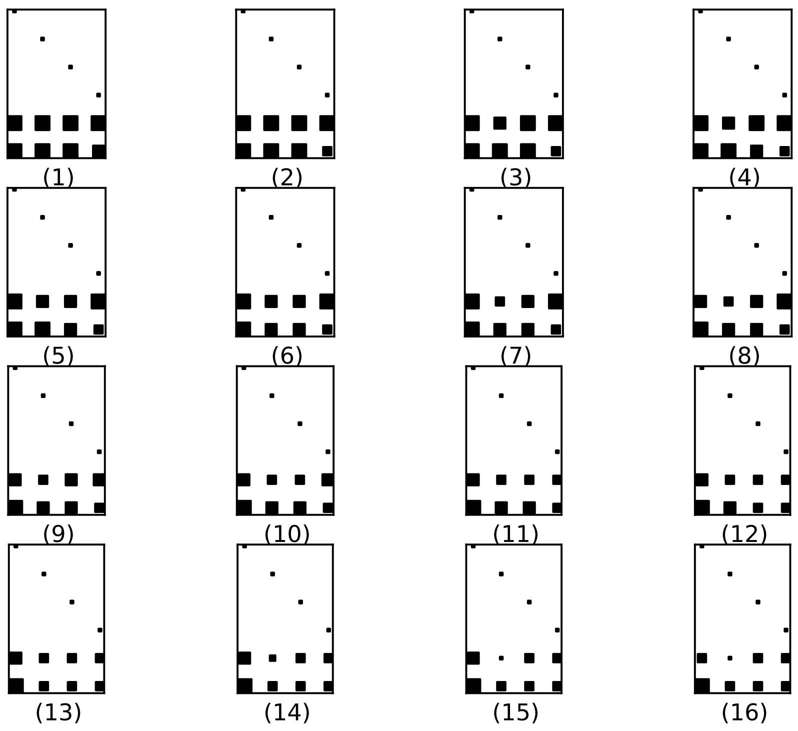


Figure 5.17.: Sequence of generator matrices. Larger sized boxes indicate a higher MSSB.

With these techniques an iterative algorithm to reduce the average MSSB in the systematic part of a code generating matrix can be formulated. A flowchart of the algorithm is shown in Figure 5.16. After generation of the Vandermonde matrix and the corresponding information dispersal matrix, a random element of the systematic part is selected. If the MSSB of this element is already at bit position 0, another element is selected. The element is then logically right-shifted, which reduces the MSSB position by one. The invertibility of the resulting matrix is then examined. If the new matrix is not invertible, all changes are reverted and another element is selected for manipulation. If invertibility has been preserved, the matrix is recorded and then used for the next iteration of the algorithm. This algorithm has the advantage that it immediately produces a valid generator matrix. However, due to the random selection of elements the matrices tend be not well balanced. Furthermore, the decision when to abort the search is difficult. Certain sequences result in matrices with a higher average MSSB, which cannot be reduced any further. It is therefore necessary to perform many rounds of this algorithm and then select the best matrix. Figure 5.17 illustrates the MSSB distribution of one sequence of generator matrices that are found by this algorithm. Only valid matrices are shown, between two steps several steps of manipulating and reverting to an earlier matrix are hidden.

Another approach is to use randomly generated elements in the systematic part. This results in consistently balanced and low average MSSB matrices. The algorithm is depicted in Figure 5.18. Two parameters are used to create the random elements: A maximum value for the random numbers (*maxelem*), and a limit for the average MSSB position (*lim*). A matrix is created randomly with all elements between one and the *maxelem*. If the average MSSB position of all elements in the systematic part is below *lim*, the matrix is evaluated for its invertibility. If any square matrix is not invertible, another random matrix is created with the old parameters. However, if the matrix is a valid generator matrix, it is recorded and the parameters are updated with the values of the found matrix. Then the

$$m = \lfloor \frac{k}{2} \rfloor (\lfloor (log_2(maxelem) \rfloor - 1) \tag{5.14}$$

largest elements in the systematic part are right-shifted and the invertibility check is performed again. This additional manipulation step accelerates the algorithm dramatically in its starting phase (if initially the parameters are set to their upper limits). The number of element manipulations is dynamically reduced depending on the current maximum value for the elements[4]. Since with every step the range of allowed values for the matrix elements is reduced, the probability increases, that concurrent manipulation creates rows that are linearly dependent.

Figure 5.20 illustrates the MSSB distribution of one sequence of generator matrices that are found by the randomized algorithm.

[4]The value of m can be tuned better for particularly small or large matrices. The factor given in Equation (5.14) is suitable for ranges of n and k that were used to generate the matrices in Appendix A.2.

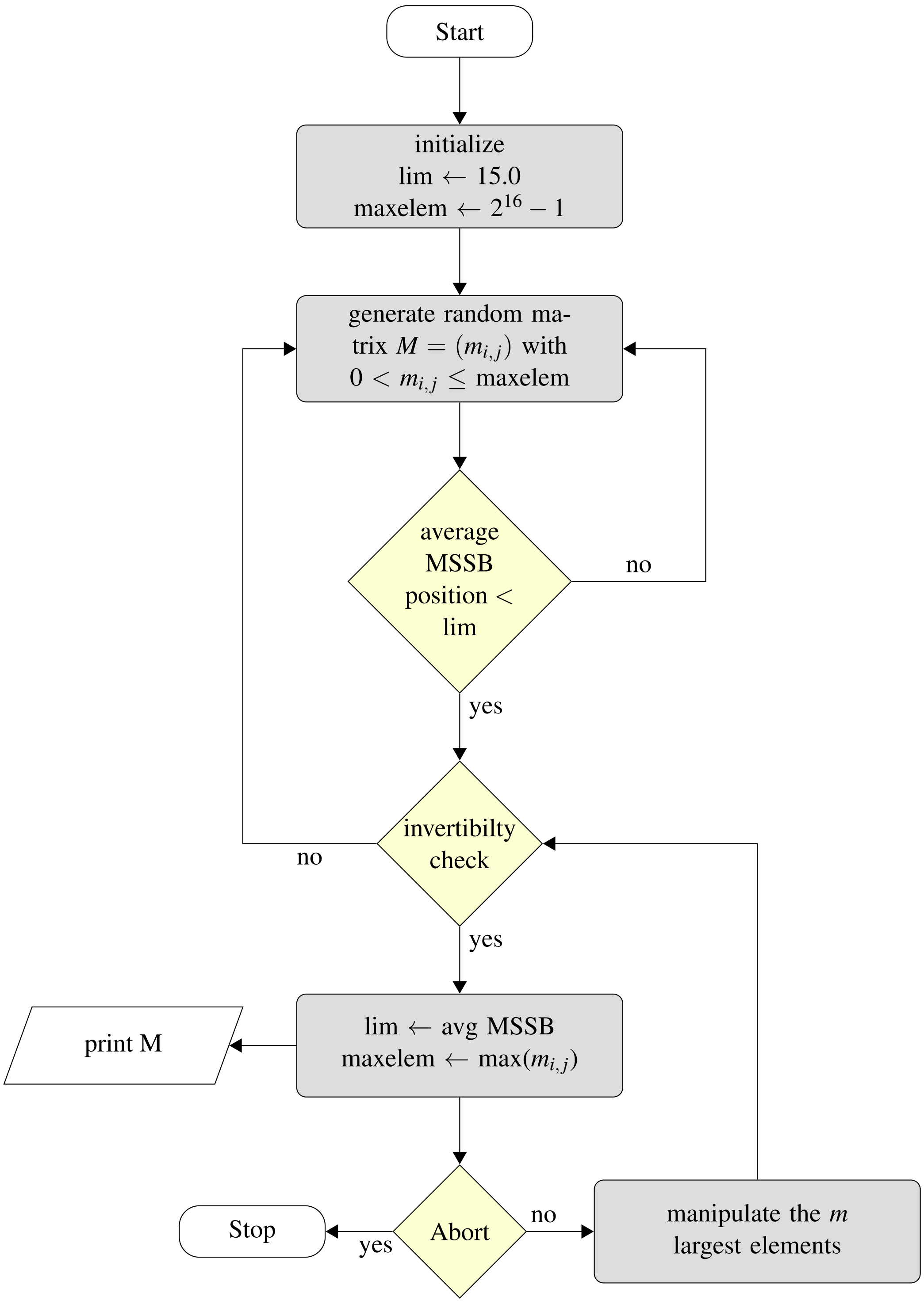


Figure 5.18.: Flowchart of the randomized matrix finding algorithm.

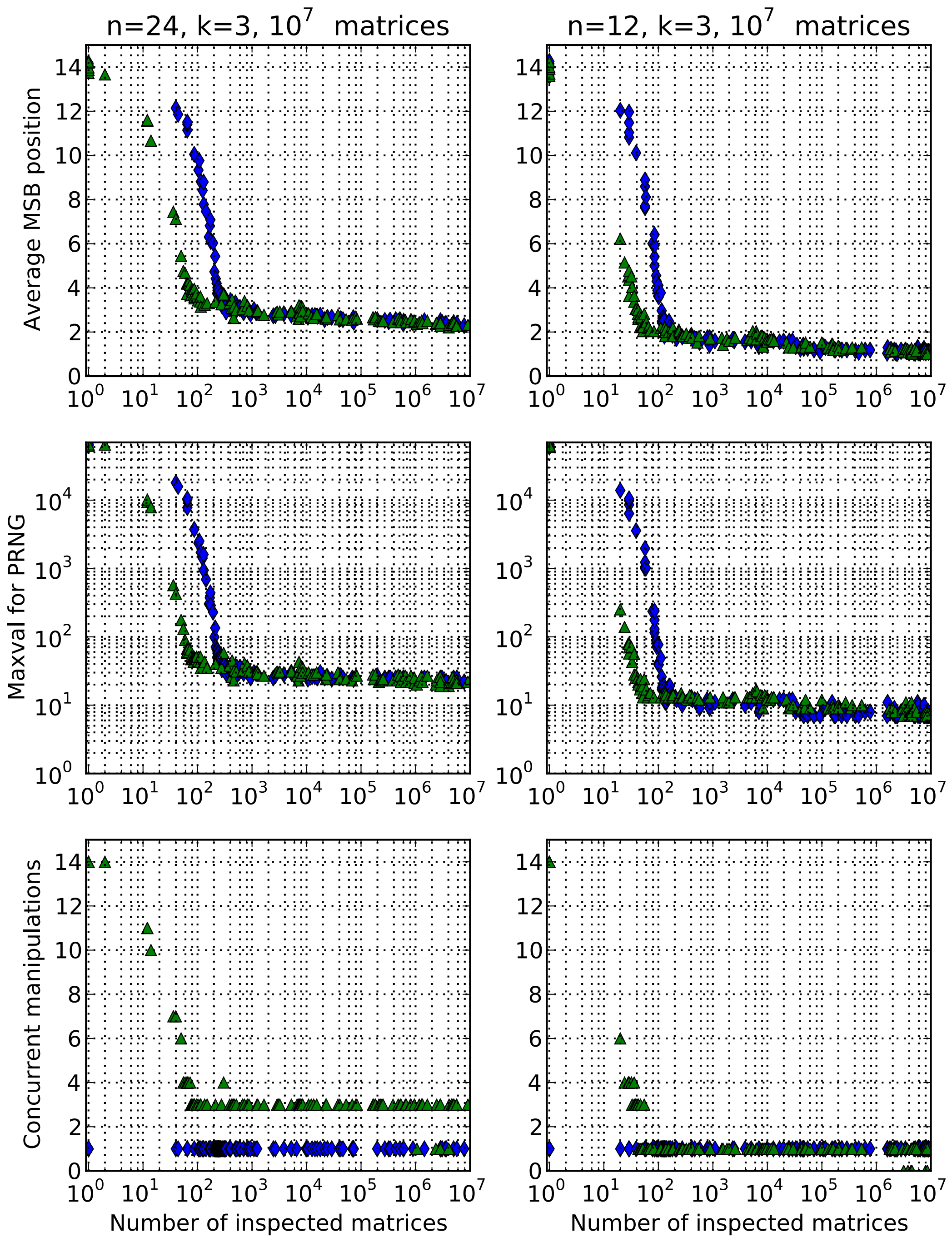


Figure 5.19.: Comparison of matrix finding with element manipulation (triangles) and without (rhombi) for two different values of *n* and *k*. The last row shows the number of allowed element manipulations in the corresponding time step.

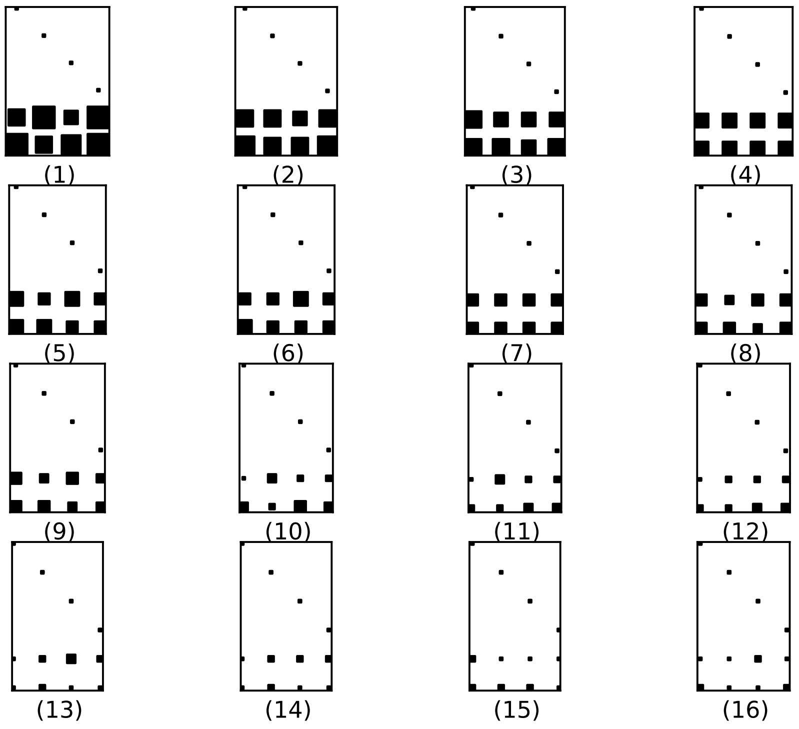


Figure 5.20.: Sequence of generator matrices for the randomized algorithm in Figure 5.18. Larger sized boxes indicate a higher MSSB.

5.8. Special-purpose generator matrices

A great advantage of the randomized matrix generation method is that it is possible to create matrices in a specific shape. In the general case a well balanced MSSB distribution might be desirable to ensure equal computational cost for the generation of redundancy information. However, there are several scenarios in which specific MSSB distributions can be beneficial. Figure 5.21 shows three different special-purpose matrices:

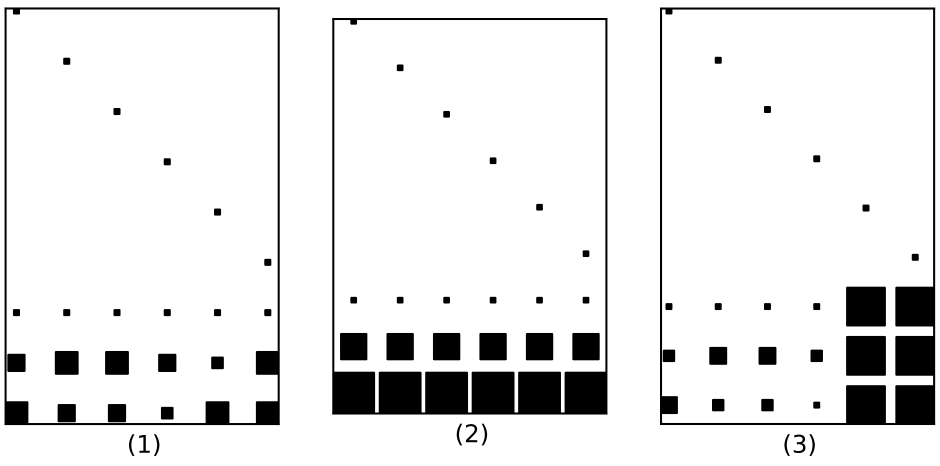


Figure 5.21.: Illustration of special-purpose matrices.

(1) One row of the systematic part can be fixed to contain only ones. Random elements are only selected for the remaining rows. Since

$$a \cdot 1 = a, \quad \forall a \in GF(2^p), \tag{5.15}$$

the symbol is effectively calculated by only adding up all data symbols. This is equivalent to the plain parity calculation of the standard RAID. As a consequence the average MSSB of the additional $k-1$ rows is increased. Yet it can be desirable to create the 1-error tolerance as fast as possible, whereas the degree of fault tolerance is gradually increased at later points in time (possibly by a different, dedicated subsystem or during system idle times). Another application scenario is archival: If the increased cost of a higher degree of fault tolerance is only justifiable for files that have been selected for long term storage, redundancy symbols can be added on demand without requiring full recalculation.

(2) Similar to the approach in (1) it could be beneficial to trade a fully balanced matrix for a version with increasing average row MSSB. In this scenario computational cost increases with every additional tolerable error. On average, the cost for the first rows can be chosen lower than for the fully balanced matrix. Again, this can be used for different fault-tolerance levels: For files that are frequently changing, only the lowest fault-tolerance could be chosen. As soon as the file becomes more stable the level could be gradually increased.

(3) Instead of balancing the rows it is also possible to use different average column MSSB positions. These larger matrix element could be reserved for devices that are most likely not in use (for instance as an option to increase the capacity of the system). Since then all data elements can be considered zero, the multiplications with these elements do not need to be executed. If for some reason theses additional devices become active, the additional computational cost is justifiable. According to Equation 4.74 this would require the update of all check symbols without needing to collect all corresponding data symbols. Another scenario is to compensate for different computational capabilities of the storage or client nodes. Load could be intentionally shifted to nodes with a higher number of cores, wider vector units, or dedicated accelerators. At the same time the load for less powerful nodes would be lowered.

Apart from the three examples any kind of special purpose matrix is imaginable. However, due to the randomized creation process, it must be assured that the matrix is securely recorded, for instance as part of the system metadata.

5.9. Algebraic signatures

In addition to the ability of reconstructing lost data due to entire disk failures it is also essential to detect data corruption. An important tool is end-to-end data protection through checksumming (Section 2.5.5). Modern enterprise disks offer therefore user-writeable data protection fields at the end of every sector. As a checksumming method the ubiquitous cyclic redundancy check (CRC) is commonly used. The implementation of a CRC16 checksum in the Linux kernel for this purpose [116] uses mainly shift, bitwise XOR and AND operations, but also relies on a small table, to generate a 2 byte checksum. Several different implementations with different generator polynomials exist. For cryptographic purposes special hash functions have been developed (for instance the SHA or MD families). Mainly designed to protect against intentional data manipulation, these could also be used as checksums for accidental data corruption. However, compared to the cyclic codes the cryptographic hash functions have a much higher computational cost. Another class of checksums that are suitable for the silent errors modes in storage systems are the so-called *Galois Power Series Signatures (GPSS)* [117, 118, 119]. GPSS belong to the class of *algebraic signatures* and they display several useful algebraic properties. As the name suggests the checksum is obtain by calculating a power series over a Galois field $GF(2^r)$. The block of data to be checksummed D is written as a series of l symbols, each r-bit wide:

$$D = d_0 d_1 d_2 \dots d_{l-1}, \tag{5.16}$$

Let β be a primitive element of the Galois field, that is, it is a root of the primitive generator polynomial and it generates all non-zero elements of the field:

$$GF_\beta(2^r) = \left\{0, 1, \beta, \beta^2, \ldots, \beta^{2^r-2}\right\}. \tag{5.17}$$

The β-signature of the block D is defined as

$$\mathrm{sig}_\beta(D) = \sum_{i=0}^{l-1} d_i \beta^i \tag{5.18}$$

and is itself a single element of the Galois field. The *n-fold β-signature* is defined as

$$\mathrm{sig}_{\beta,n}(D) = \Big(\mathrm{sig}_\beta(D), \mathrm{sig}_{\beta^2}(D), \ldots, \mathrm{sig}_{\beta^n}(D)\Big). \tag{5.19}$$

Correspondingly, the n-fold signature is a vector with n elements of size r bits. Some remarkable properties are:

- $\mathrm{sig}_{\beta,n}$ detects any changes up to n symbols for sure, given that the block length is less than $2^r - 1$ symbols.
- The probability for two uniformly distributed random blocks to have an equal $\mathrm{sig}_{\beta,n}$ is 2^{-nr}.
- If two blocks D and E of length l_D and l_E are concatenated into one block $D|E$ and $l_D + l_E \leq 2^r - 1$, the signature of $D|E$ is then calculated as

$$\mathrm{sig}_\beta(D|E) = \mathrm{sig}_\beta(D) + \beta^{l_D} \cdot \mathrm{sig}_\beta(E). \tag{5.20}$$

- If the block D is modified in m symbols starting at position s such that the vector of differences is $\delta = (\delta_0, \delta_1, \ldots, \delta_{m-1})$ with $\delta_i = d_i^{new} - d_i^{old}$, then

$$\mathrm{sig}_\beta(D^{new}) = \mathrm{sig}_\beta(D^{old}) + \beta^{s-1} \cdot \mathrm{sig}_\beta(\delta). \tag{5.21}$$

- If the horizontal parity p of n equal sized data blocks a_i is formed,

$$p = \bigoplus_{i=0}^{n-1} a_i, \tag{5.22}$$

  then the signature of the parity block $\mathrm{sig}_{\beta,n}(p)$ is equal to the parity of the signatures of the data blocks,

$$\mathrm{sig}_{\beta,n}(p) = \bigoplus_{i=0}^{n-1} \mathrm{sig}_{\beta,n}(a_i). \tag{5.23}$$

  This is also true in case of generalized Reed-Solomon codes.

Especially the last feature is immensely useful. It allows to check the consistency of an ensemble of data and check storage devices by only transmitting the β-signatures and performing the verification on them. The signatures for the individual blocks can be computed locally and compared to the signatures that have been recorded when the block was written. Conventionally, one agent has to collect all blocks from all data and check storage devices and perform the full computation. With the β-signature the computational load is distributed over all nodes and at the same time the network load is dramatically reduced since only the signatures have to be transmitted.

Current implementations utilize the power representation of the Galois field to perform up the computation. This involves once again the use of lookup tables. However, the power of the primitive element is already given as a factor in Equation 5.18 and the data symbols can be directly interpreted as logarithms. This way both table look-ups for the logarithms can be avoided. Figure 5.22 shows the pseudocode of the signature scheme specified in [117]. In this algorithm $log(0)$ is set to $2^r - 1$, therefore data blocks with this value are not accumulated.

```
1: procedure SIGNATURE($len, block[len]$)
2:     $sig \leftarrow 0$
3:     for $i \leftarrow 0, len - 1$ do
4:         if $block[i] \neq 2^r - 1$ then
5:             $sig \leftarrow \text{inv\_log}(i + block[i]) \oplus sig$
6:         end if
7:     end for
8:     return sig
9: end procedure
```

Figure 5.22.: Table-based single signature calculation.

Since two table look-ups are already eliminated, the signature scheme is does not really benefit from the polynomial algorithm. Since a mapping of the powers of β to the corresponding field element is not kept (this is the purpose of the logarithm look-up table), these powers of β would have to be computed on the fly. With wider vector units the cost for the calculation of the subsequent set of powers of β could eventually amortize. The maximum amount of 16-bit symbols than can be usefully processed into a single β-signature is $2^{16} - 2 = 65534$. In practice this is sufficient for many applications. The ANSI T10 data integrity field of modern enterprise discs protects every 512-byte sector with an additional 2-byte checksum inside an 8-byte data structure[5].

[5]The more recent 4k sectors are complemented accordingly with a 64-byte field.

5.10. Summary

In this chapter a novel erasure-resilient coding scheme has been presented. Instead of accelerating the existing multiplication operations over finite fields, a polynomial version has been proposed whose execution time depends on the MSSB in the factors. In order to fully exploit this property, it has been shown that the algebraically derived generator matrices for the linear systematic code can be replaced with matrices which are optimized for low average MSSBs in the elements of the systematic part. To find these matrices two algorithms have been presented: One based on transformations of the algebraically derived generator matrices, the other based on a Monte-Carlo approach. The latter approach has the advantage that it allows to control the MSSB distribution in the rows or the columns of the matrices. This enables special-purpose generator matrices, tailored to a specific usage scenario. The erasure-resilient code is complemented with an algebraic signature which provides an increasingly important protection against (silent) data corruption.

6. Results and Benchmarks

This chapter provides an overview of the performance of the proposed coding scheme. At first, the performance of the accelerated polynomial multiplication algorithm is evaluated individually. Thereafter, the properties of a selection of suitable matrices are presented. Vectorized multiplication and low MSSB generator matrices are then combined into a novel coding scheme. The resulting performance of the encoding and decoding operations are finally shown for the vectorized CPU and many-core GPU implementations.

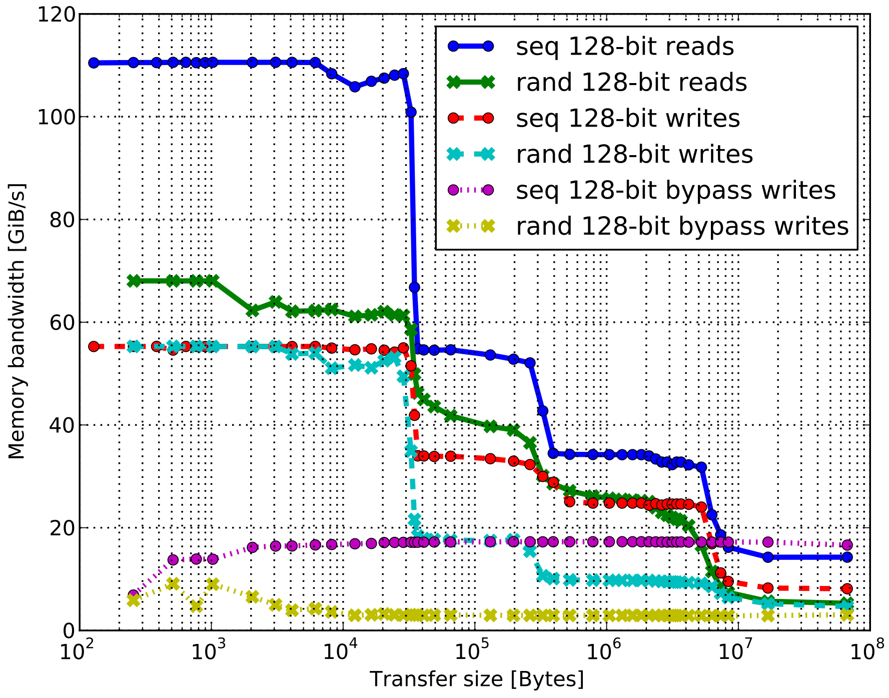
6.1. Testbeds

For the measurements in this section the following setups were used. The testbed for all CPU related measurements is described in Table A.2. The details about the GPGPU testbed are shown in Table A.3.

To better interpret the subsequently presented results, it is worthwhile to understand the limitations of the memory subsystem of the testbed. Let the XOR throughput be defined as follows: One buffer of size w is XORed with another buffer of the same size. The XOR throughput is equal to w divided by the execution time. The single thread XOR throughput for the testbed machine is

$$T_{\text{XOR}} = 5929 \pm 27 \quad \frac{\text{MiB}}{\text{s}}, \tag{6.1}$$

with w larger than 10 MiB. Figure 6.1 shows the memory bandwidth for various access patterns of the testbed system. All memory accesses use 128-bit loads and stores. Writes are also performed in the non-temporal fashion (that is, the cache is bypassed and data are directly written into memory). The accesses can be either sequential or random. The plots reveal several details about the memory hierarchy of the system. The sharp drop around 32 kiB for cached accesses is due to the L1 cache. Another drop occurs at around 256 kiB, matching the L2 cache size of the processor. The final drop between 5 and 6 MiB is caused by the L3 cache, which is shared between all cores and the integrated GPU. The lower plot shows a close up of the region that is most relevant for the streaming access pattern. For transfer sizes beyond 10 MiB the sequential read bandwidth is around 14.5 GiB/s, the sequential write bandwidth is around 8.4 GiB/s and the cache-bypassing sequential write bandwidth is a around 17 GiB/s. These numbers are later used to determine a limit for the achievable encoding and decoding performances, for both CPU- and GPU-based implementations.



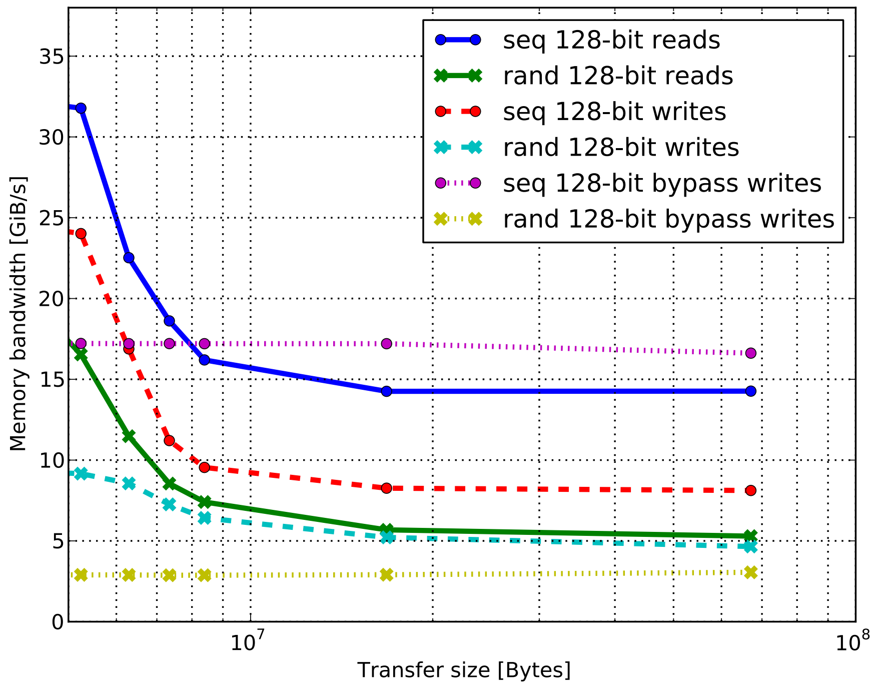
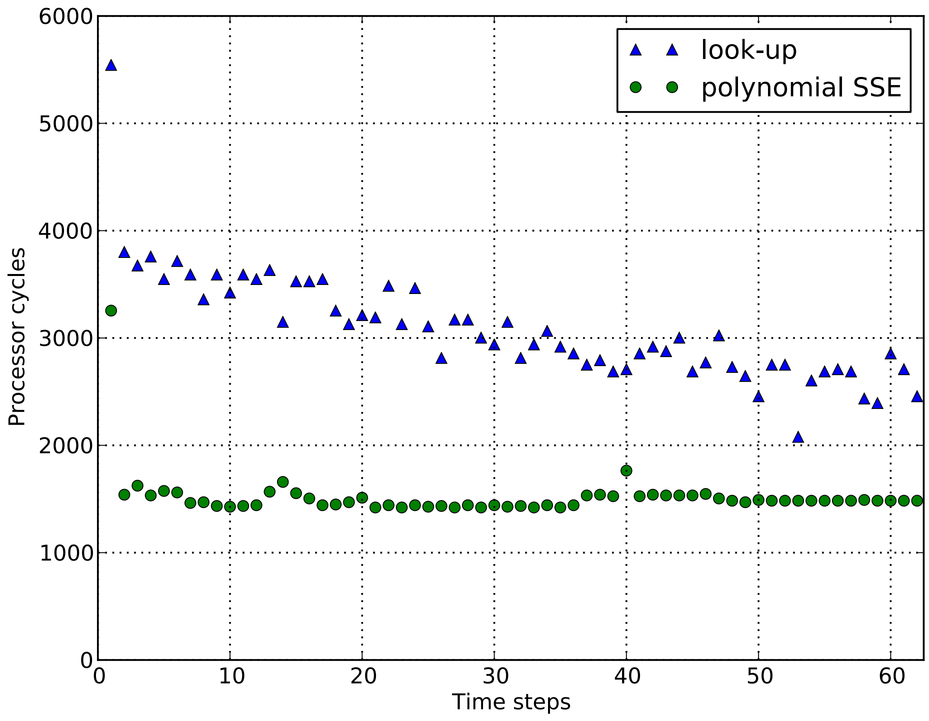


Figure 6.1.: Memory (subsystem) read and write bandwidth of the testbed machine for different access modes. Measurements were performed with [120]. The lower plot shows a close up of the relevant region beyond 1MiB transfer size.

6.2. Multiplication

6.2.1. SSE implementation

The vectorized polynomial multiplication is the basis for the low MSSB coding scheme and it is therefore of interest, to examine the (isolated) multiplication performance. In case of the differential encoding of data symbols, a single multiplication and two XORs are sufficient to recalculate a check symbol. The performance of the polynomial multiplication depends on the number of loop cycles (given by the MSSB) that have to be calculated. The multiplication performance with pseudo-random numbers for three different implementations is shown Figure 6.2: $L_{\max}$ can be determined *dynamically* as described in Figure 5.8, which introduces a fixed number of SSE instructions before the actual multiplication. In case of differential encoding one factor is a fixed matrix element and the number of loop cycles can be identified in advance for a large number of elements and simply passed to the multiplication routine as an *argument*. The most obvious implementation contains a *conditional* branch to break out of the loop in case one of the vector registers has become zero during the execution of the algorithm. Depending on the actual number of loop cycles the performance is much better than for a logarithm-based lookup table. All measurements were performed with buffer sizes beyond 50 MiB, giving thereby the table-based implementation the full benefit of the sizable processor caches in the testbed system. The polynomial implementation on the other hand works largely cache-independent (except for read-ahead of the multiplication arguments). The write-back is using non-temporal stores, in order to bypass the cache. This is reasonable for stream computing patterns, where the processed data only display little temporal locality. Figure 6.2 illustrates the behavior of both algorithms at startup. The 64-bit *Time Stamp Counter* available on x86 processors [121] can be periodically sampled and used as a probe into the code execution at cycle level. In every time step 32 multiplications are performed and the number of processor cycles are measured (processes are pinned to a fixed CPU core and executed with real-time priority. The power saving and dynamic overclocking features of the processor are disabled). Cold cache misses affect both algorithms in the very beginning. The polynomial multiplication performance then immediately steadies, whereas for the table-based multiplication the caches have to be slowly populated with the tables for logarithm and inverse logarithm. Even though the testbed system already supports the wider 256-bit vector registers, the presented implementation is using 128-bit vector instructions due to the lack of appropriate wide integer instructions. These will be available in future processor generations and nominally provide double the multiplication performance.



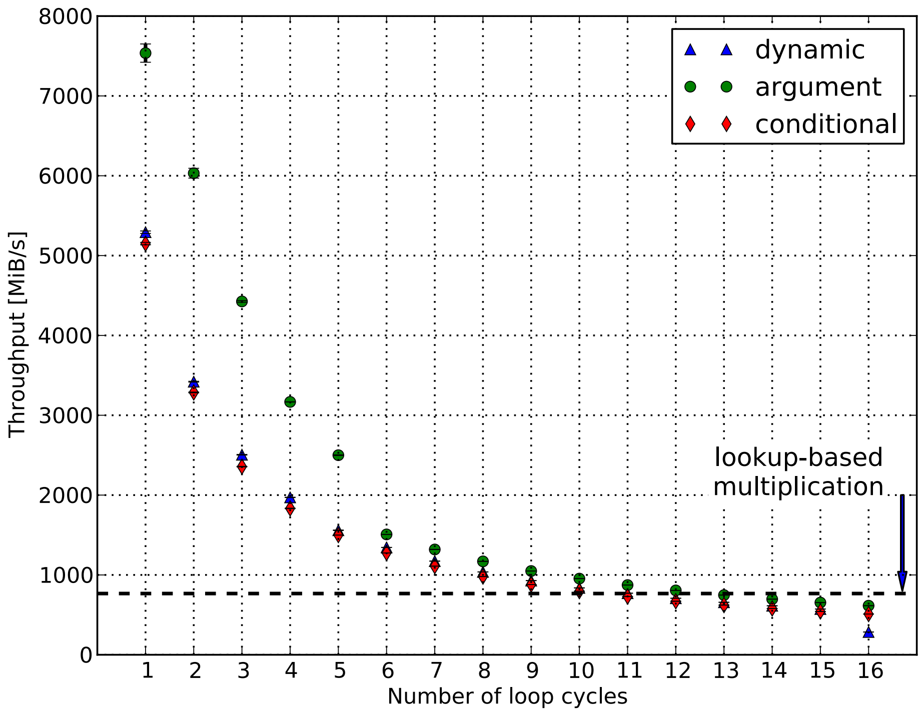


Figure 6.2.: Performance of both algorithms during startup (top). Multiplication performance for different implementations depending on the number of loop cycles (bottom).

6.2.2. GPGPU performance

Similar to the vectorized multiplication the GPGPU multiplication is examined in this section. Two buffers containing the multiplication arguments are transferred to the GPU. Every individual multiplication is then assigned to one thread and the buffer containing the multiplication result is transferred back. In order to avoid divergent execution paths of threads on the same multiprocessor (which results in serialized execution), the full multiplication is performed (with the maximum number of loop cycles). The communication cost between GPU and host limits the achievable throughput rate. Assuming comparable up- and down-stream PCI bandwidths $B_{\text{PCI,up/down}} \sim B_{\text{PCI}}$, the transfer throughput alone (without any computation on the GPU) is

$$T_{\text{transfer}} = \frac{1}{\frac{2}{B_{\text{PCI,up}}} + \frac{1}{B_{\text{PCI,down}}}} \sim \frac{1}{3} \cdot B_{\text{PCI}}. \qquad (6.2)$$

In accordance with this limit Figure 6.3 shows the overall performance (including all communication overhead) for up to four overlapping streams depending on the block size of the argument and result buffers. The product $a_i \cdot b_i = c_i$ is calculated using n streams, with n kernels for chunks of size $blocksize/n$. The throughput is then the amount of result data divided by the total time. The total time includes the time to transfer buffers a and b to the GPU, the time to perform the multiplication on the GPU and the time to transfer buffer c back to the host. The cost for setup and transfer of kernel and buffers is significant, but can be amortized by using large block sizes beyond 1MiB. The streamed execution helps to overlap communication and computation, such that the transfer limit is reached. For comparison a table-based implementation (log/inv_log) is shown (the tables are transferred to and stored on the GPU beforehand and do therefore not count for the calculation of the throughput).

6.3. Matrices

In this section the properties of the best found low-MSSB generator matrices are examined. Figure 6.4 illustrates a low-MSSB generator matrix and a corresponding decode matrix. Figure 6.5 shows the MSSB position for a range of matrices. A selection of these matrices and their properties are given in Appendix A.2. All of these matrices are fully balanced, without any privileged rows. The average MSSB position for every corresponding decoding matrix is shown in Figure 6.6.

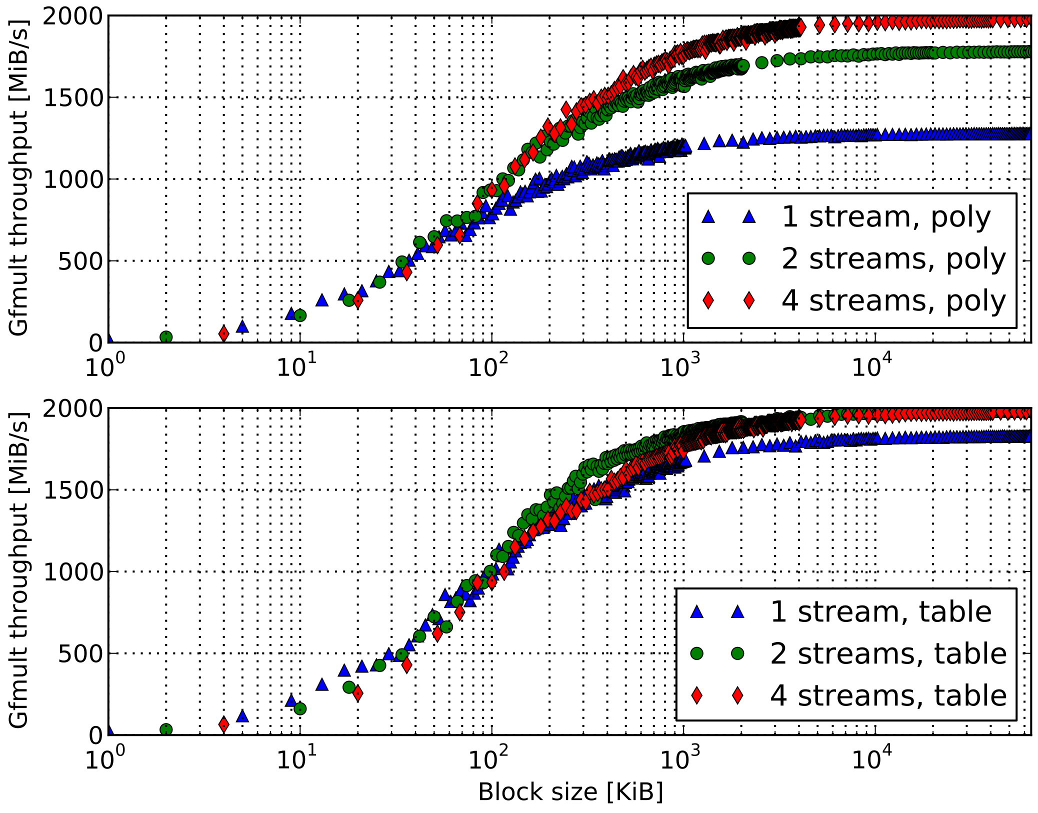


Figure 6.3.: GPGPU implementation of polynomial and table-based multiplication running on a Nvidia GTX 680 GPU with page-locked host memory.

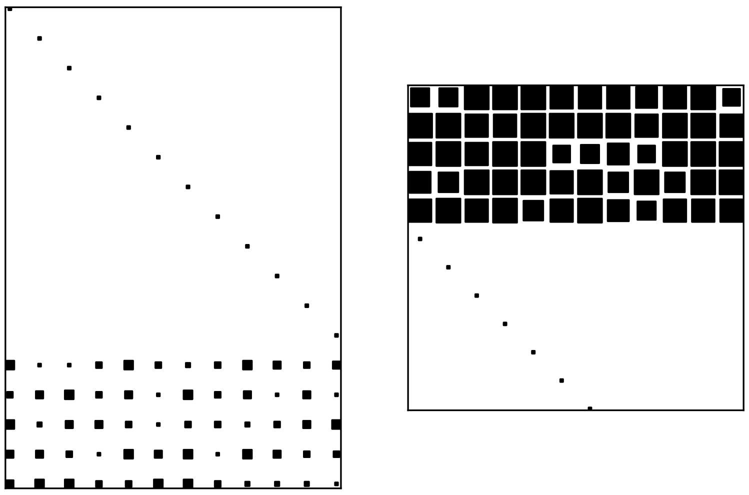


Figure 6.4.: Visualization of the MSSB distribution of a low MSSB generator matrix (left) and one of the corresponding decoding matrices (right). Larger sized boxes indicate a higher MSSB.

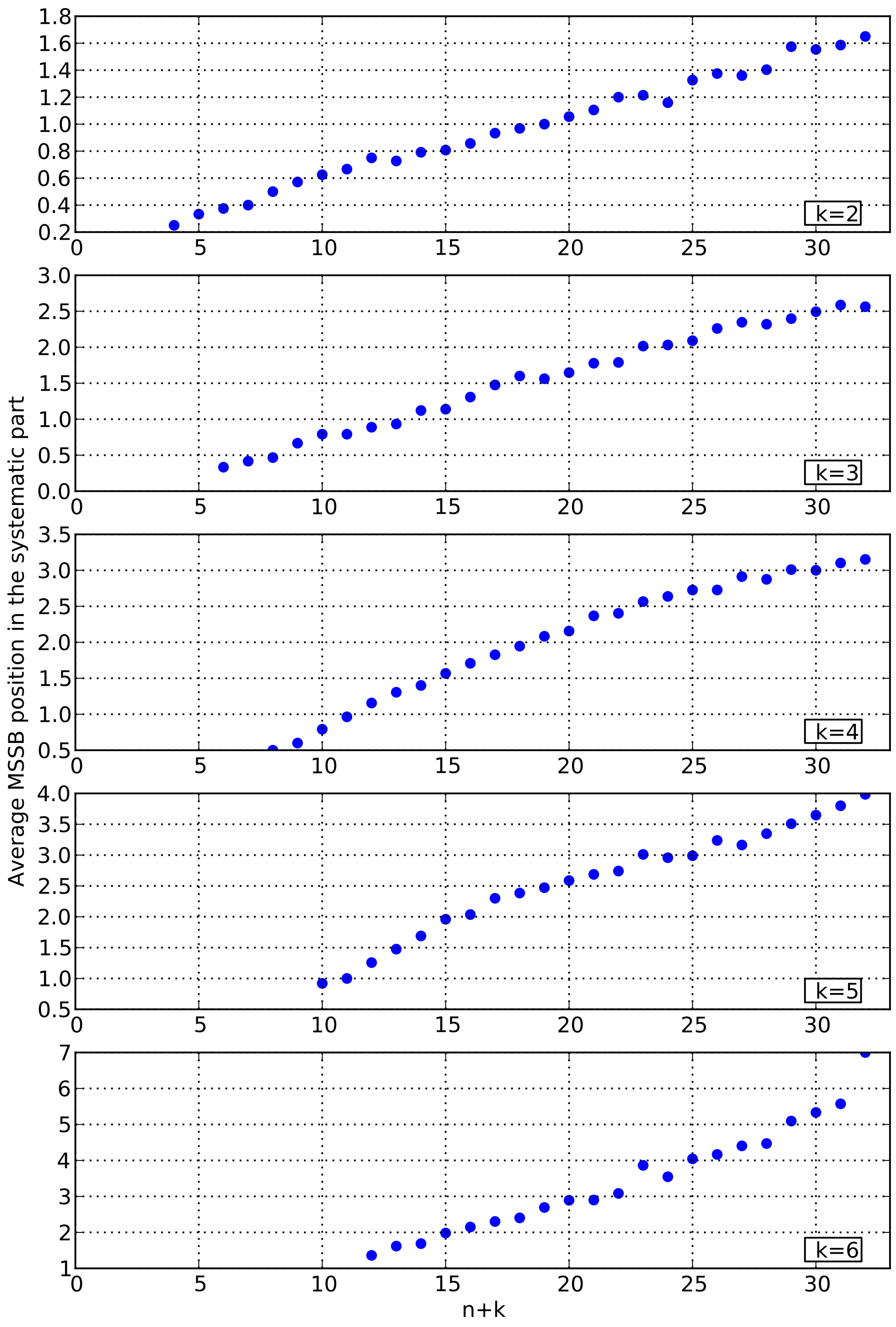


Figure 6.5.: Average MSSB position of the systematic part of the low-MSSB generator matrices.

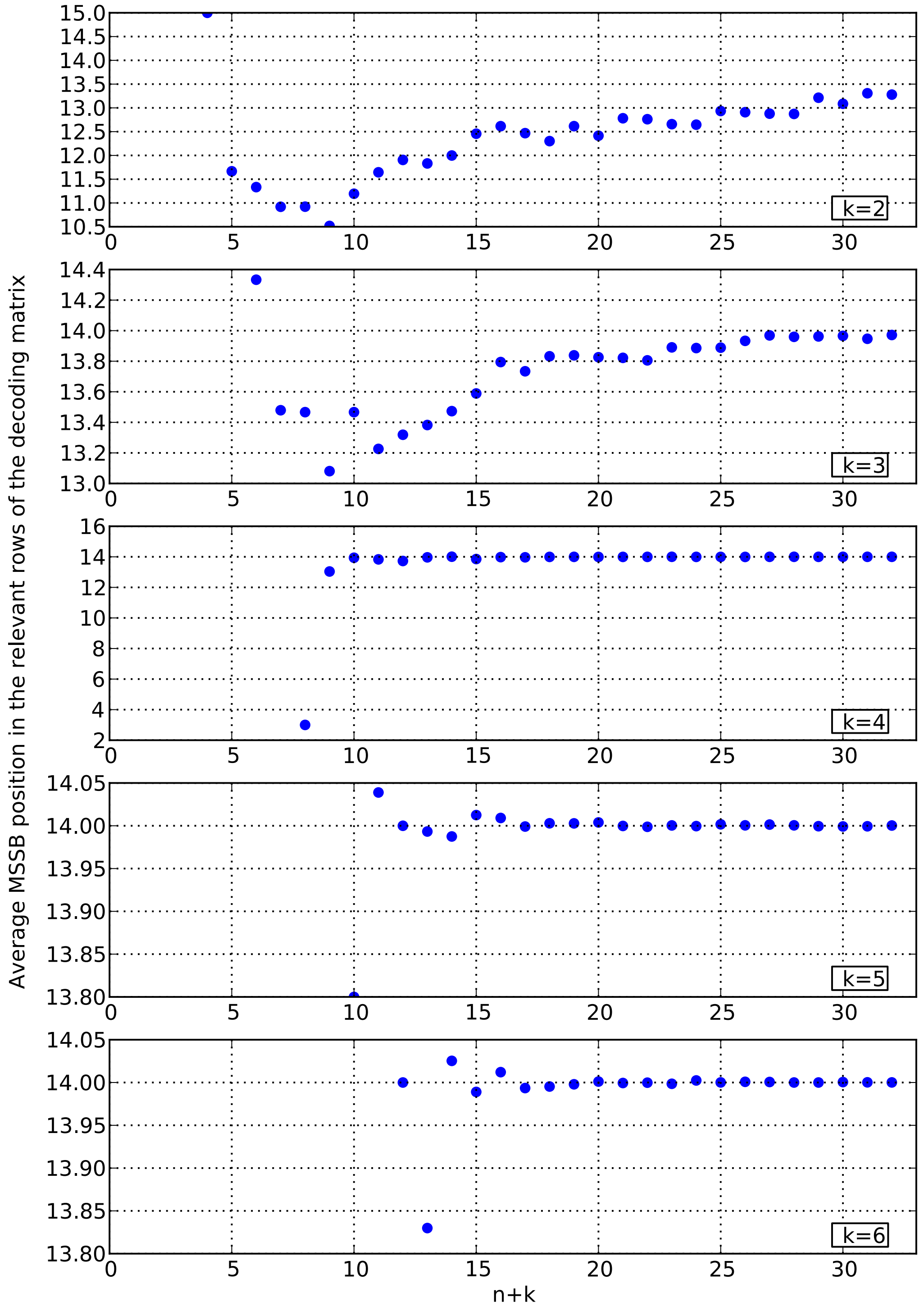


Figure 6.6.: Average MSSB position in the relevant rows of the decoding matrix. Unfortunately, the property of low average MSSB positions is not conserved.

6.4. SIMD coding

In this section the actual coding performance is presented. Figure 6.7 shows the basic task of creating k check buffers $c_0, \ldots, c_{k-1}$ for $n-k$ data buffers $d_0, \ldots, d_{n-k-1}$. The problem is independent of the size of the buffers, but for the following results it is assumed that the individual buffer size is at least $B = 4$ KiB.

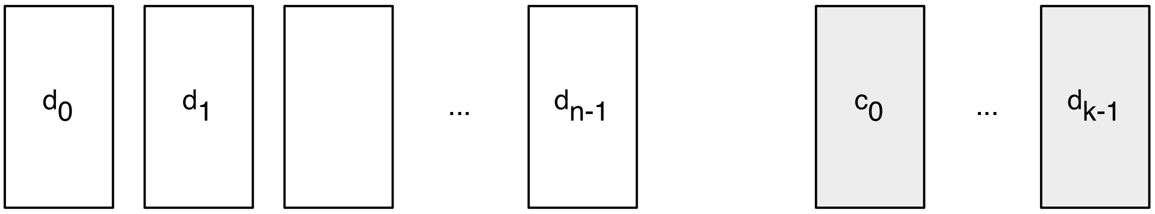


Figure 6.7.: Encoding: For n data buffers, k buffers with check symbols have to be calculated.

It is evident that the performance of the coding operations is limited by the available memory bandwidth. Every data symbol has to be read, multiplied with a matrix element in a row corresponding to a check symbol, and finally all multiplication results have to be XORed to obtain the check symbol. A good upper limit of the achievable performance is therefore the pure XOR throughput: the Galois multiplication is omitted and all data symbols are XORed horizontally. As a result the identical parity for every check symbol is obtained. Figure 6.8 shows the performance of this procedure for different values of n and k. The throughput is the total size of all data buffers divided by the processing time t_p:

$$T = \frac{n \cdot B}{t_p} \tag{6.3}$$

Figure 6.9 shows the performance of the encoding procedure with a low-MSSB generator matrix using polynomial SIMD multiplication. The result is compared to the XOR throughput in Figure 6.10. For the given parameter range the low-MSSB approach achieves between 20% and 80% of the XOR performance. Figure 6.11 shows the encoding performance with a Vandermonde generator matrix using the table-based multiplication. As expected, the encoding performance is not dependent on $n+k$. At last the relative performance with respect to the performance of the Cauchy-Reed-Solomon implementation in the Jerasure [122] library is shown. For the projection parameter $w = 16$, the low-MSSB approach is up to a factor of 2.8 faster. For $w = 8$ the performance is still considerably higher for smaller k and becomes more comparable to the Jerasure performance for large $n+k$. Absolute results are shown for both one and four participating cores (all four cores share the same memory). The multicore parallelization was achieved by using OpenMP (Open Multiprocessing) [123], a multi-platform shared memory multiprocessing API. Chunks of the data to be encoded are assigned to the participating threads. Since all vertical data symbols are independent no further synchronization between the threads is required.

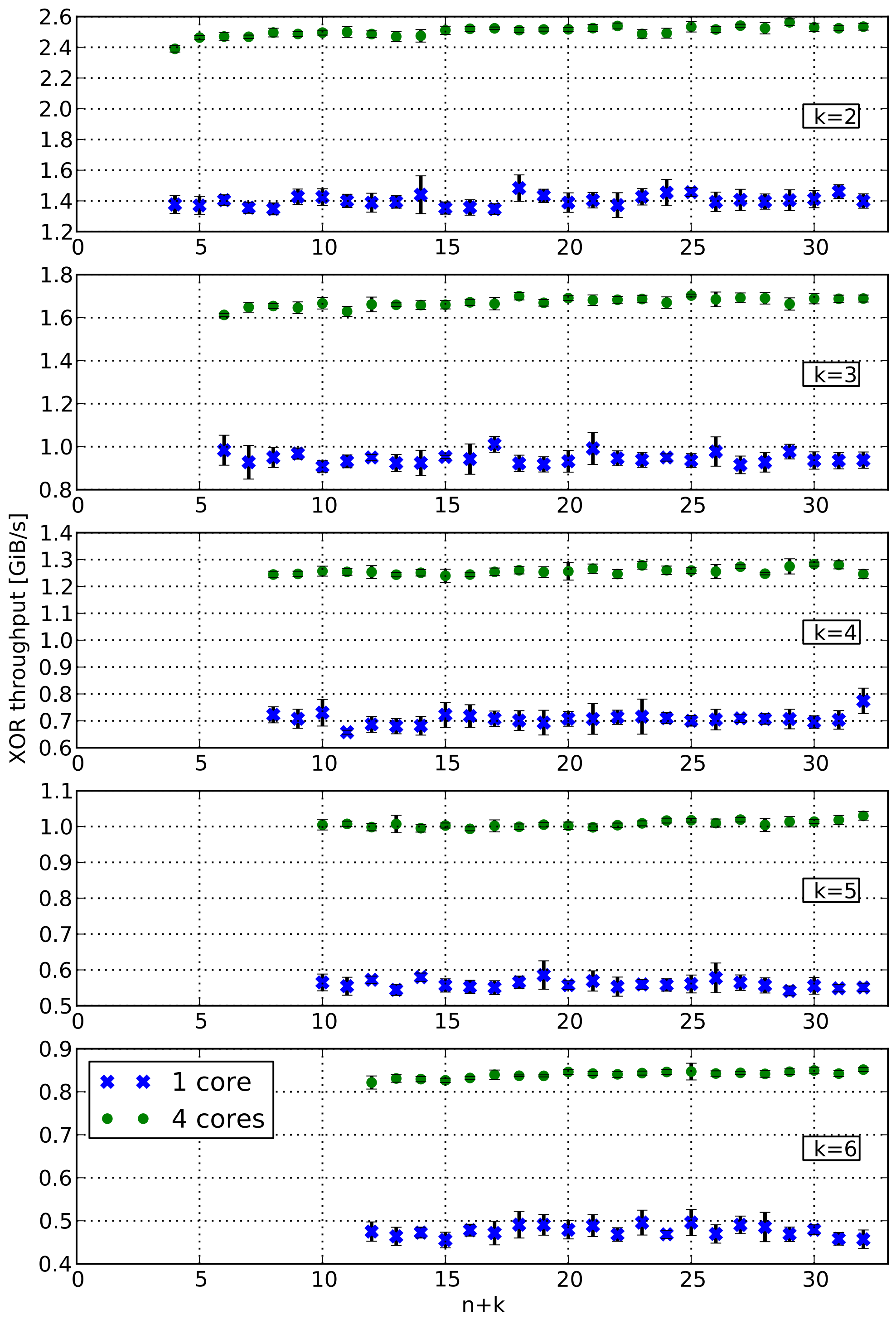


Figure 6.8.: Pure XOR throughput.

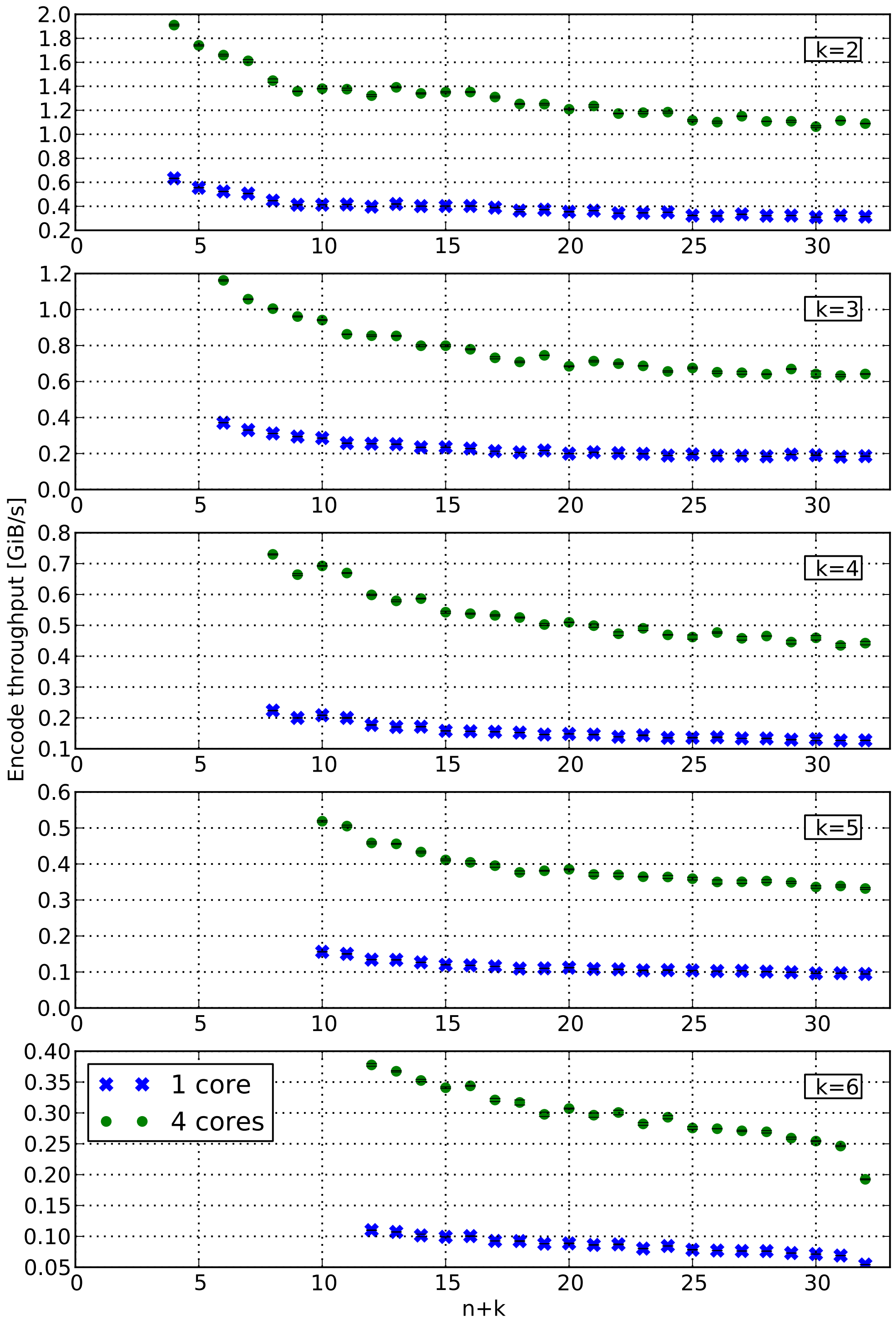


Figure 6.9.: Encoding performance for polynomial SIMD multiplication.

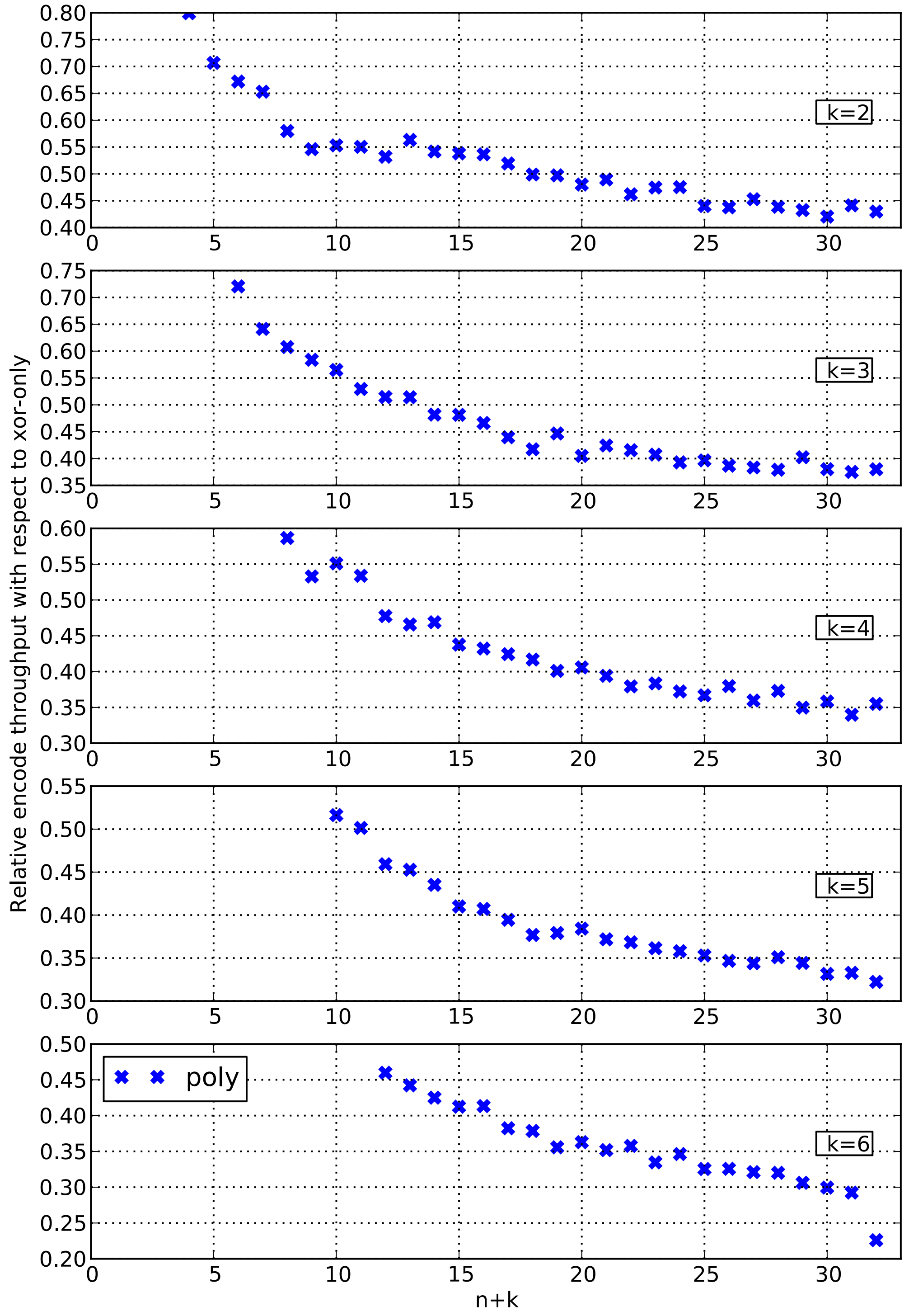


Figure 6.10.: Relative performance of the polynomial SIMD multiplication with respect to the pure XOR throughput.

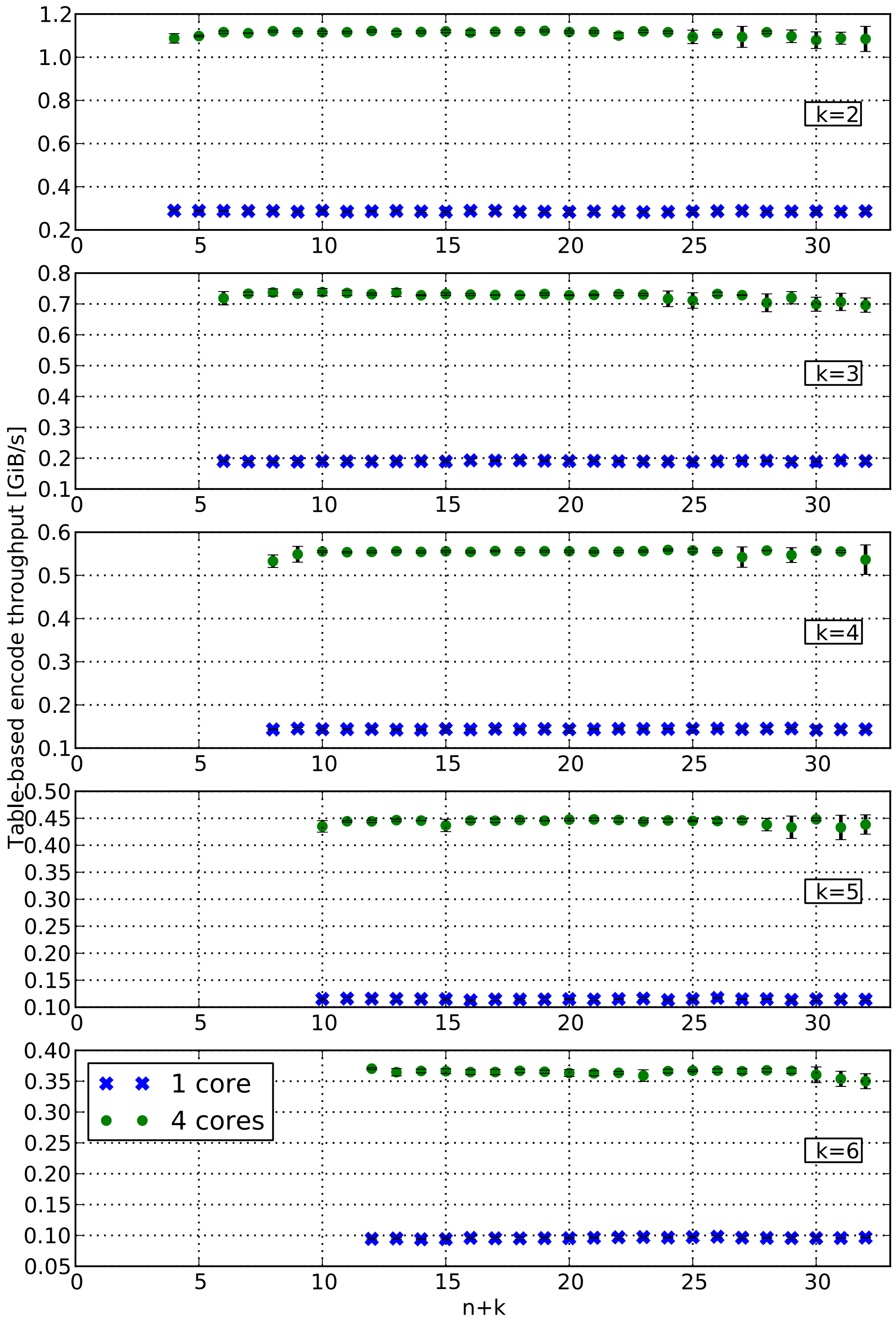


Figure 6.11.: Log/inv_log table-based encoding performance.

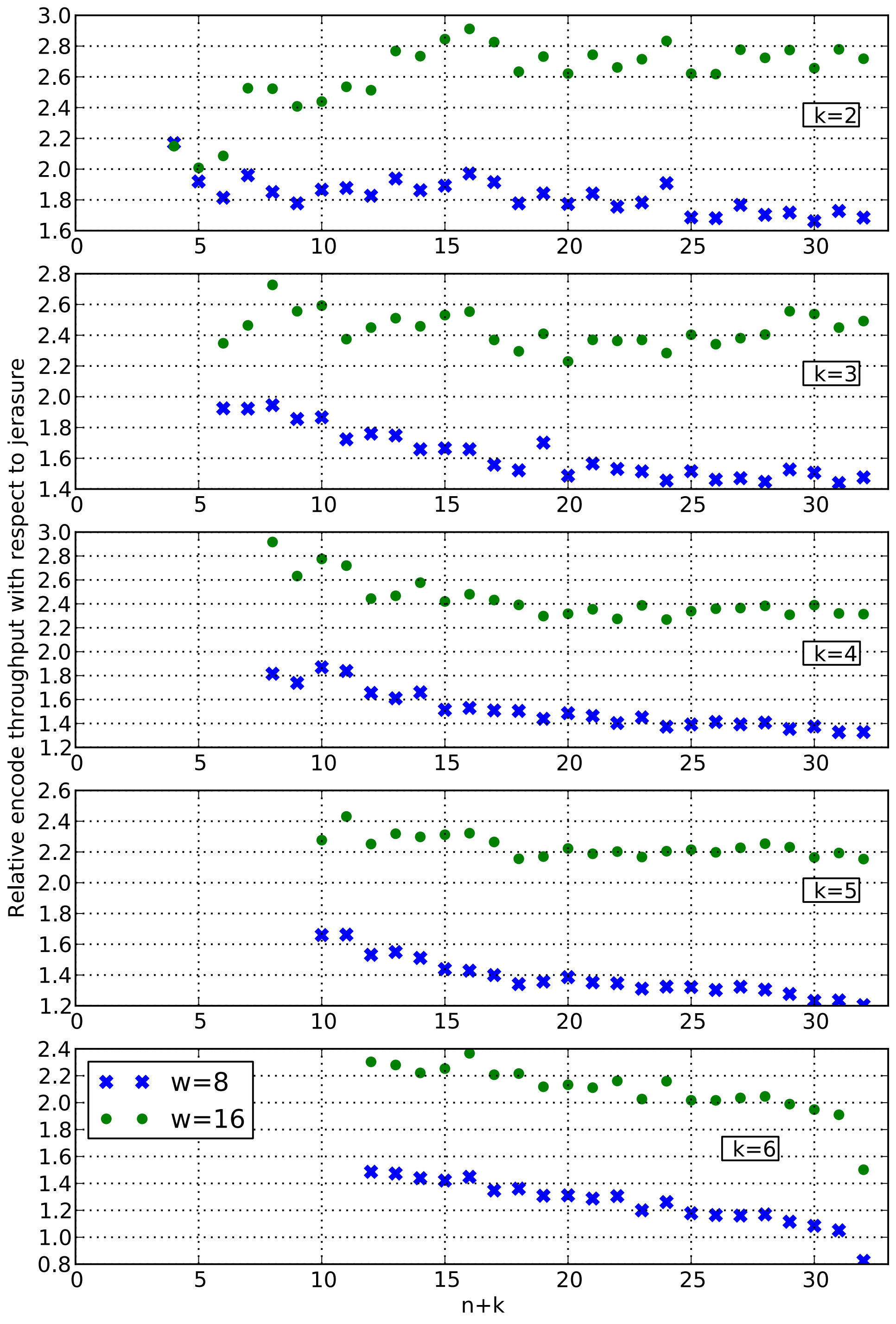


Figure 6.12.: Relative performance of the polynomial SIMD multiplication with respect to an available Cauchy-Reed-Solomon coding library. *w* is the projection parameter of the Cauchy-Reed-Solomon codes.

6.5. GPGPU coding

The performance of a GPGPU co-processor is highly dependent on the actual application [124], in particular of the memory access patterns of the application. Encoding clearly belongs to the class of throughput oriented problems. The data flow is from the host computer memory to the main memory of the GPGPU, from there to the processing units of the GPU and all the way back after comparably short amount of computation. Clearly, the slowest segment in the communication path (in terms of bandwidth) limits the overall performance. To understand what performance can be expected from a GPGPU co-processor, it is helpful to model only the transfers from host to GPGPU memory and from GPGPU memory to the processing units. The (non-pipelined) transfer throughput to the execution units of the GPGPU and back into the main memory of the host (without any calculations on the GPGPU) is dependent on n, k, and the different memory bandwidths (B_{PCI}, B_{GPU}):

$$T_{\text{transfer}} = \frac{n}{\frac{n}{B_{\text{PCI,up}}} + \frac{k}{B_{\text{PCI,down}}} + \frac{n+k}{B_{\text{GPU}}}} \tag{6.4}$$

With the reasonable assumption that the GPGPU memory bandwidth is higher by a constant factor of α than the PCI bandwidth, and that $B_{\text{PCI,up}} = B_{\text{PCI,down}}$, Equation 6.4 can be written as

$$T_{\text{transfer}} = \frac{n}{(n+k)\cdot(1+\alpha^{-1})} \cdot B_{\text{PCI}} \tag{6.5}$$

To simulate the computational execution time, a delay parameter δ is introduced, which specifies a delay as a multiple of the PCI transfer time. For instance, $\delta = 1$ indicates that the computation consumes the same amount of time as the PCI transfer.

$$T_{\text{transfer,delay}} = \frac{n}{(n+k)\cdot(\delta+1+\alpha^{-1})} \cdot B_{\text{PCI}} \tag{6.6}$$

Figure 6.13 illustrates the achievable transfer throughput for different values of n and k, $B_{\text{PCI}} = 6$ GiB/s, $\alpha = 10$, and $\delta = \{0,2\}$. This highlights that the overall performance is tied to the PCI Express bandwidth. Even when the computational part is ignored, the communication pattern alone has a severe influence on the overall performance. The goal is therefore to hide the computational part entirely behind communication by using a streamed execution pattern. The low-MSSB generator approach proves to be ideal for this. The maximum MSSB position for all elements in the systematic part of the matrix can be used to limit the number of loop cycles in the polynomial multiplication on the GPGPU. Figure 6.14 shows the performance of the GPGPU implementation of the encoding procedure with a low-MSSB generator matrix for different values of n and k. For $k = 2$ and $k = 3$ the computation can be almost completely hidden behind the data transfers and the encoding performance surpasses 5 GiB/s for larger $n+k$. Starting from $k = 4$ an additional penalty for computation becomes significant. The visual profile of the two kernel executions for fixed $n = 24$ in Figure 6.17 illustrates how the penalty accumulates. For $k = 2$ (top) the computation can always begin immediately after the transfer and can also be immediately overlapped with the transfer of the following stream. For $k = 4$ (bottom) the kernel

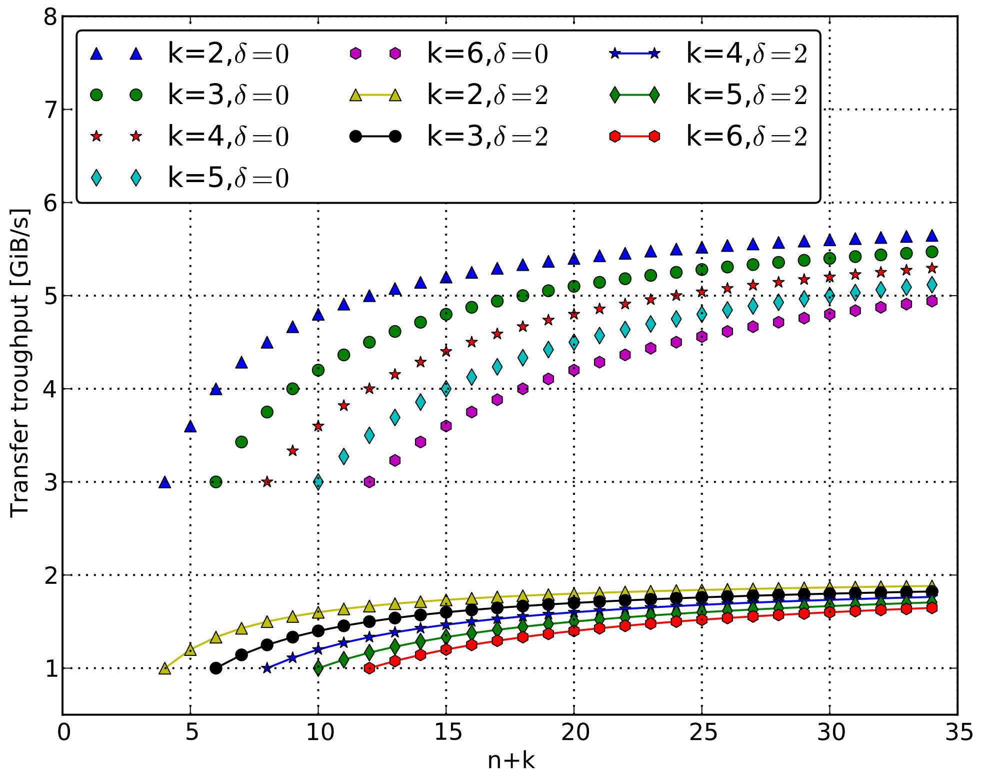


Figure 6.13.: GPGPU transfer throughput with and without artificial delay $\delta = \{2,0\}$.

execution time is longer than the transfer time. This leads to ever increasing intervals between the end of the host-to-device transfer and the beginning of the kernel execution of the corresponding stream. The steps that are visible are due to an increasing δ with increasing $n+k$. Since the execution time of the multiplication is determined by the largest matrix element, the overall performance drops whenever the maximum of the elements of the following matrix is increased. Some of this additional computational overhead can be hidden for the larger number of streams (as a trade-off between increased kernel execution time and reduced non-overlapped back transfer time). For $k = 2$ and $k = 3$ and with 16 streams encoding throughputs beyond 5 GiB/s can be achieved, close to the upper bound set by the presented model. For larger k the effects of the stepwise increase of computational load are more prominent. Figure 6.15 shows the performance of the corresponding dual table-based implementation. The increased pressure on the memory subsystem of the GPU caused by the additional look-ups has a drastic impact on the overall performance. The comparison of both approaches is shown in Figure 6.16. Particularly for larger k the low-MSSB approach is significantly faster, up to a factor of 7.5.

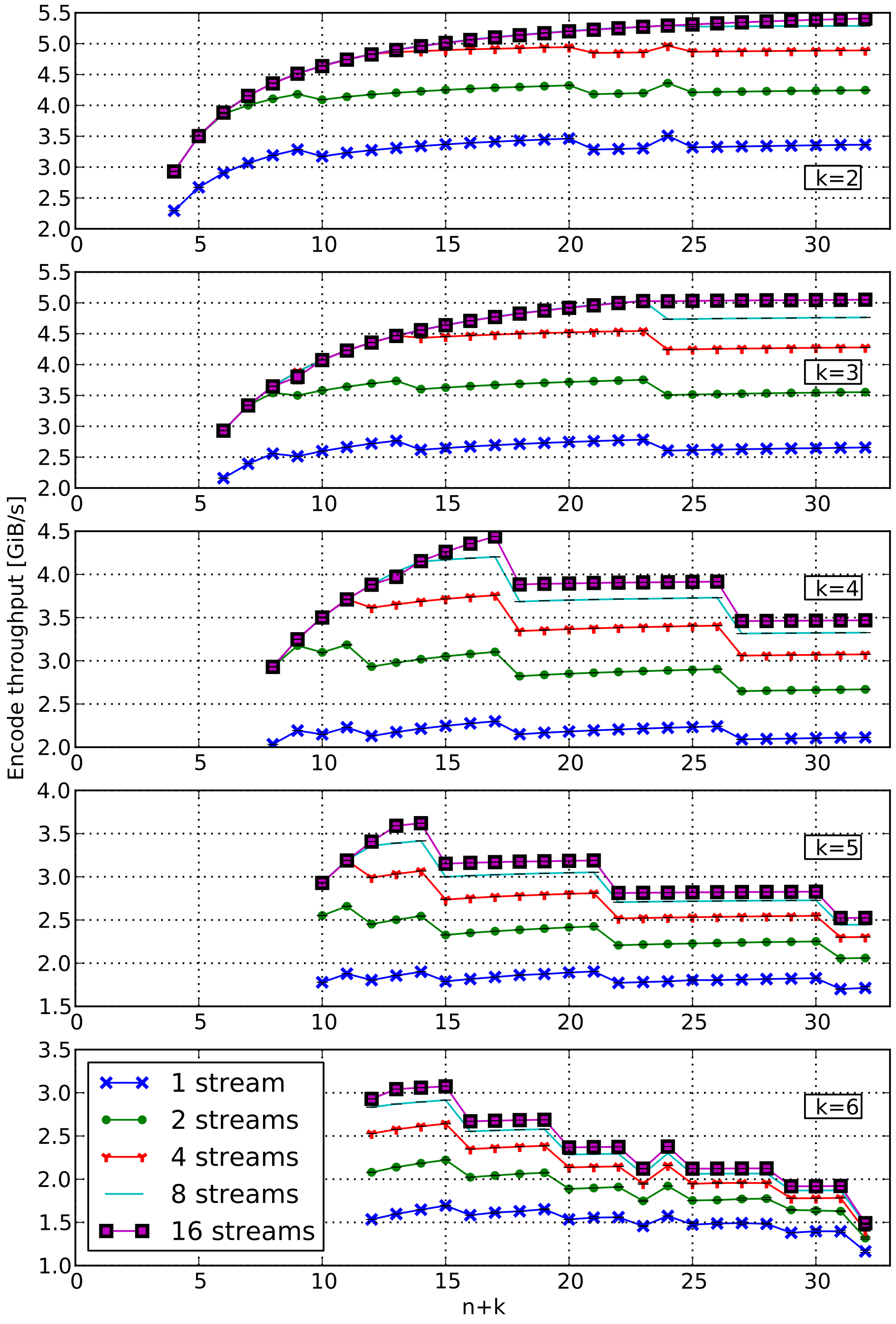


Figure 6.14.: GPGPU encoding performance for the low-MSSB approach with polynomial multiplication.

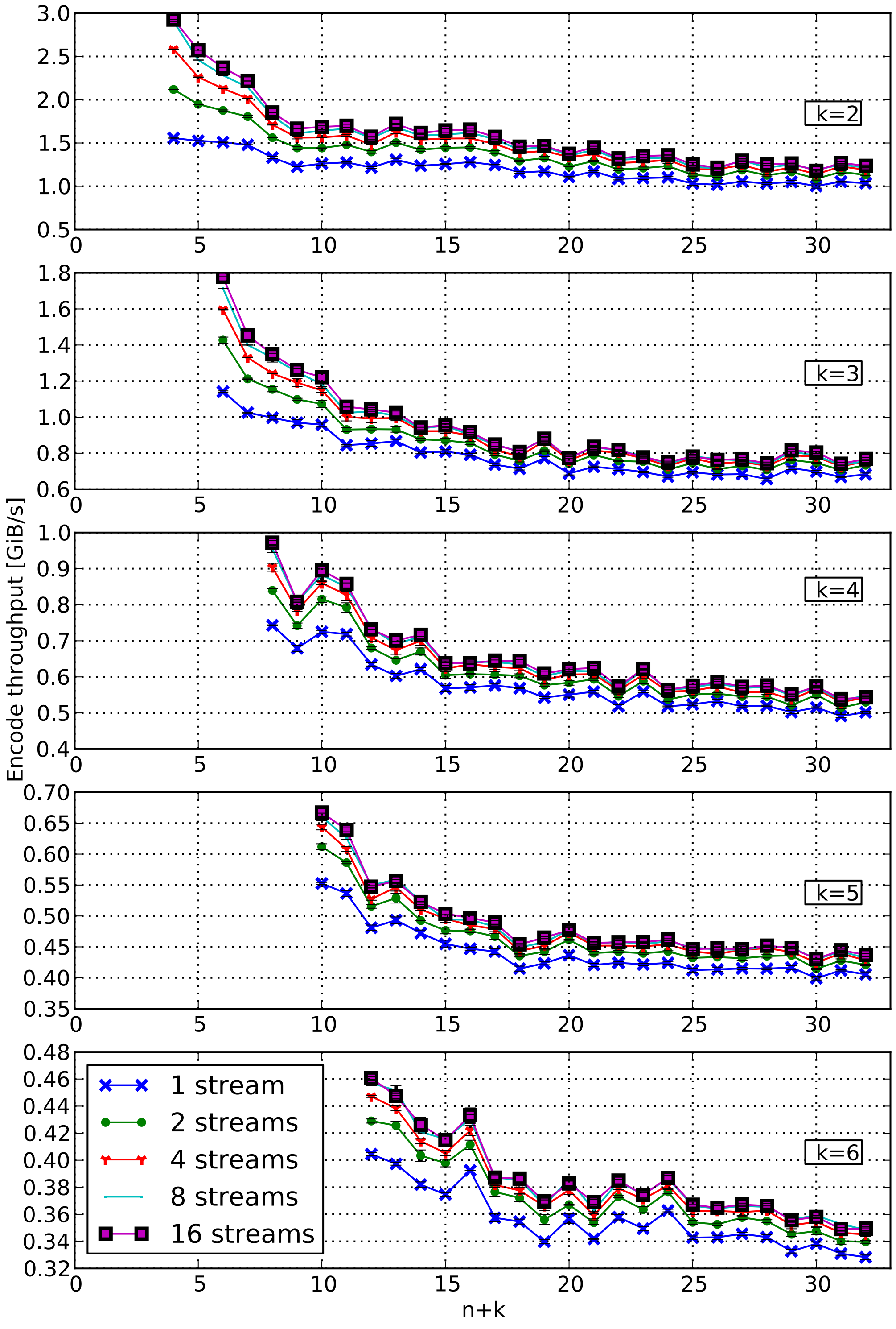


Figure 6.15.: GPGPU encoding performance for the table-based approach.

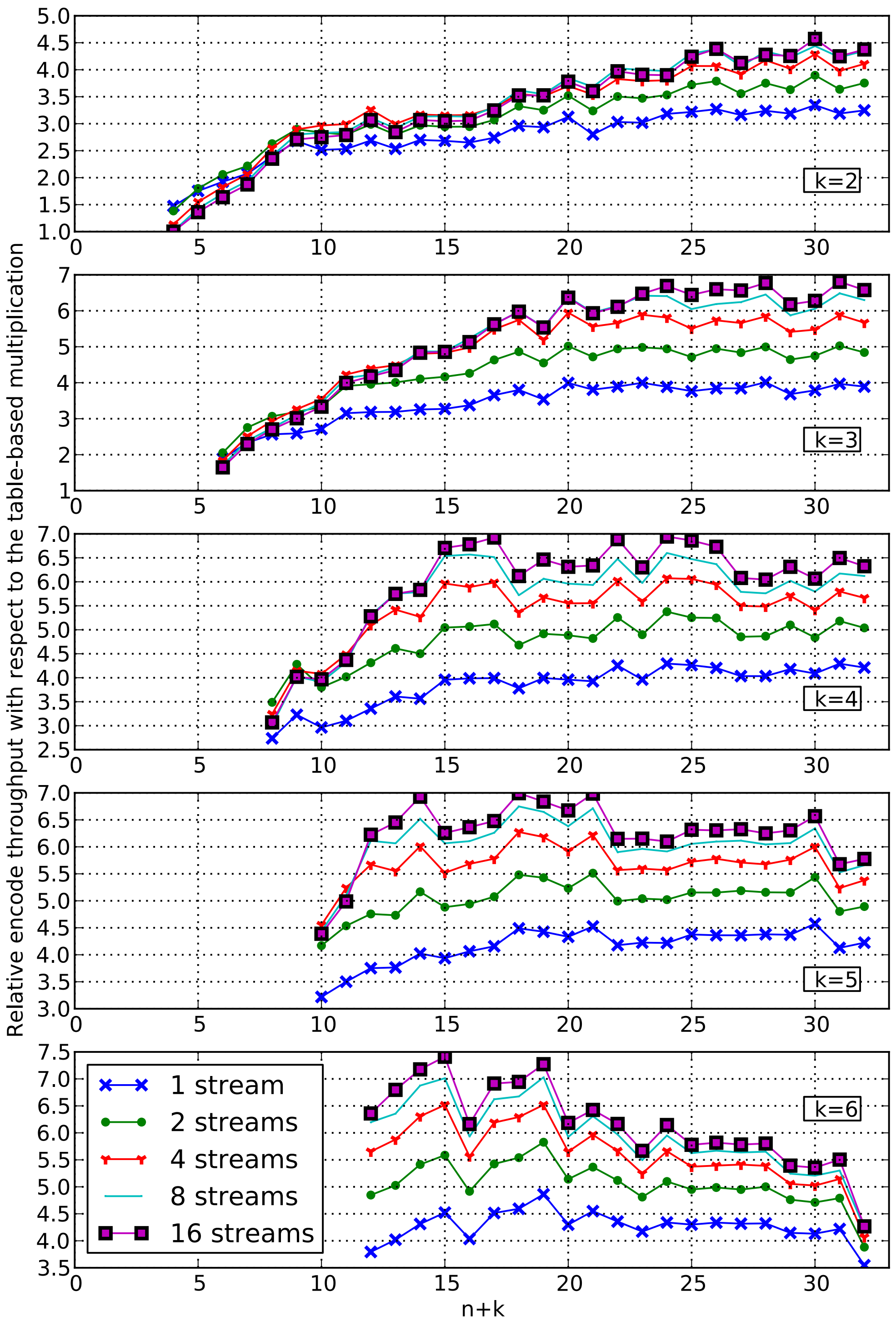


Figure 6.16.: Relative performance of the low-MSSB approach with polynomial multiplication with respect to the table-based implementation.

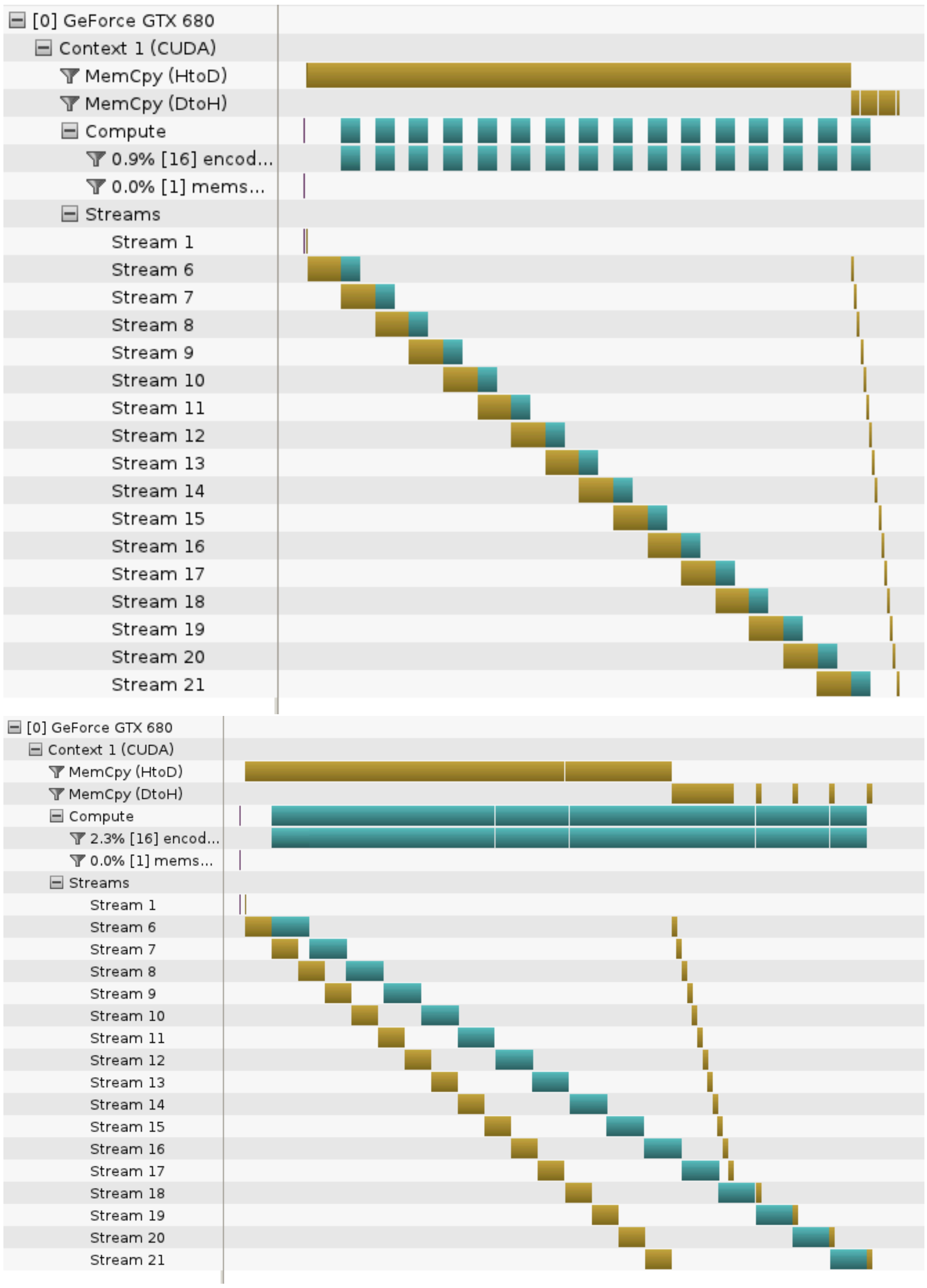
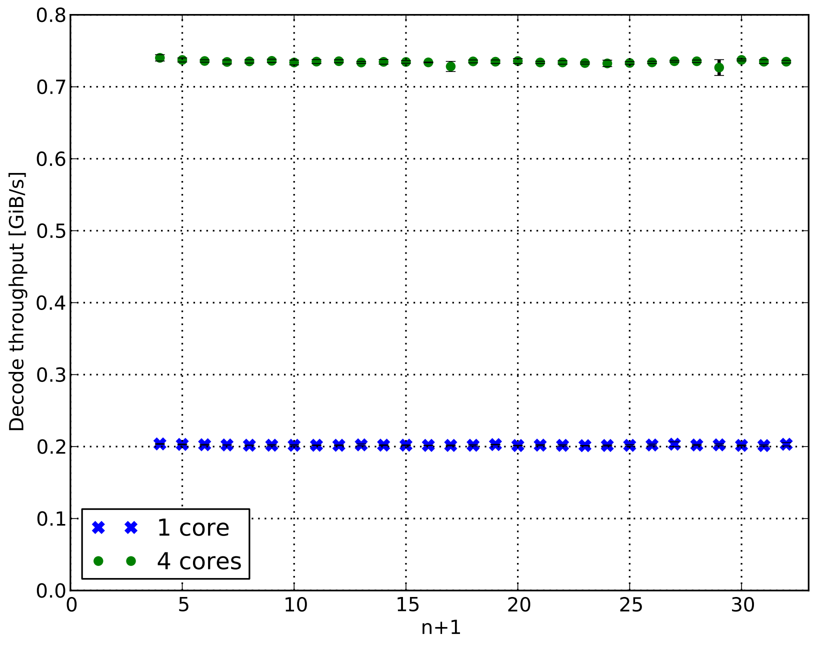


Figure 6.17.: Visual profile of a streamed encoding for $n = 24$ and $k = 2$ (top) and $k = 4$ (bottom). The plots use a different scale.

6.6. Decoding

In principle encoding and decoding are symmetrical. For each of the k check symbols the n elements in a row of the matrix have to multiplied and XORed with a vector of n elements. In case of encoding the matrix rows are the systematic part of the generator matrix and the vector consists of all data symbols. In case of decoding the matrix rows are those rows in the inverted matrix which correspond to the erased data symbols, and the vector contains the surviving data and check symbols. After the failure of a device that holds check symbols, the reconstruction is equivalent to the initial encoding step for that particular device. While the coding step usually contains the calculation of all check symbols, the reconstruction is generally initiated as soon as the failure of a device is detected. Therefore, it is reasonable to treat the single device reconstruction as the common case. The analysis of the properties of the decode matrices that are obtained from the low-MSSB generator matrices (Figure 6.6) has shown that the multiplications with elements of the decode matrices are on average more costly. Instead of enumerating all possible decode matrices and evaluating the performance that is achieved with them, a worst-case estimation of the decode performance is given. All multiplications (both on the CPU and the GPU) are adjusted such that the full number of 16 loop cycles have to be carried out. Figure 6.18 shows the worst-case single device decoding performance for both implementations. The CPU implementation again uses OpenMP for the work distribution to multiple cores. The throughput in these plots is given as the (minimum) amount of surviving data symbols divided by the time required to reconstruct the lost symbol. Therefore the throughput is also modeled by Equation 6.3, where the processing time t_p is equivalent to the time required to reconstruct one check buffer in this case. This number is an indicator how much data need to be transferred to the decoding agent in order to saturate the reconstruction. The plots also show that, in case of the GPU, the reconstruction can be completely hidden behind the streamed communication (for the presented set of parameters). The decoding throughput is higher than what most data center and high-performance interconnects, such as 10GbE/40GbE or 40Gb InfiniBand, are able to deliver over a single link.



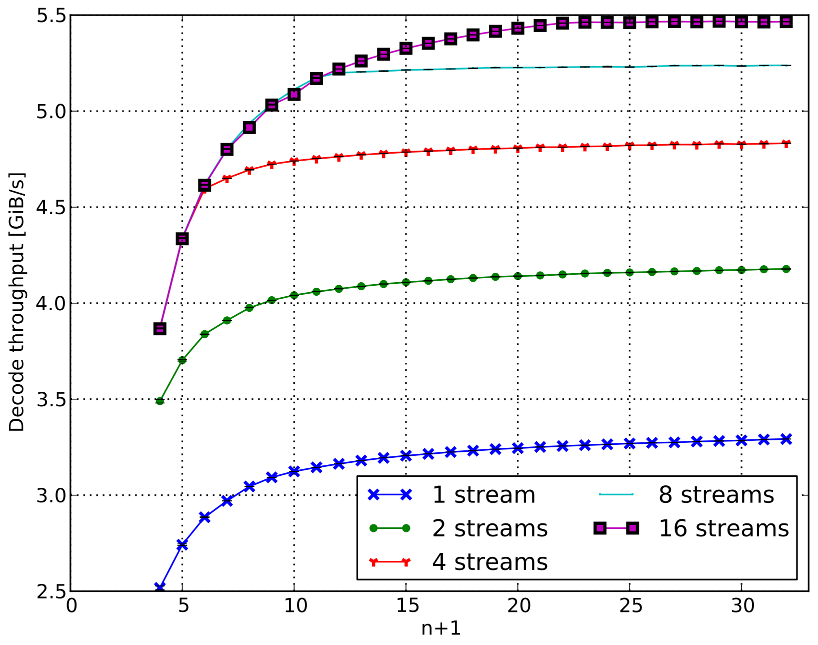


Figure 6.18.: Reconstruction of a single device with the full number of loop cycles on the CPU (top) and the GPU (bottom).

6.7. Algebraic signatures

The implementation presented in [117] already eliminates the logarithm look-up table entirely and also avoids one memory reference. Since the power series practically traverses the entire field, no restrictions on the distribution of the MSSB can be made. The adaption of the polynomial multiplication for the algebraic signature is therefore not worthwhile. The single core CPU performance for $\mathrm{sig}_{\beta,1}$ of the cited algorithm is

$$1754.6 \pm 18.5\text{MiB/s}. \tag{6.7}$$

This is sufficient to complement the erasure coding. Depending on the usage scenario, the signature calculation requires the same data as the erasure coding and can therefore benefit from temporal and spatial locality if both are properly interleaved.

6.8. Related work

The use of GPUs for Reed-Solomon coding with a log/inv_log-based multiplication has been demonstrated by Curry et al. [125]. Brinkmann and Eschweiler have demonstrated the use of GPUs for coding tasks from within the Linux Kernel [126]. Optimization of Galois field arithmetic has also been studied by Greenan et al. [127]. The authors propose the use of composite fields where multiplication in $GF((2^r)^2)$ is performed with a series of multiplications in $GF(2^r)$ (which still use table lookups). They observe a large influence of the cache size and the frequent cache eviction of the tables in real application scenarios. Huang and Xu [128] show improvements of the table-based multiplications by augmentation which increases the table sizes. An implementation of the Cauchy bit-matrix representation of the Reed-Solomon generator matrix has been proposed by Plank and Xu [129]. As mentioned in Chapter 4, in this scheme the polynomial multiplication is transformed into a larger sequence of XOR (and AND) operations. A performance comparison has been performed with the Jerasure Cauchy-Reed-Solomon coding library [122] along with the classical table-based implementation.

6.9. Outlook to upcoming vector instructions

The most current vector instructions are the *Advanced Vector Extensions (AVX)* in version 1 [130, 131]. Their vector registers are 256-bit wide and would therefore allow to compute 16 coding symbols at a time. Unfortunately, the current AVX version does not include the integer vector instructions necessary for the polynomial multiplication (for instance parallel XOR, AND, Shift). These instructions are announced for the second version of AVX and some compilers are already able to generate code with these instructions. Since CPUs that execute these instructions are not yet available[1] it is only possible to run the code with the help of an emulator. To demonstrate the benefit of the wider vector

[1]The first processor with confirmed support is an Intel CPU codenamed “Haswell”, announced for release in 2013.

registers, the polynomial multiplication has been implemented for AVX version 2 and profiled with the software development emulator by Intel [132, 103]. Results of the emulated multiplication are shown in Figure 6.19. To encode the identical amount of data only 54.3 % of overall instructions are required. While this does not translate directly into performance, it indicates the possible benefit of wider vector units for the particular task of low-MSSB encoding.

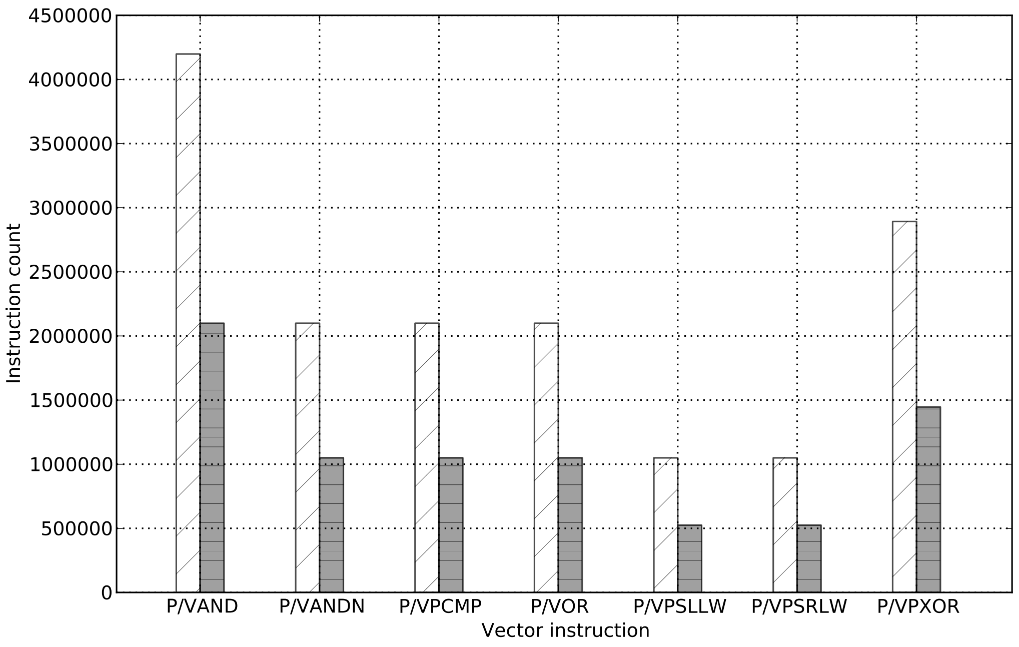


Figure 6.19.: Frequency of the relevant vector instructions for the SSE implementation (light bars) and the corresponding AVX2 implementation (dark bars) for the same amount of encoded data.

6.10. Summary

The performance of Galois field multiplication based on a polynomial multiplication algorithm (instead of using pre-computed lookup tables) has been studied. Therefore, a vectorized implementation for the Streaming SIMD units of modern x86 processors has been developed. Depending on the distribution of the polynomial coefficients in the input data, the performance can be improved by a factor of up to 10 compared to the table-based implementation. The multiplication has also been implemented as a GPU kernel using the CUDA framework and it was benchmarked against a table-based multiplication kernel. Both approaches are able to reach the limit imposed by the communication cost. A suitable choice of low-MSSB generator matrices reduces the average number of loop cycles significantly. A set of low-MSSB matrices has been identified and used in a novel coding scheme. The SIMD implementation achieves between 80% and 20% of the performance of an identical multi-parity scheme and is up to a factor of 2.8 faster than the corresponding alternative implementations. The GPU implementation of the coding scheme is up to a factor of 7.5 faster than the corresponding classical table-based implementation. For certain sets of parameters the coding calculations can be completely hidden behind the data transfers. This is the optimal case in which the overall throughput is limited by the PCI bandwidth only. Both approaches promise to scale with future developments: Compute performance of GPUs as well as PCI Express bandwidth will increase much faster than memory access latencies. Utilization of multiple GPUs with independent PCI links is also feasible, since matrix rows are independent. In the SIMD case the wider vector units are already included in upcoming processor designs and are the dominant contributor to a higher per-core performance.

7. Summary and Conclusion

Fault-tolerance and erasure-resilience are critical properties for future large-scale storage systems at peta- and exascale, easily exceeding tens of thousands of individual storage components. In this thesis a novel erasure-resilient and compute-efficient coding scheme has been presented. Instead of trying to accommodate existing, computationally expensive coding schemes to modern processor architectures, an architecture-centric coding scheme has been designed, aiming at the highest performance on modern multi-core processors and GPGPU co-processors. It exploits specific features of these architectures, such as the vector units in modern CPUs and the exceptionally high number of in-order execution units in GPGPUs. The basis of the coding scheme is a polynomial multiplication algorithm which replaces traditional lookup-tables with a sequence of (vectorizable) logical and shift instructions to perform the multiplication over a finite field. By replacing the memory accesses, the ever increasing memory latency bottleneck is alleviated and the problem is moved into the computational domain. The length of the instruction sequence is dependent on the distribution of the binary polynomial coefficients in the factors: the lower the position of the most significant set bit (MSSB) is in one of the factors, the shorter is the instruction sequence for the multiplication[1]. The encoding step of a linear block code consists of a multiplication of the generator matrix with the actual data symbols. By choosing a generator matrix whose elements are suitable in terms of the distribution of their polynomial coefficients, the performance of the coding scheme can be greatly improved. Unsurprisingly, the choice of the generator matrix is not arbitrary. It must possess certain important algebraic properties. As has been shown in this thesis, the traditional algebraic construction methods for the generator matrices do not produce matrices with the desired distribution of polynomial coefficients in their elements. Therefore, several heuristics for finding better generator matrices (in terms of computational efficiency) have been developed. Suitable generator matrices for a wide range of parameters that have been discovered using these heuristics are documented in the appendix. The use of a Monte-Carlo heuristic offers an important advantage: It allows the creation of special purpose generator matrices with specific shapes. Usage scenarios are, for example, generator matrices with exceptionally low computational cost for the first check symbol (similar to standard RAID), but with increasing cost for the following symbols. Such matrices would be suitable for creating a 1-error fault-tolerance very quickly and a deferred calculation of the higher degrees of fault-tolerance (for instance as a background activity during system idle times). In this thesis it has been shown, that for the decoding matrices that are obtained from the carefully selected generator matrices, the low average MSSB property is not preserved. However, since the most common case is a single device reconstruction, the coding scheme still performs very well, due to the reduced number

[1]In fact, the multiplication algorithm can also be optimized for a high position of the least significant set bit (LSSB).

of processed symbols. Furthermore, depending on the usage scenario, it opens the possibility to deploy one or more dedicated decoding agents equipped with powerful GPGPU accelerators and network links, that provide a high-performance on-the-fly or background reconstruction service.

The performance benchmarks presented in this thesis stress the observation, that the memory subsystem has become one of the main bottlenecks in commodity systems today. Consequently, it is of great importance to avoid accessing large in-memory data structures from inner loop code. This thesis also shows that it is feasible to create a Reed-Solomon based coding scheme which efficiently utilizes the vector units of modern commodity processors. This is the most cost-effective approach, since even entry-level commodity CPUs are equipped with the required vector units and instruction sets. Furthermore, it suggests that modern GPGPUs can be used as high-performance coding co-processors, for many configurations effectively limited by the bandwidth of the current PCI Express interconnect only. Therewith the coding throughput becomes comparable to the single-link bandwidth of modern data center and high-performance interconnects, such as 10GbE/40GbE or QDR/FDR InfiniBand. This is an important perquisite for the widespread adoption of sophisticated coding techniques in storage systems.

Both approaches were chosen with the anticipated future developments of microprocessor architectures in mind: the roadmaps of commodity CPUs show a steady increase of the width of the vector units and continuous enhancements of their vector instruction sets. Clock frequencies, as well as memory and cache performances on the other hand are not expected to increase significantly. The same holds true for GPGPUs: The number of small parallel execution units is growing considerably with every generation, while the memory latency is only slightly improving. A great leap forward for the use of GPGPUs as a coding co-processors is the upcoming transition to PCI Express 3.0, which effectively doubles the host-to-GPGPU bandwidth compared to the previous revision of PCI Express. The presented concepts could also benefit from the progressing consolidation of CPU and GPGPU into a single-die device, sometimes referred to as *accelerated processing unit (APU)*. APUs promise improved data transfer rates, lower power consumption, and an increased cost effectiveness, recommending themselves as accelerators for storage systems build of commodity components.

Directions for future work include improved techniques for obtaining the generator matrices. In general, a many-core framework for finding or creating generator matrices with the help of accelerator devices could be very useful for creating an exhaustive catalog of matrices. As already outlined in Chapter 6.9, the multiplication algorithm can be easily adapted to future vector instruction sets. Therefore, the library will be extended with every new processor generation to ensure highest achievable performance. Finally, some research efforts have to be directed towards the actual integration into large scale storage systems. Studies on how to use these kind of sophisticated erase-resilient coding schemes within modern parallel file systems, such as Lustre or FraunhoferFS, have already begun.

8. Bibliography

[1] M. Hilbert and P. Lopez, “The world’s technological capacity to store, communicate, and compute information,” *Science*, vol. 332, no. 6025, pp. 60–65, February 2011.

[2] P. Lyman and H. R. Varian, “How much information 2003?” School of Information Management and Systems at the University of California at Berkeley, Tech. Rep., 2003.

[3] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, “The diverse and exploding digital universe,” IDC, Tech. Rep., March 2008.

[4] J. Gantz and D. Reinsel, “Extracting value from chaos,” IDC, Tech. Rep., June 2011.

[5] European organization for nucelar research. [Online]. Available: http://cern.ch

[6] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” *ACM SIGMOD Record*, vol. 17, no. 3, pp. 109–116, 1988.

[7] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[8] J. S. Plank, “A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems,” *Software Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[9] U. Drepper, “What Every Programmer Should Know About Memory,” 2007. [Online]. Available: http://www.akkadia.org/drepper/cpumemory.pdf

[10] S. Lloyd, “Computational capacity of the universe,” *Phys. Rev. Lett.*, vol. 88, p. 237901, May 2002.

[11] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[12] Timm M. Steinbeck, “Commissioning and First Experiences of the ALICE High Level Trigger,” *J. Phys.: Conf. Ser.*, vol. 219, 2010.

[13] G. Brumfiel, “Down the petabyte highway,” *Nature*, vol. 469, pp. 282–283, 2011.

[14] M. Nieto-Santisteban, Y. Simmhan, R. Barga, L. Dobos, J. Heasley, C. Holmberg, N. Li, M. Shipway, A. S. Szalay, C. van Ingen, and S. Werner, “Pan-STARRS: Learning to Ride the Data Tsunami,” in *Microsoft eScience Workshop*, 2008.

[15] L. Clarke, X. Zheng-Bradley, R. Smith, E. Kulesha, C. Xiao, I. Toneva, B. Vaughan, D. Preuss, R. Leinonen, M. Shumway, S. Sherry, P. Flicek, and T. . G. P. Consortium, “The 1000 genomes project: data management and community access,” *Nature Methods*, vol. 9, no. 5, pp. 459–462, 2012.

[16] J. Dongarra, “Impact of architecture and technology for extreme scale on software and algorithm design,” Euro-Par 2010 keynote, 8 2010.

[17] D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[18] Data Center Knowledge (DCK). Report: Google Uses About 900,000 Servers. [Online]. Available: http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/

[19] ——. Estimate: Amazon Cloud Backed by 450,000 Servers. [Online]. Available: http://www.datacenterknowledge.com/archives/2012/03/14/estimate-amazon-cloud-backed-by-450000-servers/

[20] Storage Newsletter. Amazon S3 Holds 566 Billion Objects. [Online]. Available: http://www.storagenewsletter.com/news/cloud/amazon-s3-holds-566-billion-objects

[21] Facebook, Inc. (2012) Form S-1 Registration Statement. [Online]. Available: http://www.sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm

[22] ——. (2012, July) Facebook newsroom - key facts and infrastrucure. [Online]. Available: http://newsroom.fb.com/

[23] L. Null and J. Lobur, *The essentials of computer organization and architecture*. Jones and Bartlett, 2006.

[24] Oracle Corporation. Tape storage solutions. [Online]. Available: http://www.oracle.com/us/products/servers-storage/storage/tape-storage/

[25] Cost of hard drive storage space. [Online]. Available: http://ns1758.ca/winch/winchest.html

[26] Z. Kerekes. (2012) Charting the rise of the solid state disk market. [Online]. Available: http://www.storagesearch.com/chartingtheriseofssds.html

[27] Magnetic disk heritage center. [Online]. Available: http://www.magneticdiskheritagecenter.org/

[28] L. D. Stevens, W. A. Goddard, and J. J. Lynott, “Data storage machine,” United States Patent US3134097, 1964.

[29] R. E. Bohn and J. E. Short, “How much information? 2009 report on american consumers,” Global Information Industry Center University of California, San Diego, Tech. Rep., January 2010.

[30] C. Walter, “Kryder’s law,” *Scientific American*, pp. 32–33, August 2005.

[31] M. H. Kryder and C. S. Kim, “After hard drives-what comes next?” *IEEE Transactions On Magnetics*, vol. 45, no. 10, 2009.

[32] K. G. Ashar, *Magnetic Disk Drive Technology - Heads, Media, Channel, Interfaces, and Integration*. IEEE Press, 1997.

[33] SNIA. Storage networking and information management primer. [Online]. Available: http://www.snia.org/education/storage_networking_primer

[34] H.-P. Messmer, *PC Hardwarebuch . Aufbau, Funktionsweise, Programmierung*, 6th ed. Addison Wesley, 2000.

[35] R. Wood, Y. Hsu, and M. Schultz, “Perpendicular magnetic recording technology,” Hitachi Global Storage Technologies, Tech. Rep., 2007.

[36] Seagate, “Seagate Reaches 1 Terabit Per Square Inch Milestone In Hard Drive Storage With New Technology Demonstration,” Tech. Rep., 2012.

[37] M. Albrecht, J.-U. Thiele, and A. Moser, “Terabit-Speicher – bald Realitaet oder nur Fiktion?” *Physik Journal*, vol. 2, no. 10, 2003.

[38] W. Whittington, “Desktop, Nearline and Enterprise Disk Drives,” SNIA Tutorial, 2007.

[39] M. Fox, “End-to-end data protection using T10 standard data integrity field End-to-end data protection using T10 standard data integrity field,” IBM developerWorks, 2012.

[40] Western Digital. Enterprise hard drives overview. [Online]. Available: http://wdc.com/en/products/internal/enterprise/

[41] Seagate. Enterprise SSD and HDD Drives. [Online]. Available: http://www.seagate.com/www/en-us/products/enterprise-ssd-hdd/

[42] H. Charap, P. ling Lu, and Y. He, “Thermal stability of recorded information at high densities,” *IEEE Transactions on Magnetics*, vol. 33, no. 1, pp. 978–983, 1997.

[43] E. Spanjer, “Flash management – why and how?” Smart Modular Technologies, Tech. Rep., 2009.

[44] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, “Introduction to flash memory,” *Proceedings of the IEEE*, vol. 91, no. 4, 2003.

[45] N. Ekker, T. Coughlin, and J. Handy, “Solid State Storage 101 - An Introduction to Solid State Storage,” SNIA, Tech. Rep., January 2009.

[46] Intel Corporation, “Intel Solid-State Drive 910 Series Product Specification,” Tech. Rep., 2012.

[47] Hitachi Global Storage Technologies, “UltrastarTM SSD400M Specification - Enterprise Solid State Drives,” Tech. Rep., 2012.

[48] S. Aritome, R. Shirota, G. Heminek, T. Endoh, and F. Masuoka, "Reliability issues of flash memory cells," *Proceedings of the IEEE*, vol. 81, no. 5, 1993.

[49] Y. Chen, "Flash Memory Reliability NEPP 2008 Task Final Report," Jet Propulsion Laboratory, California Institue of Technology, Tech. Rep., 2009.

[50] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," *Proceedings of the 10th USENIX conference on file and storage technologies*, 2012.

[51] D. Gilbert. The Linux SCSI Generic (sg) HOWTO. [Online]. Available: http://tldp.org/HOWTO/SCSI-Generic-HOWTO/

[52] IBM Corporation. (2002) Hard disk drive specifications Ultrastar 73LZX.

[53] E. Bauer, X. Zhang, and D. A. Kimber, *Practical System Reliabilty*. John Wiley & Sons, 2009.

[54] J. W. McPherson, *Reliability Physics and Engineering*. Springer, 2010.

[55] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, 1994.

[56] K. Rao, J. L. Hafner, and R. A. Golding, "Reliabilty for networked storage nodes," IBM Research Division, Tech. Rep., 2005.

[57] J. L. Hafner and K. Rao, "Notes on Reliabilty Models for Non-MDS Erasure Codes," IBM Research Division, Tech. Rep., 2006.

[58] J. Gray and C. van Ingen, "Empirical measurements of disk failure rates and error rates," Microsoft Research, Microsoft Research Technical Report MSR-TR-2005-166, 2005.

[59] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[60] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained," in *FAST '08: 6th USENIX Conference on File and Storage Technologies*, 2008.

[61] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, June 2007.

[62] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies FAST*. USENIX Association, 2007, pp. 1–16.

[63] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[64] J. G. Elerath and M. Pecht, "A highly accurate method for assessing reliability of redundant arrays of inexpensive disks (raid)," *IEEE Transactions on Computers*, vol. 58, no. 3, 2009.

[65] J. G. Elerath, “Reliability Model And Assessment of Redundant Arrays of Inexpensive Disks (Raid) Incorporating Latent Defects and Non-Homogeneous Poisson Process Events,” Ph.D. dissertation, University of Maryland, College Park, 2007.

[66] G. Cole, “Estimating drive reliability in desktop computers and consumer electronics systems,” Seagate Personal Storage Group, Tech. Rep., 2000.

[67] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, “Are disks the dominant contributor for storage failures?” in *FAST ’08: 6th USENIX Conference on File and Storage Technologies*, 2008.

[68] B. Schroeder, S. Damouras, and P. Gill, “Understanding latent sector errors and how to protect against them,” *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, 2010.

[69] The Storage Networking Industry Association. SNIA dictionary. [Online]. Available: http://www.snia.org/education/dictionary/

[70] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, “Row-diagonal parity for double disk failure correction,” *Proceedings of the Third USENIX Conference on File and Storage Technologies*, 2004.

[71] M. Blaum, J. Brady, J. Bruck, and J. Menon, “EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures,” *Computers IEEE Transactions on*, vol. 44, no. 2, pp. 192–202, 1995.

[72] S. Müller, “Implementierung des EvenOdd-Algorithmus für das ClusterRAID,” Kirchhoff-Institut für Physik (KIP), Universität Heidelberg, Projektpraktikum, 2005.

[73] H. P. Anvin, “The mathematics of RAID-6,” Tech. Rep., 2004, 2011.

[74] Oracle Corporation. Oracle solaris zfs data sheet.

[75] A. Leventhal. Triple-Parity RAID-Z. [Online]. Available: https://blogs.oracle.com/ahl/entry/triple_parity_raid_z

[76] btrfs Wiki. [Online]. Available: https://btrfs.wiki.kernel.org/index.php/Main_Page

[77] Sun Microsystems Inc., “Lustre file system - high-performance storage architecture and scalable cluster file system,” Tech. Rep., 2007.

[78] Oracle Corporation. Lustre File System Operations Manual - Version 2.0.

[79] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[80] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” *SOSP ’03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[81] The Apache Software Foundation. Hadoop Distributed File System (HDFS). [Online]. Available: http://hadoop.apache.org/hdfs/

[82] ISBN.org FAQ. [Online]. Available: http://www.isbn.org/standards/home/isbn/us/isbnqa.asp

[83] D. R. Hankerson, D. G. Hoffman, D. A. Leonard, C. C. Lindner, K. T. Phelps, C. A. Rodger, and J. R. Wall, *Coding Theory and Cryptography, The Essentials*, second edition, revised and expanded ed. Marcel Dekker, Inc., 2000.

[84] T. K. Moon, *Error Correction Coding, Mathematical Methods and Algorithms*. Wiley, 2005.

[85] S. B. Wicker and V. K. Bhargava, Eds., *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994.

[86] R. Bose and D. Ray-Chaudhuri, "On a class of error-correcting binary codes," *Information and Control*, vol. 3, no. 68-79, 1960.

[87] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres (Paris)*, vol. 2, 1959.

[88] R. Chien, "Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 10, no. 4, 1964.

[89] G. D. Forney, "On decoding BCH codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, 1965.

[90] E. R. Berlekamp, "Nonbinary BCH Decoding," *IEEE Transactions on Information Theory*, vol. 14, 1968.

[91] J. L. Massey, "Shift-Register Synthesis and BCH Decoding," *IEEE Transactions on Information Theory*, vol. 15, 1969.

[92] L. Mirsky, *An Introduction to Linear Algebra*. Oxford at the Clarendon Press, 1955.

[93] J. Plank and Y. Ding, "Correction to the 1997 Tutorial on Reed-Solomon Coding," 2003.

[94] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckermann, "An xor-based erasure-resilient coding scheme," ICSI Technical Report TR-95-048, Tech. Rep., 1995.

[95] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, April 1989.

[96] S. Kalcher and V. Lindenstruth, "Accelerating galois field arithmetic for reed-solomon erasure codes in storage applications," *IEEE International Conference on Cluster Computing, © 2011 IEEE*, pp. 290–298, 2011.

[97] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[98] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[99] C. Lameter, "Local and Remote Memory: Memory in a Linux/NUMA System," Silicon Graphics, Inc., Tech. Rep., 2006.

[100] Intel Corporation, "Detecting memory bandwidth saturation in threaded applications," Tech. Rep., 2010.

[101] Intel Corporation, "SSE4 programming reference," Tech. Rep., 2007.

[102] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[103] Firasta Nadeem, Buxton Mark, Jinbo Paula, Nasri Kaveh, and Kuo Shihjong, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," 2008.

[104] B. J. Gimmestad, "The Russian Peasant Multiplication Algorithm: A Generalization," *The Mathematical Gazette*, vol. 75, no. 472, pp. 169 – 171, 1991.

[105] National Institute of Standards and Technology (NIST), "FIPS-197: Advanced Encryption Standard," 2001.

[106] Intel Corporation. (2007) Intel C++ Intrinsics Reference. [Online]. Available: http://softwarecommunity.intel.com/isn/downloads/softwareproducts/pdfs/347603.pdf

[107] S. Kalcher, "Optimization of a distributed fault-tolerant mass storage system for clusters," Diplomarbeit, Ruprecht-Karls-Universität Heidelberg, Germany, December 2004.

[108] NVIDIA Corporation, "NIVIDIA CUDA: Compute Unified Device Architecture Programming Guide," 2007.

[109] Khronos Group, "OpenCL Specification," 2009.

[110] NVIDIA Corporation, "Technology Overview: NVIDIA GeForce GTX 680," Tech. Rep., 2012.

[111] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, p. 40, 2008.

[112] PLX Technology Inc., "Choosing PCI Express Packet Payload Size," PLX Technology, Inc., Tech. Rep., 2007.

[113] E. W. Weisstein. Primitive polynomial. [Online]. Available: http://mathworld.wolfram.com/PrimitivePolynomial.html

[114] The On-Line Encyclopedia of Integer Sequences. A011260 Number of primitive polynomials of degree n over GF(2). [Online]. Available: http://oeis.org/A011260

[115] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes - The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007.

[116] T10 Data Integrity Field CRC16 calculation. [Online]. Available: http://lxr.linux.no/linux+v3.3.6/lib/crc-t10dif.c

[117] Q. Xin, E. L. Miller, T. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin, “Reliability mechanisms for very large storage systems,” *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS’03)*, 2003.

[118] W. Litwin and T. Schwarz, “Algebraic signatures for scalable distributed data structures,” *ICDE ’04 Proceedings of the 20th International Conference on Data Engineering*, 2004.

[119] T. Schwarz, R. W. Bowdidge, and W. A. Burkhard, “Low cost comparison of file copies,” *In Proc. of the 10th Int. Conf. on Distributed Computing Systems*, 1990.

[120] Z. Smith. Bandwidth: a memory bandwidth benchmark. [Online]. Available: http://zsmith.co/bandwidth.html

[121] Intel Corporation, “Using the RDTSC Instruction for Performance Monitoring,” 1997.

[122] J. S. Plank, S. Simmerman, and C. D. Schuman, “Jerasure: A library in c/c++ facilitating erasure coding for storage applications - version 1.2,” University of Tennessee, Tech. Rep., 2008.

[123] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. [Online]. Available: http://openmp.org

[124] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,” *ISCA ’10 Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[125] M. L. Curry, A. Skjellum, and R. Brightwell, “Accelerating Reed-Solomon coding in RAID systems with GPUs,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–6.

[126] A. Brinkmann and D. Eschweiler, “A microdriver architecture for error correcting codes inside the linux kernel,” *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[127] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, “Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications,” in *2008 IEEE International Symposium on Modeling Analysis and Simulation of Computers and Telecommunication Systems*, 2008, pp. 1–10.

[128] C. Huang and L. Xu, “Fast software implementation of finite field operations,” *Technical report*, 2003.

[129] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *NCA2006 5th IEEE International Symposium on Network Computing Applications*, 2006, pp. 173–180.

[130] Intel Corporation, "Intel architecture instruction set extensions programming reference," Intel Corporation, Tech. Rep., 2012. [Online]. Available: http://software.intel.com/en-us/avx/

[131] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions," 2009.

[132] Intel Corporation, "Intel Software Development Emulator," Intel Corporation, Tech. Rep. [Online]. Available: http://software.intel.com/en-us/articles/intel-software-development-emulator/

A. Appendix

A.1. The Galois field $GF(2^{16})$

Table A.1.: Hexadecimal representation of the primitive generator polynomials of $GF(2^{16})$

1	2	3	4	5	6	7	8	9	10	11	12
1002d	10039	1003f	10053	100bd	100d7	1012f	1013d	1014f	1015d	10197	101a1
101ad	101bf	101c7	10215	10219	10225	1022f	1025d	1026d	10285	10291	102a1
102e5	1031d	1034b	10369	10371	10387	1038d	1039f	103a3	103dd	103f9	10429
10457	10467	10483	10489	10491	104bf	104c1	10533	10547	10569	10587	105c3
105dd	105eb	10641	1064b	10653	1068b	106c3	1076b	1076d	10779	10783	107f1
1080d	10861	108bf	108d5	108df	108e3	108f1	108fb	10939	1097d	1098b	109a5
109af	109c3	109c5	109e7	109f3	10a7d	10a81	10abb	10ac5	10b01	10b13	10b15
10b51	10b5d	10bcd	10bd3	10be5	10c0f	10c1d	10c21	10c69	10c71	10c7b	10c8d
10c95	10ca3	10caf	10cf3	10d43	10d83	10d8f	10dd3	10de5	10e45	10e85	10e9d
10ea7	10f05	10f09	10f1b	10f69	10f77	10f87	10f95	10fb1	10fcf	10fdb	1100b
11043	1104f	1105d	1107f	11083	11085	11097	1111d	1116f	111c9	111db	111f3
11241	11253	11263	11277	11281	1128b	112af	112c5	112db	112ed	112f3	1133b
1133d	11375	11379	1139d	11409	1141b	1144d	11487	11493	114af	114c9	114d1
114ed	114f9	11507	1158f	115a1	115ab	115d5	115fb	1161f	11637	1163d	11643
11651	1169b	116ab	116d9	116fd	1173f	11747	11753	11799	117a3	117a9	117d7
1186f	1188d	11893	118a9	118bb	118f9	118ff	11923	11925	1196b	11975	11979
119b3	119e5	119fd	11a07	11a57	11a6b	11a89	11ae3	11ae5	11b09	11b2b	11b47
11b4b	11baf	11bbb	11bbd	11bc3	11bd7	11be1	11c29	11c61	11c85	11c9d	11cb5
11ccb	11ccd	11ce9	11cef	11cf1	11d17	11d3f	11d53	11d59	11d5f	11d65	11d77
11d81	11d87	11dff	11e21	11e3f	11e5f	11e71	11e7b	11e99	11e9f	11ec3	11ecf
11ee7	11ef3	11f29	11f3b	11f57	11f75	11f83	11f9b	11fb9	11fbf	11fc1	1203d
12051	1208f	1209b	120b9	120c7	12105	12187	12195	121a9	121b1	121e1	12227
12235	12241	12253	1228b	12293	122af	122b1	12329	12345	12357	1236d	123e3
1240f	1242d	12439	1244b	12499	1249f	124e7	124ff	12515	12557	1255d	12573
12579	1259b	125e5	12607	12615	12623	1263d	12651	1267f	12683	126fb	12711
12739	12759	12763	12787	127a5	127c5	127ed	1281d	12841	12871	1287b	128a9
128c5	128f5	1290b	12919	12943	12945	12983	12989	1299d	129b9	129c7	129d5
129f1	12a25	12a29	12a31	12a3b	12a7f	12a85	12aa7	12acd	12ad9	12b09	12b27
12b53	12b7b	12b7d	12b99	12bd1	12bdb	12c1f	12c3b	12c4f	12c5d	12c61	12c6b
12c73	12c79	12cb9	12ce3	12cf7	12cfb	12d35	12d41	12d65	12d81	12ddb	12de1
12e0f	12e47	12e4b	12e4d	12e71	12ec5	12ef9	12eff	12f3d	12f43	12f5b	12f67
12fa7	12fd3	13017	13027	13077	13081	130af	130c3	130dd	130ed	130f5	13107
1310d	13129	13173	131a7	131ab	131cd	131d9	1322f	13237	1323d	13249	13257
1325b	13275	1329d	132b5	132d9	132ef	132f1	13305	1331b	13339	1339f	133a9
133c5	133d1	133f3	13415	13425	1349b	134fd	13505	1351d	1354b	13571	1357d
135a3	135c9	135d7	1361b	13635	1366f	13699	136a9	136c3	136d1	136ed	13707

1370b	13719	13743	137cd	137e3	13801	1380d	1381f	13837	13849	1385d	1386d
138c7	138cb	138df	138fb	13921	1393f	1394d	13953	1395f	13965	13999	139b7
139bd	139c9	139ff	13a09	13a1b	13a55	13a65	13a69	13a77	13abd	13acf	13af3
13b29	13b45	13b5b	13b9d	13bcb	13bd3	13bef	13bf1	13c47	13c4b	13c69	13c6f
13c93	13d07	13d13	13d23	13d31	13d3d	13d43	13d49	13d67	13d6b	13d83	13d91
13db3	13dc1	13dcd	13e31	13e43	13e45	13e51	13e57	13e8f	13ed3	13edf	13ee9
13efb	13f1d	13f81	13fa5	13fc5	14075	1409d	140cd	14109	1413f	1414b	14153
14159	14193	14199	141af	141bb	141bd	141e1	141e7	141f3	141ff	14203	14211
1424b	14271	14281	142a9	142cf	142e1	142ed	1430b	14343	1437f	14383	143bf
143ef	143fb	1440f	14447	14477	14495	1449f	144a3	144bb	144c5	144e1	144f9
1450b	14529	14537	14573	14589	1458f	1459b	1459d	145b5	145b9	145f7	145fd
14629	14645	1465d	14673	14691	146a1	146cb	146d5	146df	146e9	146fd	14721
1477d	14793	14799	147c3	147c9	147f9	1481d	14835	1483f	14859	14881	148a9
148c3	148db	148ff	1490b	14915	1492f	14931	149df	149e5	149fb	14a13	14a15
14a2f	14a6d	14a75	14aad	14ab5	14acb	14adf	14af7	14afb	14b21	14b4d	14b63
14b7d	14bb7	14bc9	14bcf	14bdd	14bf3	14bf9	14bff	14c1f	14c67	14c73	14ca7
14cb9	14cc7	14cf7	14cfd	14d0f	14d39	14d4d	14d5f	14d69	14d71	14d8b	14d8d
14de7	14deb	14df3	14e4b	14e6f	14e81	14e87	14ea3	14eaf	14eb1	14eeb	14ef5
14f25	14f31	14f49	14f57	14f5b	14f61	14f6d	14f8f	14fa1	14fe3	15003	1500f
15059	15081	150a5	150b7	150bb	150c9	150d7	150eb	15107	15125	15149	1515d
1516d	151a1	151a7	151d3	151df	1520b	15245	1525d	15261	15297	15309	15327
15363	15377	153e1	15457	1545b	1547f	15483	154b5	154b9	154df	15527	1552d
1554b	15593	155af	155f5	15621	1562b	1562d	15647	1569f	156c5	156f5	15729
15751	157ab	15807	15825	1583d	15889	158b3	158d3	158d9	158df	1592b	1594b
1594d	15969	1598d	15993	1599f	159a3	159cf	159d7	159ff	15a55	15a71	15a7d
15a99	15aa5	15ab7	15ad7	15b45	15b4f	15b57	15bab	15bc7	15bdf	15c05	15c99
15ca3	15ca5	15ceb	15d07	15d31	15d37	15d3b	15d7f	15d91	15dab	15dc7	15df1
15dfd	15e0b	15e19	15e29	15e37	15e57	15e67	15eab	15ed5	15ee5	15f55	15f8d
15fbd	16021	16039	16063	16077	1609f	160dd	1610b	16119	1613d	16143	16167
1617f	161ad	161d3	16213	16231	1623b	16261	1626b	16273	1628f	16297	1632d
16333	16335	16353	16365	1637b	16381	163ed	163f5	16407	1641f	1643b	16443
16451	16497	1649b	164e9	164f7	1650f	1652b	16535	16539	1653f	16565	16593
165a5	165e7	165f3	16605	16671	166a9	166bb	166d7	16707	16719	1676b	16779
167a1	167bf	167d9	167df	167f7	16801	1680b	1681f	16849	1689d	168cb	168d3
168d5	16917	16947	16955	1698d	169af	169cf	169d7	16a77	16aa5	16ae7	16b01
16b0d	16b23	16b57	16b6b	16b8f	16b9d	16bab	16bc7	16bfb	16c03	16c1d	16c39
16c6f	16c81	16c93	16ca5	16ce7	16d15	16d3b	16d43	16d89	16d9d	16db3	16dbf
16dc1	16de5	16e19	16e51	16e5d	16e67	16e7f	16e85	16e91	16e97	16e9b	16ed9
16efd	16f47	16f8d	16fc9	16fdb	16fff	17025	17029	17057	1705b	17061	1706b
1706d	1709d	170b3	170fd	17111	17159	1715f	1716f	17181	171c3	171dd	171f9
17205	1721d	1722d	1725f	17263	17271	17299	172a3	172af	172cf	172dd	172e1
17329	17345	1736d	17391	1739b	173ad	173b9	17411	17439	17469	1747d	17481
17487	17495	174b7	174c5	174d7	174ed	17501	17515	17523	17549	17583	1759b
175a1	175a7	1760d	17619	1763b	1763d	17643	17657	1769d	176c7	176df	1771b
1771d	17741	1774b	17781	177a5	177b7	17809	17835	1783f	17899	178c9	178d1
178db	178dd	178eb	17901	1790d	17979	1797f	17991	17997	179e9	179ef	17a01
17a1f	17aa7	17aab	17ab9	17af7	17b05	17b39	17b63	17bb1	17bbb	17bf5	17c37
17c4f	17c5d	17ca1	17cb5	17cbf	17ccb	17cd3	17d21	17d47	17d59	17d87	17da5
17da9	17dc5	17dd7	17e1d	17e2b	17e4b	17e59	17e65	17e8b	17e93	17eaf	17ebb
17ec5	17ed1	17eed	17f0b	17f2f	17f31	17f45	17f75	17fb3	17fc7	17fd3	18013
18015	1806d	18085	180f7	18117	1812b	181b7	18211	1821b	18241	18255	1828b
18293	1829f	182bb	182c9	182f3	182ff	18307	18329	18357	1835d	18361	18379

18385	183bf	183c1	183df	183f1	18411	1841b	18433	1844d	1846f	184af	184cf
184d1	184dd	184f9	1850d	18513	18529	18537	18561	1856d	18579	1857f	18585
185c7	185d9	185e9	185fb	1860b	18619	18625	18637	186ab	186c1	186d9	186f1
186f7	1870f	1871d	18721	18733	18741	18753	187af	187b1	187c5	18833	1885f
18863	188c9	1892f	18931	1895d	18979	1897f	1899b	189a7	189ad	189ef	189fb
18a45	18a61	18a67	18a73	18a75	18a9d	18ae5	18b35	18b59	18b63	18b81	18bb7
18bd1	18bdb	18c0d	18c23	18c91	18c97	18c9d	18ccb	18cef	18d27	18d95	18da3
18da5	18dbd	18dc9	18de7	18e21	18e47	18e69	18eb1	18f57	18f89	18fd3	18fd9
18fe5	18fef	19003	1900f	1901b	19027	19047	1908d	190a5	190e7	1911f	19143
1916b	19179	1919b	191a1	19231	19251	1926d	19279	19283	19289	192a7	192fd
19305	19317	19327	19335	1934d	19355	1939f	193c5	193db	193eb	193f3	19401
1943b	19473	19489	19491	194a7	194c1	19505	1951b	1952b	19539	19547	19571
1958b	1958d	195a9	195c3	195d1	1962d	19635	1965f	19677	1967d	196b7	196db
196f9	1970d	19715	1971f	19761	197a1	197ab	197b9	197e3	197e9	197fd	1980b
19823	1982f	19843	1990f	1992b	19941	19947	1998b	1998d	199c3	199ff	19a0f
19a1d	19a35	19a6f	19ad7	19b31	19b37	19b5b	19b6d	19b79	19b7f	19bcb	19bdf
19bef	19bfd	19c53	19c65	19c69	19c8d	19ca3	19cc5	19d19	19d45	19d49	19d8f
19df7	19e07	19e4f	19e57	19e61	19e83	19e91	19ea7	19eb9	19ec7	19edf	19ef1
19f05	19f11	19f21	19f4d	19f65	19f7b	19f93	19f99	19fa5	19fb7	1a011	1a02d
1a033	1a06f	1a07b	1a095	1a0c3	1a0cf	1a0f5	1a107	1a10d	1a125	1a129	1a137
1a145	1a185	1a1d9	1a1fd	1a23b	1a283	1a289	1a291	1a2bf	1a2c1	1a2fd	1a31b
1a321	1a32b	1a333	1a353	1a365	1a3cf	1a449	1a44f	1a467	1a479	1a485	1a4bf
1a4c1	1a4cb	1a4e5	1a4e9	1a4fd	1a505	1a517	1a527	1a52b	1a535	1a555	1a559
1a581	1a5b1	1a5b7	1a5bb	1a5dd	1a62d	1a639	1a64b	1a663	1a671	1a67d	1a6a5
1a6b7	1a6c5	1a71f	1a767	1a797	1a7b3	1a7b9	1a7bf	1a83b	1a8d5	1a8df	1a8fd
1a903	1a933	1a935	1a94b	1a94d	1a953	1a98b	1a9af	1a9b1	1a9e7	1aa77	1aa9f
1aabd	1aac3	1aad1	1aaf5	1ab19	1ab1f	1ab51	1ab57	1ab75	1aba7	1abad	1abb5
1abd3	1abd5	1ac1d	1ac27	1ac5f	1ac69	1ac8d	1acb1	1accf	1ad13	1ad31	1ad4f
1ad5b	1ad79	1adad	1adc1	1adcd	1adfb	1ae15	1ae3d	1ae75	1ae97	1ae9b	1aedf
1aee5	1af41	1af65	1af93	1afaf	1afff	1b013	1b03b	1b043	1b051	1b05b	1b07f
1b083	1b09b	1b0b9	1b0c7	1b0d9	1b0ef	1b153	1b177	1b18b	1b199	1b1bb	1b1dd
1b1e1	1b1e7	1b209	1b21b	1b24d	1b259	1b287	1b2d1	1b2db	1b2eb	1b2ed	1b313
1b323	1b345	1b349	1b34f	1b35d	1b37f	1b38f	1b39d	1b3f1	1b40f	1b41b	1b41d
1b455	1b46f	1b499	1b53b	1b56b	1b5b3	1b5b9	1b5bf	1b5c7	1b5e5	1b5e9	1b625
1b63d	1b64f	1b691	1b69b	1b6d3	1b6ef	1b711	1b769	1b76f	1b793	1b7a3	1b7a9
1b7d7	1b7e1	1b7ed	1b81b	1b82b	1b84d	1b853	1b869	1b88b	1b88d	1b8a9	1b8dd
1b8e7	1b92f	1b95b	1b96d	1b975	1b991	1b9f1	1b9f7	1ba15	1ba31	1ba45	1ba57
1ba83	1ba97	1baa1	1bacd	1bafd	1bb05	1bb1b	1bb27	1bb3f	1bb4b	1bb6f	1bbb1
1bbbd	1bbcf	1bc07	1bc0b	1bc29	1bc61	1bc67	1bcf1	1bcf7	1bd8d	1bda9	1bdaf
1bdf3	1be17	1be21	1be39	1be5f	1be69	1bea5	1bec9	1bef9	1bf23	1bf25	1bf2f
1bf37	1bf43	1bf51	1bf6b	1bf85	1bfa7	1bfad	1bfdf	1c035	1c04d	1c07b	1c0b1
1c0c9	1c0cf	1c0f3	1c10b	1c115	1c119	1c137	1c151	1c175	1c179	1c183	1c1cd
1c1d9	1c251	1c25d	1c267	1c29b	1c2df	1c2e5	1c309	1c341	1c371	1c37d	1c381
1c39f	1c3b7	1c3c9	1c3e1	1c413	1c445	1c479	1c4d5	1c4e3	1c4e9	1c52d	1c533
1c541	1c553	1c577	1c57d	1c5b1	1c5b7	1c5d1	1c5ed	1c609	1c61b	1c639	1c665
1c6dd	1c6f3	1c701	1c729	1c743	1c75b	1c775	1c7ad	1c7b5	1c7fd	1c813	1c819
1c837	1c867	1c86b	1c889	1c8df	1c94b	1c955	1c95f	1c963	1c96f	1c993	1c995
1c9a9	1c9bb	1ca53	1ca65	1ca93	1caa9	1cabd	1cae1	1cae7	1caf3	1cb15	1cb19
1cb1f	1cb23	1cb5d	1cbab	1cbe9	1cbfb	1cc27	1cc41	1cc4b	1cc65	1cc6f	1cc7b
1cc87	1cca3	1cced	1ccf5	1cd0d	1cd37	1cd57	1cd79	1cdcb	1cde9	1ce13	1ce3b
1ce49	1ce57	1ce6d	1cea7	1cead	1cebf	1cef1	1cf05	1cf1b	1cf21	1cf2b	1cf4d
1cf63	1cf65	1d019	1d0fb	1d103	1d10f	1d12d	1d14b	1d171	1d193	1d19f	1d1d7

1d211	1d24d	1d263	1d28d	1d295	1d2bb	1d2eb	1d2ed	1d301	1d33d	1d3cb	1d3ef
1d41d	1d441	1d455	1d499	1d4b1	1d4bb	1d4dd	1d4e7	1d4f3	1d4f5	1d4f9	1d52f
1d549	1d567	1d583	1d589	1d5ab	1d5ad	1d5b5	1d5bf	1d5e3	1d5e5	1d5ef	1d5f1
1d5f7	1d601	1d615	1d65d	1d6b3	1d6b5	1d6cd	1d717	1d72d	1d735	1d759	1d75f
1d77d	1d7b1	1d7b7	1d7d1	1d7db	1d80f	1d827	1d839	1d87d	1d899	1d8c3	1d8d1
1d8f5	1d907	1d90b	1d91f	1d943	1d945	1d94f	1d967	1d975	1d9b3	1d9f7	1d9fb
1da15	1da5d	1dab5	1dacb	1dad3	1db03	1db0f	1db39	1db47	1db4b	1db87	1dba3
1dba5	1dbd7	1dbdd	1dbf3	1dc0d	1dc19	1dc2f	1dc45	1dc91	1dcab	1dcad	1dcb9
1dcd3	1dd1b	1dd47	1dd71	1dd95	1dde1	1de03	1de4d	1de65	1de69	1de7b	1dea5
1debd	1dec3	1df37	1df3b	1df45	1df57	1df73	1dfcd	1dfef	1e013	1e015	1e037
1e045	1e049	1e05b	1e061	1e07f	1e0b3	1e0df	1e0e9	1e117	1e133	1e14d	1e165
1e1b7	1e1c3	1e209	1e25f	1e28d	1e2f9	1e345	1e351	1e361	1e373	1e39b	1e3ad
1e3e5	1e411	1e44b	1e469	1e47d	1e4db	1e4f3	1e501	1e537	1e56b	1e59b	1e5b5
1e5df	1e607	1e60b	1e62f	1e643	1e66b	1e685	1e69d	1e6b9	1e6f1	1e72d	1e735
1e78b	1e79f	1e7a5	1e7af	1e7bb	1e7e1	1e833	1e86f	1e877	1e881	1e899	1e8a5
1e8cf	1e901	1e923	1e925	1e93b	1e957	1e9fb	1e9fd	1ea19	1ea1f	1ea43	1ea51
1ea61	1ea89	1ea91	1ea9d	1eae5	1eaef	1eafd	1eb05	1eb21	1eb2b	1eb2d	1eb55
1eb7b	1eb9f	1ebb1	1ebc3	1ebcf	1ebeb	1ec0b	1ec2f	1ec31	1ec43	1ec5b	1ec67
1ec6d	1ec79	1ecb3	1ecd9	1ece5	1ed11	1ed1d	1ed27	1edbb	1eddb	1ee1b	1ee63
1ee71	1ee99	1eeaf	1eedb	1ef23	1ef3d	1ef57	1ef85	1ef97	1efb3	1efb9	1efdf
1efe3	1eff7	1f02d	1f039	1f04d	1f05f	1f065	1f069	1f0af	1f0bd	1f0d1	1f113
1f137	1f1a7	1f1ab	1f1cb	1f1d3	1f20d	1f245	1f249	1f27f	1f283	1f2ab	1f2d5
1f2df	1f2f1	1f317	1f335	1f381	1f387	1f393	1f399	1f3af	1f3cf	1f41f	1f423
1f46b	1f48f	1f49d	1f4d3	1f4f1	1f4fb	1f51d	1f527	1f539	1f565	1f571	1f5d7
1f60f	1f621	1f627	1f62b	1f635	1f639	1f655	1f687	1f69f	1f6a5	1f6c5	1f6dd
1f6eb	1f6f3	1f6f9	1f715	1f725	1f74f	1f783	1f7b3	1f7b5	1f7cd	1f7ef	1f7fb
1f801	1f825	1f83d	1f8f1	1f905	1f939	1f94d	1f971	1f9bb	1f9d1	1fa21	1fa41
1fa4b	1fa7d	1fa8b	1fae7	1fb01	1fb57	1fb5b	1fb6d	1fb7f	1fb83	1fb85	1fbcb
1fbcd	1fbf1	1fc0f	1fc11	1fc1b	1fc55	1fc9f	1fca9	1fcc9	1fced	1fd0d	1fd23
1fd3d	1fd43	1fd75	1fd85	1fd9b	1fdb3	1fdbf	1fe25	1fe31	1fe49	1fe83	1fee9

A.2. Low MSSB generator matrices

In the following sections low weight generator matrices for different values of k and n are listed. The matrices have been found with the algorithm described in Figure 5.18. The notation is as follows:

$$(n+k,k), \text{average MSSB position, maxelem} \tag{A.1}$$

Only the systematic part of the $n+k \times n$ generator matrix is printed, i.e. the first n rows contain the identity matrix:

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \\ g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{pmatrix} \tag{A.2}$$

A.2.1. k=2

(8,2), 0.5, 3

$$\left(\begin{array}{cccccc} 1 & 3 & 2 & 1 & 1 & 2 \\ \hline 1 & 1 & 1 & 3 & 2 & 3 \end{array}\right) \tag{A.3}$$

(12,2), 0.7, 5

$$\left(\begin{array}{cccccccccc} 2 & 4 & 5 & 1 & 3 & 2 & 1 & 3 & 1 & 1 \\ \hline 1 & 1 & 1 & 3 & 2 & 3 & 1 & 1 & 2 & 4 \end{array}\right) \tag{A.4}$$

(16,2), 0.89, 7

$$\left(\begin{array}{cccccccccccccc} 4 & 3 & 1 & 7 & 3 & 1 & 3 & 7 & 2 & 3 & 1 & 1 & 1 & 2 \\ \hline 1 & 2 & 2 & 2 & 7 & 7 & 4 & 1 & 3 & 1 & 1 & 5 & 3 & 1 \end{array}\right) \tag{A.5}$$

(20,2), 1.06, 7

$$\left(\begin{array}{cccccccccccccccccc} 4 & 1 & 3 & 2 & 3 & 1 & 7 & 5 & 6 & 1 & 1 & 2 & 4 & 6 & 4 & 1 & 2 & 7 \\ \hline 3 & 4 & 2 & 1 & 7 & 5 & 1 & 1 & 1 & 3 & 2 & 7 & 1 & 2 & 5 & 1 & 3 & 2 \end{array}\right) \tag{A.6}$$

(24,2), 1.16, 7

$$\left(\begin{array}{cccccccccccccccccc}
1 & 4 & 2 & 4 & 3 & 1 & 1 & 3 & 1 & 1 & 1 & 4 & 2 & 7 & 5 & 7 & 1 & 3 \\
 & 7 & 7 & 2 & 5 & & & & & & & & & & & & & \\
\hline
3 & 7 & 1 & 5 & 7 & 1 & 5 & 1 & 2 & 6 & 7 & 1 & 5 & 2 & 2 & 6 & 4 & 2 \\
 & 1 & 5 & 7 & 1 & & & & & & & & & & & & &
\end{array}\right) \tag{A.7}$$

(28,2), 1.40, 9

$$\left(\begin{array}{cccccccccccccccccc}
2 & 2 & 9 & 7 & 7 & 6 & 4 & 2 & 1 & 9 & 1 & 8 & 1 & 7 & 7 & 1 & 4 & 5 \\
 & 8 & 4 & 2 & 5 & 5 & 3 & 3 & 7 & & & & & & & & & \\
\hline
1 & 4 & 1 & 6 & 5 & 1 & 1 & 5 & 1 & 3 & 7 & 3 & 3 & 8 & 2 & 4 & 7 & 3 \\
 & 1 & 5 & 3 & 4 & 1 & 2 & 7 & 3 & & & & & & & & &
\end{array}\right) \tag{A.8}$$

(32,2), 1.65, 13

$$\left(\begin{array}{cccccccccccccccccc}
8 & 11 & 1 & 10 & 10 & 2 & 12 & 1 & 7 & 3 & 1 & 2 & 3 & 1 & 1 & 3 & 12 & 4 \\
 & 4 & 6 & 13 & 8 & 11 & 11 & 2 & 13 & 11 & 3 & 2 & 12 & & & & & \\
\hline
3 & 4 & 10 & 1 & 7 & 11 & 13 & 9 & 3 & 2 & 5 & 7 & 3 & 7 & 2 & 1 & 9 & 1 \\
 & 3 & 1 & 1 & 1 & 2 & 1 & 12 & 4 & 3 & 12 & 1 & 10 & & & & &
\end{array}\right) \tag{A.9}$$

A.2.2. k=3

(6,3), 0.33, 2

$$\left(\begin{array}{ccc}
1 & 2 & 1 \\
\hline
2 & 1 & 1 \\
\hline
1 & 1 & 2
\end{array}\right) \tag{A.10}$$

(8,3), 0.47, 3

$$\left(\begin{array}{ccccc}
1 & 1 & 3 & 1 & 2 \\
\hline
3 & 2 & 1 & 1 & 3 \\
\hline
2 & 1 & 1 & 3 & 1
\end{array}\right) \tag{A.11}$$

(12,3), 0.89, 5

$$\left(\begin{array}{ccccccccc}
1 & 1 & 1 & 2 & 2 & 3 & 3 & 2 & 5 \\
\hline
3 & 4 & 2 & 1 & 2 & 1 & 4 & 3 & 1 \\
\hline
1 & 5 & 4 & 3 & 1 & 4 & 1 & 4 & 1
\end{array}\right) \tag{A.12}$$

(16,3), 1.31, 8

$$\left(\begin{array}{ccccccccccccc}
5 & 8 & 1 & 3 & 1 & 2 & 5 & 6 & 7 & 4 & 1 & 1 & 1 \\ \hline
3 & 2 & 8 & 2 & 4 & 5 & 1 & 6 & 1 & 3 & 3 & 2 & 5 \\ \hline
1 & 1 & 5 & 8 & 8 & 1 & 7 & 1 & 3 & 8 & 3 & 7 & 6
\end{array}\right) \tag{A.13}$$

(20,3), 1.65, 11

$$\left(\begin{array}{ccccccccccccccccc}
1 & 2 & 3 & 3 & 4 & 8 & 2 & 10 & 7 & 3 & 2 & 3 & 10 & 1 & 8 & 3 & 3 \\ \hline
11 & 5 & 3 & 5 & 3 & 10 & 1 & 1 & 4 & 8 & 11 & 9 & 6 & 2 & 1 & 7 & 4 \\ \hline
6 & 5 & 8 & 4 & 7 & 7 & 8 & 3 & 2 & 2 & 1 & 1 & 2 & 10 & 2 & 11 & 5
\end{array}\right) \tag{A.14}$$

(24,3), 2.02, 16

$$\left(\begin{array}{ccccccccccccccccc}
9 & 8 & 10 & 16 & 11 & 11 & 13 & 5 & 12 & 2 & 6 & 11 & 1 & 2 & 14 & 3 & 3 \\
 & 5 & 11 & 1 & 5 & & & & & & & & & & & & \\ \hline
1 & 5 & 2 & 15 & 8 & 5 & 10 & 2 & 1 & 3 & 15 & 3 & 13 & 11 & 2 & 15 & 14 \\
 & 14 & 1 & 15 & 6 & & & & & & & & & & & & \\ \hline
7 & 15 & 5 & 3 & 9 & 3 & 3 & 7 & 5 & 4 & 1 & 2 & 6 & 3 & 13 & 7 & 11 \\
 & 2 & 15 & 4 & 13 & & & & & & & & & & & &
\end{array}\right) \tag{A.15}$$

(28,3), 2.32, 21

$$\left(\begin{array}{ccccccccccccccccc}
1 & 2 & 1 & 4 & 15 & 10 & 11 & 1 & 9 & 12 & 17 & 4 & 12 & 3 & 15 & 3 & 16 \\
 & 13 & 1 & 2 & 15 & 2 & 5 & 9 & 11 & & & & & & & & \\ \hline
16 & 9 & 19 & 7 & 5 & 2 & 19 & 8 & 16 & 1 & 2 & 6 & 7 & 14 & 11 & 15 & 3 \\
 & 6 & 3 & 1 & 12 & 8 & 2 & 10 & 3 & & & & & & & & \\ \hline
7 & 3 & 11 & 12 & 16 & 5 & 8 & 10 & 6 & 2 & 16 & 21 & 13 & 3 & 4 & 20 & 1 \\
 & 18 & 16 & 11 & 9 & 8 & 9 & 2 & 15 & & & & & & & &
\end{array}\right) \tag{A.16}$$

(32,3), 2.56, 25

$$\left(\begin{array}{ccccccccccccccccc}
1 & 6 & 1 & 18 & 3 & 11 & 5 & 20 & 24 & 11 & 20 & 8 & 1 & 1 & 10 & 1 & 1 \\
 & 1 & 14 & 22 & 15 & 5 & 1 & 21 & 18 & 3 & 23 & 22 & 10 & & & & \\ \hline
5 & 25 & 10 & 11 & 25 & 3 & 13 & 11 & 7 & 10 & 1 & 11 & 22 & 8 & 2 & 16 & 11 \\
 & 21 & 7 & 14 & 10 & 8 & 23 & 15 & 13 & 3 & 24 & 1 & 20 & & & & \\ \hline
10 & 11 & 14 & 4 & 4 & 25 & 10 & 24 & 2 & 5 & 18 & 3 & 17 & 1 & 5 & 11 & 9 \\
 & 7 & 11 & 25 & 21 & 22 & 3 & 5 & 11 & 13 & 19 & 14 & 3 & & & &
\end{array}\right) \tag{A.17}$$

A.2.3. k=4

(8,4), 0.5, 3

$$\left(\begin{array}{cccc} 2 & 1 & 3 & 1 \\ \hline 3 & 2 & 1 & 1 \\ \hline 1 & 3 & 1 & 2 \\ \hline 1 & 1 & 2 & 3 \end{array}\right) \tag{A.18}$$

(12,4), 1.16, 8

$$\left(\begin{array}{cccccccc} 1 & 2 & 2 & 1 & 1 & 3 & 2 & 6 \\ \hline 4 & 6 & 3 & 1 & 8 & 1 & 1 & 1 \\ \hline 1 & 5 & 8 & 6 & 3 & 7 & 1 & 2 \\ \hline 6 & 3 & 8 & 7 & 2 & 3 & 5 & 1 \end{array}\right) \tag{A.19}$$

(16,4), 1.71, 13

$$\left(\begin{array}{cccccccccccc} 1 & 9 & 1 & 10 & 2 & 1 & 3 & 11 & 5 & 1 & 4 & 10 \\ \hline 3 & 1 & 6 & 3 & 7 & 9 & 11 & 13 & 2 & 7 & 10 & 7 \\ \hline 13 & 5 & 11 & 4 & 3 & 9 & 2 & 8 & 1 & 2 & 1 & 13 \\ \hline 8 & 4 & 3 & 1 & 9 & 2 & 1 & 1 & 11 & 6 & 9 & 2 \end{array}\right) \tag{A.20}$$

(20,4), 2.16, 20

$$\left(\begin{array}{cccccccccccccccc} 13 & 3 & 3 & 1 & 10 & 12 & 1 & 1 & 1 & 14 & 13 & 6 & 12 & 15 & 10 & 20 \\ \hline 10 & 12 & 1 & 16 & 3 & 15 & 11 & 10 & 5 & 7 & 4 & 20 & 17 & 2 & 8 & 14 \\ \hline 1 & 11 & 4 & 12 & 19 & 2 & 8 & 2 & 10 & 10 & 6 & 9 & 14 & 7 & 10 & 1 \\ \hline 7 & 1 & 8 & 4 & 7 & 9 & 7 & 1 & 13 & 16 & 9 & 1 & 13 & 3 & 1 & 3 \end{array}\right) \tag{A.21}$$

(24,4), 2.64, 31

$$\left(\begin{array}{cccccccccccccccc} 7 & 11 & 11 & 1 & 16 & 8 & 1 & 23 & 2 & 3 & 1 & 7 & 27 & 23 & 22 & 25 \\ & 12 & 17 & 7 & 16 & & & & & & & & & & & \\ \hline 5 & 1 & 31 & 24 & 9 & 2 & 28 & 11 & 28 & 27 & 17 & 3 & 6 & 27 & 4 & 22 \\ & 4 & 4 & 13 & 6 & & & & & & & & & & & \\ \hline 6 & 14 & 12 & 2 & 5 & 3 & 26 & 17 & 20 & 2 & 1 & 24 & 2 & 1 & 17 & 11 \\ & 3 & 13 & 21 & 31 & & & & & & & & & & & \\ \hline 8 & 6 & 2 & 13 & 1 & 6 & 4 & 4 & 9 & 16 & 9 & 15 & 29 & 21 & 16 & 3 \\ & 23 & 11 & 19 & 11 & & & & & & & & & & & \end{array}\right) \tag{A.22}$$

(28,4), 2.88, 34

$$\left(\begin{array}{cccccccccccccccc}
19 & 14 & 10 & 11 & 24 & 5 & 33 & 10 & 16 & 19 & 7 & 1 & 16 & 29 & 2 & 29 \\
 & 1 & 1 & 1 & 15 & 30 & 28 & 20 & 27 & & & & & & & \\
\hline
11 & 2 & 11 & 26 & 14 & 30 & 5 & 32 & 10 & 15 & 8 & 33 & 20 & 7 & 9 & 1 \\
 & 26 & 15 & 1 & 7 & 10 & 2 & 1 & 10 & & & & & & & \\
\hline
3 & 1 & 25 & 32 & 28 & 11 & 24 & 4 & 13 & 1 & 30 & 26 & 30 & 34 & 24 & 5 \\
 & 14 & 32 & 4 & 33 & 4 & 13 & 34 & 29 & & & & & & & \\
\hline
27 & 1 & 2 & 3 & 3 & 2 & 12 & 5 & 3 & 3 & 3 & 30 & 13 & 27 & 34 & 25 \\
 & 27 & 29 & 16 & 23 & 29 & 6 & 28 & 8 & & & & & & &
\end{array}\right) \tag{A.23}$$

(32,4), 3.15, 40

$$\left(\begin{array}{cccccccccccccccc}
23 & 21 & 5 & 9 & 35 & 4 & 21 & 30 & 37 & 32 & 25 & 37 & 19 & 11 & 4 & 34 \\
 & 2 & 14 & 12 & 8 & 29 & 32 & 17 & 14 & 4 & 15 & 10 & 27 & & & \\
\hline
29 & 6 & 17 & 5 & 2 & 25 & 11 & 38 & 1 & 30 & 3 & 22 & 40 & 21 & 13 & 1 \\
 & 40 & 38 & 21 & 34 & 4 & 19 & 19 & 7 & 10 & 22 & 26 & 13 & & & \\
\hline
5 & 32 & 21 & 30 & 7 & 2 & 16 & 22 & 20 & 23 & 32 & 2 & 23 & 3 & 37 & 3 \\
 & 26 & 28 & 4 & 34 & 23 & 20 & 7 & 37 & 25 & 8 & 3 & 7 & & & \\
\hline
1 & 19 & 18 & 35 & 8 & 25 & 10 & 6 & 3 & 27 & 1 & 10 & 37 & 27 & 22 & 7 \\
 & 27 & 13 & 1 & 3 & 2 & 9 & 1 & 2 & 1 & 4 & 28 & 40 & & &
\end{array}\right) \tag{A.24}$$

A.2.4. k=5

(10,5), 0.92, 7

$$\left(\begin{array}{ccccc}
4 & 3 & 1 & 1 & 1 \\
\hline
3 & 6 & 6 & 1 & 7 \\
\hline
1 & 1 & 6 & 2 & 3 \\
\hline
3 & 1 & 4 & 3 & 1 \\
\hline
1 & 4 & 1 & 3 & 5
\end{array}\right) \tag{A.25}$$

(12,5), 1.26, 8

$$\left(\begin{array}{ccccccc}
2 & 6 & 1 & 1 & 2 & 1 & 6 \\
\hline
3 & 4 & 7 & 3 & 1 & 1 & 1 \\
\hline
4 & 6 & 6 & 3 & 7 & 4 & 5 \\
\hline
1 & 2 & 3 & 4 & 2 & 6 & 1 \\
\hline
6 & 7 & 6 & 7 & 3 & 1 & 8
\end{array}\right) \tag{A.26}$$

(16,5), 2.04, 18

$$\begin{pmatrix} 14 & 3 & 10 & 1 & 7 & 10 & 18 & 2 & 15 & 2 & 13 \\ \hline 7 & 11 & 13 & 5 & 2 & 12 & 14 & 17 & 3 & 5 & 1 \\ \hline 16 & 15 & 15 & 2 & 6 & 1 & 14 & 6 & 9 & 7 & 2 \\ \hline 1 & 2 & 13 & 15 & 11 & 1 & 5 & 3 & 2 & 1 & 17 \\ \hline 5 & 14 & 2 & 7 & 8 & 17 & 1 & 1 & 1 & 8 & 17 \end{pmatrix} \tag{A.27}$$

(20,5), 2.59, 27

$$\begin{pmatrix} 1 & 2 & 25 & 14 & 9 & 8 & 7 & 4 & 22 & 27 & 15 & 10 & 22 & 7 & 2 \\ \hline 7 & 23 & 16 & 11 & 10 & 21 & 9 & 13 & 2 & 1 & 1 & 8 & 10 & 7 & 21 \\ \hline 6 & 8 & 23 & 4 & 21 & 5 & 11 & 1 & 8 & 6 & 1 & 22 & 15 & 22 & 2 \\ \hline 16 & 1 & 24 & 5 & 12 & 6 & 26 & 20 & 19 & 27 & 21 & 20 & 8 & 3 & 26 \\ \hline 2 & 9 & 7 & 24 & 12 & 27 & 1 & 22 & 1 & 8 & 1 & 1 & 3 & 15 & 23 \end{pmatrix} \tag{A.28}$$

(24,5), 2.96, 37

$$\begin{pmatrix} 8 & 21 & 7 & 1 & 29 & 16 & 27 & 2 & 26 & 19 & 19 & 25 & 19 & 21 & 3 \\ & 4 & 1 & 21 & 7 \\ \hline 11 & 14 & 15 & 1 & 1 & 35 & 2 & 28 & 10 & 12 & 30 & 21 & 2 & 8 & 19 \\ & 9 & 18 & 3 & 19 \\ \hline 22 & 13 & 14 & 34 & 8 & 2 & 8 & 21 & 1 & 17 & 12 & 7 & 9 & 31 & 20 \\ & 1 & 35 & 28 & 22 \\ \hline 4 & 22 & 35 & 26 & 6 & 3 & 2 & 25 & 9 & 35 & 24 & 16 & 1 & 19 & 29 \\ & 35 & 14 & 5 & 2 \\ \hline 26 & 12 & 19 & 37 & 25 & 1 & 3 & 9 & 25 & 28 & 1 & 8 & 14 & 23 & 19 \\ & 12 & 11 & 6 & 6 \end{pmatrix} \tag{A.29}$$

(28,5), 3.35, 48

$$\begin{pmatrix} 33 & 3 & 20 & 34 & 37 & 1 & 13 & 32 & 1 & 18 & 6 & 2 & 5 & 11 & 27 \\ & 24 & 8 & 47 & 21 & 8 & 21 & 33 & 26 \\ \hline 16 & 1 & 46 & 2 & 9 & 4 & 12 & 6 & 2 & 45 & 10 & 1 & 25 & 7 & 37 \\ & 34 & 15 & 11 & 44 & 1 & 11 & 2 & 37 \\ \hline 29 & 35 & 21 & 41 & 1 & 29 & 19 & 43 & 16 & 6 & 19 & 1 & 21 & 23 & 10 \\ & 24 & 23 & 33 & 26 & 41 & 48 & 34 & 46 \\ \hline 6 & 19 & 2 & 4 & 4 & 45 & 3 & 5 & 7 & 13 & 12 & 46 & 33 & 44 & 11 \\ & 16 & 35 & 15 & 4 & 1 & 43 & 9 & 22 \\ \hline 1 & 24 & 47 & 22 & 7 & 45 & 22 & 3 & 2 & 44 & 21 & 29 & 27 & 29 & 33 \\ & 19 & 43 & 45 & 13 & 34 & 20 & 35 & 2 \end{pmatrix} \tag{A.30}$$

(32,5), 3.99, 82

$$\left(\begin{array}{ccccccccccccccc}
27 & 1 & 11 & 70 & 22 & 3 & 65 & 76 & 19 & 51 & 36 & 67 & 66 & 20 & 29 \\
 & 82 & 17 & 70 & 71 & 57 & 2 & 58 & 13 & 9 & 13 & 41 & 66 & & \\
\hline
8 & 61 & 52 & 9 & 58 & 10 & 24 & 1 & 8 & 10 & 52 & 76 & 46 & 16 & 2 \\
 & 17 & 81 & 30 & 8 & 15 & 32 & 35 & 11 & 48 & 24 & 77 & 61 & & \\
\hline
29 & 13 & 2 & 47 & 30 & 80 & 11 & 46 & 66 & 11 & 6 & 3 & 56 & 20 & 1 \\
 & 80 & 2 & 27 & 37 & 25 & 15 & 14 & 63 & 76 & 21 & 14 & 21 & & \\
\hline
34 & 78 & 5 & 61 & 3 & 27 & 78 & 2 & 31 & 13 & 21 & 75 & 1 & 23 & 42 \\
 & 55 & 45 & 49 & 24 & 56 & 37 & 4 & 19 & 4 & 42 & 25 & 39 & & \\
\hline
2 & 60 & 9 & 1 & 19 & 64 & 65 & 38 & 20 & 67 & 8 & 48 & 22 & 56 & 66 \\
 & 13 & 12 & 20 & 57 & 12 & 6 & 45 & 1 & 39 & 10 & 39 & 41 & &
\end{array}\right) \tag{A.31}$$

A.2.5. k=6

(12,6), 1.36, 9

$$\left(\begin{array}{cccccc}
5 & 1 & 3 & 1 & 3 & 1 \\
\hline
9 & 1 & 1 & 7 & 2 & 4 \\
\hline
1 & 7 & 5 & 6 & 3 & 2 \\
\hline
2 & 6 & 1 & 3 & 3 & 9 \\
\hline
6 & 5 & 7 & 7 & 6 & 1 \\
\hline
7 & 7 & 6 & 8 & 1 & 6
\end{array}\right) \tag{A.32}$$

(16,6), 2.15, 19

$$\left(\begin{array}{cccccccccc}
16 & 7 & 1 & 14 & 2 & 8 & 19 & 16 & 5 & 12 \\
\hline
5 & 17 & 1 & 19 & 1 & 14 & 14 & 17 & 7 & 16 \\
\hline
6 & 14 & 3 & 4 & 1 & 19 & 9 & 2 & 14 & 9 \\
\hline
12 & 15 & 7 & 1 & 19 & 1 & 9 & 1 & 6 & 7 \\
\hline
1 & 1 & 9 & 5 & 3 & 1 & 7 & 13 & 15 & 2 \\
\hline
10 & 1 & 6 & 18 & 11 & 3 & 1 & 8 & 11 & 7
\end{array}\right) \tag{A.33}$$

(20,6), 2.89, 36

$$\left(\begin{array}{cccccccccccccc}
18 & 30 & 1 & 17 & 1 & 1 & 28 & 2 & 7 & 34 & 11 & 10 & 25 & 19 \\
\hline
30 & 33 & 3 & 26 & 13 & 2 & 12 & 3 & 18 & 29 & 15 & 29 & 3 & 15 \\
\hline
1 & 13 & 25 & 10 & 1 & 4 & 9 & 26 & 20 & 13 & 27 & 19 & 5 & 16 \\
\hline
22 & 30 & 3 & 12 & 27 & 10 & 1 & 12 & 1 & 36 & 13 & 2 & 26 & 21 \\
\hline
6 & 2 & 34 & 11 & 15 & 5 & 5 & 16 & 4 & 11 & 30 & 27 & 32 & 23 \\
\hline
14 & 4 & 11 & 1 & 3 & 15 & 12 & 28 & 10 & 19 & 9 & 32 & 14 & 25
\end{array}\right) \tag{A.34}$$

(24,6), 3.54, 61

$$\left(\begin{array}{cccccccccccccc}
41 & 15 & 1 & 7 & 1 & 55 & 12 & 48 & 23 & 52 & 26 & 39 & 10 & 9 \\
 & 4 & 1 & 16 & 14 & & & & & & & & & \\
\hline
1 & 3 & 34 & 23 & 24 & 48 & 41 & 14 & 60 & 27 & 43 & 47 & 12 & 21 \\
 & 61 & 18 & 13 & 20 & & & & & & & & & \\
\hline
1 & 57 & 36 & 1 & 8 & 9 & 24 & 50 & 19 & 15 & 29 & 24 & 25 & 27 \\
 & 15 & 21 & 5 & 52 & & & & & & & & & \\
\hline
21 & 14 & 2 & 32 & 21 & 1 & 33 & 5 & 43 & 59 & 12 & 42 & 37 & 54 \\
 & 7 & 23 & 38 & 52 & & & & & & & & & \\
\hline
39 & 30 & 31 & 34 & 1 & 20 & 1 & 16 & 19 & 50 & 7 & 43 & 46 & 61 \\
 & 11 & 3 & 15 & 26 & & & & & & & & & \\
\hline
11 & 1 & 46 & 8 & 49 & 6 & 42 & 23 & 29 & 4 & 45 & 24 & 56 & 40 \\
 & 60 & 1 & 8 & 40 & & & & & & & & &
\end{array}\right) \tag{A.35}$$

(28,6), 4.43, 113

$$\left(\begin{array}{cccccccccccccc}
78 & 62 & 28 & 50 & 49 & 21 & 2 & 27 & 16 & 93 & 36 & 1 & 61 & 85 \\
 & 37 & 38 & 61 & 47 & 3 & 25 & 59 & 4 & & & & & \\
\hline
59 & 81 & 33 & 15 & 56 & 47 & 36 & 81 & 12 & 54 & 52 & 103 & 28 & 15 \\
 & 8 & 39 & 62 & 1 & 72 & 7 & 106 & 1 & & & & & \\
\hline
69 & 49 & 20 & 19 & 44 & 103 & 43 & 30 & 113 & 90 & 15 & 39 & 3 & 92 \\
 & 22 & 26 & 77 & 54 & 65 & 55 & 3 & 96 & & & & & \\
\hline
38 & 1 & 63 & 15 & 31 & 75 & 40 & 68 & 7 & 27 & 18 & 73 & 112 & 56 \\
 & 2 & 82 & 24 & 2 & 93 & 46 & 4 & 24 & & & & & \\
\hline
97 & 84 & 97 & 66 & 25 & 4 & 99 & 72 & 95 & 95 & 62 & 8 & 42 & 11 \\
 & 100 & 111 & 51 & 7 & 65 & 92 & 43 & 95 & & & & & \\
\hline
1 & 31 & 37 & 27 & 5 & 50 & 34 & 73 & 11 & 49 & 70 & 96 & 51 & 64 \\
 & 5 & 1 & 53 & 38 & 58 & 26 & 111 & 83 & & & & &
\end{array}\right) \tag{A.36}$$

(32,6),9.09, 2033

$$\left(\begin{array}{ccccccccccc}
1837 & 1662 & 1273 & 1488 & 1500 & 525 & 923 & 1118 & 1709 & 1598 & 1957 \\
 & 1810 & 171 & 228 & 1351 & 16 & 1723 & 1941 & 736 & 1038 & 678 \\
 & 1686 & 1291 & 1185 & 1525 & 1104 & & & & & \\
\hline
1012 & 1938 & 614 & 1643 & 1337 & 404 & 1257 & 563 & 1891 & 710 & 1087 \\
 & 767 & 1828 & 749 & 317 & 1738 & 512 & 487 & 1965 & 1862 & 503 \\
 & 1641 & 1755 & 1239 & 631 & 386 & & & & & \\
\hline
877 & 1922 & 1570 & 355 & 979 & 534 & 246 & 1592 & 130 & 1583 & 1995 \\
 & 1387 & 98 & 1838 & 50 & 1185 & 558 & 1877 & 1933 & 874 & 1567 \\
 & 397 & 1360 & 1485 & 212 & 1863 & & & & & \\
\hline
1079 & 1967 & 1054 & 1710 & 305 & 1931 & 1584 & 1875 & 238 & 515 & 362 \\
 & 484 & 60 & 491 & 19 & 8 & 1878 & 116 & 1846 & 1927 & 1301 \\
 & 356 & 1756 & 1186 & 1229 & 1276 & & & & & \\
\hline
1583 & 541 & 714 & 1795 & 357 & 1792 & 1714 & 1410 & 1455 & 2019 & 1294 \\
 & 992 & 1846 & 1531 & 1506 & 160 & 2014 & 1566 & 651 & 2033 & 1573 \\
 & 481 & 101 & 1371 & 360 & 1402 & & & & & \\
\hline
1726 & 69 & 540 & 907 & 1345 & 76 & 1448 & 11 & 1871 & 1804 & 1803 \\
 & 1537 & 1167 & 1211 & 1508 & 413 & 155 & 1306 & 1944 & 1661 & 1466 \\
 & 1911 & 1179 & 69 & 1896 & 704 & & & & &
\end{array}\right) \tag{A.37}$$

A.3. Vandermonde matrices

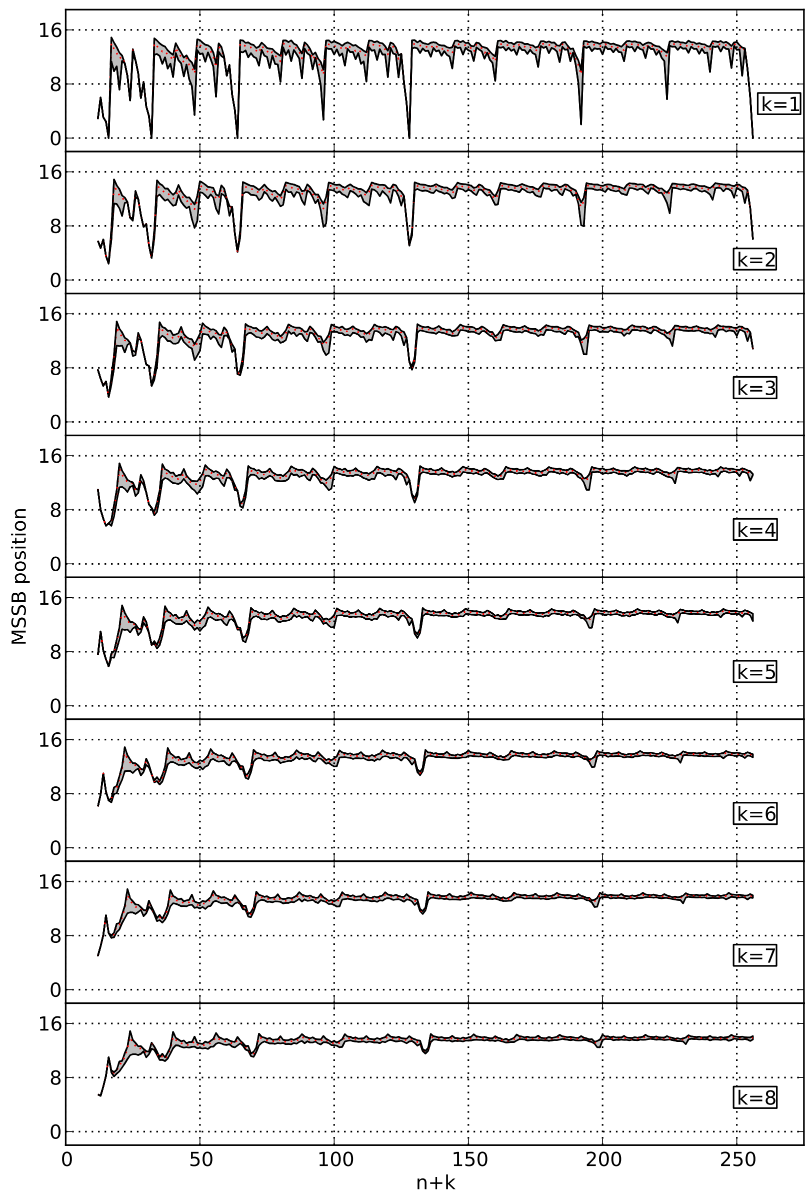


Figure A.1.: Range of the average MSSB positions of the Vandermonde-based generator matrices for all 2048 primitive generator polynomials of $GF(2^{16})$. The dotted lines indicate the mean values over all primitive polynomials.

A.4. Testbeds

A.4.1. CPU testbed

Table A.2.: CPU Testbed

Intel Core i5-2500K		
	micro architecture	Sandy Bridge
	clock speed	3.3 GHz
	cores	4
	L1/core	32 kiB data
		32 kiB inst
	L2/core	256 kiB
	L3 (shared)	6 MiB
Memory		
	type	DDR3
	capcity	2 * 4 GiB
	clock speed	1333 MHz
	peak bandwidth	2* 10667 MiB/s
Mainboard		
	model	Asus P8H67-V
Software		
	gcc	4.5.0
	kernel	3.0.0

A.4.2. GPU testbed

Table A.3.: GPU Testbed

Intel Core i7-930		
	micro architecture	Nehalem
	clock speed	2.8 GHz
	cores	4
	L1/core	32 kiB data
		32 kiB inst
	L2/core	256 kiB
	L3 (shared)	8 MiB
Memory		
	type	DDR3
	capacity	6 * 2 GiB
	clock speed	1066 MHz
	peak bandwidth	25.6 GiB/s
Mainboard		
	model	ASUSTeK P6T7 WS SUPERCOMPUTER
Software		
	gcc	4.5.2
	kernel	2.6.38-13

Table A.4.: Output of the NVIDIA deviceQuery tool for the GeForce GTX 680 GPU.

CUDA Driver Version / Runtime Version	4.2 / 4.2
CUDA Capability Major/Minor version number:	3.0
Total amount of global memory:	2048 MBytes
(8) Multiprocessors x (192) CUDA Cores/MP:	1536 CUDA Cores
GPU Clock rate:	706 MHz (0.71 GHz)
Memory Clock rate:	3004 Mhz
Memory Bus Width:	256-bit
L2 Cache Size:	524288 bytes
Max Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	2147483647 x 65535 x 65535
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and execution:	Yes with 1 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Concurrent kernel execution:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support enabled:	No
Device is using TCC driver mode:	No
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	5 / 0

List of Tables

List of Figures

Index