UNIVERSITY OF CALIFORNIA

Los Angeles

# Coding for Future Large-Scale Data Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Electrical and Computer Engineering

by

Clayton Maxwell Schoeny

2018

# ABSTRACT OF THE DISSERTATION

Coding for Future Large-Scale

Data Systems

by

Clayton Maxwell Schoeny

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Los Angeles, 2018

Professor Lara Dolecek, Chair

This dissertation is focused on creating mathematical techniques—influenced by information theory and coding theory—to address the difficulties associated with storing, transmitting, and analyzing massive amounts of data. By necessity, memory devices are being created more compactly, leading to higher rates of errors. Each of the three parts of this dissertation seeks to combat the unique challenges and potential errors associated with next-generation storage technologies. These advanced error-correcting techniques can be utilized at the system-level for a variety of purposes, e.g., reducing energy consumption, increasing storage density, or decreasing the risk of a catastrophic system failure.

The first part of the dissertation introduces Software-Defined Error-Correcting Codes: a framework for exploiting side-information to heuristically recover from detected (but uncorrectable) errors. The prominent features of this section include the underlying theory, experimental results, and an extension to error-localizing codes. The middle section of the dissertation focuses on coding for unequal message protection, in which special messages are granted extra error-correcting guarantees. A broad class of unequal message protection codes are constructed, maintaining the same amount of redundancy overhead as the baseline alternative. The final part of the dissertation includes code constructions to correct burst deletion errors in DNA storage—a very promising technology that will likely be commonplace

in the near-future, complete with its own set of features and challenges.

The coding theoretic techniques presented here, along with tools inspired by this dissertation, will play a significant role in mitigating errors in future large-scale data systems.

The dissertation of Clayton Maxwell Schoeny is approved.

Arash Ali Amini

Richard D. Wesel

Puneet Gupta

Lara Dolecek, Committee Chair

University of California, Los Angeles

2018

To my parents and Kelsey.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

This dissertation is the synthesis of many years of research—a venture that would not have possible were it not for those around me. I am immensely grateful for the support and backing from my family, friends, colleagues, and mentors.

I would like to thank my advisor, Professor Lara Dolecek, whose initial encouragement and optimism inspired me to begin my doctoral journey. After joining the Laboratory for Robust Information Systems (LORIS), Professor Dolecek instilled in me all the necessary skills required to transition to a research-based academic environment. Her extensive knowledge of cutting edge information theory and coding theory developments provided me with exciting research paths to follow. Additionally, Professor Dolecek's connections within these communities provided for additional opportunities; at conferences, professors were more than happy to talk to me upon learning that I was her student. She has lent a helping hand with every aspect of my PhD tenure—everything from the nitty-gritty proof details in up to high-level grant writing techniques.

I am grateful for the feedback from my other dissertation committee members: Professor Puneet Gupta, Professor Richard Wesel, and Professor Arash Amini. For the past few years, the weekly meetings with Professor Gupta have been personally valuable in that he often presented a fresh perspective, leaving me excited to try new approaches toward my various research problems at hand. I could not have asked for a better undergraduate advisor than Professor Wesel; his quarterly meetings consistently left me with a better understanding of the electrical engineering community, both in academia and in industry.

Thankfully, I have had the honor to collaborate on projects with professors outside of UCLA. Throughout our radiation project, Professor Dariush Divsalar (JPL) was more than happy to share bits of wisdom accumulated over decades of working alongside some of the top communication system engineers in the world. Always friendly and eager to discuss theory, Professor Yuval Cassuto (Technion) inspired a variety of projects stemming from

his wide-range of expertise. It was a pleasure to collaborate with Professor Eitan Yaakobi (Technion), Dr. Ryan Gabrys (SPAWAR), and Professor Antonia Wachter-Zeh (TUM) on the bursty deletions project. My summer research at SPAWAR would not have been as productive without their teamwork and assistance.

The Software-Defined ECC project (with Professor Gupta, Mark Gottscho, and Irina Alam of the NanoCAD Lab) was an exceedingly fruitful collaboration that led to 3 best paper awards. Due to our complementary sets of skills, Mark and I formed a terrific research partnership. This endeavor was possible in large part due to the Qualcomm Innovation Fellowship award that Mark and I received. A large thank you to Dr. Greg Wright (Qualcomm Research) for his championing of us throughout the fellowship proposal process and his mentorship from then on out.

My research has been enjoyable thanks to my labmates in LORIS. I first want to thank the members of LORIS with whom I have co-written a journal manuscript, conference paper, or workshop brief: Dr. Ryan Gabrys, Dr. Nicolas Bitouzé, Dr. Behzad Amiri, Dr. Sean Huang, Dr. Frederic Sala, Ahmed Hareedy, Kayvon Mazooji, Shahroze Kabir, Amirhossein Reisizadeh, Chinmayi Lanka, Siyi Yang, and Zehui Chen. Although I did not get to publish with them, I thank the following LORIS members for their companionship: Dr. Yao Li, Homa Esfahanizadeh, Ehsan Ebrahimzadeh, Lev Tauz, Ruiyi Wu, and Andrew Tan. My duration in LORIS has overlapped the most with Fred, Ahmed, and Homa; I cherish our conversations and arguments over the years, covering a gamut of topics from sports to politics. I owe a special thanks to Fred for taking me under his wing as his protege and teaching me the ropes. Our quarterly outings, international escapades, and general shenanigans led LORIS to have more fun than an engineering lab ought to have.

This journey began long ago. I am grateful that the Alexovich family pushed me toward electrical engineering and has continued to advise me since. Throughout my undergraduate studies, Matthew Lunny and Marvin Leiva helped transform traditionally tedious tasks and long nights of studying into entertaining activities. Thank you to the administrative staff

in the ECE department, especially Deeona Columbia who was always my go-to source for solving any logistical issues. Through a number of internships, I had the opportunity to gain valuable industry experience, and I thank Laurence Zapanta (The Aerospace Corporation), Dan Miner (DIRECTV), and Dr. Ryan Gabrys (SPAWAR), for being excellent managers and mentors.

Most importantly, thank you to my parents, Tom and Joanie, for a lifetime of unconditional love. Their unyielding support inspires me to always do my best. Thank you to Patty and the rest of my extended family for always being there for me. Kelsey has been an amazing source inspiration and encouragement; I can not imagine a better life partner. I am excited to see what adventures await us.

My work was sponsored by the Qualcomm Innovation Fellowship, UCLA Dissertation Year Fellowship, JPL R&TD program, and National Science Foundation grants CCF-1029030, CCF-1150212, CCF-1527130, CCF-1162501, CCF-1029030, BSF-1718389.

# Vita

| | |
|---|---|
| 2010, 2011 | Intern, The Aerospace Corporation |
| 2012 | Bachelor of Science, Electrical Engineering<br>University of California, Los Angeles |
| 2012, 2013 | Intern, DIRECTV |
| 2013–2018 | Research Assistant<br>Department of Electrical Engineering, University of California, Los Angeles |
| 2014 | Master of Science, Electrical Engineering<br>University of California, Los Angeles |
| 2014–2015 | Teaching Assistant<br>Department of Electrical Engineering, University of California, Los Angeles |
| 2014–2015 | Excellence in Teaching Award<br>Department of Electrical Engineering, University of California, Los Angeles |
| 2015 | PhD Intern, SPAWAR |
| 2016–2017 | Qualcomm Innovation Fellow |
| 2017–2018 | Dissertation Year Fellow<br>University of California, Los Angeles |
| 2017–2018 | Distinguished PhD Dissertation in Signals & Systems Award<br>Department of Electrical and Computer Engineering<br>University of California, Los Angeles |

# Selected Publications

**C. Schoeny**, N. Bitouzé, F. Sala, and L. Dolecek, "Efficient File Synchronization: Extensions and Simulations," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pp. 2129-2133, Pacific Grove, CA, November 2014.

**C. Schoeny**, F. Sala, and L. Dolecek, "Analysis and Coding Schemes for the Flash Normal-Laplace Mixture Channel," in *Proceedings of the IEEE International Symposium on Information Theory*, pp. 2101-2105, Hong Kong, June 2015.

M. Gottscho, **C. Schoeny**, L. Dolecek, and P. Gupta, "Software-Defined Error-Correcting Codes," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pp. 276-282, Best of SELSE Special Session, Toulouse, France, June 2016.

F. Sala, **C. Schoeny**, N. Bitouzé, and L. Dolecek, "Synchronizing Files Under a Large Number of Edits," in *IEEE Transactions on Communications*, vol. 64, no. 6, pp. 2258-2273, June 2016.

**C. Schoeny**, A. Wachter-Zeh, R. Gabrys, and E. Yaakobi, "Codes Correcting a Burst of Deletions or Insertions," in *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 1971-1985, January 2017.

F. Sala, **C. Schoeny**, S. Kabir, D. Divsalar, and L. Dolecek, "On Nonuniform Noisy Decoding for LDPC Codes With Application to Radiation-Induced Errors," in *IEEE Transactions on Communications*, vol. 65, no. 4, pp. 1438-1450, January 2017.

M. Gottscho, I. Alam, **C. Schoeny**, L. Dolecek, and P. Gupta, "Low-Cost Memory Fault Tolerance for IoT Devices," in *ACM Transactions on Embedded Computing Systems - Special Issue ESWEEK 2017*, vol. 16, no. 5, pp. 128:1–128:25, October 2017.

**C. Schoeny**, F. Sala, and L. Dolecek, "Novel Combinatorial Coding Results for DNA Sequencing and Data Storage," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pp. 511-515, Pacific Grove, CA, October 2017.

**C. Schoeny**, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-Aware Resiliency: Unequal Message Protection for Random-Access Memories," in *Proceedings of the IEEE Information Theory Workshop*, pp. 166-170, Kaohsiung, Taiwan, November 2017.

I. Alam, **C. Schoeny**, L. Dolecek, and P. Gupta, "Parity++: Lightweight Error Correction for Last Level Caches," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, Best of SELSE Special Session, Luxembourg City, Luxembourg, June 2018.

# CHAPTER 1

# Introduction

We live in a data-driven world where massive amounts of data are being stored, transmitted, and processed at unprecedented rates. It is virtually impossible to find an aspect of modern life untouched by the current data revolution. Big data is so pervasive that the underlying physical and mathematical enabling mechanisms are often brushed aside by the layperson. However, internet websites and *the cloud* are not merely abstract concepts; rather, they exist in server farms—vast warehouses of supercomputers and hard drives.

No medium of communication or storage is perfect—errors are always possible. Given this fact, it is astounding that humanity has achieved such a high level of data ubiquity. From an information-theoretic perspective, this has all been possible, in large part, due to Claude Shannon and his 1949 seminal work which creates a mathematical theory of communication in the presence of noise [1]. Shannon establishes terms such as *entropy*, *mutual information*, and *channel capacity*, giving us the mathematical language and tools necessary to reliably transmit information over noisy channels. The beauty of Shannon's work is that the framework is not tied to a specific technology; the concepts are truly fundamental.

Concurrent with Shannon's breakthrough, the first *error-correcting codes* (ECCs) were established, and in the following decades, many great mathematicians and engineers formed the basis of classical coding theory. The essential idea is that the transmitter and receiver agree upon a coding scheme in which redundancy is added to the original message. Then, if

errors occur in the channel, the receiver will try to correct the errors and decode the original message.

ECCs apply not only to transmissions over physical channels, such as light or modulated electromagnetic waves, but they also apply to storage devices. In this scenario, the *write* operation can be considered as the transmitter and the *read* operation as the receiver. The errors in the storage channel can occur for a multitude of reasons such as radiation particle strikes or charge leakage (the cause of the error is heavily dependent on the specific medium). Virtually every storage system in the modern era has been heavily dependent on ECCs for robustness. For example, a CD-ROM with a moderate amount of scratches and smudges is able to be listened to flawlessly due to the Reed-Solomon code [2].

For a given noisy channel, there are two essential coding theoretic goals. The first is to calculate the capacity of the channel, or equivalently, the minimum rate of redundancy required to reliably communicate. For finite length codes, this number translates to the largest possible codebook, i.e., the most amount of data able to be sent. The second goal is to explicitly construct a practical code with an efficient encoding/decoding scheme. However, thanks to Moore's Law [3], the definition of *practical* changes with time. The prime example is the low-density parity-check (LDPC) code, which was first developed by Gallager in 1960 [4]. However, the encoding/decoding methods were impractical, and the code was forgotten about until 1996 [5]. Today, LDPC codes are used in a variety of applications including solid state drives and satellite communication.

The power of modern computation processors allows us to design and construct more complex ECCs. Instead of a generic one-size-fits-all code, we are able to create codes that target specific error patterns of a particular medium. The underlying theme of this dissertation is the creation of mathematical techniques and powerful ECCs that will help to mitigate errors in current and future large-scale data systems.

The first part of this dissertation focuses on using side-information to heuristically decode. Our framework, called *Software-Defined Error-Correcting Codes* (SDECC), pushes beyond

the conventional error-correcting guarantees. Recent studies have shown that real workloads in computer memories have a very large amount of inherent redundant information [6, 7, 8]. The traditional information-theoretic approach would be to use standard compression techniques; however, these systems often have rigid requirements in terms of message size and maximum latency which makes compression impractical or impossible. We focus on practical computer memory systems, and offer alternatives without adding any additional redundancy. This framework is extremely useful in scenarios where it is a costly option to roll-back to a previous checkpoint.

In the second part of the dissertation, we turn away from heuristic decoding and focus on a priori unequal message protection (UMP). Once again, we focus on practical code dimensions and sizes that are currently being employed in computer memory systems. Our alternative UMP codes offer guarantees for special messages, i.e., messages that are granted additional error-correcting properties. System designers are free to treat the inner workings of our UMP code as a black box; we provide clear requirements and instructions so that the designer can deem which messages are special. Every workload and system has its own characteristics, so this design feature is an added convenience for customizability.

The final part of the dissertation looks ahead toward the technology of tomorrow. The use of DNA as an ultra-dense storage device is promising and will likely have an enormous impact in the near-future. Often times, exotic mediums benefit from exotic ECCs. We create the best-known code for correcting a burst of *deletions* (or insertions), which are error-types that commonly occur in DNA storage. It is much more difficult to protect against these types of errors that do not occur in traditional array based devices. Our code construction, and those inspired by it, will play a role in making DNA a reliable medium of storage.

## 1.1 Outline of Contributions

Each of the following chapters begins with a brief introduction, preliminary notation with the required background information, and an overview of related works.

### Chapter 2 Contributions

Chapter 2 explores the concept of Software-Defined Error-Correcting Codes—a framework for exploiting side-information to heuristically correct errors beyond the traditional guarantees. First, we establish the underlying coding principles of candidate codewords (CCs) in linear codes: the dependence of CCs on specific error patterns, an upper bound on the number of CCs given the coding parameters, the probability of randomly guessing the correct CC, a combinatorial equation for the average number of CCs, an algorithm to efficiently create the CC list, and a summary of the number of CCs of common codes implemented into our framework. For extra protection, we introduce an additional, optional, lightweight hash to prune the CC list.

We then develop a practical recovery policy, based on cacheline entropy, and demonstrate its effectiveness as compared to other recovery policies. We provide recovery results for both integer and float benchmarks, including a detailed breakdown of the panic option.

Lastly, we expand our SDECC framework to incorporate error-localizing codes. We construct a new, customizable, ultra-lightweight error-localizing code (ULELC) that is stronger than a single-bit parity-check code, but weaker than a Hamming code. We construct a ULELC to match the RV64G v.2.0 [9] instruction set architecture and demonstrate its effectiveness for three different redundancy levels. Intuitive panic policies are established based on the resulting sensitivity to the mean Hamming distance score for CCs.

The SDECC project was a multi-year collaboration, and as such, there are many results—mostly hardware architecture based—that are not included in this dissertation. In [10] and [11], we provide, in great detail, the architecture and hardware support necessary for SDECC implementation. Specifically, we describe the penalty box hardware design and the software stack that would enable the proper interrupt process (along with their respective area, power, and latency overheads). Additional hardware details for implementing the lightweight hash are also provided. We present the output quality of approximate computing applications using SDECC, with the errors broken down into the following categories: benign, crash,

hang, tolerable data corruption, and non-tolerable data corruption. Lastly, [10] and [11] include the speedup benefits for supercomputing applications and the improved mean time to failure rates.

### Chapter 3 Contributions

In Chapter 3, we focus on creating unequal message protection (UMP) codes, which grant extra error protection to messages that we a priori deem *special*. This can be viewed as a non-heuristic version of the SDECC framework developed in the previous chapter.

We provide direct alternatives to commonly used codes. First, as an alternative to the single-bit parity-check code (which can only detect errors), we present the special message single-error-correcting (sm)SEC code. We give up single-error-*detection* for all codewords in favor of single-error-*correction* for special codewords. Additionally, with an extra bit of redundancy, we create the SED-(sm)SEC code (single-error-detecting/(special message) single-error-correcting)—in [12], we call this code Parity++. This code, which brings back the guarantee of the single-bit parity-check code, explores the largely untouched redundancy space in between the single-bit parity-check code and the Hamming code.

Second, we provide an alternative to the commonly used extended Hamming code: the SEC-(sm)DEC code (single-error-correcting/(special message) double-error-detecting). We trade-off double-bit error-*detection* in favor of double-bit error *correction* for special codewords. Similarly to the previous case, we create a strictly better alternative with the addition of a single parity bit.

For these codes, we provide explicit constructions, proofs of correctness, sphere-packing upper bounds, novel programming techniques for upper bounds, and a walk-through of the decoding process.

As with the previous section, the work in this chapter has been incorporated into a larger collaborative work, whose computer architecture specific results will not be included in this dissertation. The implementation of Parity++ to protect the last level caches has been

evaluated using Gem5 in [12]. A memory speculation scheme to mitigate the additional decoding latency of Parity++ has also been evaluated and compared with SECDED in [12], along with details of additional architectural modifications in cache required to support it.

### Chapter 4 Contributions

Chapter 4 deals with error-correcting codes for burst deletions. We consider three different models: a deletion burst of exactly $b$ consecutive bits, a deletion burst of at most $b$ consecutive bits, and a non-consecutive deletion burst of size at most $b$. For each of these three models, we prove the equivalence between deletion and insertion-correcting codes. We then derive an explicit, non-asymptotic, upper bound on the code cardinality using an integer linear programming technique developed in [13].

In order to construct our codes, we establish innovative constituent mathematical techniques. First, we derive new results on the redundancy of run-length limited constraints, specifically, in the situation where the run-length is a function of the total length of the vector. Second, we create the shifted VT-code, which takes advantage of positional side-information in order to correct a single deletion. Combining these new techniques, we construct our codes for the three burst deletion models. Additionally, we provide explicit algorithms for encoding run-length limited sequences and for decoding the shifted VT-code (for both the deletion and the insertion case).

Lastly, we expand the previous results to the non-binary case, with specific attention focused on an alphabet size of 4, i.e., the natural alphabet size for DNA coding.

# CHAPTER 2

# Software-Defined Error-Correcting Codes

## 2.1 Introduction

Error-correcting codes (ECCs) are a classic way to build resilient memories by adding redundant parity bits or symbols. A code maps each information message to a unique codeword that allows a limited number of errors to be detected and/or corrected. Errors in the context of ECC can be broadly categorized as *corrected errors* (CEs), *detected-but-uncorrectable errors* (DUEs), *mis-corrected errors* (MCEs), and *undetected errors* (UDEs). CEs are harmless but are nevertheless typically reported to system software [14] as they may indicate the possibility of future DUEs. UDEs may result in *silent data corruption* (SDC) of software state, while MCEs may cause *non-silent data corruption* (NSDC); neither are desirable.

When a DUE occurs, the entire system usually panics, or in the case of a supercomputer, rolls back to a checkpoint to avoid data corruption. Both outcomes harm system availability and can cause some state to be lost. In this chapter, we consider the problem of memory DUEs, because they are more common than both MCEs and UDEs and remain a key challenge to the reliability and availability of extreme-scale systems.

The theoretical development of ECCs have – thus far – implicitly assumed that every message/information bit pattern is equally likely to occur. In general-purpose memory systems, however, this assumption does not hold true. For instance, applications exhibit unique characteristics in control flow and data that arise naturally from the algorithm, inputs, OS,

ISA, and micro-architecture. For the first time, we demonstrate how to exploit some of these characteristics to enhance the capabilities of existing ECCs in order to recover from a large fraction of otherwise-harmful DUEs.

We propose the concept of *Software-Defined ECC* (SDECC), a general class of techniques spanning hardware, software, and coding theory that improves the overall resilience of systems by enabling heuristic best-effort recovery from memory DUEs. The key idea is to add software support to the hardware ECC code so that most memory DUEs can be recovered. SDECC is best suited for approximation-tolerant applications because of its best-effort recovery approach.

Our approach is summarized as follows. When a memory DUE occurs, system software derives additional context about the error—using basic coding theory and knowledge of the ECC implementation—and quickly computes a list of all possible *candidate messages*, one of which is guaranteed to match the original information that was corrupted by the DUE. If available, an optional lightweight hash is proposed to prune the list of candidates. A software-defined data recovery policy heuristically recovers the DUE in a best-effort manner by choosing the most likely remaining candidate based on available *side-information* (SI) from the corresponding un-corrupted cacheline contents; if confidence is low, the policy instead forces a panic to minimize the risk of accidentally-induced MCEs.

We review fundamental properties of error-correcting codes that in Section 2.2 that will be useful for the remaining discussion. In Section 2.3, we present the coding theoretic aspects of SDECC, along with experimental results in Section 2.4. Lastly, in Section 2.5, we expand the SDECC framework into the context of error-localizing codes.

## 2.2 Preliminaries

In this section, we briefly review some fundamental properties of error-correcting codes that are necessary to understand the theory and analysis of SDECC. A *linear block error-correcting code* $\mathcal{C}$ is a linear subspace of all possible row vectors of length $n$. The elements

of $\mathcal{C}$ are called *codewords*, which are each made up of *symbols*. We refer to symbols as *q-ary*, which means they can take on $q$ values where $q$ is a power of 2. A symbol equivalently consists of $b = \log_2(q)$ bits. For binary codes, whenever we use the term "symbol," it is equivalent to "bit."

The code can also be thought of as an injective mapping of a given $q$-ary row vector *message* $\mathbf{m}$ of length $k$ symbols into a codeword $\mathbf{c}$ of length $n$ symbols. Because the code is linear, any two codewords $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$ sum to a codeword $\mathbf{c}'' \in \mathcal{C}$. Thus, there are $r = n - k$ redundant symbols in each codeword. The *rate* of a code is defined as $R = k/n$ and is a measure of data storage efficiency. A linear block code can be fully described by either its $q$-ary $(k \times n)$ *generator matrix* $\mathbf{G}$, or equivalently, by its $q$-ary $(r \times n)$ *parity-check matrix* $\mathbf{H}$. Each row of $\mathbf{H}$ is a parity-check equation that all codewords must satisfy: $\mathbf{c}^{\mathrm{T}} \in \mathrm{Null}(\mathbf{H})$ where T is the transpose. There are usually many ways of *constructing* a particular $\mathbf{G}$ and $\mathbf{H}$ pair for a code with prescribed $k$ and $n$ parameters.

To protect stored message data, one first encodes the message by multiplying it with the generator matrix: $\mathbf{mG} = \mathbf{c}$. One then writes the resulting codeword $\mathbf{c}$ to memory. When the system reads the memory address of interest, the ECC decoder hardware obtains the *received string* $\overline{\mathbf{c}} = \mathbf{c} + \mathbf{e}$. Here, $\mathbf{e}$ is a $q$-ary error-vector of length $n$ that represents where memory faults, if any, have resulted in changed symbols in the codeword. The decoder calculates the *syndrome*: $\mathbf{s} = \mathbf{H}\overline{\mathbf{c}}^{\mathrm{T}}$. There are no declared CEs or DUEs if and only if $\mathbf{s} = \mathbf{0}$, or equivalently, $\overline{\mathbf{c}}$ is actually some codeword $\mathbf{c}' \in \mathcal{C}$. Note that even if $\overline{\mathbf{c}} = \mathbf{c}' \in \mathcal{C}$, there is no guarantee that errors did not actually happen. For instance, if the error is itself a codeword $(\mathbf{e} \in \mathcal{C})$, then there is a UDE due to the linearity of the code: $\mathbf{c} + \mathbf{e} = \mathbf{c}' \in \mathcal{C}$.

The *minimum distance* $d_{min}$ of a linear code is defined as

$$d_{min} = \min_{\substack{\mathbf{u},\mathbf{v} \in \mathcal{C}; \\ \mathbf{u} \neq \mathbf{v}}} [\Delta_q(\mathbf{u}, \mathbf{v})] = \min_{\substack{\mathbf{c} \in \mathcal{C}; \\ \mathbf{c} \neq \mathbf{0}}} [wt_q(\mathbf{c})]$$

where $\Delta_q(\mathbf{u}, \mathbf{v})$ is the $q$-ary Hamming distance between vectors $\mathbf{u}$ and $\mathbf{v}$, and $wt_q(\mathbf{u})$ is the

$q$-ary Hamming weight of a vector $\mathbf{u}$. Notice that the minimum distance is equal to the minimum Hamming weight of all non-$\mathbf{0}$ codewords (because $\mathbf{0}$ is always a codeword in a linear code). Throughout this chapter, we will refer to the most important code parameters using the shorthand notation $[n, k, d_{min}]_q$.

The maximum number of symbol-wise errors in a codeword that the code is guaranteed to correct is given by $t = \lfloor \frac{1}{2}(d_{min} - 1) \rfloor$. Thus, we often refer to codes with even-valued $d_{min}$ as $(t)$-*symbol-correcting,* $(t+1)$-*symbol-detecting*, or $(t)$SC$(t+1)$SD.

### Traditional Decoder Assumptions

The typical decoding method for ECC hardware is to choose the maximum-likelihood codeword under two implicitly statistical assumptions:

1. All symbols in a codeword are equally likely to be afflicted by faults (the symmetric channel model).

2. All messages are equally likely to occur. Under these assumptions, the maximum-likelihood *decode target* is simply the minimum distance codeword from the received string.

Under maximum-likelihood decoding, any error $\mathbf{e}$ with $wt_q(\mathbf{e}) > t$ is guaranteed to cause either a DUE, MCE, or a UDE.

## 2.3 SDECC Theory

SDECC is based on the fundamental observation that when a $(t+1)$-symbol DUE occurs in a $(t)$SC$(t+1)$SD code, there remains significant information in the received string $\overline{\mathbf{c}}$. This information can be used to recover the original message $\mathbf{m}$ with reasonable certainty. It is not the case that the original message was completely lost, i.e., one need not naïvely choose

from all $q^k$ possible messages. In fact, given $t$ errors, there are exactly

$$N = \binom{n}{t+1}(q-1)^{(t+1)} \tag{2.1}$$

possible error patterns that could have led to the corrupted received string. Guessing correctly out of $N$ possibilities is still difficult; however, in practice, there are only a handful of likely possibilities: we call them *candidate codewords* (each corresponding to a candidate message).

If the hardware ECC decoder registers a DUE, there can be several equidistant candidate codewords at the $q$-ary Hamming distance of exactly $(t+1)$ from the received string $\overline{\mathbf{c}}$. We denote the set of candidates by $\Psi(\overline{\mathbf{c}})$; note that $\Psi(\overline{\mathbf{c}}) \subseteq \mathcal{C}$.

Without any side-information about message probabilities, under conventional principles, each candidate codeword is assumed to be equally likely. In other words, there is a candidate codeword more likely than the others if and only if it is uniquely closest to $\overline{\mathbf{c}}$; in such a case, a CE could have been registered instead of a DUE (depending on the implementation of the ECC decoder).

We retain the first assumption from Sec. 2.2, i.e., we assume all DUE error patterns are equally likely to occur. However, in the specific case of DUEs, *we drop Assumption 2*: this allows us to leverage SI about memory contents to help choose the right candidate codeword in the event of a given DUE.

The size of the candidate codeword list $|\Psi(\overline{\mathbf{c}})|$ is independent of the original codeword; it depends only on the error-vector $\mathbf{e}$ due to linearity of the code $\mathcal{C}$. That is,

$$|\Psi(\overline{\mathbf{c}})| = |\Psi(\mathbf{c}+\mathbf{e})| = |\Psi(\mathbf{e})|. \tag{2.2}$$

Note that the *actual set* of candidate codewords $\Psi(\overline{\mathbf{c}})$ still depends on both the error-vector $\mathbf{e}$ and the original codeword $\mathbf{c}$ (because $\overline{\mathbf{c}} = \mathbf{c}+\mathbf{e}$).

One can better understand candidate codewords by visualizing the Hamming space of a

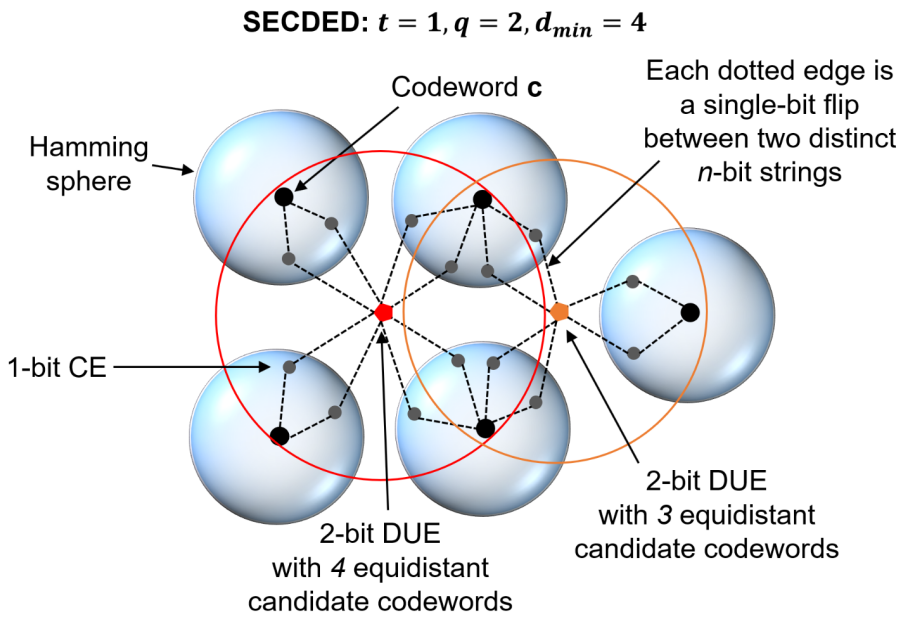



Figure 2.1: Conceptual illustration of candidate codewords for 2-bit DUEs in a 2D-represented code space of a binary SECDED code. The actual Hamming space has $n$ dimensions.

code. Consider Figure 2.1, which depicts the relationships between codewords, CEs, DUEs, and candidate codewords for individual DUEs for a SECDED code. Here, the red-point received string $\overline{\mathbf{c}}$ (a DUE) has four candidate codewords, marked within the red circle; these are each the minimum distance of 2 bit-flips away from $\overline{\mathbf{c}}$.

We derive bounds on the number of candidate codewords, show how to compute a list of candidates for a given DUE, and explain how to prune a list of candidates using a small cacheline-level second-tier checksum.

### 2.3.1 Candidate Codewords

The number of candidate codewords $|\Psi(\mathbf{e})|$ for any given $(t+1)$ DUE $\mathbf{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice.

**Lemma 1.** *For any error* $\mathbf{e}$ *with* $wt_q(\mathbf{e}) = (t+1)$ *in a* $(t)SC(t+1)SD$ *linear* $q$*-ary code* $\mathcal{C}$ *of length* $n$*,*

$$|\Psi(\mathbf{e})| \leq \left\lfloor \frac{n(q-1)}{t+1} \right\rfloor.$$

*Proof.* The received string $\overline{\mathbf{c}}$ is exactly $q$-ary distance 1 from the $t$-boundary of the nearest Hamming sphere(s). Thus, there are at most $n(q-1)$ single-element perturbations $\mathbf{p}$ such that $\mathbf{y} = \overline{\mathbf{c}} + \mathbf{p}$ is a CE inside a Hamming sphere of a codeword. For each perturbation that results in a CE, there must be exactly $t$ more single-element perturbations to fully arrive at a candidate codeword $\mathbf{c}'$. Because we cannot perturb the same elements more than once to arrive at a given $\mathbf{c}'$, there cannot ever be more than $\lfloor (n(q-1))/(t+1) \rfloor$ candidate codewords. □

The probability of correctly guessing the original codeword – without the use of any side-information – for a specific error $\mathbf{e}$ is simply the reciprocal of the number of candidate codewords: $P_G(\mathbf{e}) = 1/|\Psi(\mathbf{e})|$. Let $\overline{P_G}$ be the average probability of guessing the correct codeword over all possible $(t+1)$-symbol DUEs. Also let $\sum_{\mathbf{e}}$ represent the summation over all possible $(t+1)$ symbol-wise error-vectors $\mathbf{e}$. Then

$$\overline{P_G} = \frac{1}{N} \sum_{\mathbf{e}} \frac{1}{|\Psi(\mathbf{e})|}. \tag{2.3}$$

**Average Number of Candidates**

For a particular construction of a given code, we define $W_q(w)$ as the total number of codewords that have $q$-ary Hamming weight $w$. Then $W_q(d_{min})$ refers to the total number of minimum weight non-$\mathbf{0}$ codewords; its value depends on the exact constructions of $\mathbf{G}$ and $\mathbf{H}$ for given $[n, k, d_{min}]_q$ parameters. The average number of candidate codewords over all possible $(t+1)$-symbol DUEs is denoted as $\mu$.

**Lemma 2.** *For a linear $q$-ary $(t)SC(t+1)SD$ code $\mathcal{C}$ of length $n$ and with given $W_q(d_{min} = 2t+2)$, the average number of candidate codewords $\mu$ over all possible $(t+1)$-symbol DUEs is*

$$\mu(n, t, q) = \frac{\binom{2t+2}{t+1} W_q(2t+2)}{\binom{n}{t+1}(q-1)^{(t+1)}} + 1.$$

*Proof.* In order to find the average number of candidate codewords, we must sum the number

of candidate codewords for each unique $(t+1)$ $q$-ary error $\mathbf{e}_E$ where $E = i_1, i_2, \cdots, i_{(t+1)}$, and $i_1 \neq i_2 \neq \cdots \neq i_{(t+1)}$. We then divide that sum by the number of error-vectors ($n$ choose $(t+1)$). By linearity and without loss of generality, assume $\mathbf{c} = \mathbf{0}$. We know that the only codewords $\mathbf{c}' \in \mathcal{C}$ that can satisfy $\Delta(\mathbf{c}-\mathbf{c}', \mathbf{e}) = (t+1)$ have weight $W_q(d_{min})$. Each such $\mathbf{c}'$ that has $\mathbf{c}_{i_1} = \mathbf{e}_{i_1}$, $\mathbf{c}_{i_2} = \mathbf{e}_{i_2}$, etc., then has ($d_{min}$ choose $(t+1)$) distinct error-vectors $\mathbf{e}_E$. Thus summing over all error-vectors, each codeword $\mathbf{c}'$ with $wt(\mathbf{c}') = d_{min}$ contributes to ($d_{min}$ choose $(t+1)$) candidate codewords. To average, we divide $[(d_{min}$ choose $(t+1))] \times W_q(d_{min})$ by ($n$ choose $(t+1)$). We also divide by $(q-1)^{(t+1)}$ because each non-zero element of the error-vector $\mathbf{e}_E$ can take values from 1 to $q-1$. Finally, we add 1 to the expression since the original codeword $\mathbf{c}$ is a candidate codeword for every possible error-vector, and was not already counted in $W_q(d_{min})$. □

We find that $\mu$ is often easier to compute than $\overline{P_G}$ for long symbol-based codes; this is useful because $1/\mu$ is a lower bound on $\overline{P_G}$.

**Computing the List of Candidates**

So far we have bounded the number of candidate codewords for any $(t+1)$-symbol DUE; we now show how to find these candidates. The candidate codewords $\Psi(\overline{\mathbf{c}})$ for any $(t+1)$-symbol DUE received string $\overline{\mathbf{c}}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\overline{\mathbf{c}}$. More formally, let subscripts be used to index symbols in a vector, starting from the most-significant position. Then

$$\Psi(\overline{\mathbf{c}}) = \mathbf{c} \cup \{\mathbf{c}' \in \mathcal{C} : \Delta_q(\mathbf{c}' - \mathbf{c}) = d_{min}, \mathbf{c}'_i = \overline{\mathbf{c}}_i \ \forall i \text{ where } \mathbf{e}_i \neq 0\}. \tag{2.4}$$

*Proof.* For a codeword $\mathbf{c}' \in \mathcal{C}$ to be a candidate codeword, due to linearity, it must be that $\Delta(\mathbf{c}', \overline{\mathbf{c}}) = \Delta(\mathbf{c}', \mathbf{c}+\mathbf{e}) = \Delta(\mathbf{c}'-\mathbf{c}, \mathbf{e}) = t+1$. Since $d_{min} = 2t+2$, the only codewords $\mathbf{c}'$ other than $\mathbf{c}$ that satisfy the previous equation also satisfy $wt(\mathbf{c}'-\mathbf{c}) = d_{min}$ and $wt(\mathbf{c}'-\mathbf{e}) = d_{min}/2$. Thus a minimum weight codeword $\mathbf{c}'$ is a candidate codeword if and only if $t+1$ of

**Algorithm 1** Compute list of candidate codewords $\Psi(\overline{\mathbf{c}})$ for a $(t+1)$-symbol DUE $\overline{\mathbf{c}}$ in a linear $(t)$SC$(t+1)$SD code with parameters $[n, k, d_{min}]_q$

```
for i = 1 : n do
    for j = 1 : q − 1 do
        p ← j ∗ e_i
        y ← c̄ + p
        if Decoder(y) not DUE then
            c′ ← Decoder(y)
            if c′ ∉ Ψ(c̄) then
                Ψ(c̄) ← Ψ(c̄) ∪ c′
            end if
        end if
    end for
end for
```

the non-zero elements in $\mathbf{c}' - \mathbf{c}$ are in the same location as the $t+1$ elements in $\mathbf{e}$. □

Notice that this equation depends on the error $\mathbf{e}$ and original codeword $\mathbf{c}$, but we only know the received string $\overline{\mathbf{c}}$.

Fortunately, there is a simple and intuitive algorithm (shown in Algorithm 1) to find the list of candidate codewords $\Psi(\overline{\mathbf{c}})$ with runtime complexity $O(nq/t)$. The essential idea is to try every possible single symbol *perturbation* $\mathbf{p}$ on the received string. Each *perturbed string* $\mathbf{y} = \overline{\mathbf{c}} + \mathbf{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix $\mathbf{H}$ ($O(rn\log(q))$ bits of storage). The subscripts of error-vectors in the algorithm indicate the symbol positions of errors, but not their $q$-ary values. For example, $\mathbf{e}_3$ corresponds to $[00100\ldots0]$. Any $\mathbf{y}$ characterized as a CE produces a candidate codeword from the decoder output.

### Pruning Candidates using a Lightweight Hash

In systems that require high reliability and availability, we propose to optionally prune the list of candidate codewords using a lightweight second-tier hash of several codewords grouped together in a cacheline. This would increase the success rate of SDECC while dramatically reducing the risk of MCEs. Several previous works [15, 16, 17, 18] have also proposed the

use of RAID-like multi-tier codes, but using traditional checksums instead than hashes.

We observe that second-tier hashes can also be used to prune a list of candidate codewords for a DUE. For instance, we can compute a $h$-bit *original hash* of the $\ell \times b$ total message bits of a cacheline when it is written to memory, where $\ell$ is the total cacheline size in symbols. Then when a DUE occurs, after the candidate messages are found using Algorithm 1, we can compute in software the *candidate hashes* for each *candidate cacheline* and compare them with the original that is read from memory. On average for a universal hash function, the number of *incorrect* candidates $|\Psi(\overline{\mathbf{c}})| - 1$ will be reduced by a factor of $2^h$. In most cases, only one candidate will match the original hash and we can fully correct the DUE; there is a chance of hash collision, in which case the number of candidates is still reduced but not down to one.

Errors in the original hash can cause candidate pruning to fail. However, this is a concern only when there is simultaneously both a $(t+1)$-symbol DUE $\Psi(\overline{\mathbf{c}})$ in one of the cacheline codewords and an error in the hash. Although we consider this situation to be very unlikely, there are two possible outcomes for a universal hash. In the first outcome, the hash cannot prune the list of candidates because no candidates' computed hashes match. This case is fairly benign: SDECC just falls back to the full list of candidates. The probability of this lower bounded by

$$(2^h - |\Psi(\overline{\mathbf{c}})|)/(2^h - 1). \tag{2.5}$$

In the second outcome, the corrupted hash collides with the computed hash of an incorrect candidate. Unfortunately, this case is not benign: it causes an MCE because the original message is mistakenly pruned along with other incorrect candidates. However, for all but the smallest hashes, the probability is much less than Outcome 1 and is upper bounded by

$$(|\Psi(\overline{\mathbf{c}})| - 1)/(2^h - 1). \tag{2.6}$$

We assume there is no more than one DUE per cacheline. Accordingly, we also assume the

Table 2.1: Summary of Code Properties

| **Code Class** | **Code Params.** $[n, k, d_{min}]_q$ | **Code Type** | **DUE Type** $(t+1)$ | **# Min. wt.** $W_q(d_{min})$ | **# DUEs** $N$ | **Avg. # Cand.** $\mu$ | **Prob. Rcov.** $\overline{P_G}$ |
|---|---|---|---|---|---|---|---|
| 32-bit SECDED | $[39, 32, 4]_2$ | Hsiao [19] | 2-bit | 1363 | 741 | 12.04 | 8.50% |
| 32-bit SECDED | $[39, 32, 4]_2$ | Davydov [20] | 2-bit | 1071 | 741 | 9.67 | 11.70% |
| 64-bit SECDED | $[72, 64, 4]_2$ | Hsiao [19] | 2-bit | 8404 | 2556 | 20.73 | 4.97% |
| 64-bit SECDED | $[72, 64, 4]_2$ | Davydov [20] | 2-bit | 6654 | 2556 | 16.62 | 6.85% |
| 32-bit DECTED | $[45, 32, 6]_2$ | – | 3-bit | 2215 | 14190 | 4.12 | 28.20% |
| 64-bit DECTED | $[79, 64, 6]_2$ | – | 3-bit | 17404 | 79079 | 5.40 | 20.53% |
| 128-bit SSCDSD | $[36, 32, 4]_{16}$ | Kaneda [21] | 2-sym. | 56310 | 141750 | 3.38 | 39.88% |

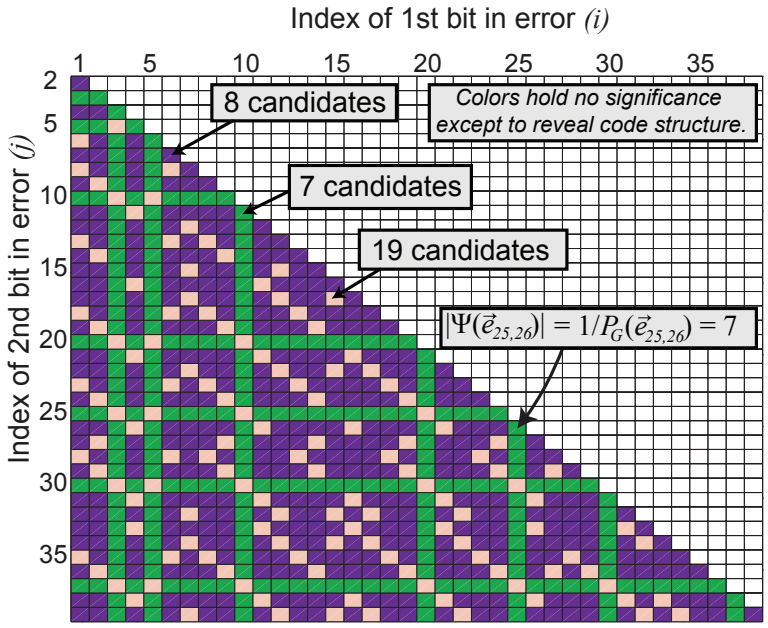



Figure 2.2: Number of candidate codewords $|\Psi(\mathbf{e}_{i,j})|$ for the Davydov $[39, 32, 4]_2$ SECDED code, where $i$ and $j$ represent the positions of the two bit-errors that cause a DUE.

hash is not corrupted in order to maintain a consistent fault model.

### 2.3.2 SDECC Analysis of Existing ECCs

Code constructions exhibit structural properties that affect the number of candidate codewords $|\Psi(\mathbf{e})|$. Certain combinations of error positions produce fewer candidate codewords than others. This favors recovery of certain errors even if one simply guesses from the corresponding list of candidate codewords. In fact, distinct code constructions with the same $[n, k, d_{min}]_q$ parameters can have different values of $\mu$ and distributions of $|\Psi(\mathbf{e})|$.

We apply the SDECC theory to seven code constructions of interest in this paper:

SECDED, DECTED, and SSCDSD (ChipKill-Correct) constructions with typical message lengths of 32, 64, and 128 bits. Table 2.1 lists properties that we have derived for each of them. Most importantly, the final column lists $\overline{P_G}$ — the *random* baseline probability of successful recovery without SI.

These probabilities are far higher than the naïve approaches of guessing randomly from $q^k$ possible messages or from the $N$ possible ways to have a DUE. Thus, our approach can handle DUEs in a more optimistic way than conventional ECC approaches.

### SECDED and DECTED

The class of SECDED codes ($t = 1$, $q = 2$, $d_{min} = 4$) is simple and effective against random radiation-induced soft bit-flips in memory. Hsiao's $[39, 32, 4]_2$ and $[72, 64, 4]_2$ constructions are the most common implementations of SECDED because they minimize the number of decoder logic gates [19]. Davydov proposed alternative and more structured SECDED codes that instead minimize the probability of an MCE when there is a triple-bit error ($wt_q(\mathbf{e}) = 3$) by minimizing $W_2(4)$ [20]. We find that Davydov codes have an additional advantage in context of SDECC: Lemma 2 tells us that these Davydov SECDED constructions also minimize the average number of candidate codewords $\mu$. This can lend them an advantage for heuristic recovery. Figure 2.2 depicts how the structure of the $[39, 32, 4]_2$ Davydov construction determines the number of candidate codewords for all $N$ possible DUE patterns. For the SECDED codes in this paper, the average number of candidate codewords $\mu$ ranges from 9.67 to 20.73, as shown in Table 2.1.

DECTED codes ($t = 2$, $q = 2$, $d_{min} = 6$) can correct random 2-bit errors and detect 3-bit errors. While they are not typically used in commodity memory systems due to high overheads, they attract continued interest by industry and researchers. For this work, we simply add one extra parity bit to the $[127, 113, 5]_2$ and $[63, 51, 5]_2$ BCH codes [22] and then truncate them to obtain our own $[45, 32, 6]_2$ and $[79, 64, 6]_2$ DECTED constructions, respectively. For the DECTED codes in this paper, a baseline random-candidate recovery

policy has around a 20-30% chance of success.

#### SSCDSD (ChipKill-Correct)

SSCDSD codes with 4-bit symbols ($t = 1$, $q = 16$, $d_{min} = 4$) are a non-binary equivalent of SECDED codes. We use Kaneda's Reed-Solomon-based $[36, 32, 4]_{16}$ construction [21]. Messages are 128 bits long and codewords are 144 bits long, so they are convenient to deploy in industry-standard DDRx-based DRAM systems that are 72 bits wide. When two DRAM channels are run in lockstep with x4 DRAM chips, $[36, 32, 4]_{16}$ SSCDSD codes have the *ChipKill-Correct* property [23]. This is because they can completely correct any errors resulting from a single-chip failure, and detect any errors caused by a double-chip failure. We find that despite there being 141750 possible ways to have a double-chip DUE, on average, there are just 3.38 candidate codewords per DUE, with the random-candidate chance of success being 39.88%. Thus, we expect ChipKill to deliver the best results in our evaluation.

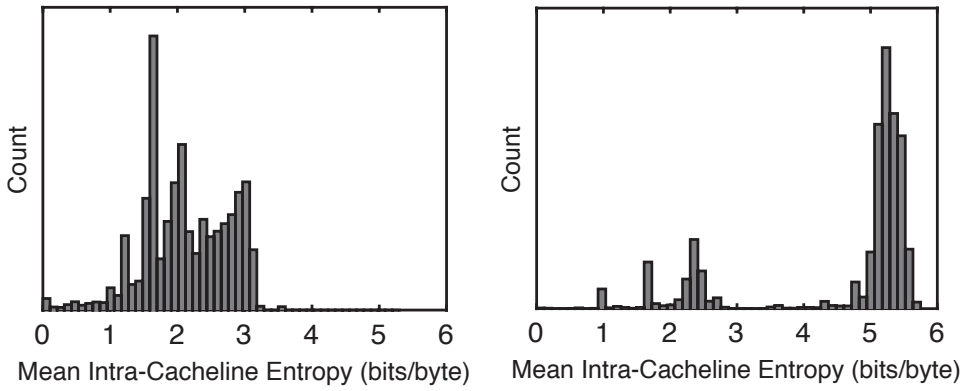
## 2.4 Experimental Results

### 2.4.1 Recovery Policy

In this section, we focus on recovery of DUEs in data (i.e., memory reads due to processor loads) because they are more vulnerable than DUEs in instructions. There are potentially many sources of SI for recovering DUEs. Based on the notion of *data similarity*, we propose a simple but effective data recovery policy called *Entropy-Z* that chooses the candidate that minimizes overall cacheline Shannon entropy.

Entropy is one of the most powerful metrics to measure data similarity. We make two general observations about the prevalence of low data entropy in memory.

1. There are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte,




(a) `464.h264ref` (integer) (b) `444.namd` (float)

Figure 2.3: Byte-granularity entropy distributions of 64-byte dynamic cacheline read data.

halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs).

2. In addition to spatial and temporal locality in their memory access patterns, applications have inherent *value locality* in their data, regardless of their hardware representation. For example, an image-processing program is likely to work on regions of pixels that exhibit similar color and brightness, while a natural language processing application will see certain characters and words more often than others.

Similar observations have been made to compress memory [6, 7, 8, 24, 25, 26].

We observe low byte-granularity intra-cacheline entropy throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite. Let $P(X)$ be the normalized relative frequency distribution of a $\ell \times b$-bit cacheline that has been carved into equal-sized $Z$-bit symbols, where each symbol $\chi_i$ can take $2^Z$ possible values. Entropy symbols are not to be confused with the codeword symbols, which can also be a different size. Then we compute the $Z$-bit-granularity `entropy` as follows:

$$\texttt{entropy} = -\sum_{i=1}^{\ell \times b/Z} P(\chi_i) log_2 P(\chi_i). \tag{2.7}$$

Consider two representative examples for $Z = 8$ and $\ell \times b = 512$ bits in Figure 2.3. The

**Algorithm 2** *Entropy-Z* data recovery policy

**Input:** $\Psi(\overline{\mathbf{c}})$, $L_n$, `PanicThreshold` value
**Output:** $\mathbf{m}_{\text{target}}$, `SuggestToPanic` value
  $M \leftarrow \Psi(\overline{\mathbf{c}})$ with the $r$ parity symbols stripped
  $L_k \leftarrow L_n$ with the $r$ parity symbols stripped //Extract cacheline SI
  Declare candidate entropy list `entropy` with $|M|$ elements
  **for** $i = 1 : |M|$ **do**
    `entropy`$[i] \leftarrow Z$-bit calculation for candidate cacheline //Eqn. 2.7
  **end for**
  $\texttt{entropy}_{\text{min}} \leftarrow \min_i(\texttt{entropy})$
  $i_{\text{min}} \leftarrow \text{argmin}_i(\texttt{entropy})$
  $\mathbf{m}_{\text{target}} \leftarrow M[i_{\text{min}}]$
  **if** tie for $\texttt{entropy}_{\text{min}}$ or $\text{mean}_i(\texttt{entropy}) >$ `PanicThreshold` **then**
    `SuggestToPanic` $\leftarrow$ `True`
  **else**
    `SuggestToPanic` $\leftarrow$ `False`
  **end if**

maximum possible intra-cacheline entropy here is six bits/byte because there can be only $2^6 = 64$ distinct byte values in a cacheline; anything less can be exploited as SI by SDECC recovery. We find that although floating-point values tend to have higher entropy *within* a word compared to integer values, entropy *between* neighboring words is often comparable. The average intra-cacheline byte-level entropy of the SPEC CPU2006 suite to be 2.98 bits/byte (roughly half of maximum).

### Entropy-Z Policy

We leverage these observations using our proposed data recovery policy, described in Algoritm 2. Essentially, with this policy, SDECC chooses the candidate message that minimizes overall cacheline entropy. We mitigate the chance that our policy chooses the wrong candidate message by deliberately forcing a *panic* whenever there is a tie for minimum entropy or if the mean cacheline entropy is above a specified threshold `PanicThreshold`. The downside to this approach is that some forced panics will be false positives, i.e., they would have otherwise recovered correctly.

In the rest of the section, unless otherwise specified, we use $Z = 8$ bits, $\ell \times b = 512$ bits

and `PanicThreshold` = 4.5 bits (75% of maximum entropy), which we determine to work well across a range of applications. Additionally, as we show later, the *Entropy-8* policy performs very well compared to several alternatives.

### 2.4.2 Methodology

We evaluate the impact of SDECC on system-level reliability through a comprehensive error injection study on memory access traces. Our objective is to estimate the fraction of DUEs in memory that can be recovered correctly using the SDECC architecture and policies while ensuring a minimal risk of MCEs.

The SPEC CPU2006 benchmarks are compiled against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [9] using the official tools [27]. Each benchmark is executed on top of the RISC-V proxy kernel [28] using the Spike simulator [29] that we modified to produce representative memory access traces. We only include the 20 benchmarks which successfully ran to completion. Each trace consists of randomly-sampled 64-byte demand read cachelines, with an average interval between samples of one million accesses.

Each trace is analyzed offline using a MATLAB model of SDECC. For each benchmark and ECC code, we randomly choose 1000 $q$-ary messages from the trace, encode them, and inject $min(1000, N)$ randomly-sampled $(t+1)$-symbol DUEs. For each codeword/error pattern combination, we compute the list of candidate codewords using Algorithm 1 and apply the data recovery policy using Algorithm 2. For Algorithm 2, we are given a $q$-ary list of $n$-symbol candidate codewords $\Psi(\overline{\mathbf{c}})$, a $q$-ary list of $n$-symbol error-free neighboring cacheline codewords $L_n$ (the SI), and a `PanicThreshold` value, and we produce a $q$-ary $k$-symbol recovery target message $\mathbf{m}_{\text{target}}$ and a flag `SuggestToPanic`. A *successful recovery* occurs when the policy selects a candidate message that matches the original; otherwise, we either cause a *forced panic* or recovery fails by accidentally inducing an MCE. Variability in the reported results is negligible over many millions of individual experiments.

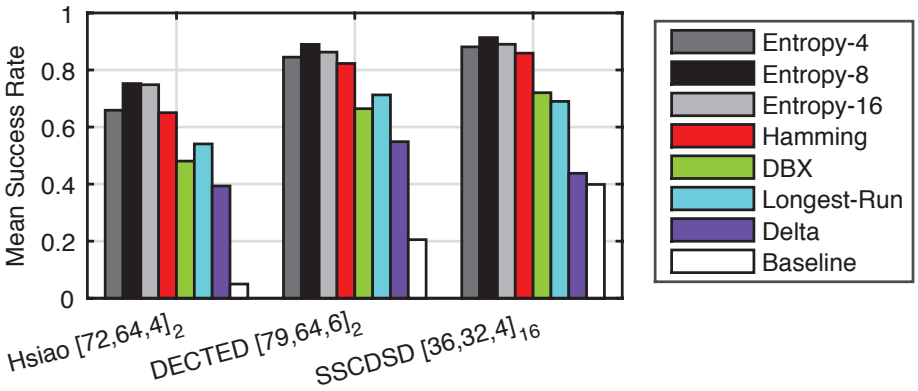



Figure 2.4: Comparison of raw success rate (no forced panics) for different SDECC data recovery policies averaged over all benchmarks. No hashes are used.

Note that the *absolute* error magnitudes for DUEs and SDECC's impact on *overall* reliability should not be compared directly between codes with distinct $[n, k, d_{min}]_q$ (e.g., a double-bit error for SECDED is very different from a double-chip DUE for ChipKill). Rather, we are concerned with the *relative* fraction of DUEs that can be saved using SDECC for a given ECC code. For evaluations that include second-tier hashes we assume there is no error in the hash itself, as explained earlier.

### 2.4.3 Comparison of Recovery Policies

We first compare the *raw* successful recovery rates of six different policies for three ECCs *without including any forced panics nor any second-tier hash.* Thus any un-successful recovery here is an MCE. The raw success rate averaged over the SPEC CPU2006 suite for each policy is shown for three ECC constructions in Figure 2.4. The depicted baseline represents the average probability $\overline{P_G}$ that we randomly select the original codeword out of a list of candidates for all possible DUEs.

The alternative policies under consideration are the following. *Hamming* chooses the candidate that minimizes the average binary Hamming distance to the neighboring words in the cacheline. *DBX* chooses the candidate that maximizes 0/1-run lengths in the output of the DBX transform [26] of the cacheline. *Longest-Run*, is inspired by frequent pattern

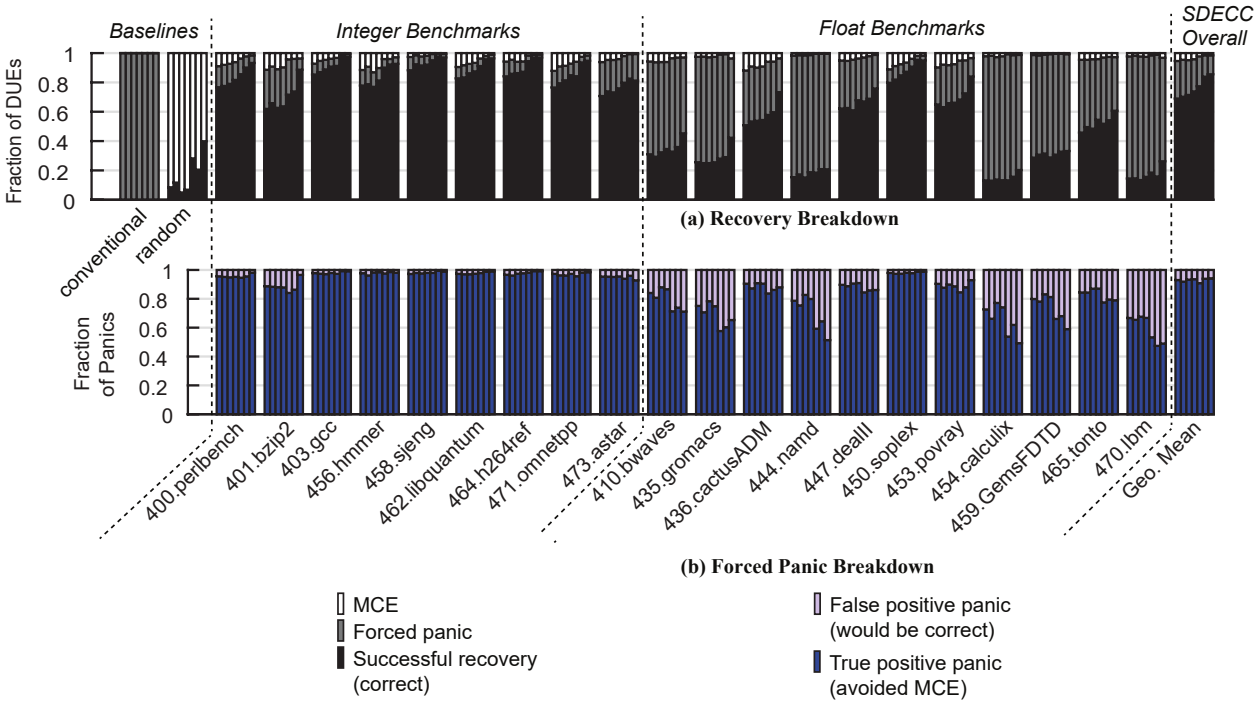



Figure 2.5: Detailed breakdown of DUE recovery results when forced panics are taken and no hashes are used. Results are shown for all seven ECC constructions, listed left to right within each cluster: $[39, 32, 4]_2$ Hsiao SECDED – $[39, 32, 4]_2$ Davydov SECDED – $[72, 64, 4]_2$ Hsiao SECDED – $[72, 64, 4]_2$ Davydov SECDED – $[45, 32, 6]_2$ DECTED – $[79, 64, 6]_2$ DECTED – $[36, 32, 4]_{16}$ SSCDSD ChipKill-Correct.

compression (FPC) [7] and chooses the candidate message with the longest run of 0/1 in the cacheline. *Delta* is inspired by frequent value compression (FVC) [6] and chooses the candidate message that minimizes the sum of squared integer deltas to the other words in the cacheline.

Our *Entropy-Z* policy variants recovered the most DUEs overall. Of these three, *Entropy-8* ($Z = 8$) performed better than $Z = 4$ and $Z = 16$. The 8-bit entropy symbol size performs best because its alphabet size ($2^8 = 256$ values) matches well with the number of entropy symbols per cacheline (64) and with the byte-addressable memory organization. For instance, both *Entropy-4* and *Entropy-16* do worse than *Entropy-8* because the entropy symbol size results in too many aliases at the cacheline-level and because the larger symbol size is less efficient, respectively.

The other four policies all significantly under-performed our *Entropy-Z* variants, with

Table 2.2: Percent Breakdown of SDECC *Entropy-8* Policy without Hashes (S = success, P = forced panic, M = MCE)

| | **panics taken** | | | **panics not taken** | | | ***random* baseline** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **S** | **P** | **M** | **S** | **P** | **M** | **S** | **P** | **M** |
| ***conv.* baseline** | | 100 | – | | | | | | |
| $[39, 32, 4]_2$ **Hsiao** | 69.1 | 25.6 | 5.3 | 72.7 | – | 27.3 | 8.5 | – | 91.5 |
| $[39, 32, 4]_2$ **Davydov** | 70.3 | 25.2 | 4.5 | 76.0 | – | 24.0 | 11.7 | – | 88.3 |
| $[72, 64, 4]_2$ **Hsiao** | 71.6 | 23.7 | 4.7 | 75.3 | – | 24.7 | 5.0 | – | 95.0 |
| $[72, 64, 4]_2$ **Davydov** | 74.0 | 21.9 | 4.1 | 77.7 | – | 22.3 | 6.9 | – | 93.2 |
| $[45, 32, 6]_2$ **DECTED** | 77.5 | 20.3 | 2.2 | 85.5 | – | 14.5 | 28.2 | – | 71.8 |
| $[79, 64, 6]_2$ **DECTED** | 84.0 | 14.5 | 1.5 | 89.0 | – | 11.0 | 20.5 | – | 79.5 |
| $[36, 32, 4]_{16}$ **SSCDSD** | 85.7 | 12.8 | 1.5 | 91.5 | – | 8.5 | 39.9 | – | 60.1 |

the exception of *Hamming*. It performs nearly as well as the *Entropy-4* policy for integer workloads but fails on many low-entropy cases that have low Hamming distances.

Because *Entropy-8* performed the best for all benchmarks and for all ECC constructions, we exclusively use it in all remaining evaluations.

### 2.4.4 Recovery Results

#### Recovery Without Hash

Having established *Entropy-8* as the best recovery policy, we now consider the impact of its forced panics on the the successful recovery rate and the MCE rate. Again, we evaluate SDECC for each ECC using its conventional form, without any second-tier hashes to help prune the lists of candidates.

The overall results with forced panics *taken* (main results, gray cell shading) and *not taken* are shown in Table 2.2. The results for *Entropy-8* on three codes shown earlier in Figure 2.4 are repeated in this table for comparison (i.e., the corresponding success rates when panics are not taken). There are two baseline DUE recovery policies: *conventional* (always panic for every DUE) and *random* (choose a candidate randomly, i.e., $\overline{P_G}$).

We observe that when panics are taken the MCE rate drops significantly by a factor of up to 7.3× without significantly reducing the success rate. This indicates that our

`PanicThreshold` mechanism appropriately judges when we are unlikely to correctly recover the original information.

These results also show the impact of code construction on successes, panics, and MCEs. When there are fewer average candidates $\mu$ then we succeed more often and induce MCEs less often. The $[72, 64, 4]_2$ SECDED constructions perform similarly to their $[39, 32, 4]_2$ variants even though the former have lower baseline $\overline{P_G}$. This is a consequence of our *Entropy-8* policy: larger $n$ combined with lower $\mu$ provides the greatest opportunity to differentiate candidates with respect to overall intra-cacheline entropy. For the same $n$, however, the effect of SECDED construction is more apparent. The Davydov codes recover about 3-4% more frequently than their Hsiao counterparts when panics are not taken (similar to the baseline improvement in $\overline{P_G}$). When panics are taken, however, the differences in construction are less apparent because the policy `PanicThreshold` does not take into account Davydov's typically lower number of candidates.

We examine the breakdown between successes, panics, and MCEs in more detail. Figure 2.5 depicts the DUE recovery breakdowns for each ECC construction and SPEC CPU2006 benchmark when forced panics are taken. Figure 2.5(a) shows the fraction of DUEs that result in success (black), panics (gray), and MCEs (white). For clarity, the two baselines from Table 2.2 are repeated on the left and the same panic taken results from the table are repeated on the right (SDECC Overall). Figure 2.5(b) further breaks down the forced panics (gray from Figure 2.5(a)) into a fraction that are *false positive* (light purple, and would have otherwise been correct) and others that are *true positive* (dark blue, and managed to avoid an MCE). Each cluster of seven stacked bars corresponds to the seven ECC constructions.

We achieve much lower MCE rates than the *random* baseline yet also panic much less often than the *conventional* baseline for all benchmarks, as shown by Figure 2.5(a). Our policy performs best on integer benchmarks due to their lower average intra-cacheline entropy. For certain floating-point benchmarks, however, there are many forced panics because they frequently have high data entropy above `PanicThreshold` (e.g., as seen earlier with `444.namd`

Table 2.3: Prct. Breakdown of SDECC *Entropy-8* Policy with Hashes
(S = success, P = panic, M = MCE)

| **checksum size** | **panics taken** | | | **panics not taken** | | | ***random* baseline** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **S** | **P** | **M** | **S** | **P** | **M** | **S** | **P** | **M** |
| ***conv.* baseline** | – | 100 | – | | | | | | |
| $[72, 64, 4]_2$ **Hsiao – 2-bit DUEs** | | | | | | | | | |
| **none** | 71.6 | 23.7 | 4.7 | 75.3 | – | 24.7 | 5.0 | – | 95.0 |
| **4-bit** | 87.8 | 11.4 | 0.8 | 93.7 | – | 6.3 | 28.8 | – | 71.2 |
| **8-bit** | 98.56 | 1.36 | 0.08 | 99.4 | – | 0.6 | 86.6 | – | 13.3 |
| $[36, 32, 4]_{16}$ **SSCDSD ChipKill-Correct – 2-chip DUEs** | | | | | | | | | |
| **none** | 85.7 | 12.8 | 1.5 | 91.5 | – | 8.5 | 39.9 | – | 60.1 |
| **4-bit** | 98.05 | 1.86 | 0.09 | 99.2 | – | 0.8 | 77.0 | – | 23.0 |
| **8-bit** | 99.940 | 0.058 | 0.002 | 99.98 | – | 0.02 | 98.1 | – | 1.9 |
| **16-bit** | 99.9999 | 9e–5 | 0* | 100* | – | 0* | 99.992 | – | 0.008 |

**out of 20 million DUE trials*

in Figure 2.3b). A `PanicThreshold` of 4.5 bits for these cases errs on the side of caution as indicated by the false positive panic rate, which can be up to 50%. Without more side-information, for high-entropy benchmarks, we believe it would be difficult for any alternative policy to frequently recover the original information with a low MCE rate and few false positive panics.

With almost no hardware overheads, SDECC used with SSCDSD ChipKill can recover correctly from up to 85.7% of double-chip DUEs while eliminating 87.2% of would-be panics; this could improve system availability considerably. However, SDECC with ChipKill introduces a 1% risk of converting a DUE to an MCE. Without further action taken to mitigate MCEs, this small risk may be unacceptable when application correctness is of paramount importance.

### Recovery With Hash

The second-tier hash can dramatically reduce the SDECC panic and MCE rates by pruning the list of candidate messages before applying the recovery policy.

The recovery breakdowns for second-tier hashes per cacheline on overall MCE rates using $[72, 64, 4]_2$ Hsiao SECDED and $[36, 32, 4]_{16}$ SSCDSD ChipKill-Correct ECCs are shown in

Table 2.3. The results for the non-hash cases are repeated from Table 2.2 for comparison. We do not include 16-bit hashes for the SECDED code because they are unsupported in our architecture.

The results show that even a small 4-bit hash added to every cacheline can reduce the induced MCE rate by up to substantial 16.6×. When high reliability is required, we suggest to use SDECC with a hash of at least 8 bits. Using SDECC with ChipKill and an 8-bit hash, we successfully recovered from 99.940% of double-chip DUEs; with a 16-bit hash, no MCE occurred at all in 20 million trials.

SDECC with hashes can recover from nearly 100% of double-chip DUEs with 4× lower storage overhead than a pure DSC ECC solution and with no common-case performance or notable energy overheads.

## 2.5 Software-Defined Error-Localizing Codes

We now present a variant of SDECC that is more suitable for microcontroller-class IoT devices: *software-defined error-localizing code* (SDELC). Typically, either basic SED parity is used to detect random single-bit errors or a Hamming SEC code is used to correct them. Unfortunately, Hamming codes are expensive for small embedded memories: they require six bits of parity per memory word size of 32 bits (an 18.75% storage overhead). On the other hand, basic parity only adds one bit per word (3.125% storage overhead), but without assistance by other techniques it cannot correct any errors.

SDELC is a novel solution that lies in between these regimes. A key component is the new class of *Ultra-Lightweight Error-Localizing Codes* (ULELCs), explored in detail in Section 2.5.1. ULELCs have lower storage overheads than Hamming codes: they can detect and then *localize* any single-bit error to a chunk of a memory codeword. We construct distinct ULELC codes for instruction and data memory that allows a software-defined recovery policy to heuristically recover the error by applying different semantics depending on the error location. The policies leverage available *side-information* (SI) about memory contents to

choose the most likely *candidate codeword* resulting from a localized bit error. In this manner, we attempt to correct a majority of single-bit soft faults without resorting to a stronger and more costly Hamming code. SDELC can even be used to recover many errors using a basic SED parity code.

#### Error-Localizing Codes

In 1963, Wolf et al. introduced *error-localizing codes* (ELC) that attempt to detect errors and identify the erroneous fixed-length chunk of the codeword. Wolf established some fundamental bounds [30] and studied how to create them using the tensor product of the parity-check matrices of an error-detecting and an error-correcting code [31]. ELC has been adapted to byte-addressable memory systems [32] but until now, they had not gained any traction in the systems community. To the best of our knowledge, ELCs in the regime between SED and SEC capabilities have not been previously studied.

### 2.5.1 Ultra-Lightweight Error-Localizing Codes

Localizing an error is more useful than simply detecting it. If we determine the error is from a *chunk* of length $\ell$ bits, there are only $\ell$ *candidate codewords* for which a single-bit error could have produced the received (corrupted) codeword.

A naïve way of localizing a single-bit error to a particular chunk is to use a trivial segmented parity code, i.e., we can assign a dedicated parity-bit to each chunk. However, this method is very inefficient because to create $C$ chunks we need $C$ parity bits: essentially, we have simply split up memory words into smaller pieces.

We create simple and custom *Ultra-Lightweight* ELCs that – given $r$ redundant parity bits – can localize any single-bit error to one of $C = 2^r - 1$ possible chunks. This is because there are $2^r - 1$ distinct non-zero columns that we can use to form the parity-check matrix $\mathbf{H}$ for our ULELC (for single-bit errors, the error syndrome is simply one of the columns of $\mathbf{H}$). To create a ULELC code, we first assign to each chunk a distinct non-zero binary column vector

of length $r$ bits. Then each column of $\mathbf{H}$ is simply filled in with the corresponding chunk vector. Note that $r$ of the chunks will also contain the associated parity-bit within the chunk itself; we call these *shared chunks*, and they are precisely the chunks whose columns in $\mathbf{H}$ have a Hamming weight of 1. Since there are $r$ shared chunks, there must be $2^r - r - 1$ *unshared chunks*, which each consist of only data bits. Shared chunks are unavoidable because the parity bits must also be protected against faults, just like the message bits.

ULELCs form a middle-ground between basic parity SED error-detecting codes and Hamming SEC ECCs. In the former case, $r = 1$, so we have a $C = 1$ monolithic chunk ($\mathbf{H}$ is a row vector of all ones). In the latter case, $\mathbf{H}$ uses each of the $2^r - 1$ possible distinct columns exactly once: this is precisely the $(2^r - 1, 2^r - r - 1)$ Hamming code. An ULELC code has a minimum distance of two bits by construction to support detection and localization of single-bit errors. Thus, the set of candidate codewords must also be separated from each other by a Hamming distance of exactly two bits.

For an example of an ULELC construction, consider the following parity-check matrix with nine message bits and $r = 3$ parity bits:

$$\mathbf{H} = \begin{array}{c} \\ \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} S_1 & S_2 & S_3 & S_4 & S_4 & S_5 & S_6 & S_6 & S_7 & S_5 & S_6 & S_7 \end{array} \\ \begin{array}{cccccccccccc} d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array},$$

where $d_i$ represents the $i$th data bit, $p_j$ is the $j$th redundant parity bit, $c_k$ is the $k$th parity-check equation, and $S_l$ enumerates the distinct error-localizing chunk that a given bit belongs to. Because $r = 3$, there are $N = 7$ chunks. Bits $d_1, d_2,$ and $d_3$ each have the SEC property because no other bits are in their respective chunks. Bits $d_4$ and $d_5$ make up an unshared chunk $S_4$ because no parity bits are included in $S_4$. The remaining data bits belong to shared chunks because each of them also includes at least one parity bit. Notice that any data or

Table 2.4: Proposed 7-Chunk ULELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0 [9])

| **bit →** | 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | -1 -3 |
|---|---|---|---|---|---|---|---|---|
| **Type-U** | imm[31:12] | | | | | rd | opcode | parity |
| **Type-UJ** | imm[20\|10:1\|11\|19:12] | | | | | rd | opcode | parity |
| **Type-I** | imm[11:0] | | | rs1 | funct3 | rd | opcode | parity |
| **Type-SB** | imm[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | parity |
| **Type-S** | imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | parity |
| **Type-R** | funct7 | | rs2 | rs1 | funct3 | rd | opcode | parity |
| **Type-R4** | rs3 | funct2 | rs2 | rs1 | funct3 | rd | opcode | parity |

| **Chunk** | $C_1$ (shared) | $C_2$ (shared) | $C_3$ (shared) | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_3$ | $C_2$ | $C_1$ |
|---|---|---|---|---|---|---|---|---|---|---|
| **H** | 00000 | 00 | 11111 | 00000 | 111 | 11111 | 1111111 | 1 | 0 | 0 |
| | 00000 | 11 | 00000 | 11111 | 000 | 11111 | 1111111 | 0 | 1 | 0 |
| | 11111 | 00 | 00000 | 11111 | 111 | 00000 | 1111111 | 0 | 0 | 1 |

parity bits that belong to the same chunk $S_l$ have identical columns of $\mathbf{H}$, e.g., $d_7$, $d_8$, and $p_2$ all belong to $S_6$ and have the column $[0; 1; 0]$.

The two key properties of ULELC (that do not apply to generalized ELC codes) are: *(i)* the length of the data message is independent of $r$, and *(ii)* each chunk can be an arbitrary length. The freedom to choose the length of the code and chunk sizes allow the ULELC design to be highly adaptable. Additionally, ULELC codes can offer SEC protection on up to $2^r - r - 1$ selected message bits by having the unshared chunks each correspond to a single data bit.

### Recovering Errors in Instruction Memory

We describe an ULELC construction and recovery policy for dealing with single-bit soft faults in instruction memory. The code and policy are jointly crafted to exploit SI about the ISA itself. Our SDELC implementation targets the open-source and free 64-bit RISC-V (RV64G) ISA [9], but the approach is general and could apply to any other fixed-length or variable-length ISA.

Our ULELC construction for instruction memory has seven chunks that align to the finest-grain boundaries of the different fields in the RISC-V codecs. These codecs, the chunk assignments, and the complete parity-check matrix $\mathbf{H}$ are shown in Table 2.4. The bit

positions -1, -2, and -3 correspond to the three parity bits that are appended to a 32-bit instruction in memory. The `opcode`, `rd`, `funct3`, and `rs1` fields are the most commonly used – and potentially the most critical – among the possible instruction encodings, so we assign each of them a dedicated chunk that is unshared with the parity bits. The fields which vary more among encodings are assigned to the remaining three shared chunks, as shown in the figure. The recovery policy can thus distinguish the impact of an error in different parts of the instruction. For example, when a fault affects shared chunk $C_1$, the fault is either in one of the five MSBs of the instruction, or in the last parity bit. Conversely, when a fault is localized to unshared chunk $C_7$ in Table 2.4, the ULELC decoder can be certain that the `opcode` field has been corrupted. Our instruction recovery policy can decide which destination register is most likely for the instruction based on program statistics collected a priori via static or dynamic profiling (the SI). The instruction recovery policy consists of three steps.

- Step 1. We apply a software-implemented instruction decoder to filter out any candidate messages that are illegal instructions. Most bit patterns decode to illegal instructions in three RISC ISAs we characterized: 92.33% for RISC-V, 72.44% for MIPS, and 66.87% for Alpha. This can be used to dramatically improve the chances of a successful SDELC recovery.

- Step 2. Next, we estimate the probability of each valid message using a small precomputed lookup table that contains the relative frequency that each instruction appears. We find that the relative frequencies of legal instructions follow power-law distribution. This is used to favor more common instructions.

- Step 3. We choose the instruction that is most common according to our SI lookup table. In the event of a tie, we choose the instruction with the longest leading-pad of 0s or 1s. This is because in many instructions, the MSBs represent immediate values. These MSBs are usually low-magnitude signed integers or they represent 0-dominant

function codes.

If the SI is strong, then we would expect to have a high chance of correcting the error by choosing the right candidate.

### Recovering Errors in Data Memory

In general-purpose embedded applications, data may come in many different types and structures. Because there is no single common data type and layout in memory, we propose to simply use evenly-spaced ULELC constructions and grant the software trap handler additional control about how to recover from errors, similar to the general idea from SuperGlue [33].

When the application does not disable recovery nor specify a custom behavior, all data memory errors are recovered using a default error handler implemented by the library. The default handler computes the average Hamming distance to nearby data in the same 64-byte chunk of memory (similar to taking the intra-cacheline distance in cache-based systems). The candidate with the minimum average Hamming distance is selected. This policy is based on the observation that spatially-local and/or temporally-local data tends to also be correlated, i.e., it exhibits *value locality* [34] that has been used in numerous works for cache and memory compression [6, 7, 24]. The Hamming distance is a good measure of data correlation, as shown later in Figure 2.8. Note the very high rates of successful recovery when the average Hamming distance candidate score is relatively low.

The application-defined error handler can specify recovery rules for individual variables within the scope of the registered subroutine. For instance, an application may wish critical data structures to never be recovered heuristically; for these, the application can choose to force a crash whenever a soft error impacts their memory addresses. This is similar to annotation-based approaches taken by others for various purposes [35, 36, 37, 38, 39].

### 2.5.2 Experimental Results

To evaluate SDELC, Spike was modified to produce representative memory access traces of all 11 benchmarks as they run to completion. Each trace consists of randomly-sampled memory accesses and their contents. We then analyze each trace offline using a MATLAB model of SDELC. For each workload, we randomly select 1000 instruction fetches and 1000 data reads from the trace and exhaustively apply all possible single-bit faults to each of them.

We evaluate SDELC recovery of the random soft faults using three different ULELC codes ($r = 1, 2, 3$). Recall that the $r = 1$ code is simply a single parity bit, resulting in 33 candidate codewords. (For basic parity, there are 32 message bits and one parity bit, so there are 33 ways to have had a single-bit error.) For the data memory, the ULELC codes were designed with the chunks being equally sized: for $r = 2$, there are either 11 or 12 candidates depending on the fault position (34 bits divided into three chunks), while for $r = 3$ there are always five candidates (35 bits divided into seven chunks). For the instruction memory, chunks are aligned to important field divisions in the RV64G ISA. Chunks for the $r = 2$ ULELC construction match the fields of the Type-U instruction codecs (the `opcode` being the unshared chunk). Chunks for the $r = 3$ ULELC code align with fields in the Type-R4 codec (as presented in Table 2.4). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting.

The overall SDELC results are presented in Figure 2.6. The recovery rates are relatively consistent over each benchmark, especially for instruction memory faults, providing evidence of the general efficacy of SDELC. One important distinction between the memory types is the sensitivity to the number $r$ of redundant parity bits per message. For the data memory, the simple $r = 1$ parity yielded surprisingly high rates of recovery using our policy (an average of 68.2%). Setting $r$ to three parity bits increases the average recovery rate to 79.2% thanks to fewer and more localized candidates to choose from. On the other hand, for the

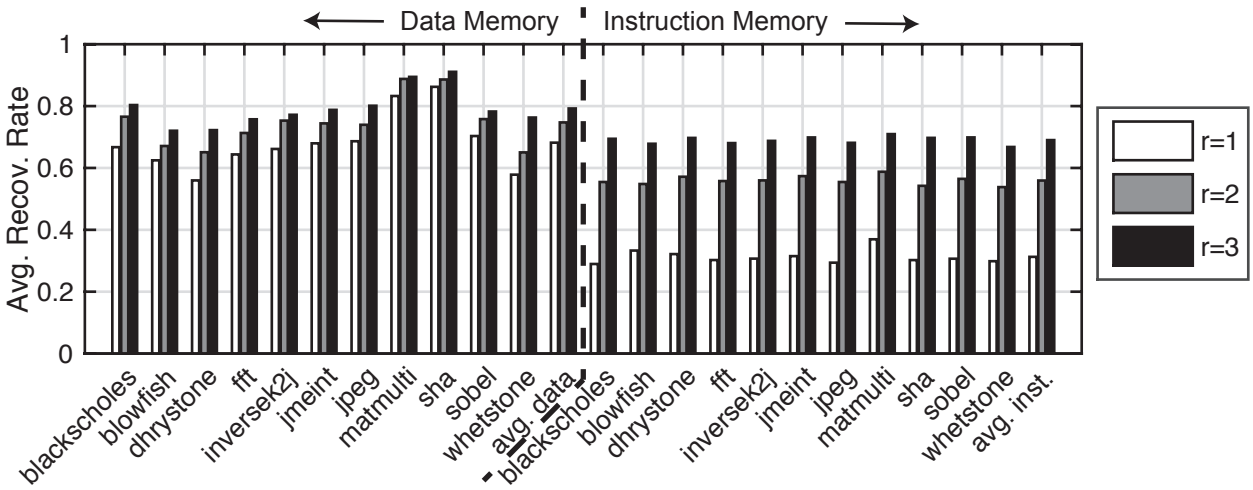



Figure 2.6: Average rate of recovery using SDELC from single-bit soft faults in data and instruction memory; $r$ is the number of parity bits in our ULELC construction.

instruction memory, the average rate of recovery increased from 31.3% with a single parity bit to 69.0% with three bits.

These results are a significant improvement over a guaranteed system crash as is traditionally done upon error detection using single-bit parity. Moreover, we achieve these results using no more than half the overhead of a Hamming SEC code, which can be a significant cost savings for small IoT devices. Based on our results, we recommend using $r = 1$ parity for data, and $r = 3$ ULELC constructions to achieve 70% recovery for both memories with minimal overhead. Next, we analyze the instruction and data recovery policies in more detail.

### Recovery Analysis

The average instruction recovery rate as a function of bit error position for all benchmarks is shown in Figure 2.7. Error positions -1, -2, and -3 correspond to the three parity bits in our ULELC construction from Table 2.4.

We observe that the SDELC recovery rate is highly dependent on the erroneous chunk. For example, errors in chunk $C_7$ – which protects the RISC-V `opcode` instruction field – have high rates of recovery because the power-law frequency distributions of legal instructions are

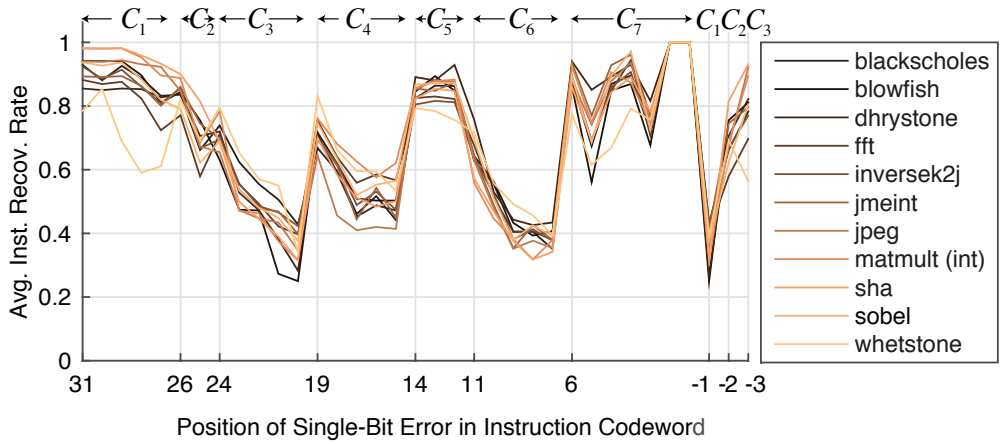
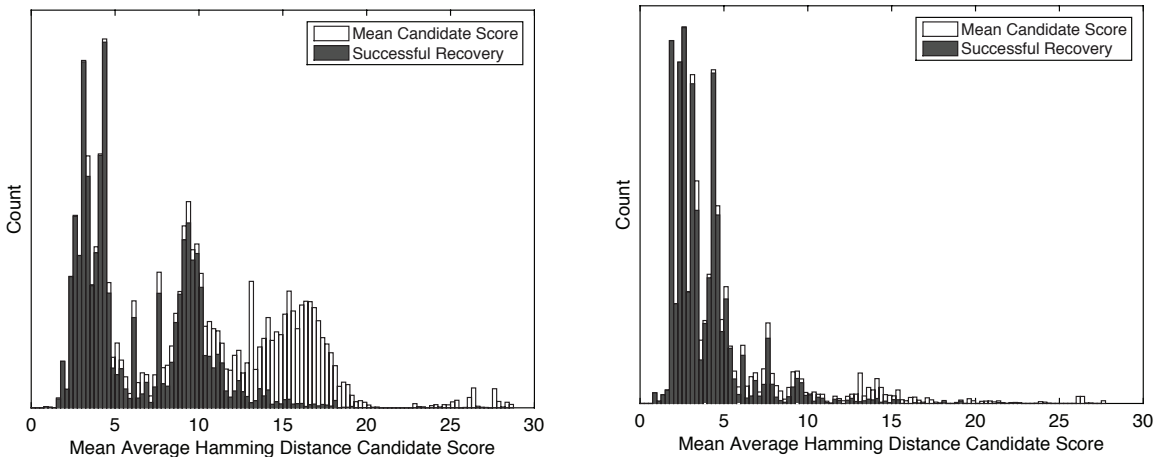



Figure 2.7: Sensitivity of SDELC instruction recovery to the actual position of the single-bit fault with the $r = 3$ ULELC construction.

a very strong form of side-information. Other chunks with high recovery rates, such as $C_1$ and $C_5$, are often (part of) the `funct2`, `funct7`, or `funct3` conditional function codes that similarly leverage the power-law distribution of instructions. Moreover, many errors that impact the `opcode` or function codes cause several candidate codewords to decode to illegal instructions, thus filtering the number of possibilities that our recovery policy has to consider. For errors in the chunks that often correspond to register address fields ($C_3$, $C_4$, and $C_6$), recovery rates are less because the side-information on register usage by the compiler is weaker than that of instruction relative frequency. However, errors towards the most-significant bits within these chunks recover more often than the least-significant bits because they can also correspond to immediate operands. Indeed, many immediate operands are low-magnitude signed or unsigned integers, causing long runs of 0s or 1s to appear in encoded instructions. These cases are more predictable, so we recover them frequently, especially for chunk $C_1$ which often represents the most-significant bits of an encoded immediate value.

The sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks is shown in Figure2.8. White bars represent the relative frequency that a particular Hamming distance score occurs in our experiments. The overlaid gray bars represent the fraction of those scores that we successfully recovered using our policy.

When nearby application data in memory is correlated, the mean candidate Hamming




(a) `blowfish` (b) `sha`

Figure 2.8: Sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks and $r = 1$ parity code.

distance is low, and the probability that we successfully recover from the single-bit soft fault is high using our Hamming distance-based policy. Because applications exhibit spatial, temporal, and value locality [34] in memory, we thus recover correctly in a majority of cases. On the other hand, when data has very low correlation – essentially random information — SDELC does not recover any better than taking a random guess of the bit-error position within the localized chunk, as expected.

## 2.6 Concluding Remarks

SDECC is a new approach to improve the resiliency of systems by recovering from a large fraction of memory DUEs. SDECC is based on the fundamental observation that when a DUE occurs, there are a small number of candidate codewords that can be computed, wherein one is practically guaranteed to be correct. Policies that leverage SI about memory contents can be used to recover successfully from many DUEs when they occur, while incurring negligible overheads in the common cases. In this chapter, we covered the underlying SDECC theory and presented thorough experimental results. We also expanded our SDECC framework to incorporate error-localizing codes, thus creating a lightweight framework appropriate for use

in IoT devices.

**Acknowledgment**

The majority of this chapter has been published in [40, 10, 41, 11] as a collaboration with Prof. Lara Dolecek, Prof. Puneet Gupta, Mark Gottscho, and Irina Alam. The author would like to specially thank Mark Gottscho for leading the experimental design and creating most of the resulting figures and tables.

# CHAPTER 3

# Unequal Message Protection

## 3.1 Introduction

In Chapter 2, we introduced *Software-Defined Error-Correcting Codes*, a class of heuristic techniques to recover from detected-but-uncorrectable errors (DUEs). Recall that SDECC can be considered as a highly practical list-decoding framework that utilizes any linear code capable of correcting $t$ errors and detecting $t+1$ errors. Traditionally, when a DUE occurs, the memory system will either crash or restore to a checkpoint [42]. In our SDECC framework, when a DUE occurs, we first compute a list of *candidate codewords*—the closest neighboring codewords—and then probabilistically decode based on available side-information. SDECC is applicable to a wide variety of memory applications and systems ranging from large-scale servers in data centers to embedded systems in Internet-of-Things devices.

In this chapter, we take a different approach and focus on the encoding-side of SDECC: instead of using side-information to heuristically decode, we *a priori* designate specific messages to have extra protection against errors. We designate two classes of messages, *normal* and *special*, and they are mapped to normal and special codewords, respectively. When dealing with the underlying data, we refer to the *messages*; when discussing error detection/correction capabilities we refer to the *codewords*. Within the SDECC framework, special codewords can be viewed as a set of codewords with the property that no two elements from the set are ever in the same candidate list, i.e., when a DUE occurs, there will never

be two or more special codewords among the neighboring codewords.

This type of *unequal message protection* (UMP) is fundamentally different from unequal error protection (UEP) [43], in which all codewords have extra protection for specific bit positions or certain error patterns (such as adjacent bit errors). UMP is a powerful approach when the special messages are chosen with regard to both the relative frequency and meaning of the stored data. In particular, UMP is useful when compression is not feasible, yet specific messages—or parts of specific messages—are very frequently stored/transmitted, as is the case in modern random-access memories [11, 10]. Additionally, given side-information about the system-level meaning of underlying messages, we may add extra protection to those messages whose miscorrections would be very costly.

In random-access memories, such as DRAM and SRAM [44, 45], the three most commonly used ECC classes are the single-bit parity-check code, the extended Hamming code, and the ChipKill (or equivalent) code. The choice of appropriate ECC class depends on many system-level requirements including latency, energy, storage overhead, etc. For each of these codes, we create an alternative UMP code with enhanced error-correcting features, while still using the same number of redundancy bits. For example, the extended Hamming code is capable of correcting any single-bit error and detecting any double-bit error; our UMP alternative code sacrifices the universal double-bit detection in order to grant double-bit correction to special codewords. For a given set of code parameters (i.e., code size, dimension, and detection/correction properties), our goal is to maximize the number of special messages.

The chapter is organized as follows. The remainder of this section is an overview of related work. In Section 3.2, we provide preliminaries, notation, and objectives. Both Sections 3.3 and 3.4 contain—for their respective codes—a derivation for the modified sphere-packing bounds on the number of special messages, an explicit code construction, a proof of correction properties, and a walk-through of the decoding process. Section 3.3 focuses on the UMP alternative to the parity-check code, in which we trade-off single-bit detection in favor of single-bit correction for special codewords. Additionally, we show how an additional re-

dundancy bit can be used to revive the single-error detection property for normal codewords. Section 3.4 deals with the UMP alternative for the extended Hamming code. In Section 3.5, we derive a novel programming bound for the number of special messages for our UMP codes. In Section 3.6, we discuss various strategies for the special message mapping and we investigate the benefits of our UMP codes on real-world memory benchmarks. We conclude in Section 3.7.

## 3.2 Preliminaries

### 3.2.1 Related Work

The majority of research into UEP codes has focused on *bit-wise* UEP, in which specific positions of a codeword are more robust to errors [43, 46]. Masnick and Wolf [43] created a framework for constructing linear bit-wise UEP codes, in which each bit in a codeword is assigned an error protection level. Bit-wise UEP codes are useful when errors in specific bit positions are more severe, e.g., the most-significant bit of a binary integer or the destination address header of a packet. A particular bit is then guaranteed to be decoded correctly if its error protection level is greater than or equal to total number of errors in the codeword.

Another type of UEP is *error-wise* UEP, in which specific error patterns are guaranteed to be correctable. Error-wise UEP codes are useful when bit-error locations are not independent. A code that is designed to correct burst errors can be thought of as an error-wise UEP code. For example, *single-error-correcting/double-error-detecting/double-adjacent-error-correcting* (SECDED-DAEC) codes guarantee correction in the case of a single-bit error or a double-bit error given that the erroneous bits are adjacent [47, 48]. Error-wise UEP codes can also be useful when different sections of the codeword are stored in different chips in computer hardware, in which case a faulty chip only causes errors on a specific subsection of the codeword [49, 23].

In this work, we focus on UMP, i.e., *message-wise* UEP, in which specific messages

have extra protection from errors. In this setting, Broade *et al.* used an information-theoretic approach to prove that it is possible to encode many special messages, even at rates approaching the channel capacity [50].

Shkel *et al.* [51] also examine the UMP problem. The main distinction between their work and ours is that they are concerned with producing information-theoretic bounds (achievability and converse) for such codes with average and maximal error probability over a probabilistic channel. Shkel *et al.* follow the line of work considered in Broade *et al.*, but they also look at the finite-length regime by applying the finite blocklength framework from Polyanski *et al.* [52] to the UMP setting. Nevertheless, theirs is a different setting compared to ours: we are interested in adversarial, not probabilistic, errors and we wish to produce short, explicit non-randomized code constructions.

Our approach also complements recent research on data compression in cache and main memory systems, an emerging topic that aims to meet the energy and storage demands brought upon by the exponential growth of produced data. Techniques include frequent value compression [6], frequent pattern compression [7], and base-delta-immediate compression [24]. These techniques add considerable complexity and overhead that may not always be tolerable; they nevertheless clearly demonstrate that there is a tremendous amount of correlation and redundancy inherent in the data used in main memory systems, which we seek to capitalize on, not for compression, but instead for resilience. This inherent data correlation is a key factor allowing our UMP coding framework to be innovative and useful.

This chapter contributes to a growing body of work that aims to exploit intrinsic source redundancy using coding techniques; e.g., Jiang *et al.* uses machine learning methods and the inherent redundancy in language-based sources to improve the rate of Polar codes and the performance of LDPC codes for non-volatile memories, [53, 54, 55]. This chapter also contributes to the classical topic of *joint source-channel coding*, in which a single encoder/decoder scheme is used to combine source and channel coding [56, 57]. Lastly, UMP is related to the *red alert problem*, in which a specific message not only requires a small

probability of missed detection, but also a small probability of false alarm [58]. In this work, we are not concerned with mitigating false alarms of our special messages.

### 3.2.2 Notation and Definitions

A code $\mathcal{C}$ is a subset of $\{1, 2, \ldots, q\}^n$, where $q \geq 2$ is the alphabet size and $n$ is the code length. We set $M = |\mathcal{C}|$ to be the cardinality of the code. As usual, for linear block codes the parameter $k$ is the code dimension (so that $M = q^k$ messages can be represented). Code $\mathcal{C}$ has minimum distance $d$ if $d = \min_{\mathbf{x},\mathbf{y}\in\mathcal{C},\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x}, \mathbf{y})$, where $d_H$ is the Hamming distance. If $\mathcal{C}$ has minimum distance $d$, it can correct $t = \lfloor (d-1)/2 \rfloor$ errors. We use the standard $(n, k, d)$ notation to denote code length, dimension, and minimum distance parameters. We use $d(\mathcal{C})$ as shorthand for the minimum distance of $\mathcal{C}$. In this chapter, logarithms are base 2.

When discussing the inputs and outputs to a channel, let $\mathbf{m}$ be the original message, $\mathbf{c}$ be the transmitted codeword, $\overline{\mathbf{c}}$ be the received (possibly erroneous) vector, $\hat{\mathbf{c}}$ be the decoded codeword, and $\hat{\mathbf{m}}$ be the final de-mapped message. Let $\mathbf{e}_i$ represent the error-vector with 0's at every index except index $i$, which has a value of 1. We use MATLAB-like notation to refer to parts of matrices, e.g., $\mathbf{H}(:, j)$ is the $j$th row of $\mathbf{H}$.

When dealing with cyclic codes, let $\alpha$ be a primitive element in GF($2^p$), $p \geq 1$, where the code length is $n = 2^p - 1$, and let $\phi_i(x)$ be the minimum polynomial of $\alpha^i$.

We partition the $M$ codewords into the sets $\mathcal{M}_i$, where the codewords in $\mathcal{M}_i$ have the property that they are guaranteed to be correctable in the presence of up to (and not more than) $i$ errors. Additionally let $M_i = |\mathcal{M}_i|$. The values of $i$ will depend on the code at hand. For example, the UMP alternative to the extended Hamming code partitions the $M$ codewords into the sets $\mathcal{M}_1$ and $\mathcal{M}_2$, in such a way that $M_2$ is maximized.

The basic approach in our UMP constructions involves the use of *subcodes*, in which every codeword in the subcode—the special codewords—is a member of a larger, overall code. All codewords not in the subcode are considered to be the normal codewords. There

are two key points worth noting about the code design. First, the overall code should not be a *perfect* code, i.e., there should be received (erroneous) vectors that are not inside the Hamming sphere of any codewords. This allows us to increase the Hamming spheres around our special codewords in order to capture these erroneous vectors. For example, if our overall code is a Hamming code—a perfect code—then increasing the Hamming spheres around any choice of special codewords would necessarily eliminate the single-error-correction guarantees of some of the normal codewords. However, the *extended* Hamming code is thus a *quasi-perfect* code and is a suitable choice for the overall code. Second, the subcode property of our coding framework must be designed at the generator matrix as opposed to the parity-check matrix. With the subcode structure explicitly represented in the generator matrix, we can encode our choice of special messages in a straightforward manner. Narrow-sense BCH codes are nested (have subcodes that are also BCH codes); however, the subcode structure is traditionally explicitly embedded in the parity-check matrix.

As an initial upper bound on the number of special codewords for our UMP parameters, we use the sphere-packing bound (also known as the Hamming bound). For a code $\mathcal{C}$, we can rewrite the sphere-packing bound as a function of the radii of the Hamming spheres (as opposed to the minimum distance of the code):

$$|\mathcal{C}| \leq \frac{q^n}{\sum_{j:\mathcal{M}_j \neq \{\emptyset\}} M_j \sum_{\ell=0}^{j} \binom{n}{\ell} (q-1)^\ell}.$$

Depending on the code at hand, we fix $n$, $k$, and the desired codeword partitions, in order to derive an upper bound on the number of special codewords. However, the sphere-packing bound bound is naïve in the sense that it does not take into account the geometry of the codespace. In Section 3.5, we derive a more sophisticated bound building upon Delsarte's linear programming bound [59].

## 3.3 Parity-Check UMP Alternative: (sm)SEC

### 3.3.1 Basic Properties

The single-bit parity-check code is a simple code that ensures every codeword has even weight. As a result, the minimum distance of the code is 2 and any single-bit error is detectable. A single-bit parity-check code is often systematic, but it can also be employed as the equivalent cyclic redundancy check CRC-1 code. In our UMP alternative, we give up single-error-detection for special message single-error-correction: *(sm)SEC*. We partition the $M$ codewords into the sets $\mathcal{M}_0$ and $\mathcal{M}_1$. Note that the codewords in $\mathcal{M}_0$ are still uniquely decodable in the presence of no errors, i.e., the code mapping is injective.

**Definition 1.** *A $(k+1, k)$ (sm)SEC code is a code whose codewords are partitioned into $\mathcal{M}_0$ and $\mathcal{M}_1$ with the following minimum distance properties:*

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_0,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 1, \tag{3.1}$$

$$\min_{\mathbf{x}\in\mathcal{M}_0,\mathbf{y}\in\mathcal{M}_1} d_H(\mathbf{x},\mathbf{y}) \geq 2, \tag{3.2}$$

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_1,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 3. \tag{3.3}$$

Let us examine a very simple partition of the $(5, 4)$ single-bit parity-check code.

**Example 1.** *$\mathcal{C} = \mathcal{M}_0 \cup \mathcal{M}_1$ is an $(5, 4)$ (sm)SEC code:*

$$\begin{aligned}\mathcal{M}_0 = \{&(00011), (00101), (00110), (01001),\\ &(01010), (01100), (01111), (10001),\\ &(10010), (10100), (10111), (11000),\\ &(11011), (11101)\},\\ \mathcal{M}_1 = \{&(00000), (11110)\}.\end{aligned}$$

Note that the partition in the above example meets the minimum distance requirements for a (sm)SEC code. Any received vector that is Hamming distance 1 away from a codeword

in $\mathcal{M}_1$ will be decoded to that codeword. The expansion of the Hamming spheres around the special codewords eliminates the single-error detection guarantee for codewords in $\mathcal{M}_0$; however, note that detection is still possible in many cases, just not *guaranteed* for all cases. For example, an error is detected if the transmitted and received words are $\mathbf{c} = (00011)$ and $\overline{\mathbf{c}} = (00111)$. Additionally, note that there is no possible partition of the $(5, 4)$ single-bit parity-check code that results in an (sm)SEC code with $M_1 > 2$. However, the following example demonstrates that we can construct a nonlinear code that has a higher number of special codewords.

**Example 2.** *$\mathcal{C} = \mathcal{M}_0 \cup \mathcal{M}_1$ is a $(5, 4)$ (sm)SEC code:*

$$\begin{aligned}\mathcal{M}_0 = \{&(11010), (11001), (10110), (10101),\\ &(01110), (01101), (10011), (01011),\\ &(10010), (01010), (10001), (01001), (11111)\},\\ \mathcal{M}_1 = \{&(00000), (11100), (00111)\}.\end{aligned}$$

To arrive at an initial upper bound on the number of possible special messages in a (sm)SEC code, we use the sphere-packing bound as follows ($|\mathcal{B}_i|$ is the size of a Hamming sphere with radius $i$):

$$\begin{aligned} M_0|\mathcal{B}_0| + M_1|\mathcal{B}_1| &\leq 2^n \\ \implies (2^k - M_1) + M_1(n+1) &\leq 2^n \\ \implies (2^k - M_1) + M_1(k+2) &\leq 2^{k+1} \\ \implies M_1 &\leq \frac{2^k}{k+1}. \end{aligned} \tag{3.4}$$

A comparison between the sphere-packing bound and our code constructions is provided later in Table 3.1.

### 3.3.2 Explicit Construction

Assume our message size, $k$, is a power of 2. The general strategy for this construction will be to use a shortened version of the extended Hamming code as the subcode of a single-bit parity-check code. Essentially, we are replacing some of the rows of the generator matrix for the single-bit parity-check code with those from a Hamming code, so that the submatrix and overall matrix have the desired minimum distance properties.

We begin the construction of our $(k+1, k)$ (sm)SEC code by first creating the generator matrix for the smallest Hamming code whose dimension is larger than $k$; since $k$ is a power of 2, this is the $(2k-1, 2k-\log(k)-2)$ Hamming code. Let $\phi_i(x)$ be the minimum polynomial of $\alpha^i$, where $\alpha$ is a primitive element of GF$(2k)$. The generator polynomial for the associated Hamming code is simply $g_1(x) = \phi_1(x)$. Let $\mathbf{G_1'}$ be the generator matrix whose rows are formed, as is usual with cyclic codes, by cyclic shifts of the coefficients of the generator polynomial.

We now *shorten* $\mathbf{G_1'}$ from a $(2k-\log(k)-2) \times (2k-1)$ matrix to a $(k-\log(k)-1) \times k$ matrix. This is accomplished by expurgating and puncturing the bottom $k-1$ rows and the right $k-1$ columns, respectively, yielding the following generator matrix for a shortened Hamming code:

$$\mathbf{G_1} = \begin{bmatrix} g_1(x) \\ xg_1(x) \\ x^2 g_1(x) \\ \vdots \\ x^{k-\log(k)-2} g_1(x) \end{bmatrix}. \tag{3.5}$$

Now we turn our attention to the overall code. We will add an overall parity-bit at a later step, so the generating polynomial for the remaining rows is simply the identity function.

At this stage, the remaining rows of the overall code are represented by

$$\mathbf{G_0} = \begin{bmatrix} x^{k-\log(k)-1} \\ x^{k-\log(k)} \\ x^{k-\log(k)+1} \\ \vdots \\ x^{k-1} \end{bmatrix}. \tag{3.6}$$

Let $\mathcal{C}_1$ and $\mathcal{C}_0$ be the codes represented by $\mathbf{G_1}$ and $\mathbf{G_0}$, respectively. At this point, we have $d(\mathcal{C}_1) = 3$ and $d(\mathcal{C}_0) = 1$, thus the addition of an overall parity bit increases each minimum distance by 1. Our final generator matrix for our (sm)SEC code is simply a vertical concatenation of matrices $\mathbf{G_1}$ and $\mathbf{G_0}$, extended with an overall parity bit (simply a '1' at the end of each row since each row has odd weight).

**Construction 1.** *With* $\mathbf{G_1}$ *and* $\mathbf{G_0}$ *defined in (3.5) and (3.6), respectively, we define the overall generator matrix:*

$$\mathbf{G} = \left[\begin{array}{c|c} \mathbf{G_0} & \mathbf{1} \\ \mathbf{G_1} & \mathbf{1} \end{array}\right].$$

Here and elsewhere in the paper, $\mathbf{1}$ represents a column vector of all 1's, of appropriate dimension dictated by the number of rows of the submatrix it is appended to. We place $\mathbf{G_0}$ above $\mathbf{G_1}$ so that the special mapping, explored further in Section 3.6, is more convenient.

**Theorem 1.** *Let* $\mathcal{M}_1$ *be the set of codewords corresponding to the set of messages that begin with* $\log(k)+1$ *0's. Then,* $\mathbf{G}$*, from Construction 1, is the generator matrix for a* $(k+1, k)$ *(sm)SEC code.*

*Proof.* We prove that the three conditions in Definition 1 are satisfied when the special messages are those that begin with $\log(k)+1$ 0's.

First, note that the special messages are only non-zero for values which multiply $[\mathbf{G_1}|\mathbf{1}]$ in the encoding step. Since $\mathbf{G_1}$ is the generator matrix for a shortened Hamming code,

Condition 3.3 is trivially satisfied.

For Condition 3.1 to be true, we need each of the $2^k$ messages to be encoded into unique codewords, i.e., if $\mathbf{m_1 G} = \mathbf{c}$ and $\mathbf{m_2 G} = \mathbf{c}$, then $\mathbf{m_1} = \mathbf{m_2}$. For this property to hold, we simply need the rows of $\mathbf{G}$ to be linearly independent. Individually, it is evident that $\mathbf{G_0}$ and $\mathbf{G_1}$ each have linearly independent rows. Let $\tilde{\mathbf{G}}$ denote $\mathbf{G}$ without the final column of 1's. Note that $\mathbf{G_0}$ can be expressed as $[\mathbf{0}|\mathbf{I}]$, where the identity matrix $\mathbf{I}$ has dimensions $(\log(k)+1)\times(\log(k)+1)$. Thus, to show that $\tilde{\mathbf{G}}$ has linearly independent rows, it is sufficient to show that no linear combination of rows in $\mathbf{G_1}$ results in a vector whose weights are entirely in the final $\log(k)+1$ bits. For a $(k+1,k)$ (sm)SEC code, the generator polynomial in $\mathbf{G_1}$ is written as $g_1(x) = \phi_1(x) = 1 + x + x^{(log(k)+1)}$. Since each row is a cyclic shift of $g_1(x)$, any combination of rows necessarily contains weights that span at least $\log(k)+2$ bits. Condition 3.3 is satisfied since $\tilde{\mathbf{G}}$ has linearly independent rows (the addition of the final 1's column does not affect this property).

Finally, given that Condition 3.1 is true, it is obvious that Condition 3.2 also holds since each row in $\mathbf{G}$ has even weight. □

**Corollary 1.** *Using* $\mathbf{G}$ *from Construction 1 with the mapping from Theorem 1, there are* $2^{k-(\log(k)+1)}$ *special messages, i.e.,* $M_1 = 2^{k-(\log(k)+1)}$.

In order to gauge the number of special messages of an (sm)SEC code, we introduce the following definition.

**Definition 2.** *An (sm)SEC code, with* $M_1$ *special messages, is bitwise optimal if there does not exist an (sm)SEC code with*

$$2^{\lceil \log(M_1) \rceil + 1}$$

*or more special messages.*

Comparing Corollary 1 to the sphere-packing bound in Equation (3.4), we arrive at the following result concerning the optimality of our (sm)SEC construction.

**Corollary 2.** *The code in Construction 1 is a bitwise optimal (sm)SEC code.*

*Proof.* We calculate the difference between the maximum number of information bits for $M_1$ from sphere-packing bound and the number of information bits for $M_1$ in our construction as follows:

$$\begin{aligned}
&\log\left(\frac{2^k}{k+1}\right) - \log\left(2^{k-(\log(k)+1)}\right) \\
&= k - \log(k+1) - (k - \log(k) - 1) \\
&= \log\left(\frac{k}{k+1}\right) + 1 < 1,
\end{aligned}$$

for positive values of $k$. □

As a concrete example, let us briefly walk-through the construction of the $(33, 32)$ (sm)SEC code. We first construct the cyclic generator matrix for the $(63, 57)$ Hamming code. Let $\alpha$ be a primitive element of GF$(2^6)$ such that $1 + x + x^6 = 0$, then our generator polynomial is simply $g_1(x) = \phi_1(x) = 1 + x + x^6$. We create the matrix $\mathbf{G_1}$ and then shorten the code to $(32, 26)$. Above it we add a $6 \times 6$ identity matrix, padded on the left with 0's, i.e., we concatenate $\mathbf{0}^{6\times 26}\mathbf{1}^{6\times 6}$ on top of $\mathbf{G_1}$. Lastly, we add a column of 1's.

Note that due to our even-parity construction, we have

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_1,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x}, \mathbf{y}) \geq 4.$$

Thus, our special codewords also have the property of double-error-detection.

### 3.3.3 Decoding

The decoding process for a (sm)SEC code is relatively simple. A slight caveat is that the overall code is not systematic. Thus, to retrieve $\hat{\mathbf{m}}$ from $\hat{\mathbf{c}}$ requires a de-mapping, which is accomplished by using the right pseudoinverse of $\mathbf{G}$ as follows: $\hat{\mathbf{c}}\mathbf{G}^{-1} = \hat{\mathbf{m}}$.

The cyclic construction of $\mathbf{G_1}$ was helpful for the proof of Theorem 1, but in practice we convert $\mathbf{G_1}$ to a systematic form by using elementary row operations, and easily obtain the corresponding parity-check matrix $\mathbf{H_1}$. Note that we don't need an overall $\mathbf{H}$ since the overall code is simply a single-bit parity-check.

There are three possible events in the decoding process. First, if the received vector $\overline{\mathbf{c}}$ has even weight, then it is a valid codeword and we declare $\hat{\mathbf{c}} = \overline{\mathbf{c}}$. Second, if the syndrome $\mathbf{s} = \mathbf{H}\overline{\mathbf{c}}^T$ is equal to column $j$ in $\mathbf{H_1}$, then we flip the $j$th bit in $\overline{\mathbf{c}}$, i.e., $\hat{\mathbf{c}} = \overline{\mathbf{c}} + \mathbf{e}_j$. Lastly, if $\mathbf{s}$ is non-zero and is not equal to a column in $\mathbf{H_1}$, then we declare a DUE, i.e., there was no special message reachable from an erroneous vector with a single bit-flip.

### 3.3.4 SED-(sm)SEC

Recall that for the (sm)SEC code we give up the single-error detection guarantee. This loss in protection might cause the trade-off to be undesirable for certain systems. However, we can extend the (sm)SEC code by a single redundancy bit in order to guarantee SED for normal codewords. The minimum distance requirements for SED-(sm)SEC are as follows.

**Definition 3.** *A $(k+2, k)$ SED-(sm)SEC code is a code whose codewords are partitioned into $\mathcal{M}_0$ and $\mathcal{M}_1$ with the following minimum distance properties:*

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_0,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 2, \tag{3.7}$$

$$\min_{\mathbf{x}\in\mathcal{M}_0,\mathbf{y}\in\mathcal{M}_1} d_H(\mathbf{x},\mathbf{y}) \geq 3, \tag{3.8}$$

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_1,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 3. \tag{3.9}$$

Using Construction 1, from the previous subsection, we meet the above requirements with the addition of a single bit that takes the value of 1 for normal messages and a value of 0 for special messages. The redundancy bit is simple to implement as it is just the logical NOR of the first $\log_2(k)+1$ bits in the message.

Comparing Definitions 1 and 3, notice that the requirements from Conditions (3.1) and (3.2) increase by 1 while the requirement from Condition (3.3) remains the same. With the addition of the nonlinear redundancy bit, it is obvious that any code satisfying (3.2) now satisfies (3.8). While the nonlinear parity bit does not affect (3.1), our (sm)SEC code from the previous subsection already satisfies (3.7) as it is an even weight code.

The sphere-packing bound requires modification to be applicable to the SED-(sm)SEC. Each special codeword still has $(n+1)$ $n$-dimensional points within its Hamming sphere. However, each point at a distance 1 away from a normal codeword is a distance 1 away from at most $n$ normal codewords. Each of these points can be thought of as being *shared* by at most $n$ normal codewords. Thus, each normal codeword has a claim to at least $(1/n)n = 1$ points (not including itself). Our modified sphere-packing bound is as follows:

$$\begin{aligned} 2M_0 + (n+1)M_1 &\leq 2^n \\ \implies 2(2^k - M_1) + M_1(k+3) &\leq 2^{k+2} \\ \implies M_1 &\leq \frac{2^{k+1}}{k+1}. \end{aligned} \tag{3.10}$$

The decoding process is slightly more involved than that of the (sm)SEC code so it is presented in algorithmic form. Let $\overline{\mathbf{c}}$ represent the received codeword, not including the nonlinear redundancy bit, and let $\eta$ represent the value of that bit. Additionally, let $\mathbf{s_1} = \mathbf{H_1}^T\overline{\mathbf{c}}$.

**Algorithm 3** Decoding algorithm for the SED-(sm)SEC code

1: **if** $wt(\overline{\mathbf{c}}) = 0 \pmod 2$ **then**
2: $\quad \hat{\mathbf{c}} \leftarrow \overline{\mathbf{c}}$
3: **else if** $\eta = 0$ **and** $\mathbf{s_1} = \mathbf{H_1}(:, j)$ **then**
4: $\quad \hat{\mathbf{c}} \leftarrow \overline{\mathbf{c}} + \mathbf{e}_j$
5: **else**
6: $\quad$ Declare DUE
7: **end if**

Even though for decoding purposes it is useful to view the nonlinear redundancy bit in

a unique light, it is not given a special channel and it is susceptible to a bit-flip in the same manner as any other bit in the codeword.

## 3.4 Hamming Code UMP Alternative: SEC-(sm)DEC

The extended Hamming code is a single-error-correcting/double-error-detecting (SECDED) code. We give up the universal DED guarantee in favor of granting special codewords double-error-correction (DEC). We thus partition the $M$ codewords into the sets $\mathcal{M}_1$ and $\mathcal{M}_2$. We call such a code a SEC-(sm)DEC (*single-error-correcting/special-message-double-error-correcting*) code. It is convenient to formally define the code in terms of the minimum Hamming distances between pairs of codewords.

**Definition 4.** *A SEC-(sm)DEC code is a code whose codewords are partitioned into* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *with the following minimum distance properties:*

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_1,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 3, \tag{3.11}$$

$$\min_{\mathbf{x}\in\mathcal{M}_1,\mathbf{y}\in\mathcal{M}_2} d_H(\mathbf{x},\mathbf{y}) \geq 4, \tag{3.12}$$

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_2,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 5. \tag{3.13}$$

We present the following simple example; we use the same codewords from the $(8,4)$ extended Hamming code, but we consider the Hamming sphere around the all-1's codeword and the all-0's codeword to have radius 2.

**Example 3.** $\mathcal{C} = \mathcal{M}_1 \cup \mathcal{M}_2$ *is an* $(8,4)$ *SEC-(sm)DEC code:*

$$\begin{aligned}\mathcal{M}_1 = \{&(11100001),(10011001),(01010101),(00101101),\\ &(00110011),(01001011),(10000111),(01111000),\\ &(10110100),(11001100),(11010010),(10101010),\\ &(01100110),(00011110)\},\end{aligned}$$

$$\mathcal{M}_2 = \{(00000000), (11111111)\}.$$

Our objective is to fully partition the code into the sets $\mathcal{M}_1$ and $\mathcal{M}_2$ and maximize $M_2$. That is, we require that every codeword is correctable given a single error, and we seek to maximize the number of codewords that are correctable in the presence of up to two errors.

To arrive at an upper bound on the number of possible special messages we use the sphere-packing bound as follows:

$$\begin{aligned} M_1|\mathcal{B}_1| + M_2|\mathcal{B}_2| &\leq 2^n \\ \implies (2^k - M_2)\sum_{j=0}^{1}\binom{n}{j} + M_2\sum_{j=0}^{2}\binom{n}{j} &\leq 2^n \\ \implies M_2 &\leq \frac{2^n - 2^k(n+1)}{\binom{n}{2}}. \end{aligned} \tag{3.14}$$

The resulting bound is intuitive: there are $2^n - 2^k(n+1)$ points outside of the radius-1 Hamming spheres, which can become radius-2 Hamming spheres with the addition of $\binom{n}{2}$ points. While the SEC-(sm)DEC code is meant as a direct alternative to the SECDED code, the above bound makes sense for any code with parameters $(n, k)$ with a redundancy level in between the respective Hamming code and $t = 2$ BCH code.

### 3.4.1 Explicit Construction

Once again, we assume our message length, $k$, is a power of 2. Thus, our SEC-(sm)DEC code has parameters $(k + \log(k) + 2, k)$. We take a similar approach to the (sm)SEC construction, but here the subcode is an extended $t = 2$ BCH code and the overall code is an extended Hamming code.

We first begin by creating the narrow-sense $t = 2$ BCH code with parameters $(2k-1, 2k-2\log(k)-3)$. The generator polynomial for the BCH code is $g_2(x) = LCM\{\phi_1(x), \phi_3(x)\}$, used to generate $\mathbf{G_2'}$.

Similarly to the previous case, we *shorten* $\mathbf{G_2'}$ from a $(2k-2\log(k)-3)\times(2k-1)$ matrix to a $(k-\log(k)-1)\times(k+\log(k)+1)$ matrix. This is accomplished by expurgating and puncturing the bottom $k-\log(k)-2$ rows and the right $k-\log(k)-2$ columns, respectively, yielding the following generator matrix for a shortened BCH code:

$$\mathbf{G_2} = \begin{bmatrix} g_2(x) \\ xg_2(x) \\ x^2g_2(x) \\ \vdots \\ x^{k-\log(k)-2}g_2(x) \end{bmatrix}. \tag{3.15}$$

We build the additional $\log(k)+1$ rows using $g_1(x) = \phi_1(x)$, the generator polynomial for the corresponding Hamming code:

$$\mathbf{G_1} = \begin{bmatrix} x^{k-\log(k)-1}g_1(x) \\ x^{k-\log(k)}g_1(x) \\ x^{k-\log(k)+1}g_1(x) \\ \vdots \\ x^{k-1}g_1(x) \end{bmatrix}. \tag{3.16}$$

We combine the matrices as before:

**Construction 2.** *With* $\mathbf{G_2}$ *and* $\mathbf{G_1}$ *defined in (3.15) and (3.16), respectively, we define the overall generator matrix:*

$$\mathbf{G} = \left[\begin{array}{c|c} \mathbf{G_1} & \mathbf{1} \\ \mathbf{G_2} & \mathbf{1} \end{array}\right].$$

An example of Construction 2 is shown in Figure 3.1, with $\mathbf{G_2}$ converted to systematic form for easier usage in practical systems. As with the (sm)SEC case, we have the following theorem and lemma.

**Theorem 2.** *Let* $\mathcal{M}_2$ *be the set of codewords corresponding to the set of messages that*

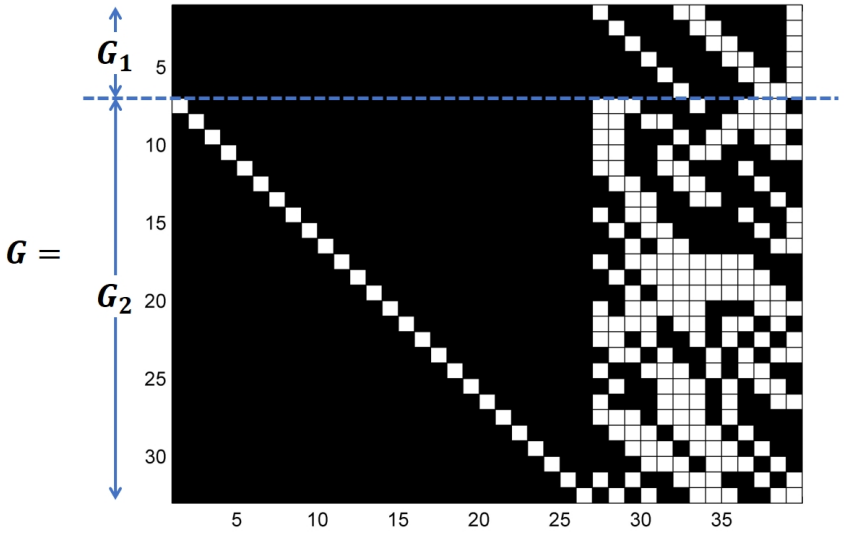



Figure 3.1: The generator matrix $\mathbf{G}$ for our $(39, 32)$ SEC-(sm)DEC code. The white and black squares represent 1s and 0s, respectively. Note that $\mathbf{G_2}$ has been converted to systematic form.

*begin with* $\log(k) + 1$ *0's. Then,* $\mathbf{G}$*, from Construction 2, is the generator matrix for a* $(k + \log(k) + 2, k)$ *SEC-(sm)DEC code.*

*Proof.* Similarly to the proof of Theorem 1, we need to prove that the conditions in Definition 3 are satisfied when the special messages are those that begin with $\log(k) + 1$ 0's. Since $\mathbf{G_2}$ is the generator matrix of a shortened $t = 2$ BCH code, it is trivially true that Condition 3.9 is true.

Once again, let $\tilde{\mathbf{G}}$ denote $\mathbf{G}$ without the final column of 1's. Recall that $g_1(x) = \phi_1(x)$ and $g_2(x) = LCM\{\phi_1(x)\phi_3(x)\}$. Since $\phi_1(x)$ and $\phi_3(x)$ are irreducible and distinct, we have $g_2(x) = \phi_1(x)\phi_3(x)$. Thus, a vector is a codeword of $\tilde{\mathbf{G}}$ (in polynomial form) if and only if it is divisible by $\phi_1(x)$; hence, $\tilde{\mathbf{G}}$ is the generator matrix for a shortened Hamming code with minimum distance 3 and Condition 3.7 is satisfied. The addition of the column of 1's makes every row have even weight, and thus Condition 3.8 is also true. □

**Corollary 3.** *Using* $\mathbf{G}$ *with the mapping from Theorem 1, there are* $2^{k-(\log(k)+1)}$ *special messages, i.e.,* $M_2 = 2^{k-(\log(k)+1)}$.

Table 3.1: Special Messages (measured in bits)

| | | Sphere-Packing Bound | | |
|---|---|---|---|---|
| **Message Bits ($k$)** | **Constructions 1 & 2** | (sm)SEC | **S**ED-(sm)SEC | **S**EC-(sm)DEC |
| 4 | 1 | 1.68 | 2.68 | 2 |
| 8 | 4 | 4.83 | 5.83 | 5.88 |
| 16 | 11 | 11.91 | 12.91 | 13.51 |
| 32 | 26 | 26.96 | 27.96 | 28.93 |
| 64 | 57 | 57.98 | 58.98 | 60.20 |

Similarly to Section 3.3.2, due to our even-parity construction, we have

$$\min_{\mathbf{x},\mathbf{y}\in\mathcal{M}_2,\mathbf{x}\neq\mathbf{y}} d_H(\mathbf{x},\mathbf{y}) \geq 6.$$

Thus, our special codewords also have the triple-error-detection property.

### 3.4.2 Decoding

As with the previous code, we focus on decoding $\hat{\mathbf{c}}$ from the received vector $\overline{\mathbf{c}}$; an additional de-mapping step with the pseudoinverse of $\mathbf{G}$ is required to arrive at $\hat{\mathbf{m}}$. We convert $[\mathbf{G_2}|\mathbf{1}]$ into systematic form, using elementary row operations, to easily retrieve the associated parity-check matrix, $\mathbf{H_2}$. Additionally, we convert $\mathbf{G}$ into systematic form to retrieve the overall parity-check matrix $\mathbf{H}$. Note that converting $\mathbf{G}$ to systematic form destroys the explicit subcode partition in $\mathbf{G}$, however, the associated parity-check matrix $\mathbf{H}$ can still be used to correct single-bit errors.

Let $\mathbf{s} = \mathbf{H}^T\overline{\mathbf{c}}$ and $\mathbf{s_2} = \mathbf{H_2}^T\overline{\mathbf{c}}$. The following pseudocode outlines the logical flow of the decoding process.

**Algorithm 4** Decoding algorithm for the SEC-(sm)DEC code

**if** $\mathbf{s} = 0$ **then**
  $\hat{\mathbf{c}} \leftarrow \overline{\mathbf{c}}$
**else if** $\mathbf{s} = \mathbf{H}(:, j)$ **then**
  $\hat{\mathbf{c}} \leftarrow \overline{\mathbf{c}} + \mathbf{e}_j$
**else if** $\mathbf{s_2} = \mathbf{H_2}(:, j) + \mathbf{H_2}(:, i)$ **then**
  $\hat{\mathbf{c}} \leftarrow \overline{\mathbf{c}} + \mathbf{e}_j + \mathbf{e}_i$
**else**
  Declare DUE
**end if**

The process above outlines the correct order of steps in the decoding process. There are a variety of physical implementations and algorithms to choose from for the BCH decoding process for the step involving $\mathbf{H_2}$.

### 3.4.3 SECDED-(sm)DEC

As in Section 3.3.4, we can use an additional nonlinear parity bit to create a code strictly better than the base code, i.e., not giving up the double-error detection guarantee for any codewords. The minimum distance requirements for SECSED-(sm)DEC are as follows.

**Definition 5.** *A $(k + \log(k) + 3, k)$ SECDED-(sm)DEC code is a code whose codewords are partitioned into $\mathcal{M}_1$ and $\mathcal{M}_2$ with the following minimum distance properties:*

$$\min_{\mathbf{x},\mathbf{y} \in \mathcal{M}_1, \mathbf{x} \neq \mathbf{y}} d_H(\mathbf{x}, \mathbf{y}) \geq 4, \tag{3.17}$$

$$\min_{\mathbf{x} \in \mathcal{M}_0, \mathbf{y} \in \mathcal{M}_1} d_H(\mathbf{x}, \mathbf{y}) \geq 5, \tag{3.18}$$

$$\min_{\mathbf{x},\mathbf{y} \in \mathcal{M}_1, \mathbf{x} \neq \mathbf{y}} d_H(\mathbf{x}, \mathbf{y}) \geq 5. \tag{3.19}$$

Using Construction 2, we meet the above requirements with the addition of a single bit that takes the value of 1 for normal messages and a value of 0 for special messages. As before, the redundancy bit is simple to implement as it is just the logical NOR of the first $\log_2(k) + 1$ bits in the message. The nonlinear parity-bit only the distances between normal

and special codewords, thus only affecting Condition 3.18; Conditions 3.17 and 3.19 were already satisfied by our original Construction 2.

## 3.5 Upper Bound on Special Codewords

We first recap our current results with Table 3.1, which compares the number of special messages for our constructions with their respective sphere-packing bounds. Each row is indexed by a value of $k$, and the second column represents the results from Corollaries 1 and 3. The third column helps to demonstrate Corollary 2, that the (sm)SEC code is bitwise optimal.

For traditional codes, the sphere-packing bound is not the tightest upper bound available in either the finite-length or asymptotic regimes. A better bound is provided in both cases by Delsarte's linear programming (LP) bound [59]. The LP bound considers the *distance distribution* vector $\mathbf{g} = (g_0, g_1, g_2, \ldots, g_n)$, where

$$g_i = |\{(\mathbf{x}, \mathbf{y}) : \mathbf{x}, \mathbf{y} \in \mathcal{C}, d_H(\mathbf{x}, \mathbf{y}) = i\}|/|\mathcal{C}|.$$

All values of $g_i$ are nonnegative, $g_0 = 1$, and $g_i = 0$ for $1 \leq i < d_{min}$. The remaining condition in the LP bound is $\mathbf{gQ} \geq 0$, where $\mathbf{Q}$ is the so-called *second eigenmatrix* of the Hamming association scheme on $\mathbb{F}_2^n$. Delsarte showed that $\mathbf{Q}$ can be formed by the relation $\mathbf{Q}_{i,j} = K_j(i)$, with the Krawtchouk polynomial defined as

$$K_k(x) = \sum_{j=0}^{k} (-1)^j \binom{x}{j} \binom{n-x}{k-j}. \tag{3.20}$$

Clearly, we have that $\sum_{i=0}^{n} g_i = |\mathcal{C}|$, and thus maximizing $\sum_{i=0}^{n} g_i$ also maximizes the size of the code.

We are ready to introduce our modified bound, first for the SEC(sm)DEC class of codes. For our purposes, we have two classes of codewords, regular codewords and special codewords,

Table 3.2: SEC-(sm)DEC Upper Bounds on $\log(M_2)$

| $(n, k)$ | Sphere-Packing Bound | Programming Bound |
|---|---|---|
| $(14, 8)$ | 7.11 | 7.00 |
| $(15, 8)$ | 8.09 | 7.99 |
| $(23, 16)$ | 14.72 | 14.56 |
| $(24, 16)$ | 15.74 | 15.56 |
| $(25, 16)$ | 16.90 | 16.00 |

and we create three separate distance distribution vectors. Distance distribution vectors $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$ contain distances between two regular codewords, one regular codeword and one special codeword, and two special codewords, respectively:

$$\begin{aligned} a_i &= |\{(\mathbf{x}, \mathbf{y}) : \mathbf{x}, \mathbf{y} \in \mathcal{M}_1, d_H(\mathbf{x}, \mathbf{y}) = i\}|/|\mathcal{C}|, \\ b_i &= |\{(\mathbf{x}, \mathbf{y}) : \ \mathbf{x} \in \mathcal{M}_1, \mathbf{y} \in \mathcal{M}_2 \text{ or} \\ &\qquad \mathbf{x} \in \mathcal{M}_2, \mathbf{y} \in \mathcal{M}_1, d_H(\mathbf{x}, \mathbf{y}) = i\}|/|\mathcal{C}|, \\ c_i &= |\{(\mathbf{x}, \mathbf{y}) : \mathbf{x}, \mathbf{y} \in \mathcal{M}_2, d_H(\mathbf{x}, \mathbf{y}) = i\}|/|\mathcal{C}|. \end{aligned}$$

Recall that our goal is to maximize $M_2$. Summing over all the entries in $\mathbf{c}$, we have:

$$\sum_{i=0}^{n} c_i = \frac{(M_2)^2}{|\mathcal{C}|} \implies M_2 = \sqrt{2^k \sum_{i=0}^{n} c_i},$$

and thus, for given $n$ and $k$, our objective function is to maximize $\sum_{i=0}^{n} c_i$. Note that $\mathbf{a}$, $\mathbf{c}$, and $\mathbf{a} + \mathbf{b} + \mathbf{c}$ are valid (scaled) distance distribution vectors of codes. Due to the multiple distance distribution vectors, there are a number of substantial differences in the constraints of our programming bound and those from Delsarte's LP bound.

We first establish *inequality* constraints. Our first three inequality constraints are $\mathbf{aQ} \geq 0$, $\mathbf{cQ} \geq 0$, and $(\mathbf{a} + \mathbf{b} + \mathbf{c})\mathbf{Q} \geq 0$, where $\mathbf{Q}$ is the same eigenmatrix based on the Krawtchouk polynomial in Equation 3.20. Similarly to the LP bound, we require all $a_i$, $b_i$ and $c_i$ to be nonnegative.

We now establish the *equality* constraints. We have $\sum_{i=0}^{n} b_i = \frac{2M_1M_2}{|\mathcal{C}|}$. Thus our total

codewords condition is:

$$\sum_{i=0}^{n}(a_i + b_i + c_i) = \frac{(M_1)^2 + 2M_1M_2 + (M_2)^2}{|\mathcal{C}|} = \frac{|\mathcal{C}|^2}{|\mathcal{C}|} = 2^k.$$

Due to the minimum Hamming distances in the distribution vectors, we have $a_i = 0$ for $1 \leq i \leq 2$, $b_i = 0$ for $0 \leq i \leq 3$ and $c_i = 0$ for $1 \leq i \leq 4$. Additionally, since $a_0 = M_1/|\mathcal{C}|$ and $c_0 = M_2/|\mathcal{C}|$, we have that $a_0 + c_0 = 1$.

So far, we have constructed a linear program, i.e., the objective function as well as all the constraints are all *affine* functions. Unfortunately, while the condition $a_0 + c_0 = 1$ is necessary, it is not specific enough to guarantee a solution consistent with our distribution vector definitions. We require an extra condition on $a_i$ and $c_i$, as follows:

$$\begin{aligned} a_0 = &\frac{M_1}{2^k} \implies 2^k(a_0)^2 - \sum_{i=0}^{n} a_i = 0, \\ c_0 = &\frac{M_2}{2^k} \implies 2^k(c_0)^2 - \sum_{i=0}^{n} c_i = 0. \end{aligned}$$

This equality constraint is not affine, and thus our program is no longer a convex optimization. However, given the smoothness of our quadratic constraints, there are many efficient optimization techniques for this nonlinear program (NLP) [60].

**Theorem 3.** *For an $(n, k)$ SEC-(sm)DEC code, we have*

$$M_2 \leq \sqrt{2^k \textstyle\sum_{i=0}^{n} c_i^*},$$

*where* $\mathbf{c}^*$ *is the solution to the following nonlinear program:*

$$\textit{maximize:} \sum_{i=0}^{n} c_i \quad \textit{subject to:}$$

| *Inequality Constraints* | *Equality Constraints* |
|---|---|
| $\mathbf{a} \geq 0,$ | $a_i = 0, \quad 1 \leq i \leq 2,$ |
| $\mathbf{b} \geq 0,$ | $b_i = 0, \quad 0 \leq i \leq 3,$ |
| $\mathbf{c} \geq 0,$ | $c_i = 0, \quad 1 \leq i \leq 4,$ |
| $\mathbf{aQ} \geq 0,$ | $a_0 + c_0 = 1,$ |
| $\mathbf{cQ} \geq 0,$ | $\sum_{i=0}^{n}(a_i + b_i + c_i) = 2^k,$ |
| $(\mathbf{a} + \mathbf{b} + \mathbf{c})\mathbf{Q} \geq 0,$ | $2^k(a_0)^2 - \sum_{i=0}^{n} a_i = 0,$ |
| | $2^k(c_0)^2 - \sum_{i=0}^{n} c_i = 0.$ |

The NLP bound correctly returns *infeasible solution* for any parameters $(n, k)$ with less redundancy than the associated Hamming code. Additionally, for any $(n, k)$ with more redundancy than the associated $t = 2$ BCH code, the program correctly returns $M_2 = 2^k$.

Only the first three equality constraints need to be changed to arrive at the (sm)SEC and SED-(sm)SED programs. The individual equality constraints for the (sm)SEC program are

$$b_i = 0, \quad 0 \leq i \leq 1,$$

$$c_i = 0, \quad 1 \leq i \leq 2.$$

Note that $a_i$ is not forced to be 0 for any value. Similarly, the individual equality constraints for the SED-(sm)SEC program are

$$a_i = 0, \quad i = 1,$$

$$b_i = 0, \quad 0 \leq i \leq 2,$$

$$c_i = 0, \;\; 1 \leq i \leq 2.$$

Our NLP bound does not improve on the sphere-packing bound when we use the minimum number of redundancy bits required for our UMP constructions. However, we do obtain tighter bounds with the usage of additional redundancy bits.

For example, with $k = 16$ message bits, the optimal SECDED code has parameters $(22, 16, 4)$, and the optimal DEC code has parameters $(26, 16, 5)$. Codes with lengths in between these are largely unexplored since the minimum distance of the code cannot increase from 4 to 5. However, since we are interested in more than just the overall minimum distance of the code, it is useful to obtain bounds on the number of special messages for these code parameters as well. Table 3.2 provides the NLP results for the SEC-(sm)DEC codes with parameters in between SECDED and DEC for $k = 8$ and $k = 16$.

## 3.6 Special Mapping Strategies and Results in Random-Access Memories

Now that we have established the code constructions and bounds, we switch our focus toward their practical usage in real-life systems. Data or instructions stored in memory are generally structured. Data in memory is usually low-magnitude signed or unsigned data of a certain data type. These low magnitude values get inefficiently represented by fixed size data type, for e.g., a 4-byte integer type used to represent values that usually need only 1-byte. This means in most cases the MSBs would be a leading pad of 0's or 1's. Also, frequencies of instructions in most applications follow a power-law distribution [10]; some instructions are much more frequently accessed than the other instructions. If the *opcode*, which primarily determines the action taken by the instruction for a certain instruction set architecture (ISA), is for example, the first x bits, then the relative frequency of the opcodes of the common instructions are high. Thus, most instructions in the memory would have the same prefix of

Table 3.3: Fraction of Special Messages per Benchmark within Suite

| | 32-bit Architecture | | | | 64-bit Architecture | | | |
|---|---|---|---|---|---|---|---|---|
| | Most Freq. 2 opcode (Instruction Memory) | | First 6 bits are 0 (Data Memory) | | Most Freq. opcodes (Instruction Memory) | | First 7 bits are 0 (Data Memory) | |
| **Suite** | **Max** | **Mean** | **Max** | **Mean** | **Max** | **Mean** | **Max** | **Mean** |
| AxBench | 0.51 | 0.46 | 0.92 | 0.86 | 0.27 | 0.26 | 0.89 | 0.82 |
| SPEC CPU2006 | 0.56 | 0.37 | 0.99 | 0.89 | 0.31 | 0.22 | 0.99 | 0.60 |

x-bits.

We collected dynamic memory access traces of various benchmarks that were compiled for both the 64-bit and 32-bit RISC-V instruction sets v2.0 and analyzed them to determine the most frequent opcodes (in instruction memory) and the relative frequency of common patterns (in data memory) over the entire suite; the results are shown in Table 3.3. We find that the distribution of opcodes is highly asymmetric—the two most frequent instructions, `LOAD` and `OP-IMM` [61], comprise an average of 51% and 56% of the instructions in the AxBench [39] and SPEC CPU2006 suites, respectively. For data memory the majority of stored vectors begin with a run of 0's consistently throughout each benchmark (as demonstrated by the low variance values).

Due to the popularity of $(39, 32)$ SECDED codes in byte-oriented architecures, we seek a $(39, 32)$ SEC-(sm)DEC coding framework that efficiently maps special messages of our choice to special codewords. For a $(39, 32)$ SEC-(sm)DEC code formed via Construction 2, Lemma 3 yields $\log(M_2) = 26$. There are two natural choices for our special messages. First, since there are 25 non-opcode bits in the message, we are able to offer DEC protection to 2 opcodes and all of their associated messages. Alternatively, we can offer full DEC protection to any message beginning with a run of 0's of length at least $32 - 26 = 6$ bits. Similarly, for the $(72, 64)$ SEC-(sm)DEC code, Lemma 3 yields $\log(M_2) = 57$. Thus, we offer DEC protection to the single most likely opcode (and associates messages), or alternatively, to any message whose first 7 bits are 0's.

Using the same number of redundancy bits as the SECDED code, our SEC-(sm)DEC coding scheme offers full DEC protection to special messages based on a customizable mapping

scheme. Depending on the user goal, our construction could lead to system-level benefits such as less frequent checkpoints in supercomputers and decreased risk of catastrophic failure from erroneous special messages. Our results indicate that the $(39, 32)$ SEC-(sm)DEC scheme can improve the overall failure rate (in systems were DUEs are critical) by up to 9x with no additional redundancy using the leading run of 0's mapping technique.

Implementing SEC-(sm)DEC coding would requires changes to the hardware that already supports SECDED. The encoding latency and energy when writing to the memory are almost identical for the two protection schemes. The decoding requires an additional clock cycle for the non-special messages in the case of SEC-(sm)DEC. This is because for SEC-(sm)DEC, the first 6-bits of a 32-bit message is non-systematic. For special messages, this first 6-bits is the special prefix that is known and hence, the trailing systematic 26-bits can simply be truncated from the received message when there is no error and the special prefix can be added to construct the original message. However, for non-special messages the first 6-bits is not known and hence, the entire received codeword needs to be go through an additional cycle of matrix multiplication to retrieve the original message. An additional cycle latency while reading from the main memory would have minimal impact on the performance of the system.

## 3.7 Concluding Remarks

In this chapter, we studied a practical class of UMP codes, introduced bounds on code cardinality, created explicit constructions based on subcodes, and provided motivating applications based on real data. Our UMP codes provide advantageous direct alternatives to very popular codes such as the single-bit parity-check code and the SECDED code.

**Acknowledgment**

The majority of this chapter was submitted for publication in [62] as a collaboration with Prof. Lara Dolecek, Prof. Puneet Gupta, Frederic Sala, Mark Gottscho, and Irina Alam;

preliminary results were presented at [63].

# CHAPTER 4

# Coding for Burst Deletions in DNA Storage

## 4.1 Introduction

The explosion in the amount of data being generated and stored has led to novel challenges in data storage technologies. One key aspect is archival storage, that is, storage devices that are required to hold on to data for a very long amount of time, but are written to relatively rarely. Such devices require a large capacity and excellent durability. These features are difficult to find in many modern memories, leading to proposals and experimentation with molecular storage.

In fact, DNA offers a uniquely well-suited medium for archival storage [64, 65, 66]. DNA itself is highly resilient, as shown by the fact that recently, a 700,000 year-old horse genome was successfuly sequenced (read) [67]. DNA is also incredibly dense, with recent experimental media reaching a density of 2 PB/gram [68]. The downside of DNA-based storage is that writing requires performing the slow and expensive synthesis process, and even reading requires sequencing. However, these issues are not critical in the archival storage domain.

With DNA storage, the information is stored in strands of nucleotides. The four nucleotides are cytosine, guanine, adenine, and thymine (CGAT); thus information is stored in 4-ary sequences. Writing is performed through DNA synthesis, that is, the process of constructing the desired strands. There are a variety of synthesis methods; one such example is chemical oligonucleotide synthesis, first developed in the 1950's.

The reading process is known as sequencing. Long strands of nucleotides cannot be read in a single shot. Instead, sequencing devices read short fragments of contiguous nucleotides. A large number of fragments is collected; afterwards, the fragments must be stitched together to read the original sequence. This process has motivated a large amount of recent research into assembly and reconstruction algorithms, seeking to answer, for example, the minimum number of fragments that must be read, the smallest amount of overlap, and the most efficient algorithm required for successful assembly [69, 70, 71, 72].

There have been multiple generations of sequencing technologies. These technologies typically trade-off accuracy for efficiency. For example, modern, third-generation sequencers offer longer reads, but also induce more exotic errors. In addition, sequencers are not the only source of error: processes affecting the molecules themselves (e.g., mutations) also produce errors. Naturally, these errors must be dealt with through error-correcting codes. Although early work applied simplistic codes, an efficient data storage system necessitates sophisticated codes tailored to the error profiles induced by specific synthesis and sequencing technologies [73, 74, 75, 76].

In this chapter, we create codes to correct burst deletions, a type of error that arises in nanopore sequencing technologies [77]. Additionally, even in traditional communication systems, symbols are often inserted or deleted due to synchronization errors. These errors can be caused by a variety of disturbances such as timing defects or packet-loss. Constructing codes that correct insertions or deletions is a notoriously challenging problem since a relatively small number of edits can cause the transmitted and received sequences to be vastly different in terms of the Hamming metric.

For disconnected, intermittent, and low-bandwidth environments, the problem of recovering from symbol insertion/deletion errors becomes exacerbated [78]. From the perspective of the communication systems, these errors manifest themselves in bursts where the errors tend to cluster together. Our goal in this chapter is the study of codes capable of correcting bursts of insertion/deletion errors. Such codes have many applications pertaining to the

synchronization of data in wireless sensor networks and satellite communication devices [79].

In the 1960s, Varshamov, Tenengolts, and Levenshtein laid the foundations for codes capable of correcting insertions and deletions. In 1965, Varshamov and Tenengolts created a class of codes (now known as VT-codes) that is capable of correcting asymmetric errors on the Z-channel [80, 81]. Shortly thereafter, Levenshtein proved that these codes can also be used to correct a single insertion or deletion [82] and he also constructed a class of codes that can correct two adjacent insertions or deletions [83].

The main goal of this chapter is to study codes that correct a *burst of deletions* which refers to the deletion of a fixed number of consecutive bits (or symbols in the non-binary regime). A code will be called a *b-burst-deletion-correcting code* if it can correct any deletion burst of size $b$. For example, the codes studied by Levenshtein in [83] are two-burst-deletion-correcting codes.

Establishing tight upper bounds on the cardinality of burst-deletion-correcting codes is a challenging task since the burst deletion balls are not all of the same size. In [82], Levenshtein derived an asymptotic upper bound on the maximal cardinality of a $b$-burst-deletion-correcting code, given by $\frac{2^{n-b+1}}{n}$. Therefore, the minimum redundancy of such a code should be approximately $\log(n) + b - 1$. Using the method developed recently by Kulkarni and Kiyavash in [13] for deriving an upper bound on deletion-correcting codes, we establish a non-asymptotic upper bound on the cardinality of $b$-burst-deletion-correcting codes which matches the asymptotic upper bound by Levenshtein.

On the other hand, the best construction of $b$-burst-deletion-correcting codes, that we are aware of, is Construction 1 by Cheng *et al.* [84]. The redundancy of this construction is $b(\log(n/b + 1))$ and therefore there is still a significant gap between the lower bound on the redundancy and the redundancy of this construction. One of our main results in this chapter is showing how to improve the construction from [84] and deriving codes whose redundancy is at most

$$\log(n) + (b-1)\log(\log(n)) + b - \log(b), \tag{4.1}$$

which is larger than the lower bound on the redundancy by roughly $(b-1)\log(\log(n))$.

This chapter is organized as follows. In Section 4.2, we define the common terms used throughout the chapter and we detail the previous results that will be used as a comparison. In particular, throughout this chapter, we analyze three models

1. a deletion burst of exactly $b$ consecutive bits,
2. a deletion burst of at most $b$ consecutive bits,
3. a non-consecutive deletion burst of size at most $b$ (i.e, up to $b$ deletions occur within a block of $b$ bits).

We also extend these definitions to insertions. In Section 4.2.3, we prove the equivalence between correcting insertions and deletions in each of the three burst models studied in the chapter. We dedicate Section 4.3 to deriving an explicit upper bound on the code cardinality of $b$-burst-deletion-correcting codes using techniques developed by Kulkarni and Kiyavash [13]. Note that in the asymptotic regime, our bound yields the bound established by Levenshtein [82]. In Section 4.4, we construct $b$-burst-deletion-correcting codes with the redundancy stated in Equation 4.1. Then, in Section 4.5, we extend the $b$-burst-deletion-correcting code to the non-binary regime so as to be useful in coding for DNA storage. In Sections 4.6 and 4.7, we present code constructions that correct a deletion burst of size at most $b$ and codes that correct a non-consecutive burst of size at most three and four, respectively. Lastly, Section 4.8 concludes the chapter.

## 4.2 Preliminaries

### 4.2.1 Notation and Definitions

Let $\mathbb{F}_2^n$ denote the set of all binary vectors (sequences) of length $n$. A *subsequence* of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is formed by taking a subset of the symbols of $\mathbf{x}$ and aligning them without changing their order. Hence, for $m < n$, any vector $\mathbf{y} = (x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ is a

subsequence of $\mathbf{x}$ if $1 \leq i_1 < i_2 < \cdots < i_m \leq n$, and in this case we say that $n-m$ *deletions* occurred in the vector $\mathbf{x}$ and $\mathbf{y}$ is the result.

A *run* of length $r$ of a sequence $\mathbf{x}$ is a subvector of $\mathbf{x}$ such that $x_i = x_{i+1} = \cdots = x_{i+r-1}$, in which $x_{i-1} \neq x_i$ if $i > 1$, and if $i + r - 1 < n$, then $x_{i+r-1} \neq x_{i+r}$. We denote by $\rho(\mathbf{x})$ the number of runs of a sequence $\mathbf{x} \in \mathbb{F}_2^n$.

We refer to a *deletion burst of size* $b$ when exactly $b$ consecutive deletions have occurred, i.e., from $\mathbf{x}$, we obtain a subsequence $(x_1, \ldots, x_i, x_{i+b+1}, \ldots, x_n) \in \mathbb{F}_2^{n-b}$. Similarly, a *deletion burst of size at most* $b$ results in a subsequence $(x_1, \ldots, x_i, x_{i+a+1}, \ldots, x_n) \in \mathbb{F}_2^{n-a}$, for some $a \leq b$. More generally, a *non-consecutive deletion burst of size at most* $b$ is the event where within $b$ consecutive symbols of $\mathbf{x}$, there were some $a \leq b$ deletions, i.e., we obtain a subsequence $(x_1, \ldots, x_i, x_{i+i_1}, x_{i+i_2}, \ldots, x_{i+i_{b-a}}, x_{i+b+1}, \ldots, x_n) \in \mathbb{F}_2^{n-a}$, for some $a \leq b$, where $1 \leq i_1 < i_2 < \cdots < i_{b-a} \leq b$.

The *b-burst-deletion ball* of a vector $\mathbf{x} \in \mathbb{F}_2^n$, is denoted by $D_b(\mathbf{x})$, and is defined to be the set of subsequences of $\mathbf{x}$ of length $n-b$ obtained by the deletion of a burst of size $b$. Similarly, $D_{\leq b}(\mathbf{x})$ is defined to be the set of subsequences of $\mathbf{x}$ obtained from a deletion burst of size at most $b$.

A *b-burst-deletion-correcting code* $\mathcal{C}$ is a set of codewords in $\mathbb{F}_2^n$ such that there are no two codewords in $\mathcal{C}$ where deletion bursts of size $b$ result in the same word of length $n-b$. That is, for every $\mathbf{x}, \mathbf{y} \in \mathcal{C}$, $D_b(\mathbf{x}) \cap D_b(\mathbf{y}) = \emptyset$.

We will use the following terms for bursts of insertions, namely: *insertions burst of size (at most) b*, *b-burst-insertion ball*, and *b-burst-insertion-correcting code*.

Throughout this chapter, we let $b$ be a fixed integer which divides $n$. Similar to [84], for

a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, we define the following $b \times \frac{n}{b}$ array:

$$A_b(\mathbf{x}) = \begin{bmatrix} x_1 & x_{b+1} & \ldots & x_{n-b+1} \\ x_2 & x_{b+2} & \ldots & x_{n-b+2} \\ \vdots & \vdots & \ddots & \vdots \\ x_b & x_{2b} & \ldots & x_n \end{bmatrix},$$

and for $1 \leq i \leq b$ we denote by $A_b(\mathbf{x})_i$ the $i$th row of the array $A_b(\mathbf{x})$.

For two vectors $\mathbf{x}$, $\mathbf{y} \in \mathbb{F}_2^n$, the *Levenshtein distance* $d_L(\mathbf{x}, \mathbf{y})$ is the minimum number of insertions and/or deletions that is necessary to change $\mathbf{x}$ into $\mathbf{y}$. Unless stated otherwise, all logarithms in this chapter are taken according to base 2.

We provide the following example to demonstrate the previous definitions.

**Example 4.** *Let* $\mathbf{x} = (010100001011011)$. *There are 10 runs in* $\mathbf{x}$, *with the longest run having length 4. The following set is the b-burst-deletion ball of* $\mathbf{x}$ *for* $b = 3$*:*

$$D_3(\mathbf{x}) = \{(100001011011), (000001011011), (010001011011), \\ (010101011011), (010100011011), (010100001011)\}.$$

*The array* $A_b(\mathbf{x})$, *for* $b = 3$, *is formed as follows:*

$$A_3(\mathbf{x}) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

*As described previously, we denote the ith row of* $A_3(\mathbf{x})$ *as* $A_3(\mathbf{x})_i$. *For instance,* $A_3(\mathbf{x})_2 = (10011)$. *Let* $\mathbf{y} = (010101011011)$ *be our received vector after* $\mathbf{x}$ *is passed through a b-burst-*

*deletion channel of* $b = 3$. *We form the array* $A_b(\mathbf{y})$, *for* $b = 3$, *as follows:*

$$A_3(\mathbf{y}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}.$$

### 4.2.2 Related Work

In this subsection, we recall known results on codes which correct deletions and insertions. These results will be used later as a comparison reference for our constructions.

#### Single-deletion-correcting codes

The Varshamov-Tenengolts (VT) codes [81] are a family of single-deletion-correcting codes (see also Sloane's survey in [85]) and are defined as follows.

**Definition 6.** *For* $0 \leq a \leq n$, *the Varshamov-Tenengolts (VT) code* $VT_a(n)$ *is defined to be the following set of binary vectors:*

$$VT_a(n) \triangleq \left\{ \mathbf{x} = (x_1, \ldots, x_n) : \sum_{i=1}^{n} i x_i \equiv a \pmod{(n+1)} \right\}.$$

Levenshtein proved in [82] that VT-codes can correct either a single deletion or insertion. It is also known that the largest VT-codes are obtained for $a = 0$, and these codes are conjectured to have the largest cardinality among all single-deletion-correcting codes [85]. The redundancy of the $VT_0(n)$ code is at most $\log(n+1)$ (for the exact cardinality of the code $VT_0(n)$, see [85, Eq. (10)]). For all $n$, the union of all VT-codes forms a partition of the space $\mathbb{F}_2^n$, that is $\cup_{a=0}^{n} VT_a(n) = \mathbb{F}_2^n$.

### $b$-burst-deletion-correcting codes

We next review the existing constructions of $b$-burst-deletion-correcting codes, as given in [84].

- Construction 1 from [84, Section III]: the constructed code is defined to be the set of all codewords $\mathbf{c}$ such that each row of the $b \times \frac{b}{n}$ array $A_b(\mathbf{c})$ is a codeword of the code $VT_0(\frac{n}{b})$. A deletion burst of size $b$ deletes exactly one symbol in each row of $A_b(\mathbf{c})$ which can then be corrected by the VT-code. The redundancy of this construction is

$$b\left(\log\left(\frac{n}{b}+1\right)\right).$$

- Construction 2 from [84, Section III]: for every codeword $\mathbf{c}$ in this construction, the first row of the $b \times \frac{b}{n}$ array $A_b(\mathbf{c})$ is $(1, 0, 1, 0, \dots)$ (to obtain the position of the deletion of each row to within one symbol). All the other rows are codewords from a code that can correct one deleted bit if it is known to be in one of two adjacent positions. The redundancy of this construction is

$$\frac{n}{b} + (b-1)\log(3).$$

- Construction 3 from [84, Section III]: for every codeword $\mathbf{c}$, the first two rows of the $b \times \frac{b}{n}$ array $A_b(\mathbf{c})$ are VT-codes together with the property that the run length is at most two. The other rows are again codewords that can correct the deleted bit if it is known to occur in one of two adjacent positions. The redundancy of this construction is approximately:

$$\begin{aligned} &2\frac{n}{b} + (b-2)\log(3) - \log\left(\frac{4\cdot 3^{\frac{n}{b}-1}}{(\frac{n}{b}+1)^2}\right) \\ =&\frac{n}{b} + 2\log\left(\frac{n}{b}+1\right) + (b-2)\log(3) + c, \end{aligned}$$

for some constant $c$.

### Correcting a deletion burst of size at most $b$

To the best of our knowledge, the only known construction to correct a burst of size *at most* $b$ is the one from [86]. Here, encoding is done in an array of size $\frac{n}{b} \times b$ and the stored vector is taken row-wise from the array. The first $\frac{n}{b} - 1$ rows are codewords of a comma-free code (CFC) and the last row is used for the redundancy of an erasure-correcting code (applied column-wise). Using the size of a CFC from [86, p. 9], it is possible to derive that the redundancy of this construction is at least $\frac{n}{b}$ and therefore the code rate is less than one.

### Correcting $b$ deletions (not a burst)

The Helberg code [87, 88] was the first explicit code construction to correct $b$ deletions at arbitrary positions (not necessarily in a burst) in a vector of length $n$. The correctness of the Helberg code was later proven in [89]. Since then, there have been various constructions with improved rate; see, e.g. [90, 91]. Recently, Brakensiek et al. [92] presented a code correcting $b$ arbitrary deletions whose redundancy is given by

$$c \cdot b^2 \log(b) \log(n),$$

for some constant $c$.

### Correcting a burst of at most $b$ non-consecutive deletions

Whenever we refer to a burst of at most $b$ non-consecutive deletions, we refer to the case where $a \leq b$ deletions occur within $b$ consecutive positions, but not necessarily on $a$ consecutive positions. To our knowledge, there is no previous work on this model.

### 4.2.3 Equivalence of Bursts of Deletions and Bursts of Insertions

In the following, we show the equivalence of bursts of deletions and bursts of insertions. Thus, in the remainder of the chapter, whenever we refer to bursts of deletions, all the results hold equivalently for bursts of insertions as well.

**Theorem 4.** *A code $\mathcal{C}$ is a $b$-burst-deletion-correcting code if and only if it is a $b$-burst-insertion-correcting code.*

*Proof.* Note that if $\mathcal{C}$ is a $b$-burst-deletion-correcting code of length $n$, then there is no vector in $\mathbb{F}_2^{n-b}$ that stems from deleting $b$ consecutive symbols in two distinct codewords.

Now, assume that $\mathcal{C}$ is *not* a $b$-burst-insertion-correcting code. Then, there are two different codewords $\mathbf{x}$, $\mathbf{y} \in \mathcal{C}$ of length $n$ such that inserting a $b$-burst in both codewords leads to two equal vectors of length $n+b$. That is, there are two integers $i, j$ (w.l.o.g. $i \leq j$) and two vectors $(s_1, \ldots, s_b)$, $(t_1, \ldots, t_b)$ such that for $\mathbf{v} \triangleq (x_1, \ldots, x_i, s_1, \ldots, s_b, x_{i+1}, \ldots, x_n)$ and
$\mathbf{w} \triangleq (y_1, \ldots, y_j, t_1, \ldots, t_b, y_{j+1}, \ldots, y_n)$, it holds that $\mathbf{v} = \mathbf{w}$.

Define a set $\mathcal{J} = \{i+1, \ldots, i+b, j+1, \ldots, j+b\}$. If $|\mathcal{J}| = 2b$, then let $\mathcal{I} \triangleq \mathcal{J}$, else $\mathcal{I} = \mathcal{J} \cup \{j+b+1, \ldots, j+3b-|\mathcal{J}|\}$ such that in either case $|\mathcal{I}| = 2b$.

Denote by $\mathbf{v}_{\mathcal{I}}$ and $\mathbf{w}_{\mathcal{I}}$ the two vectors of length $n-b$ that stem from deleting the symbols at the positions in $\mathcal{I}$ in $\mathbf{v}$ and $\mathbf{w}$. Clearly, $\mathbf{v}_{\mathcal{I}} = \mathbf{w}_{\mathcal{I}}$. Further, $\mathbf{v}_{\mathcal{I}} = (x_1, \ldots, x_\ell, x_{\ell+b+1}, \ldots, x_n)$, where $\ell = i$ if $j \leq i+b$ and $\ell = j-b$ else, and $\mathbf{w}_{\mathcal{I}} = (y_1, \ldots, y_i, y_{i+b+1}, \ldots, y_n)$. However, this is a contradiction since $\mathbf{x}$ and $\mathbf{y}$ are codewords of a $b$-burst-deletion-correcting code and thus, the code $\mathcal{C}$ is also a $b$-burst-insertion-correcting code.

The other direction can easily be shown with the same strategy. □

The proofs of the next two theorems are similar to the one of Theorem 4 and thus we omit them.

**Theorem 5.** *A code $\mathcal{C}$ can correct a deletion burst of size at most $b$ if and only if it can correct an insertion burst of size at most $b$.*

**Theorem 6.** *A code $\mathcal{C}$ can correct a non-consecutive deletion burst of size at most $b$ if and only if it can correct a non-consecutive insertion burst of size at most $b$.*

## 4.3 An Upper Bound on the Code Size

The goal of this section is to provide an explicit upper bound on the cardinality of burst-deletion-correcting codes. For large $n$, Levenshtein [83] derived an asymptotic upper bound on the maximal cardinality of a binary $b$-burst-deletion-correcting code $\mathcal{C}$ of length $n$. This bound states that for $n$ large enough, an upper bound on the cardinality of the code $\mathcal{C}$ is approximately

$$\frac{2^{n-b+1}}{n},$$

and hence its redundancy is at least roughly $\log(n) + b - 1$.

Our main goal in this section is to provide an explicit upper bound on the cardinality of $b$-burst-deletion-correcting codes. We follow a method which was recently developed by Kulkarni and Kiyavash in [13] to obtain such an upper bound.

The size of the $b$-burst-deletion ball for a vector $\mathbf{x}$ was shown by Levenshtein [83] to be

$$|D_b(\mathbf{x})| = 1 + \sum_{i=1}^{b} \Big( \rho(A_b(\mathbf{x})_i) - 1 \Big), \tag{4.2}$$

where $\rho(A_b(\mathbf{x})_i)$ denotes the number of runs in the $i$-th row of the array $A_b(\mathbf{x})$. Notice that $1 \leq |D_b(\mathbf{x})| \leq 1 + (\frac{n}{b} - 1) \cdot b = n - b + 1$.

**Lemma 3.** *Let $\mathbf{x} \in \mathbb{F}_2^n$ and $\mathbf{y} \in \mathbb{F}_2^{n+b}$ be two vectors such that $\mathbf{x} \in D_b(\mathbf{y})$. Then, $|D_b(\mathbf{y})| \geq |D_b(\mathbf{x})|$.*

*Proof.* If $\mathbf{x} \in D_b(\mathbf{y})$ then for all $1 \leq i \leq b$, $A_b(\mathbf{x})_i \in D_1(A_b(\mathbf{y})_i)$, and hence $\rho(A_b(\mathbf{x})_i) \leq \rho(A_b(\mathbf{y})_i)$, [13, Lemma 3.2]. Therefore, according to (4.2), we get that

$$|D_b(\mathbf{x})| = 1 + \sum_{i=1}^{b} \Big( \rho(A_b(\mathbf{x})_i) - 1 \Big) \leq 1 + \sum_{i=1}^{b} \Big( \rho(A_b(\mathbf{y})_i) - 1 \Big) = |D_b(\mathbf{y})|.$$

□

We are now ready to provide an explicit upper bound on the cardinality of burst-deletion-correcting codes.

**Theorem 7.** *Any b-burst-deletion-correcting code $\mathcal{C}$ of length n satisfies*

$$|\mathcal{C}| \leq \frac{2^{n-b+1} - 2^b}{n - 2b + 1}.$$

*Proof.* We proceed similarly to the method presented by Kulkarni and Kiyavash in [13, Theorem 3.1]. Let $\mathcal{H}_{2,b,n}$ be the following hypergraph:

$$\mathcal{H}_{2,b,n} = (\mathbb{F}_2^{n-b}, \{D_b(\mathbf{x}) : \mathbf{x} \in \mathbb{F}_2^n\}).$$

The size of the largest $b$-burst-deletion-correcting code equals the matching number of $\mathcal{H}_{2,b,n}$, denoted as in [13] by $\nu(\mathcal{H}_{2,b,n})$. By [13, Lemma 2.4], to obtain an upper bound on $\nu(\mathcal{H}_{2,b,n})$, we can construct a fractional transversal, which will give an upper bound on the matching number. The best upper bound according to this method is denoted by $\tau^*(\mathcal{H}_{2,b,n})$ and is calculated according to the following linear programming problem

$$\begin{aligned} \tau^*(\mathcal{H}_{2,b,n}) = \min_{w:\mathbb{F}_2^{n-b}\to\mathbb{R}} \left\{ \sum_{\mathbf{x}\in\mathbb{F}_2^{n-b}} w(\mathbf{x}) \right\} \\ \text{subject to} \sum_{\mathbf{x}\in D_b(\mathbf{y})} w(\mathbf{x}) \geq 1, \forall \mathbf{y} \in \mathbb{F}_2^n \\ \text{and} \qquad w(\mathbf{x}) \geq 0, \forall \mathbf{x} \in \mathbb{F}_2^{n-b}. \end{aligned}$$

Next, we will show a weight assignment $w$ to the vectors in $\mathbb{F}_2^{n-b}$ which provides a fractional transversal. This weight assignment is given by

$$w(\mathbf{x}) = \frac{1}{|D_b(\mathbf{x})|}, \quad \forall \mathbf{x} \in \mathbb{F}_2^{n-b},$$

which clearly satisfies that $w(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{F}_2^{n-b}$. Furthermore, according to Lemma 3, we also get that for every $\mathbf{y} \in \mathbb{F}_2^n$:

$$\sum_{\mathbf{x} \in D_b(\mathbf{y})} w(\mathbf{x}) = \sum_{\mathbf{x} \in D_b(\mathbf{y})} \frac{1}{|D_b(\mathbf{x})|} \geq \sum_{\mathbf{x} \in D_b(\mathbf{y})} \frac{1}{|D_b(\mathbf{y})|} \geq 1,$$

and hence $w$ indeed provides a fractional transversal.

For $1 \leq i \leq n-b+1$, let us denote by $N(n,b,i)$ the size of the set $\{\mathbf{x} \in \mathbb{F}_2^n : |D_b(\mathbf{x})| = i\}$. We show in Appendix 4.9.1 that $N(n,b,i) = 2^b \binom{n-b}{i-1}$. The weight of this fractional transversal is given by

$$\begin{aligned}
\sum_{\mathbf{x} \in \mathbb{F}_2^{n-b}} w(\mathbf{x}) &= \sum_{\mathbf{x} \in \mathbb{F}_2^{n-b}} \frac{1}{|D_b(\mathbf{x})|} \\
&= \sum_{i=1}^{n-2b+1} \frac{N(n-b,b,i)}{i} \\
&= 2^b \sum_{i=1}^{n-2b+1} \frac{\binom{n-2b}{i-1}}{i} \\
&= 2^b \sum_{i=1}^{n-2b+1} \frac{(n-2b)!}{(i-1)!(n-2b-i+1)!i} \\
&= 2^b \sum_{i=1}^{n-2b+1} \frac{(n-2b+1)!}{i!(n-2b-i+1)!(n-2b+1)} \\
&= \frac{2^b}{n-2b+1} \sum_{i=1}^{n-2b+1} \binom{n-2b+1}{i} \\
&= \frac{2^{n-b+1}-2^b}{n-2b+1}.
\end{aligned}$$

Therefore, the value $\frac{2^{n-b+1}-2^b}{n-2b+1}$ is an upper bound on the maximum cardinality of any binary $b$-burst-deletion-correcting code. □

Notice that for $b = 1$ our upper bound in Theorem 7 coincides with the upper bound in [13, Theorem 3.1] for single-deletion-correcting codes. Furthermore, for $n$ large enough our upper bound matches the asymptotic upper bound from [83].

Lastly, we conclude that the redundancy of a $b$-burst-deletion-correcting code is lower

bounded by the following value

$$\log(n-2b+1) - \log(2^{-b+1} - 2^{b-n}) \approx \log(n) + b - 1. \tag{4.3}$$

## 4.4 Construction of $b$-Burst-Deletion-Correcting Codes

The main goal of this section is to provide a construction of $b$-burst-deletion-correcting codes, whose redundancy is better than the state of the art results we reviewed in Section 4.2.2 and is close to the lower bound on the redundancy, which is stated in (4.3). We will first explain the main ideas of the construction and will then provide the specific details of the construction.

### 4.4.1 Background

As shown in Section 4.2, we will treat the codewords in the $b$-burst-deletion-correcting code as $b \times \frac{n}{b}$ codeword arrays, where $n$ is the codeword length and $b$ divides $n$. Thus, for a codeword $\mathbf{x}$, the codeword array $A_b(\mathbf{x})$ is formed by $b$ rows and $\frac{n}{b}$ columns, and the codeword is transmitted column-by-column. Thus, a deletion burst of size $b$ in $\mathbf{x}$ deletes exactly one bit from each row of the array $A_b(\mathbf{x})$. That is, if a codeword $\mathbf{x}$ is transmitted, then the $b \times (\frac{n}{b} - 1)$ array representation of the received vector $\mathbf{y}$ has the following structure

$$A_b(\mathbf{y}) = \begin{bmatrix} y_1 & y_{b+1} & \cdots & y_{n-2b+1} \\ y_2 & y_{b+2} & \cdots & y_{n-2b+2} \\ \vdots & \vdots & \ddots & \vdots \\ y_b & y_{2b} & \cdots & y_{n-b} \end{bmatrix}.$$

Each row is received by a single deletion of the corresponding row in $A_b(\mathbf{x})$ [84], i.e., $A_b(\mathbf{y})_i \in D_1(A_b(\mathbf{x})_i)$, $\forall\ 1 \leq i \leq b$.

Since the channel deletes a burst of $b$ bits, the deletions can span at most two columns of

the codeword array. Therefore, information about the position of a deletion in a single row provides information about the positions of the deletions in the remaining rows. However, note that deletion-correcting codes are not always able to determine the exact position of the deleted bit. For example, assume the all-zero codeword was transmitted and a single deletion of one of the bits has occurred. Even if the decoder can successfully decode the received vector, it is not possible to know the position of the deleted bit since it could be any of the bits.

In order to take advantage of the correlation between the positions of the deleted bits in different rows and overcome the difficulty that deletion-correcting codes cannot always provide the location of the deleted bits, we construct a single-deletion-correcting code with the following special property. The receiver of this code can correct the single deletion and determine its location within a certain predetermined range of consecutive positions. This code will be used to encode the first row of the codeword array and will provide partial information on the position of the deletions for the remaining $b-1$ rows. In these rows, we use a different code that will take advantage of this positional information.

The following is a high-level outline of the proposed codeword array construction:

- The first row in the array is encoded as a VT-code in which we restrict the longest run of 0's or 1's to be at most $\log(2n)$. The details of this code are described in Section 4.4.2.

- Each of the remaining $(b-1)$ rows in the array is encoded using a modified version of the VT-code, which will be called a *shifted VT* ($SVT$)*-code*. This code is able to correct a single deletion in each row once the position where the deletion occurred is known to within $\log(2n)+1$ consecutive positions. The details of these codes are discussed in Section 4.4.3.

Section 4.4.4 presents the full code construction. Let us explore the different facets of our proposed codeword array construction in more detail.

### 4.4.2 Run-length Limited (RLL) VT-Codes

In general, a decoder for a VT-code can decode a single deletion while determining only the position of the run that contains the deletion, but not the exact position of the deletion itself. For this reason, we seek to limit the length of the longest run in the first row of the codewords array.

A length-$n$ binary vector is said to satisfy the $(d, k)$ *Run Length Limited (RLL)* constraint, denoted by $RLL_n(d, k)$, if between any two consecutive 1's there are at least $d$ 0's and at most $k$ 0's [93]. Since we are concerned with runs of 0's or 1's, we will state our constraints on the longest runs of 0's and 1's. Note that the maximum rate of codes which satisfy the $(d, k)$ RLL-constraint for fixed $d$ and $k$ is less than 1. To achieve codes with asymptotic rate 1, the restriction on the longest run is a function of the length $n$.

**Definition 7.** *A length-$n$ binary vector $\mathbf{x}$ is said to satisfy the **f(n)-RLL(n)** constraint, and is called an **f(n)-RLL(n)** vector, if the length of each run of 0's or 1's in $\mathbf{x}$ is at most $f(n)$.*

A set of $f(n)$-RLL($n$) vectors is called an *$f(n)$-RLL($n$) code*, and the set of all $f(n)$-RLL($n$) vectors is denoted by $S_n(f(n))$. The *capacity* of the $f(n)$-RLL($n$) constraint is

$$C(f(n)) = \limsup_{n\to\infty} \frac{\log(|S_n(f(n))|)}{n},$$

and for the case in which the capacity is 1, we define also the *redundancy* of the $f(n)$-RLL($n$) constraint to be

$$r(f(n)) = n - \log(|S_n(f(n))|).$$

**Lemma 4.** *The redundancy of the $\log(2n)$-RLL($n$) constraint is upper bounded by 1 for all $n$, and it asymptotically approaches $\log(e)/4 \approx 0.36$.*

*Proof.* For simplicity let us assume that $n$ is a power of two. Let $X_n$ be a random variable that denotes the length of the longest run in a length-$n$ binary vector, where the vectors

are chosen uniformly at random. We will be interested in computing a lower bound on the probability

$$P(X_n \leq \log(2n)) = P(X_n \leq 1 + \log(n)),$$

or an upper bound on the probability $P(X_n \geq 2 + \log(n))$. By the union bound it is enough to require that every window of $2 + \log(n)$ bits is not all zeros or all ones and thus we get that

$$P(X_n \geq 2 + \log(n)) \leq n \cdot \frac{2}{2^{2+\log(n)}} = \frac{1}{2},$$

and thus $P(X_n \leq 1 + \log(n)) \geq 1/2$. Therefore the size of the set $S_n(\log(2n))$ is at least $2^n/2$ and its redundancy $r(\log(2n))$ is at most one bit.

In order to find the asymptotic behavior of $r(\log(2n))$, we use two results from [94]. First, we have the following equation:

$$P(X_n \leq x) = P(Y_{n-1} \leq x - 1),$$

where $Y_n$ is a random variable that denotes the length of the longest run of ones in a length-$n$ binary vector which is chosen uniformly at random. Second, we use the following approximation:

$$P(Y_n = x) \approx P(x - \log(n/2) < W \leq x + 1 - \log(n/2)),$$

where $W$ is a continuous random variable whose cumulative distribution function is given by $F_W(x) = e^{-(1/2)^x}$. Then, the following holds:

$$\begin{aligned} P(X_n \leq \log(n) + 1) =& P(Y_{n-1} \leq \log(n)) \\ \approx& P\left(W \leq \log(n) + 1 - \log\left(\frac{n-1}{2}\right)\right) \\ =& P\left(W \leq \log\left(\frac{n}{n-1}\right) + 2\right) \end{aligned}$$

$$=e^{-(1/2)^{\log\left(\frac{n}{n-1}\right)+2}} = e^{-(1/4)\cdot\frac{n-1}{n}} = \left(\frac{1}{e^{1/4}}\right)^{1-\frac{1}{n}}.$$

Therefore, for $n$ large enough $P(X_n \leq \log(n)+1) \approx e^{-1/4}$, and $r(\log(2n)) \approx \log(e)/4 \approx 0.36$. □

**Remark 1.** *Since* $\log(e)/2 < 1$*, we can guarantee that the encoded vector will not have a run of length longer than* $\log(2n)$ *with the use of a single additional redundancy bit. Thus* $\log(2n)$ *is a proper choice for our value of* $f(n)$*; a smaller* $f(n)$ *would substantially increase the redundancy of the first row, and a larger* $f(n)$ *would not help since setting* $f(n) = \log(2n)$ *already only requires at most a single bit of redundancy. Note that Lemma 4 agrees with the results from [95, 94] which state that the typical length of the longest run in* $n$ *flips of a fair coin converges to* $\log(n)$*. Lastly we note that in Appendix 4.9.2, we provide an algorithm to efficiently encode/decode run-length limited sequences for the* $(\log(n)+3)$*-RLL*$(n)$ *constraint.*

Recall that our goal was to have the vector stored in the first row be a codeword in a VT-code so it can correct a single deletion and also limit its longest run. Hence we define a family of codes which satisfy these two requirements by considering the intersection of a VT-code with the set $S_n(f(n))$.

**Definition 8.** *Let* $a, n$ *be two positive integers where* $0 \leq a \leq n$*. The* $VT_{a,f(n)}(n)$ *code is defined to be the intersection of the codes* $VT_a(n)$ *and* $S_n(f(n))$*. That is,*

$$VT_{a,f(n)}(n) = \left\{\mathbf{x} \ : \ \mathbf{x} \in VT_a(n), \mathbf{x} \in S_n(f(n))\right\}.$$

Note that since $VT_{a,f(n)}(n)$ is a subcode of $VT_a(n)$, it is also a single-deletion-correcting code. The following lemma is an immediate result on the cardinality of these codes.

**Lemma 5.** *For all* $n$*, there exists* $0 \leq a \leq n$ *such that*

$$|VT_{a,f(n)}(n)| \geq \frac{|S_n(f(n))|}{n+1}.$$

*Proof.* The VT-codes form a partition of $\mathbb{F}_2^n$ into $n+1$ different codebooks $VT_0(n)$, $VT_1(n)$, $\ldots$ , $VT_n(n)$. Using the pigeonhole principle, we can determine the lower bound of the maximum intersection between these $n+1$ codebooks and $S_n(f(n))$ and get that

$$\max_{0\leq a\leq n}\left\{|S_n(f(n))\cap VT_a(n)|\right\}\geq\frac{|S_n(f(n))|}{n+1}.$$

□

We conclude with the following corollary and example.

**Corollary 4.** *For all $n$, there exists $0\leq a\leq n$ such that the redundancy of the code $VT_{a,\log(2n)}(n)$ is at most $\log(n+1)+1$ bits.*

**Example 5.** *In this example we list the codewords for the $VT_{a,\log(2n)}(n)$ code for $n=8$ and $a=0$. Thus, the following list contains all the codewords of the $VT_0(8)$ code with codewords that have the length of their longest run longer than $\log(2n)=4$ crossed out.*

$$\begin{aligned}VT_{0,4}(8)=\{&\cancel{(00000000)},(01110000),(10101000),(00011000),(11000100),(00100100)\\&(11011100),(00111100),(01000010),(11101010),(01011010),(01100110),\\&(10010110),(00001110),\cancel{(01111110)},\cancel{(10000001)},(11110001),(01101001),\\&(10011001),(10100101),(00010101),(10111101),(11000011),(00100011),\\&(11011011),(00111011),(11100111),(01010111),(10001111),\cancel{(11111111)}\}.\end{aligned}$$

*Since there are 26 codewords in $VT_{0,4}(8)$, we see that Corollary 4 is satisfied for $n=8$:*

$$\log\left(\frac{2^8}{|VT_{0,4}(8)|}\right)=\log\left(\frac{2^8}{26}\right)\approx 3.30<\log(9)+1\approx 4.17.$$

### 4.4.3 Shifted VT-Codes

Let us now focus on the remaining $(b-1)$ rows of our codeword array. Decoding the first row in the received array allows the decoder to determine the locations of the deletions of the remaining rows up to a set of consecutive positions. We define a new class of codes with this positional knowledge of deletions in mind.

**Definition 9.** *A **P-bounded single-deletion-correcting code** is a code in which the decoder can correct a single deletion given knowledge of the location of the deleted bit to within $P$ consecutive positions.*

We create a new code, called a *shifted VT ($SVT$)-code*, which is a variant of the VT-code and is able to take advantage of the positional information as defined in Definition 9.

**Construction 3.** *For $0 \leq c < P$ and $d \in \{0,1\}$, let the shifted Varshamov-Tenengolts code $SVT_{c,d}(n,P)$ be:*

$$SVT_{c,d}(n,P) \triangleq \left\{\mathbf{x} : \sum_{i=1}^{n} ix_i \equiv c \ (\mathrm{mod} P), \sum_{i=1}^{n} x_i \equiv d \ (\mathrm{mod} 2)\right\}.$$

Other modifications of the VT-code have previously been proposed in [96] to improve the upper bounds on the cardinality of deletion-correcting codes. The next lemma proves the correctness of this construction and provides a lower bound on the cardinality of these codes.

**Lemma 6.** *For all $0 \leq c < P$ and $d \in \{0,1\}$, the $SVT_{c,d}(n,P)$-code (as defined in Construction 3) is a $P$-bounded single-deletion-correcting code.*

*Proof.* In order to prove that the $SVT_{c,d}(n,P)$-code is a $P$-bounded single-deletion-correcting code, it is sufficient to show that there are no two codewords $\mathbf{x}, \mathbf{y} \in SVT_{c,d}(n,P)$ that have a common subvector of length $n-1$ where the locations of the deletions are within $P$ positions.

Assume in the contrary that there exist two different codewords $\mathbf{x}, \mathbf{y} \in SVT_{c,d}(n,P)$, where there exist $1 \leq k, \ell \leq n$, where $|\ell - k| < P$, such that $\mathbf{z} = \mathbf{x}_{[n]\setminus\{k\}} = \mathbf{y}_{[n]\setminus\{\ell\}}$, and

assume that $k < \ell$. Since $\mathbf{x}, \mathbf{y} \in SVT_{c,d}(n, P)$, we can summarize these assumptions in the following three properties:

1. $\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i \equiv 0 \pmod 2$.
2. $\sum_{i=1}^{n} i x_i - \sum_{i=1}^{n} i y_i \equiv 0 \pmod P$.
3. $\ell - k < P$.

According to these assumptions and since $\mathbf{x}_{[n]\setminus\{k\}} = \mathbf{y}_{[n]\setminus\{\ell\}}$, it is evident that $k$ is the smallest index for which $x_k \neq y_k$, and $\ell$ is the largest index for which $x_\ell \neq y_\ell$. Additionally, from the first property $\mathbf{x}$ and $\mathbf{y}$ have the same parity and thus $x_k = y_\ell$. Outside of the indices $k$ and $\ell$, $\mathbf{x}$ and $\mathbf{y}$ are identical, while inside they are shifted by one position:

$$\begin{aligned} x_i &= y_i \quad \text{for } i < k \text{ and } i > \ell, \\ x_i &= y_{i-1} \quad \text{for } k < i \leq \ell. \end{aligned}$$

We consider two scenarios: $x_k = y_\ell = 0$ or $x_k = y_\ell = 1$. First assume that $x_k = y_\ell = 0$, and in this case we get that

$$\begin{aligned} &\sum_{i=1}^{n} i x_i - \sum_{i=1}^{n} i y_i = \sum_{i=k}^{\ell} i x_i - \sum_{i=k}^{\ell} i y_i = \sum_{i=k+1}^{\ell} i x_i - \sum_{i=k}^{\ell-1} i y_i \\ &= \sum_{i=k+1}^{\ell} i y_{i-1} - \sum_{i=k}^{\ell-1} i y_i = \sum_{i=k}^{\ell-1} (i+1) y_i - \sum_{i=k}^{\ell-1} i y_i = \sum_{i=k}^{\ell-1} y_i. \end{aligned}$$

The sum $\sum_{i=k}^{\ell-1} y_i$ cannot be equal to zero or else we will get that $\mathbf{x} = \mathbf{y}$, and hence

$$0 < \sum_{i=1}^{n} i x_i - \sum_{i=1}^{n} i y_i = \sum_{i=k}^{\ell-1} y_i \leq \ell - k < P,$$

in contradiction to the second property.

A similar contradiction can be shown for $x_k = y_\ell = 1$. Thus, the three properties cannot all be true, and the $SVT_{c,d}(n, P)$-code is a $P$-bounded single-deletion-correcting code. $\square$

**Lemma 7.** *There exist* $0 \leq c < P$ *and* $d \in \{0,1\}$ *such that the redundancy of the* $SVT_{c,d}(n,P)$ *code as defined in Construction 3 is at most* $\log(P)+1$ *bits.*

*Proof.* Similarly to the partitioning of the VT-codes, the $2P$ codes $SVT_{c,d}(n,P)$, for $0 \leq c < P$ and $d \in \{0,1\}$, form a partition of all length-$n$ binary vectors into $2P$ mutually disjoint sets. Using the pigeonhole principle, there exists a code whose cardinality is at least $\frac{2^n}{2P}$ and thus its redundancy is at most $\log(2P) = \log(P)+1$ bits. □

There are two major differences between the SVT-codes and the usual VT-codes. First, the SVT-codes restrict the overall parity of the codewords. This parity constraint costs an additional redundancy bit, but it allows us to determine whether the deleted bit was a 0 or a 1. Second, in the VT-code, the weights assigned to each element in the vector are $1,2,\ldots,n$; on the other hand, in the SVT-code, these weights can be interpreted as repeatedly cycling through $1,2,\ldots,P-1,0$ (due to the ( mod $P$) operation). Because of these differences, a VT-code requires roughly $\log(n+1)$ redundancy bits while a SVT-code requires approximately only $\log(P)+1$ redundancy bits.

The proof of Lemma 6 motivates also the operation of a decoder to the SVT-code. In order to complete the description of this code we show in Appendix 4.9.3 the full description of this decoder for the SVT-codes.

**Example 6.** *In this example we list the codewords for the* $SVT_{0,0}(7,4)$ *code:*

$$\begin{aligned} SVT_{0,0}(7,4) = \{&(0000000),(1010000),(0010100),(1101100),(0100010),(1110010),\\ &(0110110),(1001110),(1000001),(0111001),(0000101),(1010101),\\ &(1100011),(0011011),(0100111),(1110111)\}. \end{aligned}$$

*Using this example, we now highlight the major difference between the VT-code and the SVT-code: an SVT-code code can have codewords with a Levenshtein distance of 2. For example, both of the codewords* (1000001) *and* (0000101) *can lead to* (000001) *with a single deletion.*

*However, no codewords have overlapping single-deletion balls given that the location of the deletion is constrained to within a block of P bits (in this case 4 bits).*

*Additionally, note that the* $SVT_{0,0}(7,4)$ *code satisfies Lemma 6 with equality:*

$$\log\left(\frac{2^7}{|SVT_{0,0}(7)|}\right) = \log\left(\frac{2^7}{16}\right) = 3,$$
$$\log(P)+1 = \log(4)+1 = 3.$$

### 4.4.4 Code Construction

We are now ready to construct $b$-burst-deletion-correcting codes by combining the ideas from the previous two subsections into a single code.

**Construction 4.** *Let* $\mathcal{C}_1$ *be a* $VT_{a,\log(2n/b)}(n/b)$ *code for some* $0 \le a \le n/b$ *and let* $\mathcal{C}_2$ *be a shifted VT-code* $SVT_{c,d}(n/b, \log(n/b)+2)$ *for* $0 \le c < n/b+2$ *and* $d \in \{0,1\}$*. The code* $\mathcal{C}$ *is constructed as follows*

$$\mathcal{C} \triangleq \{\mathbf{x} : A_b(\mathbf{x})_1 \in \mathcal{C}_1, A_b(\mathbf{x})_i \in \mathcal{C}_2, \text{ for } 2 \le i \le b\}.$$

**Theorem 8.** *The code* $\mathcal{C}$ *from Construction 4 is a b-burst-deletion-correcting code.*

*Proof.* Assume $\mathbf{x} \in \mathcal{C}$ is the transmitted vector and $\mathbf{y} \in D_b(\mathbf{x})$ is the received vector. In the $b \times (n/b-1)$ array $A_b(\mathbf{y})$, every row is therefore received by a single deletion of the corresponding row in $A_b(\mathbf{x})$.

Since the first row of $A_b(\mathbf{x})_1$ belongs to a $VT_{a,\log(2n/b)}(n/b)$ code, the decoder of this code can successfully decode and insert the deleted bit in the first row of $A_b(\mathbf{y})_1$. Furthermore, since every run in $A_b(\mathbf{x})_1$ consists of at most $\log(2n/b)$ bits, the locations of the deleted bits in the remaining rows are known within $\log(n/b)+2$ consecutive positions. Finally, the remaining $b-1$ rows decode their deleted bit since they belong to a shifted VT-code $SVT_{c,d}(n/b, \log(n/b)+2)$ (Lemma 6). □

To conclude this discussion, the following corollary summarizes the result presented in this section.

**Corollary 5.** *For sufficiently large $n$, there exists a $b$-burst-deletion-correcting code whose number of redundancy bits is at most*

$$\log(n) + (b-1)\log(\log(n)) + b - \log(b).$$

**Example 7.** *In this example we analyze the code $\mathcal{C}$ from Construction 4 with parameters $n = 24$ and $b = 3$. Note that we can encode each of the 3 rows of $A_3(\mathbf{x})$ independently. To begin, we choose $a = 0$ and encode $A_3(\mathbf{x})_1$ as the $VT_{a,\log(2n/b)}(n/b) \to VT_{0,4}(8)$ code. Note that this is the code from Example 5, and $|VT_{0,4}(8)| = 26$. Next, by choosing $c = d = 0$, we now can encode $A_3(\mathbf{x})_2$ and $A_3(\mathbf{x})_3$ with the $SVT_{c,d}(n/b, \log(n/b) + 2) \to SVT_{0,0}(8, 5)$ code from Construction 3.*

*We find that $|SVT_{0,0}(8,5)| = 26$ as well, so the total number of codewords for our choices of parameters are $|\mathcal{C}| = 26^3 = 17576$. The number of redundant bits required for this code is:*

$$\log\left(\frac{2^{24}}{|\mathcal{C}|}\right) = \frac{2^{24}}{17576} \approx 9.90.$$

*Note that this redundancy is lower than the upper bound from Corollary 7:*

$$\log(24) + 2\log(\log(24)) + 3 - \log(3) \approx 10.39.$$

## 4.5 Non-binary Extension for DNA Coding

We now transform the code from the previous section to the non-binary regime, thus adapting its use toward DNA storage. In Section 4.5.2, we create the $q$-ary shifted Varshamov-Tenengolts (SVT) code, and present the code construction, a proof of correction capability,

and a bound on redundancy. Then, in Section 4.5.3, we introduce the run-length limited property for a subsection of the code, and we present the construction and redundancy for the overall burst-deletion correcting code. Since we now deal with non-binary codes, let $\mathbb{F}_q$ be a finite field of order $q$, where $q$ is a power of a prime and let $\mathbb{F}_q^n$ denote the set of all vectors (sequences) of length $n$ over $\mathbb{F}_q$.

### 4.5.1 Non-Binary VT-Code

Tenengolts generalized the VT-code for non-binary alphabets as follows:

**Definition 10.** *([80]). For $0 \leq c < n$, and $0 \leq d < q$, the non-binary VT-code $VT_{c,d}(n,q)$ is defined to be the following set of $q$-ary vectors:*

$$VT_{c,d}(n,q) \triangleq \Bigg\{ \mathbf{x} = (x_1, \ldots, x_n) : \sum_{i=1}^{n} (i-1)\alpha_i \equiv c \pmod{n}, \sum_{i=1}^{n} x_i \equiv d \pmod{q} \Bigg\},$$

*where $\alpha_1 = 1$ and for $1 < i \leq n$,*

$$\alpha_i = \begin{cases} 1 & \text{if } x_i \geq x_{i-1}, \\ 0 & \text{if } x_i < x_{i-1}. \end{cases}$$

Tenengolts also presented the systematic form of the code as well as a decoding algorithm [80]. The basic idea behind the code is that the (binary) $\alpha_i$ sequence measures the increasing/decreasing behavior of the non-binary codeword, so that a deletion in the overall codeword also results in a deletion in the $\alpha_i$ sequence. This sequence is protected by a binary VT-code, so that a corrected deletion determines the run of increasing or decreasing symbols for the non-binary symbol that was deleted. The second constraint determines what symbol was deleted, and this symbol can then be immediately placed into its correct location in the increasing or decreasing run.

In the following two subsections, we create the non-binary version of the coding framework from Section 4.4, making it suitable for the DNA setting.

### 4.5.2 Non-Binary Shifted VT-Code

In general, we consider $1 \leq P \leq n$. The following code, the $q$-ary shifted VT (SVT)-code, uses the positional knowledge of the deletion as defined in Definition 9 (with the caveat that we are dealing with symbols, not bits).

**Construction 5.** *For $0 \leq c \leq P$, $0 \leq d < q$, and $e \in \{0,1\}$ let the $q$-ary shifted Varshamov-Tenengolts code $SVT_{c,d,e}(n,P,q)$ be:*

$$SVT_{c,d,e}(n,P,q) \triangleq \Bigg\{ \mathbf{x} = (x_1,\ldots,x_n) : \sum_{i=1}^{n} i\alpha_i \equiv c \pmod{(P+1)}, \sum_{i=1}^{n} x_i \equiv d \pmod{q}, \sum_{i=1}^{n} \alpha_i \equiv e \pmod{2} \Bigg\}.$$

*where $\alpha_1 = 1$ and for $1 < i \leq n$,*

$$\alpha_i = \begin{cases} 1 & \text{if } x_i \geq x_{i-1}, \\ 0 & \text{if } x_i < x_{i-1}. \end{cases}$$

Our $q$-ary SVT-code is a modified version of the $q$-ary VT-code. The first major change is that the congruency of the initial condition is (mod $(P+1)$) instead of (mod $n$). This alteration induces a large redundancy savings; however, the addition of a new condition, the third congruency, is necessary for the code to obtain the P-bounded single-deletion correcting property. Additionally, we changed the $(i+1)\alpha_i$ term in the first condition from the non-binary VT-code to simply $i\alpha_i$ for simplicity of notation.

**Lemma 8.** *For all $0 \leq c \leq P$, $0 \leq d < q$, and $e \in \{0,1\}$ the $SVT_{c,d,e}(n,P,q)$-code (as*

*defined in Construction 5) is a P-bounded single-deletion-correcting code.*

*Proof.* Let $\mathbf{x}$ be the transmitted $q$-ary vector of length $n$ and $\mathbf{x}'$ be the vector after a single symbol deletion. Similarly, let $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}'$ be the associated binary difference vectors, as defined in Definition 10, of $\mathbf{x}$ and $\mathbf{x}'$, respectively. It is important to note that $\boldsymbol{\alpha}'$ is in the single-deletion ball of $\boldsymbol{\alpha}$; however, the index of the deleted symbol in $\mathbf{x}$ does not necessarily match the index of the deleted bit in $\boldsymbol{\alpha}$. Let us denote $k$ as the index for the deleted symbol in $\mathbf{x}$, and $\ell$ as the index for the deleted bit in $\boldsymbol{\alpha}$. For example, let $\mathbf{x} = (0, 1, 2, 3, 2, 1)$, so that $\boldsymbol{\alpha} = (1, 1, 1, 1, 0, 0)$. If the 4th symbol in $\mathbf{x}$ is deleted, we are left with $\mathbf{x}' = (0, 1, 2, 2, 1)$ and $\boldsymbol{\alpha}' = (1, 1, 1, 1, 0)$. Here we have $k = 4$ and $\ell = 5$. Additionally, note that the index of a deletion is only unique to within the positions of the original run. In the previous example, it would have been correct to alternatively claim $\ell = 6$.

We briefly recap the proof of correctness of the $q$-ary VT-code from [80]. By analyzing the weight of $\boldsymbol{\alpha}'$ and least nonnegative residue of the first congruence, $\sum_{i=1}^{n}(i-1)\alpha_i \equiv a \pmod{n}$, we can determine the value of the binary symbol deleted from $\boldsymbol{\alpha}$. Then, we are able to find a proper position in $\boldsymbol{\alpha}'$, $\ell$, to insert the deleted bit. Tenengolts points out that the index of the deleted symbol in $\mathbf{x}$, $k$, corresponds to the run in $\boldsymbol{\alpha}$ containing $\alpha_\ell$ or to the preceding run. Recall that the value of the deleted symbol, $x_k$ is easily computed through the least nonnegative residue of the second congruence, $\sum_{i=1}^{n} x_i \equiv d \pmod{q}$. Lastly, since a run in $\boldsymbol{\alpha}$ corresponds to a series of nondecreasing (or strictly decreasing) symbols in $\mathbf{x}$, we can uniquely determine the location $k$ [80].

Let us now turn our focus to the $q$-ary SVT-code (Construction 5). The second congruence is the same as that in the $q$-ary VT-code, so we can easily compute the symbol value of $x_k$. Since Construction 5 uses the same transformation on $\mathbf{x}$ to yield at $\boldsymbol{\alpha}$, using the same logic as in [80], it is sufficient to prove that $\boldsymbol{\alpha}$ can be recovered from $\boldsymbol{\alpha}'$. The first and third congruencies form a binary SVT-code over $\boldsymbol{\alpha}$, that is, $\boldsymbol{\alpha}$ is guaranteed recoverable after a single-deletion occurs given that the receiver knows the position of the deleted bit to within $P$ bits.

All that remains to be shown is that the location of the deleted bit in $\boldsymbol{\alpha}$ can be narrowed down to $P+1$ positions using knowledge of the $P$ possible positions of the deletion in $\mathbf{x}$. Central to this argument is the fact that a deletion is unique only to within its run. Deletion of symbol $x_k$ can only lead to the deletion of bit $\alpha_k$ or $\alpha_{k+1}$. Thus, knowledge that the deleted symbol is one of the following symbols $(x_i, x_{i+1}, \ldots, x_{i+P-1})$ reveals that the deleted bit in $\boldsymbol{\alpha}$ is within $(\alpha_i, \alpha_{i+1}, \ldots, \alpha_{i+P})$. Therefore, $\boldsymbol{\alpha}$ is protected by a binary SVT-code through the first and third congruencies (see Lemma 6) and is decodable through Algorithm 6 in Appendix 4.9.3. Now that $\boldsymbol{\alpha}$ is proven recoverable, the rest of the logic follows directly from the proof of Theorem 1 in [80].

□

**Lemma 9.** *There exist parameters $0 \leq c \leq P$, $0 \leq d < q$, and $e \in \{0,1\}$ such that the redundancy of the $SVT_{c,d,e}(n,P,q)$ code as defined in Construction 5 is at most $\log_q(2P+2)+1$ symbols.*

*Proof.* The $SVT_{c,d,e}(n,P,q)$ code can be viewed as a partition of all length-$n$ $q$-ary vectors into $2q(P+1)$ mutually disjoint sets. Then, using the pigeonhole principle, there must exist an $SVT_{c,d,e}(n,P,q)$ code whose cardinality is at least $(q^n)/(2q(P+1))$ and whose redundancy is thus at most $\log_q(2q(P+1)) = \log_q(2P+2)+1$ symbols. □

It is worthwhile to highlight the fact that the redundancy of the $SVT_{c,d,e}(n,P,q)$ code is independent of $n$; as shown in Lemma 9, for a given alphabet size $q$, the minimum redundancy of an $SVT_{c,d,e}(n,P,q)$ code is a function of $P$.

Table 4.1 helps to visualize the results from Lemma 9. Each row corresponds to a different alphabet size $q$, and each column corresponds to a specific number of redundancy symbols. The values in the table represent the maximum value of $P$, in terms of the $P$-bounded single-deletion correcting property, guaranteed by Lemma 9, for an SVT-code with the given alphabet size and redundancy symbols. For example, in Table 4.1, we see that for alphabet size $q=4$, there exists a parameter set $(c,d,e)$ such that an $SVT_{c,d,e}(n,7,4)$ code exists with

Table 4.1: P Guaranteed by Lemma 2

| $q$ \ red. | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 3 | - | 3 | 12 | 39 | 12 |
| 4 | 1 | 7 | 31 | 127 | 511 |
| 5 | 1 | 11 | 61 | 311 | 1561 |
| 6 | 2 | 17 | 107 | 647 | 3887 |
| 7 | 2 | 23 | 170 | 1199 | 8402 |
| 8 | 3 | 31 | 255 | 2047 | 16383 |

3 or fewer redundancy symbols.

**Example 8.** *In this example we list the codewords for the* $SVT_{0,0,0}(5,2,4)$ *code (the commas separating symbols are omitted for brevity):*

$$
\begin{aligned}
&SVT_{0,0,0}(5,2,4)\\
&= \{(21010),(11110),(01120),(32120),(00220),(22220),(32030),(00130),\\
&\quad (12230),(11330),(02330),(33330),(30001),(20011),(10111),(32021),\\
&\quad (00121),(12221),(00031),(31031),(11231),(02231),(01331),(23331),\\
&\quad (20002),(10012),(31112),(30122),(21122),(20222),(21032),(11132),\\
&\quad (01232),(00332),(22332),(13332),(10003),(30113),(21113),(30023),\\
&\quad (20123),(10223),(32223),(20033),(10133),(31233),(30333),(21333).\}
\end{aligned}
$$

*This example highlights the fact that codewords within an* $SVT_{c,d,e}(n,P,q)$ *code can have a Levenshtein distance of 2, which is not permissible in the traditional VT-code. For example, the vector* (1333) *can be received from a single deletion from either of the following codewords:* (13332) *or* (21333)*. However, notice that positions of the deletions that produce the overlapping single-deletion balls are at a distance greater than* $P$ *symbols apart (in this code* $P = 2$ *while the deleted symbols in this example span 5 positions).*

*Additionally, note that the* $SVT_{0,0,0}(5,2,4)$ *code satisfies Lemma 9:*

$$\log_4\left(\frac{4^5}{|SVT_{0,0,0}(5,2,4)|}\right) = \log_4\left(\frac{1024}{48}\right) \approx 2.21$$
$$< \log_4(2P+2)+1 = \log_4(6)+1 \approx 2.29.$$

### 4.5.3 Overall Construction

In this section we introduce the coding technique and redundancy required for both $A_m(\mathbf{x})_1$ (the first row of the codeword array) as well as the overall framework.

#### Non-Binary Run-Length Limited VT-Codes

The underlying mathematics in extending the encoding of $A_m(\mathbf{x})_1$ from binary to non-binary is relatively straightforward so we will present a brief overview. As in Section 4.4.2, we require a definition tying the maximum run-length property of a vector to its length.

**Definition 11.** *A length-n q-ary vector* $\mathbf{x}$ *is said to satisfy the* ***f(n)-RLL(n,q)*** *constraint, and is called an* ***f(n)-RLL(n,q)*** *vector, if the length of the longest run in* $\mathbf{x}$ *is at most* $f(n)$*.*

We define a set of $f(n)$-RLL$(n,q)$ vectors to be an $f(n)$-*RLL*$(n,q)$ *code*, and the set of all $f(n)$-RLL$(n,q)$ vectors is denoted by $S_n(f(n))$. As in Section 4.4.2, we denote the *capacity* of the $f(n)$-RLL$(n,q)$ constraint as

$$C(f(n)) = \limsup_{n\to\infty} \frac{\log_q(|S_n(f(n))|)}{n},$$

and for the case in which the capacity is 1, we define also the *redundancy* of the $f(n)$-RLL$(n,q)$ constraint to be

$$r(f(n)) = n - \log_q(|S_n(f(n))|).$$

**Lemma 10.** *The redundancy of the* $\log_q(2n)$*-RLL(n,q) constraint is upper bounded by 1 symbol for all* $n$*.*

*Proof.* We omit the full proof—it simply follows the proof of Lemma 4 with the modification that instead of the longest runs of fair coin flips, we are interested in the longest runs of fair ($q$-sided) die tosses. We make the change by setting the Bernoulli probability of success $p = 1/q$, and forming the new approximation of the longest run of successes in $n$ die rolls, $Y_n$, as follows [94]:

$$\begin{aligned} P(Y_n = x) \approx P(x - \log_q(n(1-1/q)) < W \\ \leq x + 1 - \log_q(n(1-1/q))), \end{aligned}$$

where $W$ is a continuous random variable whose cumulative distribution function is given by $F_W(x) = e^{-(1/q)^x}$. □

Through simple use of the pigeonhole principle in conjunction with Lemma 10, we determine the number redundancy bits required for a single-deletion correcting code with desired RLL properties in the following corollary.

**Corollary 6.** *For all* $n$*, there exists a* $q$*-ary single-deletion correcting code that satisfies the* $\log_q(2n)$*-RLL(n,q) constraint and uses at most* $\log_q(n) + 2$ *symbols of redundancy.*

### Final Construction

We use both the $q$-ary SVT-code and the single-deletion correcting RLL-constrained code to construct our final burst-deletion correcting code.

**Construction 6.** *Let* $\mathcal{C}_1$ *be a* $q$*-ary single-deletion correcting code that satisfies the* $\log_q(2n/b)-$ *RLL*$(n/b, q)$ *constraint. Let* $\mathcal{C}$ *be an* $SVT_{c,d,e}(n/b, \log_q(n/b) + 2, q)$ *for some* $0 \leq c \leq P$*,* $0 \leq d < q$*, and* $e \in \{0, 1\}$*. The code* $\mathcal{C}$ *is constructed as follows*

$$\mathcal{C} \triangleq \{\mathbf{x} : A_b(\mathbf{x})_1 \in \mathcal{C}_1, A_b(\mathbf{x})_i \in \mathcal{C}_2, \text{ for } 2 \leq i \leq b\}.$$

The burst deletion correcting capabilities of the code follow directly from the arrangement of the constituent codes within the $A_b(\mathbf{x})$ codeword array framework.

**Theorem 9.** *The code $\mathcal{C}$ from Construction 6 is a $q$-ary $m$-burst-deletion-correcting code.*

Through the pigeonhole principle, we arrive at total redundancy for the overall code.

**Corollary 7.** *There exists a $q$-ary $b$-burst-deletion-correcting code whose number of redundancy symbols is at most*

$$\log_q(n/b) + (b-1)\log_q(2\log_q(n/b) + 6) + b + 1.$$

## 4.6 Correcting a Burst of Length at most $b$ (consecutively)

In this section, we return to the binary regime and consider the problem of correcting a burst of consecutive deletions of length at most $b$. As defined in Section 4.2, a code capable of correcting a burst of at most $b$ consecutive deletions needs to be able to correct any burst of size $a$ for $a \leq b$. For the remainder of this section, we assume that $(b!)|n$.

The case $b = 2$ was already solved by Levenshtein with a construction that corrects a single deletion or a deletion of two adjacent bits [83]. The redundancy of this code, denoted by $\mathcal{C}_L(n)$, is at most $1 + \log(n)$ bits. Hence this code asymptotically achieves the upper bound for correcting a burst of exactly 2 deletions.

The general strategy we use in correcting a burst of length *at most* $b$ is similar to the work in [97] where the author proposed constructing codes capable of correcting bursts of deletions by intersecting codes that correct a fixed number of deletions. In particular, we construct a code from the intersection of the code $\mathcal{C}_L(n)$ with the codes that correct a burst of length *exactly* $i$, for $3 \leq i \leq b$. We refer to each $i$ as a *level* and in each level we will have a set of codes which forms a partition of the space. Thus, our overall code will be the largest intersection of the codes at each level.

Let us first introduce a simple code construction that can be used as a baseline comparison. We use Construction 1 from [84], which is reviewed in Section 4.2.2, to form the code in each level $3 \leq i \leq b$. Note that in each level we can have a family of codes which forms a partition of the space. Then, the intersection of the codes in each level together with $\mathcal{C}_L(n)$ forms a code that corrects a burst of consecutive deletions of length at most $b$. This baseline code will be denoted by $\mathcal{C}_B(n)$ whenever we refer to it in the following.

As we mentioned above, the redundancy of the code $\mathcal{C}_L(n)$ is $\log(n)+1$ and it partitions the space into $2n$ codebooks. Similarly, for $3 \leq i \leq b$, the redundancy of the codes from [84] in the $i$th level is $i\left(\log(n/i+1)\right)$, and they partition the space into $\left(\frac{n}{i}+1\right)^i$ codebooks. Therefore, we can only claim that the redundancy of this code construction will be approximately

$$\log(2n)+\sum_{i=3}^{b} i\left(\log\left(\frac{n}{i}+1\right)\right) \geq \left(\binom{b}{2}-2\right)\log(n)-\log\left(\prod_{i=2}^{b} i!\right).$$

The approach we take in this section is to build upon the codes we develop in Section 4.4 and leverage them as the codes in each level instead of the ones from [84]. However, since the codes from Section 4.4 do not provide a partition of the space we will have to make one additional modification in their construction so it will be possible to intersect the codes in each level and get a code which corrects a burst of size at most $b$.

Recall that in our code from Construction 4 we needed the first row in our codeword array, $A_b(\mathbf{x})_1$, to be run-length limited so that the remaining rows could effectively use the SVT-code. Similarly, in order to correct at most $b$ consecutive deletions we want the first row of each level's codeword array to be an $N_b$-RLL($\frac{n}{i}$)-vector, where $N_b = \lceil \log(n\log(b)) \rceil + 1$. In other words, $A_i(\mathbf{x})_1$ will satisfy the $N_b$-RLL($\frac{n}{i}$) constraint for $3 \leq i \leq b$. Note that the $f(n)$-RLL($\frac{n}{i}$) constraint does not depend on $i$. We add the term *universal* to signify that an RLL-constraint on a vector refers to the RLL-constraint on the first row of each level.

**Definition 12.** *A length-$n$ binary vector $\mathbf{x}$ is said to satisfy the* ***f(n)-URLL****$(n,b)$ constraint,*

*and is called an* ***f(n)-URLL****$(n,b)$ vector, if the length of each run of 0's or 1's in $A_i(\mathbf{x})_1$ for $3 \leq i \leq b$, is not greater than $f(n)$. Additionally, the set of all* ***f(n)-URLL****$(n,b)$ vectors is denoted by $U_{n,b}(f(n))$.*

We define the *redundancy* of the $f(n)$-URLL$(n,b)$ constraint to be

$$r_U(f(n)) = n - \log(|U_{n,b}(f(n))|).$$

**Lemma 11.** *The redundancy of the $N_b$-URLL(n,b) constraint is upper bounded by* $\log(\log(b))-1$ *bits:*

$$r_U(N_b) \leq \log(\log(b)) - 1.$$

*Proof.* Using the union bound, we can derive an upper bound on the fraction of sequences in which $A_i(\mathbf{x})_1$ does not satisfy the $N_b$-RLL$(\frac{n}{i})$ constraint for $3 \leq i \leq b$.

$$\begin{aligned}
\frac{|\{\mathbf{x} : A_i(\mathbf{x})_1 \notin S_{\frac{n}{i}}(N_b)\}|}{2^n} &\leq \frac{n}{i} \cdot \left(\frac{1}{2}\right)^{N_b-1} \\
&= \frac{n}{i} \cdot \left(\frac{1}{2}\right)^{\lceil \log(n\log(b)) \rceil} \\
&\leq \frac{n}{in\log(b)} \\
&= \frac{1}{i\log(b)}.
\end{aligned}$$

Using the previous result we find an upper bound on the fraction of sequences which do not satisfy the universal RLL-constraint.

$$\begin{aligned}
\frac{|\{\mathbf{x} : \mathbf{x} \notin U_{n,b}(N_b)\}|}{2^n} &\leq \sum_{i=3}^{b} \left(\frac{1}{i\log(b)}\right) \\
&= \left(\frac{1}{\log(b)}\right) \sum_{i=3}^{b} \left(\frac{1}{i}\right) \\
&< \left(\frac{1}{\log(b)}\right) (\ln(b) - 2) \\
&= 1 - \frac{2}{\log(b)},
\end{aligned}$$

where the last inequality holds since $\sum_{i=1}^{n}(1/i) < \ln(n)+1$, for all $n$. Therefore, we can lower bound the total number of sequences that meet our universal RLL-constraint by:

$$|\{\mathbf{x} : \mathbf{x} \in U_{n,b}(N_b)\}| > 2^n \left[1 - \left(1 - \frac{2}{\log(b)}\right)\right] = \frac{2^{n+1}}{\log(b)}.$$

Finally, we derive an upper bound on the redundancy of the set $U_{n,b}(N_b)$ to be

$$\begin{aligned} r_U(N_b) &= n - \log(|U_{n,b}(N_b)|) \\ &< n - \log\left(\frac{2^{n+1}}{\log(b)}\right) \\ &= n - (n+1) + \log(\log(b)) \\ &= \log(\log(b)) - 1. \end{aligned}$$

□

In addition to limiting the longest run in the first row of every level, each vector $A_i(\mathbf{x})_1$ should be able to correct a single deletion. We define the following family of codes.

**Construction 7.** *Let $n$ be a positive integer and $\mathbf{a} = (a_3, \ldots, a_b)$ a vector of non-negative integers such that $0 \le a_i \le n/i$ for $3 \le i \le b$. The $\overline{VT}_{\mathbf{a},f(n)}(n)$ code is defined as follows:*

$$\overline{VT}_{\mathbf{a},f(n)}(n) \triangleq \left\{\mathbf{x} \ : \ A_i(\mathbf{x})_1 \in VT_{a_i}\left(\frac{n}{i}\right), 3 \le i \le b, \mathbf{x} \in U_{n,b}(f(n))\right\}.$$

**Lemma 12.** *For all $n$, there exists vector $\mathbf{a} = (a_3, \ldots, a_b)$ such that $0 \le a_i \le n/i$ for all $3 \le i \le b$ and*

$$|\overline{VT}_{\mathbf{a},f(n)}(n)| \ge \frac{|U_{n,b}(f(n))|}{n^{b-2}}$$

*Proof.* For $3 \le i \le b$, the VT-code $VT_{a_i}\left(\frac{n}{i}\right)$ for $A_i(\mathbf{x})_1$ forms a partition of all length-$n$ binary sequences into $\frac{n}{i}+1$ different codebooks. Using the pigeonhole principle, we can determine the lower bound of the maximum intersection between the $\frac{n}{i}+1$ codebooks on

each level and $U_n(f(n))$ to get

$$\max_{\mathbf{a}} \left\{ |\overline{VT}_{\mathbf{a},f(n)}(n)| \right\} = \frac{|U_{n,b}(f(n))|}{\prod_{i=3}^{b}\left(\frac{n}{i}+1\right)} \geq \frac{|U_{n,b}(f(n))|}{n^{b-2}}$$

□

We combine Lemma 11 and Lemma 12 to find the total redundancy required to satisfy our conditions for the first rows in the codeword arrays. To simplify notation, in the rest of this section whenever we refer to a vector $\mathbf{a}$ we refer to $\mathbf{a} = (a_3, \ldots, a_b)$ where $0 \leq a_i \leq n/i$ for $3 \leq i \leq b$.

**Corollary 8.** *For all $n$, there exists a vector $\mathbf{a} = (a_3, \ldots, a_b)$ such that the redundancy of the code $\overline{VT}_{\mathbf{a},N_b}(n)$ is at most $(b-2)\log(n) + \log(\log(b))$ bits.*

With the universal RLL-constraint in place, we can use the SVT-codes defined in Section 4.4 for each of the remaining rows in each level.

**Construction 8.** *Let $\mathcal{C}_L(n)$ be the code from [83], $\mathcal{C}_1$ be the code $\overline{VT}_{\mathbf{a},N_b}(n)$ for some vector $\mathbf{a}$, and for $3 \leq i \leq b$ let $\mathcal{C}_{2,i}$ be a shifted VT-code $SVT_{c_i,d_i}(n/i, N_b+1)$ for $0 \leq c_i \leq n/i$ and $d_i \in \{0,1\}$. The code $\mathcal{C}$ is constructed as follows*

$$\mathcal{C} \triangleq \{\mathbf{x} : \ \mathbf{x} \in \mathcal{C}_L(n), \mathbf{x} \in \mathcal{C}_1 A_i(\mathbf{x})_j \in \mathcal{C}_{2,i}, \ \textit{for } 3 \leq i \leq b, 2 \leq j \leq i\}.$$

**Theorem 10.** *The code $\mathcal{C}$ from Construction 8 can correct any consecutive deletion burst of size at most $b$.*

*Proof.* Assume $\mathbf{x} \in \mathcal{C}$ is the transmitted vector and $\mathbf{y} \in D_i(\mathbf{x})$ is the received vector, $0 \leq i \leq b$. First, by the length of $\mathbf{y}$ we can easily determine the value of $i$. Recall that the received vector $\mathbf{y}$ can be represented by an $i \times (n/i - 1)$ array $A_i(\mathbf{y})$ in which every row is received by a single deletion of the corresponding row in $A_i(\mathbf{x})$.

Since the first row $A_i(\mathbf{x})_1$ belongs to a $\overline{VT}_{\mathbf{a},N_b}(n)$ code, the decoder of this code can successfully decode and insert the deleted bit in the first row of $A_i(\mathbf{y})$. Furthermore, since

every run in $A_i(\mathbf{x})_1$ consists of at most $N_b$ bits, the locations of the deleted bits in the remaining rows are known within $N_b + 1$ consecutive positions. Finally, the remaining $i - 1$ rows decode their deleted bit since they belong to a shifted VT-code $SVT_{c_i,d_i}(n/i, N_b + 1)$ (Lemma 6). □

The following example lists the steps to create the code from Construction 8.

**Example 9.** *In this example we create a code $\mathcal{C}$ of length $n$ that can correct any consecutive deletion burst of size at most 4. We construct 3 separate codes and then take the intersection to form our overall code as follows.*

- *Create the code $\mathcal{C}_L(n)$ from [83]. (This takes care of burst deletions of size 1 or 2.)*
- *Create the code $\overline{VT}_{\mathbf{a},N_b}(n)$, which we will call $\mathcal{C}_1$.*
  - *Calculate $N_b = \lceil \log(n \log(b)) \rceil + 1$.*
  - *Create $\mathcal{C}_1' = \{\mathbf{x} : A_3(\mathbf{x})_1 \in VT_{0,N_b}(n/3)\}$.*
  - *Create $\mathcal{C}_1'' = \{\mathbf{x} : A_4(\mathbf{x})_1 \in VT_{0,N_b}(n/4)\}$.*
  - *Set $\mathcal{C}_1 = \mathcal{C}_1' \cap \mathcal{C}_1''$.*
- *Create $\mathcal{C}_2$, which corresponds to the remaining rows in $A_3(\mathbf{x})$ and $A_4(\mathbf{x})$.*
  - *Create $\mathcal{C}_2' = \{\mathbf{x} : A_3(\mathbf{x})_2, A_3(\mathbf{x})_3 \in SVT_{0,0}(n/3, N_b + 1)\}$.*
  - *Create $\mathcal{C}_2'' = \{\mathbf{x} : A_4(\mathbf{x})_2, A_4(\mathbf{x})_3, A_4(\mathbf{x})_4 \in SVT_{0,0}(n/4, N_b + 1)\}$.*
  - *Set $\mathcal{C}_2 = \mathcal{C}_2' \cap \mathcal{C}_2''$.*
- *Set $\mathcal{C} = \mathcal{C}_L(n) \cap \mathcal{C}_1 \cap \mathcal{C}_2$.*

To conclude, we calculate the amount of redundancy bits needed for Construction 8.

**Corollary 9.** *For sufficiently large $n$, there exists a code which can correct a consecutive deletion burst of size at most $b$ whose number of redundancy bits is at most*

$$(b-1)\log(n) + \left(\binom{b}{2} - 1\right)\log(\log(n)) + \binom{b}{2} + \log(\log(b)).$$

*Proof.* As previously noted, the code $\mathcal{C}_L(n)$ requires $\log(n)+1$ redundancy bits. Corollary 8 yields the total number of redundancy bits required for $\mathcal{C}_1$. For each level $i$, $3 \leq i \leq b$, there are $i-1$ rows we encode with an SVT-code, which yields $\binom{b}{2}-1$ total rows. The redundancy for the SVT-code is given by Lemma 7. □

Note that Corollary 9 yields a redundancy substantially lower than the redundancy required for the baseline comparison code $\mathcal{C}_B(n)$ which was introduced in the beginning of this section. In the latter code the $\log(n)$ redundancy term is quadratic in $b$, while in the redundancy in Corollary 9 the $\log(n)$ term is linear in $b$.

## 4.7 Correcting a Burst of Length at most $b$ (non-consecutively)

In this section, we will describe a construction for correcting a non-consecutive deletion burst of length at most $b$ for $b \leq 4$. Note that for $b = 1$, we can use a VT-code and for $b = 2$, we use Levenshtein's construction [83]. The construction uses a code which can correct two deletions immediately followed by an insertion. For the remainder of this section, we assume that $(b!)|n$.

### 4.7.1 A 2-Deletion-1-Insertion-Burst Correcting Code

This subsection describes a code that corrects a deletion burst of size 2 followed by an insertion at the same position. For shorthand, we refer to this type of error as a $(2,1)$-*burst*, such a code is called a $(2,1)$-*burst-correcting code*, and the set of all $(2,1)$-bursts of a vector $\mathbf{x}$ is denoted by $D_{2,1}(\mathbf{x})$. For instance, if the vector $\mathbf{x} = (0,1,0,0,1,0) \in \mathbb{F}_2^6$ is transmitted then the set of possible received sequences given that a single $(2,1)$-burst occurs to $\mathbf{x}$ is

$$D_{2,1}(\mathbf{x}) := \{(0,0,0,1,0), (1,0,0,1,0), (0,1,0,1,0),$$
$$(0,1,1,1,0), (0,1,0,0,0), (0,1,0,0,1)\}.$$

Note that $D_1(\mathbf{x}) \subseteq D_{2,1}(\mathbf{x})$ and hence every $(2,1)$-burst-correcting code is a single-deletion-correcting code as well.

We now introduce a construction for $(2,1)$-burst-correcting codes.

**Construction 9.** *For three integers $n \geq 4$, $a \in \mathbb{Z}_{2n-1}$, and $c \in \mathbb{Z}_4$, the code $\mathcal{C}_{2,1}(n,a,c)$ is defined as follows:*

$$\mathcal{C}_{2,1}(n,a,c) \triangleq \left\{\mathbf{x} \in \mathbb{F}_2^n : \sum_{i=1}^{n} x_i \equiv c \pmod 4, \sum_{i=1}^{n} i \cdot x_i \equiv a \pmod{(2n-1)}\right\}.$$

Notice that $\mathcal{C}_{2,1}(n,a,c)$ is a single-deletion-correcting code [82].

In order to prove the correctness of this construction, we introduce some additional terminology. For $(b_1,b_2) \in \mathbb{F}_2^2$, $d \in \mathbb{F}_2$, and $\mathbf{x} \in \mathbb{F}_2^n$ let $D_{2,1}(\mathbf{x})^{(b_1,b_2)\to d} \subseteq D_{2,1}(\mathbf{x})$ be the set of vectors from $D_{2,1}(\mathbf{x})$ that result from the deletion of the subvector $(b_1,b_2)$ followed by the insertion of $d$. For example, for the vector $\mathbf{x} = (0,1,0,0,0,1,0)$,

$$D_{2,1}^{(0,0)\to 1}(\mathbf{x}) = \{(0,1,1,0,1,0),(0,1,0,1,1,0)\},$$
$$D_{2,1}^{(0,0)\to 0}(\mathbf{x}) = \{(0,1,0,0,1,0)\}.$$

The following claim follows in a straightforward manner.

**Claim 1.** *For any $(d,b_1,b_2) \notin \{(1,0,0),(0,1,1)\}$ $D_{2,1}^{(b_1,b_2)\to d}(\mathbf{x}) \subseteq D_1(\mathbf{x})$.*

We are now ready to prove the correctness of Construction 9.

**Theorem 11.** *Let $n \geq 4$, $a \in \mathbb{Z}_{2n-1}$, and $c \in \mathbb{Z}_4$ be three integers. Then, the code $\mathcal{C}_{2,1}(n,a,c)$ from Construction 9 is a $(2,1)$-burst-deletion correcting code.*

*Proof.* We will show that for all $\mathbf{x},\mathbf{y} \in \mathcal{C}_{2,1}(n,a,c)$, $\mathcal{D}_{2,1}(\mathbf{x}) \cap \mathcal{D}_{2,1}(\mathbf{y}) = \emptyset$.

Assume in the contrary that $\mathbf{z} \in \mathcal{D}_{2,1}(\mathbf{x}) \cap \mathcal{D}_{2,1}(\mathbf{y})$. Then, there exist $(d,b_1,b_2),(d',b_1',b_2')$ such that

$$\mathbf{z} \in \mathcal{D}_{2,1}^{(b_1,b_2)\to d}(\mathbf{x}) \cap \mathcal{D}_{2,1}^{(b_1',b_2')\to d'}(\mathbf{y}),$$

and assume also that $\mathbf{z}$ is the result of deleting bits $i$ and $i+1$ from $\mathbf{x}$ and $j$ and $j+1$ from $\mathbf{y}$, and without loss of generality $i < j$.

Since $\mathcal{C}_{2,1}(n, a, c)$ is a single-deletion-correcting code, according to Claim 1, we can assume that at least one of $(d, b_1, b_2), (d', b_1', b_2')$ belongs to the set $\{(0,1,1),(1,0,0)\}$, and without loss of generality, assume that $(d, b_1, b_2) \in \{(0,1,1),(1,0,0)\}$. First suppose $(d, b_1, b_2) = (1,0,0)$. Since $\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i \equiv 0 \pmod 4$, we have $(b_1', b_2') = (0,0) = (b_1, b_2)$. Furthermore, since $\mathbf{z} \in \mathcal{D}_{2,1}^{(b_1,b_2)\to d}(\mathbf{x}) \cap \mathcal{D}_{2,1}^{(b_1',b_2')\to d'}(\mathbf{y})$, $d' + b_1 + b_2 \equiv d + b_1' + b_2' \pmod 4$ and so $d' = d = 1$. Next, suppose $(d, b_1, b_2) = (0,1,1)$. Then, using idential logic $(b_1', b_2') = (b_1, b_2) = (1,1)$ and $d' = d = 0$ so that we conclude that if one of $(d, b_1, b_2), (d', b_1', b_2')$ is in the set $\{(0,1,1),(1,0,0)\}$, then $(d, b_1, b_2) = (d', b_1', b_2')$.

We consider the case where $(d, b_1, b_2) = (0,1,1)$. In this case, $\mathbf{x}, \mathbf{y}$ will have the following structure:

$$\begin{aligned}
\mathbf{x} =& (x_1, \ldots, x_{i-1},\ 1,1,\ x_{i+2}, \ldots, x_j,\quad 0,\quad x_{j+2}, \ldots x_n),\\
\mathbf{y} =& (y_1, \ldots, y_{i-1},\ 0,\quad y_{i+1}, \ldots, y_{j-1},\ 1,1,\ y_{j+2}, \ldots y_n),
\end{aligned}$$

where $x_\ell = y_\ell$ for $1 \le \ell \le i-1$ and $j+2 \le \ell \le n$, and $x_{i+2} = y_{i+1}$, $x_{i+3} = y_{i+2}$, $x_{i+4} = y_{i+3}, \ldots, x_j = y_{j-1}$. Since $\mathbf{x} \neq \mathbf{y}$ and $j - i > 0$, we have

$$\begin{aligned}
&\sum_{\ell=1}^{n} \ell \cdot y_\ell - \sum_{\ell=1}^{n} \ell \cdot x_\ell = \sum_{\ell=i}^{j+1} \ell \cdot y_\ell - \sum_{\ell=i}^{j+1} \ell \cdot x_\ell\\
=&(2j+1) - (2i+1) - \mathrm{wt}((x_{i+2}, \ldots, x_j))\\
=&2(j-i) - \mathrm{wt}((x_{i+2}, \ldots, x_j)),
\end{aligned}$$

where $\mathrm{wt}((x_{i+2}, \ldots, x_j))$ denotes the Hamming weight of $(x_{i+2}, \ldots, x_j)$. Since

$$0 \le \mathrm{wt}((x_{i+2}, \ldots, x_j)) \le j - i - 1,$$

we conclude that

$$2 \leq j - i + 1 \leq \sum_{\ell=1}^{n} \ell \cdot y_\ell - \sum_{\ell=1}^{n} \ell \cdot x_\ell \leq 2(j-i) \leq 2(n-1),$$

in contradiction to $\sum_{\ell=1}^{n} \ell \cdot y_\ell - \sum_{\ell=1}^{n} \ell \cdot x_\ell \equiv 0 \pmod{(2n-1)}$. The case where $(d, b_1, b_2) = (1, 0, 0)$ can be proven in a similar manner and so the details are omitted. Therefore, we conclude that $\mathcal{D}_{2,1}(\mathbf{x}) \cap \mathcal{D}_{2,1}(\mathbf{y}) = \emptyset$ and thus $\mathcal{C}_{2,1}(n, a, c)$ is a single-deletion-correcting code.

□

The following corollary summarizes this discussion.

**Corollary 10.** *For all $n \geq 4$ there exist $a \in \mathbb{Z}_{2n-1}$ and $c \in \mathbb{Z}_4$ such that the redundancy of the code $\mathcal{C}_{2,1}(n, a, c)$ from Construction 9 is at most $\log(4(2n-1)) < \log(n) + 3$.*

### 4.7.2 Correcting a Burst of Length at most $b$

We are now ready to show our constructions for $b = 3, 4$.

**Construction 10.** *Let $\mathcal{C}_3$ denote the code from Construction 4 for $b = 3$. For integers $n$ and $a_1 \in \mathbb{Z}_n$, $a_2, a_3 \in \mathbb{Z}_{n-1}$, $c_2, c_3 \in \mathbb{Z}_4$, let $\mathcal{C}_{b\leq 3}(n, a_1, a_2, a_3, c_2, c_3)$ be the following code:*

$$\begin{aligned}\mathcal{C}_{b\leq 3} \triangleq \Big\{ \mathbf{x} \in \mathbb{F}_2^n : \; & \mathbf{x} \in VT_{a_1}(n), \\ & \mathbf{x} \in \mathcal{C}_3, \\ & A_2(\mathbf{x})_1 \in \mathcal{C}_{2,1}\left(\frac{n}{2}, a_2, c_2\right), \\ & A_2(\mathbf{x})_2 \in \mathcal{C}_{2,1}\left(\frac{n}{2}, a_3, c_3\right) \Big\}.\end{aligned}$$

**Theorem 12.** *The code from Construction 10 can correct a non-consecutive deletion burst of size at most three.*

*Proof.* Let $\mathbf{x}$ be the transmitted codeword and $\mathbf{y}$ is the received vector. From the length of the received vector $\mathbf{y}$, we know the number of deletions that occurred, denoted by $k$. If

$k = 1$, the deletion can be corrected since $\mathbf{x}$ is a codeword of the VT-code $VT_{a_1}(n)$. If $k = 3$, we have a *consecutive* deletion burst of size three which can be corrected since $\mathbf{x}$ is a codeword in $\mathcal{C}_3$, which is a three-burst-deletion-correcting code.

If $k = 2$, then the $(2, 1)$-burst correcting code succeeds in any case as will be shown in the following. If the two deletions occur consecutively, each of the two rows of the array $A_2(\mathbf{y})$ corresponds to a codeword from a code $\mathcal{C}_{2,1}$ with a single deletion which can be corrected. If the two deletions occur at positions $i$ and $i + 2$ (they have to be within three bits), then:

$$\mathbf{y} = (x_1, \ldots, x_{i-1}, x_{i+1}, x_{i+3}, \ldots, x_n)$$

and (assuming w.l.o.g. that $i$ is even)

$$A_2(\mathbf{y}) = \begin{bmatrix} x_1 & x_3 & \ldots & x_{i-3} & x_{i-1} & x_{i+3} & \ldots & x_{n-1} \\ x_2 & x_4 & \ldots & x_{i-2} & x_{i+1} & x_{i+4} & \ldots & x_n \end{bmatrix}.$$

Compared to $A_2(\mathbf{x})$, the first row suffers from a single deletion ($x_{i+1}$) and the second from two deletions ($x_i$ and $x_{i+2}$) immediately followed by an insertion ($x_{i+1}$). This can also be corrected by the code $\mathcal{C}_{2,1}$. If $i$ is odd, there is a single deletion in the second row and two deletions followed by one insertion in the first row. □

**Theorem 13.** *There exists a code by Construction 10 which can correct a non-consecutive burst of size at most 3 with redundancy at most* $4\log(n) + 2\log(\log(n)) + 6$.

*Proof.* The set of $n + 1$ VT-codes $VT_{a_1}(n)$ for $0 \leq a_1 \leq n$ as well as the set of $n$ codes $\mathcal{C}_{2,1}(n, a_2, c)$ and $\mathcal{C}_{2,1}(n, a_3, c)$ for $0 \leq a_2, a_3 \leq n - 1, 0 \leq c \leq 3$ form partitions of the space; i.e., $\cup_{a_1=0}^{n} VT_{a_1}(n) = \mathbb{F}_2^n$, $\cup_{a_2=0}^{n-1} \cup_{c=0}^{3} \mathcal{C}_{2,1}(n, a_2, c) = \mathbb{F}_2^n$ and $\cup_{a_3=0}^{n-1} \cup_{c=0}^{3} \mathcal{C}_{2,1}(n, a_3, c) = \mathbb{F}_2^n$. In particular, they also form a partition of the code $\mathcal{C}_3$ from Construction 4. Therefore, by the pigeonhole principle, there are choices for $a_1, a_2, a_3, c$ such that the intersection of the three codes requires redundancy at most the sum of the redundancies of the three codes. □

We now turn to the case of $b = 4$, which follows the same ideas as for $b = 3$, so we explain

its main ideas.

**Construction 11.** *Let $\mathcal{C}_4$ denote the code from Construction 4 for $b = 4$. For integers $n$ and $a_1, a_2 \in \mathbb{Z}_{n-1}$, $b_1, b_2, b_3 \in \mathbb{Z}_{2n/3-1}$, $c_1, c_2, d_1, d_2, d_3 \in \mathbb{Z}_4$, let $\mathcal{C}_{b\leq 4}$ be as follows:*

$$\begin{aligned}\mathcal{C}_{b\leq 4} \triangleq \Big\{\mathbf{x} \in \mathbb{F}_2^n : \mathbf{x} \in VT_{a_1}(n),& \\ \mathbf{x} \in \mathcal{C}_4,& \\ A_2(\mathbf{x})_i \in \mathcal{C}_{2,1}\left(\frac{n}{2}, a_i, c_i\right), i = 1, 2,& \\ A_3(\mathbf{x})_i \in \mathcal{C}_{2,1}\left(\frac{n}{3}, b_i, d_i\right), i = 1, 2, 3\Big\}.&\end{aligned}$$

**Theorem 14.** *The code from Construction 11 can correct a non-consecutive deletion burst of size at most four.*

*Proof.* Let $\mathbf{x}$ be the transmitted codeword and $\mathbf{y}$ is the received vector. As for $b \leq 3$, we know the number of deletions that occurred, denoted by $k$. If $k = 1$, the deletion can be corrected since each codeword is from a VT-code. If $k = 4$, we have a *consecutive* deletion burst of size four which can be corrected since each codeword of $\mathcal{C}_{b\leq 4}$ is a codeword of $\mathcal{C}_4$. If $k = 2$, the following cases can happen:

- The two deletions occur consecutively, then each row of $A_2(\mathbf{x})$ is affected by a single deletion.
- The two deletions occur with one position in between, then one row is affected by a single deletion and the other one by a $(2, 1)$-burst (similar to the proof of Theorem 12).
- There are two positions between the two deletions, i.e., positions $i$ and $i+3$ are deleted. Then:

$$\mathbf{y} = (x_1, \ldots, x_{i-1}, x_{i+1}, x_{i+2}, x_{i+4}, \ldots, x_n)$$

and (assuming w.l.o.g. that $i$ is even)

$$A_2(\mathbf{y}) = \begin{bmatrix} x_1 & \dots & x_{i-1} & x_{i+2} & x_{i+5} & \dots & x_{n-1} \\ x_2 & \dots & x_{i+1} & x_{i+4} & x_{i+6} & \dots & x_n \end{bmatrix}$$

and both rows are affected by a $(2,1)$-burst.

Since the rows of $A_2(\mathbf{x})$ are codewords of $\mathcal{C}_{2,1}$, we can correct the deletions in any of these cases.

Similarly, for $k = 3$, the following cases can happen:

- The three deletions occur consecutively, then each row of $A_3(\mathbf{x})$ is affected by a single deletion.
- The deletions occur at positions $i$, $i+1$ and $i+3$. Then:

  $$\mathbf{y} = (x_1, \dots, x_{i-1}, x_{i+2}, x_{i+4}, \dots, x_n)$$

  and (assuming w.l.o.g. that $i$ is divisible by three)

  $$A_2(\mathbf{y}) = \begin{bmatrix} x_1 & \dots & x_{i-2} & x_{i+4} & \dots & x_{n-2} \\ x_2 & \dots & x_{i-1} & x_{i+5} & \dots & x_{n-1} \\ x_2 & \dots & x_{i+2} & x_{i+6} & \dots & x_n \end{bmatrix},$$

  then the last row is affected by a $(2,1)$-burst and the other ones by a single deletion.
- The deletions occur at positions $i$, $i+2$ and $i+3$. Then, similarly to before, two rows are affected by a single deletion and one row by a $(2,1)$-burst.

Since the rows of $A_3(\mathbf{x})$ are codewords of $\mathcal{C}_{2,1}$, we can correct the deletions in either of these cases. □

The next theorem summarizes this construction and its redundancy. The redundancy follows as in Theorem 12 by the pigeonhole principle.

**Theorem 15.** *There exists a code constructed by Construction 11 with redundancy at most* $7\log(n) + 2\log(\log(n)) + 4$.

We note that for $b > 4$ we cannot extend this idea and it remains as an open problem to construct efficient codes for correcting a non-consecutive burst of deletions of size $b > 4$. These constructions give some first ideas to correct a burst of non-consecutive deletions/insertions. The only construction we are aware of that can correct the class of bursts studied in this section with asymptotic rate 1 is the one from [92] which corrects arbitrary number of deletions, and in particular any kind of burst. However, the authors of [92] used asymptotic considerations which do not explicitly state the exact redundancy, and thus we could not provide a comparison with their codes. Moreover, we believe that our constructions for $b \leq 4$ are more practical.

## 4.8 Concluding Remarks

In this chapter, we have studied codes for correcting a burst of deletions or insertions in three models. Our main contribution is the construction of binary $b$-burst-deletion-correcting codes with redundancy at most $\log(n) + (b-1)\log(\log(n)) + b - \log(b)$ bits and a non-asymptotic upper bound on the cardinality of such codes. We have extended this construction to codes which correct a consecutive burst of size at most $b$, and studied codes which correct a burst of size at most $b$ (not necessarily consecutive) for the cases $b = 3, 4$. Additionally we created a non-binary code that corrects a burst deletion of exactly $m$ symbols. By setting the alphabet size to 4, this code can be specifically suited for correcting burst-deletions in DNA storage.

**Acknowledgment**

The majority of the material in this chapter was published in [98] and [99]. The author thanks Prof. Eitan Yaakobi, Prof. Antonia Wachter-Zeh, and Ryan Gabrys for their contributions, particularly for Sections 4.6 and 4.7. The author also thanks Frederic Sala and Prof. Lara Dolecek for their contributions, particularly for Section 4.5.

## 4.9 Appendix

### 4.9.1 Calculating the value of $N(n, b, i)$

In this appendix we calculate the value of $N(n, b, i) = |\{\mathbf{x} \in \mathbb{F}_2^n : |D_b(\mathbf{x})| = i\}|$.

**Lemma 13.** *For $1 \leq i \leq n - b + 1$ we have that*

$$N(n, b, i) = 2^b \binom{n-b}{i-1}.$$

*Proof.* Recall that we can arrange a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ into a $b \times \frac{n}{b}$ array $A_b(\mathbf{x})$.

Let $\rho(\mathbf{x}_j)$ denote the number of runs in the $j$th row of $A_b(\mathbf{x})$. From equation (4.2), we have that

$$|D_b(\mathbf{x})| = \left( \sum_{j=1}^{b} \rho(\mathbf{x}_j) \right) - b + 1.$$

Thus, counting the number of vectors of length $n$ whose $b$-burst deletions ball size is $i$, is equivalent to counting the number of vectors of length $n$ for which

$$\left( \sum_{j=1}^{b} \rho(\mathbf{x}_j) \right) = i + b - 1.$$

The number of binary vectors of length $n$ with $r$ runs is

$$2\binom{n-1}{r-1} \triangleq M(n, r).$$

For $b = 2$, $N(n, 2, i)$ is given by

$$\begin{aligned}
&\sum_{0<r_1,r_2:r_1+r_2=i+2-1} M\left(\frac{n}{2}, r_1\right) \cdot M\left(\frac{n}{2}, r_2\right) \\
&= \sum_{r_1=1}^{i} M\left(\frac{n}{2}, r_1\right) \cdot M\left(\frac{n}{2}, i+1-r_1\right) \\
&= \sum_{r_1=1}^{i} 2\binom{\frac{n}{2}-1}{r_1-1} \cdot 2\binom{\frac{n}{2}-1}{i-r_1}
\end{aligned}$$

$$= 4 \sum_{r_1=0}^{i-1} \binom{\frac{n}{2}-1}{r_1} \cdot \binom{\frac{n}{2}-1}{i-1-r_1}$$
$$= 4\binom{n-2}{i-1}.$$

We used Vandermonde's identity in the final step which states that for any nonnegative integer $n$ the following relation holds true:

$$\sum_{k=0}^{n} \binom{x}{k}\binom{y}{n-k} = \binom{x+y}{n}.$$

We prove the lemma's statement by induction on $b$. We have already established the base case for $b = 2$ (the $b = 1$ case is trivially given by $M(n, r)$).

Assume the following holds for $b = k$:

$$\sum_{0<r_1,r_2,\ldots,r_k:\ r_1+r_2+\ldots+r_k=i+k-1} M\left(\frac{n}{k}, r_1\right) \cdot M\left(\frac{n}{k}, r_2\right) \cdots M\left(\frac{n}{k}, r_k\right) = 2^k \binom{n-k}{i-1}.$$

We wish to show that for $b = k + 1$,

$$\sum_{\substack{0<r_1,r_2,\ldots,r_{k+1}: \\ r_1+r_2+\ldots+r_{k+1}=i+k}} M\left(\frac{n}{k+1}, r_1\right) \cdot M\left(\frac{n}{k+1}, r_2\right) \cdots M\left(\frac{n}{k+1}, r_{k+1}\right)$$
$$= 2^{k+1}\binom{n-(k+1)}{i-1}.$$

Let us now prove the previous equation using the inductive assumption:

$$\sum_{\substack{0<r_1,r_2,\ldots,r_{k+1}: \\ r_1+r_2+\ldots+r_{k+1}=i+k}} M\left(\frac{n}{k+1}, r_1\right) \cdot M\left(\frac{n}{k+1}, r_2\right) \cdots M\left(\frac{n}{k+1}, r_{k+1}\right)$$
$$= \sum_{r_{k+1}=1}^{i} M\left(\frac{n}{k+1}, r_{k+1}\right) \cdot \sum_{\substack{0<r_1,r_2,\ldots,r_k: \\ r_1+r_2+\ldots+r_k=i+k-r_{k+1}}} M\left(\frac{n}{k+1}, r_1\right) \cdots M\left(\frac{n}{k+1}, r_k\right) \tag{4.4}$$
$$= \sum_{r_{k+1}=1}^{i} M\left(\frac{n}{k+1}, r_{k+1}\right) \cdot 2^k \binom{\frac{nk}{k+1}-k}{i-r_{k+1}} \tag{4.5}$$

$$
\begin{aligned}
&= \sum_{r_{k+1}=1}^{i} 2\binom{\frac{n}{k+1}-1}{r_{k+1}-1} \cdot 2^k \binom{\frac{nk}{k+1}-k}{i-r_{k+1}} \\
&= 2^{k+1} \sum_{r_{k+1}=0}^{i-1} \binom{\frac{n}{k+1}-1}{r_{k+1}} \cdot \binom{\frac{nk}{k+1}-k}{i-r_{k+1}-1} \\
&= 2^{k+1} \binom{\frac{n}{k+1}-1+\frac{nk}{k+1}-k}{i-1} \\
&= 2^{k+1} \binom{n-(k+1)}{i-1}.
\end{aligned}
$$

We used the induction assumption to simplify (4.4) to (4.5). □

### 4.9.2 Encoding of Run-Length Limited Sequences

In this appendix we describe how to efficiently encode vectors that satisfy the $(\log(n)+3)$-RLL$(n)$ constraint. Namely, Algorithm 5 uses one redundancy bit in order to encode vectors of maximum run length at most $\lceil \log(n) \rceil + 3$.

**Algorithm 5** Run-Length Encoding

**Input:** Sequence $\mathbf{x} \in \mathbb{F}_2^n$
**Output:** Sequence $\mathbf{y} \in \mathbb{F}_2^{n+1}$ with run length $\leq \lceil \log(n) \rceil + 3$

```
Define y = (x_1, x_2, ..., x_n, 0) ∈ F_2^{n+1}
Set i = 1 and i_end = n
while i ≤ i_end do
    if length of run starting at y_i is ≥ ⌈log(n)⌉+4 then
        p(i): binary representation of i with ⌈log(n)⌉ bits
        remove ⌈log(n)⌉ + 3 bits of this run from y
        append (1, p(i), 01) on the right of y
        set i_end = i_end − log(n) − 3
    else
        set i = i + 1
    end if
end while
```

Notice that in Algorithm 5 if there is a run of length at least $a \cdot (\lceil \log(n) \rceil + 3) + 1$, for some $a \geq 2$, then the same vector $(1, p(i), 01)$ is appended $a$ times, where $p(i)$ denotes the binary representation of $i$ with $\lceil \log(n) \rceil$ bits.

**Theorem 16.** *Given any sequence* $\mathbf{x} \in \mathbb{F}_2^n$*, Algorithm 5 outputs a sequence* $\mathbf{y} \in \mathbb{F}_2^{n+1}$ *where any run has length at most* $\lceil \log(n) \rceil + 3$ *and such that* $\mathbf{x}$ *can uniquely be reconstructed given* $\mathbf{y}$*.*

*Proof.* First, let us explain the length of $\mathbf{y}$. Some runs of length $\lceil \log(n) \rceil + 3$ are removed and a block $(1, p(i), 01)$ is appended. Both blocks have length $\lceil \log(n) \rceil + 3$, so this does not change the length of the vector and we have only one additional bit, which is the zero bit that was appended in Step 1.

Second, let us consider the maximum run length. The longest run in $\mathbf{y}$ is of length $\lceil \log(n) \rceil + 3$, since any longer run is removed and replaced by $(1, p(i), 01)$. Clearly, in the newly appended blocks, the run length is at most $\lceil \log(n) \rceil + 1$ due to the "01". The first "1" in $(1, p(i), 01)$ is necessary to avoid the following case: the sequence $\mathbf{x}$ ends with $\log(n)$ zeros and there is a sequence of $2\log(n)$ zeros at the beginning. We have to write the number zero in binary to the right of the redundancy bit. This would create a sequence of $2\log(n) + 1$ zeros if the first one of $(1, p(i), 01)$ was not there.

To reconstruct $\mathbf{x}$ given $\mathbf{y}$, we start from the right. Check if the rightmost bit is 0 or 1. If it is 0, then the leftmost $n$ bits of $\mathbf{y}$ are equal to $\mathbf{x}$. If it is 1, we know that the rightmost $\lceil \log(n) \rceil + 3$ bits are an encoded block, where $p(i)$ provides the position where to insert a run of length $\lceil \log(n) \rceil + 3$. The value of this run is the value of the bit at position $i$. We can therefore insert such a run and remove the rightmost $\lceil \log(n) \rceil + 3$ bits. Then, we check again the rightmost bit. We repeat the previous strategy until the rightmost bit is 0, in which case the first $n$ bits correspond to $\mathbf{x}$ and we have decoded our original sequence. □

**Example 10.** *Let* $n = 16$ *and therefore* $\log(n) = 4$ *and* $\log(n) + 3 = 7$*. Consider the following sequence:*

$$\mathbf{x} = (0111111111111111),$$

*where the one-run has length* 15*. Let us go through the steps of Algorithm 5.*

1. $\mathbf{y} = (01111111111111110)$

2. $i = 1$ *and* $i_{end} = 16$.

3. *for* $i = 1$*: do nothing.*

4. $i = 2$*: the run starting at* $x_2$ *is at least* 8 *bits long.*

   *Define* $p(2) = (0010)$*, remove* 7 *bits from the one run in* $\mathbf{y}$ *and append* $(1001001)$.

   *Thus,* $\mathbf{y} = (01111111101001001)$.

   $i_{end} = 16 - 7 = 9$.

5. $i = 2$*: the run starting at* $x_2$ *is* 8 *bits long.*

   *Define* $p(2) = (0010)$*, remove* 7 *bits from the one run in* $\mathbf{y}$ *and append* $(1001001)$.

   *Thus,* $\mathbf{y} = (01010010011001001)$.

   $i_{end} = 9 - 7 = 2$.

6. $i = 2$*: do nothing and then the while-loop stops.*

*The decoding works as described in the proof of Theorem 16.*

### 4.9.3 Decoder of Shifted VT-Codes

In order to better understand the rationale behind the SVT-code, let us explore the details of the decoding algorithm (presented in pseudocode form in Algorithm 6).

The decoder receives the vector $\mathbf{y} = (y_1, \ldots, y_{n-1}) \in \mathbb{F}_2^{n-1}$ which is the vector $\mathbf{x}$ with a single bit deleted. The decoder knows the first possible location of the deleted bit, $u$, as well as the number of possible positions of the deleted bit, $P$. In our overall code construction, the parameter $c$, the weighted sum from Definition 1, and $P$ are both known to the decoder ahead of time, while $u$ is gleaned from decoding the first row of our codeword array. The value of the deleted bit, *DelVal*, is found by simply checking the overall parity of the received vector.

We define $\hat{\mathbf{y}} = (y_u, y_{u+1}, \ldots, y_{u+P-2})$. This vector contains the $P-1$ bits in which we are not certain about their position in $\mathbf{x}$. Any bit in position $i, i < u$ is in its proper position,

**Algorithm 6** Decoding algorithm for the $SVT_{c,d}(n, P)$ code (deletion case)

**Input:** Received vector $\mathbf{y}$, integers *c, d, u, P*
**Output:** Corrected vector $\mathbf{y}$ (equal to original vector $\mathbf{x}$)

$DelVal \leftarrow wt(\mathbf{y}) + d \pmod 2$
$\hat{\mathbf{y}} \leftarrow (y_u, y_{u+1}, \ldots, y_{u+P-2})$
$c' \leftarrow \sum_{i=1}^{u+P-2} iy_i + \sum_{i=u+P-1}^{n-1} (i+1)y_i \pmod P$
$\Delta \leftarrow c - c' \pmod P$
$k \leftarrow 1$
**if** $DelVal = 0$ **then**
    **while** $R_1(k) \neq \Delta$ **do**
        $k \leftarrow k + 1$
    **end while**
**else**
    **while** $L_0(k) \neq \Delta - u - wt(\hat{\mathbf{y}}) \pmod P$ **do**
        $k \leftarrow k + 1$
    **end while**
**end if**
$DelPos \leftarrow k$
Insert $DelVal$ into position $DelPos$ of $\hat{\mathbf{y}}$

and any bit in position $i, i > u + P - 2$ will be shifted one position to the right once we insert the deleted bit.

In the decoding algorithm, $c'$ is the *augmented* weighted sum of our received vector $\mathbf{y}$. We define the difference between the original weighted sum of $\mathbf{x}$ and our augmented weighted sum of $\mathbf{y}$ as $\Delta$. Since our calculation of $c'$ properly weights every bit outside of $\hat{\mathbf{y}}$, we can focus our attention solely on $\hat{\mathbf{y}}$, i.e., inserting a bit to increase the weighted sum of $\hat{\mathbf{y}}$ by $\Delta$ also increases the weighted sum of $\mathbf{y}$ by $\Delta$ (thus yielding $\mathbf{x}$).

If we insert the missing bit into position $k$ in $\hat{\mathbf{y}}$, then we denote the number of 0's and 1's to the left of the bit we insert as $L_0(k)$ and $L_1(k)$, respectively. Similarly, let us call the number of 0's and 1's to the right of the bit we insert as $R_0(k)$ and $R_1(k)$.

Inserting a 0 into $\hat{\mathbf{y}}$ increases its weighted sum by $R_1(k) \pmod P$ since all the 1's are shifted one space to the right. Note that this is true even if the 1 is pushed from weight $P - 1$ to weight $P \pmod P = 0$. Thus, if a 0 was deleted, we insert a 0 into position $k$ such that $R_1(k) = \Delta$.

Inserting a 1 into the $k$th position of $\hat{\mathbf{y}}$ increases its weighted sum by $R_1(k) + k + u - 1 \pmod{P}$. Since $k = L_0(k) + L_1(k) + 1$, this implies $\Delta = R_1(k) + L_1(k) + L_0(k) + u \bmod P$. Since $wt(\hat{\mathbf{y}}) = L_1(k) + R_1(k)$, we have $\Delta = L_0(k) + wt(\hat{\mathbf{y}}) + u \pmod{P}$. Solving for $L_0(k)$ yields $L_0(k) = \Delta - u - wt(\hat{\mathbf{y}}) \pmod{P}$. Thus, if the deleted bit was a 1, we insert a 1 into position $k$ such that $L_0(k) = \Delta - u - wt(\hat{\mathbf{y}}) \pmod{P}$.

**Example 11.** *Let us assume the transmitted vector was the following* $SVT_{0,0}(16, 5)$ *codeword:* $\boldsymbol{x} = (1111011\boldsymbol{0}01100011)$. *Based on previous information, the decoder knows* $u = 8$. *During transmission, the 9th bit was deleted (bolded), so the received vector was* $\boldsymbol{y} = (111101101100011)$. *The receiver determines the value of the deleted bit:*

$$DelVal = wt(\boldsymbol{y}) + d \pmod{2} = 10 + 0 \pmod{2} = 0.$$

*The receiver calculates the augmented weighted sum of the received vector* $c' = 3$. *Now the receiver calculates the differences in the weighted sums:*

$$\Delta = c - c' \pmod{5} = 0 - 3 \pmod{5} = 2.$$

*Since* $u = 8$, *we have* $\hat{\boldsymbol{y}} = (0110)$, *underlined in* $\boldsymbol{y}$. *Since* $DelVal = 0$, $DelPos$ *is the position in which there are* $\Delta = 2$ *1's to the right of it within* $\hat{y}$, *yielding* $\hat{\boldsymbol{y}} = (0\boldsymbol{0}110)$. *With the insertion of this bit, we have successfully decoded the original sent codeword* $\boldsymbol{x}$.

Using similar logic from the deletion case, we present Algorithm 7, the decoding algorithm for the $SVT_{c,d}(n, P)$ code in the case of an insertion.

**Algorithm 7** Decoding algorithm for the $SVT_{c,d}(n, P)$ code (insertion case)

**Input:** Received vector $\mathbf{y}$, integers *c, d, u, P*

**Output:** Corrected vector $\mathbf{y}$ (equal to original vector $\mathbf{x}$)

$InsVal \leftarrow wt(\mathbf{y}) + d \pmod 2$
$\hat{\mathbf{y}} \leftarrow (y_u, y_{u+1}, \ldots, y_{u+P-1})$
$a' \leftarrow \sum_{i=1}^{u+P-1} iy_i + \sum_{i=u+P}^{n+1} (i-1)y_i \pmod P$
$\Delta \leftarrow a' - a \pmod P$
**if** $InsVal = 0$ **then**
    **while** $R_1(k) \neq \Delta$ **do**
        $k \leftarrow k + 1$
    **end while**
**else**
    **while** $L_0(k) \neq \Delta - u - wt(\hat{\mathbf{y}}) + 1 \pmod P$ **do**
        $k \leftarrow k + 1$
    **end while**
**end if**
$InsPos \leftarrow k$
Delete the bit $(InsVal)$ from position $InsPos$ of $\hat{\mathbf{y}}$

# CHAPTER 5

# Conclusion

## 5.1 Summary of Our Results

This dissertation dealt with the creation of ECCs to mitigate errors in current and future large-scale data systems. Due to the exponential growth of data being stored, processed, and transmitted, ECCs are likely to play a crucial role in data robustness.

In the first part of the dissertation, we focused on how to best exploit side-information from the underlying data. By establishing the fundamental coding principles, matched to real-life systems, we created a practical form of list-decoding. We demonstrated that the entropy within a cacheline contains sufficient information to successfully decode from a significant portion of the detected errors. Furthermore, using an entropy based policy reveals how likely we are to successfully recover, thus allowing us to roll-back instead of attempting a low-percentage decoding operation. Additionally, we presented an ultra-lightweight version of the SDECC framework that is more suitable for IoT devices.

While still exploiting side-information, we switched gears away from heuristic recovery in favor of a priori special message designations. Our broad class of UMP codes offer direct alternatives to commonly used codes in computer memory systems without any additional redundancy. Once again, we combine new coding theoretic results with the limits and requirements of real-world systems to produce practical enhancements to state-of-the-art error correction techniques.

Lastly, we tackled the problem of correcting burst deletions. Deletions and insertions are notoriously more difficult to correct than the usual substitution errors. In addition to establishing theoretical results, we explicitly constructed the best-known codes for correcting a burst of deletions. We hope that our extension of these codes to the non-binary regime will be useful in making DNA storage a reality.

## 5.2 Future Directions

Directions for the future of SDECC include adaptive software and ECC support for memory fault models and development of software mechanisms that can eventually verify the correctness of SDECC recovery. Being able to exploit the fault model itself, in addition to the underlying data, can only increase our recovery results.

Our class of UMP codes are formulated to allow the mapping of special messages as the system designer chooses. This is a simple task for basic mapping patterns, but it can become practically infeasible for very complex mappings. A future direction for our UMP codes is to expand the user interface to allow for more complex mappings to be done inside the black box.

Lastly, state-of-the-art DNA storage techniques and methods are constantly evolving. The future direction is to combine our burst deletion correcting code with the latest coding scheme that enables the data stored in DNA to be byte-addressable.

In this dissertation, we established mathematical principles and ECCs that will be useful in mitigating errors in future data systems; however, the technology of tomorrow always contains unforeseeable developments and innovations. For this reason, a coding theorist's work is never done, and must continue to adapt to the current landscape. With continual mathematical innovations, humanity is well-positioned to flourish in the age of information.

# References


[1] C. E. Shannnon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

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

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[4] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[5] D. J. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IET Electronics Letters*, vol. 32, no. 18, p. 1645, 1996.

[6] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Monterey, California, Dec. 2000, pp. 258–265.

[7] A. Alameldeen and D. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," University of Wisconsin, Madison, Tech. Rep., 2004.

[8] S. Mittal and J. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1524–1536, May 2015.

[9] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0," 2014. [Online]. Available: https://riscv.org

[10] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined ECC: Heuristic recovery from uncorrectable memory errors," University of California, Los Angeles, Tech. Rep., Oct. 2017.

[11] M. Gottscho, "Opportunistic memory systems in presence of hardware variability," Ph.D. dissertation, University of California, Los Angeles, Jun. 2017.

[12] I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Parity++: Lightweight error correction for last level caches," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg City, Luxembourg, Jun. 2018.

[13] A. A. Kulkarni and N. Kiyavash, "Nonasymptotic upper bounds for deletion correcting codes," *IEEE Transactions on Information Theory*, vol. 59, no. 8, pp. 5115–5130, Aug. 2013.

[14] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the impact of memory errors on application performance," *IEEE Computer Architecture Letters*, Aug. 2016.

[15] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: localized and tiered reliability mechanisms for commodity memory systems," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 285–296, Jun. 2012.

[16] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-power, low-storage-overhead Chipkill Correct via multi-line error correction," in *Proc. IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, Nov. 2013, p. 24.

[17] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High performance, energy efficient Chipkill Correct memory with multidimensional parity," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 39–42, Jul. 2013.

[18] X. Jian, V. Sridharan, and R. Kumar, "Parity Helix: Efficient protection for single-dimensional faults in multi-dimensional memory systems," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, Mar. 2016, pp. 555–567.

[19] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[20] A. A. Davydov and L. M. Tombak, "An alternative to the Hamming code in the class of SEC-DED codes in semiconductor memory," *IEEE Transactions on Information Theory*, vol. 37, no. 3, pp. 897–902, May 1991.

[21] S. Kaneda and E. Fujiwara, "Single byte error correcting – double byte error detecting codes for memory systems," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 596–602, Jul. 1982.

[22] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, Mar. 1960.

[23] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, vol. 11, Nov. 1997.

[24] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate compression: Practical data compression for on-chip caches," in *Proc. ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, Sep. 2012.

[25] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Davis, CA, Dec. 2013, pp. 172–184.

[26] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proc. ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 329–340.

[27] Q. Nguyen, "RISC-V Tools (GNU Toolchain, ISA Simulator, Tests) – git commit 816a252." [Online]. Available: https://github.com/riscv/riscv-tools

[28] A. Waterman, "RISC-V Proxy Kernel – git commit 85ae17a." [Online]. Available: https://github.com/riscv/riscv-pk/commit/85ae17a

[29] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator – git commit 3bfc00e." [Online]. Available: https://github.com/riscv/riscv-isa-sim

[30] J. K. Wolf and B. Elspas, "Error-locating codes – a new concept in error control," *IEEE Transactions on Information Theory*, vol. 9, no. 2, pp. 113–117, Apr. 1963.

[31] J. K. Wolf, "On an extended class of error-locating codes," *Information and Control*, vol. 8, no. 2, pp. 163–169, Apr. 1965.

[32] E. Fujiwara and M. Kitakami, "A class of error locating codes for byte-organized memory systems," in *Proc. IEEE International Symposium on Fault-Tolerant Computing*, Toulouse, France, Jun. 1993, pp. 110–119.

[33] J. Song, G. Bloom, and G. Palmer, "SuperGlue: IDL-based, system-level fault tolerance for embedded systems," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, Jun.-Jul. 2016, pp. 227–238.

[34] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 31, no. 9, Cambridge, MA, Oct. 1996, pp. 138–147.

[35] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 46, no. 6, San Jose, CA, Jun. 2011, pp. 164–174.

[36] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," in *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 47, no. 4, Newport Beach, CA, Mar. 2011, pp. 213–224.

[37] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "VaMV: Variability-aware memory virtualization," in *Proc. IEEE Design, Automation, and Test in Europe (DATE)*, Dresden, Germany, Mar. 2012, pp. 284–287.

[38] M. Gottscho, L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "ViPZonE: Hardware power variability-aware memory management for energy savings," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1483–1496, 2015.

[39] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.

[40] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined error-correcting codes," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Dresden, Germany, Jun.-Jul. 2016, pp. 276–282.

[41] M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Low-cost memory fault tolerance for IoT devices," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, p. 128, 2017.

[42] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *Proc. ACM SIGMETRICS*, vol. 37, no. 1, Seattle, Washington, Jun. 2009, pp. 193–204.

[43] B. Masnick and J. K. Wolf, "On linear unequal error protection codes," *IEEE Transactions on Information Theory*, vol. 13, no. 4, pp. 600–607, Oct. 1967.

[44] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, "Modeling the implications of DRAM failures and protection techniques on datacenter TCO," in *Proc. ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, Dec. 2015, pp. 572–584.

[45] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S.-L. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka *et al.*, "The 65-nm 16-MB shared on-die L3 cache for the dual-core Intel Xeon processor 7100 series," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 4, pp. 846–852, Mar. 2007.

[46] I. Boyarinov and G. Katsman, "Linear unequal error protection codes," *IEEE Transactions on Information Theory*, vol. 27, no. 2, pp. 168–175, Mar. 1981.

[47] N. Abramson, "A class of systematic codes for non-independent errors," *IRE Transactions on Information Theory*, vol. 5, no. 4, pp. 150–157, Dec. 1959.

[48] P. Reviriego, J. Martínez, S. Pontarelli, and J. A. Maestro, "A method to design SEC-DED-DAEC codes with optimized decoding," *IEEE Transactions on Device and Materials Reliability*, vol. 14, no. 3, pp. 884–889, Sep. 2014.

[49] S. Kaneda and E. Fujiwara, "Single byte error correcting double byte error detecting codes for memory systems," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 596–602, Jul. 1982.

[50] S. Borade, B. Nakiboglu, and L. Zheng, "Unequal error protection: An information-theoretic perspective," *IEEE Transactions on Information Theory*, vol. 55, no. 12, pp. 5511–5539, Dec. 2009.

[51] Y. Y. Shkel, V. Y. Tan, and S. C. Draper, "Unequal message protection: Asymptotic and non-asymptotic tradeoffs," *IEEE Transactions on Information Theory*, vol. 61, no. 10, pp. 5396–5416, Oct. 2015.

[52] Y. Polyanskiy, H. V. Poor, and S. Verdu, "Channel coding rate in the finite blocklength regime," *IEEE Transactions on Information Theory*, vol. 56, no. 5, pp. 2307–2359, May 2010.

[53] Y. Wang, M. Qin, K. R. Narayanan, A. A. Jiang, and Z. Bandic, "Joint source-channel decoding of polar codes for language-based sources," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, Washington D.C., Dec. 2016, pp. 1–6.

[54] Y. Wang, K. R. Narayanan, and A. A. Jiang, "Exploiting source redundancy to improve the rate of polar codes," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Aachen, Germany, Jun. 2017, pp. 864–868.

[55] P. Upadhyaya and A. A. Jiang, "LDPC decoding with natural redundancy," in *Proc. Non-Volatile Memory Workshop (NVMW)*, San Diego, CA, Mar. 2017.

[56] I. Csiszár, "Joint source-channel error exponent." *Problems of Control and Information Theory*, vol. 9, no. 5, pp. 315–328, 1980.

[57] Y. Y. Shkel, V. Y. Tan, and S. C. Draper, "Second-order coding rate for m-class source-channel codes," in *Proc. Allerton Conference on Communication, Control, and Computing*, Monticello, IL, Sep.-Oct. 2015, pp. 620–626.

[58] B. Nazer, Y. Y. Shkel, and S. C. Draper, "The AWGN red alert problem," *IEEE Transactions on Information Theory*, vol. 59, no. 4, pp. 2188–2200, Apr. 2013.

[59] P. Delsarte, "An algebraic approach to the association schemes of coding theory," Ph.D. dissertation, Université Catholique de Louvain, Jun. 1973.

[60] D. P. Bertsekas, *Nonlinear programming*. Athena Scientific, 1999.

[61] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0," DTIC Document, Tech. Rep., 2014.

[62] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," *IEEE Transactions on Information Theory*, (submitted) 2018.

[63] ——, "Context-aware resiliency: Unequal message protection for random-access memories," in *Proc. IEEE Information Theory Workshop (ITW)*, Kaohsiung, Taiwan, Nov. 2017, pp. 166–170.

[64] G. M. Church, Y. Gao, and S. Kosuri, "Next-generation digital information storage in DNA," *Science*, p. 1226355, Aug. 2012.

[65] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney, "Towards practical, high-capacity, low-maintenance information storage in synthesized DNA," *Nature*, vol. 494, no. 7435, pp. 77–80, Feb. 2013.

[66] Y. Erlich and D. Zielinski, "DNA Fountain enables a robust and efficient storage architecture," *Science*, vol. 355, no. 6328, pp. 950–954, Mar. 2017.

[67] L. Orlando, A. Ginolhac, G. Zhang, D. Froese, A. Albrechtsen, M. Stiller, M. Schubert, E. Cappellini, B. Petersen, I. Moltke *et al.*, "Recalibrating equus evolution using the genome sequence of an early middle pleistocene horse," *Nature*, vol. 499, no. 7456, pp. 74–78, 2013.

[68] S. H. T. Yazdi, H. M. Kiah, E. Garcia-Ruiz, J. Ma, H. Zhao, and O. Milenkovic, "DNA-based storage: Trends and methods," *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, vol. 1, no. 3, pp. 230–248, Sep. 2015.

[69] R. Heckel, I. Shomorony, K. Ramchandran, and N. David, "Fundamental limits of DNA storage systems," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Aachen, Germany, Jun. 2017, pp. 3130–3134.

[70] S. H. T. Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic, "A rewritable, random-access DNA-based storage system," *Scientific Reports*, vol. 5, p. 14138, Sep. 2015.

[71] Z. Chang, J. Chrisnata, M. F. Ezerman, and H. M. Kiah, "Rates of DNA sequence profiles for practical values of read lengths," *IEEE Transactions on Information Theory*, vol. 63, no. 11, pp. 7166–7177, Nov. 2017.

[72] H. M. Kiah, G. J. Puleo, and O. Milenkovic, "Codes for DNA sequence profiles," *IEEE Transactions on Information Theory*, vol. 62, no. 6, pp. 3125–3146, Jun. 2016.

[73] R. Gabrys, H. M. Kiah, and O. Milenkovic, "Asymmetric Lee distance codes for DNA-based storage," *IEEE Transactions on Information Theory*, vol. 63, no. 8, pp. 4982–4995, May 2017.

[74] S. Jain, F. Farnoud, M. Schwartz, and J. Bruck, "Duplication-correcting codes for data storage in the DNA of living organisms," *IEEE Transactions on Information Theory*, vol. 63, no. 8, pp. 4996–5010, Mar. 2017.

[75] M. Levy and E. Yaakobi, "Mutually uncorrelated codes for DNA storage," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Aachen, Germany, Jun. 2017, pp. 3115–3119.

[76] L. Dolecek and V. Anantharam, "Repetition error correcting sets: Explicit constructions and prefixing methods," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 4, pp. 2120–2146, Jan. 2010.

[77] C. R. O'Donnell, H. Wang, and W. B. Dunbar, "Error analysis of idealized nanopore sequencing," *Electrophoresis*, vol. 34, no. 15, pp. 2137–2144, Aug. 2013.

[78] F. Dandashi, A. Griggs, J. Higginson, J. Hughes, W. Narvaez, M. Sabbouh, S. Semy, and B. Yost, "Tactical edge characterization framework," *MITRE Technical Report MTR070331*, 2007.

[79] J. Jeong and C. T. Ee, "Forward error correction in sensor networks," *University of California at Berkeley*, pp. 1–13, 2003.

[80] G. M. Tenengolts, "Nonbinary codes, correcting single deletion or insertion (corresp.)," *IEEE Transactions on Information Theory*, vol. 30, no. 5, pp. 766–769, 1984.

[81] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors (in Russian)," *Automatika i Telemkhanika*, vol. 161, no. 3, pp. 288–292, 1965.

[82] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals (in Russian)," *Doklady Akademii Nauk SSR*, vol. 163, no. 4, pp. 845–848, 1965.

[83] V. I. Levenshtein, "Asymptotically optimum binary code with correction for losses of one or two adjacent bits," *Systems Theory Research (translated from Problemy Kibernetiki)*, vol. 19, pp. 293–298, 1967.

[84] L. Cheng, T. G. Swart, H. C. Ferreira, and K. A. S. Abdel-Ghaffar, "Codes for correcting three or more adjacent deletions or insertions," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Honolulu, HI, Jun. 2014, pp. 1246–1250.

[85] N. J. A. Sloane, "On single-deletion-correcting codes," in *Codes and designs*, vol. 10, Columbus, OH, May 2000, pp. 273–291.

[86] P. A. H. Bours, "Codes for correcting insertions and deletion errors," Ph.D. dissertation, Eindhoven University of Technology, Jun. 1994.

[87] A. S. J. Helberg, "Coding for the correction of synchronization errors," Ph.D. dissertation, Rand Afrikaans University, 1993.

[88] A. S. J. Helberg and H. C. Ferreira, "On multiple insertion/deletion correcting codes," *IEEE Transactions on Information Theory*, vol. 48, no. 1, pp. 305–308, Jan. 2002.

[89] K. A. S. Abdel-Ghaffar, F. Paluncic, H. C. Ferreira, and W. A. Clarke, "On Helberg's generalization of the Levenshtein code for multiple deletion/insertion error correction," *IEEE Transactions on Information Theory*, vol. 58, no. 3, pp. 1804–1808, Mar. 2012.

[90] I. Landjev and K. Haralambiev, "On multiple deletion codes," *Serdica Journal of Computing*, vol. 1, no. 1, pp. 13–26, 2007.

[91] F. Paluncic, K. A. S. Abdel-Ghaffar, H. C. Ferreira, and W. A. Clarke, "A multiple insertion/deletion correcting code for run-length limited sequences," *IEEE Transactions on Information Theory*, vol. 58, no. 3, pp. 1809–1824, Mar. 2012.

[92] J. Brakensiek, V. Guruswami, and S. Zbarsky, "Efficient low-redundancy codes for correcting multiple deletions," in *Proc ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Arlington, VA, Jan. 2016, pp. 1884–1892.

[93] K. A. S. Immink, *Coding Techniques for Digital Recorders*. Prentice Hall, College Div., 1991.

[94] M. F. Schilling, "The surprising predictability of long runs," *Mathematics Magazine*, vol. 85, no. 2, pp. 141–149, Apr. 2012.

[95] A. Rényi, *Probability Theory*. Budapest, Akad. Kiadó, 1970.

[96] D. Cullina, A. A. Kulkarni, and N. Kiyavash, "A coloring approach to constructing deletion correcting codes from constant weight subgraphs," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Cambridge, MA, Jul. 2012, pp. 513–517.

[97] L. Cheng, "Coding techniques for insertion/deletion error correction," Ph.D. dissertation, University of Johannesburg, Jun. 2012.

[98] C. Schoeny, A. Wachter-Zeh, R. Gabrys, and E. Yaakobi, "Codes correcting a burst of deletions or insertions," *IEEE Transactions on Information Theory*, vol. 63, no. 4, pp. 1971–1985, Apr. 2017.

[99] C. Schoeny, F. Sala, and L. Dolecek, "Novel combinatorial coding results for DNA sequencing and data storage," in *Proc. Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Nov. 2017, pp. 511–515.