Compressie en interoperabele representatie van genomische informatie

Compression and Interoperable Representation of Genomic Information

Tom Paridaens

Promotoren: prof. dr. P. Lambert, prof. dr. W. De Neve
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2017 - 2018

ISBN 978-94-6355-133-5
NUR 965
Wettelijk depot: D/2018/10.500/51

Examination board

Chair

prof. dr. ir. Filip De Turck *Ghent University, Belgium*

Secretary

dr. ir. Glenn Van Wallendael *Ghent University, Belgium*

Reading Committee

prof. dr. Jaime Delgado *Universitat Politcnica de Catalunya, Spain*
dr. Hannes Pousseele *Applied Maths NV, Belgium*
prof. dr. Yvan Saeys *Ghent University, Belgium*
prof. dr. ir. Jan Fostier *Ghent University, Belgium*

Promotors

prof. dr. ir. Peter Lambert *Ghent University, Belgium*
prof. dr. Wesley De Neve *Ghent University, Belgium & GUGC, Korea*

Acknowledgements

The past five years were a rollercoaster, and not the calm and easy wooden type. But as with any rollercoaster, the fun is at the end and once you're out, you want in again :-). And all this work started with Wesley, Erik and Peter, who offered me the chance to return to MultiMediaLab in 2013. Wesley had a great vision: we should use or multimedia compression and representation knowledge to genomic data. And I was stubborn enough to follow that vision and go for it, resulting in a very beautiful diamond at the end: the MPEG-G standard.

All this would have never been possible without my wife Kaat. She was there all the time. She had to handle all those weeks that I was physically away for conferences and MPEG meetings. During these weeks she had to take care of her work, the kids, the family, and, if some time was left, herself. And on top of that she had to handle far more weeks during which I was physically there but mentally away: The weeks and weekends before MPEG contribution deadlines, before journal paper submissions, the months that I was writing this dissertation, the weeks before my internal defense and even the weeks after that. Luckily, my parents and my parents-in-law were there to help her if things really got to much.

I would like to express a special thanks to Wesley, Glenn, and Peter for supporting me and for reviewing my papers and this dissertation and keeping me motivated and on the right track.

I would like to thank the members of my examination board for their time, effort, and feedback: Prof. Jaime Delgado, Dr. Hannes Pousseele, Prof. Yvan Saeys, Prof. Jan Fostier, Dr. Glenn Van Wallendael, and Prof. Filip De Turck.

Thank you Claudio, Marco, Jan, Mikel, Daniele, Giorgio, Jaime, Daniel, Leonardo, and all the others who made the MPEG experience so nice and fruitful. I hope we can continue working on this diamond together for a long time.

Furthermore, I would like to thank everybody at Multimedialab/DSLab/IDLab who helped me with smaller and/or bigger things, with whom I had fun conversations at the coffee machine, in the hallway, or during the noon break: Niels, Glenn, Ignace, Johan, Ruben, Baptist, Aza, Vasileios, Hannes, Olivier, Laura, Kristof, Martin, Florian, Jasper, Thijs, Dieter, Jonas, Steven, Anastasia, Miel, Jan, Chris, Ellen, Jürgen, Saar, Sarah, Stijn, Stefaan I en II, Bart, Davy I and II, Wim, Dieter, Yves, Sam, Gaëtan, Jozef, Sebastiaan and all the others (it is a really long list). Yes, you all made being at work a nice experience.

I would like to thank my children, Jore and Nelle, for the magnificent experiences and feelings, my parents, my grandparents (dankjewel meme & pepe, meter Denise), my parents-in-law (who are a lot nicer than the connotation of the name implies ;-), oma & opa, and my sister. Thank you Wouter for the many evening talks in the pub and during sports, thank you Lissa for the crowd surfing bet, and thanks to all of my friends and my godchildren Indra & Wolf for putting a smile on my face during the tougher days.
Finally, I would like to thank two people who gave me a huge boost (probably without knowing it). Thank you Gary Sullivan for talking to me on my research at the DCC conference. It was extremely inspiring and a huge honour. Thank you Eddy De Clercq, you convinced me, during my third-year mental crash, to continue and to finish my PhD. It worked :-).

Gent, 2018
Tom Paridaens

"Aan allen die ik vermeld heb, dank u.
Aan allen die ik niet vermeld heb... ook dank u."
Walter D. Donder (vrije vertaling)

Table of Contents

Acknowledgements **i**

Nederlandstalige samenvatting **xix**

English summary **xxiii**

1 Introduction **1-1**
1.1 Introduction 1-1
1.1.1 Deoxyribo Nucleic Acid (DNA) 1-2
1.1.2 DNA Sequencing & Assembly 1-3
1.1.2.1 Sequencing & Read Assembly Workflow 1-3
1.1.2.2 Comparing Sequencing Technologies 1-5
1.1.2.3 The History of DNA Sequencing 1-6
1.1.3 Sequencing and Mapping Data 1-11
1.1.3.1 Data generated during the Sequencing Process . 1-11
1.1.3.2 Data generated during the Read Assembly Process 1-11
1.1.4 Storage of Genomic Data 1-12
1.1.4.1 File Size 1-15
1.1.4.2 Storage Cost 1-15
1.1.5 Compression of Genomic Data 1-18
1.1.5.1 The Three Dimensions of Compression 1-18
1.1.5.2 Compression of Nucleotidic Data 1-19
1.1.5.3 Compression of Quality Scores 1-20
1.1.5.4 Generic Compression 1-21
1.1.6 Exchange of Genomic Data 1-22
1.1.7 Standardisation 1-23
1.1.8 Research & Standardisation Timeline 1-25
1.2 Outline 1-26
1.3 Publications 1-27
1.3.1 Publications in International Journals 1-27
1.3.1.1 Author 1-27
1.3.1.2 Co-author 1-27
1.3.2 Publications in International Conferences 1-28
1.3.2.1 Author 1-28
1.3.2.2 Co-author 1-28

1.3.3 MPEG Input Contributions 1-29
1.3.3.1 Author 1-29
1.3.3.2 Co-author 1-30
1.3.4 MPEG Standardization Documents 1-30
References 1-31

2 Coding Framework **2-1**
2.1 Features 2-1
2.2 Workflow 2-2
2.3 Framework Flexibility 2-4
2.4 Block Structure 2-5
2.5 Context-Adaptive Binary Arithmetic Coding 2-6
2.5.1 Binarization 2-9
2.5.2 Context Selection 2-9
2.5.3 Random Access 2-10
2.6 Conclusions and Original Contributions 2-10
References 2-12

3 AFRESh **3-1**
3.1 Introduction 3-1
3.2 Related Work 3-2
3.3 Framework Extensions 3-3
3.3.1 Alphabets 3-3
3.3.2 Prediction and Encoding Tools 3-4
3.3.2.1 Prediction Tools 3-5
3.3.2.2 Encoding Tools 3-5
3.3.2.3 Removed Coding Tools 3-6
3.4 Optimization Methodology 3-6
3.4.1 Binarization and Context Modeling of Alphabet Indicators 3-7
3.4.2 Binarization and Context Modeling of Residue 3-8
3.4.3 Binarization and Context Modeling of Predictor Indicators 3-9
3.5 Random Access 3-10
3.6 Experimental Results 3-11
3.6.1 Experimental Setup 3-11
3.6.2 Reads 3-13
3.6.3 Assembled Sequences 3-15
3.6.4 Tool Selection 3-18
3.7 Support for new Sequencing Technologies 3-20
3.8 Conclusions and Original Contributions 3-21
References 3-22

4 AQUa **4-1**
4.1 Introduction 4-1
4.2 Related Work 4-1
4.3 Coding Tools 4-3
4.3.1 DFC - Difference Coder 4-3
4.3.2 ADFC - Average Difference Coder 4-4
4.3.3 CVP - Convolutional Predictor 4-4
4.3.4 SRP - Single Repeat Predictor 4-5
4.3.5 AVP - Average Predictor 4-5
4.3.6 NSP - Normal Search Predictor 4-6
4.3.7 HNSP - Hierarchical Normal Search Predictor 4-6
4.3.8 Removed Coding Tools 4-6
4.4 Binarization and Context Modeling 4-7
4.4.1 Value Representations 4-7
4.4.2 Binarization and Context Modeling of Residue 4-9
4.4.3 Binarization and Context Modeling of CVP Mode 4-10
4.4.4 Binarization and Context Modeling of NSP and HNSP Pointers 4-11
4.4.5 Binarization and Context Modeling of Coding Tool Identification 4-11
4.5 Random Access 4-14
4.6 Experimental Results 4-16
4.6.1 Experimental Setup 4-16
4.6.2 Window Size 4-16
4.6.3 Compression Results 4-17
4.7 Tool Selection 4-20
4.8 Support for new Sequencing Technologies 4-21
4.9 Conclusions and Original Contributions 4-25
References 4-26

5 Standardization: MPEG-G **5-1**
5.1 Introduction 5-1
5.2 MPEG-G 5-3
5.2.1 Descriptor Streams 5-4
5.2.2 Data Classes 5-6
5.3 Random Access within MPEG-G 5-7
5.4 Encryption, Privacy & Integrity 5-7
5.5 Proposed Coding Solution 5-8
5.5.1 Input Data Parsing 5-8
5.5.2 Value Transformation 5-9
5.5.3 Value Binarization 5-11
5.5.4 Context Selection 5-14
5.5.4.1 Context Sets 5-14
5.5.4.2 Context Set Selection 5-15
5.5.4.3 CABAC Encoding 5-15

5.5.5 Example: *RCOMP 5-16
5.6 Experimental Results 5-19
5.6.1 Experimental Setup 5-19
5.6.2 Compression Results 5-20
5.6.2.1 Compression Results per Encoding Mode . . . 5-21
5.6.2.2 Compression Results per Descriptor Stream Type 5-22
5.6.2.3 Compression Results per Test File 5-24
5.6.3 Memory Usage 5-35
5.7 Proposed Decoding Syntax 5-37
5.7.1 Encoding Parameters Signaling Syntax 5-37
5.7.1.1 Syntax Semantics 5-37
5.7.2 Lookup Table Signaling Syntax 5-39
5.7.2.1 Syntax Definitions 5-39
5.8 Support for new Sequencing Technologies 5-41
5.9 Conclusions and Original Contributions 5-42
References 5-43

6 Overall Conclusion **6-1**
6.1 Summary 6-1
6.1.1 Coding Framework 6-2
6.1.2 AFRESh and AQUA 6-2
6.1.2.1 AFRESh 6-2
6.1.2.2 AQUa 6-3
6.1.3 MPEG-G Standardization 6-4
6.2 Contributions 6-5
6.3 Future Work 6-7
6.3.1 Data Compression and Representation 6-7
6.3.2 MPEG-G Applications 6-8
6.3.2.1 Encoders, Decoders, & Data Management . . . 6-8
6.3.2.2 Sequencing & Analysis Chains 6-10
6.3.3 The Future of Sequencing 6-11

List of Figures

1.1 The double helix molecular structure of DNA as discovered by Watson and Crick. . . . 1-2
1.2 Example of two complementary DNA strands. . . . 1-3
1.3 A modern (simplified) sequencing and assembly workflow. . . . 1-4
1.4 Output of the Sanger sequencing of a DNA strand. . . . 1-7
1.5 Output of a single-reaction Sanger sequencing process, using fluorescent colour labels. . . . 1-8
1.6 The Illumina NovaSeq 6000 DNA sequencer. . . . 1-9
1.7 The Oxford Nanopore MinION, a thumb drive sequencer. . . . 1-10
1.8 Examples of mapping corrections: Insertion, Deletion, and Single Nucleotide Polymorphism. . . . 1-11
1.9 Example of two reads in a FASTA data file, using the current Illumina read identifier syntax. The text in bold is for clarity, this text is not part of the actual file. . . . 1-13
1.10 Example of two reads in a FASTQ data file, using the NCBI Sequence Read Archive read identifier syntax. The text in bold is for clarity, this text is not part of the actual file. . . . 1-13
1.11 Example of a SAM file containing three reads. . . . 1-14
1.12 Evolution of sequencing and storage costs (2002-2017). Generated from data provided by [19], [20], and [21]. . . . 1-16
1.13 Evolution of sequencing cost and size of the NCBI WGS database (2002-2017). Generated from data provided by [19], [20], and [21]. 1-16
1.14 Evolution of sequencing cost and the storage cost of the NCBI WGS database (2002-2017). Generated from data provided by [19], [20], and [21]. . . . 1-17

2.1 The different steps used by the coding framework. . . . 2-3
2.2 Data structure of a block. . . . 2-5
2.3 Encoding the binary sequence 001 using Binary Arithmetic Coding. 2-6
2.4 Examples of different contexts. . . . 2-8

3.1 Example of two reads in a FASTA data file. . . . 3-2
3.2 Example of two reads in a FASTQ data file. . . . 3-2
3.3 Coding toolset currently available in AFRESh. . . . 3-4

3.4 Example of a double repeat prediction with prediction error correction. . . . 3-8
3.5 Visualization of a residue error correction with a diagonal orientation. . . . 3-8
3.6 Boxplot of the effect of the block size on the resulting compression rate over the different chromosomes of the human genome (in bits per base). . . . 3-16
3.7 Loss in compression effectiveness, compared to the complete toolset. 3-19
3.8 Total encoding time, compared to the complete toolset. . . . 3-19

4.1 Example of two reads in a FASTQ data file. . . . 4-2
4.2 Minimum/average/maximum occurrence of the DFC coding tool residue values. . . . 4-10
4.3 Usage of the different CVP modes (min/median/max), sorted by decreasing usage. . . . 4-11
4.4 Coding tool usage for test file 02. . . . 4-12
4.5 Coding tool usage for test file 05. . . . 4-12
4.6 Coding tool usage for test file 10. . . . 4-13
4.7 Coding tool usage for test file 23. . . . 4-13
4.8 Overhead of smaller random access sizes versus the largest random access size (window size 16). . . . 4-14
4.9 Total compressed size, compared to a window size of one read. . . . 4-17
4.10 Loss in compression effectiveness, compared to the complete toolset. 4-23
4.11 Total encoding time, compared to the complete toolset. . . . 4-24

5.1 The different steps used by the proposed coding solution for MPEG-G. . . . 5-9
5.2 Uncompressed file size per test file. . . . 5-20
5.3 Uncompressed file size per descriptor stream type. . . . 5-21
5.4 Output file size for the complete benchmarking set per encoding mode, compared to 7-Zip. . . . 5-22
5.5 Total encoding time for the complete benchmarking set per encoding mode, compared to 7-Zip. . . . 5-22
5.6 Total decoding time for the complete benchmarking set per encoding mode, compared to 7-Zip. . . . 5-23
5.7 Compression gain by sorting UREADS descriptor streams per test set. . . . 5-24
5.8 Output file size for the complete benchmarking set per descriptor stream type, compared to 7-Zip. . . . 5-25
5.9 Total encoding time for the complete benchmarking set per descriptor stream type, compared to 7-Zip. . . . 5-26
5.10 Encoding speed for the complete benchmarking set per descriptor stream type, compared to 7-Zip. . . . 5-27
5.11 Total decoding time for the complete benchmarking set per descriptor stream type, compared to 7-Zip. . . . 5-28

5.12 Decoding speed for the complete benchmarking set per descriptor stream type, compared to 7-Zip. 5-29
5.13 Output file size for the complete benchmarking set per test file, compared to 7-Zip. 5-30
5.14 Total encoding time for the complete benchmarking set per test file, compared to 7-Zip. 5-31
5.15 Encoding speed (in MiB/s) for the complete benchmarking set per test file, compared to 7-Zip. 5-32
5.16 Total decoding time for the complete benchmarking set per test file, compared to 7-Zip. 5-33
5.17 Decoding speed (in MiB/s) for the complete benchmarking set per test file, compared to 7-Zip. 5-34

List of Tables

1.1 File size of the human genome NA12878 (52x coverage) in the mainstream file formats for the storage of genomic data. 1-15
1.2 Total transmission time for four different data sets with respect to network band width. 1-22
1.3 Combined timeline of the research described in this dissertation and MPEG-G standardisation. 1-24

2.1 The Truncated Unary binarization for values 0 to 3 (`cMax=3`). . . 2-9

3.1 The binarization scheme for the alphabet indicator field. 3-7
3.2 Binarization scheme for prediction error corrections. 3-9
3.3 Binarization scheme for the predictor indicator. 3-9
3.4 Percentage of overhead versus the optimal CABAC reset window (131,072 blocks). 3-11
3.5 Detailed information of reads test set and optimal compression settings. 3-12
3.6 Compression results - reads (in bits per base). 3-14
3.7 Compression results - reads (file size, compared to other solutions). 3-14
3.8 Compression results - assembled sequences. 3-17

4.1 The 32 filters that are used by the CVP coding tool. 4-5
4.2 Examples of the Truncated Unary and Unsigned Exponential Golomb binary representations. 4-8
4.3 Examples of the Signed Exponential Golomb binary representation. 4-8
4.4 Detailed information of the quality score test set. 4-15
4.5 Compression results - single-pass compressors. 4-19
4.6 Compression results - dual-pass QVZ compressor. 4-20

5.1 The Binary binarization for input value 3 for different values of `cLength`. 5-12
5.2 The Truncated Unary binarization for values 0 to 3 (`cMax=3`). . . 5-12
5.3 The Exponential Golomb Binarization for values 0 to 8. 5-12
5.4 The Signed Exponential Golomb binarization for values -4 to 4 and their corresponding mapping for Exponential Golomb. 5-12

5.5 The Truncated Exponential Golomb binarization for values 0 to 4 (cMax=2). 5-13
5.6 The Signed Truncated Exponential Golomb binarization for values -4 to +4 (cMax=2). 5-13
5.7 Zero-order frequency distribution for the RCOMP descriptor stream for test file 02. 5-16
5.8 First-order frequency distribution for the RCOMP descriptor stream for test file 02 (rows: previous value, columns: current value). . . 5-16
5.9 Lookup tables to be used for the discussed example (rows: previous value, columns: current value). 5-17
5.10 Overview of the benchmarking set as proposed in [8] (U=Unmapped).5-18
5.11 Proposed syntax for signaling of the encoder parameters. 5-38
5.12 Values of binarization_id and their corresponding binarizations 5-39
5.13 Proposed syntax for signaling of a look-up table. 5-39

List of Acronyms

A

A	Adenine
ADFC	Average DiFference Coder
AFRESh	Adaptive Framework for compression of REads and assembled Sequences
ASCII	American Standard Code for Information Interchange
AVP	AVerage Predictor
AQUa	Adaptive framework for compression of sequencing QUality scores

B

BAM	Binary Alignment Map
BE	Binary Encoding

C

C	Cytosine
CABAC	Context-Adaptive Binary Arithmetic Coding
CALQ	Coverage-Adaptive Lossy Quality value compression
CARGO	Compressed ARchiving for GenOmics
CD	Comittee Draft
CfP	Call for Proposals
CoR	Codon Repetition
CVP	ConVolutional Predictor

D

DFC	DiFference Coder
DIS	Draft International Standard
DNA	Deoxyribo Nucleic Acid
DNR	Double Nucleotide Repetition
DRP	Double Repetition Predictor
DSRC2	DNA Sequence Reads Compressor

E

EBI	European Bioinformatics Institute
ERGC	Efficient Referential Genome Compressor

G

G	Guanine
GB	GigaByte, i.e. 1000 * 1000 * 1000 bytes
GiB	GibiByte, i.e. 1024 * 1024 * 1024 bytes
GNU	GNU's Not Unix
GWAS	Genome-Wide Association Study

H

HNSP	Hierarchical Normal Search Prediction
HRCSP	Hierarchical Reverse Complement Search Prediction
HTS	High-Throughput Sequencing
HxE	Huffman x-values Encoding

I

ISO/IEC	International Organization for Standardisation/International Electrotechnical Commission

IUB/IUPAC	International Union of Biochemistry/International Union of Pure and Applied Chemistry

K

KB	KiloByte, i.e. 1000 bytes
KiB	KibiByte, i.e. 1024 bytes

L

LFQC	Lossless FastQ Compressor
LPS	Least Probable Symbol
LUT	Look-Up Table
LZMA	Lempel-Ziv-MArkov chain algorithm

M

MB	MegaByte, i.e. 1000 * 1000 bytes
MiB	MibiByte, i.e. 1024 * 1024 bytes
MPEG	Moving Picture Experts Group
MPS	Most Probable Symbol

N

NCBI	National Center for Biotechnology Information
NGS	Next-Generation Sequencing
NSP	Normal Search Predictor

O

ORCOM	Overlapping Reads COmpression with Minimizers

P

PCR	Polymerase Chain Reaction

Q

QVZ	Quality Values Zip

R

RAM	Random Access Memory
RCSP	Reverse Complement Prediction

S

SAM	Sequence Alignment Map
SCALCE	Sequence Compression Algorithms using Locally Consistent Encoding
SNP	Single Nucleotide Polymorphisms
SNR	Single Nucleotide Repetition
SRP	Single Repetition Predictor

T

T	Thymine
TB	TeraByte, i.e. 1000 * 1000 * 1000 * 1000 bytes
TiB	TibiByte, i.e. 1024 * 1024 * 1024 * 1024 bytes
TEG	Truncated unary Exponential Golomb

V

VCF	Variant Calling Format

W

WGS	Whole Genome Shotgun

X

XML	eXtensible Markup Language

Nederlandstalige samenvatting –Summary in Dutch–

In de voorbije decennia is er een enorme vooruitgang geboekt bij de technologieën die gebruikt worden voor het digitaal uitlezen van DNA (Eng. DNA sequencing). Door deze vooruitgang is de kost voor het digitaal uitlezen van het menselijk genoom gezakt van meer dan 10 miljoen dollar in 2007 naar ongeveer 1000 dollar nu. In diezelfde tijdsspanne is de tijd die nodig is om een menselijk genoom uit te lezen gezakt van enkele jaren naar uren. Door deze prijsdaling en snelheidsverbeteringen komen vele applicaties die mogelijk zijn rond het uitlezen van DNA, zoals Genome-Wide Associatie Studies (GWAS) en studies omtrent de detectie van uitbraken van ziektes, meer en meer binnen het bereik van patiënten en (medische) onderzoeksinstituten.

Deze (r)evolutie resulteert echter in een nieuwe uitdaging: hoe gaan we om met de exponentieel groeiende hoeveelheid aan gegevens die gegenereerd wordt door het uitlezen van DNA. Als voorbeeld: één enkel menselijk genoom is al snel groter dan 350 GiB. Deze gegevens bestaan uit drie onderdelen: nucleotiden (het eigenlijke DNA), kwaliteitsscores (geven aan hoe zeker de sequencer is dat een nucleotide juist is geïdentificeerd) en read-namen (bevatten informatie over het gebruikte DNA uitleesproces en/of tot welk monster deze nucleotiden behoren). Gedurende de verwerking van deze gegevens, bv. bij het afbeelden van deze uitgelezen gegevens op een referentiegenoom, worden bovendien extra gegevens gegenereerd. Dit leidt uiteindelijk tot groottes van 550 GiB per menselijk genoom. Daarbij komt nog dat studies, om betrouwbaar te zijn, tientallen of zelfs honderden van deze genomen moeten analyseren (en dus opslaan), wat resulteert in petabytes aan gegevens. Bovendien is het in sommige landen (zoals de Verenigde Staten) verplicht om gegevens te bewaren die gebruikt werden bij het nemen van beslissingen rond de behandeling van patiënten. Deze gegevens moeten op een verliesloze wijze en langdurig bewaard worden. Bijgevolg is er een nood aan oplossingen voor het effectief (om de benodigde opslagruimte te beperken), efficiënt (om de benodigde verwerkingstijd en verwerkingskracht te beperken) en praktisch (om bv. gedeeltelijke opslag en verzending van gegevens mogelijk te maken) opslaan en beheren van deze gegevens.

Naast bovengenoemde gegevens voor onderzoek en behandeling van ziektes, zijn er ook genomische gegevens die gegenereerd en geanalyseerd worden in een tijdskritische context. Voor deze tijdskritische gegevens (bv. gegevens die gegenereerd worden voor de detectie van het uitbreken van ziektes) kunnen compressie-

oplossingen die toegevoegde functies aanbieden, zoals live encodering en streaming, verwerking van deze gegevens vergemakkelijken, bijvoorbeeld voor snelle analyse in gespecialiseerde netwerken ("in de cloud").

In deze dissertatie worden drie codeeroplossingen gepresenteerd die deze vereisten proberen te vervullen:

- AFRESh, een effectieve codeeroplossing voor het comprimeren van nucleotische gegevens met ondersteuning voor willekeurige toegang;
- AQUa, een effectieve codeeroplossing voor het comprimeren van kwaliteitsscores met ondersteuning voor willekeurige toegang;
- Een derde codeeroplossing die ontworpen is voor het comprimeren van de datastromen van de opkomende MPEG-G-standaard, en dit met een hoge effectiviteit en efficiëntie.

AFRESh en AQUa zijn gebouwd bovenop een generiek raamwerk dat ontworpen is voor de effectieve compressie van genomische gegevens (zoals nucleotiden en kwaliteitsscores). Bij de ontwikkeling van dit raamwerk werd gekeken naar de huidige aanpak van opslag, beheer en compressie van mediabestanden. Verder werden belangrijke onderzoeksfuncties toegevoegd, zoals uitgebreide configuratiemogelijkheden en eenvoudige uitbreiding van het raamwerk met bestandsformaten voor invoer en uitvoer, codeeralgoritmen en alfabetten. Het raamwerk biedt verder ondersteuning voor eenstapsencodering (Eng. single-pass encoding) zonder referentie en met ondersteuning voor willekeurige toegang. Bijgevolg maakt het raamwerk live codering mogelijk en biedt het optimale transporteffectiviteit, aangezien subsets van de gegevens kunnen worden uitgewisseld. Gedurende de verwerking van de gegevens splitst het raamwerk de gegevens in aparte blokken. Elk blok wordt dan gecodeerd met één codeeralgoritme dat is geselecteerd uit een verzameling van op maat gemaakte codeeralgoritmen. Voor elk blok wordt uit deze verzameling het meest effectieve algoritme geselecteerd. De resulterende datastroom wordt vervolgens gecomprimeerd met behulp van CABAC, een Context-Adaptieve Binaire Aritmetische Codeeroplossing.
Om de compressie van nucleotiden te ondersteunen en te optimaliseren, breidt AFRESh het generische codeerraamwerk uit met een verzameling van drie verschillende alfabetten (inclusief ondersteuning voor complementen) en negen op maat gemaakte codeeralgoritmen. Deze algoritmen kunnen in twee categorieën worden onderverdeeld: encodeeralgoritmen en predictie-algoritmen. Encodeeralgoritmen vertalen de gegevens in een blok naar een binaire representatie; predictiealgoritmen genereren een predictie (voorspelling) en de informatie nodig om deze predictie te corrigeren. De gegevens gegenereerd door deze algoritmen worden vervolgens omgezet naar bitsequenties (binarisaties genoemd). Dit gebeurt met behulp van op maat gemaakte binarisatieprocessen. Tenslotte worden deze binarisaties gecomprimeerd met behulp van CABAC. Deze aanpak resulteert in een verbetering in effectiviteit tot 41% voor reads en 34% voor geassembleerde se-

quenties in vergelijking met generische compressie-algoritmen zoals Gzip. Vergeleken met gespecialiseerde compressie-algoritmen, zoals SCALCE, LFQC en ORCOM, biedt AFRESh een verbetering in effectiviteit tot 51%. Daarbovenop biedt AFRESh ondersteuning voor willekeurige toegang tot de gecodeerde gegevens, een functie die heel waardevol is.

Om de compressie van kwaliteitsscores te ondersteunen en te optimaliseren, breidt AQUa het generische codeerraamwerk uit met een alfabet voor kwaliteitsscores en een verzameling van zeven codeeralgoritmen. Vier van deze algoritmen werden op maat gemaakt voor kwaliteitsscores, drie werden overgenomen van AFRESh, wat mogelijk is gemaakt door het concept van alfabetten. De gegevens gegenereerd door deze algoritmen worden vervolgens omgezet naar binarisaties en gecomprimeerd met CABAC. Deze aanpak resulteert in een verbetering van effectiviteit tot 38%, vergeleken met het generische compressie-algoritme Gzip, en tot 21% in vergelijking met het gespecialiseerde compressie-algoritme SCALCE. Verder werd AQUa ook vergeleken met een state-of-the-art compressie-algoritme dat data verwerkt in twee stappen (Eng. two-pass): QVZ. Dit tweestapsalgoritme analyseert en herwerkt/sorteert de gegevens in een eerste stap om vervolgens, zodra deze analyse en voorverwerking zijn uitgevoerd, deze voorbewerkte gegevens te comprimeren. Tweestapsalgoritmen zijn typisch meer complex dan eenstapsalgoritmen en hebben vaak nood aan tijdelijke opslag voor het bewaren van de analysegegevens en/of het bewaren van de voorbewerkte gegevens. Bovendien laten deze algoritmen geen live encodering toe, aangezien de data compleet moeten verwerkt zijn voor optimale compressie-effectiviteit. Vergeleken met QVZ biedt AQUa een effectiviteit die 6% tot 33% lager is, met uitzondering van één bestand waar AQUa een 1% hogere effectiviteit biedt. Verder biedt AQUa ondersteuning voor willekeurige toegang tot de gecodeerde gegevens.

De derde codeeroplossing die wordt geïntroduceerd in deze dissertatie is ontworpen als een oplossing voor het comprimeren van de verschillende datastromen binnen de MPEG-G-standaard voor de representatie, compressie en het beheer van genomische gegevens. Het doel van de MPEG-G-standaard is om een alternatief te bieden voor de huidige de facto standaarden FASTA/FASTQ en SAM/BAM (Eng. Sequence Alignment Map/Binary Alignment Map). MPEG-G zal hiertoe een verbeterde effectiviteit en efficiëntie bieden, samen met toegevoegde functionaliteiten zoals ingebouwde ondersteuning voor willekeurige toegang.

De voorgestelde codeeroplossing is gebaseerd op het codeerraamwerk dat werd ontworpen als basislaag voor de AFRESh- en AQUa-codeeroplossingen. Aangezien willekeurige toegang wordt voorzien in een hogere laag van de MPEG-G-standaard, werd deze functionaliteit verwijderd. Aan de andere kant werd het raamwerk uitgebreid met ondersteuning voor meerdere representaties voor de invoergegevens (dit vervangt de alfabetten), gegevenstransformaties (die encoders toelaten om bv. analyse-informatie te gebruiken door middel van opzoektabellen), en een uniforme decodeersyntax voor het signaleren van invoer-, transformatie-, binarisatie- en contextselectie-informatie. Andere belangrijke functies, zoals flexibele configuratie, zijn bewaard of zelfs uitgebreid om zo gebruikers toe te laten het encodeeralgoritme optimaal te configureren voor elk type van invoergegevens.

De codeeroplossing bestaat uit vier configureerbare stappen: data-invoer, transformatie, binarisatie en contextsetselectie. In de data-invoerstap worden de invoergegevens verwerkt volgens één van de drie mogelijke niveaus van granulariteit. Vervolgens worden deze gegevens getransformeerd met behulp van zes mogelijke transformatie-algoritmen. Deze gegevens worden vervolgens verwerkt door de binarisatiestap met behulp van één van de zes verschillende binarisatie-algoritmen. In de laatste stap wordt tenslotte de set van contexten geselecteerd, met behulp van één van de drie mogelijke selectie-algoritmen, en waarbij deze contexten zullen gebruikt worden bij de compressie van de binarisaties met behulp van CABAC.
Om het decodeeralgoritme toe te laten gecodeerde bitstromen te decoderen, wordt de benodigde informatie gesignaleerd in een daarvoor ontwikkelde syntax. Deze syntax is (samen met de decodeeralgoritmen) voorgesteld aan het MPEG-G standaardisatiecomité (werkgroep ISO/IEC JTC1 SC29/WG11) en werd geselecteerd als de basis voor het codeergedeelte van de MPEG-G-standaard. Deze basis is vervolgens verder uitgebreid met extra binarisaties, transformaties, ondersteuning voor sequenties van transformaties en syntaxelementen. Deze syntaxelementen bevatten de gegevens die nodig zijn voor deze toegevoegde technologieën en die niet konden weergegeven worden met de voorgestelde syntax.
Voor de analyse van de effectiviteit en efficiëntie van het voorgestelde algoritme werden twee configuratiesets gecreëerd: een snelle modus (Eng. fast mode), met een focus op efficiëntie, en een trage modus (Eng. slow mode), met een focus of effectiviteit.
Wanneer we de codeeroplossing vergelijken met de state-of-the-art generische compressor 7-Zip, in LZMA-ultraconfiguratie, over de gehele MPEG-G-dataset voor prestatie-analyse (Eng. benchmarking), biedt de codeeroplossing een hogere effectiviteit (de gegevens worden gereduceerd tot 21.97% van de ongecomprimeerde grootte in snelle modus en 20.99% in trage modus, vergeleken met 22.25% voor 7-Zip). Daarbovenop biedt de codeeroplossing (die niet is geïmplementeerd met het oog op optimale efficiëntie en die wordt uitgevoerd als een ééndraadsproces) een efficiëntie die 6.14 keer (trage modus) tot 16.96 keer (snelle modus) hoger is dan 7-Zip (een tweedraadsproces).
Een ander belangrijk kenmerk van deze codeeroplossing is de beperkte hoeveelheid geheugen die nodig is voor de opslag van (tijdelijke) gegevens tijdens het encoderen/decoderen: maximaal 64 KiB is nodig, tijdens het verwerken van de gehele MPEG-G-dataset voor prestatie-analyse, voor het opslaan van alle transformatiegegevens (bv. opzoektabellen, zoekvensters), invoergegevens, tussentijdse waarden (transformaties, binarisaties) en uitvoergegevens, maar ook voor de contextsets en de tabellen nodig voor de werking van CABAC. Ter vergelijking: 7-Zip heeft 709 MiB geheugen nodig voor encodering en 66 MiB voor decodering.

Het onderzoek dat werd beschreven in deze dissertatie resulteerde in een codeeroplossing die zal worden gebruikt door de MPEG-G-standaard, gegeven enkele uitbreidingen zoals extra binarisaties, transformaties, ondersteuning voor sequenties van transformaties, en aanpassingen aan de syntax die volgen uit deze uitbreidingen.

English summary

In the past decades, significant advancements have been made in technology used for DNA sequencing. As a result of these advancements, the cost for sequencing a human genome has dropped from over $10 million in 2007 to around $1,000, nowadays. Concurrently, the time required for sequencing a human genome has dropped from multiple years to hours. With this price drop and speed increase, many applications relying on DNA sequencing (such as, personalized medicine, Genome-Wide Association Studies (GWAS), and studies on and detection of outbreaks of diseases) have come within reach of more and more people and (medical) research institutes.
This (r)evolution results in a major new challenge: how to handle the exponentially growing amount of data generated through DNA sequencing. As an example, the sequencing data associated with a single human genome can easily exceed 350 GiB. These data consist of three parts: nucleotides (the actual DNA), quality scores (indicating the certainty with which the sequencer identified a nucleotide), and read names (containing information on the used sequencing process and/or information on the sample to which these nucleotides belong). During processing of these data (e.g., by mapping these sequencing data onto a reference genome) additional data are generated, resulting in total file sizes of more than 550 GiB. Furthermore, for studies to be reliable, tens or even hundreds of these genomes need to be stored and analysed, resulting in petabytes of data. Additionally, it is required in some countries (such as the USA) to store data that have been used for decisions for or during a medical treatment. These data have to be stored in a lossless manner and for longer periods. Hence, there is a need for solutions that offer an effective (to limit the required storage space), efficient (to limit the required processing time/power) and practical (e.g., to allow for partial transmission and storage of data) solution for storage and management of these data.
Besides the types of genomic data that are used for research and treatment, there are genomic data that are created and analysed in a time-critical context. These time-critical data (e.g., data generated for disease outbreak detection) can benefit from compression tools that offer additional features, such as live encoding and streaming to, for example, cloud analysis networks.

In this dissertation, three coding solutions are presented that strive to meet these requirements:

- AFRESh, an effective coding solution for the compression of nucleotidic

data with support for random access;

- AQUa, an effective coding solution for the compression of quality scores with support for random access; and
- a third coding solution that has been designed to compress the data streams for the upcoming MPEG-G standard with high effectiveness and high efficiency.

AFRESh and AQUa are built on top of a generic framework that has been designed to allow for effective compression of genomic data (i.e., nucleotides and quality scores). This framework has been inspired by the approach used for storage, management, and compression of media files and, additionally, offers key features for research purposes such as, configurability and easy extensibility with input/output file formats, coding tools, and symbol alphabets. The framework offers single-pass, no-reference encoding, together with random access. As such, it allows for live encoding and optimal transmission effectiveness, as partial data can be sent.
During processing of data, the framework splits the input data into separate blocks. Each block is then encoded using one coding tool, selected from a set of tailor-made coding tools. For each block, the most effective coding tool is selected. The resulting data stream is then compressed using a Context-Adaptive Binary Arithmetic Coder (CABAC).
To support and optimize the compression of nucleotidic data, AFRESh extends the generic coding framework with a set of three symbol alphabets (including complement information) and nine tailor-made coding tools. These tools can be split into two major categories: encoding tools and prediction tools. Encoding tools convert the data in a block into a binary representation, whereas prediction tools generate a prediction and create correction information. The output data of these coding tools are then converted into bit sequences (called binarizations) using tailor-made binarization processes and these bit sequences are then finally processed by CABAC. This approach offers an effectiveness improvement of up to 41% for reads and 34% for assembled sequences, when compared to commonly used generic data compressors such as Gzip. When compared to specialized compressors such as SCALCE, LFQC, and ORCOM, an effectiveness improvement of up to 51% is provided. Additionally, AFRESh offers support for random access within the coded data, a feature that is deemed very valuable, especially for data exchange.
To support and optimize the compression of quality scores, AQUa extends the generic coding framework with a symbol alphabet for quality scores and a set of seven coding tools. Four of these coding tools were tailor-made for quality scores, three are inherited from AFRESh, which is achievable thanks to the concept of alphabets. The output data of these coding tools are then converted into binarizations, which are then processed by CABAC. This approach offers an effectiveness improvement of up to 38%, when compared to the commonly used generic data compressor Gzip, and up to 21% when compared to the purpose-built compression format SCALCE. Additionally, AQUa is compared to a two-pass state-of-the-art compressor: QVZ. This two-pass compressor analyzes and pre-processes the in-

put data in a first pass and compresses, after the analysis and pre-processing, the output of this first pass in a second pass. This approach is typically more complex than single-pass approaches and requires temporal storage space for the analysis data and/or the storage of the pre-processed data. Furthermore, this approach does not allow for live encoding, as the data need to be available to provide an optimal coding effectiveness. Compared to QVZ, the effectiveness of AQUa is 6% to 33% lower, except for one test file, where AQUa is 1% more effective. Additionally, AQUa offers support for random access within the coded data, a feature that is not offered by the other solutions.

The third solution, as introduced in this dissertation, has been designed as a solution for the compression of the different data streams of the MPEG-G standard for the representation, compression, and management of genomic data. The goal of the MPEG-G standard is to provide an alternative to the current de facto standards FASTA/FASTQ and SAM/BAM (Sequence Alignment Map/Binary Alignment Map). MPEG-G will provide improved effectiveness, improved efficiency, and additional functionalities (such as built-in support for random access).

The presented solution is based upon the coding framework that was designed as a base layer for AFRESh and AQUa. As random access will be handled in a higher layer of the MPEG-G standard, this feature has been removed. On the other hand, the framework has been extended with support for multiple representations of input data (replacing symbol alphabets), data transformations (allowing encoders to use analysis information, e.g., through look-up tables), and a unified decoding syntax for signaling input, transformations, binarizations, and context selection parameters. Other key features, such as a flexible configuration, have been preserved (or extended), hence allowing users to optimize the encoding process for each type of input.

The coding solution consists of four configurable processes: data input, transformation, binarization, and context set selection. In the data input step, the input data are processed in one of three different granularities. In the transformation step, the input data can be transformed using six possible transformation algorithms (or none, i.e., passing through the input data). The output of the transformation step is then processed by the binarization step, which offers six different binarization algorithms. In the final step, a set of contexts is selected for processing the binarizations with CABAC. This selection can be performed using one of three presented context selection algorithms.

To provide the decoder with the information needed for decompression, a decoder syntax has been designed. This decoder syntax (together with the decoding algorithms) has been proposed to the MPEG-G standardization committee (working group ISO/IEC JTC1 SC29/WG11) and acts as a baseline for the coding part of the MPEG-G standard, which has been extended with additional binarizations, transformations, support for transformation chains, and additional syntax elements. These syntax elements signal the information required for these additional technologies which could not be represented with the proposed syntax.

For the analysis of the effectiveness and efficiency of the proposed algorithm, two sets of configurations have been created: fast mode, with a focus on efficiency, and

slow mode, with a focus on effectiveness.
When comparing the coding solution to the state-of-the-art generic compressor 7-Zip, with the LZMA ultra setting, across the MPEG-G benchmarking data set, the coding solution offers a higher effectiveness (reduction to 21.97% of the original file size in fast mode and to 20.99% in slow mode, compared to 22.25% for 7-Zip). Additionally, the coding solution (which has not been implemented for optimal efficiency and is only executed in single-threaded mode) offers encoding time reductions over 7-Zip (in dual-threaded modus) of 6.14 times (slow mode), and 16.96 times (fast mode).
Another important feature offered by the coding solution is the limited amount of memory required to store (temporary) values during encoding/decoding: the maximum memory footprint during the encoding/decoding of the complete MPEG benchmarking set for the storage of all transformation data (e.g., look-up tables, search windows), input (input value), intermediate (transformation and binarization), and output data, and context sets and arrays required for CABAC is 64 KiB. As a comparison, the total memory required for 7-Zip (in the tested LZMA ultra configuration) is 709 MiB for encoding and 66 MiB for decoding.

The research described in this dissertation has resulted in a coding solution that will be adopted by the MPEG-G standard, adding several extensions such as additional binarizations, transformations, transformation chains, and limited adaptations to the syntax following from these extensions.

1 Introduction

1.1 Introduction

The past decade has seen several (r)evolutions in Deoxyribo Nucleic Acid (DNA) sequencing (i.e., reading). Each of these (r)evolutions resulted in faster and cheaper sequencing of DNA. Therefore, the use of DNA sequencing has increased significantly as previous limitations (speed and cost) have been overcome. Many applications (e.g., personalized medicine, Genome-Wide Association Studies (GWAS), and studies of outbreaks of diseases) have now (or will soon) come within reach of more people and (medical) research institutes.
However, with the rising popularity of DNA sequencing, new issues can be identified regarding the storage and transmission of the resulting data. As an example, one human genome can require several hundreds of gigabytes of storage space. This poses a challenge for institutions that want (or are required) to store/archive genomic data or require large sets of genomes for (medical) research. Additionally, transmission of one such human genome can still take hours or days, even over fast broadband networks. As a result, transmission of such large amounts of genomic data is typically handled by shipping hard drives through a courier service.
In this dissertation, technologies are presented that aim to lower the storage and transmission costs, improve transmission speeds, and facilitate the handling of these genomic data by offering an efficient, effective, and flexible compression solution.

Figure 1.1: The double helix molecular structure of DNA as discovered by Watson and Crick.

The final result of this dissertation has formed the basis for the coding part of an international standard for the representation, compression, and management of genomic data, called MPEG-G.
In this chapter, an overview is provided of:

- the concepts that are important to understand the content of this dissertation;
- the evolution of the technologies used to sequence DNA and their effect on sequencing speed and cost; and
- the problems that arise from the ever increasing popularity of DNA sequencing, thanks to the decreasing costs and improved speeds.

1.1.1 Deoxyribo Nucleic Acid (DNA)

DNA is a molecule that forms the basis of all living organisms on earth. It contains the genetic code that is responsible for the growth, development, functioning, and reproduction of these organisms. The first extraction of DNA was performed by Friedrich Miescher in 1869. Almost a century later, in 1953, James Watson and Francis Crick identified the double helix molecular structure of DNA [1] (see Figure 1.1). Watson and Crick had discovered that DNA is constructed out of two strands, each consisting of a sequence of nucleotides: A for Adenine, C for Cytosine, G for Guanine, and T for Thymine. Figure 1.2 shows an example of two such strands. To connect these strands into a double helix structure, each nucleotide on one strand is connected to its complement on the other strand, i.e., a nucleotide T on one strand is connected to a nucleotide A on the complementary strand (and vice versa), and a nucleotide C on one strand is connected to a nucleotide G on the complementary strand (and vice versa). Given this complementarity, one can reconstruct the complete genome of an organism by reading only one of these

strands. In case of the human genome, this strand contains around 3.2 billion nucleotides [2].

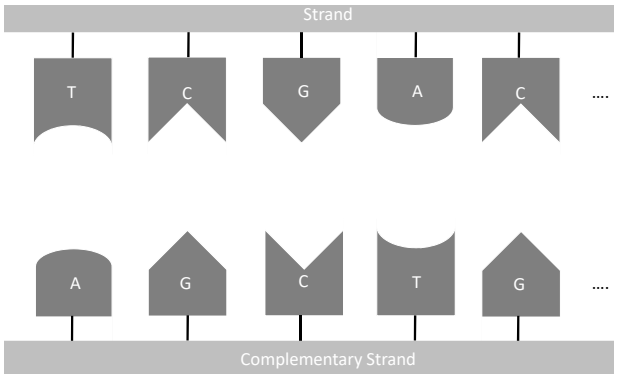

Figure 1.2: Example of two complementary DNA strands.

1.1.2 DNA Sequencing & Assembly

After the discovery of the extraction and the (chemical) structure of DNA, the next challenge was to determine the exact order of the nucleotides within a DNA sample. Currently, this process consists of two steps: sequencing and assembly. In the sequencing step, the order of the nucleotides is determined for segments of a DNA sample, called reads. In the assembly step, these segments are reassembled to recreate the nucleotide sequence of the DNA sample. In this section, a more detailed sequencing and assembly workflow is discussed, followed by a brief overview of the evolution of DNA sequencing technologies.

1.1.2.1 Sequencing & Read Assembly Workflow

A modern (simplified) sequencing and assembly workflow can be split into four steps: sample preparation, library preparation, sequence generation, and (optionally) read assembly (see Figure 1.3). In the sample preparation step, a DNA strand is extracted from the DNA sample. In the library preparation step, the DNA strand is prepared for sequencing. This step, in its turn, can consist of three separate processes:

- **Target selection** - selecting a specific region in the DNA sample to limit the sample size (and as a consequence the sequencing cost). For whole genome sequencing, this step is skipped;
- **Cloning/Clustering** - increasing the number of DNA samples that can be used in the sequencing step to improve the sequencing accuracy; and

- **Fragmentation** - splitting the DNA samples into random segments (reads) that can be processed by the sequencing machine.

In the sequencing step, the order of the nucleotides is defined for all segments created in step two.

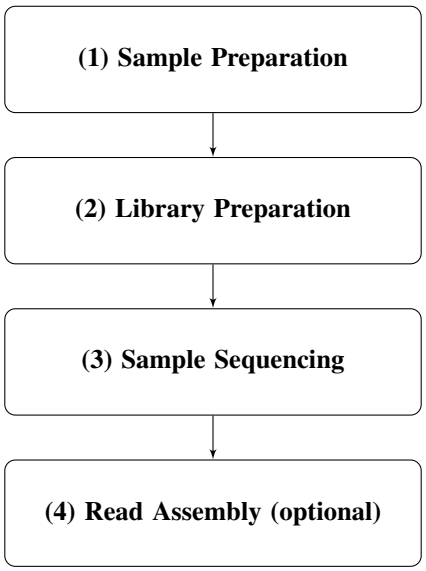

Figure 1.3: A modern (simplified) sequencing and assembly workflow.

After sequencing, the available data consist of the set of randomly generated segments as created during fragmentation, possibly containing errors. To recreate the DNA sequence of the original DNA sample, read assembly is performed (i.e., step four in Figure 1.3).
Read mapping can be divided into two categories, based on whether a reference sequence is used for mapping or not: reference mapping and de novo mapping.
De novo assembly does not use a reference sequence and as such reassembles the DNA sequence by mapping the sequenced segments onto each other. De novo assembly is highly complex: millions of short segments, possibly distorted by sequencing errors, have to be merged into a single ”puzzle”, without information of what the final result should look like.
Reference assembly (i.e., mapping reads onto an existing reference sequence) is significantly less complex, as an ”example solution” of the output is available. However, finding a mapping for each of the millions of segments is still a complex task, especially when taking into account that these segments can contain sequencing errors and hence might require corrections to map onto the reference sequence. The data created by the assembly step will be referred to as mapping data.

1.1.2.2 Comparing Sequencing Technologies

Before discussing the evolution of sequencing technologies, it is important to identify the four dimensions along which sequencing technologies can be compared:

- **Cost** - what is the total cost of sequencing one base (or one genome)? Cost can be expressed in a highly diverse manner (e.g., including sequencing machine purchase cost and/or preparation cost or not). Therefore, cost will only be discussed on a magnitude basis, with some indications of a typical cost;
- **Speed** - how long does it take to sequence one base (or one genome)? As with cost, speed can be expressed in many ways (e.g., by including preparation time or not, or by deducting the per-base speed from a larger test (to divide the preparation time across all processed nucleotides)). Therefore, speed will only be discussed on a magnitude basis, with some indications of the typical sequencing speed;
- **Accuracy** - what percentage of bases is read without an error? This is an important measure. It should be noted that none of the currently existing technologies has an accuracy of 100%. This, together with the limited maximum read length, explains the need for the cloning and fragmentation procedure in the sequencing workflow. By cloning the DNA samples and splitting the samples in reads of a (random) length, multiple copies of each nucleotide in the genome are created (i.e., coverage[1], or number of segments containing a nucleotide at locus x, where locus is a position on a genome). Based upon these multiple copies, the most probable call for a given position can be identified. If the accuracy of a certain technique is lower, one can improve the accuracy by increasing the coverage. However, it is important to keep in mind that the amount of output data to be stored increases linearly with coverage. Besides accuracy, the type of errors can be important, as some types of errors (e.g., due to inability to capture long repetitions of nucleotides) cannot be countered effectively; and
- **Read Length** - what is the number of bases that are contained in one read? Longer reads will be easier to map as there are less segments needed for equal coverage and longer segments are easier to map correctly. This is especially important for de novo sequencing.

[1]The average coverage is indicated using a number followed by an 'x'. 52x indicates that the average coverage of a given file is 52.

1.1.2.3 The History of DNA Sequencing

Since the development of the first sequencing technologies, three generations of sequencing have been developed. The first generation of sequencing technologies is a fully manual process, both for sample preparation and base calling (i.e., the identification of the order of nucleotides in a sample). The second generation, which is still widely used, provides many cost and speed improvements by automating and parallelizing these process steps. The third generation follows a different approach, where the preparation step is (almost) non-existent and DNA strands are sequenced by 'reading' the DNA strands from start to end, without splitting the strands into smaller segments. In the rest of this section, a more detailed discussion of the different generations is provided.

Generation 1: Manual Sequencing The Sanger method [3] is seen as the first mainstream method for sequencing DNA. Succeeding sequencing methods by Wu [4], and Maxam and Gilbert [5], it was widely used thanks to its relative reliability and safety. It consists of two steps: the sample preparation step and the base calling step. During the sample preparation step, a set of DNA segments (of a random length) is generated from the original DNA sample. At the end of each sample, a radio-active label is installed to allow identification in the second step. This process is performed in fourfold, where each instance creates DNA samples that end with one specific nucleotide (i.e., A, C, G, or T). The resulting segments of each of these instances are then put into a separate lane on a glass plate. Through capillary electrophoresis, the segments in each sample are then 'sorted' on the glass plate by size.
In the next step, an X-ray photograph is taken from the glass plate to identify the position of the different reads for each of the instances (i.e., the instances that end with nucleotide A, C, G, or T, respectively). Figure 1.4 shows a simplified result of this process. Each bar indicates samples that contain a radio-active label at their end position for that specific instance. As it is known which nucleotide is radio-actively labelled (indicated at the top of the column) and the segments are sorted by length, the sequence of the DNA sample can be read by identifying the nucleotide for each row (i.e., for each base position). This process is performed from bottom to top.
Given the manual processing of all preparation and sequencing steps (including the identification of the order of the nucleotides), it is clear that the first generation of sequencing techniques is very slow and requires a lot of manual labour, resulting in high costs.

Generation 2a: Automated Sequencing To circumvent the extensive manual labour that is required for Sanger sequencing, several improvements have been applied to the original method to lower preparation time (and thus cost) and to

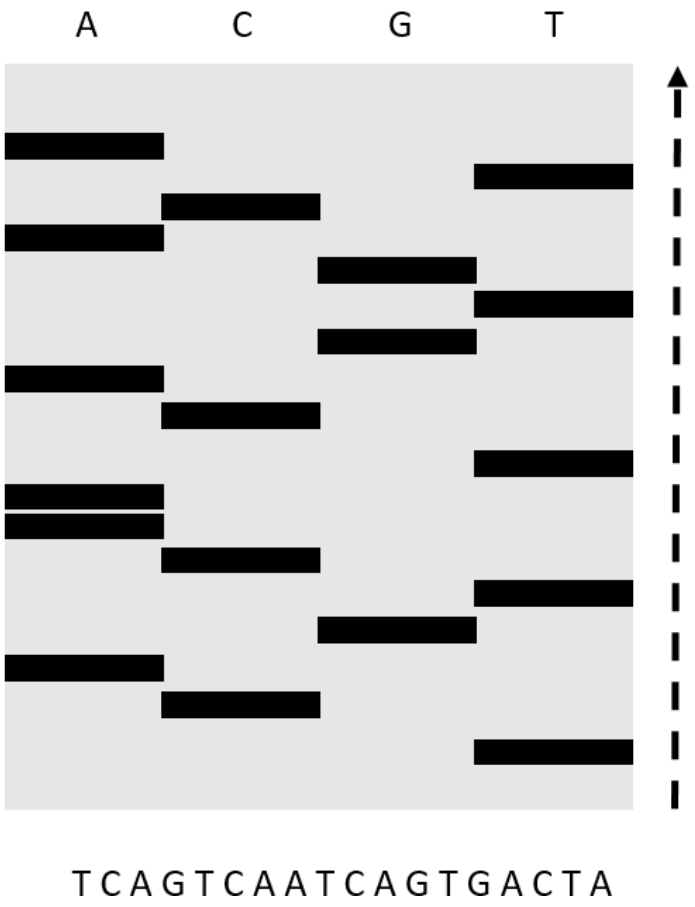

Figure 1.4: Output of the Sanger sequencing of a DNA strand.

allow for automatic (and full-genome) sequencing. The first major improvement was the automated and parallel creation of clones of the DNA sample. In the cloning process, the DNA sample is divided into segments and inserted into the DNA of a self-replicating host [6]. This process can generate more than 10^{12} identical copies in less than a day, as such amplifying the fragment accordingly. A sample of this colony of clones can then be picked (colony picking) and used for sequencing. To sequence a human genome, this repetitive and labour intensive task has to be performed 100,000s of times, so automation of this whole task has been a major step to faster and cheaper sequencing. A second improvement was the replacement of radioactive labels with fluorescent labels with a separate colour for each nucleotide. This allows for the merging of the four reactions into one

reaction (see Figure 1.5). By using these four different colours for the fluorescent labels, the result after capillary electrophoresis can be read automatically (e.g., by using a laser, and in one pass).
The development of these first steps towards automated sequencing coincided with the sequencing of the first human reference genome. This project took 13 years and cost around 2.7 billion dollar [7]. After this project, it was estimated that, based on the development of these automation technologies, sequencing a second human reference genome would cost significantly less: around 50 million dollar [7].

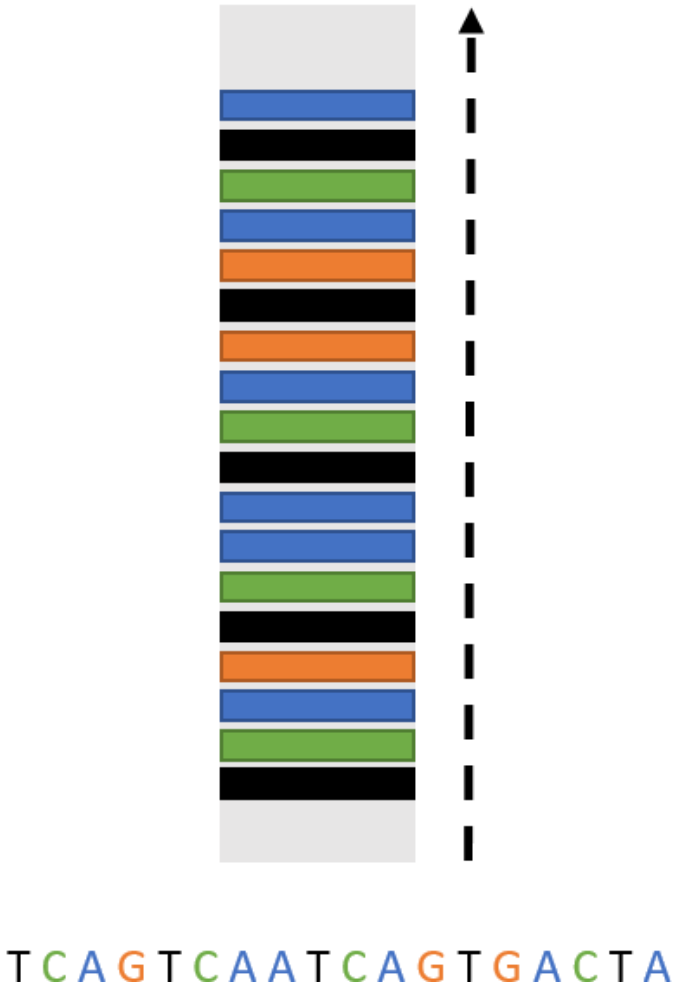

Figure 1.5: Output of a single-reaction Sanger sequencing process, using fluorescent colour labels.

Generation 2b: High-Throughput Sequencing In a second stage of automation (called Next-Gen Sequencing or High-Throughput Sequencing), the sample preparation step has been further optimized by replacing the host-based cloning method by so-called in vitro cloning. In vitro cloning is a cloning process that uses an emulsion or bridge Polymerase Chain Reaction (PCR) to clone the DNA segments.
This evolution, together with further optimizations of the preparation process and further technological advances in sequencing technology (e.g., improved sensors, increased parallelism), resulted in the current cost of less than 1,000 dollar for sequencing a whole human genome[2] [9], and a total sequencing time of one hour per genome, when sequencing with the Illumina NovaSeq 6000[3] (see Figure 1.6). The typical read lengths of this technology are currently 50 to 150 bases[4] with a minimal confidence[5] of 85% and 75%, respectively, and with 91.3% of the bases sequenced with a confidence above 99.9% [11].

Figure 1.6: The Illumina NovaSeq 6000 DNA sequencer.

[2]This is based on a setup of 10 Hiseq X sequencers, including instrument depreciation, DNA extraction, sample preparation, and labour.
[3]The NovaSeq 6000 is able to sequence 48 genomes at a time, resulting in a one genome per hour claim. The run time of the shortest process that can sequence one genome is around 13 hours [10].
[4]The Illumina MiSeq produces read lengths of up to 250 bases.
[5]i.e., a value calculated by the sequencing machine to indicate the probability that a base is sequenced correctly.

Generation 3: Single-Molecule Sequencing To further lower the cost of DNA sequencing and to increase the read lengths, a new generation of sequencers is currently being developed that handle single strands of DNA, thus avoiding the need for cloning. Multiple solutions are currently available:

- **PacBio** - uses a microscope to "watch" the process of a single strand of DNA replicating itself (thus voiding the cloning step and lowering the preparation time). The approach of reading a single strand also results in significantly larger reads, having a length of 2,000 to 60,000 bases. The accuracy of this sequencing technology is around 86%. This accuracy can be improved to around 99.999% by running the sequencing process at a 50x coverage [12]; and
- **Oxford Nanopore** - "pulls" a single DNA strand through a pore and monitors the chemical process inside the pore to call each of the passing bases. This method is already implemented inside a thumb drive sequencer: the Oxford Nanopore minION (see Figure 1.7). This device is available for $1,000[6]. The reads produced by this technology can be longer than 150,000 bases, with an accuracy of more than 80% and a consensus accuracy of 99.5% at 30x coverage [14]. Each run can process up to 4.4 million reads within 48 hours [15]. An important feature of this technology is the availability of the sequencing data in real time, allowing for live streaming of data to the point of analysis[7].

Figure 1.7: The Oxford Nanopore MinION, a thumb drive sequencer.

[6]This is significantly less than the Illumina Novaseq 6000 sequencer, which costs around $985,000 [13].

[7]The time to the sequencing of the first base is two minutes.

1.1.3 Sequencing and Mapping Data

In the previous section, the sequencing and assembly of a DNA sample has been discussed. In this section, an overview is provided of the different types of data that are generated during these sequencing and assembly processes.

1.1.3.1 Data generated during the Sequencing Process

During the sequencing process, sequencers of the second and third generation output three different types of data per sequenced read[8]:

- **Read name** - textual information to identify a read. This information can also contain data on the sequencing process and equipment;
- **Nucleotidic data** - the nucleotides of the read; and
- **Quality data** - the quality scores, which are defined by the sequencing machine and represent the error probability per nucleotide.

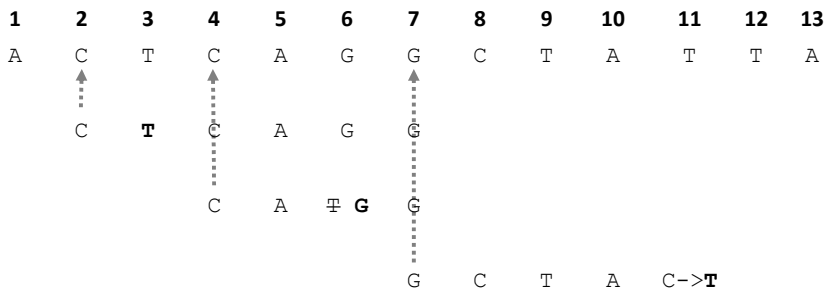

Figure 1.8: Examples of mapping corrections: Insertion, Deletion, and Single Nucleotide Polymorphism.

1.1.3.2 Data generated during the Read Assembly Process

To recreate a sequenced genome[9], based upon the sequencing data from Section 1.1.3.1, an assembly process can be performed (See Section 1.1.2.1). The assembly process creates three types of data per read[10]:

[8] Sequencing technologies of the first-generation sequencing process are exclusively manual and only result in a set of base calls. Hence, the output consists of nucleotidic data (and optionally read name information).

[9] The term genome has been selected as a reference to the different omics data, such as (16S) metagenomics, RNASeq, and transcriptomics. From a coding point of view, these data are equal as they are stored using the same data formats and alphabets.

[10] These data will be referenced to with the generic term mapping data.

- **Read position** - the position at which each read is mapped (on the respective reference genome);
- **Mapping score** - the certainty with which this mapping is correct; and
- **Mapping corrections** - the corrections that need to be applied to the read to create a perfect mapping, containing pairs of a base position (i.e., the position in the read that needs to be corrected) and a correction. These corrections are one of three types:
  - **Insertions** - insertion of a certain nucleotide at a given position. In Figure 1.8, a nucleotide T was inserted into read `CCAGG`, to map the (first) read onto position 2;
  - **Deletions** - deletion of the nucleotide at a given position. In Figure 1.8, a nucleotide T was deleted, to map read `CATGG` onto position 4; or
  - **Single Nucleotide Polymorphism (SNP)** - replacement of the nucleotide at a given position by another nucleotide. In Figure 1.8, a nucleotide C was replaced with nucleotide T, to map read `GCTAC` onto position 7.

If reads cannot be mapped[11], they are stored as unmapped reads, hence voiding the need for the storage of mapping information.

1.1.4 Storage of Genomic Data

To preserve the data described in Section 1.1.3.1 and in Section 1.1.3.2, two file formats became prevalent for data storage and data exchange: FASTA/FASTQ [16] for sequencing data and SAM (Sequence Alignment Map) [17] and its binary equivalent BAM (Binary Alignment Map) for read mapping data (which inherently includes the sequencing data). These file formats are the de facto standards and are highly popular as they offer human readability[12], but more importantly, they can be easily processed using scripting languages and are the input/output formats supported by most of the software libraries and tools used in the bioinformatics field. However, they are not effective at storing genomic data as they store the data in a raw form (i.e., without compression) and typically in the ASCII format (except for BAM).
In this section, a short overview will be given of both the FASTA/FASTQ and SAM/BAM file formats, including an example.

[11] The thresholds that define whether a read is categorized as unmapped are defined by the mapping software and, as such, can differ between software solutions and even between software versions.
[12] Note: BAM is the binary representation of SAM and, as such, not human readable by default.

```
Line 1: >EAS149:136:FC806VJ:2:2104:1543:19393
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT

Line 1: >EAS139:136:FC706VJ:2:5:1000:12850
Line 2: AAGAGCAGTATCGATCATAATAGTATAACCAATGTACATTTGTCAGCG
```

Figure 1.9: Example of two reads in a FASTA data file, using the current Illumina read identifier syntax. The text in bold is for clarity, this text is not part of the actual file.

```
Line 1: @SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 4: EEDEEDEDCCDDDCCBA><>>>>>>>>>====<<<888976333#####

Line 1: @SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 4: IIIIIIIIHFGFFEEDDCDDDCBBA>>>><<88887778776666665
```

Figure 1.10: Example of two reads in a FASTQ data file, using the NCBI Sequence Read Archive read identifier syntax. The text in bold is for clarity, this text is not part of the actual file.

FASTA/FASTQ Sequencing data (see Section 1.1.3.1) are typically stored in the FASTA/FASTQ file format. FASTA/FASTQ files contain reads consisting of two lines (FASTA, see Figure 1.9) or four lines (FASTQ, see Figure 1.10) of data:

- Line 1: **Read Name Identifier** - a sequence of characters describing the data in the read and/or providing information about the sequencing process (e.g., identification number of the used sequencing machine or bar code information).
  The read name identifier line starts with an ">" (FASTA) or an "@" (FASTQ) to facilitate efficient parsing. The syntax of the sequence of characters in the read name identifier is not fixed but manufacturers and large genome archives typically have a fixed syntax. Figure 1.9 shows an example of an Illumina identifier, whereas Figure 1.10 shows an example of the NCBI Sequence Read Archive identifier;

- Line 2: **Nucleotidic data** - the (read) nucleotides;

- Line 3 (FASTQ only): **Identifier** - the character "+" followed by an optional sequence of characters, typically a repetition of the read name identifier or empty; and

- Line 4 (FASTQ only): **Quality scores** - one value, ranging from value 33, i.e., ('!'), to value 126 ('∼')[13], per nucleotide in line 2, indicating the certainty that the respective nucleotide is sequenced correctly (i.e., base call accuracy). These values are defined by adding a fixed offset (typically 33) to the Phred quality scores. Phred quality scores are a logarithmic representation of the base call accuracy (i.e., the certainty that a nucleotide is sequenced correctly). This offset ensures that the representation of the Phred quality scores are limited to the range of human-readable ASCII characters (i.e., [33,126]). In the FASTQ files created by more recent sequencers (and the corresponding base callers), such as the Novaseq 6000, the set of quality scores has been limited to a small subset of 4 quality scores [18].

```
@HD VN:1.5 SO:coordinate\\
@SQ SN:ref LN:45\\
r001 99 ref 7 30 8M2I4M1D3M = 37 39 TTAGATAAAGGATACTG *\\
r002 0 ref 9 30 3S6M1P1I4M * 0 0 AAAAGATAAGGATA *\\
r003 0 ref 9 30 5S6M * 0 0 GCCTAAGCTAA * SA:Z:ref,29,-,6H5M,17,0;
```

Figure 1.11: Example of a SAM file containing three reads.

SAM/BAM Mapping data (see Section 1.1.3.2) are typically stored in the SAM file format, or its binary equivalent, the BAM file format. SAM/BAM files enable the storage of information that has been used in or has been generated during the sequencing and read mapping stage. Figure 1.11 shows an example of a SAM file, including an optional header on line 1 and line 2. The main information stored in a SAM/BAM file is:

- **Mapping Reference** - identification of the reference to which the read is mapped;
- **Read Length** - the length of the read in number of nucleotides;
- **Read Mapping Positions** - the position in the reference to which a read is mapped;
- **CIGAR Information** - the list of positions and corrections applied to a read, as discussed in Section 1.1.3.2;
- **Unmapped Reads** - reads that were not mapped onto a reference, are stored raw. This replaces Read Mapping Positions and CIGAR information;

[13]It should be noted that the FASTQ standard does not define the values and their corresponding certainty.

- **Mapping Quality Score** - the (estimated) quality of the mapping; and
- **Quality Scores** - the read quality scores.

1.1.4.1 File Size

As can be seen in the examples of FASTA/FASTQ and SAM in Section 1.1.4, genomic data are typically stored in a human readable manner, and thus uncompressed. BAM files are stored in a more effective representation (rendering it unreadable for humans), including a compression layer. To provide an indication of the storage required for these files, Table 1.1 shows the file size for the FASTA, FASTQ, SAM, and BAM versions of the NA12878 test file (with an average coverage of 52x, used in Chapter 3 and Chapter 4). The sequencing data for a single human genome can easily exceed 350 GiB when stored in the FASTQ format. Including the mapping data, the size can increase to around 550 GiB, which still exceeds 110 GiB in its binary equivalent BAM version. In the next section, we will discuss the cost of storing such a genome, compare the evolution of this cost with the evolution of sequencing cost, and discuss the effect of low sequencing costs on the generation of genomic data.

File Format	File Size
FASTA	215.3 GiB
FASTQ	368.6 GiB
SAM	548.6 GiB
BAM	113.3 GiB

Table 1.1: File size of the human genome NA12878 (52x coverage) in the mainstream file formats for the storage of genomic data.

1.1.4.2 Storage Cost

As discussed in Section 1.1.2, the cost of sequencing DNA dropped significantly compared to the cost of sequencing the first Human Genome. In this section, we will discuss the evolution of the resulting storage cost.
Figure 1.12 shows the evolution of sequencing cost from 2002 to 2017, compared to the evolution of the storage cost (represented by the hard drive cost, expressed in dollars per Gigabyte), both on a logarithmic scale. This graph shows that both costs decrease at a similar pace, resulting in a stable storage cost if the number of sequenced nucleotides rises proportional to the decrease in sequencing costs. However, as can be expected, the decreasing costs result into a higher than proportional growth. As an example of this growth, Figure 1.13 shows the evolution of the size of the nucleotidic information stored in the Whole Genome Shotgun (WGS) database of the National Center for Biotechnology Information (NCBI),

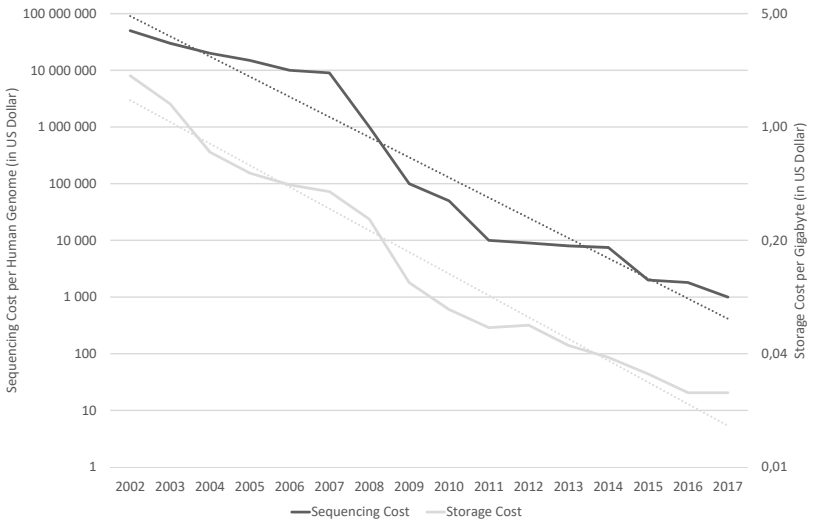

Figure 1.12: Evolution of sequencing and storage costs (2002-2017). Generated from data provided by [19], [20], and [21].

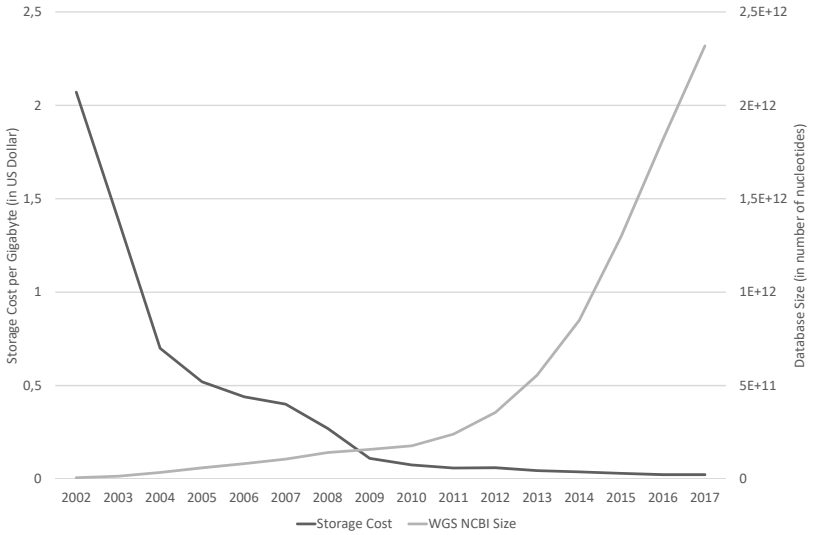

Figure 1.13: Evolution of sequencing cost and size of the NCBI WGS database (2002-2017). Generated from data provided by [19], [20], and [21].

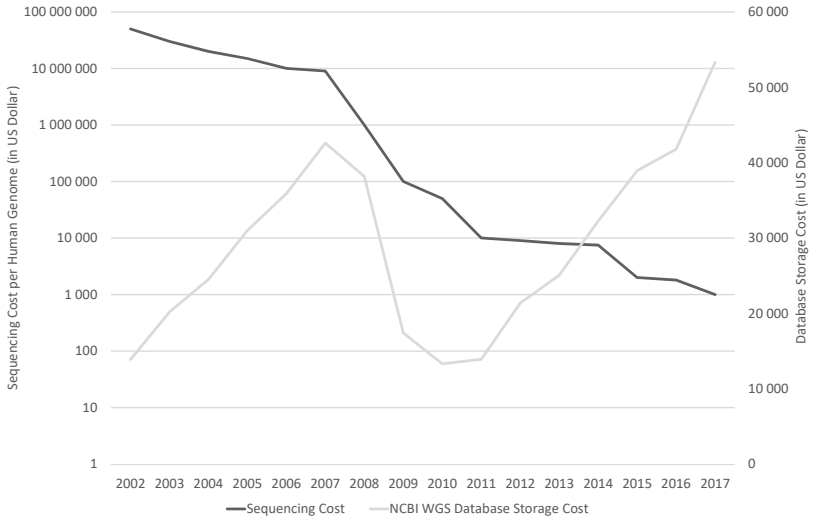

Figure 1.14: Evolution of sequencing cost and the storage cost of the NCBI WGS database (2002-2017). Generated from data provided by [19], [20], and [21].

which grows more than exponentially, especially after 2010. To illustrate this, the total cost of this database (expressed in hard drive purchase cost, excluding operating and maintenance costs) is shown in Figure 1.14. This figure shows that, since 2010, i.e., when the sequencing cost dropped to around $10,000 per human genome, the storage cost for the complete database started rising in an almost exponential proportion, as confirmed by the observations by Stein *et al.* [22]. The cost peak in the period 2004-2007 is mainly caused by the combination of the stagnation of storage costs and the stable growth of the archive content, as shown in Figure 1.13. Based on the recent exponential growth of genomic data, it is to be expected that the storage cost will become a more significant part of the total cost of sequencing and handling genomic data. One possible way to lower the storage cost and the band width cost is to apply compression to the different types of genomic data. In the next section, the current solutions for the compression of genomic data will be discussed.

1.1.5 Compression of Genomic Data

In this section, the different existing approaches that are used for the compression of genomic data will be discussed: compression algorithms that are specifically designed for the compression of nucleotidic data, for the compression of quality scores, and generic algorithms that are applied to existing file formats, such as FASTA/FASTQ.
Before discussing the different compression algorithms, it is important to elaborate on the three dimensions along which compression algorithms can be compared.

1.1.5.1 The Three Dimensions of Compression

When comparing different compression algorithms, performance can be analysed across three different dimensions [23]:

- **Effectiveness** - the compression ratio offered by the algorithm/file format, i.e., how small is the storage footprint of the compressed file;
- **Efficiency** - what is the (de)compression complexity? How complex are the compression and decompression process, i.e., how fast can data be compressed and decompressed; and
- **Functionality** - what are the additional features offered by the algorithm/file format? For example, random access, support for file streaming, support for metadata, and encryption.

Ideally, a compression algorithm (for genomic data) offers a high effectiveness (allowing for lower storage costs and faster transmission), is highly efficient (hundreds of gigabytes of genomic data need to be processed), and offers additional functionality (random access to limit data exchange and decompression to the required data, encryption to protect privacy,...). Depending on the application, priorities can be set to each of these dimensions. For example, in case of archiving, the priority will be more on effectiveness (storage cost), than on efficiency (speed). However, in case of real-time transmission (transmission during sequencing), a technology that has only been enabled recently by sequencers such as the Oxford Nanopore minION (See Section 1.1.2) with their support for sequencing data provisioning during the sequencing process, the priority will be more on efficiency. This real-time sequencing and transmission can be powerful when e.g., studying the outbreak of diseases or food contamination with central data analysis (e.g., Pulsenet [24]). In these cases efficiency will be of a higher importance. Given this wide range of possible applications, it would be beneficial to have compression algorithms that offer the possibility to select a trade-off between efficiency and effectiveness.
Additionally, applications that only require specific parts of genomic data, can

benefit from features such as random access. Given random access, only the parts of interest have to be stored or transmitted, and decoded.

One- or Two-Pass Besides these three dimensions, one can identify another property to distinguish compressors: the use (or not) of a pre-processing step. This pre-processing step is used to reorder individual reads and/or group them into bins which are consequently compressed using, e.g., dictionary-based encoding. Examples of this approach are SCALCE [26] and ORCOM [27]. These two-pass processes are typically more effective when compared to one-pass solutions, as they can analyse the input data, sort the input data (which is a highly complex process), and optimize the compression algorithm. However, if data need to be available immediately (e.g., for streaming purposes) or the available hardware is limited (e.g., for storing temporary files or dictionaries), such dual-pass solutions cannot be applied.

1.1.5.2 Compression of Nucleotidic Data

In general, compression algorithms for nucleotidic information (i.e., the second line in FASTA/FASTQ files, as discussed in Section 1.1.4) can be split into five categories [25]:

- **Bit Encoding** - nucleotides are stored as a sequence of two bits per nucleotide (in case of the 4-symbol ACGT alphabet), a sequence of three nucleotides per byte (when the N symbol is used, which signals an uncertain base call), or as a sequence of four bits per nucleotide (when the complete IUB/IUPAC alphabet is used), instead of the eight-bit ASCII format [28];
- **Dictionary-based Encoding** - groups of nucleotides are stored as a reference to entries in a dictionary. Well-known examples of this category of compressors are the Lempel-Ziv-based algorithms, such as LZ77 and LZ78 [29], and DSRC [30];
- **Statistical Encoding** - nucleotides are predicted based on a probabilistic model. A well-known and widely used example of this category of compressors is Huffman encoding [31]. Another type of statistical encoding algorithms is based on hidden Markov models [32];
- **Reference-based Encoding** - groups of nucleotides are stored as a pointer to a position in a reference genome. Reference-based compression algorithms, such as CRAM [33], can be highly effective for storing a genome, provided that a good reference (i.e., of a similar species) is selected. However, finding long identical matches between the genome and the reference genome can be complex. Additionally, the (correct version of the) reference genome has to be stored and/or transmitted to enable decoding.

- **Neural Networks** - more recently, another category of compressors is being proposed, based upon machine learning [34] [35] [36]. In particular, these algorithms use neural networks, such as auto encoders, to predict the sequence of nucleotides and store the parameters for the neural network, together with a correction (i.e., the residue).

The encoding framework presented in Chapter 2 and the AFRESh compression solution presented in Chapter 3 offer a novel hybrid one-pass solution, where coding tools of different categories compete and the parameters and output data are stored using statistical compression. Furthermore, the input data are processed in independent blocks, allowing for random access, and thus for streaming and parallel processing.

1.1.5.3 Compression of Quality Scores

Besides the division into the categories discussed in Section 1.1.5.2, solutions for the compression of quality scores (i.e., the fourth line in FASTQ files, as discussed in Section 1.1.4) can additionally be divided into categories along a second dimension: lossy compression versus lossless compression (i.e., the categories discussed for the compression of nucleotidic data). Recently, it has been shown that lossy compression can maintain, and in some cases even improve, the performance of variant calling algorithms[14] [37] [38]. Based on this observation, and the promise of significant gains in compression effectiveness, the focus in the area of quality score compression is moving towards lossy compression. Lossy quality score compression algorithms are typically based on quantization. Two types of quantization can be identified:

- **Fixed Quantization** - Each of the values within a range are mapped onto one single value. Both the number of quantized values and the mapping are fixed. An example of this approach is Illumina 8-level binning [39], which divides the quality score range (which is typically 94 values wide) into eight separate (fixed) ranges, which can be identified using a 3-bits code word; and
- **Adaptive Quantization** - Each of the values within a range are mapped onto one single value. The number of quantized values and/or the mapping are defined adaptively. Two different types can currently be identified:
  - **Position-based Adaptive Quantization** - For each position in the read, a set of quantizers is defined. An example of this approach is Quality

[14] I.e., algorithms that identify variants (mutations) of a base on a given position, indicating e.g., sensitivity for certain diseases.

Value ZIP (QVZ) [40], which calculates for each position a set of quantizers that is defined in such a way that it minimizes the distortion for the quality scores at the position in the read. Optionally, QVZ allows to pre-process the input data and to group the input data in clusters, according to the Euclidean distance of the quality values of each read compared to a predefined number of randomly selected reads; and

– **Locus-based Adaptive Quantization** - For each position in the genome (i.e., locus), a set of quantizers is defined. An example of this approach is Coverage-Adaptive Lossy Quality value compression (CALQ) [41], which introduced the concept of locus-based quantization, where the number of quantization steps is defined based upon the ”genotype uncertainty” of a certain locus in the genome.

1.1.5.4 Generic Compression

Despite the availability of specialized compression solutions, generic compression algorithms such as LZ77 (e.g., in Gzip) are widely applied for the compression of genomic data. The efficiency, acceptable effectiveness, and wide availability of Gzip makes it even the de facto industry standard for the compression of FASTA/-FASTQ files.
Applying this type of generic compressors to the example data file NA12878, used in Section 1.1.4 as an indicator for file sizes for FASTA/FASTQ and SAM/BAM file formats, can result in a significant file size reduction. When using the Gzip compressor, the FASTQ file is reduced from 368.6 GiB to 82.5 GiB. In case of the SAM file, the file is reduced from 548.6 GiB to 103.5 GiB. In the results discussed in Chapter 3, Chapter 4, and Chapter 5, a second generic compression algorithm is used for comparison purposes: LZMA. LZMA and LZMA2 are highly effective variants of LZ77, used in 7-Zip. These algorithms trade efficiency[15] and memory usage[16] for a higher effectiveness. Using LZMA/LZMA2 in ultra settings provides an additional file size reduction of approximately 20%, resulting in file sizes of 67.5 GiB for the FASTQ file and 82.5 GiB for the SAM file.
Unfortunately, the functionality offered by these generic algorithms is typically limited: there is no or limited support for random access and all different data streams (e.g., quality scores, read names, and nucleotides) are compressed as one input stream.

[15] LZMA and LZMA2 ar approximately 10 times slower than Gzip, when configured in ultra mode.
[16] LZMA, limited to two processing threads, uses a maximum of 709 MiB of RAM. LZMA2, which supports more than 2 processing threads, uses a maximum of 4 GiB of RAM. Gzip only requires 4 MiB of RAM.

1.1.6 Exchange of Genomic Data

A second part of data management is data exchange: how do we transmit data from one point (e.g., a sequencing machine, a database, or computer storage) to another (e.g., another database, an analysis center, or a cloud service)? Currently, two types of data exchange are competing for the transmission of large data sets: transmission over computer networks and transport via courier. It is clear that for smaller files or even one human genome, transmission of the data over a high-bandwidth network is the fastest solution.
Table 1.2 shows the transmission durations for a single human genome (NA12878) and a set of five such genomes in both BAM and FASTQ file format across multiple network connections offering different band widths. The table shows that transmit-

	10Mbit/s	25Mbit/s	100Mbit/s	1Gbit/s
NA12878-BAM (113.3GiB)	1 day	10 hours	3 hours	15 minutes
NA12878-FASTQ (368.6GiB)	3 days	1 day	8 hours	49 minutes
5 x NA12878-BAM	5 days	2 days	13 hours	1 hour
5 x NA12878-FASTQ	17 days	7 days	2 days	4 hours

Table 1.2: Total transmission time for four different data sets with respect to network band width.

ting the BAM file over a 25 Mbit/s network connection will take around 10 hours (hence, faster than the typical 24-hour courier service). However, only 12% of the broadband connections in the world exceed this 25Mbit/s (actual) band width [42]. Moreover, data are rarely limited to a single file. In case of wider studies, such as genome-wide association studies (GWAS), it can be required to exchange multiple (up to hundreds) of human genomes. As shown in Table 1.2, transmission of this relatively small set of five human genomes can already take days, even when using relatively fast network connections of 25-100 Mbit/s. Therefore, the transmission via courier is still a very popular option. As Andrew S. Tanenbaum once concluded:

> *"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."*
> Andrew S. Tanenbaum

Indeed, sending a state-of-the-art 14TB hard drive (i.e., 14,000,000,000,000 bytes) filled with data using a 24-hour courier service (commonly called Sneakernet) equals to a bandwidth of 1,236 Mbit/s[17]. Large cloud operators, such as

[17] If this hard drive would be replaced by microSD cards of 400 GiB, more than 2,360 cards would fit into the same volume, which would result in a maximum storage capacity of 945 TiB and an equivalent band width of around 83 Gbit/s. Of course, given the price of around $250 per card, the total hardware cost would be around $590,000.

Google and Microsoft, offer solutions to transmit data to their cloud services this way [43].

If latency is not an issue, it is clear that Sneakernet is a valid option to transmit large amounts of data. However, if low latency is important, other solutions such as (more) effective compression[18] with support for random access (for partial file transmission) and/or faster networks are the only solutions.

1.1.7 Standardisation

As discussed in Section 1.1.4, genomic data are currently stored using the de facto standards FASTA/FASTQ and SAM/BAM. This (de facto) standardisation enables interoperability: it ensures that software and hardware solutions that are designed to be standard-compliant can process all data stored in a standard-compliant bitstream. As a result, standardisation enables researchers, doctors and institutions to exchange data in a manner that is understood by all.
Examples of these positive effects of standardisation are seen with the H.264/AVC [44] and H.265/HEVC [45] video coding standards by the Moving Picture Experts Group (MPEG). As with other MPEG standards, the specification of H.264/AVC and H.265/HEVC describes the syntax and processes used by the decoder to access (and decode) the content of each standard-compliant bitstream. This approach allows encoder developers to design (and constantly improve) their own encoding algorithms according to their own priorities, while the output can still be decoded with any standard-compliant decoder. In case of H.264/AVC and H.265/HEVC, this resulted in a large diversity of encoders, each with their own focus, priorities, and/or strengths.
Currently, the MPEG standardisation committee is finalizing such a standard for the representation, compression, and management of genomic data. In the next section, an overview will be provided of the evolution of this standardization process, combined with the evolution of the research discussed in this dissertation.

[18] More effective compression algorithms can be applied to both Sneakernet and network transmissions. Therefore, it will move the switching point between these two types towards larger data set sizes.

03-2014 *Genome Sequences as Media Files.* Biostec, 2014	
	07-2014 *Issues in Genome Compression and Storage.*
12-2014 *Towards Block-Based Compression of Genomic Data with Random Access Functionality.* Globalsip, 2014	
	06-2015 *Requirements on Genome Compression and Storage.*
	10-2015 *Call for Evidence for Genome Compression and Storage.*
03-2016 *Leveraging CABAC for No-Reference Compression of Genomic Data with Random Access Support.* DCC, 2016	
	06-2016 *Joint Call for Proposals for Genome Compression and Storage.*
01-2017 *AFRESh : an adaptive framework for compression of reads and assembled sequences with random access functionality.* Bioinformatics, 2017	
09-2017 *AQUa: an adaptive framework for compression of sequencing quality scores with random access functionality.* Bioinformatics, 2018	
	01-2018 *ISO/IEC CD 23092-2 Coding of Genomic Information.*

Table 1.3: Combined timeline of the research described in this dissertation and MPEG-G standardisation.

1.1.8 Research & Standardisation Timeline

Figure 1.3 shows the timeline of the research described in this dissertation (left side), combined with the timeline of the standardization efforts by MPEG (right side). In March 2014, a first publication on the compression of genomic data was published. This paper discussed the vision on how technologies for media compression, distribution, and management can be useful for the compression, distribution, and management of genomic data. Additionally, the paper discussed the positive effects of standardisation on acceptance and, as a result, on the interest in and opportunities for research and continuous improvement.
A few months later, MPEG published its first document describing the issues in genome compression and storage. This document discussed the issues with current genomic data file formats and the requirements needed to solve these issues, confirming the high-level requirements that were listed in my first publication, such as random access, encryption, and privacy protection.
In December 2014, the first results were published on the development and the coding effectiveness of the framework that is discussed in Chapter 2 of this dissertation. These results demonstrated that the block-based approach with a set of competing coding tools was a promising approach.
Early 2016, the results were published of the integration of CABAC in the framework. Compression gains of up to 70% were shown, compared to the earlier version of the framework, without CABAC. During the integration of CABAC into the framework, MPEG published an overview of the identified requirements for genome compression and storage, followed by a Call for Evidence (CfE) and a joint Call for Proposals (CfP). In a response to this CfP, the solutions discussed in Chapter 3 and Chapter 4 of this dissertation were presented as possible solutions for the compression of genomic data.
Based upon the requirements and the solutions proposed during the CfP, the concept of descriptor streams, each containing a single type of data (e.g., quality scores, mapping position, or pairing information), had been proposed and accepted. Given this heterogeneous set of descriptor streams, a need was identified for a flexible, efficient and effective compression solution. As a response to this need, the solution discussed in Chapter 5 was developed and presented. The syntax and concepts of this solution have been selected to form the baseline of the coding part of MPEG-G. This syntax and the concepts have been further extended and adapted and finally lead to the current version of the MPEG-G standard which is described in the Committee Draft (CD) of the standard.

1.2 Outline

This dissertation is organized as follows:

- Chapter 2 discusses the novel coding framework that has been built as a basis for the coding solutions presented in the successing chapters.
- Chapter 3 discusses the extensions that have been designed for the coding framework to enable effective coding of nucleotidic data (both reads and full genomes) and analyses the compression performance.
- Chapter 4 discusses the extensions that have been designed for the coding framework to enable effective coding of quality scores and analyses the compression performance.
- Chapter 5 discusses the solution that has been proposed as a compression solution for the MPEG-G standard and analyses the compression performance.
- Chapter 6 offers an overview of the different conclusions and discusses directions for future research.

1.3 Publications

The research discussed in this dissertation resulted in 5 international publications, of which 2 are A1 publications, 13 are MPEG input documents, and 1 is a draft standard specification document (as first author and co-author) for the MPEG-G standard.
Besides this research, 9 international publications, of which 3 are A1 publications, were published on video coding technologies (as first author and co-author).

1.3.1 Publications in International Journals

1.3.1.1 Author

AFRESh : an adaptive framework for compression of reads and assembled sequences with random access functionality.
T. Paridaens, G. Van Wallendael, W. De Neve, and P. Lambert
Bioinformatics, Vol. 33, No. 10, pp. 1464-1472, 2017.

AQUa: an adaptive framework for compression of sequencing quality scores with random access functionality.
T. Paridaens, G. Van Wallendael, W. De Neve, and P. Lambert
Bioinformatics, Vol. 34, No. 3, pp. 425-433, 2018.

1.3.1.2 Co-author

Simultaneous encoder for high-dynamic-range and low-dynamic-range video.
J. De Praeter, A. Jesus Diaz-Honrubia, T. Paridaens, G. Van Wallendael, and P. Lambert
IEEE Transactions on Consumer Electronics, Vol. 62, No. 4, pp. 420-428, 2016.

NinSuna: a fully integrated platform for format-independent multimedia content adaptation and delivery using Semantic Web technologies.
D. Van Deursen, W. Van Lancker, W. De Neve, T. Paridaens, E. Mannens and R. Van de Walle
Multimedia Tools and Applications, Vol. 46, No. 2-3, pp. 371-398, 2010.

Moving object detection in the H.264/AVC compressed domain for video surveillance applications.
C. Poppe, S. De Bruyne, T. Paridaens, P. Lambert and R. Van de Walle
Journal of Visual Communication and Image Representation, Vol. 20, No. 6, pp. 428-437, 2009.

1.3.2 Publications in International Conferences

1.3.2.1 Author

Leveraging CABAC for no-reference compression of genomic data with random access support.
T. Paridaens, J. Panneel, G. Van Wallendael, W. De Neve, P. Lambert and R. Van de Walle
Data Compression Conference (DCC), 2016.

Towards block-based compression of genomic data with random access functionality.
T. Paridaens, Y. Van Stappen, W. De Neve, P. Lambert and R. Van de Walle
IEEE Global Conference on Signal and Information Processing (Globalsip), 2014.

Genome sequences as media files.
T. Paridaens, W. De Neve, P. Lambert and R. Van de Walle
8th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC), 2014.

XML-driven bitrate adaptation of SVC bitstreams.
T. Paridaens, D. De Schrijver, W. De Neve and R. Van de Walle
Workshop on Image & Audio Analysis for Multimedia Interactive Services (WIAMIS), 2007.

1.3.2.2 Co-author

A Just Noticeable Difference Subjective Test for High Dynamic Range Images.
A. Ahar, S. Mahmoudpour, G. Van Wallendael, T. Paridaens, P. Lambert and P. Schelkens
Qomex, 2018.

Multistream video encoder for generating multiple dynamic range bitstreams.
C. Van Goethem, J. De Praeter, T. Paridaens, G. Van Wallendael and P. Lambert
Picture Coding Symposium(PCS), 2016.

Perceptual quality of 4K-resolution video content compared to HD.
G. Van Wallendael, P. Coppens, T. Paridaens, N. Van Kets, W. Van den Broeck and P. Lambert
Eighth International Conference on Quality of Multimedia Experience (Qomex), 2016.

Statistical multiplexing using SVC.
M. Jacobs, J. Barbarien, S. Tondeur, R. Van de Walle, T. Paridaens and P. Schelkens
IEEE International Symposium on Broadband Multimedia Systems and Broadcasting, 2008.

NinSuna: a format-independent multimedia content adaptation platform based on semantic web technologies.
D. Van Deursen, W. Van Lancker, T. Paridaens, W. De Neve, E. Mannens and R. Van de Walle
IEEE International Symposium on Multimedia (ISM), 2008.

1.3.3 MPEG Input Contributions

1.3.3.1 Author

Proposal of a genomic data file compression framework, based on existing MPEG practices and technologies (m39200).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 116, 2016

CE1: Cross-check of the PirBright Genomic Data Compression proposal (m39875).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 117, 2017

CE2: Cross-check of the PirBright Genomic Data Compression proposal (m39876).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 117, 2017

CE1 & CE2: Proposal for a genomic data file compression framework, based on existing MPEG practices and technologies - update (m39881).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 117, 2017

CE5: Results of syntax compression based on CABAC (m40272).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 118, 2017

Summary of Core Experiment 5 on Entropy Coding (m40850).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 119, 2017

Core Experiment 5 on Genomic Information Representation results (m40851).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 119, 2017

Proposal for decoding process of MPEG-G descriptor streams (m41595).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 120, 2017

Study on ISO/IEC 23092-1 (m42003).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 121, 2018

Study on ISO/IEC 23092-2 (m42004).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 121, 2018

Study on ISO/IEC 23092-2 (m42549).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 122, 2018

Overview of coding modes (m42550).
T. Paridaens, G. Van Wallendael, W. De Neve and P. Lambert
MPEG 122, 2018

1.3.3.2 Co-author

Unified representation of sequencing quality values (m40222).
J. Voges, C. Alberti, M. Hernaez, T. Paridaens, J. Bonfield, P. Ribeca, J. Delgado
MPEG 118, 2017

1.3.4 MPEG Standardization Documents

ISO/IEC 23092-2: Coding of Genomic Information.
C. Alberti, J. Voges, J. Ahmad, T. Paridaens
2018

References

[1] J. D. Watson et al., "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid", Nature, Vol. 171, No. 4356, pp. 737-738, 1953.

[2] NIH, "Genetics by the Numbers", https://publications.nigms.nih.gov/insidelifescience/genetics-numbers.html

[3] F. Sanger et al., "DNA sequencing with chain-terminating inhibitors", Proceedings of the National Academy of Sciences of the United States of America, Vol. 74, No. 12, pp. 5463-5467, 1977.

[4] R. Wu, "Nucleotide sequence analysis of DNA: I. Partial sequence of the cohesive ends of bacteriophage and 186 DNA", Journal of Molecular Biology, Vol. 51, No. 3, pp. 501-521, 1970.

[5] A.M. Maxam et al., "A new method for sequencing DNA", Proceedings of the National Academy of Sciences of the United States of America, Vol. 74, No. 2, 1977.

[6] B. Alberts et al., "Isolating, Cloning, and Sequencing DNA", Molecular Biology of the Cel 4th edition, Garland Science, 2002.

[7] NIH, "The Cost of Sequencing a Human Genome", https://www.genome.gov/27565109/the-cost-of-sequencing-a-human-genome/

[8] K. Mullis et al., "Specific enzymatic amplification of DNA in vitro: the polymerase chain reaction", Cold Spring Harbor Symposia on Quantitative Biology, Vol. 51, pp. 263-273, 1986.

[9] "Illumina Hiseq X sequencing system", https://www.illumina.com/systems/hiseq-x-sequencing-system.html, 2014.

[10] "Scalability for sequencing like never before", https://www.illumina.com/systems/sequencing-platforms/novaseq/specifications.html, 2017.

[11] "Human Whole-Genome Sequencing with the NovaSeq 6000 Sequencing", https://support.illumina.com/content/dam/illumina-marketing/documents/products/appnotes/novaseq-hiseq-q30-app-note-770-2017-010.pdf, 2017.

[12] "SMRT sequencing: consensus accuracy", http://www.pacb.com/smrt-science/smrt-sequencing/accuracy/, 2017.

[13] "Illumina unveils novaseq 5000 and 6000", https://blog.genohub.com/2017/01/10/illumina-unveils-novaseq-5000-and-6000/, 2017.

[14] M. Jain, "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community", Genome Biology, Vol. 17, No. 1, pp. 239-249, 2016.

[15] "Nanopore product specifications comparison", https://nanoporetech.com/products#comparison, 2017.

[16] P. J. A. Cock et al., "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants", Nucleic Acids Research, Vol. 38, No. 6, pp. 1767-1771, 2010.

[17] H. Li et al., "The Sequence Alignment/Map format and SAMtools", Bioinformatics, Vol. 25, No. 16, pp. 2078-2019, 2009.

[18] "NovaSeq 6000 System Quality Scores and RTA3 Software", https://www.illumina.com/content/dam/illumina-marketing/documents/products/appnotes/novaseq-hiseq-q30-app-note-770-2017-010.pdf

[19] Matthew Komorwski, "A History of Storage Cost", http://www.mkomo.com/cost-per-gigabyte and http://www.mkomo.com/cost-per-gigabyte-update

[20] Backblaze, "Hard Drive Cost Per Gigabyte",https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/

[21] NCBI, "GenBank and WGS Statistics", https://www.ncbi.nlm.nih.gov/genbank/statistics/

[22] L. D. Stein, "The case for cloud computing in genome informatics", Genome Biology, Vol. 11, No. 5, pp. 207, 2010

[23] T. Paridaens et al., "Genome sequences as media files", 8th International joint conference on biomedical engineering systems and technologies, Proceedings, 2014.

[24] "Pulsenet", https://www.cdc.gov/pulsenet/index.html

[25] S. Wandelt, M. Bux, and U. Leser, "Trends in Genome Compression", CBIO Current Bioinformatics, vol. 9, no. 3, pp. 315-326, 2014.

[26] F. Hach et al., "SCALCE: boosting sequence compression algorithms using locally consistent encoding", Bioinformatics, Vol. 28, No. 23, pp. 3051-3057, 2012.

[27] S. Grabowski, "Disk-based genome sequencing data compression", Bioinformatics, Vol. 31, No. 9, pp. 1389-95, 2015.

[28] S. Grumbach et al., "A new challenge for compression algorithms: genetic sequences", Information Processing & Management, Vol. 30, No. 6, pp. 875-886,1994.

[29] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343, 1977.

[30] S. Deorowicz et al., "Compression of DNA sequence reads in FASTQ format", Bioinformatics, Vol. 27, No. 6, pp. 860-862, 2011.

[31] D. Huffman, "A method for the construction of minimum-redundancy codes", Proceedings of the Institute of Radio Engineers, Vol. 40, No. 9, pp. 1098-1101, 1952.

[32] G. Cormack et al., "Data compression using dynamic markov modelling", The Computer Journal, Vol. 30, No. 6, pp. 541-550, 1987.

[33] M. Fritz et al., "Efficient storage of high throughput DNA sequencing data using reference-based compression", Cold Spring Harbor Laboratory Press, 2011.

[34] M. Duarte et al., "Bacterial DNA Sequence compression models using artificial neural networks", Entropy, Vol. 15, No. 9, pp. 3435-3448, 2013.

[35] M. Hinderyckx, "Non-reference-based DNA sequence compression using machine learning techniques", UGent, 2016.

[36] T. Mortier, "Non-reference-based DNA read compression using machine learning techniques", UGent, 2016.

[37] C. Kozanitis et al., "Compressing Genomic Sequence Fragments Using SlimGene.", Journal of Computational Biology, vol. 18, no. 3, pp. 401-413, 2011.

[38] I. Ochoa et al., "Effect of lossy compression of quality scores on variant calling", Briefings in Bioinformatics, Vol. 18, no. 2, pp. 183-194, 2017.

[39] "Reducing Whole-Genome Data Storage Footprint", http://www.illumina.com/documents/products/whitepapers/whitepaper_datacompression.pdf

[40] G. Malysa et al., "QVZ: Lossy compression of quality values", Bioinformatics, Vol. 31, No. 19, pp. 3122-3129, 2015.

[41] J. Voges et al., "CALQ: compression of quality values of aligned sequencing data", Bioinformatics, to be published. DOI:10.1093/bioinformatics/btx737.

[42] "Akamai state of the internet report Q1 2017", https://www.akamai.com/us/en/about/our-thinking/state-of-the-internet-report/, 2017

[43] "Introducing Transfer Appliance: Sneakernet for the cloud era.", https://cloudplatform.googleblog.com/2017/07/introducing-Transfer-Appliance-Sneakernet-for-the-cloud-era.html

[44] T. Wiegand et al., "Overview of the H.264/AVC Video Coding Standard", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 7, pp. 560-576, 2003.

[45] G. J. Sullivan et al., "Overview of the High Efficiency Video Coding (HEVC) Standard", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 22, No. 12, pp. 1649-1668, 2012.

2 Coding Framework

In this chapter, the novel coding framework will be presented that has been designed as the foundation for the coding solutions for nucleotidic data (AFRESh, see Chapter 3) and quality scores (AQUa, see Chapter 4).
First, the main features of the coding framework are outlined, followed by a discussion of the different steps in the processing workflow. Then, the different forms of flexibility and configurability, provided by the coding framework, are detailed. In the final part of this section, the main concepts of Context-Adaptive Binary Arithmetic Coding (CABAC) are explained to provide the reader with the knowledge required to understand the content of the succeeding chapters.

2.1 Features

In this section, an overview is provided of the key features of the coding framework. These features are selected based on observations on how large media files are handled, which applications are currently important for genomic data, and which applications will become important. During the design of this coding framework, special attention has been paid to flexibility, allowing for easy setup and extensibility. While some of the choices will negatively affect efficiency, this is deemed as less important for research usage.

The key features of the presented coding framework are:

- **single-pass encoding** - input data can be processed immediately, without

the need for an analysis or pre-processing pass, hence allowing for live encoding;

- **stand-alone, no-reference encoding** - input data are coded without any external dependencies, hence voiding the need for reference management, storage capacity for the reference, and limiting encoding complexity. This approach is followed by many popular genome data banks such as the DNA Data Bank of Japan (ftp.ddbj.nig.ac.jp), NCBI (www.ncbi.nlm.nih.gov) and EBI (www.ebi.ac.uk). These data banks provide all data available in non-reference file formats such as FASTQ and BAM[1], while only a subset of the data is available in reference-based formats, such as CRAM [1];
- **random access in combination with arithmetic coding** - it is impossible to discern individual symbols in an arithmetically coded bit stream, hence technologies have been integrated into the framework to allow for random access support in combination with arithmetic coding;
- **flexible configuration of the coding tools and random access parameters** - the coding framework allows to set parameters for specific coding tools (e.g., block size[2], search window sizes[3]) and for the coding framework (e.g., random access block sizes[4]);
- **flexible configuration of coding complexity and effectiveness** - the coding framework offers the possibility to select (and configure) a set of coding tools to choose a bias between efficiency and effectiveness; and
- **extensibility** - the coding framework has been designed to provide easy extensibility with:
  - input file formats;
  - symbol alphabets;
  - coding tools; and
  - output file formats.

2.2 Workflow

The coding framework processes the input data as a continuous stream of symbols (i.e., single-pass encoding), without any reference to external data (i.e., no-reference encoding), thus following an approach that is similar to the approach

[1]BAM files can contain links to references but are stored as stand-alone files, i.e., not using a reference.

[2]The number of input values that are processed as one entity (block) by the coding framework.

[3]The number of blocks that is available for reference during encoding.

[4]The number of blocks that is contained within one random access block.

typically used in the area of video compression [2] [3]. Figure 2.1 shows the different steps used by the coding framework:

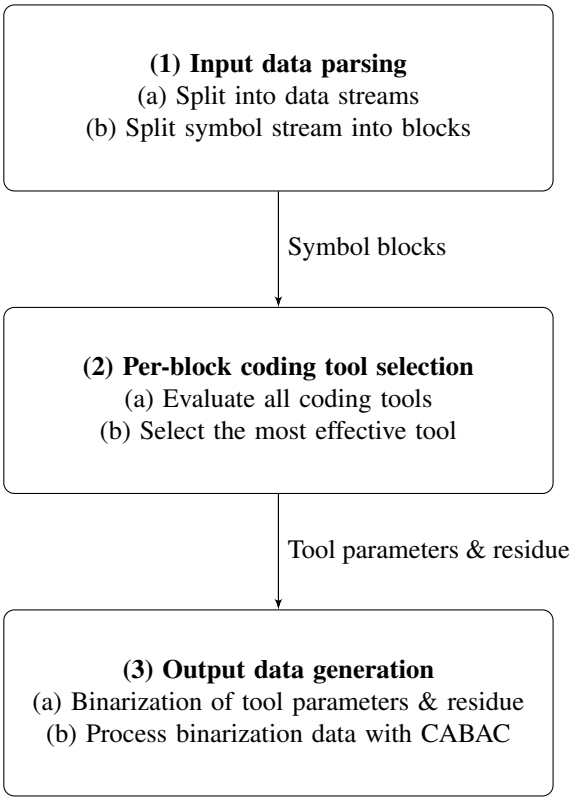

Figure 2.1: The different steps used by the coding framework.

1. The input data (in FASTA or FASTQ format) are parsed and split into two data streams: a data stream of symbols (nucleotides or quality scores) and a data stream containing the data that are not processed by the coding framework (i.e., read names, extended with quality scores, in case of AFRESh, or nucleotides, in case of AQUa) (a). The data stream containing the metadata is stored separately; the symbol stream is split into blocks of a fixed length (b).

2. A set of prediction and encoding tools is used to process each block (a). The tool with the highest effectiveness is selected (b). The effectiveness is defined by performing step 3, without writing the resulting data to disk, and resetting the arithmetic encoder to the state before this test.

3. The tool identifier, parameters, and residual data are then converted to an optimized binary representation (a) which is then processed by the CABAC arithmetic coder (b). The output of CABAC is subsequently written to disk.

2.3 Framework Flexibility

The presented coding framework has been designed with two forms of flexibility in mind:

- extensibility, i.e., adding new capabilities such as:
  - **new input file formats** - a new input filter can be added which takes the new file format as an input and provides the data that need to be compressed (e.g., nucleotides or quality scores) as an output;
  - **new symbol alphabets** - if an input file format supports symbols that are not part of one of the existing symbol alphabets, then the set of symbol alphabets needs to be extended with a new symbol alphabet. This alphabet consists of a list of symbols and, if required, their complements. In case of nucleotides, these are the real complements; in case of quality scores, no complements are set as quality scores do not have complementary values/symbols;
  - **new coding tools** - if a new input file format generates new types of data (e.g., with a new alphabet, or data differing from nucleotides or quality scores), the set of coding tools can be extended with new coding tools that offer a more effective compression. A coding tool generates a prediction and returns this prediction, together with a list of parameters that are needed for decoding;
  - **new output file formats** - a new output filter can be added which takes all data generated by the coding tools (e.g., alphabet identification, coding tool parameters, residue), encodes these data, and writes the output of the encoding process to a file or transmits the output over a network.
- configurability, i.e., selection of:
  - **used symbol alphabets** - selecting the symbol alphabet(s) that can occur in the input data stream;
  - **used coding tools** - selecting the (set of) coding tool(s) that will compete during the coding tool selection. A smaller list of coding tools will lower the compression effectiveness but will typically improve the compression efficiency;

– **configuration parameters for coding tools** - e.g., the window size used by some coding tools. A larger window size will lower the compression efficiency but will typically improve the compression effectiveness;

– **configuration parameters for the coding framework** - e.g., the size of random access blocks, block sizes (i.e., the size of the segments that are created by the input data parsing step as discussed in Section 2.2).

2.4 Block Structure

The data that is created during step 2 of the workflow discussed in Section 2.2 is represented by a data structure, consisting of up to 5 parts (see Figure 2.2):

- **Last Block Flag** - a flag that indicates if the current block is the last block in the data stream. If the Last Block Flag is true, the size of this block is signaled;
- **Alphabet ID** - an ID that signals the alphabet that is used during the encoding of this block;
- **Coding Tool ID** - an ID that signals the predictor that has been used during the encoding of this block;
- **Coding Tool Parameters** - this data structure signals the configuration parameters for the used coding tool. In case of encoding tools, this structure contains the encoded values;
- **Residual Data (only for prediction tools)** - this data structure contains the residual data, used to correct the output of a prediction tool.

Figure 2.2: Data structure of a block.

Each of these parts is then separately processed in step 3, where all data are converted to a binary representation which can than be encoded using CABAC, which is discussed in Section 2.5.

2.5 Context-Adaptive Binary Arithmetic Coding

In this section, the basic concepts of Context-Adaptive Binary Arithmetic Coding (CABAC) will be discussed. CABAC is an entropy coder that is used in modern standards for video compression, such as High-Efficiency Video Coding (HEVC) [4] [5], to compress the different syntax elements that are generated by the encoder. In the coding framework that is discussed in the previous section, CABAC is applied to the syntax elements that form the output of the encoder (i.e., the output of step 3.a). CABAC is a core element of all work discussed in this dissertation, hence it is important that the reader understands the high-level concepts of CABAC. As the name implies, CABAC is an **arithmetic coder** for **binary** values that **adapts** the probabilities that are represented in **contexts**. The meaning of each highlighted concept is discussed below:

Binary Arithmetic Coder Arithmetic coding is a form of lossless compression that represents a set of symbols (in case of a binary arithmetic coder, the set of symbols is $\{0,1\}$) into a single number [6]. To define this output number, and as such the compressed output, the arithmetic coder partitions the range [0.0,1.0] into subintervals according to the probability distribution of the different symbols. The top of Figure 2.3 shows an example of this partitioning for a probability of 0.8 (i.e., 80%) for 0 and 0.2 (i.e., 20%) for 1, resulting in two ranges: [0.0,0.8[and [0.8,1.0].

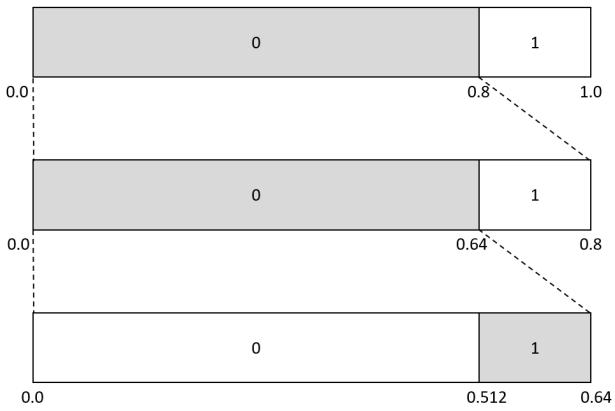

Figure 2.3: Encoding the binary sequence 001 using Binary Arithmetic Coding.

To encode a symbol, the arithmetic coder retains the partition that corresponds to the symbol to be encoded. If additional symbols are to be processed, the resulting partition is further partitioned into intervals, according to the probability distribution. This process is performed iteratively, until the set of symbols has been

processed.
Figure 2.3 shows an example of the encoding of the sequence of symbols '0 0 1' with a fixed probability distribution of 0.8 for '0' (P_0) and 0.2 for '1' (P_1) for each of the input symbols[5]. In the first step, the interval representing the first symbol ('0') is selected (i.e., [0.0,P_0[or [0.0,0.8[). To encode the second symbol, this interval is further partitioned, using the same probability distribution, resulting in the intervals [0.0,$P_0 * P_0$[and [$P_0 * P_0$,P_0] (i.e., [0.0,0.64[and [0.64,0.8[). In the second step, the interval representing the second symbol ('0') is selected ([0.0,0.64[) and further partitioned, resulting in the intervals [0.0,$P_0 * P_0 * P_0$[and [$P_0 * P_0 * P_0$,$P_0 * P_0$[(i.e., [0.0,0.512[and [0.512,0.64[). Finally, the interval representing the last symbol ('1') is selected ([0.512,0.64[). This range can then be represented by any single value[6], contained within this range: e.g., 0.5625.
As shown in the example in Figure 2.3, the encoding of each symbol (except for the first symbol) depends on the encoding of the set of previous symbols. Therefore, symbols cannot be discerned separately. This property is therefore not compatible with random access.
To decode the symbol sequence, the decoder will follow a similar approach as the encoder. In the first step, the decoder will partition the interval according to the probability distribution and select the interval in which the output value of the encoder is situated. Given the encoding example, the decoder will select the interval [0.0,0.8[as 0.5625 is contained within this range. The first output symbol will be the corresponding symbol '0'. In the second step, the decoder will further partition the interval [0.0,0.8[in the intervals [0.0,0.64[and [0.64,0.8[and again select the interval that contains the output value 0.5625, i.e., interval [0.0,0.64[, which corresponds to symbol '0'. In the third and final step, the decoder will further partition this interval into the intervals [0.0,0.512[and [0.512,0.64[. Given that 0.5625 falls within the second interval, corresponding to symbol '1', the output of the decoder will be '0 0 1'.

Adaptive An Adaptive Binary Arithmetic Coder is a binary arithmetic coder that is capable of adapting the probability distribution of the symbols. When a symbol has been encoded, the probability distribution is adapted to the new probability distribution (i.e., the probability of the coded symbol is increased, the probability of the other symbol is lowered). The process of probability distribution adaptation is often referred to as modeling. It is important that the adaptation of the probability distribution is performed after coding an input symbol as the decoder is not aware of this symbol before decoding it.
During the development of the coding solution in Chapter 5, some experiments were performed on adapting the speed of adaptation. The results of these adapta-

[5]Note that the sum of all probabilities P_i is 1.
[6]Provided that the length of the sequence is known.

tions were showing no significant gains in compression effectiveness (across the test set). Furthermore, adapting parts of the CABAC encoder would require existing CABAC implementations[7] to be adapted. Therefore, it was decided to not perform further research on this topic.

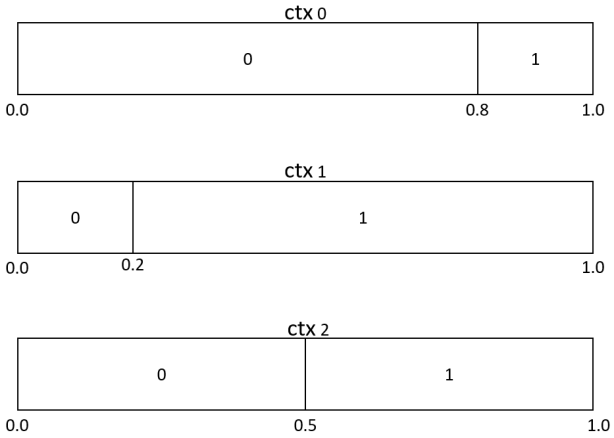

Figure 2.4: Examples of different contexts.

Context A Context-Adaptive Binary Arithmetic Coder is an adaptive binary arithmetic coder that allows to use multiple (adaptive) probability distributions, represented in so-called contexts. Figure 2.4 shows some examples of different contexts: context 0 (ctx 0) represents a probability of 80% for symbol '0', context 1 (ctx 1) represents a probability of 80% for symbol '1', context 2 (ctx 2) represents an equi-probability state i.e., each symbol has a probability of 50%. When encoding a symbol, exactly one context is used and adapted. The decision on which context is to be selected is performed on a higher level and can be based on any type of information that is available to both the encoder and the decoder (Section 2.5.2) provides more details on how this process is performed). In a binary arithmetic coder, each context can be represented by one value, e.g., the probability of '0'. The probability of the other symbol (in this case '1') can be calculated from the probability of the symbol: $probability(1) = 1.0 - probability(0)$.
In addition to the context-adaptive mode, CABAC offers a bypass mode. In this mode, CABAC assumes that the input symbols are equally distributed and hence codes these bins assuming a fixed probability distribution with equal probability for the input symbol to be 0 or 1.

[7] CABAC implementations are widely available in both hardware and software solutions. Therefore, CABAC was selected for the research described in this dissertation.

2.5.1 Binarization

CABAC can only be applied on a set of binary symbols, i.e., symbols can only be 0 or 1. Therefore, there is a need for transforming non-binary values to a sequence of binary symbols, called bins. These transformations are called binarizations. Besides the conversion to a binary string, it is important that a binarization is designed in such a way that each bin (i.e., a bit in the binarization output) is as predictable as possible as this improves the effectiveness of the arithmetic coder. Some binarization processes even create an output that is larger than the representation of the input symbol in binary coding. However, if the bins in these binarizations are more predictable, the output can be (significantly) smaller after arithmetic coding. An example of a binarization is Truncated Unary [4], which represents the non-binary value N by N 1-values followed by a 0. If N is equal to a defined maximum value `cMax`, the trailing 0-value is discarded. Table 2.1 shows the Truncated Unary binarization for values 0 to 3 with `cMax` equal to 3. In this case, the binary representation of value 0 is one bit shorter and the representation of values 2 and 3 is one bit longer than the standard binary representation, which would need two bits to represent each of the values.

value	binarization
0	0
1	10
2	110
3	111

Table 2.1: The Truncated Unary binarization for values 0 to 3 (`cMax=3`).

2.5.2 Context Selection

To encode a symbol, CABAC requires an expected probability distribution for this symbol. This probability distribution is stored in a context[8]. To identify the context that will be applied to the specific symbol, a process called ”context selection” is performed. This process selects the contexts based on information outside of the CABAC process. Examples of such information are:

- The position of the input symbol within a binarization;
- The previous input symbol;
- The previous input value to the binarization;
- The correctness of previous predictions.

[8] At the start of the CABAC coding process, each context is set to a predefined value. This process is called ”context initialization”.

It is advised to limit the number of contexts used for coding. A larger set of contexts will typically allow for a better adaptation to the exact probabilities of the input data but will require more memory to store these contexts and will slow down the adaptation process as less symbols will be coded using this context. Therefore, it is advised to use a specific context for multiple symbols which are (expected to be) distributed according to the same probability distribution. This will reduce memory requirements and speed up the adaptation process as more symbols are coded with a single context.

2.5.3 Random Access

As discussed in Section 2.5, the impact of CABAC on random access is defined by the concept of arithmetic coding; it is impossible to discern individual symbols or syntax elements in an arithmetically coded bit stream. Following this reasoning, the only point in the compressed bit stream where decompression can start, is at the very beginning.

To allow for random access, additional entry points are created (i.e., points where decompression can start), by resetting the complete status of the encoder (CABAC contexts, search window, ...) to its initial state every m blocks. That way, groups of CABAC-encoded blocks are created. Each group of blocks can be decoded separately, and as such provides a random access point. For ease of use, random access blocks are byte-aligned.

2.6 Conclusions and Original Contributions

In this section, a novel coding framework has been presented that can be used for the compression of genomic data and forms the foundation of the coding solutions discussed in Chapter 3 and Chapter 4. The coding framework processes the input data in a single pass and without any reference to external data. The coding framework consists of three steps: input parsing, coding tool selection, and data output. In the first step, data are read from an input file and split into blocks (a set of multiple input values, e.g., nucleotides or quality scores). In the second step, the framework tests a set of coding tools and selects the most effective coding tool. In the third step, the parameters for the selected coding tool and (if necessary) the residue are processed using Context-Adaptive Binary Arithmetic Coding (CABAC). All input data are processed in one pass and without reference to external data.

The coding framework has been designed to offer a flexible foundation for the coding of genomic data: the sets of input file formats, alphabets, coding tools, and output file formats can be extended easily without the need for adapting the core of the coding framework. Additionally, the trade-off between effectiveness and

efficiency can be selected by configuring the set of coding tools, and framework parameters (such as block size and search window size).
Finally, to enable random access in combination with arithmetic coding and coding tools that use previously encoded data, the concept of random access entry points has been introduced. At these points, the encoder status (i.e., CABAC contexts and the search window) is reset.

Binarization	A binary representation of a given symbol.
Block Size	The number of symbols stored in one block.
Coding Tool	An algorithm used to encode or predict the data contained in a block.
Context	A value that indicates the expected probability for a given symbol.
Effectiveness	The compression ratio provided by a coding solution. A higher effectiveness results in a smaller output.
Efficiency	The complexity of a coding solution. A higher efficiency results in faster encoding.
Random Access Block Size	The number of blocks contained in one Random Access Block.
Random Access Block	A unit of blocks, that can be decoded without additional information. At the beginning of a Random Access Block, the arithmetic coder is reset and the search window is cleared.
Residual Data	The data that contains the information required to correct a prediction.
Search Window	The set of blocks that can be used as a reference by coding tools.
Symbol Alphabet	The set of symbols that can occur in the input data stream.
(Search) Window Size	The number of blocks that are contained within the (sliding) search window.

References

[1] M. H.-Y. Fritz et al., "Efficient storage of high throughput DNA sequencing data using reference-based compression", Genome Research, Vol. 21, no. 5, pp. 734-740, 2011.

[2] M. Wien, "High efficiency video coding: coding tools and specification.", Berlin, DE: Springer, 2015.

[3] V. Sze et al., "High Efficiency Video Coding (HEVC): Algorithms and Architectures.", Basel, CH: Springer, 2014.

[4] D. Marpe et al., "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard", IEEE Trans. Circuits Syst. Video Technol. IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 620-636, 2003.

[5] V. Sze et al., "High Throughput CABAC Entropy Coding in HEVC", IEEE Trans. Circuits Syst. Video Technol. IEEE Transactions on Circuits and Systems, Vol. 20, no. 12, pp. 1778-1791, 2012

[6] G. G. Langdon, "An Introduction to Arithmetic Coding", IBM Journal of Research and Development IBM J. Res. and Dev., vol. 28, no. 2, pp. 135149, 1984.

3

AFRESh: Compression of Reads and Assembled Sequences

3.1 Introduction

AFRESh is a compression solution for genomic reads and assembled genomic sequences. These are the data that are contained in line 2 of each read in FASTA[1] files (see Figure 3.1[2]) and FASTQ[3] files (see Figure 3.2[4]). To enable effective compression for genomic reads and assembled genomic sequences, the coding framework presented in Chapter 2 has been extended with alphabets, prediction tools, and encoding tools. To encode the data generated by the framework (i.e., signaling information), the prediction tools, and the encoding tools, a set of binarizations and contexts have been designed and defined for use with CABAC.
First, an overview is provided of existing approaches for genomic data compression. Subsequently, the designed framework extensions are discussed, followed by the discussion of the designed binarizations and the approaches used for context modeling. Finally, the effect of random access block granularity on coding effectivity is discussed, followed by an extensive analysis of the coding performance offered by AFRESh when coding reads and assembled sequences, respectively.

[1] FASTA is a file format designed by Lipman and Pearson for use with the FASTA similarity search tool [1].
[2] For reading comfort, Figure 1.9 is repeated here.
[3] FASTQ is an extension of the FASTA file format, with added support for quality scores.
[4] For reading comfort, Figure 1.10 is repeated here.

```
Line 1: >EAS149:136:FC806VJ:2:2104:1543:19393
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT

Line 1: >EAS139:136:FC706VJ:2:5:1000:12850
Line 2: AAGAGCAGTATCGATCATAATAGTATAACCAATGTACATTTGTCAGCG
```

Figure 3.1: Example of two reads in a FASTA data file.

```
Line 1: @SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 4: EEDEEDEDCCDDDCCBA><>>>>>>>>>====<<<888976333#####

Line 1: @SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 4: IIIIIIIIHFGFFEEDDCDDDCBBA>>>><<88887778776666665
```

Figure 3.2: Example of two reads in a FASTQ data file.

3.2 Related Work

In general, two different types of compression solutions are used for the compression of genomic data: generic compression tools and specialized genomic data compression algorithms.

Generic compression tools such as GNU Gzip and 7-Zip provide support for the compression of both reads and assembled sequences, are easy-to-use and offer acceptable compression rates. As a result, these tools are commonly used; however, the aforementioned tools do not come with support for random access. Hence, it is necessary to transmit complete files, even when only a subset of the data is needed.

Specialized genomic data compression tools can be split further into two different types, depending on the input data: read data compression tools (e.g., the reference-based tools Deez [2] and CRAM [3], and the referenceless tools LFQC[5] [4] and the referenceless two-pass algorithms SCALCE[6] [5] and ORCOM[7] [6].), and reference-based assembled sequence compression tools (e.g., ERGC[8] [7] and iDoComp [8]).

LFQC uses an approach where a read is split into l-mers of a length between

[5]Lossless FASTQ Compressor
[6]Sequence Compression Algorithms using Locally Consistent Encoding
[7]Overlapping Reads COmpression with Minimizers
[8]Efficient Referential Genome Compressor

4 and 12 nucleotides. These l-mers are put into a bucket[9], based on the nucleotide that occurs in more than 50% of the cases. If no such nucleotide exists, the l-mer is put into a default bucket. Depending on the bucket to which the l-mer has been assigned to, a different Huffman tree is used for compression.

The two-pass algorithms SCALCE and ORCOM use an approach were reads are grouped into bins that share substrings (i.e., a chunk of nucleotides) followed by a second step in which the reads are compressed. After grouping the reads into bins SCALCE compresses the binned data, using GZIP. ORCOM however, first reorders the reads in a bin in such a way that the overlapping parts of the reads are close to each other. In the final step, ORCOM uses one of the previous reads as a reference and encodes the differences between these reads, with a different encoding depending on the overlap and the number of differences. The data that is created in this step is then compressed using an arithmetic coder.

The other tools mentioned above rely on the use of so-called reference sequences. Such an approach allows for significantly higher compression rates compared to stand-alone compression. However, to allow every file to be handled as a separate entity (and as such avoid the storage and management of reference files and their different versions), a stand-alone solution was chosen. This approach is followed by many popular genome data banks such as the DNA Data Bank of Japan (ftp.ddbj.nig.ac.jp) and EBI (www.ebi.ac.uk). Both provide all data available in non-reference file formats such as FASTQ and BAM[10], while only a subset of the data is available in reference-based formats, such as CRAM [3]. These elements lead to the investigation of a referenceless approach.

3.3 Framework Extensions

To support the compression of nucleotidic information, the framework proposed in Chapter 2, has been extended with a set of alphabets and prediction and encoding tools. These extensions are discussed in the next Sections. Section 3.4 will discuss the processes and extensions related to Context-Adaptive Binary Arithmetic Coding (CABAC) i.e., binarizations and context modeling.

3.3.1 Alphabets

Although DNA is made up of only four nucleotides (A, C, G, T), sequenced (and assembled) genomic data can contain additional so-called IUB/IUPAC nucleic acid codes [9]. AFRESh supports three different (sub)sets of these codes, referred to as alphabets, of these IUB/IUPAC nucleic acid codes:

[9]Note that this bucket is not a storage concept, as LFQC is a single-pass compressor the bucket is only a label to identify the coding tree during compression.

[10]BAM files can contain links to references but are stored as stand-alone files, i.e., not using a reference.

- **Simple Alphabet (S):** A, C, G, T
- **Extended Alphabet (E):** A, C, G, T, N
- **Full Alphabet (F):** A, C, G, T, N, R, Y, K, M, S, W, B, D, H, V, -

During encoding, AFRESh identifies and signals the alphabet on a block-per-block basis, allowing for an effective encoding per block, while maintaining support for all IUB/IUPAC nucleic acid codes.

3.3.2 Prediction and Encoding Tools

To code the genomic symbols, AFRESh uses a set of coding tools. This set of coding tools allows AFRESh to exploit, on a block-by-block basis, the different types of redundancy that can be found within genomic data. Figure 3.3 shows the different coding tools that are currently available in AFRESh, split into two categories:

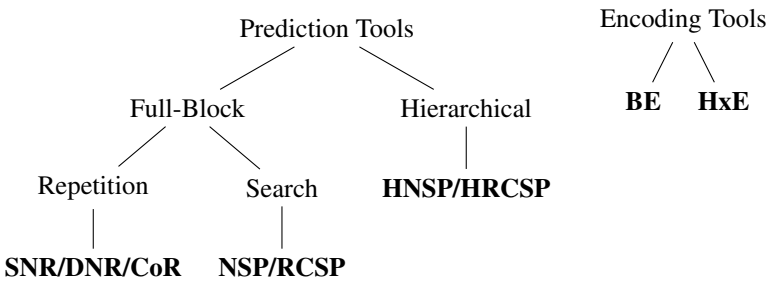

Figure 3.3: Coding toolset currently available in AFRESh.

- **Prediction tools** - define a prediction and create correction information (residual data). The predictions are based on previously encoded parts of the input file or on a repetition of one or more bases. To mitigate compression complexity, the search and hierarchical tools will only look for the best matching blocks within a certain search window of n blocks. The latter parameter can be configured at initialisation.

- **Encoding tools** - convert blocks to a bit representation. This can be either plain bit encoding (Binary Encoding, or BE) or based on the statistical analysis of previously encoded parts of the input file (Huffman Encoding with code words consisting of x nucleotides, HxE). The statistical analysis for the HxE tools is, similar to prediction tools, limited to the search window.

3.3.2.1 Prediction Tools

The selection of provided prediction tools is such that they are able to exploit a variety of known characteristics of genomic data. They are as follows:

- **Single Nucleotide Repetition (SNR)** - generates a prediction based on the repetition of one nucleotide. This tool is mainly applied to homopolymers (repetitions of a single nucleotide) and regions of uncertainty (positions where reliable base calling was not possible, indicated with the symbol/ nucleotide N).
- **Double Nucleotide Repetition (DNR)** - generates a prediction based on the repetition of a pair of nucleotides. This tool is mainly applied to dinucleotide repeats.
- **Codon Repetition (CoR)** - generates a prediction based on the repetition of codons (or, as such, amino acids, represented by a triplet of nucleotides).
- **Normal Search Prediction (NSP)** - selects, within the search window, the contingent sequence of nucleotides of length `block_size` that has the least amount of mismatches when compared to the current block. This tool can be used to compress larger tandem repeats, such as minisatellites, and overlapping reads.
- **Reverse Complement Search Prediction (RCSP)** - reverses and complements the current block, followed by a NSP based on this converted block. This tool can be used to compress the two strands of the same genomic region.

For larger block sizes, it may be more effective to split a block into two smaller parts, so to be able to look for a match separately. These tools can be used for reads or blocks where both halves differ significantly (e.g., one half contains a region of uncertainty). Therefore, two hierarchical prediction tools are provided:

- **Hierarchical Normal Search Prediction (HNSP)**
- **Hierarchical Reverse Complement Search Prediction (HRCSP)**

3.3.2.2 Encoding Tools

In cases where prediction tools are not effective one of two (types of) encoding tools can be selected:

- **Binary Encoding (BE)** - represents all nucleotides in a binary manner using two bits (simple alphabet), three bits (extended alphabet), or four bits (full alphabet). BE is needed to encode the first block after a Random Access

starting point and provides a lower boundary for compression effectiveness when encoding blocks.

- **Huffman Encoding (HxE)** - represents the input nucleotides using Huffman Encoding. Three Huffman trees are generated, based on the nucleotide frequencies in the search window: **H1E** for encoding of single nucleotides, **H2E** for encoding of pairs of nucleotides, and **H3E** for encoding of triplets of nucleotides[11]. These tools can be used in situations where a bias is observed towards certain symbols and/or symbol combinations, hence improving coding effectiveness over fixed-length Binary Encoding.

3.3.2.3 Removed Coding Tools

During the development of AFRESh, two additional prediction tools were tested:

- **Reverse Search Predictor (RSP)** - reverses the current block, followed by an NSP based on this converted block.
- **Complement Search Preditor (CSP)** - creates the complement of the current block, followed by an NSP based on this converted block.

However, both of these prediction tools were used in less than 0.03% of the cases and resulted in a compression gain that was below 0.001 bits per base. Given that these prediction tools resulted in severe efficiency drop (up to 40% slower encoding, compared to the current toolset), it was decided to remove these prediction tools.

3.4 Optimization Methodology

In this section, the process of statistical analysis of the main syntax elements is discussed, followed by the resulting definition of the binarization and the context modelling for these elements. Based on this process, the effectiveness (i.e., the compression ratio) of the CABAC arithmetic coder was maximized.

The optimization process consisted of two iterations and was based on two test files[12]:

- **The Human Y Chromosome** [13]
- **The Arabidopsis Thaliana Genome** [14]

[11] These triplets correspond to codons or amino acids.

[12] Given the extensive set of configurations that have to be tested, this test set is limited to two test files, each representing a different type of species: human and plants. The Y chromosome has been selected as it is smallest chromosome.

[13] ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/CHR_Y/

[14] https://www.arabidopsis.org/download_files/Sequences/TAIR10_blastsets/TAIR10_seq_20101214_updated

In the first iteration, the test files were encoded without CABAC. The resulting coded bitstream decisions were then analysed in order to define the binarization and context selection procedure for the alphabet indicator and the residue. In the second iteration, the test files were encoded with CABAC enabled and the process of analysis has been repeated to define the binarization and context selection for the predictor indicator and the predictor specific parameters. Thanks to this iterative approach, the influence of residue compression was incorporated on the predictor selection.

Additional tests showed that, thanks to the adaptive nature of CABAC and the design of the proposed binarizations, no changes were needed for the compression of reads. In what follows, the binarization and context modeling of the most important syntax elements will be discussed.

3.4.1 Binarization and Context Modeling of Alphabet Indicators

To find a suitable binarization for the alphabet indicator syntax element, the reasoning was adopted that nucleotides other than A, C, G, T may be present, due to uncertainties introduced by sequencing machines and sequencing algorithms. These uncertainties (especially, uncertainties denoted by the character 'N') often occur in bursts. It is therefore expected that, for most blocks, the alphabet indicator will be the same as the previous block. This is confirmed by a statistical analysis, which shows that the assumption is correct for 95.56% to 99.99% of the cases, depending on the block size and test file.

Based on the above observation, the binarization of the alphabet indicator was defined as the combination of a prefix and a suffix. The prefix is a flag indicating whether or not the current alphabet is the same as in the previous block. If another alphabet indicator is used, the suffix is appended. The suffix is used in order to discern between the remaining possible alphabets. Table 3.1 shows the binarization scheme applied when three alphabets are configured.

	alphabet index	prefix	suffix
current == previous	NA	1	
current != previous	0	0	0
	1	0	1

Table 3.1: The binarization scheme for the alphabet indicator field.

In the context modeling procedure, a different context is selected for the prefix and the suffix. The context used for the prefix is initialized with the most probable symbol (MPS) equal to one and a probability state resembling a high degree of certainty. The context used for the suffix is initialized with MPS equal to zero, and

a probability state assigning equal probability to zero and one. The characteristics of the suffix are then learned by CABAC throughout the encoding process.

3.4.2 Binarization and Context Modeling of Residue

A residue is the correction that needs to be applied to a prediction in order to generate the original block. A residue consists of two parts: error positions (represented by a Residue Mask, i.e., a series of ones and zeros) and error corrections (i.e., a list of nucleotides). Figure 3.4 shows an example of a double repeat prediction, based on the nucleotide pair CG, generating the correct output block by applying the residue.

Original Block:	`CGCGTCAGCTCA`
Prediction:	`CGCGCGCGCGCG`
Residue Mask:	`000011100101`
Error Corrections:	`TCATA`
Reconstruction:	`CGCGTTAGCTCA`

Figure 3.4: Example of a double repeat prediction with prediction error correction.

The error positions are binarized as a mask of length `block_size`. For each position in the block, a zero indicates a correct prediction and a one indicates an incorrect prediction (see Figure 3.4). It is assumed that the number of mismatches is lower than the number of matches (zeros), as the best prediction is selected. This is a highly desirable property for CABAC. For context modeling of the error positions, a different context is selected per coding tool. That way, it is acknowledged that performance of (different) predictors can vary, and as such, that it is necessary to take into account a different (and possibly evolving) ratio between matches and mismatches.

Figure 3.5: Visualization of a residue error correction with a diagonal orientation.

The error corrections are encoded as a sequence of corrections. For blocks using the full or extended alphabet, the fixed-length binary index of the correct letter in the corresponding alphabet is encoded in the CABAC bypass mode. For the majority of the blocks (that is, blocks that use the simple alphabet), a binarization of the correct nucleotide is applied.

For ease of comprehension, consider a 2×2 square filled left to right and top to bottom with the letters A, T, C, and G. Corrections are represented as the orienta-

tion of the arrow pointing from the prediction to the correction. The orientation is either horizontal, vertical, or diagonal. Figure 3.5 shows an example of a diagonal orientation, having G as the prediction and A as the correction.

Statistical analysis of the prediction errors that occurred in the test files generated in iteration one showed that diagonal corrections are occurring more frequent than horizontal and vertical corrections, especially for predictions of C and G, where probability of such correction (i.e., T, and A, respectively) is over 50%. Therefore, a short binarization (only consisting of the prefix) has been assigned to the diagonal orientation (Table 3.2). An additional suffix is added to select between Horizontal and Vertical corrections.

Correction	prefix	suffix
Diagonal	0	
Vertical	1	0
Horizontal	1	1

Table 3.2: Binarization scheme for prediction error corrections.

For context modeling, a different context is selected based on the predicted nucleotide, as the relative frequencies of the orientations differ per nucleotide.

3.4.3 Binarization and Context Modeling of Predictor Indicators

To define a suitable binarization for the predictor indicator, the encodings of the second iteration were analysed for the relative frequency of the different predictors and the frequency of two neighbouring blocks using the same predictor. Table 3.3 shows the resulting binarization scheme.

For the majority of the predictors (all predictors, with the exception of the split predictors), the probability of two neighbouring blocks using the same predictor is between 35% and 50%. In case of reads, the probability rises to more than 95% for NSP and RCSP. Based upon these observations, a first flag is used to indicate whether or not the predictor is the same for this and the previous block. That way, one prefix suffices to represent the predictor for approximately 40% of the blocks, with a peak of 95+% for high-coverage aligned reads.

	Flag 1	Flag 2	Flag 3	Suffix
Same as previous	1			
Encoding tools	0	1		identifier
NSP and RCSP	0	0	1	identifier
Other prediction tools	0	0	0	identifier

Table 3.3: Binarization scheme for the predictor indicator.

From the analysis of the relative frequencies of the different predictors, it was deducted that the encoding tools (Binary and Huffman) were used for more than 60% of the blocks, in case of assembled sequences. Therefore, a second flag is used in the binarization scheme to indicate whether the current coding tool is part of this group or not. For aligned reads, NSP and RCSP are typically used more than 95% of the blocks, which leads to a high probability of the first flag being equal to 1, hence skipping the second and third flag.

Some of the tools (e.g., Single Nucleotide Repetition, Double Nucleotide Repetition, and Codon Repetition) are only used in very specific cases and are therefore uncommon. To separate these from the other predictors, a third flag is used. Finally, a suffix is added to identify the predictor in each of the subgroups. A simplified binarization scheme is shown in Table 3.3.

In the context modeling procedure, multiple context models are applied for the first flag. The actual context model to be used is selected based on the previously used coding tool, as one tool is more prone to repetition than the other. For the second flag, one of two context models is selected, based on whether or not the previous coding tool was an encoding tool. For the third flag, one of two context models is selected, based on whether or not the previously used prediction tool was common (i.e., NSP or RCSP). Lastly, for each suffix, a context model is provided.

During the development of AQUa (See Chapter ch4), the observation was made that while this approach is very effective, it is not guaranteed that it will work across all possible data sets, including data sets that result from future sequencing technologies. Therefore, a binarization has been selected for AQUa that allows for adaptation to different distributions.

3.5 Random Access

In this section, the effect of random access on the usage and effectiveness of CABAC is discussed. As discussed in Chapter 2, it is impossible to discern individual symbols in an arithmetically coded bitstream. Therefore, the concept of CABAC-encoded blocks was introduced, where each of these blocks can be decoded separately.

Table 3.4 shows the typical loss in compression effectiveness for different reset window sizes, compared to the optimal size of 131,072 blocks[15]. From these results, it can be concluded that the choice of the CABAC reset window (and as such the random access block size) has a minor effect on the compression effectiveness.

[15]The total coverage of a random access block can be calculated by `random_access_block_size * block_size`.

	CABAC reset window				
	4,096	16,384	65,536	131,072	262,144
Overhead	0.19%	0.16%	0.12%	-	0.15%

Table 3.4: Percentage of overhead versus the optimal CABAC reset window (131,072 blocks).

3.6 Experimental Results

In this section, the experimental setup and the effectiveness of AFRESh when encoding reads and assembled sequences are discussed.

3.6.1 Experimental Setup

To analyse the compression effectiveness for genomic reads, five test files were used that are part of the MPEG benchmark set[16]. The selected files contain aligned data in SAM/BAM format (both high and low coverage) and were converted to FASTQ. The files with aligned reads were selected as they improve the efficiency of the search tools, as best matches are expected to be available in close proximity. For files with non-aligned reads, it is advised to pre-process them to generate "clusters" of similar reads. This will improve compression effectiveness and efficiency as it removes the need for very large window sizes to exploit redundancies, and ideally optimizes the read order for the prediction tools. As discussed further, the framework AFRESh has been designed to support easy modification and extension, and is therefore not optimized for speed.
Quality scores and metadata information are ignored for the measurements for all tested compression solutions. Details on the sequences selected can be found in Table 3.5. All files contain the nucleotides A, C, G, T, and N.

To analyse the compression effectiveness for assembled sequences, assembled sequences from the NCBI archive[17] were used. A selection has been made that contains all Human Chromosomes (of which ChrY has been used to configure CABAC), and multiple genomes originating from plants and bacteria. A list of the test files and their corresponding sizes (in number of nucleotides) can be found, together with the compression results, in Table 3.5.

The compression tests were performed in parallel on a set of five servers, each equipped with 2 Intel Xeon E5-2650 v3 CPUs (10 cores + 10 HT cores each) and 128GB of RAM. Each computing core was dedicated to the encoding of one test file with one configuration (a window size/block size combination) at a time.

[16] http://mpeg.chiariglione.org/standards/exploration/genome-compression/database-evaluation-genome-compression-and-storage
[17] ftp://ftp.ncbi.nih.gov/genomes/

Table 3.5: Detailed information of reads test set and optimal compression settings.

Abbreviation	Dataset	Uncompressed Size		Compression Parameters	
		Size (Nucleotides)	Coverage	Block Size	Window Size
HS (High)	NA12878_S1	159,859,872,414	52×	101	5
HS (Low)	9827_2#49	5,646,323,600	1.75×	100	2
HCC	HCC1954.mix1.n80t20	54,412,950,282	26×	101	16
MIS	MiSeq_Ecoli_DH10B_110721_PF	1,976,351,850	448×	150	3
HS RNA	K562_cytosol_LID8465_TopHat_v2	18,732,205,716	16×	76	2

Speed tests were performed sequentially on a workstation, equipped with an Intel i7 4790K processor and 16GiB of RAM.

3.6.2 Reads

In this section, AFRESh will be compared to three existing algorithms: ORCOM [6], SCALCE [5] and LFQC [4]. These algorithms are the best-performing algorithms currently available, outperforming other state-of-the-art approaches such as QUIP [10] and DSRC2 [11] by a significant margin. It should be noted that ORCOM, SCALCE and LFQC do not support random access, thus allowing them to be more effective. Indeed, they can exploit redundancy that is available across all input data. This observation holds particularly true for SCALCE and ORCOM, as these two tools take advantage of a two-pass approach that first analyses redundancy throughout the whole input file before applying compression. However, this two-step approach typically requires a large amount of memory and/or disk space for temporal storage. Additionally, a single-step approach allows for ”live encoding”. Live encoding makes it possible for the framework to act as a filter that compresses genomic data as it is being generated.
Additionally, AFRESh is compared to two generic algorithms, both at their highest compression setting: GNU Gzip (-best setting) and 7-Zip (LZMA ultra setting).

The configuration of AFRESh was as follows:

- The random access block size was set at 131,072 blocks. As shown in Table 3.4, this is the optimal value.
- The block size was selected to match the length of the reads in the different test files.
- The window size was selected based on the coverage of the reads and the type of genome.

The actual values used for block and window size can be found in Table 3.5.

The results for the different algorithms are limited to the genomic symbols. Quality scores and metadata information are ignored[18]. The compression results for AFRESh, together with the results of the other algorithms, are shown in Tables 3.6 and 3.7. The results are expressed in bits per base (bpb), with a base denoting a nucleotide. It can be observed that the compression rates obtained by AFRESh range from 0.1523 bits/base for Bacteria to 1.1074 bits/base for the low-coverage Homo Sapiens sequence. Comparing the compression results with GNU Gzip and 7-Zip, SCALCE, and LFQC, AFRESh provides a better compression rate for all of the test files, while additionally offering random access.

[18]The compression of quality scores is handled in Chapter 4

Comparing the compression results to ORCOM, AFRESh provides a gain of 3% to 44% for Homo Sapiens (low), Cancer Cell Lines, and Bacteria. For Homo Sapiens (high) and Homo Sapiens RNA on the other hand, the compression rate provided by AFRESh is 30% lower. While the random access overhead can be a cause of loss in compression effectiveness, this cannot explain such a large difference. Indeed, further analysis showed that the coverage of the reads in these files is not equally divided.

Dataset	Compression Rate (Bits per Base)						
	Gzip	7-Zip (LZMA)	ORCOM	SCALCE	LFQC	AFRESh	AFRESh (RAW)
HS (High)	0.4819	0.3352	**0.2371**	0.5330	0.4287	0.3061	0.4916
HS (Low)	1.7464	1.4158	1.9627	1.4280	1.4004	**1.1074**	1.6581
HCC	0.7043	0.5422	0.4553	0.6936	0.6254	**0.4423**	0.7064
MIS	0.2595	0.1857	0.1763	0.3101	0.2373	**0.1523**	0.2770
HS RNA	0.2122	0.1583	**0.1187**	0.1926	0.2693	0.1556	0.4121

Table 3.6: Compression results - reads (in bits per base).

Dataset	AFRESh File Size					
	Gzip	7-Zip (LZMA)	ORCOM	SCALCE	LFQC	AFRESh (RAW)
HS (High)	-36.46%	-8.66%	29.16%	-42.55%	-28.57%	-37.72%
HS (Low)	-36.59%	-21.79%	-43.58%	-22.46%	-20.93%	-33.21%
HCC	-37.20%	-18.42%	-2.85%	-36.23%	-29.27%	-37.39%
MIS	-41.30%	-17.99%	-13.61%	-50.88%	-35.82%	-45.02%
HS RNA	-26.71%	-1.72%	31.09%	-19.25%	-42.24%	-62.25%

Table 3.7: Compression results - reads (file size, compared to other solutions).

Especially in the last 4% of the reads, coverage is much lower[19]. With lower coverage (or no mapping information), a larger window size will offer better compression rates. As the window size parameter is fixed in the current version of AFRESh, this cannot be handled efficiently. Initial simulations on the Homo Sapiens (High) test file with an adaptive window size show compression gains of at least 12%, compared to the non-adaptive version. Further, it should be noted that, next to the additional functionality (random access), AFRESh is single-pass and as such can only exploit redundancy based on previously processed data. Dual-pass solutions, such as ORCOM and SCALCE, can exploit redundancy across the complete input file at a cost of temporal disk storage or extensive RAM usage. Single-pass processing, on the other hand, has the advantage to be able to compress data

[19] Most of these reads are actually unmapped, and as such not sorted.

as they are being generated. For completeness, a column has been added to Table 3.6 and Table 3.7 that shows the actual gain using a CABAC arithmetic coder versus raw syntax and residue storage - AFRESh (RAW). CABAC offers a gain of between 33.21% and 62.25% across the test set.

The compression speeds, given the compression settings shown in Table 3.5, range from 190 KiB/s to 579 KiB/s. It needs to be emphasized that during development of AFRESh, focus was on extensibility and adaptability of the different processing steps and readability of the code (i.e., the algorithms), not on speed. Regarding efficiency, a practical approach will be discussed in Chapter 5.

3.6.3 Assembled Sequences

In this section, AFRESh will be compared to two generic compression algorithms, as no solutions were identified that support compression of large assembled sequences without reference files. The configuration of AFRESh was as follows:

- The random access block size was set at 131,072 blocks.
- The block size was set to 132 bases. As shown in Figure 3.6, this value offers the highest effectiveness over the different chromosomes of the human genome.
- The window size has been set to 5 values: 1, 4, 64, 1,024 and 8,192. Each window size will result in its own bias of effectiveness and efficiency.
- CABAC was enabled for all window sizes.
- To display the effectiveness of CABAC, the test is also run with CABAC disabled and window size 8,192.

Table 3.8 shows the compression results for different window sizes: 1, 4, 64, 1,024, and 8,192, comparing them to both GNU Gzip (-best setting) and 7-Zip (LZMA ultra setting). An additional column shows the compression results for a window size of 8,192 with CABAC disabled (RAW). Looking at the window size, it can be seen that with every increase of the window size, the compression results improve significantly. As complexity increases linearly with the window size, a trade-off can be selected easily, depending on the effectiveness requirements and available computing resources.
Compared to the other solutions, it can be seen that AFRESh outperforms GNU Gzip, even at the smallest window size, both for human genomes and other types of genomes. Compression gains of up to 34% were seen. AFRESh outperforms 7-Zip for most of the sequences at a window size of 4 and shows high gains of up to 16% in compression effectiveness at larger window sizes such as 1,024 and especially at 8,192. Finally, the results show that CABAC offers an additional compression gain of between 7% and 19% over RAW syntax storage.

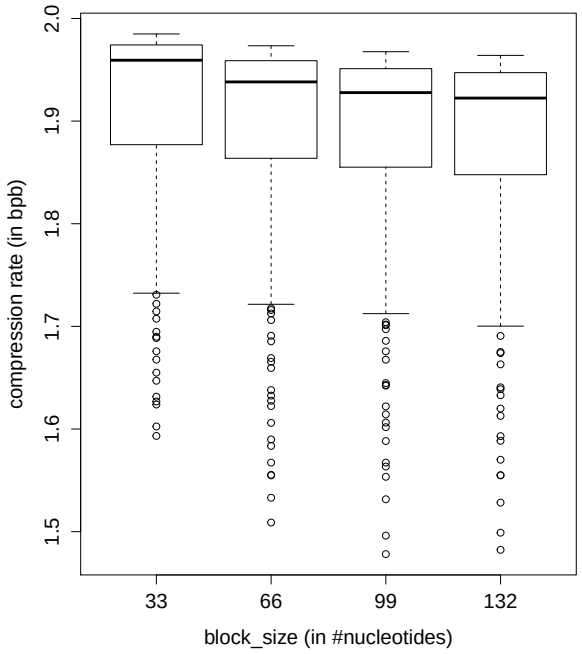

Figure 3.6: Boxplot of the effect of the block size on the resulting compression rate over the different chromosomes of the human genome (in bits per base).

It needs to be noted that the Chromosome Y test file was also part of the CABAC training set, although this is not expected to have a big influence on the compression effectiveness. Other chromosomes, such as Chromosome 19, can be compressed equally well and have equal gains with the usage of CABAC. Additionally, it can be observed that the compression efficiency for Chromosome Y is also significantly better, compared to the other Chromosomes, when CABAC is not used (i.e., AFRESh (RAW)).

As mentioned in the previous section on reads, the current implementation of the supporting framework for AFRESh has not been optimized for speed. Compression speeds range from 382 KiB/s for window size 1 down to less than 1 KiB/s for window size 8,192.

Table 3.8: Compression results - assembled sequences.

Sequence Name	Compression Rate (Bits per Base)								AFRESh File Size Reduction		
	Gzip	7-Zip	AFRESh (CABAC)					(RAW)	Gzip	7-Zip	AFRESh (RAW)
Configuration/window size	-best	LZMA	1	4	64	1,024	8,192	8,192			
hs_alt_HuRef_Chr1	2.3958	1.9538	1.9579	1.9321	1.8533	1.7834	**1.7441**	1.9898	−27.20%	−10.73%	−12.35%
hs_alt_HuRef_Chr2	2.4057	1.9721	1.9563	1.9363	1.8688	1.8029	**1.7624**	1.9963	−26.74%	−10.63%	−11.72%
hs_alt_HuRef_Chr3	2.4077	1.9645	1.9598	1.9380	1.8700	1.7963	**1.7539**	1.9910	−27.15%	−10.72%	−11.91%
hs_alt_HuRef_Chr4	2.4105	1.9550	1.9533	1.9350	1.8730	1.7875	**1.7402**	1.9784	−27.81%	−10.99%	−12.04%
hs_alt_HuRef_Chr5	2.4078	1.9635	1.9572	1.9365	1.8706	1.7917	**1.7492**	1.9869	−27.35%	−10.91%	−11.96%
hs_alt_HuRef_Chr6	2.4063	1.9588	1.9573	1.9355	1.8668	1.7890	**1.7469**	1.9869	−27.40%	−10.82%	−12.08%
hs_alt_HuRef_Chr7	2.3871	1.9362	1.9487	1.9200	1.8428	1.7703	**1.7253**	1.9757	−27.72%	−10.89%	−12.68%
hs_alt_HuRef_Chr8	2.4067	1.9715	1.9576	1.9362	1.8686	1.7968	**1.7549**	1.9939	−27.08%	−10.99%	−11.99%
hs_alt_HuRef_Chr9	2.3922	1.9625	1.9508	1.9271	1.8530	1.7851	**1.7454**	1.9856	−27.04%	−11.06%	−12.10%
hs_alt_HuRef_Chr10	2.4000	1.9690	1.9580	1.9351	1.8593	1.7955	**1.7566**	1.9977	−26.81%	−10.79%	−12.07%
hs_alt_HuRef_Chr11	2.4014	1.9572	1.9603	1.9338	1.8616	1.7818	**1.7422**	1.9872	−27.45%	−10.99%	−12.33%
hs_alt_HuRef_Chr12	2.3925	1.9778	1.9567	1.9268	1.8471	1.7732	**1.7325**	1.9840	−27.59%	−12.40%	−12.68%
hs_alt_HuRef_Chr13	2.4124	1.9936	1.9531	1.9359	1.8762	1.8152	**1.7724**	2.0012	−26.53%	−11.10%	−11.43%
hs_alt_HuRef_Chr14	2.4018	1.9684	1.9594	1.9353	1.8597	1.7848	**1.7432**	1.9894	−27.42%	−11.44%	−12.38%
hs_alt_HuRef_Chr15	2.3931	1.9655	1.9563	1.9315	1.8518	1.7889	**1.7474**	1.9908	−26.98%	−11.10%	−12.23%
hs_alt_HuRef_Chr16	2.3693	1.9439	1.9477	1.9112	1.8151	1.7533	**1.7207**	1.9816	−27.38%	−11.48%	−13.17%
hs_alt_HuRef_Chr17	2.3587	1.9232	1.9487	1.9077	1.7988	1.7464	**1.7128**	1.9766	−27.38%	−10.94%	−13.34%
hs_alt_HuRef_Chr18	2.4107	2.0048	1.9577	1.9394	1.8763	1.8170	**1.7801**	2.0062	−26.16%	−11.21%	−11.27%
hs_alt_HuRef_Chr19	2.3083	1.7999	1.9330	1.8632	1.7232	1.6128	**1.5548**	1.9030	−32.64%	−13.62%	−18.30%
hs_alt_HuRef_Chr20	2.3944	1.9912	1.9620	1.9348	1.8548	1.7997	**1.7638**	2.0067	−26.34%	−11.42%	−12.10%
hs_alt_HuRef_Chr21	2.3998	2.0071	1.9481	1.9282	1.8606	1.8040	**1.7679**	2.0051	−26.33%	−11.92%	−11.83%
hs_alt_HuRef_Chr22	2.3610	1.9444	1.9496	1.9074	1.8028	1.7486	**1.7146**	1.9789	−27.38%	−11.82%	−13.36%
hs_alt_HuRef_ChrX	2.2924	1.8033	1.8684	1.8460	1.7800	1.6630	**1.6039**	1.8891	−30.03%	−11.06%	−15.09%
hs_alt_HuRef_ChrY	2.2748	1.7428	1.8507	1.8244	1.7445	1.5931	**1.4990**	1.8491	−34.10%	−13.99%	−18.94%
Volvox_Carteri	2.1747	1.8094	1.7893	1.7803	1.7351	1.6821	**1.6322**	1.8317	−24.95%	−9.80%	−10.89%
Micromonas_Pusilla	2.3035	2.1066	1.8761	1.8416	1.7983	1.7840	**1.7734**	1.9562	−23.01%	−15.82%	−9.34%
Ostreococcus_Tauri	2.4032	2.2461	1.9685	1.9549	1.9290	1.9166	**1.9065**	2.0487	−20.67%	−15.12%	−6.94%
Monoraphidium_Neglectum	2.3680	2.1733	1.8765	1.8671	1.8467	1.8419	**1.8378**	2.0009	−22.39%	−15.44%	−8.15%
Aeromonas_Australiensis	2.2527	2.1781	1.9757	1.9577	1.9500	1.9558	**1.9481**	2.0949	−13.52%	−10.56%	−7.01%

3.6.4 Tool Selection

In the previous result sections on reads and assembled sequences, the provided results were generated with all available tools enabled. As discussed before, it is possible to select a subset of tools in order to exchange compression effectiveness for a higher compression efficiency. In this section, two configurations will be compared to the configuration that uses the complete toolset, both in effectiveness and efficiency.

The first configuration (without HxE) is using the complete toolset, except for the Huffman Encoding tools. Huffman Encoding is mainly used in cases where there is a low redundancy between blocks, such as assembled sequences, unmapped reads, and reads of low-coverage sequencing data. It is therefore to be expected that the effectiveness penalty will be higher for these types of data than for reads of higher-coverage sequencing data.

The second configuration (which is a subset of the first configuration) is limited to three tools: Binary Encoding, Normal Search Predictor, and Reverse Complement Search Predictor. Given the small size of the toolset it is expected that the efficiency of this configuration will be significantly higher than the configuration using the full toolset. As the Search Predictor tools are used extensively for the coding of reads, it is expected that the effectiveness of the compression of these data will not be affected significantly. In case of assembled sequences and unmapped read data, it is to be expected that effectiveness will drop significantly as the search predictors are less effective for these data.

Figure 3.7 and Figure 3.8 show, for both configurations and for different test files[20], the effect on the effectiveness and the efficiency, respectively. The figures confirm the expected effect on compression effectiveness. The first configuration (without HxE) has a low effect on coding effectiveness for read data ($<$0.16%), but has a higher effectivity loss of 1.1% to 1.6% for the assembled sequence ChrY with window size 4 and 64 respectively. The total encoding times drop to between 27.8% and 95.4% of the encoding times when using the complete toolset.

The second configuration has a significantly higher effect on effectiveness and efficiency (which is to be expected given the small set of tools). As expected, the effect of the smaller toolset is lower on read data (between 0.70% and 2.0%, with HS(High) as an outlier: 3.36%) than on assembled sequences (between 3.21% and 3.93%). The total encoding times drop to between 12.41% and 44.9% of the encoding times when using the complete toolset.

[20]For reads: HS(Low), HCC, HS RNA, MIS, and HS(High). For assembled sequences: ChrY at window sizes 4 and 64.

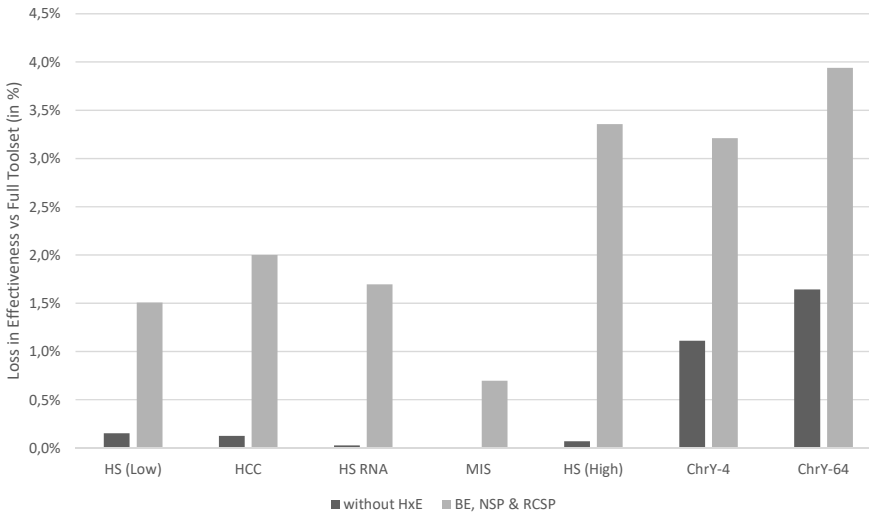

Figure 3.7: Loss in compression effectiveness, compared to the complete toolset.

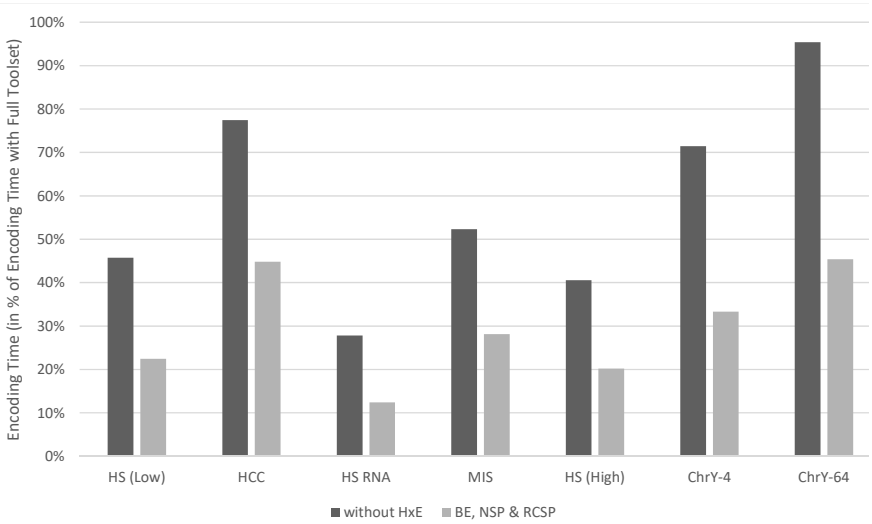

Figure 3.8: Total encoding time, compared to the complete toolset.

3.7 Support for new Sequencing Technologies

As discussed in Section 1.1.2.3, a new generation of sequencing technologies is appearing: single-molecule sequencing. These single-molecule sequencing technologies generate significantly longer reads of variable length (up to 150,000 bases or even longer). Additionally, these technologies show a higher occurrence of indels.

As the compression framework currently processes the data in blocks of a fixed length, adaptations are required to support variable read lengths. A straightforward approach would be to store the read lengths in a separate data stream and split the reads to their respective lengths during decoding in a post-processing step. However, this approach is expected to be sub-optimal for coding effectiveness as a block can contain data of multiple reads and reads can be stored across multiple blocks. Therefore, it would be advised to refactor the existing framework (and coding tools) to adapt the block size to the size of the reads. In case of longer reads, it would also be advised to allow for the splitting of reads in order to increase the prediction performance of the different coding tools.

To improve coding effectiveness for data with a higher occurrence of indels, additional (search prediction) tools should be designed that allow for insertion or deletion of bases.

3.8 Conclusions and Original Contributions

In this chapter, AFRESh, a solution for compression of genomic data, built on top of the framework proposed in Chapter 2, was discussed. The designed alphabets, prediction tools, and encoding tools are discussed, together with the binarizations of the parameters and the residue generated by the framework and its coding tools. The designed binarizations (and the subsequent processing using CABAC) result in a compression effectiveness gain of up to 19% for assembled sequences and up to 62% for reads when compared with raw syntax and residue storage.
Thanks to the combination of specifically designed alphabets, purpose-built coding tools and binarizations, AFRESh outperforms common solutions, such as GNU Gzip, with a compression effectiveness improvement of up to 41% for reads and 34% for assembled sequences. When compared to state-of-the-art 7-Zip compressor, AFRESh still offers a compression effectiveness improvement of up to 22% for reads and up to 16% for assembled sequences. For reads, the proposed framework outperforms specialized compressors such as SCALCE by up to 51%, LFQC by up to 42%, and ORCOM by up to 44% in terms of compression effectiveness.
These results show that, given a properly designed set of coding tools, a block-based compression solution with a set of alphabets and competing coding tools can outperform generic and specialized compressors by a significant margin when compressing nucleotidic data, while offering support for random access.

AFRESh is available for download at `https://github.com/tparidae/AFresh`.

References

[1] D. Lipman et al, "Rapid and Sensitive Protein Similarity Searches", Science, New Series, vol. 227, no. 4693, pp. 1435-1441, 1985.

[2] F. Hach et al., "DeeZ: reference-based compression by local assembly", Nature Methods, vol. 11, no. 11, pp. 1082-1084, 2014.

[3] M. H.-Y. Fritz et al., "Efficient storage of high throughput DNA sequencing data using reference-based compression", Genome Research, Vol. 21, no. 5, pp. 734-740, 2011.

[4] S. Pathak et al., "LFQC: a lossless compression algorithm for FASTQ files", Bioinformatics, vol. 31, no. 20, pp. 3276-3281, 2015.

[5] F. Hach et al., "SCALCE: boosting sequence compression algorithms using locally consistent encoding", Bioinformatics, vol. 28, no. 23, pp. 3051-3057, Sep. 2012.

[6] S. Grabowski et al., "Disk-based compression of data from genome sequencing", Bioinformatics, vol. 31, no. 9, pp. 1389-1395, 2014.

[7] S. Saha et al., "ERGC: an efficient referential genome compression algorithm", Bioinformatics, vol. 31, no. 21, pp. 3468–3475, Feb. 2015.

[8] I. Ochoa et al., "iDoComp: a compression scheme for assembled genomes", Bioinformatics, vol. 31, no. 5, pp. 626-633, 2014.

[9] A. Cornish-Bowden, "Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984", Nucleic Acids Research, Vol. 13, No. 9, pp. 3021-3030, 1985.

[10] D. C. Jones et al., "Compression of next-generation sequencing reads aided by highly efficient de novo assembly." Nucleic Acids Research, vol. 40, no. 22, 2012.

[11] U. Roguski and S. Deorowicz, "DSRC 2–Industry-oriented compression of FASTQ files.", Bioinformatics, vol. 30, no. 15, pp. 2213-2215, 2014.

4

AQUa: Compression of Quality Scores

4.1 Introduction

AQUa is a compression solution for quality scores associated with nucleotides. These are the data that are contained in line 4 of each read in FASTQ[1] files (see Figure 4.1[2]). To enable effective compression of quality scores, the coding framework presented in Chapter 2 has been extended with a quality score alphabet, and a set of prediction and encoding tools. To encode the resulting data (i.e., syntax elements) using CABAC, a set of binarizations and contexts have been designed. First, an overview is provided of existing approaches for the compression of quality scores. Subsequently, the designed coding tools are discussed, followed by the discussion of the used binarizations and the approaches used for context modeling. Finally, the effect of random access block granularity and search window size on coding effectiveness is discussed, followed by an analysis of the compression performance offered by AQUa.

4.2 Related Work

In general, there are two different categories of quality score compressors: compressors using a lossless approach and compressors using a lossy approach. While **lossy quality score compression** is a promising technique for significantly im-

[1] FASTQ is an extension of the FASTA file format, with added support for quality scores.
[2] For reading comfort, Figure 1.10 is repeated here.

```
Line 1: @SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS149:136:FC806VJ:2:2104:1543:19393
Line 4: EEDEEDEDCCDDDCCBA><>>>>>>>>>====<<<888976333#####

Line 1: @SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 2: GATTTGGCGGTTCAAGAGCAGTATCGATCATAATAGTATAATCCATTT
Line 3: +SRR001666.1 EAS139:136:FC706VJ:2:5:1000:12850
Line 4: IIIIIIIIHFGFFEEDDCDDDCBBA>>>>><88887778776666665
```

Figure 4.1: Example of two reads in a FASTQ data file.

proving the effectiveness of compression, it is still a highly sensitive topic in real-world application domains. In particular, the loss in accuracy for the quality scores is feared to influence the outcome of genomic data analysis. However, initial research on the effect of lossy compression of quality scores on variant calling algorithms[3] [1] [2] shows that lossy compression, with tools such as QVZ[4] [3], RBlock and Pblock [4] can maintain, and in some cases even improve, variant calling performance. Additionally, the MPEG standardization committee proposed a framework for the evaluation of the impact of lossy compressors on human genome variant calling [5], aiding researchers in estimating the effects of lossy compression algorithms on variant calling. However, given the early status of lossy quality score compression performance measurement, it was decided to optimize for lossless compression. Nevertheless, the user has, as with other lossless solutions, the freedom to apply a lossy transformation to the input data before lossless compression.

In the domain of **lossless quality score compression**, both generic solutions, such as GNU Gzip and 7-Zip, as well as specialized quality score compressors, such as SCALCE[5] [6] and QVZ[6] [3], are used. For lossless quality score compression, SCALCE uses a 3-rd order arithmetic encoder, which uses the 3 previously encoded quality scores to select a context. As a result, this approach assumes that for all reads the quality scores behave in a similar way and ignores information such as the position of a quality score within a read. QVZ, on the other hand, uses a two-pass approach where it defines a Markov model by its transition probabilities, based on empirical analysis of the entire data set, and uses them to design a codebook. This codebook contains a set of values indexed by position and previous value. In the second pass, the data set is then compressed using this codebook

[3] I.e., algorithms that identify variants (mutations) of a base on a given position, indicating e.g., sensitivity for certain diseases.

[4] Quality Values Zip

[5] Sequence Compression Algorithms using Locally Consistent Encoding

[6] QVZ also supports lossy compression.

and using an adaptive arithmetic encoder which uses a separate model (context) for each position (within a read) and previous value. It is clear that such an approach will typically result to higher compression ratios than a single-pass solution using a similar approach (but not using an analysis step), but will result in higher complexity, more data access, and memory or storage usage.

Other lossless compressors, such as CARGO[7] [7], offer a hybrid solution, in which the genomic data are split into different "types" of data, followed by compression of these different types of data based on (a set of) generic compressors.

While all state-of-the-art solutions provide a significant gain in compression effectiveness, when compared to RAW storage (e.g., in FASTQ), they all lack support for random access and/or require pre-processing. As discussed in Chapter 2, a modern genomic data format should support at least random access and can benefit of a coding mode that can be used for live encoding.

4.3 Coding Tools

This section discusses the different coding tools that were designed to compress the input quality scores. Three of these tools have been adopted from the AFRESh framework, discussed in Chapter 3: the Single Nucleotide Repetition, and the Normal and Hierarchical Normal Search Predictors; the other tools have been designed specifically for the compression of quality scores. Each of the quality score specific tools focuses on dealing with a different type of redundancy within a set of quality scores, both between the quality scores of a specific read and between the quality scores of successive reads. All proposed tools generate a prediction for the quality scores, either per position or for all positions of a read at once. To confirm or correct the predictions, a residue is generated and stored per position.

4.3.1 DFC - Difference Coder

The difference coder is a prediction tool that uses the quality score at the previous position as a prediction for the current quality score. In other words, the residue is the difference between two consecutive quality scores. As a result, this predictor exploits the observation that the difference between neighbouring scores is typically small. However, sequencing technologies that are prone to small error bursts, with larger differences between neighbouring scores, will not benefit from this coding tool as larger differences are costly to represent (for more detail on how differences are represented, see Section 4.4.2). To encode the first quality score of a read, the difference coder uses the first quality score of the previous read as a prediction. If there is no previous read available, the prediction is equal to the quality score represented by the ASCII character 'E', based upon the observation

[7] Compressed ARchiving for GenOmics

that the first quality scores of a read are typically high. This tool is mainly applied to reads where differences between neighbouring scores are small, but existing[8].

4.3.2 ADFC - Average Difference Coder

The average difference coder generates a prediction based on the average value of all previous quality scores within the current read:
$Prediction_i = Average([q_0, q_{i-1}])$ for $i = [0, read\_size - 1]$
Compared to DFC, ADFC will better handle single-score peaks as the peak will only affect one score prediction (the peak score itself) instead of two ((1) the difference between the peak score and its predecessor and (2) the difference between the peak score and its successor). The encoding of the first quality score is equal to the encoding of the first quality score by DFC. This tool is mainly applied to reads where difference between neighbouring scores are small, but short spikes occur within the read.

4.3.3 CVP - Convolutional Predictor

The convolutional predictor generates 32 predictions by applying a set of 32 (convolutional) filters to a combination of previously encoded quality scores at the same or neighbouring positions from the three previous reads. As a result, this predictor exploits positional redundancies. The predictor will select the filter that provides the best prediction, based on the cost of the error correction (encoded in Signed Exponential Golomb notation, which is discussed in Section 5.1).

Table 4.1 shows the matrix containing all 32 available filters. The rows indicate the quality scores to which the filter is applied, whereas the columns indicate the operator used. For those combinations that are used, the mode number is indicated in the matrix. The quality scores are indicated as $Q_{i,j}$, with `i` being the position of the read (i is the read to be encoded) and j the position within the read (0-based). Square brackets indicate a range. For example, $[i-3, i-1]$ contains all reads between (and including) $i-3$ and $i-1$ (which relates to the previous three reads). As an example, Mode 8 will calculate the Mean value of the quality scores $Q_{[i-2,i-1],j}$, which are the scores at position j in the two previously encoded reads. Filters that are using invalid positions j will ignore these positions. Filters that are using unavailable reads are discarded. The filters are based on one of five operators, applying these operators to the input values in order to generate a prediction for each position in the read. These operators are:

- **Mean** - returns the average value;
- **Median** - returns the median value, thus ignoring outliers (e.g., local uncertainties in a read);

[8] If all (or most) quality scores within a read are of the same value, the SRP predictor will be used.

- **Weighted_Mean** - returns a weighted combination of the input values. These weights are adapted per input value, based on the distance to the position of which the value is to be calculated.
- **Min** - returns the lowest value; and
- **Max** - returns the highest value.

This tool is applied to reads that show similar behaviour to the previous reads (according to one of the 32 models).

	Min	Max	Mean	Weighted Mean	Median
$Q_{[i-2,i-1],j}$	-	-	8	17	-
$Q_{[i-2,i-1],j-1}$	-	-	9	18	-
$Q_{[i-2,i-1],j+1}$	-	-	10	19	-
$Q_{[i-3,i-1],j}$	0	2	11	20	26
$Q_{[i-3,i-1],j-1}$	-	3	12	21	27
$Q_{[i-3,i-1],j+1}$	-	4	13	22	28
$Q_{i-1,[j-1,j+1]}$	-	5	14	23	29
$Q_{[i-2,i-1],[j-1,j+1]}$	1	6	15	24	30
$Q_{i-1,[j-1,j+1]}$,$Q_{i-2,j}$	-	7	16	25	31

Table 4.1: The 32 filters that are used by the CVP coding tool.

4.3.4 SRP - Single Repeat Predictor

The single repeat predictor is a tool based on the SRP tool in AFRESh, generating a prediction for the full read that consists of the repetition of one specific quality score. This predictor is especially useful for reads with stable quality scores (i.e., reads that contain many identical scores, such as reads that are completely unreliable, or scores that are fluctuating closely around a certain score). In contrast with the SRP tool in AFRESh, the base quality score (i.e., the score that is used as a prediction) is stored as the difference between the first quality score of the read and the quality score that appears the most in the previous read. In case of a tie, the lowest quality score is selected. This tool is applied to reads that contain a high number of one quality score, possibly extended with a high number of quality scores that are close to this quality score.

4.3.5 AVP - Average Predictor

The average predictor generates a prediction based on the average value of the previous quality score in the same read and the co-located value in the previous

read. The quality score at the beginning of the read is calculated based on the first two values of the previous read. This tool is applied to reads that show similar behaviour to the previous read, but contain quality scores that are typically offset by a small value that can be corrected by using the previously encoded quality score in the current read.

4.3.6 NSP - Normal Search Predictor

The normal search predictor is based on the NSP tool in AFRESh; it selects, within a search window, the contingent chunk of quality scores of length `read_size` that has the lowest cost of signaling the prediction errors when used as a prediction for the current chunk. This chunk of quality scores can start at any position (not only at read borders). The currently encoded read is assumed to be appended to the search window; it can, as such, be used as a prediction, as long as at least one quality score is predicted from the search window. This tool is applied to reads for which a read can be found with which it shares a large contingent chunk of quality scores.

4.3.7 HNSP - Hierarchical Normal Search Predictor

The hierarchical normal search predictor is based on the NSP tool in AFRESh; it applies the NSP prediction tool to the first half and the second half of the read separately. Splitting the reads increases the likeliness of finding a better prediction, at the cost of having to signal an additional pointer. This tool is applied to reads for which no read can be found with which it shares a large contingent chunk, but where for each half a separate read can be found with which it shares a large contingent chunk of quality scores.

4.3.8 Removed Coding Tools

As discussed previously, three coding tools have been migrated from AFRESh to AQUa. The other tools have been removed:

- **Double Nucleotide Repetition (DNR)** - double quality score repetitions over a longer region are rare.
- **Codon Repetition (CoR)** - there is no concept of codons in quality scores and triple quality score repetitions over a longer region are rare.
- **(Hierarchical) Reverse Complement Search Prediction (HRCSP/RCSP)** - there is no concept of complements in quality scores.
- **Huffman Coding (HxE)** - the usage frequency of the H1E coding tool was below 0.06%, except for test file 05 where H1E is used for more than 3%

of the blocks. Additionally, the largest gain in compression effectiveness across the test set is below 0.0005%. Therefore, it was decided to remove H1E. The higher order Huffman coding tools (H2E and H3E) were almost never used and hence were removed too.

4.4 Binarization and Context Modeling

This section discusses the different binarizations that were designed and used to represent the parameters and the residues of the different coding tools. Compared to AFRESh, binarization and context modeling of residual data have been adapted to support the full range of quality score values. Furthermore, binarization and context modeling of the parameters of the reused coding tools have been adapted, as well as binarization and context modeling of the coding tool identification parameter. Finally, binarization and context modeling of the parameters of the novel coding tools, as presented in this chapter, have been added. The binarizations and the contexts have been created by analysing the test files from Table 4.4. This test set consists of both low- and high-coverage files with a fixed-length read size, generated with Illumina HiSeq and MiSeq sequencers and, in case of file 23, an artificially mixed file. All binarizations were selected based on the distribution of the values of the syntax elements to be encoded. When a syntax element comes with varying distributions across different files, binarizations are selected that help CABAC to adapt to the specific characteristics of the different files.

4.4.1 Value Representations

Five different representations are used for the parameters and the residual values:

- Binary representation [8];
- Truncated Unary representation [8];
- Unsigned Exponential Golomb representation [9];
- Signed Exponential Golomb representation [10]; and
- Signed Truncated Exponential Golomb (STEG) representation [8].

The **Binary** representation corresponds to a base-2 representation, with a length y for value x.
The **Truncated Unary** representation of value x consists of x 1-bits, followed by a 0-bit. When x is equal to the maximum value, the trailing 0-bit is discarded. The first column of Table 4.2 shows a number of example truncated unary representations.
The **Unsigned Exponential Golomb** representation of value x consists of a suf-

Value	Truncated Unary	Unsigned Exponential Golomb
0	0	1
1	10	010
2	110	011
3	1110	00100
...	...	...
9	1111111110	0001010
10	1111111111	0001011

Table 4.2: Examples of the Truncated Unary and Unsigned Exponential Golomb binary representations.

fix based on the Binary representation of $x + 1$ and a prefix of 0-bits of $length = suffix\_length - 1$. The second column of Table 4.2 shows a number of example Unsigned Exponential Golomb representations.
The **Signed Exponential Golomb** representation is an extension of the Unsigned Exponential Golomb representation, adding support for negative values by mapping a value $x \leq 0$ onto the value $-2 * x$ and a value $x > 0$ onto the value $2 * x - 1$. Table 4.3 provides a number of example representations for Signed Exponential Golomb.
The **Signed Truncated Exponential Golomb** encoding consists of three parts:

- a Truncated Unary representation;
- an Unsigned Exponential Golomb representation; and
- a one-bit Binary representation.

The Truncated Unary representation is used to represent all values x where $x \leq y$, with y a fixed value. The Unsigned Exponential Golomb representation is used to represent the value of $|x| - y$, provided that $|x| \geq y$. The Binary representation is used to signal the sign of x.

Value	Signed Exponential Golomb
0	1
1	010
-1	011
2	00100
...	...

Table 4.3: Examples of the Signed Exponential Golomb binary representation.

4.4.2 Binarization and Context Modeling of Residue

Given that a coding tool is selected based on its effectiveness, it is assumed that the prediction of that coding tool is highly accurate. As a result, the prediction errors are expected to be sparse and to be typically small and centered around zero, with zero being highly likely. Therefore, in a first step, a residual mask with length `block_size` is generated, which indicates for each position whether the value is correct (zero) or not (one). For each of the positions where the residual mask is one, the correction is stored. Corrections are not stored as the binary representation of the correct quality score (which would cost a fixed $\lceil (log_2(x+1) \rceil)$ bits, with x being the size of the quality score alphabet), but as a difference value `prediction - actual_quality_score`. This difference value is encoded by making use of a Signed Exponential Golomb binary representation, where all residual values $x > 0$ are replaced by $x - 1$, given that the residual value zero does not occur (by definition of the residual mask). This binarization is used for all coding tools, with the exception of DFC.

The context modeling of the residual mask uses three contexts per coding tool, so to be able to handle the possible difference in prediction accuracy of each of these tools. One of the three contexts (ctx0, ctx1, and ctx2) is selected per quality score, based on the prediction accuracy of the coding tool for the previously encoded quality scores in the read. A correct prediction will select a prior context; an incorrect prediction will select the next context. For example, when a correct prediction was making use of ctx1, ctx0 will be selected for the following quality score; in case of an incorrect prediction, ctx2 will be selected. The contexts are initialized with decreasing probability values (states).
For context modeling of the residual corrections, a context is provided for each bit position of the Signed Exponential Golomb representation. To further improve the context modeling, this set of contexts per bit position is provided per possible quality score. As a result, the arithmetic coder can adapt to the different distributions that can be expected based on the predicted quality score. At start-up, contexts are initialized to a fixed set of states, based upon the expected relative frequencies of residual corrections for each of the given predicted quality scores.

The residue of the DFC coding tool (i.e., the differences between sequential quality scores) shows a significantly different distribution, compared to the residue of the other coding tools, and as such, requires a slightly different approach for binarization. Figure 4.2 shows the minimum, average, and maximum occurrence of the residue values across the different test files for the DFC coding tool. It can be seen that, while there is a clear Laplacian distribution around the center (zero, or perfect prediction), there are smaller spikes around other position values within the range of $[-10, +10]$. The positions of these spikes differ between test files. Therefore, the residual values should be encoded differently. To effectively

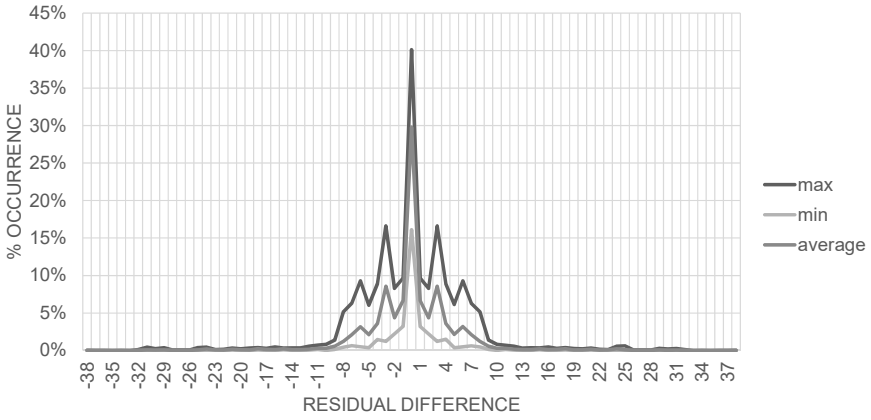

Figure 4.2: Minimum/average/maximum occurrence of the DFC coding tool residue values.

encode the DFC residual values, STEG encoding is used, with $y = 10$. This representation is also used for the encoding of motion vectors by the H.264/AVC video coding standard.
For context modeling of the residual values produced by the DFC coding tool, a context is provided for each bit position of the representation. As with the context modeling of the residual corrections of the other tools, this set of contexts per bit position is provided per possible quality score. Each of the contexts is initialized with the equi-probable state.

4.4.3 Binarization and Context Modeling of CVP Mode

The CVP tool needs to signal which filter (mode) was used during encoding. Figure 4.3 shows, for each mode, the minimum, maximum, and median usage across the test files for these blocks for which the most effective tool was CVP, sorted by decreasing usage. Given the geometric distribution, the selected filter is signaled using the Unsigned Exponential Golomb representation[9].
For context modeling of the signaling of the selected filter, a context is provided per bit position in the Unsigned Exponential Golomb representation. Each of the contexts is initialized with the equi-probable state.

[9]The filter is signaled with a value in the range [0,31], from most probable filter to least probable filter. E.g., filter 2 will be represented by 0, filter 7 by 1, and filter 6 by 31 (See Figure 4.3).

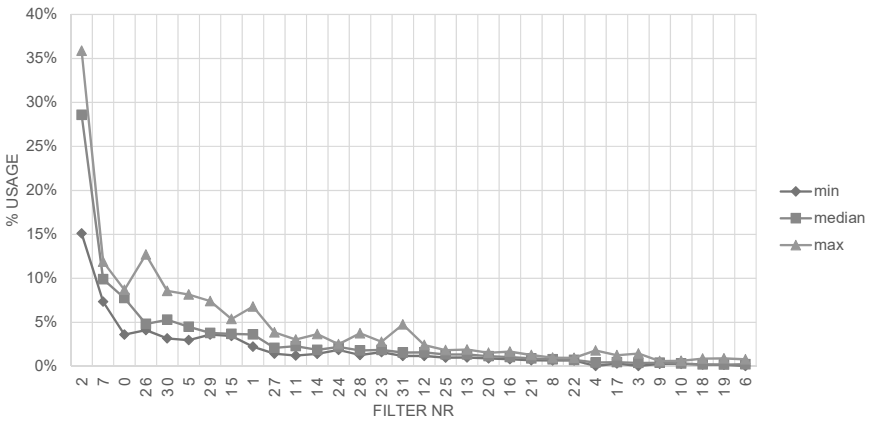

Figure 4.3: Usage of the different CVP modes (min/median/max), sorted by decreasing usage.

4.4.4 Binarization and Context Modeling of NSP and HNSP Pointers

The NSP and HNSP tools need to signal the pointer to the best prediction within a search window. As these pointer values are equally distributed, a standard Binary representation of length $\lceil log_2(min(\texttt{window\_size}, \texttt{actual\_window\_size}) + 1) \rceil$ is being used, with `actual_window_size` being the size of the window at encoding time. As an example, in case of encoding read 3, only read 1 and read 2 are available as a reference. Hence, signaling the pointer to the different reads in the window can be done by making use of a single bit[10].
The binarization of the pointer is processed by the bypass arithmetic coding engine, assuming an equi-probable distribution.

4.4.5 Binarization and Context Modeling of Coding Tool Identification

Figure 4.4-4.7 show the usage of the different coding tools for a number of test files (See Table 4.4), across a set of `window_sizes`. As can be seen in the aforementioned figures, the usage of the different coding tools can significantly differ amongst different files and settings: e.g., the DFC encoding tool is used in the majority of the cases for file 02, but less so for the other files, and is even not used for file 05. Furthermore, the NSP tool is used in the majority of the cases for file 05 at larger window sizes, but is hardly used for smaller window sizes. To be

[10]For simplicity, this example assumes NSP and HNSP are only capable of searching at positions that start at a block border.

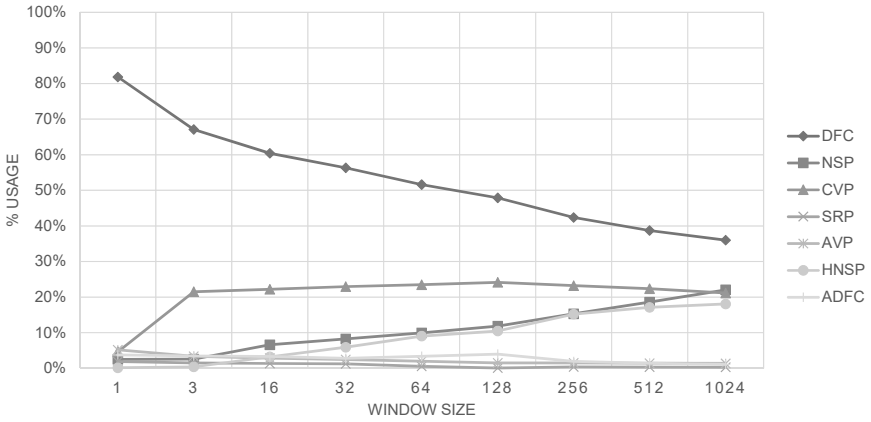

Figure 4.4: Coding tool usage for test file 02.

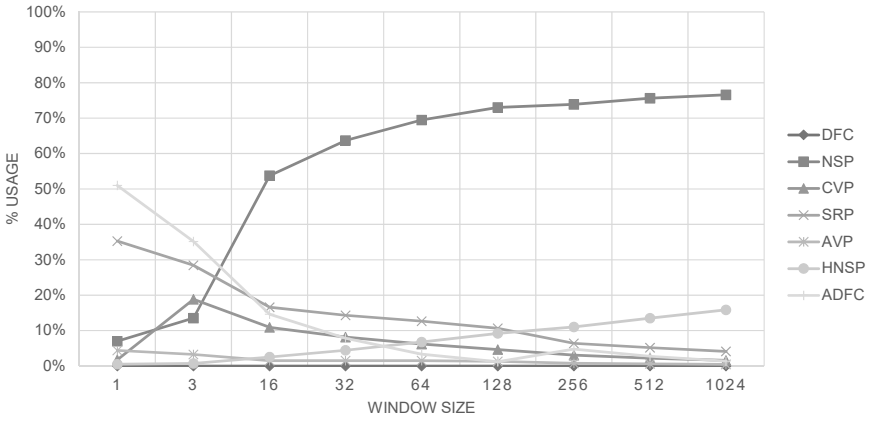

Figure 4.5: Coding tool usage for test file 05.

able to adapt to these different usage statistics, the coding tools are identified by a Truncated Unary representation for their identification number.
For context modeling, a context is provided for each bit position of the Truncated Unary representation. Each of the contexts is initialized with the equi-probable state. Given the adaptivity of CABAC, the contexts will adapt to the actual coding tool usage for each specific file.

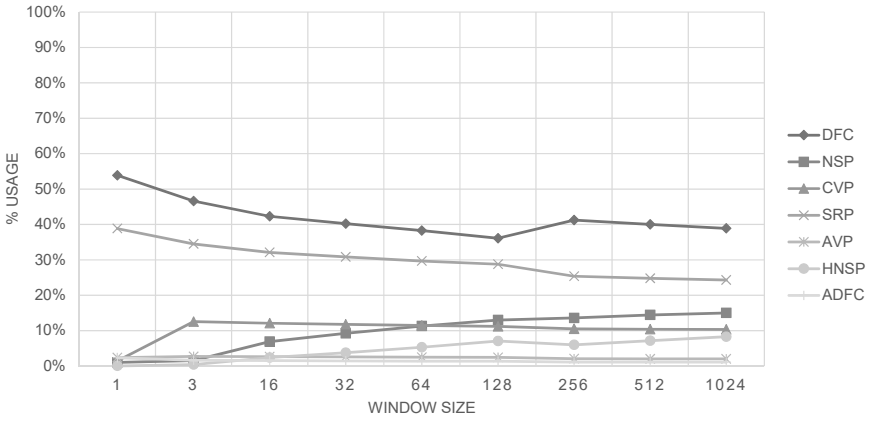

Figure 4.6: Coding tool usage for test file 10.

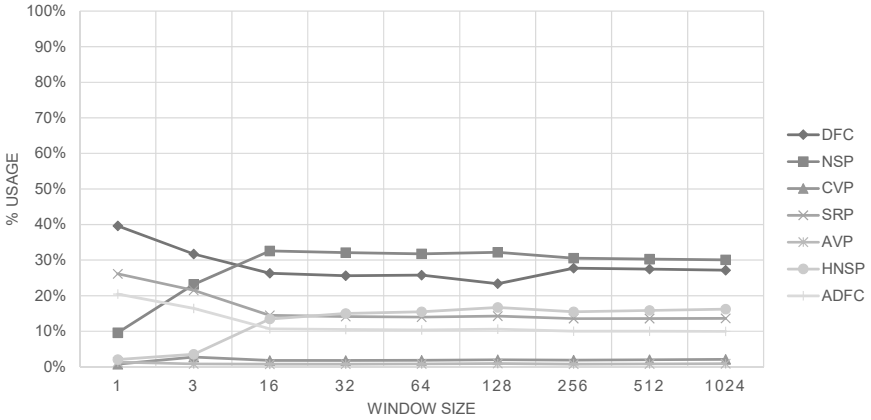

Figure 4.7: Coding tool usage for test file 23.

4.5 Random Access

In this section, the effect of random access on the usage and effectiveness of CABAC is discussed. As discussed in Chapter 2, it is impossible to discern individual symbols in an arithmetically coded bitstream. Therefore, the concept of CABAC-encoded blocks was introduced, where each of these blocks can be decoded separately.

To measure the effect of the random access block size on the compression effectiveness, the reset frequencies were set to the powers of 2 within the range $[32, 768 - 1, 048, 576]$. The total coverage of a random access block can be calculated by `random_access_block_size * block_size`. For example, for test file 10, given a read length of 76 and given the tested range of $[32, 768 - 1, 048, 576]$, the reset frequency results in a random access block size of 2.49 to 79.69 megascores (million scores). For test file 16, with a read length of 150, random access block sizes range from 4.92 to 157.29 megascores, with the same test range.

Figure 4.8 shows the loss in compression effectiveness for a set of random access block sizes, compared to a random access block size of 1,048,576. For each of the test files and for a window size of 16 reads. From these results, it can be concluded that the choice of the CABAC reset window (and as such, the random access block size) has a minor effect on the compression effectiveness. For most test files, the overhead is lower than 0.5%, even with random access block sizes of 32k reads. For files 10, 06, and 23, the overhead is, even at the small random access block size, limited to 3.42%, 3.02%, and 1.96% respectively, with overhead decreasing rapidly for larger random access block sizes.

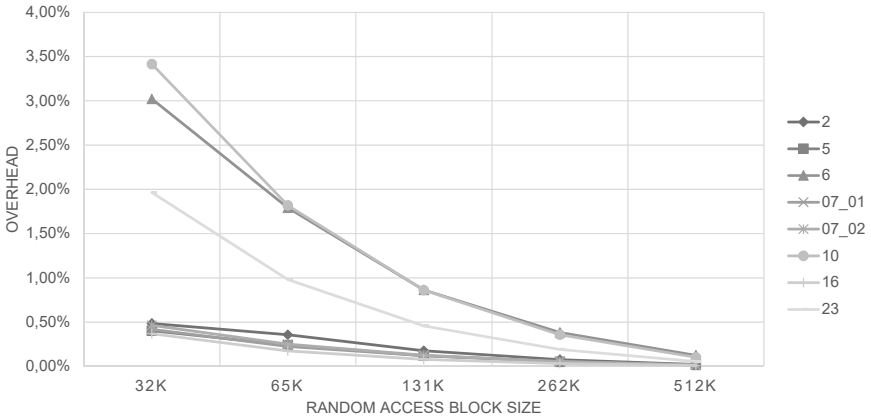

Figure 4.8: Overhead of smaller random access sizes versus the largest random access size (window size 16).

Code	Filename	#Quality Scores	Read Length
02	NA12878_S1.bam	159,859,872,414	101
05	9827_2#49.bam	5,646,323,600	100
06	NA21144.chrom11.ILLUMINA.bwa.GIH.low_coverage.20130415.bam	1,014,768,000	100
07_01	ERR174310_1.FASTQ	20,965,526,167	101
07_02	ERR174310_2.FASTQ	20,965,526,167	101
10	K562_ cytosol_LID8465_TopHat_v2.bam	16,511,933,432	76
16	MiSeq_Ecoli_DH10B_110721_PF.bam	1,976,351,850	150
23	HCC1954.mix1.n80t20.bam	95,181,910,958	101

Table 4.4: Detailed information of the quality score test set.

4.6 Experimental Results

This section discusses the experimental setup and the effectiveness of the proposed framework when encoding the quality scores of reads.

4.6.1 Experimental Setup

To investigate the effectiveness of lossless compression of quality scores, a diverse set of eight test files has been selected from the benchmark set used by MPEG for the analysis of quality score compression [11]. The subset only contains fixed-length reads (as variable-length reads are currently not supported) and files that are larger than 1 GiB. Table 4.4 gives a more detailed overview of the test files selected.

The BAM input files have first been converted to the FASTQ file format. These files were then used for testing the different approaches towards quality score compression. In case of the generic compression tools, a filtered version of the aforementioned FASTQ files was used, only containing the quality scores. To improve the processing speed of the whole benchmark set with the AQUa compression framework (especially for the larger test files, combined with larger window sizes), the files were split into smaller files of 2,621,440 reads, a multiple of all tested CABAC random access windows. This results in a slightly larger output size due to the extra headers and footers, but this overhead is negligible ($<= 21$ bytes per output file). The compression tests were performed in parallel on a set of five servers, each equipped with 2 Intel Xeon E5-2650 v3 CPUs (10 cores + 10HT cores each) and 128GB of RAM. Each computing core was dedicated to the encoding of one small test file of 2,621,440 reads, using one encoder configuration (`block_size`, `window_size`, `random_access_block_size`) at a time. Speed tests were performed sequentially on a workstation, equipped with an Intel i7 4790K processor and 16GB of RAM.

4.6.2 Window Size

This section discusses the effect of the window size on the compression effectiveness and compression speed. Figure 4.9 shows the relative compressed size of the different test files for different window sizes, compared to a window size of one read. It can be seen that, for some files, increasing the window sizes offers gains in compression effectiveness (e.g., $> 7.5\%$ for test file 05), while others gain less than 1%, even for window sizes of 1,024 reads. This indicates that redundancy between the quality scores of different reads is higher for some test files (e.g., 05) than others (e.g., 10). This is confirmed by the evolution of the total usage of the NSP and HNSP coding tools for the different test files. As shown in Figure 4.5,

[11] http://mpeg.chiariglione.org/standards/exploration/genome-compression/updated-database-evaluation-genomic-information-compression

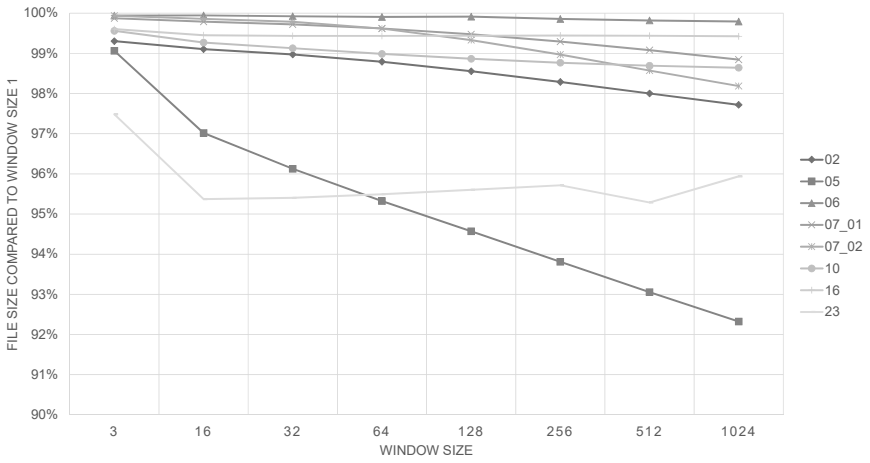

Figure 4.9: Total compressed size, compared to a window size of one read.

the total usage of the NSP and HNSP coding tools for test file 05 increases significantly with larger window sizes (to up to 92.45% at window size 1,024). For test file 10, the total usage of these coding tools increases significantly slower, with a total usage of 23.3% at window size 1,024. Finally, the artificially generated test file 23 shows a gain in compression effectiveness between window sizes 2 and 16, followed by losses in compression effectiveness for larger window sizes (within a range of $< 1\%$).

The compression speed for window sizes 1 to 512 ranges from 5 KiB/s for window size 1,024 to 488 KiB/s for window size 1 (7-Zip processes around 748 KiB/s on the test equipment, using 2 threads). While the compression speed does not drop linearly (a 60$\times$ speed drop with a 512$\times$ larger window), it is still clear that window sizes should be selected carefully for an optimal trade-off between compression effectiveness and compression speed.
It needs to be emphasized that during development of AQUa, focus was on extensibility and adaptability of the different processing steps and readability of the code (i.e., the algorithms), and not on speed. Speedwise, a practical approach will be discussed in Chapter 5.

4.6.3 Compression Results

In this section, the compression effectiveness of AQUa is compared to the compression effectiveness of the commonly used single-pass (generic) algorithms GNU Gzip (-fast and -best setting) and 7-Zip (LZMA setting), the state-of-the-art single-pass algorithm SCALCE, and finally, the state-of-the-art dual-pass algorithm QVZ.

The configuration of the framework was as follows:

- the random access block size was set at 1,048,576 blocks;
- the block size was selected in such a way that it matches the length of the reads in the different test files; and
- the window size was set to 1, 3, 16, 32, 64, 128, 256, 512, and 1,024. Window size 1 is the minimal window size, whereas window size 3 is the minimal size that enables all modes of the CVP tool.

The compression results for AQUa, together with the results obtained for the other single-pass algorithms, are shown in Table 4.5. The results are expressed in bits per score (bps), with score being one quality score value.
Comparing the best compression results of AQUa for each of the test files with the commonly used GNU Gzip tool, shows a better compression rate for all of the test files, with file sizes being 21.67% to 38.49% smaller than GNU Gzip at the fastest setting, and being 13.69% to 22.46% smaller than GNU Gzip at the best setting, while additionally offering random access. Comparing the results of AQUa with the more advanced 7-Zip compressor at the Ultra settings (with 4GB random access blocks), AQUa achieves mixed results, ranging from 3.41% larger files to 6.48% smaller files, while additionally offering random access.

As explained in Section 7.3 and as shown in Figure 4.9, the compression effectiveness of AQUa can be improved significantly by increasing the window size in case of files with higher redundancy between reads, such as file 05. It is for these test files that 7-Zip is performing better than AQUa. Comparing the results of AQUa with the purpose-built single-pass algorithm SCALCE, AQUa offers a better compression rate for all of the test files, with file sizes being 13.26% to 21.14% smaller than SCALCE, while additionally offering random access.

Table 4.6 shows the compression results for AQUa, compared to the 2-pass QVZ algorithm. As can be expected, the analysis step (and the lack of random access support) allows the QVZ algorithm to offer higher compression effectiveness. However, for test file 23, AQUa offers a better compression rate. These results show that a dual-pass solution can offer (but does not guarantee) a higher compression effectiveness. It should be noted that, while AQUa does not support dual-pass compression, input data can be pre-processed (e.g., applying a sorting algorithm) before it is processed by the AQUa compressor, effectively processing input files in two passes. The latter observation holds particularly true for file 05: the compression gain when enlarging the search window (see Figure 4.9), together with the increasing usage of NSP and HSNP when enlarging the search window (see Figure 4.5), shows that a reshuffle of the input data can provide high effectivity gains.

Dataset	Compression Rate (bits per quality score)					AQUa File Size vs			
	AQUa	SCALCE	Gzip -fast	Gzip -best	7-Zip LZMA	SCALCE	Gzip -fast	Gzip -best	7-Zip LZMA
02	**2.55**	3.02	3.53	2.98	2.56	-18.37%	-38.49%	-16.96%	-0.18%
05	3.65	4.24	4.62	4.26	**3.52**	-16.25%	-26.64%	-16.66%	+3.41%
06	**3.71**	4.20	4.51	4.24	3.80	-13.26%	-21.67%	-14.22%	-2.40%
07_01	2.86	3.29	3.76	3.26	**2.82**	-14.85%	-31.38%	-13.69%	+1.47%
07_02	2.67	3.12	3.62	3.08	**2.65**	-16.96%	-35.86%	-15.45%	+0.64%
10	**2.81**	3.30	3.79	3.29	2.92	-17.43%	-34.88%	-17.12%	-3.81%
16	**3.05**	3.51	3.95	3.48	3.06	-15.19%	-29.52%	-14.14%	-0.49%
23	**2.68**	3.25	3.69	3.29	2.86	-21.14%	-37.54%	-22.46%	-6.48%

Table 4.5: Compression results - single-pass compressors.

Dataset	Compression Rate (bits per quality score)		AQUa File Size vs
	AQUa	QVZ	QVZ
02	2.55	**2.16**	+18.17%
05	3.65	**2.73**	+33.47%
06	3.71	**3.39**	+09.24%
07_01	2.86	**2.39**	+20.08%
07_02	2.67	**2.23**	+19.81%
10	2.81	**2.64**	+06.42%
16	3.05	**2.58**	+18.03%
23	**2.68**	2.69	-00.38%

Table 4.6: Compression results - dual-pass QVZ compressor.

4.7 Tool Selection

In the previous result section, the provided results were generated with all available tools enabled. As discussed before, it is possible to select a subset of tools in order to exchange compression effectiveness for a higher compression efficiency. In this section, three configurations will be compared to the configuration that uses the complete toolset, both in effectiveness and efficiency. The results discussed here are based on the quality scores of the first million reads of each file in the test set (see Table 4.4).

The first configuration (DFC/DFT only) is using only the difference coding tools. Given the low complexity of these tools, it is expected that encoding efficiency will be significantly higher, compared to the full toolset. Window sizes don't have an effect on the efficiency and effectiveness of these tools, hence it is to be expected that the loss in effectiveness and the gain in compression efficiency will grow with higher window sizes. The second configuration (no NSP/HNSP) is using all coding tools, except for the search-based tools (NSP and HNSP). Given the slightly higher complexity of the additional tools, it is to be expected that encoding efficiency will be lower than the first configuration, but still significantly higher than the configuration with the full toolset. Window sizes don't have an effect on the efficiency and effectiveness of these tools (except for window size 1, where CVP limits the set of modes that can be used to 8), hence it is to be expected that the loss in effectiveness and the gain in compression efficiency will grow with window sizes, especially with window sizes larger than 16[12]. The third configuration (no HNSP) is using all coding tools, except for the hierarchical search prediction tool (HNSP). Given that HNSP is the most complex predictor, it is to be expected that

[12]CVP will enable all modes, and hence reach maximum compression effectiveness, with a window size larger or equal to three. With the used set of window sizes, this means that all modes are enabled for window size 16 or larger.

removing HNSP will significantly improve compression efficiency, especially with higher window sizes. Figure 4.10 and Figure 4.11 show, for all configurations and for different test files at four window sizes, the effect on the coding effectiveness and efficiency, respectively. The figures confirm the expected effect on compression effectiveness. The first configuration (DFC/DFT only) has a low effect on coding effectiveness at smaller window sizes, compared to the configuration with the full toolset enabled. At window size 1, the loss in coding effectiveness is between 0.05% and 2.13%. At window size 16, the loss in coding effectiveness is between 0.22% and 9.78%. At larger window sizes (256 and 1024), the loss in coding effectiveness rises to between 0.32% to 12.39%.
At window size 1, the total encoding times drop to between 23.04% and 29.41% of the encoding time with the full toolset enabled and drop further to between 0.09% and 0.20% of the encoding time (at window size 1024), which illustrates the high complexity of the search tools.
The second configuration (no NSP/HNSP) has, as expected, a lower effect on coding effectiveness. At window size 1 (which enables only 8 modes in CVP), the loss in effectiveness, compared to the configuration with the full toolset enabled, is between 0.02% and 1.34%. At window size 16, the loss in coding effectiveness is between 0.15% and 5.34%. At larger window sizes (256 and 1024), the loss in coding effectiveness is between 0.21% and 8.08%.
At window size 1, the total encoding times drop to between 44.61% and 68.30% of the encoding time with the full toolset enabled and drop further to between 0.23% and 0.54% of the encoding time at window size 1024.
Finally, the third configuration (no HNSP) demonstrates that HNSP is highly complex (disabling HNSP at window sizes 16 or higher, lowers the encoding time to around 50% of the encoding time with HNSP enabled) and offers limited gains in effectiveness (between 0.00% and 0.28% for window size 1 and between 0.13% and 1.91% for window size 1024). From these results it can be concluded that HNSP should only be enabled in cases where maximum compression effectiveness is required.

4.8 Support for new Sequencing Technologies

During and after the development of AQUa, significant changes have been introduced to how quality scores are generated and handled. Illumina first presented so-called 8-binning, which limited the set of quality scores to 8 values [11]. This process has then further been refined and currently the NovaSeq 6000 sequencer produces only four different quality scores [12].
Given that these quality scores are within the range of the quality scores alphabet in AQUa, AQUa will be able to compress these data. However, to effectively compress the data created using these sets of quality scores, a new alphabet should

be added to AQUa that supports only these values as this would make the coding of differences and of residue significantly more efficient. With this new alphabet, the different scores of the NovaSeq 6000 quality scores (2, 12, 23, 37) will be represented internally by the values 0, 1, 2, 3 hence resulting is smaller values for differences between quality scores (E.g., for DFC/DFT), lower values for the residue for prediction tools, and better predictions when using the CVP coding tool.

Variable length reads can be handled with the same approaches as discussed in Section 3.7.

Other characteristics of the single-molecule sequencing methods, such as quality drops, can be handled using the existing coding tools.

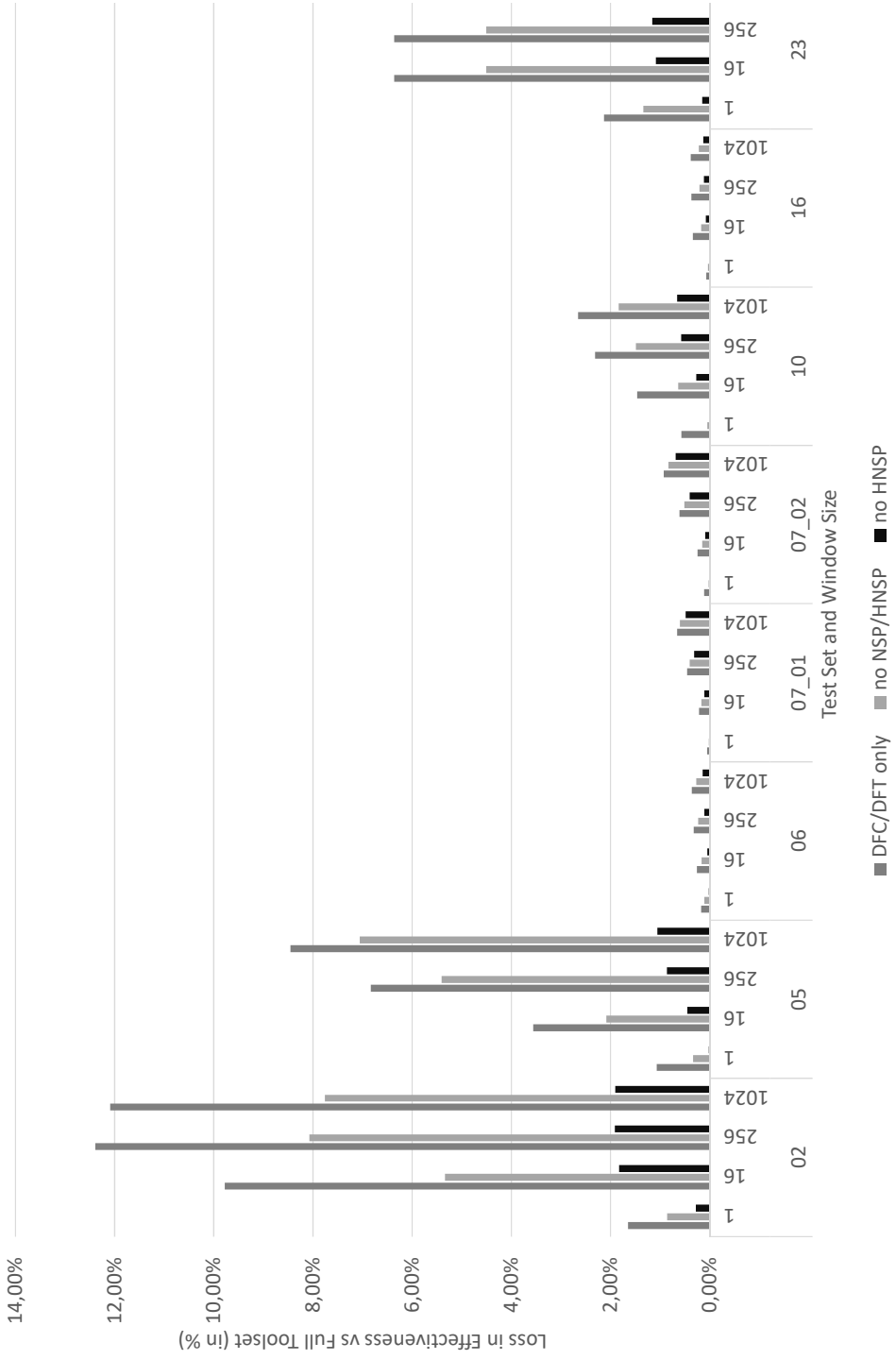

Figure 4.10: Loss in compression effectiveness, compared to the complete toolset.

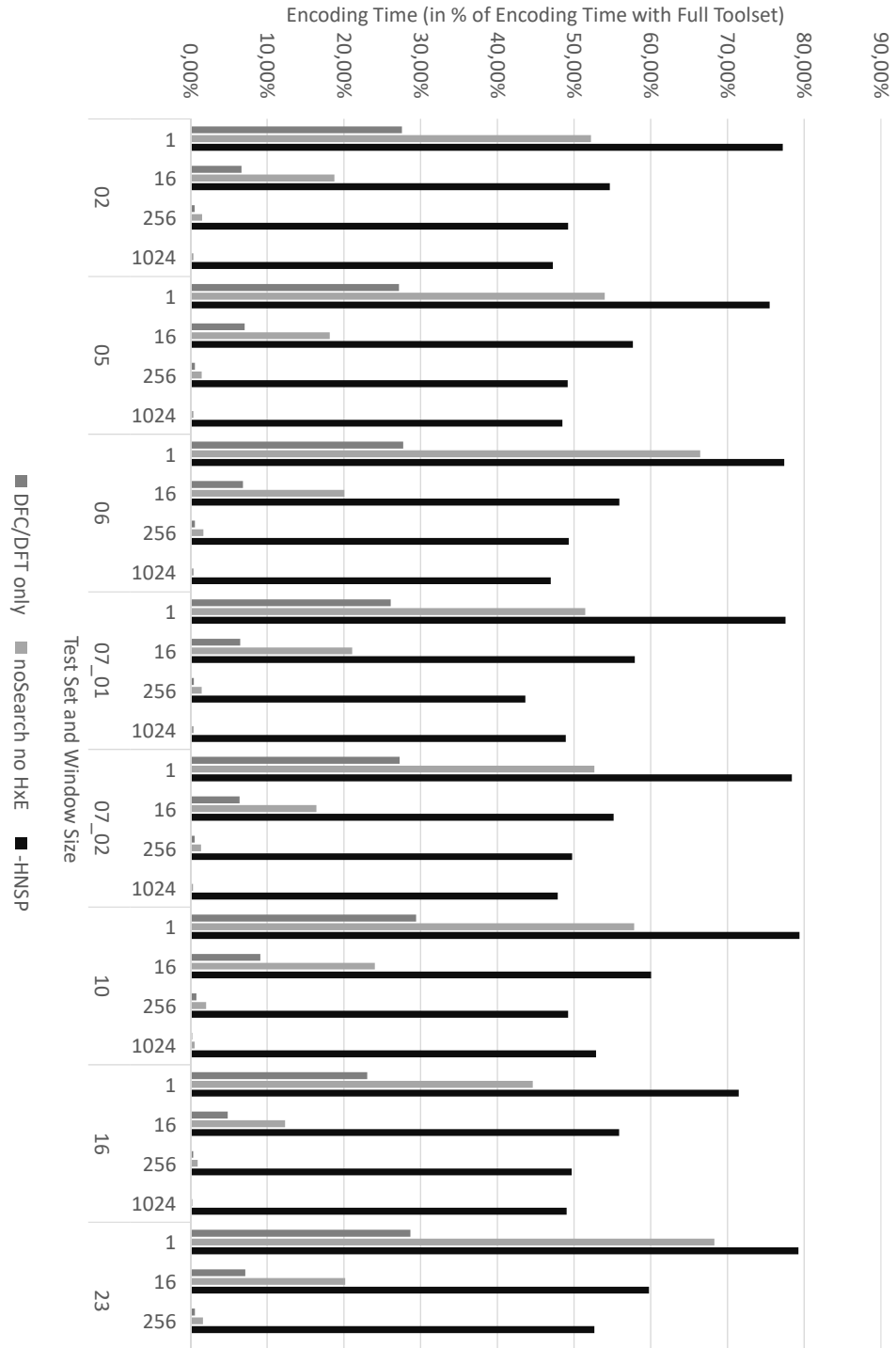

Figure 4.11: Total encoding time, compared to the complete toolset.

4.9 Conclusions and Original Contributions

In this chapter, AQUa, a compression solution for quality scores, built on top of the framework proposed in Chapter 2, was discussed. The designed coding tools are discussed, together with the binarizations of the parameters and the residue generated by the framework and its coding tools. The combination of purpose-built coding tools, together with the selected binarizations of the parameters and the residue, result in AQUa outperforming the commonly used single-pass compression format GNU Gzip and the purpose-built compression format SCALCE by producing output files that are 13.69% to 38.49% and 13.26% to 21.14% smaller, respectively, while still supporting random access. When compared to the state-of-the-art generic compressor 7-Zip (using the LZMA Ultra setting), the proposed framework produces files that are either larger (up to 3.41%) or smaller (up to 6.48%), while still offering random access. Compared to the dual-pass state-of-the-art QVZ compression tool, the proposed framework produces files that are 6.42% to 33.47% larger, except for one file where the proposed framework outperforms QVZ by a small margin of 0.38%, showing that a dual-pass solution may offer a higher compression effectiveness, although such behaviour is not guaranteed.

These results show that, given a properly designed set of coding tools, a block-based single-pass compression solution can outperform generic and specialized compressors when compressing quality scores, while offering support for random access.

AQUa is available for download at `https://github.com/tparidae/AQUa`.

References

[1] C. Kozanitis et al., "Compressing Genomic Sequence Fragments Using SlimGene.", Journal of Computational Biology, vol. 18, no. 3, pp. 401413, 2011.

[2] Ochoa I. et al, "Effect of lossy compression of quality scores on variant calling", Briefings in Bioinformatics, Vol. 18, no. 2, pp. 183-194, 2017.

[3] Malysa G. et al., "QVZ: lossy compression of quality values", Bioinformatics, Vol. 31, no. 19, pp. 3122-3129, 2015.

[4] Canovas R. et al., "Lossy compression of quality scores in genomic data", Bioinformatics, Vol. 30, no. 15, pp. 2130-2136, 2014.

[5] Alberti C. et al., "An Evaluation Framework for Lossy Compression of Genome Sequencing Quality Values", Data Compression Conference (DCC) proceedings, pp. 223-230, 2016.

[6] Hach F. et al., "SCALCE: boosting sequence compression algorithms using locally consistent encoding", Bioinformatics, Vol. 28, no. 23, pp. 3051-3057, 2012.

[7] Roguski L. et al., "CARGO: effective format-free compressed storage of genomic information, Nucleic Acids Research, Vol. 44, no. 12, pp. e114, 2016.

[8] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard", IEEE Trans. Circuits Syst. Video Technol. IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 620-636, 2003.

[9] J. Teuhola, A compression method for clustered bit-vectors, Information Processing Letters, vol. 7, pp. 308-311, Oct. 1978.

[10] M. Wien, "High efficiency video coding: coding tools and specification." Berlin: Springer, 2015.

[11] "Reducing Whole-Genome Data Storage Footprint", http://www.illumina.com/documents/products/whitepapers/whitepaper_datacompression.pdf

[12] "NovaSeq 6000 System Quality Scores and RTA3 Software", https://www.illumina.com/content/dam/illumina-marketing/documents/products/appnotes/novaseq-hiseq-q30-app-note-770-2017-010.pdf

5 Standardization: MPEG-G

5.1 Introduction

During the development of the AFRESh and AQUa frameworks for the compression of nucleotides and quality scores, development of an MPEG (Moving Picture Experts Group) standard for the representation, compression, and management of genomic data was initialized: MPEG-G [1]. The goal of this standardization effort[1] is to provide an alternative to the current de facto standards FASTA/FASTQ [2] and SAM/BAM (Sequence Alignment Map/Binary Alignment Map) [3]. MPEG-G will provide improved effectiveness, additional functionalities (such as built-in support for random access across multiple axes), and standardized processes for the conversion of data in FASTA/FASTQ and SAM/BAM to MPEG-G.

As discussed in Section 1.1.7, the major advantage of standardization is interoperability. In other words, standardization ensures that software and hardware solutions that are standard-compliant can handle all data stored in a standard-compliant bitstream.

As with other MPEG standards, such as the H.264/AVC [4] and H.265/HEVC [5] video coding standards, the MPEG-G standard will be defined by a (set of) decoding syntaxes and processes which can be used by the decoder to access (and decode) the content of each standard-compliant bitstream. This approach allows encoder designers to develop their own encoding algorithms (as long as the output

[1] This effort is performed within the ISO/IEC working group ISO/IEC JTC1 SC20/WG11.

is standard-compliant), with their own algorithms and priorities.
In this chapter, a solution will be introduced and discussed for the compression of the heterogeneous set of data streams that are generated in the MPEG-G standard. This solution has been developed as a simplified, more practical, version of the AFRESh (see Chapter 3) and AQUa (see Chapter 4) coding solutions. The main differences between this solution and AFRESh and AQUA are that:

- the alphabet concept has been replaced by parsing parameters;
- random access has been removed from the coding level, as this feature will be provided by a separate part of the MPEG-G standard; and
- the set of coding tools has been replaced by pipelines consisting of transformation algorithms and binarizations, which can be set on a per-stream basis. This feature enables support for the large set of different, heterogeneous data streams that are described as the inputs to MPEG-G.

The solution introduced in this chapter offers the following features:

- **input data flexibility** - the input data can be parsed according to multiple representations;
- **dual-pass coding**- the input data can be processed in two passes: pass 1 for data analysis (e.g., for look-up table generation), pass 2 for the actual encoding;
- **flexible configuration to optimize coding effectiveness** - the parsed input data is coded through a pipeline consisting of a (set of) transformation algorithm(s) and a (set of) binarization process(es);
- **a unified decoding syntax** - the parameters required for decoding the output is described in a syntax. This syntax has been proposed, and has been used, as a base for the MPEG-G coding syntax [6];
- **a flexible coding solution** - the coding solution allows for easy adaptation to future developments in data pre-processing and representation.

First, an overview will be provided of the different parts of the MPEG-G coding standard, followed by the different data streams that are provided as an input to the coding solution. Subsequently, the proposed coding solution is presented (from an encoder point of view), including an extensive discussion of the different processes that are part of the coding solution. To illustrate the configuration of the different processes in the encoding process, an example will be provided. Following this example, an extensive analysis of the coding effectiveness and efficiency of the proposed solution will be provided. To illustrate the flexibility of the syntax, two sets of configurations (modes) have been designed: fast mode (high efficiency, low

effectiveness), and slow mode (low efficiency, high effectiveness). Both modes will be compared to the state-of-the-art generic compressor 7-Zip (with LZMA ultra settings). In short, the decoding syntax that has been proposed as a decoding syntax for the coding solution of MPEG-G is presented and discussed.

5.2 MPEG-G

The MPEG-G standard is a standard to unify the representation, compression, and management of genomic data. This standard is currently being developed within MPEG[2]. Besides the capability to represent all data that can be contained within the FASTA/FASTQ and SAM/BAM file formats, the standard aims to provide additional capabilities such as random access in multiple dimensions, incremental updates, signaling of encryption mechanisms, and a transport and storage format. The MPEG-G standard consists of five parts, each providing a subset of the capabilities or supportive technologies and processes:

- **Part 1:** Transport and Storage of Genomic Information [ISO/IEC 23092-1]; MPEG-G Part 1 defines the storage and transport file formats used for the exchange of data generated using the technologies defined in Part 2. Part 1 defines, amongst others, the required syntax and technologies to support random access.

- **Part 2:** Coding of Genomic Information [ISO/IEC 23092-2]; MPEG-G Part 2 defines the syntax, pre-processing processes, and compression technologies used for the coding of genomic information.

- **Part 3:** Genomic Information Metadata and Application Programming Interfaces (APIs) [ISO/IEC 23092-3]; MPEG-G Part 3 defines the syntax and semantics for the metadata and APIs for genomic information representation.

- **Part 4:** Reference Software [ISO/IEC 23092-4]; MPEG-G Part 4 provides reference software that can be used to perform conformance testing (Part 5) on the data that are generated by software and/or hardware, according to Part 1, Part 2, and Part 3.

- **Part 5:** Conformance Testing [ISO/IEC 23092-5]; MPEG-G Part 5 provides the description of a test process which allows creators of MPEG-G software to check conformance with the standard as defined in Part 1, Part 2, and Part 3.

[2]The standard will be released January 2019.

The technologies and syntax discussed in this chapter have been presented as a solution (and acted as a starting point) for Part 2 of the MPEG-G standard: coding of genomic information. All research discussed below is represented in the current version of Part 2 of the MPEG-G standard (i.e., as described in the ISO/IEC CD 23092-2 document [7]), and will as such be representative for the performance of the MPEG-G standard.
In the current stage of the MPEG-G standard, Part 2 offers (compared to this solution) the following adaptations and extensions:

- additional binarizations;
- two additional transformation algorithms;
- support for layered transformations;
- compression of Lookup tables;
- signaling of external dependencies;
- revised syntax, to support the signaling of the data required for technologies proposed by other participants that cannot be mapped onto the original syntax, as discussed in this section;
- algorithms for lossy quality score generation and read tokenization.

5.2.1 Descriptor Streams

In this section, the set of data streams (so-called descriptor streams) is discussed that acts as an input for the coding solution that is discussed in the following sections.
In [8], C. Alberti *et al.* proposed a set of descriptor streams that supports the representation of the information contained in FASTA/FASTQ and SAM/BAM files. For each type of data (e.g., read length information, quality scores), a separate descriptor stream is defined that contains a representation of these data, hence creating a heterogeneous set of data streams. These data streams can be categorized, according to their origin, into three categories: sequence reads (i.e., all data linked to nucleotidic data, and mapping of these data), quality scores, and read names (i.e., metadata, including information on the content and/or sequencing process used to generate reads).

The set of proposed descriptor streams contains (per category):

- Sequence Reads:
  - **Pairing information (PAIR)** - contains information regarding the (relative) position of the two segments in a read pair;
  - **Read Mapping Position information (POS)** - contains the position to which a read is mapped in the reference;
  - **Insertions, Deletions, and Substitution Position information (INDP, RFTP, and SNPP)** - contains a list of positions in a read at which CIGAR operations (see Section 1.1.3.2) need to be performed for the read to map it onto the reference[3];
  - **Insertions, Deletions, and Substitution Type description (INDT, RFTT, and SNPT)** - contains the parameters for the CIGAR operations that are signaled in the INDP, RFTP, and SNPP descriptor streams[4];
  - **Read Length information (LEN)** - contains the length of each read, in case of variable read lengths.[5];
  - **Reference Type information (RTYPE)** - identifies the subset of descriptors that is used to encode an unmapped read or unmapped read pair;
  - **Strand information (RCOMP)** - identifies the strand on which the read is mapped;
  - **Substitution Type information (SUBTYPE)**[6];
  - **Soft and Hard Clip information (INDC)** - contains information on clipped bases: their position in the read and the list of nucleotides.[7];
  - **Read Flag information (TFL)**[8];
  - **Unmapped Reads (UREADS)** - contains the reads that are unmapped, these reads are represented in plain format (i.e., in ASCII code).
- Quality Scores:
  - **Quality Values (QVIndex and QVCodebookIdentifier)** - contains the quality scores for each nucleotide;

[3]The INDP and SNPP descriptor streams are deprecated and are now replaced by one single stream MMPOS.

[4]These INDT and SNPT descriptor streams are deprecated and are now replaced by one single stream MMTYPE.

[5]In the current version of the MPEG-G standard, the name of this descriptor stream has been changed to RLEN.

[6]This descriptor stream is deprecated and has been integrated into the MMTYPE descriptor stream.

[7]In the current version of the MPEG-G standard, the name of this descriptor stream has been changed to CLIPS.

[8]this descriptor stream is deprecated and is now replaced by the FLAGS stream

- Read Names:
  - **Tokens (TOKEN)** - contains a tokenized version of the read names;

The subdivision in descriptor streams facilitates random access at the level of the data type and offers the option to apply different compression solutions to the different data types.
Within this chapter, the focus will be on data contained in the descriptors of the sequence reads. Quality scores and read names can be coded with the proposed coding format. However, these descriptor streams allow for significant gains in compression effectiveness by encoding the data lossy (in case of quality scores) or by exploiting the fixed syntax in read names[9].

5.2.2 Data Classes

Besides descriptor streams, an additional dimension of random access is offered, based upon the type of mapping onto the reference of the read this data is corresponding to. To facilitate this random access dimension, the information in the descriptor streams are split in multiple descriptor streams (one for each type of mapping).
Six different classes were defined, based upon the corresponding mapping type:

- **P-Class** - reads matching perfectly to the reference sequence;
- **N-Class** - reads containing mismatches of type N only;
- **M-Class** - reads containing substitution mismatches only;
- **G-Class** - reads containing substitution mismatches, indel mismatches and soft clips;
- **U-Class** - unmapped reads;
- **HM-Class** - read pairs where only one read is mapped[10].

[9] There is no syntax described in the FASTA/FASTQ and SAM/BAM specifications. However, some manufacturers and/or genomic archives defined a fixed syntax for read names, as discussed in Section 1.1.4.

[10] This class has been added in a later stage. Therefore, test files were not available for the experimental results (see Section 5.6).

5.3 Random Access within MPEG-G

As discussed in Section 5.2.1 and Section 5.2.2, the data contained in an MPEG-G dataset is split into different descriptor streams and classes. Additionally, the dataset can be split per genomic region. Splitting the data (and hence supporting random access) across these three dimensions offers significant efficiency gains for many applications.

As an example, to analyse gene expression, one can count the number of reads for a specific region by only accessing, decoding and processing the POS descriptor streams (and RLEN descriptor streams in case of variable length reads) that correspond to the genomic region of interest. With the data contained in these descriptor streams one can calculate the coverage for each genomic position within the region of interest. It is clear that this is a significantly smaller set of data that needs to be decoded (and transmitted or read from storage).

Another example is SNP-calling. For SNP-calling, it is sufficient to analyse those parts of the data that map onto the genomic region of interest and contain perfectly mapped reads (P-class) or reads that map with mismatches (M-class and G-class). From all the descriptor streams of these classes, one only needs the POS descriptor streams to identify the positions of the reads and the SNPP and SNPT streams to respectively calculate the exact position of the mismatch and the type of the mismatch. From the P-class, only the POS descriptor stream (and the RLEN descriptor stream) are needed, to calculate the coverage at the mismatch positions.

5.4 Encryption, Privacy & Integrity

MPEG-G provides built-in support for signaling encryption, privacy and integrity information. This information can be signaled for different levels of the MPEG-G file, down to the level of a block (i.e., data corresponding to a given genomic region, of a given data class and of a given data stream) and up to the level of a complete dataset.

5.5 Proposed Coding Solution

In this section, the coding solution that has been proposed as a baseline for the MPEG-G coding solution for descriptor streams will be discussed. An overview will be provided of the different processes that are part of the coding solution.
As discussed in Section 5.2.1, the descriptor streams that form the input to the MPEG-G coding solution are heterogeneous: each descriptor stream has its own combination of alphabet (ranging from small alphabets (e.g., [0-4]) to large alphabets (e.g., [0-2^{32}])), inter-value dependencies, and probability distributions that can be exploited. Furthermore, these dependencies and probability distributions can vary significantly between descriptor streams of the same type, depending on the used sequencing technologies, sequencing parameters (such as coverage) and processing pipelines.
Therefore, a novel coding solution has been designed, based upon the CABAC and binarization layers of the AFRESh and AQUa framework, that offers an (extensible) set of transformations and binarizations[11], and multiple approaches for context selection, allowing the coding solution (and other implementations of the proposed syntax and processes) to select its preferred approach to coding the different descriptor streams. Figure 5.1 displays the different steps of the coding process, as implemented in the proposed coding solution and described in the corresponding syntax (see Section 5.7). In this section, the different steps will be discussed (from an encoder point of view), including the different parameters that are set for each step, followed by an example encoding process. The corresponding syntax will be discussed in Section 5.7.

5.5.1 Input Data Parsing

In step 1 of Figure 5.1, the input descriptor streams are parsed as described in the descriptor stream specifications in [8], as such creating a sequence of values[12]. In case of complex descriptor streams, such as pair descriptors, the different syntax elements are split into separate streams[13], which are named substreams. Each substream contains one type of data, both in semantic sense (e.g., nucleotides or positions) as in representation sense (e.g., 8-bit bytes or 32-bit integers).

After parsing the input descriptor stream, each substream that has been generated by the input parsing step is provided to the value transformation step (step 2 in Figure 5.1).

[11] In the current status of MPEG-G, new binarizations have been added by other contributors.

[12] It is not obligatory to decode the input descriptor streams according to the description. E.g., an encoder can decide to parse a descriptor stream consisting of 32-bit integer values as 8-bit byte values.

[13] It is not obligatory to split complex descriptor streams into their substreams[14]. E.g., an encoder can decide to parse the complex descriptor stream as a single substream and apply, e.g., Longest Match transformation.

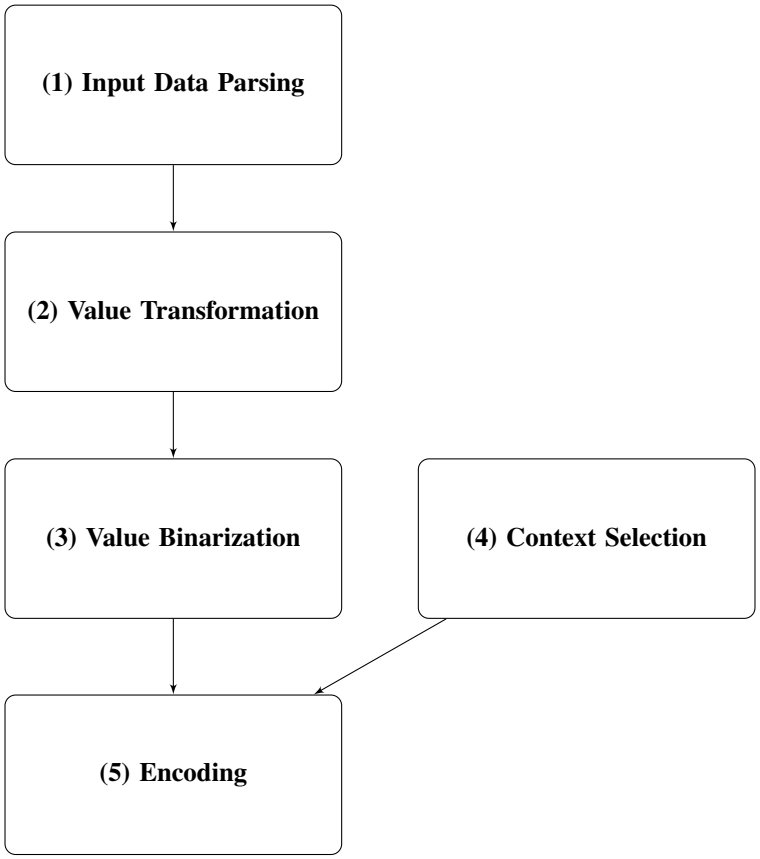

Figure 5.1: The different steps used by the proposed coding solution for MPEG-G.

5.5.2 Value Transformation

Step 2 in Figure 5.1 is the transformation step. In this step, a transformation can be applied to the values in the substream[15]. In the proposed coding solution, five different transformation types are defined:

- **Look-up table transformation** - each value in the input substream is replaced by a value that is represented in a look-up table. This look-up table is described in the header and can be:
  - **zero-order** - each value is replaced by a value using one fixed look-up table (i.e., independent of previous values); or

[15]Transformation is an optional process. If transformation is not required, the input substream will be provided to the next processing step unaltered.

  - **first-order** - each value is replaced by a value, using a look-up table that is selected based on the previous value.

- **Differential transformation** - each value in the input substream is replaced by the arithmetic difference with the previous value. The initial previous value is assumed equal to 0x00 or its equivalent in the representation of the input data (e.g., 0x00000000 in case of 32-bit integer values).

- **Equality transformation** - each value in the input substream is replaced by a maximum of two values, each in a separate substream:

  - The first substream contains 1-bit equality flags, indicating if the value is equal to the previous value in the substream (1) or not (0).
  - If the equality flag in the first substream is equal to 0 (i.e., the input value is different from the previous value), the correct value is stored in the second substream. Values larger than the previous value are replaced by the value `value-1` as the equality flag indicates that the value is not equal to the previous value.

  The two substreams generated by this transformation are then processed separately in next processing steps.

- **Previous Read transformation** - this transformation is a variation on the Equality transformation that can for instance be used in case of unmapped reads, where the reference value is not the previous value but the value `read_size` values ago. In other words, the value that was at the same position as the current value but in the previously encoded read.

- **Longest Match transformation** - a sequence of values in the input is replaced by two values. The two values are stored across three different substreams:

  - The first substream contains length values, representing the length of the match.
  - If the length value in the first bitstream is equal to 0 or 1, no match longer than one value has been found and the value is stored in the second substream.
  - Else, a pointer to the matching sequence in the previously encoded values (the range of encoded values can be limited to a given `window_size`) is stored in the third substream.

After the transformation step, the substream(s) created in this processing step are provided to the binarization step (step 3 in Figure 5.1).

5.5.3 Value Binarization

Step 3 in Figure 5.1 is the binarization step. In this step, each input value is transformed into a set of bits (called bins). To generate this set of bins, six binarization processes are available[16]:

- **Binary [9]** - each input value is represented by its 2-base (i.e., binary) representation using a defined number of bits (`cLength`). Table 5.1 shows the Binary binarization for input value 3 for a `cLength` value of 2 and 4.
- **Truncated Unary [9]** - each input value N is represented by N 1-values followed by a 0. If N is equal to a defined maximal value (`cMax`), the trailing 0-value is discarded. Table 5.2 shows the Truncated Unary binarization for input values 0 to 3 with `cMax` equal to 3.
- **Exponential Golomb [10]** - each input value N is represented by a prefix and a suffix. The suffix is equal to the binary representation of N+1, whereas the prefix is represented by a sequence of 0-bits with a length equal to the length of the suffix minus 1. Table 5.3 shows the Exponential Golomb binarization for input values 0 to 8.
- **Signed Exponential Golomb [11]** - each input value N is mapped onto a positive number. This number is then binarized using Exponential Golomb. Negative input values N (N$<=$0) are mapped onto -2*N, whereas positive input values (N$>$0) are mapped onto 2*N$+1$. Table 5.4 shows the Signed Exponential Golomb binarization for values -4 to +4.
- **Truncated Exponential Golomb [9]** - each input value N is represented by (a set of) 1 or 2 binarizations. Input values smaller than or equal to a given value `cMax` are represented by their Truncated Unary binarization. Input values larger than or equal to `cMax` are represented by the Truncated Unary binarization of `cMax`, combined with the Exponential Golomb binarization of the input value minus `cMax`. Table 5.5 shows the Truncated Exponential Golomb binarization for values 0 to 4 with `cMax` equal to 2.
- **Signed Truncated Exponential Golomb [9]** - each input value N is represented by their Truncated Exponential Golomb representation of $Abs(input\_value)$ and a sign bit, if the $input\_value$ is not equal to 0. Table 5.6 shows the binarization of values -4 to +4.

[16] This is comparable to the list of binarizations in Chapter 4, with the addition of Truncated Exponential Golomb and using the parameter names as used in the MPEG-G Part 2 specification.

value	cLength	binarization
3	2	11
3	4	0011

Table 5.1: The Binary binarization for input value 3 for different values of `cLength`.

value	binarization
0	0
1	10
2	110
3	111

Table 5.2: The Truncated Unary binarization for values 0 to 3 (`cMax=3`*).*

value	binarization	
	prefix	suffix
0		1
1	0	10
2	0	11
3	00	100
4	00	101
5	00	110
6	00	111
7	000	1000
8	000	1001

Table 5.3: The Exponential Golomb Binarization for values 0 to 8.

value	mapped value	binarization	
		prefix	suffix
0	0	1	
1	1	0	10
-1	2	0	11
2	3	00	100
-2	4	00	101
3	5	00	110
-3	6	00	111
4	7	000	1000
-4	8	000	1001

Table 5.4: The Signed Exponential Golomb binarization for values -4 to 4 and their corresponding mapping for Exponential Golomb.

value	Binarization	
	Truncated Unary	Exponential Golomb
0	0	
1	10	
2	11	1
3	11	010
4	11	011

Table 5.5: The Truncated Exponential Golomb binarization for values 0 to 4 (`cMax=2`).

value	Binarization		
	Truncated Unary	Exponential Golomb	Flag
-4	11	011	1
-3	11	010	1
-2	11	1	1
-1	10		1
0	0		
1	10		0
2	11	1	0
3	11	010	0
4	11	011	0

Table 5.6: The Signed Truncated Exponential Golomb binarization for values -4 to +4 (`cMax=2`).

To allow for effective compression, the combination of transformation (see Section 5.5.2) and binarization should be selected in such a way that the value of each bin is as predictable as possible. A high predictability for each bin allows for high effectiveness of the arithmetic coder (as discussed in Section 2.5) in the encoding step (Step 5).

5.5.4 Context Selection

Step 4 in Figure 5.1 is the context selection step. In this step, the context sets that will be used during the encoding step (Step 5 in Figure 5.1) are identified. Each context set contains the contexts required to encode one input value (using the mapping discussed further in this section). The goal of the context selection step is to switch between context sets in a fixed manner (e.g., based on the previous value), so to allow each context set to adapt to the probability distribution of a subset of input values. In this section, the concept of contexts sets and how the contexts in these context sets are mapped onto the different types of binarizations will be discussed. Subsequently, an overview will be provided of the different context set selection processes.

5.5.4.1 Context Sets

Within this framework, contexts are grouped into context sets. Each context set contains the contexts that are needed to support the encoding of one input value in its binarized representation.
Each bit (bin) in a binarization is identified by its position in the binarization (represented by the value `binIdx`), where the first bin is identified with $binIdx = 0$, the second bin with $binIdx = 1$,...
Each separate context of a context set is identified by a `ctxIdx`, where the first context is identified with $ctxIdx = 0$, the second context with $ctxIdx = 1$,...
Depending on the selected binarization, different context sets are defined:

- In case of **Binary Binarization**, each bin is encoded using a separate context, where $ctxIdx = binIdx$. This is especially beneficial in cases where large representations (e.g., 16-bit code words) are used to encode (mainly) smaller values. In this case, the leading bins can be encoded efficiently.

- In case of **Truncated Unary Binarization**, each bin is encoded using a separate context, so $ctxIdx = binIdx$. This approach allows for an additional layer of modeling, as each input value of the Truncated Unary Binarization is represented by an additional bin. In other words, the context that is for instance applied to the third bin of a Truncated Unary representation models the probability of the input value being 3 when the input value is not 0, 1, or 2.

- In case of **(Signed) Exponential Golomb**, the first `prefix_length + 1` bins are encoded with $ctxIdx = binIdx$. The rest of the suffix bins are encoded in bypass mode. In this mode, the bins are processed by the arithmetic coder with the assumption that all symbols have an equal probability. In bypass mode, no context adaptation is performed and bins are encoded assuming an equal probability distribution between 0 and 1.

- For **Flags** (as in Table 5.6), the bin is encoded with $ctxIdx = 0$.
- In case of **(Signed) Truncated Exponential Golomb**, each of the subsections (Truncated Unary, Exponential Golomb, and Flag) are encoded with their respective context sets, as discussed above.

At the start of the coding of a data stream, all contexts are initialized with a value representing an equal probability for all symbols. The adaptivity of CABAC will result in a quick adaptation to the actual probabilities. This approach was selected as the optimal values for each context can vary widely between different substreams. To allow for codec initialization, the MPEG-G standard will offer the possibility to define initialization values.

5.5.4.2 Context Set Selection

In some cases, it might be beneficial to be able to select a separate context set based upon previous data, e.g., the previous value or the sequence number (`value_sequence_number`) of the value. To select a specific context set, three methods are provided by the proposed coding solution:

- **Single Context Set:** each input value is encoded using the same context set;
- **Context Set Cycle selection:** given `N` context sets, each value is encoded with context set `value_sequence_number` % `N`;
- **Value-based Context Set selection:** given N context sets, each value is encoded with context set $Min((\texttt{transformed\_})\texttt{previous\_value}, \texttt{N})$.

5.5.4.3 CABAC Encoding

The final step of the proposed coding solution is the CABAC encoding step. In this step, the bins of the binarizations in step 3 are encoded, using the contexts that have been selected in step 4. This encoding is performed using CABAC, as previously described in Section 2.5.

5.5.5 Example: *RCOMP

To clarify the workings of the process discussed in Section 5.5, an example will now be given. This example is based on the *RCOMP descriptor stream types, due to their simplicity. As described in [8], The RCOMP-type descriptor streams consist of bytes representing strand information. The bytes represent one of four values:

- **0x00:** both reads are located on the forward strand;
- **0x01:** the first read is located on the forward strand, the second read on the reverse strand;
- **0x02:** the first read is located on the reverse strand, the second read on the forward strand;
- **0x03:** both reads are located on the reverse strand.

	% of values
0	0.18%
1	49.95%
2	49.85%
3	0.03%

Table 5.7: Zero-order frequency distribution for the RCOMP descriptor stream for test file 02.

	0	1	2	3
0	8.64%	3.3%	**87.73%**	0.33%
1	0.31%	0.81%	**98.84%**	0.05%
2	0.01%	**99.39%**	0.6%	0.0%
3	1.95%	5.24%	**81.24%**	11.57%

Table 5.8: First-order frequency distribution for the RCOMP descriptor stream for test file 02 (rows: previous value, columns: current value).

Table 5.7 shows the zero-order distribution of the values in the RCOMP descriptor stream for test file 02 from the benchmarking set, used for MPEG-G (see Table 5.10 for an overview of the benchmarking set). Table 5.7 shows that values 1 and 2 are the most-prevalent values in the descriptor stream, with an almost equal distribution between both values, hence indicating that no significant gains can be made using 0-order compression algorithms[17].

[17] Except by limiting the representation to two bits, the minimum required number of bits to represent four different values.

However, when observing the first-order distribution of the values in the RCOMP descriptor stream (as shown in Table 5.8), it can be observed that values can be predicted based upon the previous input value with accuracies of more than 81%. Hence, a solution that uses 1-order LUT transformation (i.e., a look-up table is selected for each input value, based upon the previous value) is expected to provide significant effectiveness improvements, when compared to 0-order coding. Table 5.9 shows the look-up tables that were generated, based on the frequency information in Table 5.8, with value 0 being the input value with the highest frequency (given a certain previous value) and value 3 being the input value with the lowest frequency.

After selection of the transformation, a binarization has to be adopted. In case of *RCOMP descriptor streams, Truncated Unary binarization offers the best compression performance. Given the different frequency distributions of the transformed values, for the different previous values, it was furthermore decided to provide a set of context sets of size four, using value-based context set selection. As a result, the contexts are adapted to the specific frequency distributions per previous value.

	0	1	2	3
0	**1**	**2**	**0**	**3**
1	**2**	**1**	**0**	**3**
2	**2**	**0**	**1**	**3**
3	**3**	**2**	**0**	**1**

Table 5.9: Lookup tables to be used for the discussed example (rows: previous value, columns: current value).

It should be noted that not all data streams containing *RCOMP descriptor information show the same statistical behaviour. Hence, this set of transformation, binarization, and context selection configurations can be suboptimal for these streams. However, in this case an encoder can select (and signal) a different set of configurations. This is the strength of a standard that only has a specification of the decoder, allowing the encoder to select the optimal settings.

Table 5.10: Overview of the benchmarking set as proposed in [8] (U=Unmapped).

ID	Original Filename	Description
02	NA12878_S1.bam	Human WGS, high coverage (53x), Illumina
02U[18]	NA12878_S1.bam	Human WGS, high coverage (53x), Illumina
03	NA12878.pacbio.bwa-sw.20140202.bam	Human, medium coverage (8.4x), Pacbio
05	9827_2#49.bam	Human WGS, low coverage (2.3x), Illumina
05U[15]	9827_2#49.bam	Human WGS, low coverage (2.3x), Illumina
07_Chr1	ERR174310_1.fastq.gz	Human WGS, Low coverage, Illumina
07_Chr2	ERR174310_2.fastq.gz	Human WGS, Low coverage, Illumina
07U[15]	ERR174310_{1-2}.fastq.gz	Human WGS, Low coverage, Illumina
08-m131003	m131003	Pacbio
08-m131004	m131004	Pacbio
09	sample_2-10_sorted.bam	high coverage (274x), Ion Torrent
10	K562_ cytosol_LID8465_TopHat_v2.bam	RNA-Seq, medium coverage (16x)
20-MH1-1	MH0001_081026_clean.1.fq.gz	metagenomics (human gut), Illumina
20-MH1-2	MH0001_081026_clean.2.fq.gz	metagenomics (human gut), Illumina
20-MH2-1	MH0002_081203_clear.1.fq.gz	metagenomics (human gut), Illumina
20-MH2-2	MH0002_081203_clear.2.fq.gz	metagenomics (human gut), Illumina
20-MH3-1	MH0003_081203.clean.1.fq.gz	metagenomics (human gut), Illumina
20-MH3-2	MH0003_081203.clean.2.fq.gz	metagenomics (human gut), Illumina

[18] Due to external factors, a part of the content of test files 02, 05, and 07 has been stored in a separate ”unmapped” file.

5.6 Experimental Results

In this section, the experimental setup is discussed, as well as the effectiveness and efficiency of the proposed coding solution. For analysis ofthe effectiveness and efficiency, the solution is compared to the LZMA ultra setting of the state-of-the-art generic compressor 7-Zip. First, the experimental setup will be described, including the benchmarking test set as provided by the MPEG-G standardization committee. Subsequently, the effectiveness and efficiency will be discussed for sets of configurations (”fast mode” and ”slow mode”) using the complete benchmarking set (see Table 5.10), followed by an analysis per descriptor stream type and an analysis per test file. The benchmarking set consists of multiple types of libraries (Human WGC, metagenomics, RNA-Seq data) sequenced with multiple technologies (Illumina HiSeq, Pacbio, IonTorrent), and containing different levels of coverage. Finally, the memory usage of the proposed coding solution will be analysed and discussed.

5.6.1 Experimental Setup

To analyse the compression effectiveness (compression ratio) and efficiency (processing speed) of the proposed coding solution for the compression of the different descriptor stream types (as discussed in Section 5.2.1), a diverse benchmarking set consisting of 16 files[19] has been selected by the MPEG-G standardization group [8][20]. An overview of this benchmarking set can be found in Table 5.10. This benchmarking set exists of a variety of test files: files with high and low coverage, files with fixed and variable read lengths, and files with short reads and extremely large reads. To offer the reader an impression of the sizes of the different files and descriptor streams, the total file sizes per test file and per descriptor stream type are shown in Figure 5.2 and Figure 5.3, respectively. The total size of the benchmarking set is 36.81 GiB.
The compression tests were performed on a workstation, equipped with an Intel i7 4790K processor and 16GB of RAM. The test files were stored on a Samsung 850 Pro SSD. The results of the efficiency tests are based on the fastest of five runs[21]. A run starts before reading input data, and ends after all output data has been written to disk. The tests are performed both for encoding and decoding and both for the proposed coding solution and 7-Zip.

[19]The term file, represents a set of descriptor streams that are generated from one input file (i.e., a BAM file or a FASTQ file).

[20]During a further stage in standardization, some of the descriptor streams and classes have been adapted. However, it is to be expected that, given the configurability of the proposed coding solution and the resemblance to the original descriptor streams, the proposed coding solution will offer similar effectiveness and efficiency when applied to these streams.

[21]Given that each test run is an exact repetition of instructions, this allows to filter out delays caused by I/O or external processes.

To keep the efficiency tests manageable, 7-Zip was configured to run in two parallel threads[22]. The proposed coding solution was tested single-threaded as the coding solution has not been designed for multithreaded processing. Given the large set of descriptor streams, combined with the splitting of descriptor streams in random access blocks in real-life applications, it is assumed that encoders can process a significant amount of data streams in parallel, if multithreaded operation is required.

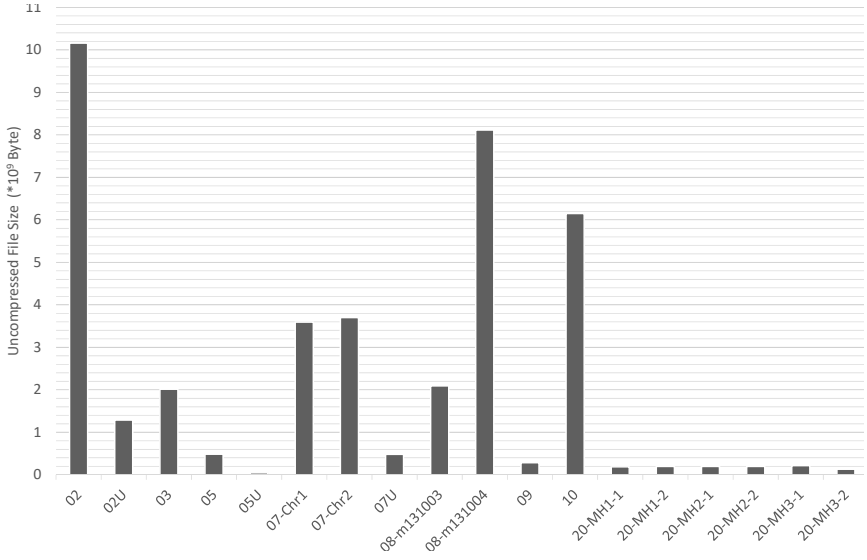

Figure 5.2: Uncompressed file size per test file.

5.6.2 Compression Results

In this section, the compression effectiveness and efficiency of the proposed coding solution is compared to the compression effectiveness and efficiency of the generic 7-Zip compressor at LZMA Ultra settings (dictionary size: 64MB, word size: 64 bits, 2 threads). The results are generated for two sets of configurations with a different trade-off between effectiveness and efficiency, called fast mode (low effectiveness, high efficiency) and slow mode (high effectiveness, low efficiency). As discussed in Section 5.5.2, the difference between these two modes is based on the use of Longest Match transformation for some descriptor streams in slow mode. The configurations and technologies used for the results discussed in this section, are all supported in the current version of the MPEG-G standard.

[22]The maximum number of threads that can be used by LZMA. LZMA2 provides support for more simultaneous threads but requires significantly more memory.

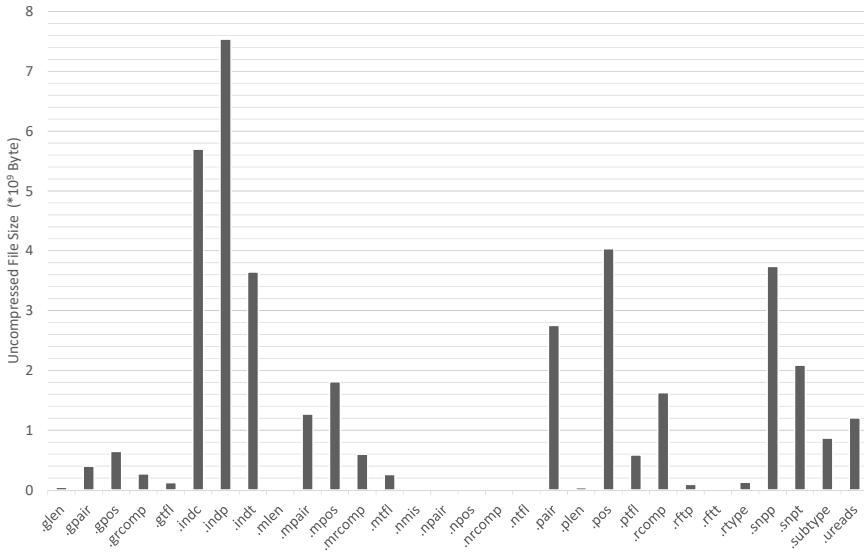

Figure 5.3: Uncompressed file size per descriptor stream type.

Therefore, these results are representative for the effectiveness and efficiency that can be expected from MPEG-G.

5.6.2.1 Compression Results per Encoding Mode

Figure 5.4 shows the compressed size (for the complete benchmarking set) for 7-Zip and the two modes of the proposed coding solution: 7-Zip reduces the input files to 22.25% of the original file size, fast mode to 21.97%, and slow mode to 20.99%.

Additionally, Figure 5.5 shows that both fast (1,471 seconds, or 25,62 MiB/s) and slow mode (4,066 seconds, or 9.27 MiB/s), both single-threaded, are significantly faster than 7-Zip (24,957 seconds, or 1.51 MiB/s when performed using 2 threads)[23].

Figure 5.6 shows that at decoding side, fast mode (1,375 seconds, or 27.43 MiB/s) and slow mode (1,342 seconds[24], or 28.10 MiB/s) are significantly slower than 7-Zip (426 seconds, or 88.56 MiB/s).

When comparing the encoding and decoding efficiency of the fast mode, it is clear that the complexity of the encoding and decoding processes is almost symmetrical. The main difference in speed is caused by the analysis needed to create a

[23] It should be noted that the presented solution contains only limited optimizations (especially, when compared to the highly-optimized 7-Zip). In particular, it is expected that the Longest Match transformation can be significantly optimized for efficiency, as it is currently using a non-optimized search operation.

[24] Slow mode offers a higher decoding speed, which will be discussed in Section 5.6.2.2.

look-up table. The difference between encoding and decoding efficiency in slow mode is higher due to the significantly higher complexity of the Longest Match transformation.

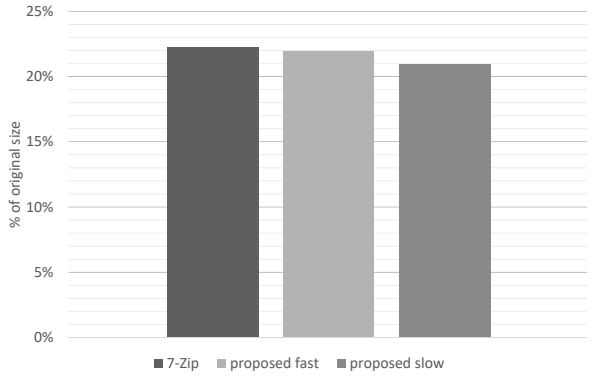

Figure 5.4: Output file size for the complete benchmarking set per encoding mode, compared to 7-Zip.

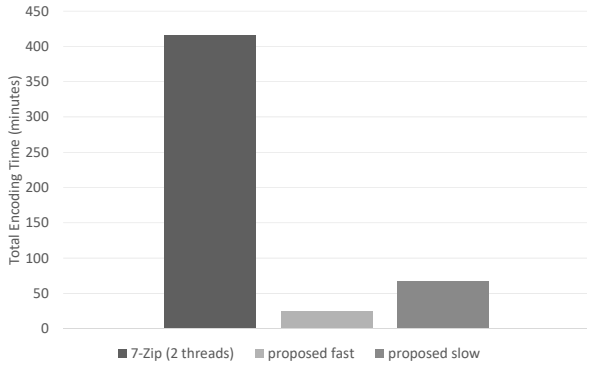

Figure 5.5: Total encoding time for the complete benchmarking set per encoding mode, compared to 7-Zip.

5.6.2.2 Compression Results per Descriptor Stream Type

Figure 5.8 shows the compressed size per descriptor stream type (for all test files combined) for 7-Zip and split per compression mode (fast mode and slow mode for the proposed coding solution, and LZMA ultra mode for 7-Zip). In this figure, it is clearly shown that some SNPT and SNPP descriptor streams can be compressed with a much higher effectiveness by using slow mode (i.e., with Longest Match

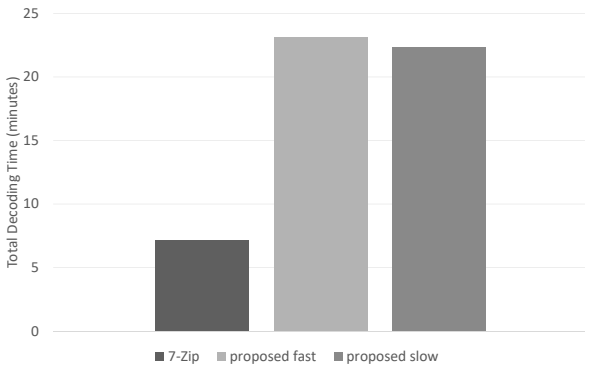

Figure 5.6: Total decoding time for the complete benchmarking set per encoding mode, compared to 7-Zip.

transformation). However, the total compressed size for these descriptor streams is still larger than the total size of 7-Zip.

For some files of the descriptor stream types INDP, INDT, SNPP, SNPT and NMIS, the original configuration, as used in fast mode, has also been replaced by an encoder configuration using Longest Match transformation, hence providing a higher effectiveness for these descriptor stream types when compared to fast mode[25]. In case of NMIS, MLEN, PLEN, and RFTP descriptor streams, 7-Zip outperforms the proposed coding solution by a small margin.

In case of UREADS, containing unmapped reads, the results are based on input files that have been reordered, without reordering the compressed size of UREADS is 15.7% larger (9.0% larger for 7-Zip). An overview of the compression gain per test set by reordering is shown in Figure 5.7.

Figure 5.9 and Figure 5.10 show the total time needed to encode all test files, divided per descriptor stream, and the resulting encoding speed in MiB/s (at input side), respectively. In case of SNPP and SNPT test files, 7-Zip is significantly faster than slow mode, as Longest Match is applied in this case[26]; in all other cases, the proposed coding solution is between 1.39 times and 26.99 times faster.

Figure 5.11 and Figure 5.12 show the total time needed to decode all test files and the resulting decoding speed in MiB/s (at output side), divided per descriptor stream.

In almost all cases, the observations on decoding speed in Section 5.6.2.1 are confirmed. Only in the cases of MLEN, NRCOMP, and NTFL descriptor streams is 7-Zip outperformed by the proposed coding solution. Additionally, the decoding

[25] In case of NMIS the gain is limited, hence not visible in the figure.

[26] It is expected that the Longest Match transformation can be significantly optimized for efficiency, as it is currently using a non-optimized search operation.

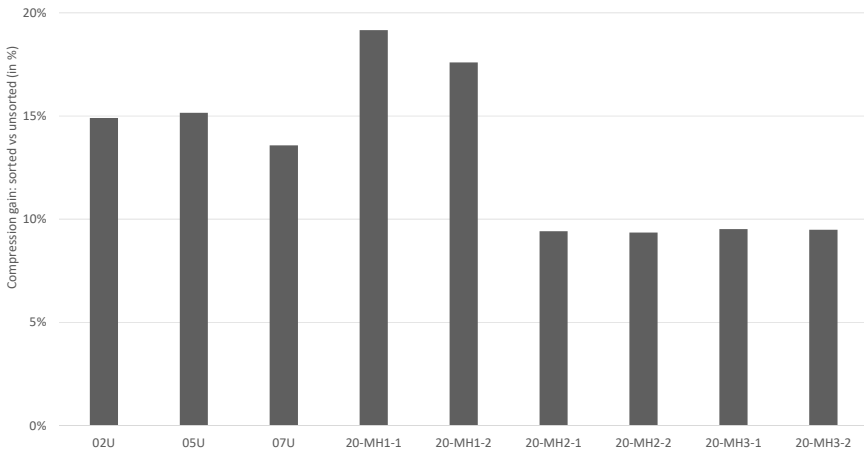

Figure 5.7: Compression gain by sorting UREADS descriptor streams per test set.

time for those modes that are (partly) encoded with the Longest Match transformation (i.e., INDT, INDP, NMIS, SNPP, and SNPT) is (significantly) lower. This is caused by the lower amount of values that have to be unbinarized and decoded using CABAC. Indeed, instead of processing each value separately, Longest Match transformation typically only has to decode one position value and one length value per `length` number of values. This is mainly visible in the file types with larger uncompressed sizes, such as INDP, SNPP, and SNPT. This speed improvement is the cause of the speed difference between slow and fast mode that has been observed in Section 5.6.2.1.

5.6.2.3 Compression Results per Test File

Figure 5.13 shows the compressed size per test file (for all descriptor streams combined) and split per compression mode (fast mode and slow mode for the proposed coding solution, and LZMA ultra mode for 7-Zip). For each of the test files, both encoding modes provide a higher effectiveness than 7-Zip, except for test file 10 (especially for fast mode, which does not use the Longest Match transformation).

Figure 5.15 shows that for all test files (except 10 in slow mode, due to the use of the Longest Match transformation) both fast and slow mode (single-threaded) are significantly faster than 7-Zip (running 2 threads).
Figure 5.17 shows that for all test files, decoding is significantly faster with 7-Zip.

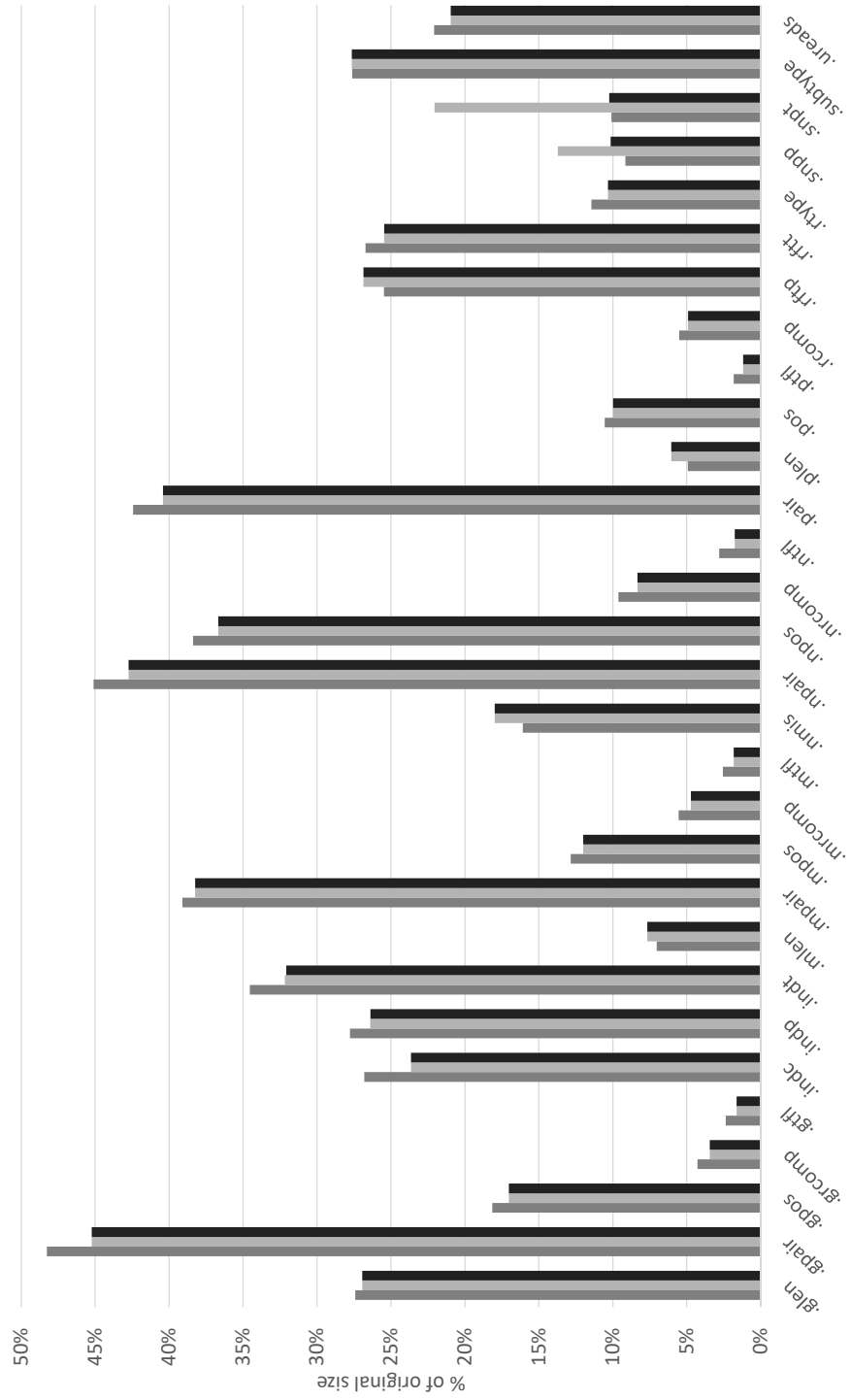

Figure 5.8: Output file size for the complete benchmarking set per descriptor stream type, compared to 7-Zip.

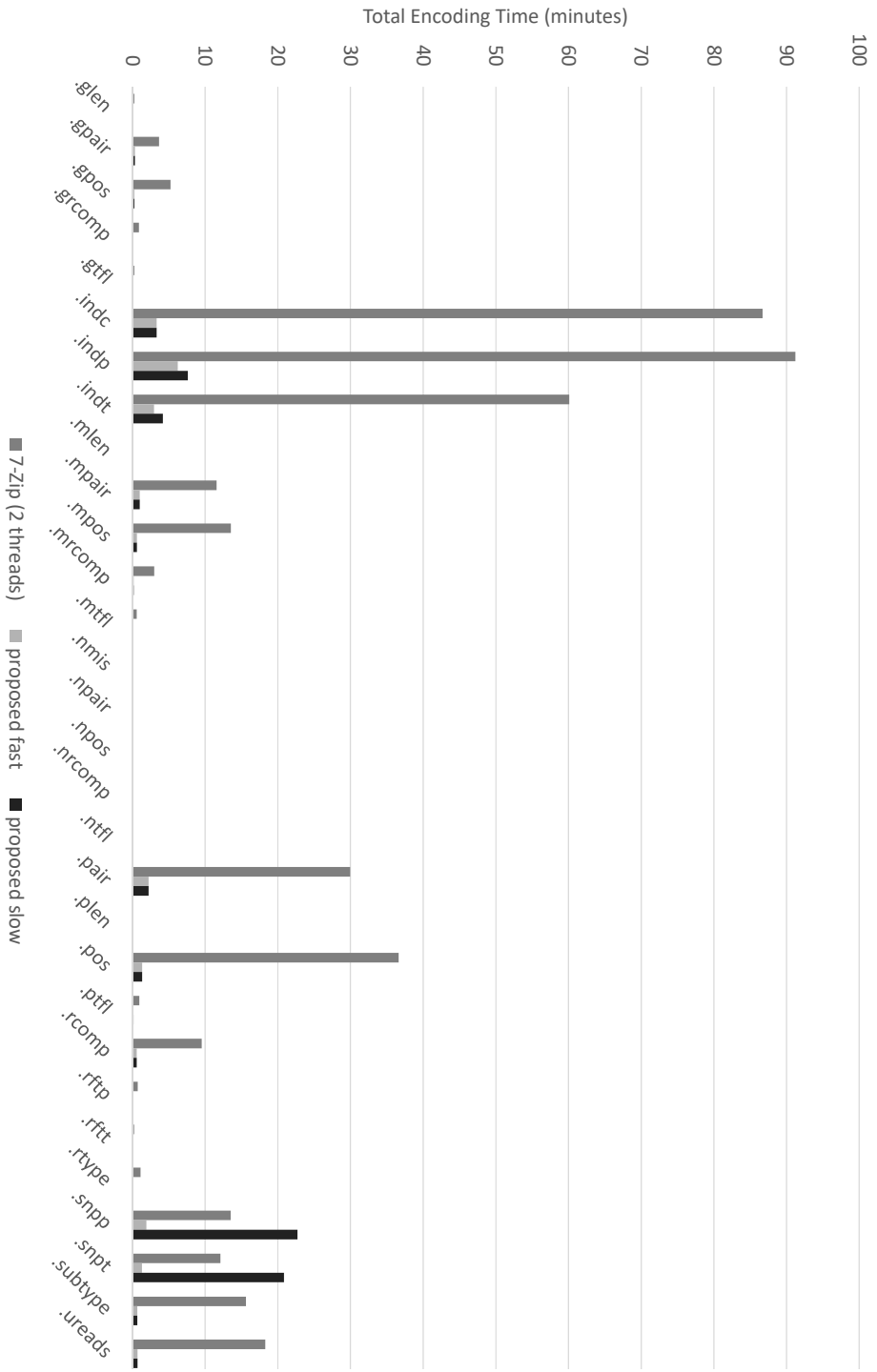

Figure 5.9: Total encoding time for the complete benchmarking set per descriptor stream type, compared to 7-Zip.

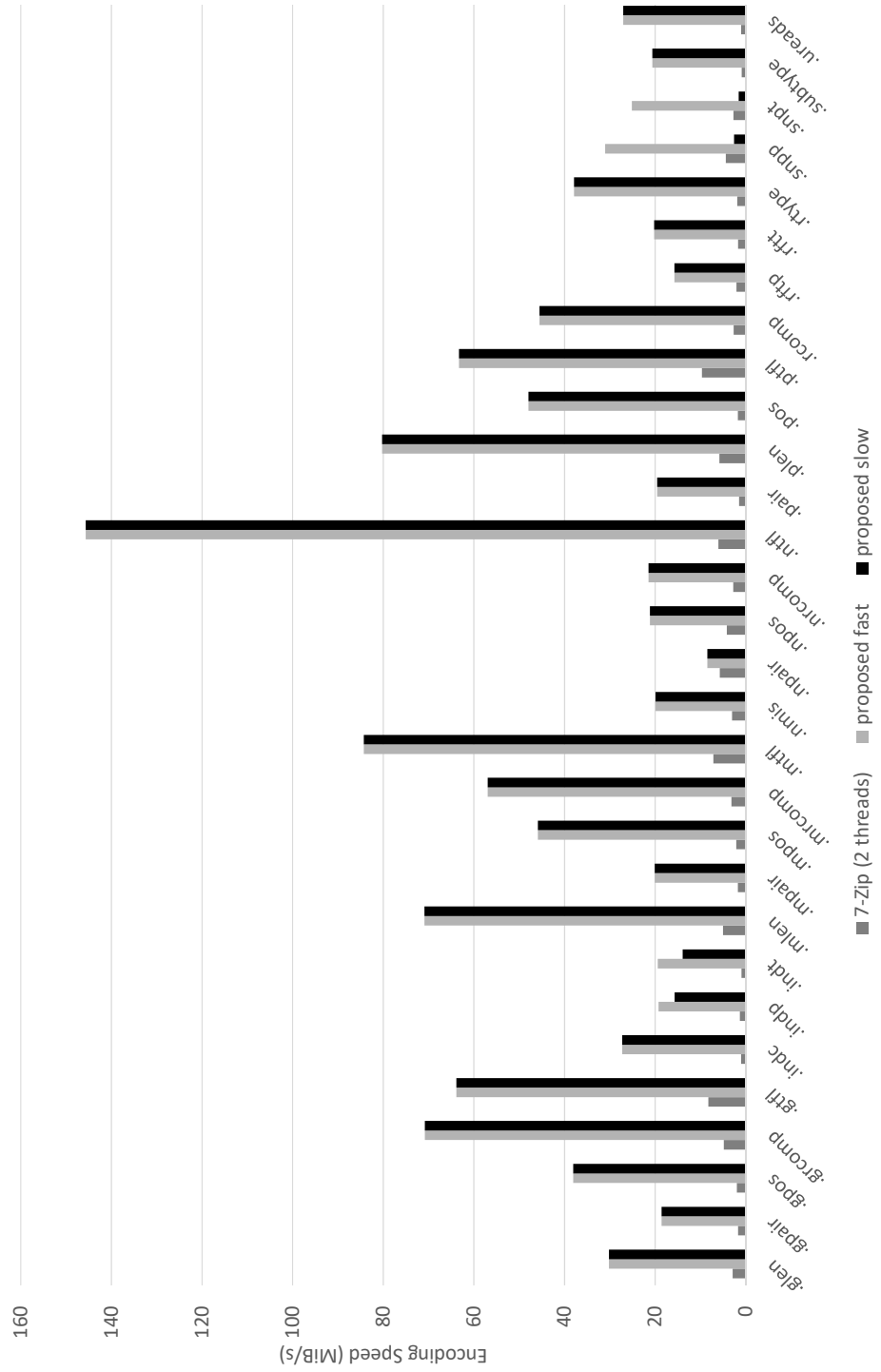

Figure 5.10: Encoding speed for the complete benchmarking set per descriptor stream type, compared to 7-Zip.

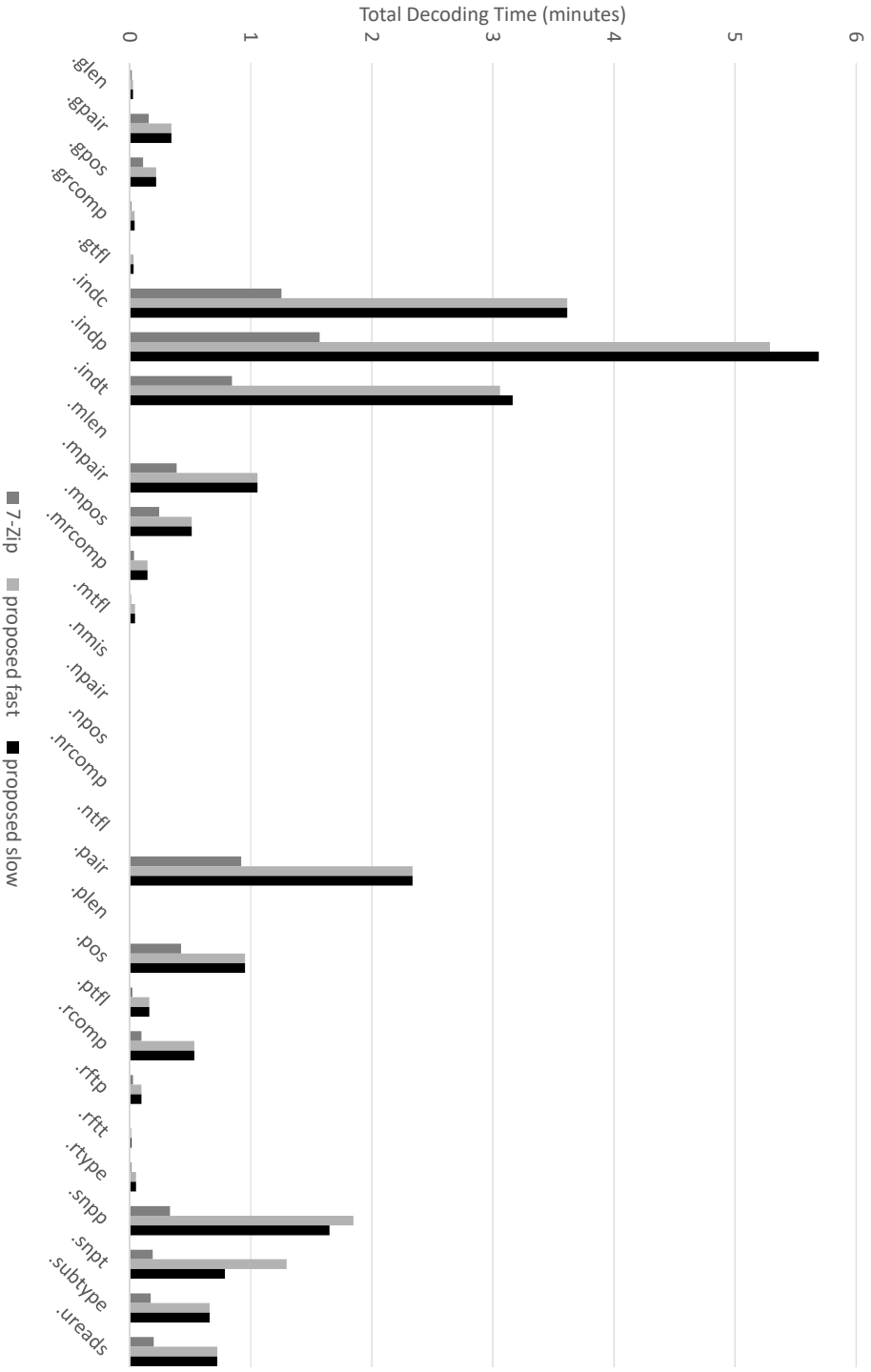

Figure 5.11: Total decoding time for the complete benchmarking set per descriptor stream type, compared to 7-Zip.

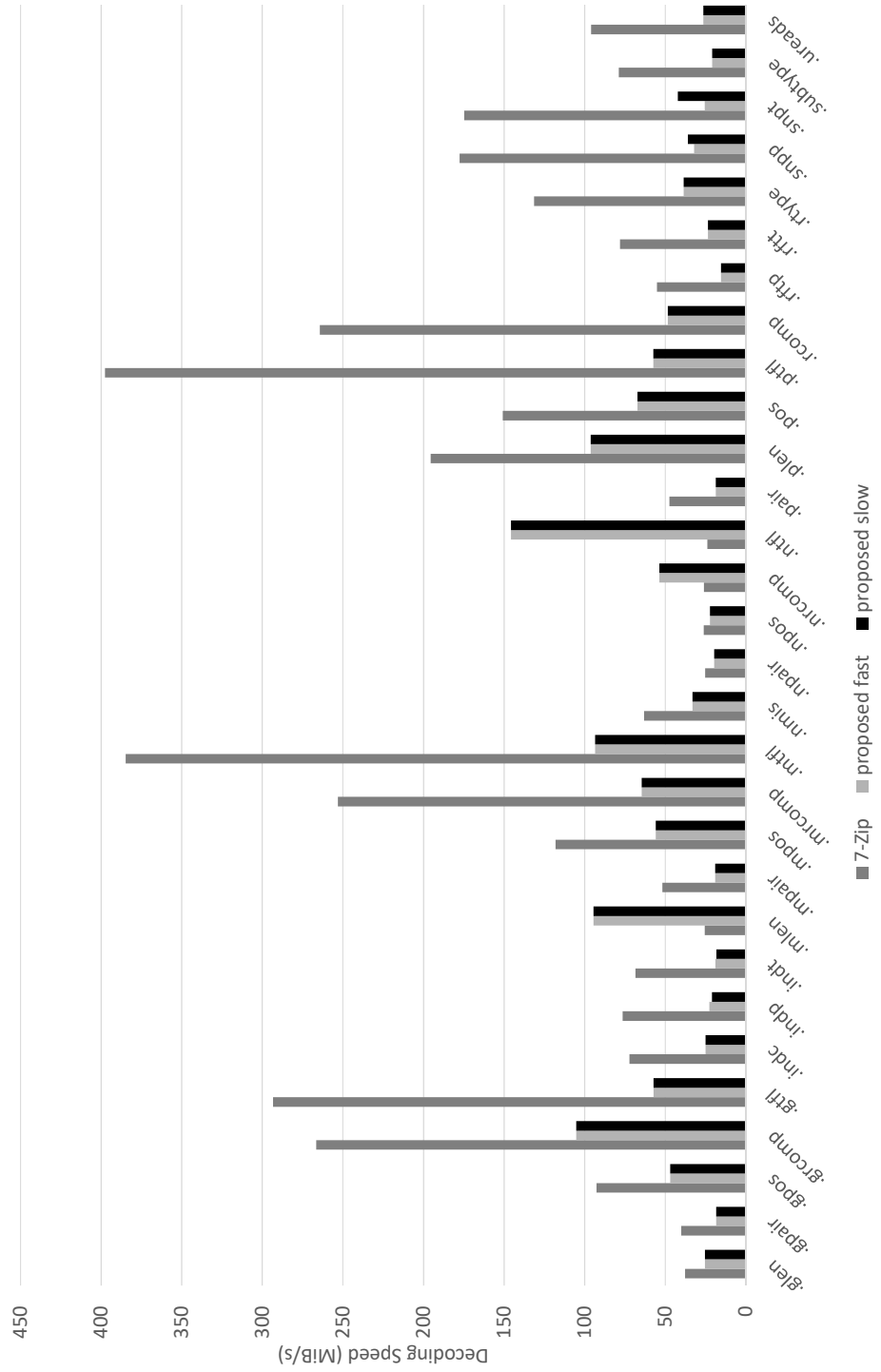

Figure 5.12: Decoding speed for the complete benchmarking set per descriptor stream type, compared to 7-Zip.

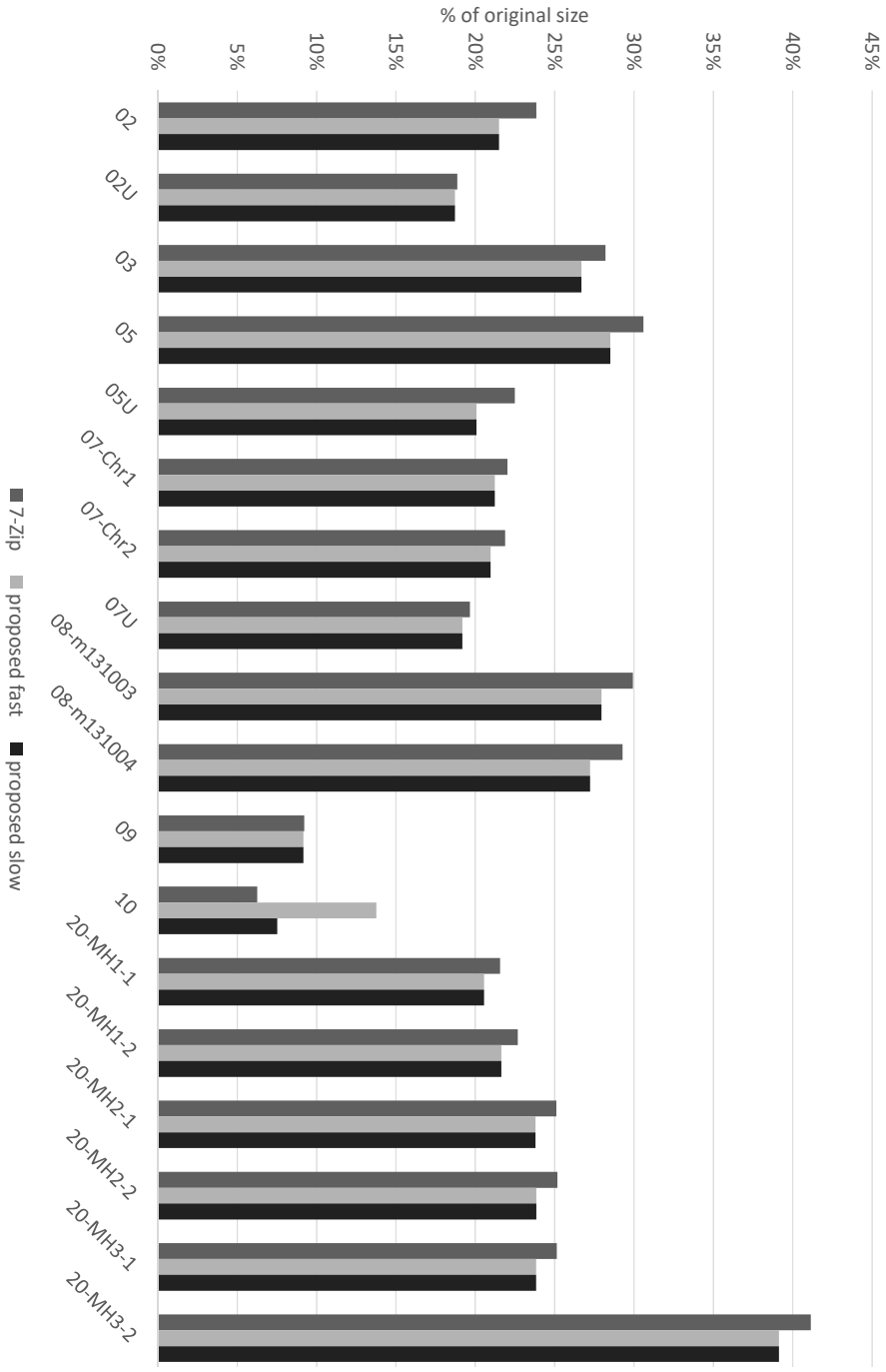

Figure 5.13: Output file size for the complete benchmarking set per test file, compared to 7-Zip.

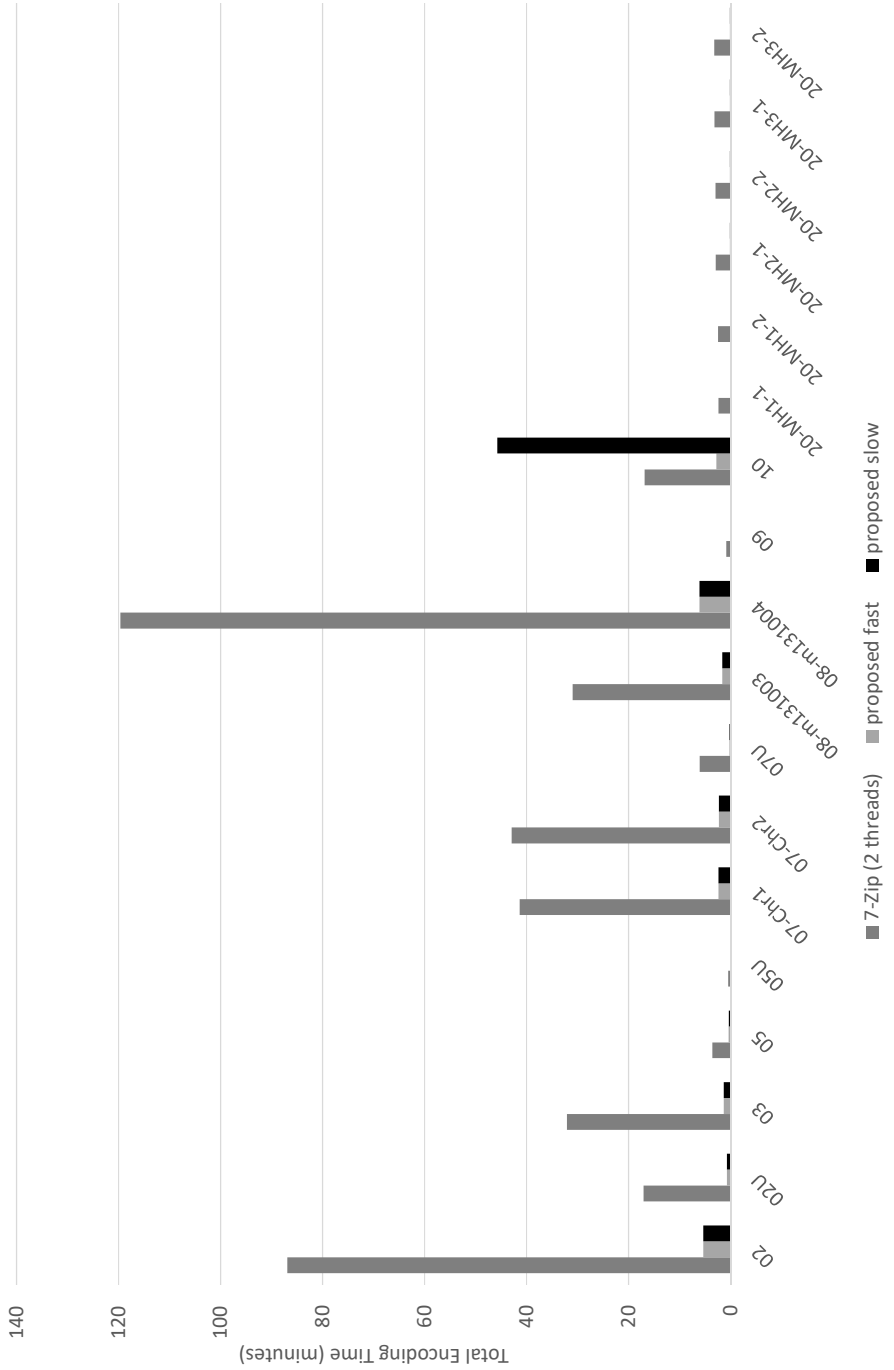

Figure 5.14: Total encoding time for the complete benchmarking set per test file, compared to 7-Zip.

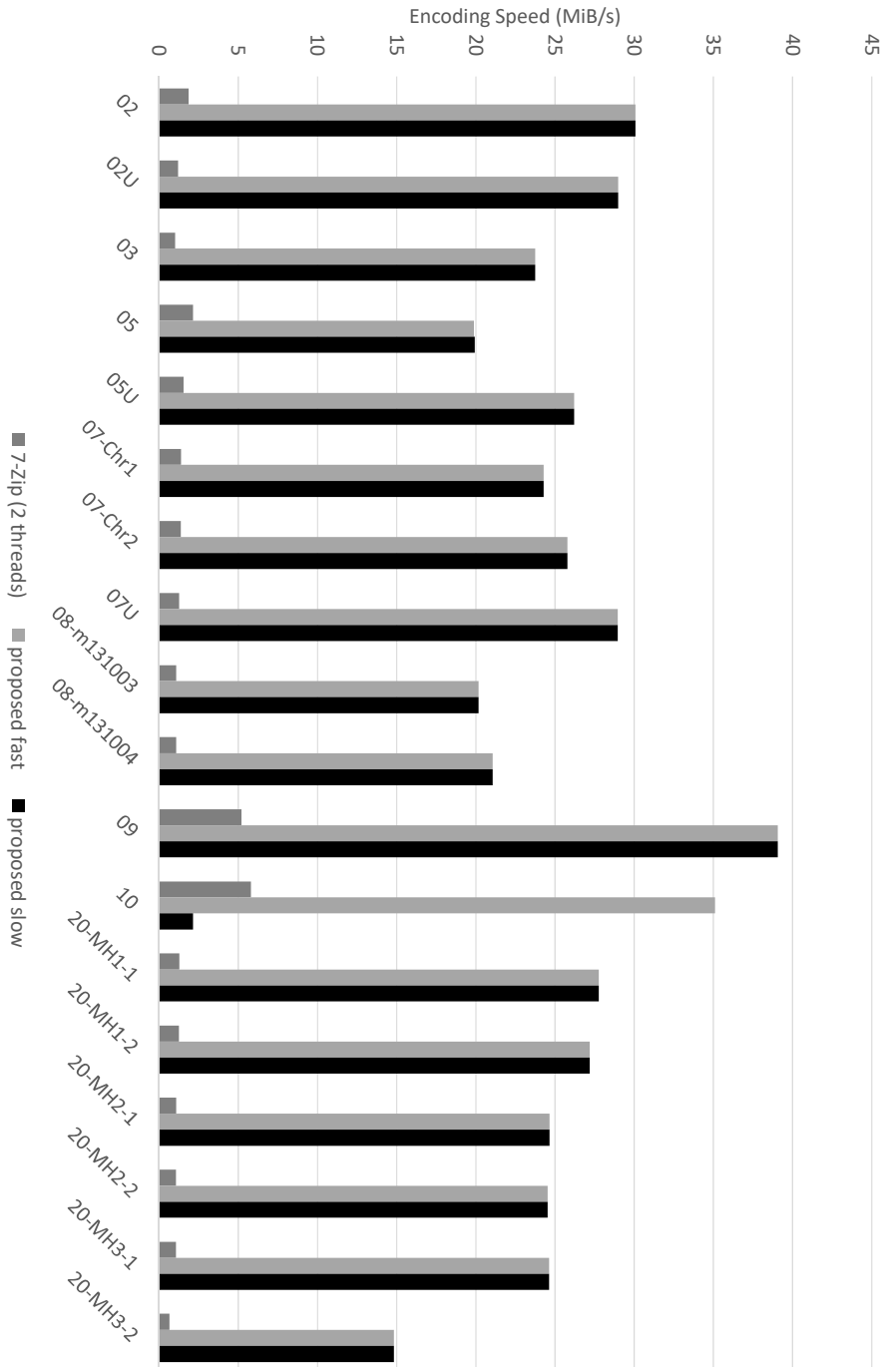

Figure 5.15: Encoding speed (in MiB/s) for the complete benchmarking set per test file, compared to 7-Zip.

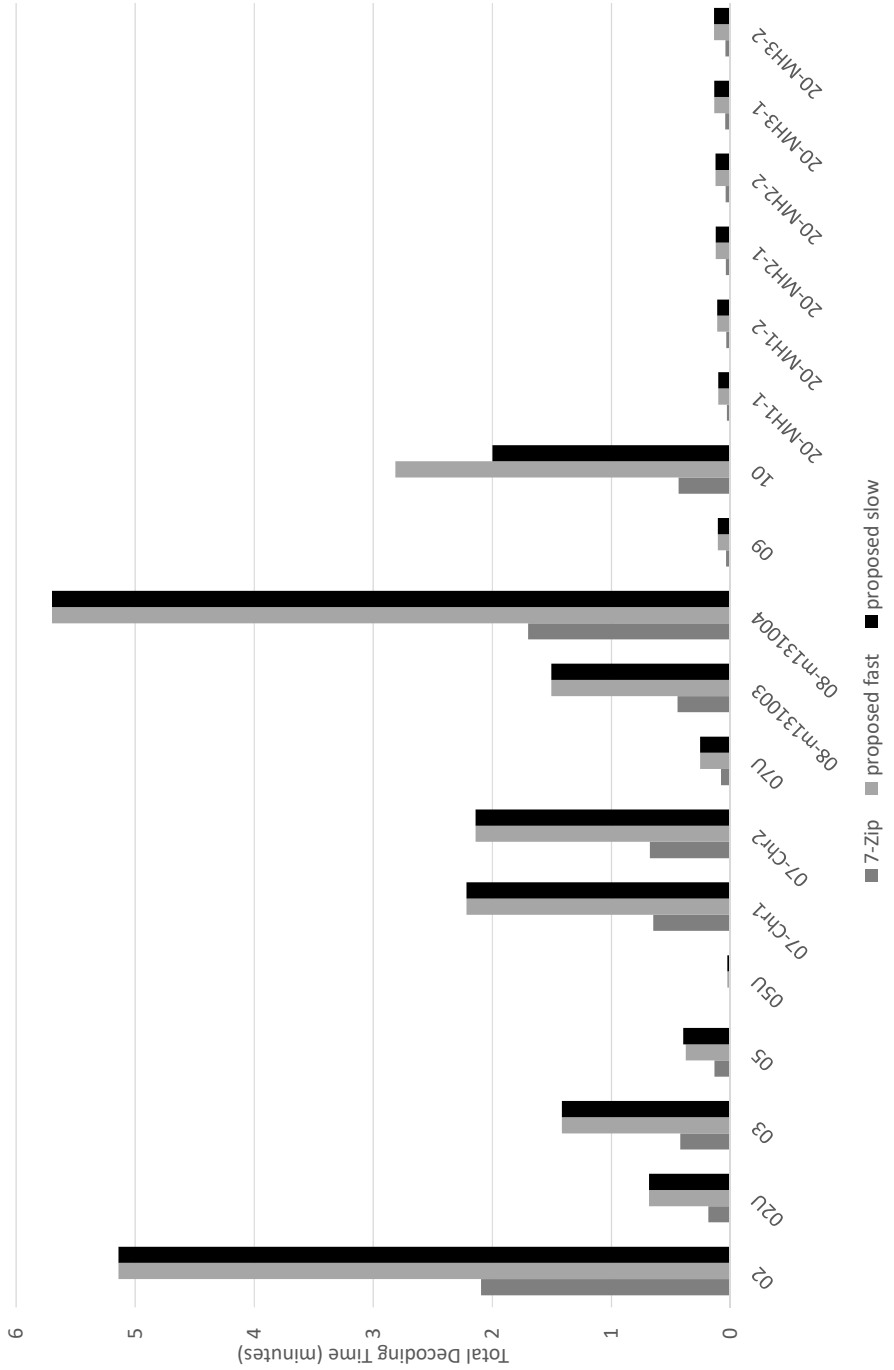

Figure 5.16: Total decoding time for the complete benchmarking set per test file, compared to 7-Zip.

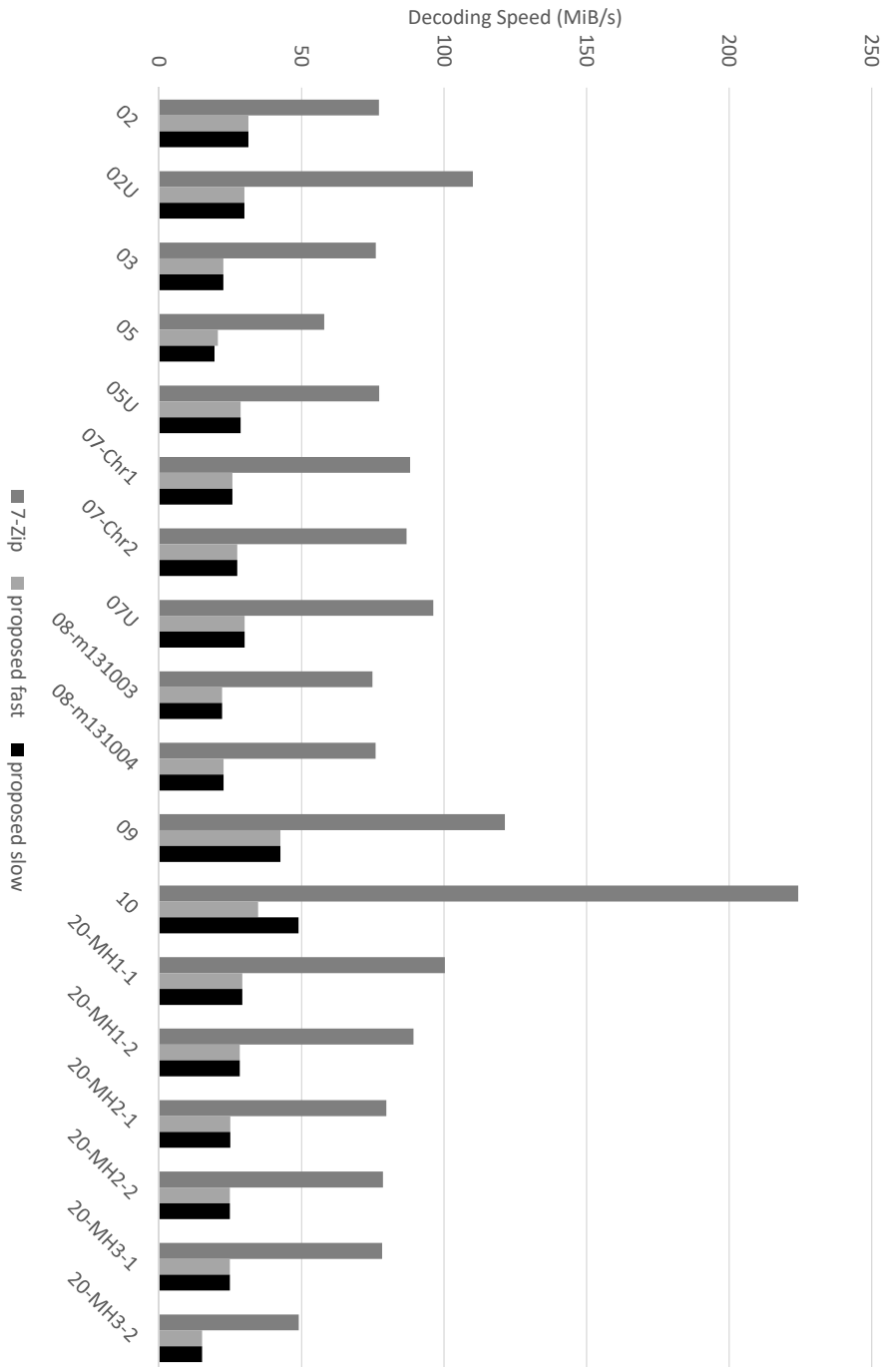

Figure 5.17: Decoding speed (in MiB/s) for the complete benchmarking set per test file, compared to 7-Zip.

5.6.3 Memory Usage

In this section, the memory usage of the proposed coding solution will be compared to the memory usage of 7-Zip.
To encode data with 7-Zip in LZMA at ultra settings, a maximum of 709 MiB is required[27]. As the proposed coding solution is designed in Java, it is hard to measure the actual memory requirements. Therefore, an estimation will be provided, based on a worst-case scenario.
The memory usage by the proposed solution can be divided into multiple parts:

- **input symbol** - the symbol that is to be encoded next. The largest supported input symbol size is 32 bits;
- **previous_input_symbol** - in case of differential or equality transformation, the value of the previous `input_symbol`. The size of this symbol is equal to the size of the input symbol;
- **binarization** - the binarized representation of the `input_symbol` or its transformation. The longest binarization is a Signed Truncated Exponential Golomb representation, which can contain a 32-bit Truncated Unary part, a 31-bit Exponential Golomb[28] part, and a 1-bit Sign Flag, resulting in a total size of 64 bits;
- **context sets** - the context sets that are required to encode the binarized representations. Each context is represented by one byte. The largest context set is the context set that is applied to Signed Truncated Exponential Golomb binarizations: 32 bytes for the truncated unary part, 16 bytes for the Exponential Golomb part[29], and 1 byte for the Sign Flag, resulting in a total size of 49 bytes per context set. Multiple context sets can be available during encoding. During the experimental results, the number of context sets has been limited to five, resulting in a maximum size of 485 bytes;
- **look-up table** - the set of key-value pairs that is used for look-up table transformation. The largest look-up table that was created during the experimental results was 63,360 bytes long[30];
- **search window** - in case of Longest Match transformation and previous read transformation, the previous N encoded `input_symbols` are stored in memory. The largest search window is 32,768 bytes long;

[27] LZMA2, which offers better multithreading performance thanks to the support for more than 2 parallel encoding threads, requires 4413 MiB.
[28] The length of an Exponential Golomb binarization is always odd.
[29] In case of Exponential Golomb coding, the suffix is encoded using bypass mode, i.e., without the usage of contexts.
[30] This is with hard-coded keys. In the current standard, hard-coded keys are replaced by an index in the look-up table, hence lowering the required memory to store the look-up table.

- **CABAC arrays** - the set of arrays that contain information on next states, renormalization,... and are used to improve CABAC processing speed. This set of arrays has a total size of 352 bytes;
- **coding-specific data** - e.g., program code and speed optimizations (e.g., pre-calculated tables for binarization).

Based on the worst-case sizes of each of these parts, one can derive that the maximum memory usage for encoding a descriptor stream (i.e., for integer values, using Signed Truncated Exponential Golomb binarization, with 5 context sets and the largest LUT that was identified in the experimental results together with the CABAC-specific arrays) is below 64 KiB. This memory usage needs to be extended with the coding-specific data. However, it is expected to be significantly less than the additional 708 MiB that would be required to match the memory usage of 7-Zip.
In case of longest match transformation, the maximum memory usage is even lower as there is no LUT to be stored in memory and the maximum search window size is 32,768 bytes long.

5.7 Proposed Decoding Syntax

In this section, the decoding syntax will be discussed that has been proposed as a baseline for the MPEG-G descriptor stream coding. This decoding syntax is based upon the syntax that has been used for the experimental results in Section 5.6. The main difference is to be found in the way the multiple substreams in complex descriptor streams are stored. The software used for the experimental results processes the values of the different substreams in the order of appearance in the input descriptor stream (interleaved). However, this approach required multiple decoding syntaxes with only minor differences. Hence, a generic decoding syntax was designed and proposed for substreams[31], which will be discussed in the next Section[32].

5.7.1 Encoding Parameters Signaling Syntax

In this section, an overview is given of the syntax that has been designed to signal the information required at the decoder side to decode the compressed data streams. Table 5.11 shows the proposed syntax in the same layout as used by the HEVC video coding standard [12]. For each of the syntax elements in this table, the semantics are described in the corresponding subsection.

5.7.1.1 Syntax Semantics

file_size signals the length of the decoded bitstream in bytes. The decoder uses this value to identify when decoding of the bitstream is finished.

wordsize_minus1 signals the number of bytes (minus 1) used for the representation of decoded values. E.g., if `wordsize_minus1` is 0, each decoded value will be represented as one byte.

binarization_id indicates which binarization scheme is used. Table 5.12 shows the list of values for `binarization_id` and their corresponding binarization type. The description of the different binarization types can be found in Section 5.5.3.

cMax signals the `cMax` parameter as used for Truncated Unary, and (Signed) Truncated Exponential Golomb binarization (see Section 5.5.3).

[31] The decoding syntax that has been proposed [6], contained an error that was discovered during the adaptation of the syntax to support the representation of the technologies by other contributors. This error has been fixed in this section.

[32] In the MPEG-G standard, the interleaving process is described per complex descriptor stream.

[33] u(x) defines the representation of the syntax element. u(x) identifies this representation as an unsigned binary representation of length x.

encoding_parameters(){	
file_size	u(64)[33]
wordsize_minus1	u(8)
binarization_id	u(8)
if (binarization_id==0 || binarization_id==3 || binarization_id==5){	
cMax	u(8)
}	
full_repetition_flag	u(1)
if (!full_repetition_flag){	
diff_coding_enabled_flag	u(1)
equality_enabled_flag	u(1)
lut_enabled_flag	u(1)
if (lut_enabled_flag){	
read_lut()	See 5.7.2
second_layer_lut_enabled_flag	u(1)
if(second_layer_lut_enabled_flag){	
for(int i=0;i<main_lut_size;i++){	
read_lut()	See 5.7.2
}	
}	
}	
context_set_size	u(8)
context_cycle_size	u(8)
}	

Table 5.11: Proposed syntax for signaling of the encoder parameters.

full_repetition_flag signals if the bitstream contains a continuous repetition of a single value.

diff_coding_enabled_flag signals if the data are transformed using the differential transformation (see Section 5.5.2).

equality_enabled_flag signals if the data are transformed using the equality transformation (see Section 5.5.2).

lut_enabled_flag signals if the data are transformed using the zero-order look-up table transformation (see Section 5.5.2).

read_lut() parses a look-up table. The syntax for `read_lut()` is described in Section 5.7.2.

binarization_id	Type of Binarization
0	Truncated Unary
1	Exponential Golomb
2	Binary
3	Truncated Exponential Golomb
4	Signed Exponential Golomb
5	Signed Truncated Exponential Golomb

Table 5.12: Values of `binarization_id` *and their corresponding binarizations*

second_layer_lut_enabled_flag signals if the data are transformed using the first-order look-up table transformation (see Section 5.5.2).

context_set_size signals how many different context sets are needed for decoding. One context set is a set of contexts, which are used to decode one value (see Section 5.5.4).

context_cycle_size signals the size of the cycle in which context sets are selected. If `context_cycle_size` is equal to 1, only one context set will be used. If `context_cycle_size` is equal to 2, values at odd positions will be encoded with context set 1, values at even positions with context set 2.

5.7.2 Lookup Table Signaling Syntax

read_lut(){	
lut_size	u(24)
value_length_in_bits	u(8)
for(i=0;i<lut_size;i++){	
key[i]	u($\lceil \text{Log}_2(\text{lut\_size}+1) \rceil$)
value[i]	u(value_length_in_bits)
}	
}	

Table 5.13: Proposed syntax for signaling of a look-up table.

5.7.2.1 Syntax Definitions

lut_size indicates the amount of key-value pairs that are encoded in the look-up table (LUT).

value_length_in_bits indicates the number of bits that are used to encode the values.

key[i] is the i-th element in the table of keys.

value[i] is the i-th element in the table of values.

5.8 Support for new Sequencing Technologies

During the development of the technologies described in this Chapter and the further adaptations and extensions that form MPEG-G, design choices were made in such a way that data produced by all emerging sequencing technologies can be represented using MPEG-G. Each syntax element and each descriptor stream has been designed to support values that would result from these emerging sequencing technologies and evolutions in the output of existing technologies. Examples are the support for the signaling of ultra-long reads (in case of the RLEN descriptor stream) and extensions for access unit start and stop positions to 64 bit for large files, if required.

Additionally, the workflow described in this Chapter has been designed in such a way that it can be configured extensively. This approach is expected to allow for adaptation to properties of the data created by different current and future sequencing technologies.

5.9 Conclusions and Original Contributions

In this chapter, a coding solution (and its corresponding syntax) is discussed that has been proposed as the baseline for the compression of genomic data within the MPEG-G standardization ad-hoc group and has been selected as the starting point for this standard. This coding solution has been developed from a simplified, more practical, version of the AFRESh and AQUa coding solutions and has been extended with features such as, input data flexibility and an extensive coding flexibility, in order to support effective compression of the heterogeneous types of input data, processed by MPEG-G.

The effectiveness and efficiency of the coding solution have been analysed for two different configuration sets (fast mode, offering high efficiency and low effectiveness, and slow mode, offering high effectiveness and low efficiency) and compared to the LZMA ultra settings of the state-of-the-art generic compressor 7-Zip. In slow mode, the complete MPEG-G benchmarking set is compressed to 20.99% of its original file size. In fast mode, the file size is reduced to 21.97%. As a comparison, 7-Zip LZMA with ultra settings compresses the benchmarking set to 22.25% of its original file size. Additionally, with the proposed solution, which is not optimized for efficiency and running in single-threaded mode, the whole benchmarking set can be compressed in 1,471 or 4,066 seconds in fast mode and slow mode, respectively, whereas 7-Zip encodes the same benchmarking set in 24,957 seconds (or 16.97 and 6.14 times slower). To decode the complete benchmarking set, 7-Zip needs 426 seconds, which is significantly shorter than fast mode (1,374 seconds) and slow mode (1,341 seconds).

Besides the significantly higher time efficiency, the proposed solution requires a significantly lower amount of memory. 7-Zip requires a maximum of 709 MiB of memory, the proposed solution less than 64 KiB (excluding coding-specific data, such as program code and potential speed optimizations such as pre-calculated tables for binarization).

In the final part of the chapter, the syntax that is derived from this coding solution and that has been proposed (and selected) as a baseline for the MPEG-G descriptor stream coding is presented and described.

The coding solution discussed in this chapter is available for download at `https://github.com/tparidae/MPEG-G-proposal`.

References

[1] https://mpeg.chiariglione.org/standards/mpeg-g

[2] P. J. A. Cock et al., "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants", Nucleic Acids Research, Vol. 38, No. 6, pp. 1767-1771, 2010.

[3] H. Li et al., "The Sequence Alignment/Map format and SAMtools", Bioinformatics, Vol. 25, No. 16, pp. 2078-2019, 2009.

[4] T. Wiegand et al, "Overview of the H.264/AVC Video Coding Standard", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 7, pp. 560-576, 2003.

[5] G. J. Sullivan, "Overview of the High Efficiency Video Coding (HEVC) Standard", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 22, No. 12, pp. 1649-1668, 2012.

[6] Paridaens T. et al, "Proposal for decoding process of MPEG-G descriptor streams", MPEG input document m41595, http://wg11.sc29.org/doc_end_user/current_document.php?id=60309&id_meeting=172 (restricted access)

[7] ISO/IEC JTC1/SC29/WG11, "CD ISO/IEC 23092-2 Coding of Genomic Information", https://mpeg.chiariglione.org/standards/mpeg-g/genomic-information-representation/study-isoiec-cd-23092-2-coding-genomic , 2017.

[8] Alberti C. et al, "Core Experiments on Genomic Information Representation", MPEG output document N17143, http://wg11.sc29.org/doc_end_user/current_document.php?id=60680&id_meeting=172 (restricted access)

[9] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard", IEEE Trans. Circuits Syst. Video Technol. IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 620-636, 2003.

[10] J. Teuhola, A compression method for clustered bit-vectors, Information Processing Letters, vol. 7, pp. 308-311, Oct. 1978.

[11] M. Wien, "High efficiency video coding: coding tools and specification." Berlin, DE: Springer, 2015.

[12] ITU-T, "High efficiency video coding", Rec. ITU-T H.265, April 2013

6 Conclusions and Future Work

6.1 Summary

The past decade has seen several (r)evolutions in Deoxyribo Nucleic Acid (DNA) sequencing (i.e., reading). Each of these technologies resulted in faster and cheaper sequencing of DNA. Therefore, the use of DNA sequencing has increased significantly, as previous limitations (speed and cost) have been overcome. Many applications (e.g., personalized medicine, Genome-Wide Association Studies (GWAS), and studies of outbreaks of diseases) have now (or will soon) come within reach of more people and (medical) research institutes.
However, with the rising popularity of DNA sequencing, new issues can be identified regarding the storage and transmission of the resulting data. As an example, one human genome can require several hundreds of gigabytes of storage space. This poses a challenge for institutions that want (or are required) to store/archive genomic data or need to work with large sets of genomes for (medical) research. Additionally, transmission of one such human genome can still take hours or days, even over fast broadband networks. As a result, transmission of such large amounts of genomic data is typically handled by shipping hard drives through a courier service.
In this dissertation, several compression solutions have been presented that have been designed with a focus on improved effectiveness and functionalities, such as random access. All solutions have been presented to the MPEG-G standardization committee (in ISO/IEC working group ISO/IEC JTC1 SC29/WG11). The final solution has been selected as the starting point (for processes, syntax, effectiveness,

and efficiency) for the coding part of MPEG-G.

6.1.1 Coding Framework

In this dissertation, a coding framework has been introduced that has been designed to form the baseline for compression solutions for nucleotidic information and quality scores. This framework provides important features for the storage, management, and compression of large data files (inspired by the storage, management, and compression of media files), whilst offering key features for research purposes (such as easy extensibility and configurability). The main features of the coding framework are:

- **single-pass encoding** - input data are processed in one pass, hence enabling features such as live streaming during sequencing or encoding, and limiting memory and storage requirements by voiding the need for temporal storage of analysis data or pre-processing data;
- **stand-alone, no-reference encoding** - input data are compressed without any external reference, hence voiding the need for exchange and management of references, and limiting encoding complexity by voiding analysis of the reference;
- **random access** - data encoded with the coding framework are stored in random access blocks with a minimal impact on coding effectiveness. Each random access block can be decoded separately, hence voiding the need to transmit complete data sets when only a subset of the data is needed.
- **flexible configuration** - the framework can be configured along several dimensions to select a trade-off point between efficiency and effectiveness. The user can identify the symbol alphabet(s) and coding tool(s) to be used, set the granularity of the random access blocks, and other parameters such as size of the search window and blocks.
- **extensibility** - the framework can be extended easily with additional input and output file formats, symbol alphabets, and coding tools.

6.1.2 AFRESh and AQUA

Starting from the presented coding framework, two coding solutions have been designed: AFRESh for nucleotidic information and AQUa for quality scores.

6.1.2.1 AFRESh

The main extensions added to the coding framework by AFRESh are a set of three symbol alphabets and nine coding tools, designed to exploit redundancies within

nucleotidic information (reads and assembled sequences). The coding tools can be split into two major categories: encoding tools and prediction tools. Encoding tools convert input blocks (a sequence of multiple input values, i.e., nucleotides) into a (fixed or adaptive) binary representation. Prediction tools generate a prediction and create correction information (residue). The output data of the coding tools are then converted into bit sequences (binarizations) using tailor-made binarization processes. These binarizations are then processed through Context-Adaptive Binary Arithmetic Coding (CABAC). During the coding process, each of the coding tools is applied. Subsequently, the coding tool offering the highest effectiveness is used.
When compared to generic data compressors (Gzip and the state-of-the-art 7-Zip), AFRESh offers an effectiveness improvement of up to 41% for reads and 34% for assembled sequences, and up to 22% for reads and 16% for assembled sequences, when compared to Gzip and 7-Zip, respectively. When compared to specialized compressors, AFRESh offers an effectiveness improvement of up to 51% (SCALCE), 42% (LFQC), and 44% (ORCOM). Additionally, none of these generic and state-of-the-art compressors support random access, a feature that is deemed very valuable.

6.1.2.2 AQUa

The main extensions added by AQUa are a symbol alphabet for quality scores and a set of seven coding tools. Four of these coding tools are tailor-made for the coding of quality scores; three are inherited from AFRESh. The output data of the coding tools are then converted into binarizations, using binarization processes built on top of the binarization processes that are used in the H.265/HEVC video coding standard. These binarizations are then processed through Context-Adaptive Binary Arithmetic Coding (CABAC).
When compared to the generic compressor Gzip and the purpose-built compression format SCALCE, AQUa offers an effectiveness improvement of up to 38.49%, and 21.14%, respectively, while still supporting random access. When compared to the generic state-of-the-art compressor 7-Zip, AQUa offers an effectiveness that is slightly lower (up to 3.41%) or slightly higher (up to 6.48%), while still supporting random access.
Compared to the dual-pass state-of-the-art quality score compressor QVZ, AQUa has an effectiveness which is lower (6.42% to 33.47%). However, for one test file, AQUa outperforms QVZ by 0.38%, while offering support for random access and processing the data in one pass (without pre-analysis or pre-processing). This shows that, as expected, a dual-pass coding solution typically offers a higher compression effectiveness, although such behaviour is not guaranteed.

6.1.3 MPEG-G Standardization

During the development of the AFRESh and AQUa frameworks for the compression of nucleotides and quality scores, development of an MPEG (Moving Picture Experts Group) standard for the representation, compression, and management of genomic data was initialized: MPEG-G. The goal of this standardization effort[1] is to provide an alternative to the current de facto standards FASTA/FASTQ and SAM/BAM (Sequence Alignment Map/Binary Alignment Map). MPEG-G will provide improved effectiveness, additional functionalities (such as built-in support for random access), and standardized processes for the conversion of data in FASTA/FASTQ and SAM/BAM to MPEG-G.
The major advantage of standardization is interoperability. In other words, standardization ensures that software and hardware solutions that are standard-compliant can handle all data stored in a standard-compliant bitstream.
In a first step, a coding solution has been designed to compress the data streams for the MPEG-G standard. This coding solution, which is presented in this dissertation, is based upon the coding framework that was used as a base layer for AFRESh and AQUa. As random access will be handled in a higher layer of the MPEG-G standard, this feature has been removed. On the other hand, the framework has been extended with support for multiple representations of input data, data transformations (allowing encoders to use analysis information, e.g., through look-up tables), and a unified decoding syntax for signaling input, transformation, binarization, and context selection parameters. Other key features, such as flexible configuration, have been preserved (or extended), hence allowing users to optimize the encoding process for each type of input.
The coding solution consists of four configurable processes: data input, transformation, binarization, and context set selection. In the data input step, the input data is processed in one of three different granularities. In the transformation step, the input data can be transformed using one of six transformation algorithms (or none, i.e., passing through the input data). The output of the transformation step is then processed by the binarization step, which offers six different binarization algorithms. In the final step, a set of contexts is selected for processing the binarizations with CABAC. This selection can be performed using one of three presented context selection algorithms.
To provide the decoder with the information needed for decompression, a decoder syntax has been designed. This decoder syntax has been proposed to the MPEG-G standardization committee and acts as a baseline for the coding part of the MPEG-G compression standard.
When comparing the coding solution with 7-Zip across the MPEG-G benchmarking set, the coding solution offers a higher effectiveness (reduction to 21.97% of

[1] This effort is performed within the ISO/IEC working group ISO/IEC JTC1 SC20/WG11.

the original file size in fast mode, 20.99% in slow mode, compared to 22.25% for 7-Zip). Additionally, the coding solution (which has not been implemented for optimal efficiency and is only executed in single-threaded mode) offers encoding efficiency improvements over 7-Zip (in dual-threaded modus) of 6.14 times (slow mode), and 16.96 times (fast mode).
Another important feature offered by the coding solution is the limited amount of memory required to store (temporary) values during encoding/decoding: a maximum of 64 KiB is required during the encoding/decoding of the complete MPEG benchmarking set for the storage of all transformation data (e.g., look-up tables, search windows), input (input value), intermediate (transformation and binarization), and output data, and context sets and arrays required for CABAC. As a comparison, the total memory required for 7-Zip (in the tested LZMA ultra configuration) is 709 MiB for encoding and 66 MiB for decoding.

6.2 Contributions

The work presented in this dissertation consists of the following contributions:

- A generic coding framework for genomic data, inspired by technologies used for media compression, with support for random access, flexible configuration, and extensibility.
- Use of Context-Adaptive Arithmetic Coding for the coding of other data than video.
- AFRESh, a coding solution, based on the generic coding framework, for nucleotidic information (reads and assembled sequence). This solution offers gains in compression effectiveness of up to 41% for reads and 34% for assembled sequences (compared to generic data compressors), and up to 51%, when compared to specialized compressors. Additionally, the coding solution offers a solution for random access, which is not available in other compressors.
- AQUa, a coding solution, based on the generic coding framework, for quality scores. This solution offers gains in compression effectiveness of up to 39% (compared to generic data compressors). Compared to a two-pass state-of-the-art compressor, effectiveness is between 6% to 33% lower, which is to be expected given the two-pass approach. Additionally, AQUa offers random access, does not require additional memory/storage for the storage of analysis information, and even outperforms the two-pass state-of-the-art compressor by 1% for one test file.
- A coding solution, and corresponding syntax, has been derived from the AFRESh and AQUa solutions and has been extended with the technologies

needed for the compression of the different (heterogeneous) data streams within the MPEG-G standard. This coding solution offers a compressed output size of down to 20.99% of the original size (compared to 22.25% for 7-Zip, using LZMA ultra settings). Additionally, the encoding times are up to 16.96 times smaller than the 7-Zip compressor, with the same settings, and memory usage is significantly lower (64KiB vs 709 MiB).

6.3 Future Work

The work presented in this dissertation presents several opportunities for additional research. These opportunities can be divided into two categories: further improvements in data compression and representation, and applications based on the MPEG-G standard.

6.3.1 Data Compression and Representation

During the discussion of the AFRESh solution in Chapter 3, it was clear that in the case of low-coverage sequencing files and unmapped reads, AFRESh is less effective than the two-pass solution ORCOM. Therefore, it would be interesting to investigate the integration of an (optional) two-pass solution. For AFRESh, a significant compression gain is to be expected from using the ORCOM binning process as a pre-processor as the output of this process is a set of bins that contain reads that share a contingent chunk of nucleotides. The effect of the bin-wise sorting process should also be investigated, as it could further improve the compression effectiveness. If the output of this approach is as expected, this technology might be a strong candidate technology for integration in a future standard (MPEG-G v2), especially for the descriptor stream that contains the unmapped reads: UREADS.

Another potential synergy with MPEG-G can be found in the framework that has been used for AQUa and AFRESh[2]. This would require the extension of the framework with support for the different data types that are used in the descriptor streams. Additionally, the concept of blocks will have to be revised to support random access based on genomic positions. The main incentive to move back to the original framework would be the concept of coding tool set(s). While this concept has a negative effect on coding efficiency, it allows a coding solution to adapt to the different characteristics of (and within) the different descriptor streams. Ideally, a solution consists of a complementary set of coding tools for each of the different descriptor streams. And more importantly, this approach allows for easy extension with (and analysis of) new and/or specialized coding tools and as such enable worldwide development of such tools. This approach has been proven highly successful with video coding standards such as H.264/AVC and H.265/HEVC. It might be beneficiary to extend the concept of coding tools, when compared to the coding tools used in Chapter 3 with transformations and variable binarizations.

A third opportunity to improve the effectiveness of the current MPEG-G standard can be found in new coding technologies. More specifically, machine learning

[2]Note that a simplified version of this framework has been selected as a baseline for the current MPEG-G standard. However, to further improve coding effectiveness, it might be required to return to the original framework with competing coding tools.

technologies such as convolutional auto encoders, which can act as data compressors, or (online and offline) neural networks that generate predictions are interesting solutions to be investigated.

Besides these coding improvements for MPEG-G, it is clear that some types of data that are generated in sequencing and analysis processes are not supported by the current representations in MPEG-G. The main missing type of data is the data that are stored in Variant Calling Format (VCF) files and its significantly larger variant: genomic Variant Calling Format (gVCF). These files contain information on variant calling, which is generated at the end of a typical analysis chain. These data are, as with FASTA/FASTQ and SAM/BAM, stored in a highly-ineffective, human-readable way and contain data that are expected to be representable in an effective way with the current coding approaches used in MPEG-G.

6.3.2 MPEG-G Applications

Given the current state of the MPEG-G standard (DIS, or Draft International Standard), it is clear that the focus should be on promoting the MPEG-G standard and its capabilities. Without general adoption of the standard, development of applications and research on improved effectiveness, efficiency, and functionality will be limited.
A first step towards adoption of a standard is the availability of encoders, decoders, and data management solutions. In a second step (which can be performed in parallel with the first step), existing sequencing and analysis chains need to be adapted and optimized for use with MPEG-G. These adaptations and optimizations can then be used to demonstrate the effect of MPEG-G on the speed of analysis, the storage footprint and bandwidth usage.

6.3.2.1 Encoders, Decoders, & Data Management

Given the MPEG-G standard specification, it is relatively easy to design and implement a standard-compliant decoder and even encoder. However, designing an effective and efficient encoder does require a significant amount of research and development. The results of this effort can be significant, as demonstrated by the evolution of video quality and compression speed of H.264/AVC and H.265/HEVC video encoders since their original design. It is therefore to be expected that significant amounts of research will be performed for implementations, in order to allow for differentiation. Interesting research topics are:

- Optimization of the creation process for descriptor streams from FASTA/-FASTQ or SAM/BAM files. E.g., what is the optimal threshold to apply reference adaptation to optimize the effectiveness of storing mismatches? What is the most efficient approach to generate all descriptor streams, e.g., through parallelization?

- Optimization of the coding efficiency by e.g., parallel processing of the different descriptor streams and random access units?

- Analysis of the effect of the size of access units (i.e., the size of the genomic range contained within an access unit) on compression effectiveness and on the applications. And how can we minimize any negative effect. E.g., If small access units are preferred, how can we optimize the effectiveness? E.g., by using context initialization, selecting different coding parameters, or disabling certain transformations with significant overhead, such as Lookup Tables?

- Analysis of the effect of the different transformations and binarizations on the coding effectiveness and efficiency. What is the effectiveness cost of disabling certain transformations and binarizations in order to limit the processing and memory requirements at the decoder or encoder side?

- Optimization of coding efficiency by limiting the set of coding configurations that are tested during encoding (per descriptor stream). This should result in heuristics for the selection of (near-)optimal coding configurations, potentially depending on the type of input data or on the selected coding configurations in previous access units.

- Efficient adaptation of data contained within an MPEG-G data file. Examples of such adaptations are the conversion from lossless quality scores to lossy quality scores and more complex adaptations such as efficiently updating all descriptor streams when changing the reference sequence (e.g, to a newer version of the reference).

Besides encoders and decoders, solutions are required that support the storage, management and exchange of the data. These solutions need to support the random access functionality of the MPEG-G standard, combined with the encryption, privacy, and integrity functionality. To enable data exchange, solutions are required that allow for identifying the requested data and its storage location and allow for transmitting/streaming these data. Existing standards, such as the MPEG-21 Binary Syntax Description Language (BSDL) for format-agnostic and fast adaptation of files and MPEG-DASH for the streaming of data, are existing enablers for these functionalities. Once these functionalities are supported, a reliable service can be offered to users to be used in their sequencing and analysis chains.
In multimedia, the advent of these technologies enabled a multitude of services and technologies that are highly popular and would not be possible without the continuous process of improving and extending standards: YouTube, Netflix, Hulu, Amazon, streaming on broadcasters' websites, TV distributors that offer hundreds of channels, video/audio recording using smartphones,...

6.3.2.2 Sequencing & Analysis Chains

Provided that encoders, decoders, and data management solutions are available, a significant set of optimizations can be integrated into a range of applications used in sequencing & analysis chains:

- Genome browsers can benefit significantly from the random access functionality on genomic region, data type, and class level. This enables limiting data transmission to only these data that are required for the selected view.

- An analysis chain can benefit significantly from the concept of descriptor streams. This concept allows tools, such as assembly tools, to add mapping information (e.g., mapping position, mismatches, and indels) without the need for compression the complete data set.

- MPEG-G allows for random access on genomic region, data type, and class level. This offers opportunities to speed up many processes. However, these new processes need to be analysed and tested for reliability. Some examples of such processes are:

  - Selection of incorrect mappings for re-assembly. This process can be limited to the selection of reads of class M (and the reads in class G that contain mismatches) with more mismatches than a given threshold and perform the re-assembly. All other types of reads will not be accessed and decoded.
  - Detection of potential contaminations. This process can be limited to the reads that are contained in the unmapped reads descriptor stream. Potentially, this might also require the analysis of those reads that have a mapping with a large number of mismatches.
  - Gene expression analysis. This process can be limited to the descriptor streams that contain the mapping position and (optional) read length data.
  - SNP calling. This process can be limited to the selection of reads of class M (and the reads in class G that contain mismatches) and the reads of class P. SNP calling very sensitive to any change in the complete sequencing and analysis chain. Therefore, the acceptance of this process will require extensive analysis and proven reliability.

- Remote analysis tools can benefit from the random access functionality as they can access the data in the order required and only need to transmit (and decode) the data that are required. This approach can help lowering the duration of an analysis.

To enable adoption of the MPEG-G standard, it is clear that quantification of the gains provided by MPEG-G for genome browsing, local analysis, and remote analysis is highly important, as well as the analysis of its effect on the output of the analysis chain.

6.3.3 The Future of Sequencing

One can wonder what the future of sequencing will bring. Will there be a sequencing technology that is able to sequence a whole genome, transcriptome, metagenome,... in one go, with 100% reliability? Will this minimize the data to be stored, due to the lack of quality scores and 1x coverage? Will this technology become so cheap that it will become a daily practice to e.g., sequence and analyse a persons' genome or, in case of single-cell genomics, multiple sequencing and analysis processes a day, one for each cell type? Will we then store all these data?

What is clear is that the amount of data that will be generated will be of an unimaginable size. This will pose challenges for storage and will probably ask for creative measures and require an analysis of which data really need to be archived and which not.