Basic Research in Computer Science

BRICS DS-01-9 F. F. Rodler: Compression with Fast Random Access

Compression with Fast Random Access

Flemming Friche Rodler

BRICS Dissertation Series **DS-01-9**

ISSN 1396-7002 **November 2001**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/01/9/`

Compression with Fast Random Access

Flemming Friche Rodler

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Compression with Fast Random Access

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Flemming Friche Rodler
April 10, 2002

Another wavelet skimmed in and broke a little further up the sand. A sense of freshness, of expectation was in the air. The great gathered ocean was stirring itself in the distance.

— Red Pottage (1899), Mary Cholmondeley (1859-1925).

Abstract

The main topic of this dissertation is the development and use of methods for space efficient storage of data combined with fast retrieval. By fast retrieval we mean that a single data element can be randomly selected and decoded efficiently. In particular, the focus will be on compression of volumetric data with fast random access to individual voxels, decoded from the compressed data.

Volumetric data is finding widespread use in areas such as medical imaging, scientific visualization, simulations and fluid dynamics. Because of the size of modern volumes, using such data sets in uncompressed form is often only possible on computers with extensive amounts of memory. By designing compression methods for volumetric data that support fast random access this problem might be overcome. Since lossless compression of three-dimensional data only provides relatively small compression ratios, lossy techniques must be used. When designing compression methods with fast random access there is a choice between designing general schemes that will work for a wide range of applications or to tailor the compression algorithm to a specific application at hand. This dissertation will be concerned with designing general methods and we present two methods for volumetric compression with fast random access. The methods are compared to other existing schemes showing them to be quite competitive.

The first compression method that we present is suited for online compression. That is, the data can be compressed as it is downloaded utilizing only a small buffer. Inspired by video coding the volume is considered as a stack of slices. To remove redundancies between slices a simple "motion estimation" technique is used. Redundancies are further reduced by wavelet transforming each slice using a two-dimensional wavelet transform. Finally, the wavelet data is thresholded and the resulting sparse representation is quantized and encoded using a nested block indexing scheme, which allows for efficient retrieval of coefficients. While being only slightly slower than other existing schemes the method improves the compression ratio by about 50%.

As a tool for constructing fast and efficient compression methods that support fast random access we introduce the concept of lossy dictionaries and show how to use it to achieve significant improvements in compressing volumetric data. The dictionary data structure is widely used in computer science as a tool for space efficient storage of sparse sets. The main performance parame-

ters of dictionaries are space, lookup time and update time. In this dissertation we present a new and efficient dictionary scheme, based on hashing, called CUCKOO HASHING. CUCKOO HASHING achieves worst case constant lookup time, expected amortized constant update time and space usage of three words per element stored in the dictionary, i.e., the space usage is similar to that of binary search trees. Running extensive experiments we show CUCKOO HASHING to be very competitive to the most commonly used dictionary methods. Since these methods have nontrivial worst case lookup time CUCKOO HASHING is useful in time critical systems where such a guarantee is mandatory.

Even though, time efficient dictionaries with a reasonable space usage exist, the space usage of these are still too large to be of use in lossy compression. However, if the dictionary is allowed to store a slightly different set than intended, new and interesting possibilities originate. Very space efficient and fast data structures can be constructed by allowing the dictionary to discard some elements of the set (false negatives) and also allowing it to include elements not in the set (false positives). The lossy dictionary we present in this dissertation is a variant of CUCKOO HASHING which results in fast lookup time. We show that our data structure is nearly optimal with respect to space usage. Experimentally, we find that the data structure has very good behavior with respect to keeping the most important elements of the set which is partially explained by theoretical considerations. Besides working well in compression our lossy dictionary data structure might find use in applications such as web cache sharing and differential files.

In the second volumetric compression method with fast random access that we present in this dissertation we look at a completely different and rather unexploited way of encoding wavelet coefficients. In wavelet based compression it is common to store the coefficients of largest magnitude, letting all other coefficients be zero. However, the new method presented allows a slightly different set of coefficients to be stored. The foundation of the method is a three-dimensional wavelet transform of the volume and the lossy dictionary data structure that we introduce. Comparison to other previously suggested schemes in the literature, including the two-dimensional scheme mentioned above, shows an improvement of up to 80% in compression ratio while the time for accessing a random voxel is considerably reduced compared to the first method.

Acknowledgments

I would like to use this opportunity to thank all the people who gave me their time and support during my years as a student.

First and foremost I would like to thank my supervisor, Professor Brian H. Mayoh, for his support, encouragement and confidence. Brian was the first to introduce me to the interesting aspects of wavelet theory and has always inspired me to investigate various interesting topics guiding me in the right directions.

A special thanks goes to Professor Robert M. Gray for letting me visit his research group, Signal Compression and Classification Group, at Stanford University. I would like to thank everyone there for being very friendly. Also, a special thanks to Professor Bernd Girod for providing me with a research project and guidance while at Stanford. Also, thanks to Eckehard Steinbach for many fruitful discussions on my research.

Thanks to Rasmus Pagh for co-authoring several papers. We had many good and productive discussions.

During my time as a Ph.D. student I have had the opportunity to share office with Thomas Hune, Rasmus Pagh and Jacob Pagter. Besides functioning as lunch partners these people have provided me with many interesting discussions both within and outside computer science.

Also, I would like to thank everyone at BRICS in Århus for providing a warm and interesting research environment. BRICS is a very dynamic and pleasant place to work.

A special thanks goes to my undergraduate study group, Claus Brabrand, Thomas Hune and Tom Sørensen. Without them I would never have made it (at least they claim) through the first years at University.

Finally I would like to thank my family, especially my parents, Kirsten and Herbert Rodler, who have given me love, encouragement and support. Also thanks to my brother and best friend, Peter Rodler, for his love and also for always being there to play squash. I would also like to thank my girlfriend, Xenia Andersen, for proofreading my English but also for all her love and understanding, especially during the last two months of writing this dissertation.

Flemming Friche Rodler,
Århus, April 23, 2002.

Contents

Chapter 1

Introduction

"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."

— Alice's Adventures in Wonderland (1865), Lewis Carroll (1832-98).

The main topic of this dissertation is the development of methods for data compression of volumetric data, in particular methods that support fast random access to individual voxels, decoded from the compressed data. As a part of this we introduce the concept of lossy dictionaries and present two techniques, based on hashing. One for solving the dictionary problem and one for solving the lossy dictionary problem.

Volumetric data is becoming increasingly important and common in areas such as scientific visualization and medical imaging of scannings such as Magnetic Resonance (MR), Computer Tomography (CT) and Positron Emission Tomography (PET). For example three-dimensional data obtained from a MR scanner can be used in diagnostics or pre-operational planning. Volume data also occurs in physical simulations, fluid dynamics and many more areas.

Applications that manage and process volumetric data often implicitly assume that the data can be loaded completely into memory. Unfortunately, the size of volumetric data is often prohibitive unless large and expensive computers with extensive amounts of memory are used. To alleviate the memory problem compression methods can be utilized, the idea being that a compressed representation of the volume can be loaded into memory with voxels reconstructed on the fly when needed. In order to facilitate the high compression ratios required, such methods must necessarily be lossy in nature. Normally in lossy compression the only concern is to achieve a good trade-off between rate and distortion. In volumetric compression, however, the time it takes to reconstruct a single voxel from the compressed representation is also a major concern for interactive applications, since volume based applications often access data in unpredictable ways. That is, we strive to achieve a good trade-off between rate, distortion, and reconstruction time. Ideally, if the bit-budget and the reconstruction time can be made small enough for an acceptable distortion then it will appear to the volume based application as if it accessed an uncompressed volume, i.e., the

compression and decompression become transparent. This is illustrated in Figure 1.1. We note that compression methods that are specially designed to work with a specific application might give better results with respect to both compression ratio and access speed than general purpose methods. However, if the application changes or if we wish to use the compression method for a different application then the compression algorithm and data structure may have to be redesigned. The development of general purpose methods are therefore very important for maximum flexibility.

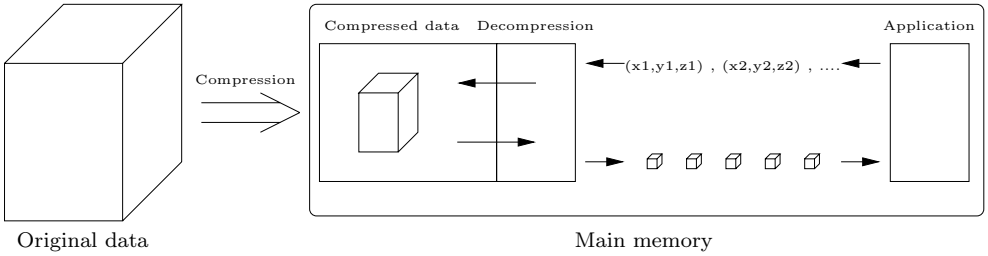


Figure 1.1: The general concept of volumetric compression with fast random access. If decompression is fast enough and the reconstruction quality is good, then decompression may seem transparent to the application.

Wavelet based methods are among the most successful in lossy image compression. Since the coining of the name wavelet in the early 1980s and the introduction of multiresolution analysis by Mallat and Meyer in the late 1980s the theory of wavelets has been developed as a formal framework and analytical tool in mathematics and computer science. Many of the properties associated with wavelets such as localized analysis, good energy compaction, and time-frequency analysis have been and are still applied with success in areas such as computer graphics and signal analysis and processing. Data compression is only one example where the introduction of wavelets has pushed the envelope of state-of-the art algorithms considerably. For example, besides being eminent at energy compaction the hierarchical or multiresolutional representations of wavelets provide an elegant way to describe an image or an volumetric object at different levels of detail. This hierarchical property is useful in compression, especially in compression with fast random access, where it can be used for selective refinement. For example, in a real time application a coarse representation of the volume can be decoded first and when time permits, details can be added for more accurate computations. The significance of wavelets in data compression is evident in the fact that all recent standards such as JPEG2000 and MPEG-4 include wavelet based methods. The two methods for compressing volumetric data with fast random access that are presented in this dissertation are both based on wavelets.

In the search for faster and better compression schemes for volumetric compression with fast random access we introduce the concept of *lossy dictionar-*

ies which we use to gain significant improvements over existing compression schemes. Dictionaries are part of many algorithms and data structures. A dictionary provides access to information indexed by a set S of keys drawn from a universe U, i.e., $S \subset U$. Given a key, the dictionary returns the associated information or reports that the key is not in the set. The important parameters when designing dictionaries are the space usage, the time it takes to look up an element, and how long it takes to update the dictionary, i.e., insert and delete elements in S. Theoretically, there is no trade-offs of these parameters as there exist dictionaries with space usage close to the information theoretical minimum having constant lookup time and expected amortized constant update time. However, large constant factors and difficulty of implementation hinder practical use of these data structures. By using more space very fast data structures can be constructed. The challenge which we consider in this dissertation is therefore to combine speed with a reasonable space usage.

A lossy dictionary is much like a normal dictionary with the exception that we allow it to store a slightly different set S' than the intended set S. The error we allow can be two-sided. That is, we allow the lossy dictionary to include elements not originally in S and we also allow it to report some elements not being in S when in fact they are. This relaxation allows for fast and very space efficient dictionaries. The formalization of which elements to keep from the original set S is done by associating weights to each element in S. The task of the lossy dictionary is then to maximize the sum of weights of the keys stored under a given space constraint. We present a lossy dictionary and show how it can be used with success in the setting of lossy compression with fast random access.

1.1 Outline of Dissertation

Chapter 2 introduces the underlying wavelet theory used throughout the dissertation. The theory will be presented by means of multiresolution analysis, since this is a good framework for describing not only wavelets and their properties but also how the fast wavelet transform is computed. We also explain why wavelets are well suited for data compression.

Chapter 3 briefly introduces the dictionary problem and serves as an introduction to Chapter 4 and Chapter 5.

Chapter 4 presents a new and efficient dictionary with worst case constant lookup time and amortized expected constant time for updates. A nice property of our dictionary is that it is simple to implement. The space usage is similar to that of binary search trees, i.e., three words per key on average. The practicality of the scheme is backed by extensive experiments and comparisons with several well known methods, showing it to be quite competitive also in the average case. We call the scheme Cuckoo hashing motivated by the way elements are inserted into the data structure.

Chapter 5 introduces the concept of *lossy dictionaries* and presents an efficient lossy data structure. Experimentally, we find that the lossy dictionary data structure has good behavior with respect to keeping the most important keys. By assuming stronger hash functions, than those used in the experiments, we obtain theoretical results which partly explain the experimental results. Furthermore, we show that our data structure is nearly optimal with respect to space usage. We conclude the chapter by a small example to motivate the use of lossy dictionaries in lossy compression with fast random access.

Chapter 6 serves as an introduction to Chapter 7 and Chapter 8. It motivates the need for volumetric compression with fast random access and considers good design criteria for such methods. Also, an extensive survey of related work on volumetric compression is given.

Chapter 7 presents the first of two methods for volumetric compression with fast random access presented in this dissertation. The method is based on the observation that one of the three dimensions of volumetric data can be considered similar to time. For this reason the volume is considered as a time sequence of slices. To remove redundancies in this "temporal" direction, a simple motion estimation technique is employed. Redundancies are then further reduced by decomposing the motion compensated slices into a two-dimensional wavelet basis. In order to encode the wavelet coefficients of largest magnitude and their significance map, we use a three-step block indexing scheme to locate the nonzero coefficients. The method improves the compression ratio of existing schemes by about 50% while offering acceptable access times when accessing voxels. Since the method only needs access to a few slices at a time it can be used in an online setting where data is compressed as it is downloaded or otherwise transferred to the system.

Chapter 8 describes the second volumetric compression method which follows a completely different approach to the storage of wavelet transformed data. While it is common to store the coefficients of largest magnitude, and let all other coefficients be zero, we allow a slightly different set of coefficients to be stored. This brings into play the lossy hashing technique of Chapter 5 that allows space efficient storage and very efficient retrieval of wavelet coefficients. The coefficients are computed by a three-dimensional wavelet transform. Our approach is applied to compression of volumetric data sets. For the Visible Man CT volume [86] we obtain up to 80% improvement in compression ratio over previously suggested schemes, including the first method based on temporal prediction presented in Chapter 7. Further, the time for accessing a random voxel is quite competitive with existing schemes.

Chapter 9 provides a comparison between the two compression methods. In particular, we look at the settings in which one method is preferable over the other in relation to a set of design criteria given in Chapter 6.

Chapter 10 summarizes the conclusions of the dissertation and provides pointers and directions for future research.

1.2 List of Contributions

This section relates the scientific papers, that I have written during my Ph.D studies, to the chapters of this dissertation.

1.2.1 Chapter 4

Chapter 4 is based on the technical report *Cuckoo Hashing.* The paper has been published as a technical report [63] and as a conference paper [64].

[63] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Technical Report RS-01-32*, BRICS, August 2001.

[64] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01),* volume 2161 of *Lecture notes in Computer Science,* pages 121-133, Berlin, 2001. Springer-Verlag.

The technical report extends the conference paper by including a new section where experimental results are explained in terms of a simple model. Also, more detailed text and figures have been added.

1.2.2 Chapter 5

Chapter 5 is based on the technical report *Lossy Dictionaries.* The paper has been published as a technical report [65] and as conference paper [66].

[65] Rasmus Pagh and Flemming Friche Rodler. Lossy Dictionaries. *Technical Report RS-01-33*, BRICS, August 2001.

[66] Rasmus Pagh and Flemming Friche Rodler. Lossy Dictionaries. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01),* volume 2161 of *Lecture notes in Computer Science,* pages 300-311, Berlin, 2001. Springer-Verlag.

The technical report extends the conference paper by simplifying the proof of the construction algorithm and by adding more detailed text and figures. Also, a section on applications of the presented data structure has been added.

1.2.3 Chapter 7

Chapter 7 is based on the Technical Report *Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data.* The paper has been published as a technical report [70] and as a conference paper [71].

[70] Flemming Friche Rodler. Wavelet Based 3D Compression for Very Large Volume Data Supporting Fast Random Access. *Technical Report RS-99-34*, BRICS, October 1999.

[71] Flemming Friche Rodler. Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data. In *Proceedings of the Seventh Pacific Conference on Computer Graphics and Applications*, pages 108-117, Seoul, Korea, 1999.

The technical report extends the conference paper by adding a section which gives a more thorough introduction to wavelet theory.

1.2.4 Chapter 8

Chapter 8 is based on the technical report *Fast Random Access to Wavelet Compressed Volumetric Data Using Hashing.* The paper has been published as a technical report [73] and has been accepted for publication as a journal paper [72].

[73] Flemming Friche Rodler and Rasmus Pagh. Fast Random Access to Wavelet Compressed Volumetric Data Using Hashing. *Technical Report RS-01-34*, BRICS, August 2001.

[72] Flemming Friche Rodler and Rasmus Pagh. Fast Random Access to Wavelet Compressed Volumetric Data Using Hashing. *To Appear in ACM Transactions on Graphics.*

There are only minor changes between the two versions.

Chapter 2

Wavelet Theory

In this chapter we present the underlying mathematical foundations of the wavelet theory and hierarchical decomposition used throughout the dissertation. We start by introducing the multiresolution analysis (MRA) concept which can be used as a method to describe hierarchical bases. This will lead to the definition of wavelets and the fast wavelet transform. Furthermore, we discuss the properties of wavelets and explain why they work so well in compression.

2.1 Multiresolution Analysis

The theory of multiresolution analysis was initiated by Stéphane Mallat [52] and Yves Meyer[1].

Definition 2.1 (Multiresolution Analysis) *A multiresolution analysis (MRA) is a sequence $(V_j)_{j\in\mathbf{Z}}$ of closed subspaces of $L^2(\mathbf{R})$ satisfying the following five properties:*

1. $\forall j \in \mathbf{Z} : V_j \subset V_{j+1}$
2. $\bigcap_{j\in\mathbf{Z}} V_j = \{0\}$
3. $\overline{\bigcup_{j\in\mathbf{Z}} V_j} = L^2(\mathbf{R})$
4. $f(x) \in V_j \Leftrightarrow f(2x) \in V_{j+1}$
5. $\exists \phi \in V_0$ *such that* $\{\phi(x-k)\}_k$ *constitutes an orthonormal basis for* V_0.

The first and third property of the definition gives us the approximation feature of the MRA. Furthermore the third property states that any function in $L^2(\mathbf{R})$ can be approximated arbitrarily well by its projection onto V_j for a suitably large j, i.e., if P_{V_j} denotes the projection operator onto V_j then:

$$\lim_{j\to\infty} \|f - P_{V_j} f\| = 0 \ . \tag{2.1}$$

[1] Yves Meyer, 1986 – *Ondelettes, fonctions splines et analyzes graduées.* Lectures given at the University of Torino, Italy.

The first property is a causality property which guarantees that an approximation at a resolution j contains all the information necessary to compute an approximation at a coarser resolution $j-1$. The second property proves that as j tends towards $-\infty$ the projection of f onto V_j contains arbitrarily little energy, or stated differently we lose all the details of f when the resolution j goes towards $-\infty$:

$$\lim_{j\to-\infty} \|P_{V_j} f\| = 0 \ . \tag{2.2}$$

The aspect of a MRA comes with the fourth property which tells us that the resolution increases with j and that all the spaces are scaled versions of each other[2].

Define $\phi_{j,k}$ as

$$\phi_{j,k}(x) = 2^{\frac{j}{2}}\phi(2^j x - k) \ . \tag{2.3}$$

As a direct consequence of properties *4* and *5* we have that $\{\phi_{j,k}(x) = 2^{\frac{j}{2}}\phi(2^j x - k)\}_{k\in\mathbf{Z}}$ constitutes an orthonormal basis for V_j. Since the orthonormal projection of f onto V_j is obtained by the following expansion

$$P_{V_j} f = \sum_{k=-\infty}^{\infty} \langle f, \phi_{j,k}\rangle \phi_{j,k} \ , \tag{2.4}$$

we have that

$$a_j[k] = \langle f, \phi_{j,k}\rangle \tag{2.5}$$

provides a discrete approximation of f at scale j, where $\langle\cdot,\cdot\rangle$ denotes the inner product and is defined for $f, g \in V_j$ as

$$\langle f, g\rangle = \int_{\mathbf{R}} f(x) g^*(x) dx \ , \tag{2.6}$$

where $g^*(x)$ is the complex conjugate of $g(x)$. The functions $\phi_{j,k}$ are called scaling functions. The requirement for the $\phi_{j,k}$'s to generate orthonormal bases for the V_j's can be relaxed and it can be shown that we only need to require that they generate a Reisz basis for the MRA spaces, see e.g. [17].

Example 2.1 *As an example of a multiresolution analysis we introduce the Haar MRA [34] where the approximations are composed as piecewise constant functions. The MRA spaces are defined as*

$$V_j = \{f \in L^2(\mathbf{R}) : \forall k \in \mathbf{Z} \ , \ f|_{[k2^{-j},(k+1)2^{-j}]} = \mathit{Constant}\} \tag{2.7}$$

and the scaling functions are generated from the box function $1_{[0,1]}$*. The properties in the definition of a MRA are easily verified.*

[2]Henceforth resolution and scale will be used interchangeably

2.2 Wavelets

The basic tenet of multiresolution analysis is that whenever there exists a sequence of closed subspaces satisfying Definition 2.1 then there exists an orthonormal wavelet basis $\{\psi_{j,k}\}_{j,k}$ given by the following definition.

Definition 2.2 (Wavelet) *A wavelet is a function $\psi \in L^2(\mathbf{R})$ chosen such that the dyadic family*

$$\psi_{j,k}(x) = 2^{\frac{j}{2}}\psi(2^j x - k), \quad k, j \in \mathbf{Z} \tag{2.8}$$

of functions constitutes an orthonormal basis for $L^2(\mathbf{R})$. ψ is often referred to as the mother wavelet since it generates the whole family.

This family of functions is called the dyadic wavelet family since it is generated by dyadic dilates and translates of a single function ψ. Given the above definition we can write the *dyadic wavelet transform* as

$$d_{j,k} = \langle f, \psi_{j,k} \rangle = \int_{\mathbf{R}} f(x)\psi^*_{j,k}(x)dx, \quad k, j \in \mathbf{Z} \ , \tag{2.9}$$

where the $d_{j,k}$'s denote the *wavelet coefficients*. The *reconstruction formula* becomes

$$f(x) = \sum_j \sum_k d_{j,k}\psi_{j,k}(x) \ . \tag{2.10}$$

The connection to multiresolution analysis arises because the function f at scale j, i.e. $P_{V_j}f$, can be written as

$$P_{V_j}f = P_{V_{j-1}}f + \sum_k \langle f, \psi_{j-1,k}\rangle \psi_{j-1,k} \ , \tag{2.11}$$

where ψ is a wavelet and the summation describes the detail necessary to go from the coarser space $P_{V_{j-1}}$ to the finer space P_{V_j}. In the following this will be formalized.

For every $j \in \mathbf{Z}$ we define the *complement space* W_j as the orthogonal complement of V_j in V_{j+1}, i.e.: $W_j = V_{j+1} \cap V_j^{\perp}$. We can then write V_{j+1} as

$$V_{j+1} = V_j \oplus W_j \ , \tag{2.12}$$

where $\oplus$ is the orthogonal direct sum. By definition all of the spaces W_j satisfy

$$W_j \perp W_{j'} \quad \text{for } j \neq j' \ , \tag{2.13}$$

since for $j > j' : W_{j'} \subset V_j$ and $V_j \perp W_j$. By the definition of the complement spaces and by iteration it follows that for $j > J$:

$$V_j = V_J \oplus \bigoplus_{l=J}^{j-1} W_l \ . \tag{2.14}$$

Because $V_J \to \{0\}$ for $J \to -\infty$ in $L^2(\mathbf{R})$ this implies

$$V_j = \bigoplus_{l=-\infty}^{j-1} W_l \ . \tag{2.15}$$

Again by noticing that $V_j \to L^2(\mathbf{R})$ when $j \to \infty$ we get

$$L^2(\mathbf{R}) = \bigoplus_{l=-\infty}^{\infty} W_j \ , \tag{2.16}$$

which by virtue of (2.13) implies that we have decomposed $L^2(\mathbf{R})$ into a set of mutually orthogonal subspaces. Since it is easy to prove that the complement spaces W_j inherit the scaling property

$$f(x) \in W_j \Leftrightarrow f(2x) \in W_{j+1} \ , \tag{2.17}$$

all we need to do in order to construct a wavelet basis is to find a function $\psi \in L^2(\mathbf{R})$ such that $\{\psi(x-k)\}_{k\in\mathbf{Z}}$ constitutes a basis for W_0. Then for a fixed $j \in \mathbf{Z}$ we have that $\{\psi_{j,k}\}_{k\in\mathbf{Z}}$ is an orthonormal basis for W_j. Finally, we have by means of (2.16) that $\{\psi_{j,k}\}_{(j,k)\in\mathbf{Z}^2}$ constitutes an orthonormal basis for $L^2(\mathbf{R})$. Finding the functions ϕ and ψ is in general nontrivial and requires some mathematical work.

Example 2.2 *In the previous example we introduced the Haar MRA. The corresponding Haar wavelet is given by*

$$\psi(x) = \begin{cases} 1 & \text{for } 0 \le x < \frac{1}{2} \\ -1 & \text{for } \frac{1}{2} \le x < 1 \\ 0 & \text{otherwise} \ . \end{cases} \tag{2.18}$$

2.2.1 Properties

In compression and many other applications we use the ability of wavelets to efficiently approximate many different signals with few nonzero wavelet coefficients. Much of this ability can be attributed to the properties of the wavelets given in the following list.

- **Support:** The support of a wavelet is given by the following closure:

$$\text{supp}(\psi) = \overline{\{x \in \mathbf{R} : \psi(x) \neq 0\}} \tag{2.19}$$

  If there exist $a, b \in \mathbf{R}$ such that $\text{supp}(\psi) \subset [a, b]$ then ψ is said to be compactly supported. There are two main reasons why a wavelet with a small support is preferable in compression. Firstly, if the function $f(x)$ that we want to compress has a singularity at x' within the support of $\psi_{j,k}$ then $\langle f, \psi_{j,k} \rangle$ might have large magnitude. Now, if ψ has compact support with width S then at each scale j the support of $\psi_{j,k}$ will include x', S times. This makes it desirable to have S as small as possible. Secondly, as we shall see in Section 2.3.1 a small support for the wavelet implies faster decomposition and reconstruction algorithms. This is essential since we are aiming for fast reconstruction times.

- **Vanishing moments:** A function ψ is said to have n vanishing moments if the following holds true:

$$\int_{\mathbf{R}} x^k \psi(x) dx = 0 \quad \text{for } k = 0, .., n-1 \ . \tag{2.20}$$

  Many signals can be approximated well piecewisely by low order polynomials of degree p. So if the analyzing wavelet has $n > p$ vanishing moments this results in wavelet coefficients close to zero. Unfortunately, vanishing moments come at the expense of wider support, see e.g. [53, pp. 241-245]. In fact, for orthogonal wavelets with n vanishing moments the width of the support will be at least $2n - 1$. The Daubechies wavelets are optimal in this respect. If we expect our signals to be highly regular with only a few isolated singularities then wavelets with many vanishing moments are preferable. On the other hand wavelets with small support might be the better choice if our signals are expected to contain many singularities.

- **Smoothness:** The smoothness or regularity of a wavelet is usually measured in terms of the number of continuous derivatives it has. Smooth wavelets are important in lossy compression of images. In lossy wavelet based compression, errors are mostly introduced during quantization of the wavelet coefficients. We see from the reconstruction formula

$$f(x) = \sum_{j,k} d_{j,k} \psi_{j,k}(x) \tag{2.21}$$

  that if the wavelet coefficient $d_{j,k}$ is changed by ϵ the error $\epsilon \psi_{j,k}(x)$ will be added to $f(x)$. If ψ is smooth the introduced artifacts will also be smooth and smooth artifacts are perceptually less annoying than irregular ones. Smoothness comes at the expense of wider support [53, pp. 241-245].

- **Symmetry:** Wavelets that are symmetric or antisymmetric are important for several reasons. The wavelet transform is a transform over signals in $L^2(\mathbf{R})$. In order to transform a signal with finite support (i.e., in $L^2([0,1])$) it must be extended to $L^2(\mathbf{R})$. To this end several ways of performing the extension exist, many resulting in boundary effects in the wavelet domain (i.e. high coefficients) near 0 and 1. This is undesirable for many applications especially compression. Symmetric or antisymmetric wavelets allow for (anti)symmetric extensions at the boundaries which partly solves the problem.

  Symmetric and antisymmetric wavelets are synthesized with filters having linear phase. Wavelets and their synthesizing filters are introduced in Section 2.3. The linear phase property of the filters are important for some applications.

- **Orthogonality:** Daubeschies [17] proved that except for the Haar wavelet there exist no (anti)symmetric real orthogonal compactly supported wavelet bases. By giving up orthogonality and allowing for biorthogonal

wavelets[3] it is possible to construct compactly supported (anti)symmetric wavelet bases [17, 53]. For a complete description of biorthogonal wavelets we refer to [53].

- **Localization:** Wavelets with small support or rapid decay towards zero are said to be well localized in the spatial domain. In contrast is localization in the spectral or frequency domain which is important for some applications. The Heisenberg uncertainty principle[4] [35] gives a bound on how localized a function can be simultaneously in time and frequency

$$\sigma_t^2 \sigma_\omega^2 \geq \frac{1}{4} \ , \tag{2.22}$$

where

$$\sigma_t^2 = \int_{\mathbf{R}} (t - t_c) \, |g(t)|^2 \, dt \qquad \sigma_\omega^2 = \int_{\mathbf{R}} (\omega - \omega_c) \, |\hat{g}(\omega)|^2 \, d\omega \tag{2.23}$$

denotes the standard deviation from the centers of gravity given by

$$t_c = \int_{\mathbf{R}} t \, |g(t)|^2 \, dt \qquad \omega_c = \int_{\mathbf{R}} \omega \, |\hat{g}(\omega)|^2 \, d\omega \ . \tag{2.24}$$

Thus good spatial localization comes at the expense of poorer localization in frequency. For a wavelet basis, which consists of scaled versions of the mother wavelet, this means that high frequencies (narrow wavelets) are analyzed with good positional accuracy whereas low frequencies (wide wavelets) are analyzed more accurately in frequency. This adaption to frequency is contrary to other time-frequency analysis methods such as the windowed Fourier transform and it is an important property in, e.g., sound processing and analysis.

2.3 The Fast Wavelet Transform and its Inverse

The fast wavelet transform (FWT) decomposes the signal f in the wavelet basis by recursively convolving the signal with filters H and G. Assume that we have a function $f_J \in V_J$ given by

$$f_J(x) = \sum_k a_{J,k} \phi_{J,k}(x) \in V_J \tag{2.25}$$

with

$$a_{J,k} = \langle f_J, \phi_{J,k} \rangle = \int_{\mathbf{R}} f_J(x) \phi_{J,k}^*(x) dx \ . \tag{2.26}$$

[3]Biorthogonal wavelets are given by two dual bases $\{\psi_{j,k}\}_{(j,k)\in\mathbf{Z}^2}$ and $\{\tilde{\psi}_{j,k}\}_{(j,k)\in\mathbf{Z}^2}$. For any $(j, j', k, k') \in \mathbf{Z}^4$ we have $\langle \psi_{j,k}, \psi_{j',k'} \rangle = \delta_{j,j'} \delta_{k,k'}$. The dual bases span the spaces W_j and $\tilde{W}_j$ respectively. Similarly, there are two scaling functions $\phi_{j,k}$ and $\tilde{\phi}_{j,k}$ spanning the two multiresolution approximations V_j and $\tilde{V}_j$. The biorthogonality implies that W_j is not orthogonal to V_j but to $\tilde{V}_j$. Likewise, $\tilde{W}_j$ is not orthogonal to $\tilde{V}_j$ but to V_j. A function $f \in L^2(\mathbf{R})$ has two possible decompositions $f = \sum_{j,k} \langle f, \psi_{j,k} \rangle \tilde{\psi}_{j,k} = \sum_{j,k} \langle f, \tilde{\psi}_{j,k} \rangle \psi_{j,k}$.

[4]The Heisenberg uncertainty principle originates from quantum mechanics, but is in fact a general property of functions.

The fast wavelet transform then computes the wavelet coefficients of the discrete signal

$$a_J[k] = a_{J,k}, \tag{2.27}$$

where each sample of $a_J[k]$ according to (2.26) is a weighted average of f around a neighborhood of f with averaging kernel $\phi_{J,k}$. We will now look at the FWT of a signal f_J.

We have $f_J \in V_J = V_{J-1} \oplus W_{J-1}$ and thus there exist sequences $a_{J-1,k}$ and $d_{J-1,k}$ such that

$$f_J(x) = \sum_k a_{J-1,k}\phi_{J-1,k}(x) + \sum_k d_{J-1,k}\psi_{J-1,k}(x) \ , \tag{2.28}$$

where $d_{J-1,k}$ are the wavelet coefficients at scale $J-1$. Because of property 1 of Definition 2.1 of a MRA the sequences $a_{J-1,k}$ and $d_{J-1,k}$ can be computed as

$$a_{J-1,k} = \langle f_J, \phi_{J-1,k}\rangle = \int_{\mathbf{R}} f(x)\phi^*_{J-1,k}(x)dx \tag{2.29}$$

$$d_{J-1,k} = \langle f_J, \psi_{J-1,k}\rangle = \int_{\mathbf{R}} f(x)\psi^*_{J-1,k}(x)dx \ . \tag{2.30}$$

By repeating on the $a_{J-1,k}$'s the complete set of discrete wavelet coefficients $d_{J-1,k}, d_{J-2,k}, \ldots, d_{0,k}$ can be computed. So the FWT successively decomposes the approximation $P_{V_j}f$ into a coarser approximation $P_{V_{j-1}}$ plus the wavelet coefficients $P_{W_{j-1}}$.

Unfortunately, computing the wavelet coefficients by means of (2.29) and (2.30) would not be very efficient. We find hope in the following theorem, which is due to Mallat [52].

Theorem 2.1 (The fast orthogonal wavelet transform) *For an orthogonal wavelet basis there exist filters $H = \{h_n\}_n$ and $G = \{g_n\}_n$ such that*

$$a_{j-1,k} = \sum_n h_{n-2k}a_{j,n} \tag{2.31}$$

$$d_{j-1,k} = \sum_n g_{n-2k}a_{j,n} \ . \tag{2.32}$$

Similarly the inverse computation is given by

$$a_{j,k} = \sum_n h_{k-2n}a_{j-1,n} + \sum_n g_{k-2n}d_{j-1,n} \ . \tag{2.33}$$

Theorem 2.1 connects wavelets with *filter banks*. The convolution in (2.31) and (2.32) can be interpreted as filtering the signal a_j with filters H and G, respectively as illustrated in Figure 2.1. Note that because of the $2k$ in the sum a dyadic downsampling takes place. This is important since the downsampling ensures that the data is not doubled. The inverse transform in (2.33) first upsamples by inserting zeros and then interpolates by filtering its input signals a_{j-1} and d_{j-1} to obtain the reconstructed signal a_j.

Example 2.3 *The Haar wavelet corresponds to two-tap*[5] *filters given by $H = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ and $G = [\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]$.*

[5]Tap is the term used to denote the filter length of a Finite Impulse Response (FIR) filter.

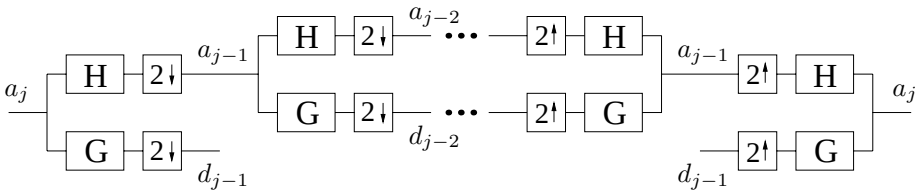


Figure 2.1: A 3-channel filter bank for computing a two level wavelet decomposition and its inverse.

2.3.1 Complexity

A finite signal $a_J[k]$ cannot be decimated indefinitely. The iterative process must at least terminate when there is only one approximation coefficient a_0 left. Normally, for compression purposes, only a few iterations, say 2 to 5, are applied. For I iterations the wavelet decomposition is composed of the wavelet coefficients at scale $J - I \leq j < J$ plus the remaining approximation coefficients at scale $J - I$. The time complexity of the algorithm is easy to analyze. If we start with N samples and two filters H and G having at most K filter coefficients then

$$KN + \frac{KN}{2} + \frac{KN}{4} + \cdots + 1 \leq 2KN \tag{2.34}$$

is an upper bound on the number of additions and multiplications that will be performed. Table 2.1 shows the filter length, the support width, and the number of vanishing moments of some well known wavelets.

Wavelet family	Haar	Daubechies	Coiflets	Symlets
Order	1	M	M	M
Support width	1	$2M-1$	$6M-1$	$2M-1$
Filter length	2	$2M$	$6M$	$2M$
Vanishing moments	1	M	$2M$	M

Table 2.1: The filter length, the support width, and the number of vanishing moments of some well known wavelets of order M.

Instead of using the number of multiplications and additions, we can alternatively use the number of wavelet coefficients needed for reconstruction as a measure of complexity. It is easy to see that in order to compute one coefficient of $a_j[k]$, K coefficients are needed, i.e., $K/2$ coefficients from both $a_{j-1}[k]$ and $d_{j-1}[k]$, respectively.

Because the wavelet coefficients must be retrieved from memory in order to compute the inverse transform and since memory lookups can be expensive compared to the traditional arithmetic operations this is an important complexity measure for our applications. In fact, we consider this measure to be more exact and revealing since the number of multiplications and additions often can be optimized by reusing previous computations (especially in higher dimensions) or by using the lifting scheme [80].

2.3.2 Initialization

As mentioned in the beginning of Section 2.3 the FWT computes the wavelet coefficients of a discrete signal $a_J[k]$ given by

$$a_J[k] = \langle f, \phi_{J,k} \rangle, \tag{2.35}$$

meaning that $a_J[k]$ is a local average of $f \in V_J$ around k but not precisely equal to $f(k)$. So we need to find $a_J[k]$ from f in order to start the algorithm. Often this is omitted and the algorithm is applied directly on a sampled version $f[n]$ of $f(x)$. Frequently, $f[n]$ is given as samples recorded by a device of finite resolution such as a CCD camera or MR scanner that averages and samples an analog signal, so without information about the averaging kernel of the sampling device, decomposing the samples $f[n]$ directly is justified.

2.4 Multidimensional Bases

The above constructions have been concerned with the task of decomposing functions in $L^2(\mathbf{R})$. In order to analyze multidimensional functions these techniques must be extended to $L^2(\mathbf{R}^d)$. Defining a MRA $\{V_j^d\}$ of $L^2(\mathbf{R}^d)$ formally is a straightforward extension of Definition 2.1. In this section we only consider separable MRA's.

2.4.1 Nonstandard Basis

We start by an extension to two dimensions. If $\{V_j\}_j$ is a MRA of $L^2(\mathbf{R})$ then the tensor product spaces defined as

$$\{V_j^2 = V_j \otimes V_j\}_{j \in \mathbf{Z}} \tag{2.36}$$

constitute a *separable two-dimensional MRA* for $L^2(\mathbf{R}^2)$ and in the case where $\{\phi_{j,k}\}_{k \in \mathbf{Z}}$ is an orthonormal basis for V_j, the product functions

$$\{\phi_{j,k,l}^2(x,y) = \phi_{j,k}(x)\phi_{j,l}(y)\}_{(k,l) \in \mathbf{Z}^2} \tag{2.37}$$

form an orthonormal basis for V_j^2. As for the one-dimensional case this allows for the definition of the complement spaces W_j^2. We have

$$\begin{aligned} V_{j+1}^2 &= V_{j+1} \otimes V_{j+1} \\ &= (V_j \oplus W_j) \otimes (V_j \oplus W_j) \\ &= V_j^2 \oplus [(V_j \otimes W_j) \oplus (W_j \otimes V_j) \oplus (W_j \otimes W_j)] \ . \end{aligned} \tag{2.38}$$

This shows that

$$\{\psi_{j,k,l}^{2,\lambda} : \lambda \in \{v,h,d\}\}_{(k,l) \in \mathbf{Z}^2} \ , \tag{2.39}$$

with the wavelets $\psi_{j,k,l}^{2,\lambda}$ given by

$$\begin{aligned} \psi_{j,k,l}^{2,v}(x,y) &= \phi_{j,k}(x)\psi_{j,l}(y), \\ \psi_{j,k,l}^{2,h}(x,y) &= \psi_{j,k}(x)\phi_{j,l}(y), \\ \psi_{j,k,l}^{2,d}(x,y) &= \psi_{j,k}(x)\psi_{j,l}(y) \end{aligned} \tag{2.40}$$

is an orthonormal basis for W_j^2 and therefore $\{\psi_{j,k,l}^{2,\lambda}\}_{(j,k,l)\in\mathbf{Z}^3}$ is an orthonormal basis for $L^2(\mathbf{R}^2)$.

The interpretation of the wavelets in terms of filters is carried over from the one-dimensional case and we obtain separable two-dimensional filters

$$\begin{aligned} h[x,y] &= h[x]h[y] \quad , \quad g^v[x,y] = h[x]g[y] \\ g^h[x,y] &= g[x]h[y] \quad , \quad g^d[x,y] = g[x]g[y] \ . \end{aligned} \tag{2.41}$$

Filtering with these corresponds to filtering first along the rows of the discrete signal and then along the columns. After filtering, downsampling in each direction is performed. This is illustrated in Figure 2.2. Note that for clarity subsampling is illustrated as the last step in the figure but for efficiency it takes place during filtering.

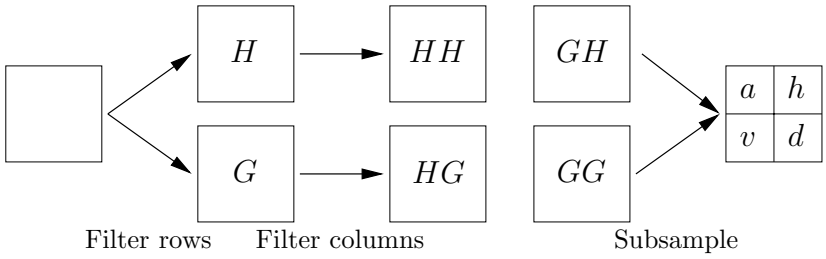


Figure 2.2: One decomposition level of a two-dimensional wavelet transform. The a, h, v, and d in the result indicate which filter was used to compute the subband.

Example 2.4 *Using the filters of the previous example we see that the two-dimensional Haar decomposition is given by*

$$\begin{aligned} a_{ll} &= ((c_1+c_2)/\sqrt{2} + (c_3+c_4)/\sqrt{2})/\sqrt{2} = (c_1+c_2+c_3+c_4)/2 \\ d_{lh} &= ((c_1+c_2)/\sqrt{2} - (c_3+c_4)/\sqrt{2})/\sqrt{2} = (c_1+c_2-c_3-c_4)/2 \\ d_{hl} &= ((c_1-c_2)/\sqrt{2} + (c_3-c_4)/\sqrt{2})/\sqrt{2} = (c_1-c_2+c_3-c_4)/2 \\ d_{hh} &= ((c_1-c_2)/\sqrt{2} - (c_3-c_4)/\sqrt{2})/\sqrt{2} = (c_1-c_2-c_3+c_4)/2 \ , \end{aligned} \tag{2.42}$$

where $c_1, \ldots, c_4$ on the right-hand side of the equations are data values in a 2×2 sub-block. a_{ll} is the average coefficient and d_{lh}, d_{hl}, and d_{hh} are the detail coefficients corresponding to filtering with g^h, g^v and g^d respectively. We use the subscripts l and h to denote whether we used the lowpass or the highpass filter. A subscript lh indicates that the lowpass filter was used horizontally and the highpass filter was used vertically. Reconstruction is given by

$$\begin{aligned} c_1 &= (a_{ll} + d_{lh} + d_{hl} + d_{hh})/2 \\ c_2 &= (a_{ll} + d_{lh} - d_{hl} - d_{hh})/2 \\ c_3 &= (a_{ll} - d_{lh} + d_{hl} - d_{hh})/2 \\ c_4 &= (a_{ll} - d_{lh} - d_{hl} + d_{hh})/2 \ . \end{aligned} \tag{2.43}$$

Note that if we want fast reconstruction we can divide by 4 on the right-hand side of (2.42) instead of dividing by 2. That way the 2 in the reconstruction equations becomes a 1 for easier computation.

2.4.2 Standard Basis

The above construction of multidimensional bases is often referred to as the nonstandard basis. Another construction is the standard basis where (2.37) and (2.40) are replaced by

$$\begin{aligned}
\phi_{j_1,j_2,k,l}(x,y) &= \phi_{j_1,k}(x)\phi_{j_2,l}(y), \\
\psi^{2,v}_{j_1,j_2,k,l}(x,y) &= \phi_{j_1,k}(x)\psi_{j_2,l}(y), \\
\psi^{2,h}_{j_1,j_2,k,l}(x,y) &= \psi_{j_1,k}(x)\phi_{j_2,l}(y), \\
\psi^{2,d}_{j_1,j_2,k,l}(x,y) &= \psi_{j_1,k}(x)\psi_{j_2,l}(y) \ .
\end{aligned} \tag{2.44}$$

We note that in the standard basis the wavelets can have varying resolutions in different spatial directions. That is, the standard basis does not allow for strictly different resolution levels. For this reason, the standard basis construction is not often used in compression since direct control of the resolution is lost.

2.4.3 Higher Dimensions

As mentioned the extension to higher dimensions are analogous to two dimensions. For completeness, we show, in Figure 2.3, how the different wavelet coefficients are arranged into subbands for a two-level three-dimensional transform. In three dimensions we obtain seven detail subbands on each level and one average subband. Using the Haar wavelet the decomposition equations are given by

$$\begin{aligned}
a &= (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8)/2\sqrt{2} \\
d^1 &= (c_1 + c_2 + c_3 + c_4 - c_5 - c_6 - c_7 - c_8)/2\sqrt{2} \\
d^2 &= (c_1 + c_2 - c_3 - c_4 + c_5 + c_6 - c_7 - c_8)/2\sqrt{2} \\
d^3 &= (c_1 + c_2 - c_3 - c_4 - c_5 - c_6 + c_7 + c_8)/2\sqrt{2} \\
d^4 &= (c_1 - c_2 + c_3 - c_4 + c_5 - c_6 + c_7 - c_8)/2\sqrt{2} \\
d^5 &= (c_1 - c_2 + c_3 - c_4 - c_5 + c_6 - c_7 + c_8)/2\sqrt{2} \\
d^6 &= (c_1 - c_2 - c_3 + c_4 + c_5 - c_6 - c_7 + c_8)/2\sqrt{2} \\
d^7 &= (c_1 - c_2 - c_3 + c_4 - c_5 + c_6 + c_7 - c_8)/2\sqrt{2} \ ,
\end{aligned} \tag{2.45}$$

where $c_1, \ldots, c_8$ are the eight coefficients in a $2 \times 2 \times 2$ subregion of the three-dimensional signal. The numbering $0, \ldots, 7$ of the detail coefficients corresponds to the different combinations of applying the lowpass and highpass filters in each direction.

2.4.4 Complexity of Multidimensional Transforms

In one dimension it takes, according to Section 2.3.1, K wavelet coefficients to reconstruct one coefficient of the next resolution level of the inverse transform. For a d-dimensional transform it follows that K^d coefficients are needed. For the

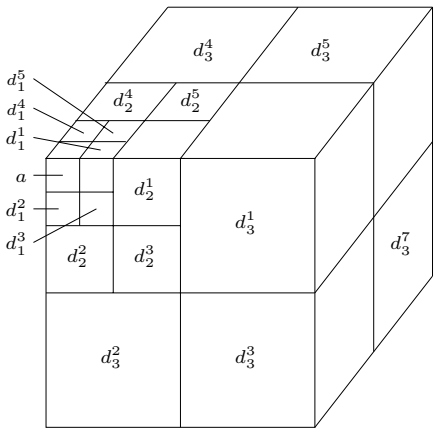


Figure 2.3: Subbands of a three-level three-dimensional wavelet transform. The subbands are numbered d_j^i, where j denotes the resolution level while i denotes the subband number within the level.

Haar wavelet this means that 4 coefficients must be retrieved in two dimensions and 8 in three dimensions. Other orthogonal wavelet filters are at least four taps long, so at least 16 and 64 coefficients must be accessed to reconstruct one coefficient at the next resolution level for two and three dimensions respectively. As seen in (2.43) of Example 2.4 the extracted wavelet coefficients can be reused to reconstruct neighboring data values, if they all are needed. However, because our aim is to achieve fast random access this is not the case. For this reason we use the Haar wavelet in the two compression methods described in Chapter 6.

2.5 Thresholding

In Section 2.2.1 we pointed out that the wavelet representation for many functions is able to concentrate most of the energy in a small number of wavelet coefficients with the rest of the coefficients being zero or close to zero. By setting all the small valued coefficients to zero a very sparse representation can be obtained and exploited for compression purposes. For an orthonormal wavelet transform this thresholding of the coefficients corresponds to a global optimal approximation in terms of the Mean Square Error (MSE) given by

$$\begin{aligned} MSE &= \frac{1}{N}\sum_k (x_k - \hat{x}_k)^2 = \frac{1}{N}(x-\hat{x})^T \cdot (x-\hat{x}) \\ &= \frac{1}{N}(W^T W(x-\hat{x}))^T \cdot (W^T W(x-\hat{x})) \\ &= \frac{1}{N}((y-\hat{y})^T \cdot WW^T \cdot ((y-\hat{y}) \\ &= \frac{1}{N}\sum_k (y_k - \hat{y}_k)^2 \ , \end{aligned} \tag{2.46}$$

where W is an orthonormal transform and the x_k and $\hat{x}_k$ are the original and reconstructed signals and the y_k and $\hat{y}_k$ are the transform coefficients before

and after thresholding. As a consequence one can explicitly give the exact error that occurs due to thresholding.

The wavelet coefficients that remain nonzero after thresholding are called the *significant coefficients* and their positions in the transformed signal is called the *significance map*. In compression both the values of the significant coefficients and the significance map must be coded.

2.6 Wavelets on the Interval

The above mentioned theory has described how to decompose signals of $L^2(\mathbf{R}^d)$. However, in data compression we are more often concerned with finite signals, i.e., signals on an interval. To decompose a signal defined on an interval $[a, b]$, wavelet bases for $L^2([a, b])$ must be constructed. In the following sections we briefly describe four approaches to the construction of such wavelets. It suffices to look at the case $[a, b] = [0, 1]$. For any interval $[a, b]$ a wavelet basis for $L([a, b])$ can be constructed from the wavelet basis for $L([0, 1])$ by a dilation of $b - a$ and a translation of a. Extensions to multi-dimensional wavelets on an interval can be performed as described in Section 2.4.

Note that wavelets on an interval are not so important in compression with fast random access. Especially, if a three-dimensional wavelet transform is used, since far too many wavelet coefficients must be retrieved in order to reconstruct a single data element if other wavelets than the Haar wavelet are used. Corresponding to a two-tap filter boundary problems are easily avoided with the Haar wavelet. However, in Chapter 7 and Chapter 8 it will become apparent that the compression methods we present can easily be converted into using higher order wavelets on the interval.

2.6.1 Zero Padding

In *zero padding* the signal f is extended from the interval $[0, 1]$ to the whole real line $\mathbf{R}$ as:

$$f^{pad}(x) = \begin{cases} f(x), & \text{for } x \in [0, 1] \\ 0, & \text{otherwise .} \end{cases} \tag{2.47}$$

Since f^{pad} is defined on $\mathbf{R}$, Theorem 2.1 can be applied to decompose $f^{pad}(x)$. The main advantage of this method is simplicity since the MRA can be used without modification at the border. However, there are two major disadvantages of the method. The first disadvantage is that even though f is smooth, discontinuities can occur at the border. From Section 2.2.1 we know that this might yield large coefficients near the boundary. The second issue is that too many coefficients are generated. Ideally, it should suffice to use $N - 1$ wavelet coefficients plus 1 average coefficient to represent a signal with N samples. For zero padding this is generally not the case.

2.6.2 Periodic Extension

Instead of zero padding we can extend the signal $f(x) \in L^2([0,1])$ by the periodic repetition

$$f^{per}(x) = \sum_{i=-\infty}^{\infty} f^{pad}(x+i) \ . \tag{2.48}$$

This is illustrated in Figure 2.4. Since this extension to $\mathbf{R}$ might create discontinuities near the boundary, large wavelet coefficients may be introduced. This was discussed in Section 2.2.1.

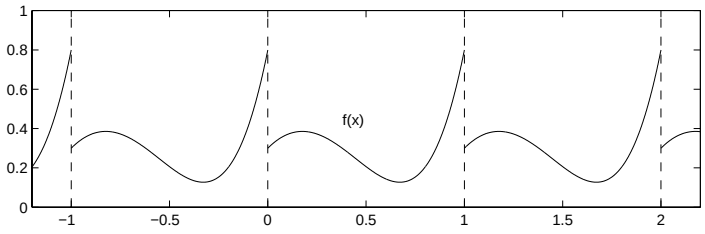


Figure 2.4: Periodic extension of $f(x) \in L^2([0,1])$.

As $f^{per} \in L^2(\mathbf{R})$ the wavelet coefficients can be computed as

$$d_{j,k} = \langle f^{per}, \psi_{j,k} \rangle = \int_{\mathbf{R}} f^{per}(x) \psi^*_{j,k}(x) dx \ . \tag{2.49}$$

The periodic wavelet transform can be computed efficiently by replacing the convolution of (2.31) and (2.32) by circular convolution.

Since it can be proven that

$$\int_{\mathbf{R}} f^{per}(x) \psi_{j,k}(x) dx = \int_0^1 f(x) \psi^{per}_{j,k}(x) dx \ , \tag{2.50}$$

where $\psi^{per}_{j,k}$ is a periodized version of $\psi_{j,k}$ given by

$$\psi^{per}_{j,k}(x) = 2^{j/2} \sum_{i=-\infty}^{\infty} \psi(2^j(x+i) - k) \ , \tag{2.51}$$

extending $f(x)$ by periodization corresponds to using wavelets modified to the interval. If $\text{supp}(\psi_{j,k}) \in [0,1]$ and $x \in [0,1]$ then $\psi^{per}_{j,k}(x) = \psi_{j,k}(x)$. The restriction to the interval $[0,1]$ thus means that only the boundary wavelets, whose support cross either $x = 0$ or $x = 1$, are modified.

2.6.3 Symmetric Extension

The problem with discontinuities at the boundary can partly be alleviated by folding the signal $f(x) \in L^2([0,1])$ to obtain a symmetric extension, illustrated in Figure 2.5, and given by

$$f^{sym}(x) = \sum_{i=-\infty}^{\infty} f^{pad}(x - 2i) + \sum_{i=-\infty}^{\infty} f^{pad}(2i - x) \ . \tag{2.52}$$

Similar to (2.50) we can verify that

$$\int_{\mathbf{R}} f^{sym}(x)\psi_{j,k}(x)dx = \int_0^1 f(x)\psi_{j,k}^{sym}(x)dx \tag{2.53}$$

with $\psi_{j,k}^{sym}(x)$ given by

$$\psi_{j,k}^{sym}(x) = \sum_{i=-\infty}^{\infty} \psi_{j,k}(x-2i) + \sum_{i=-\infty}^{\infty} \psi_{j,k}(2i-x) \tag{2.54}$$

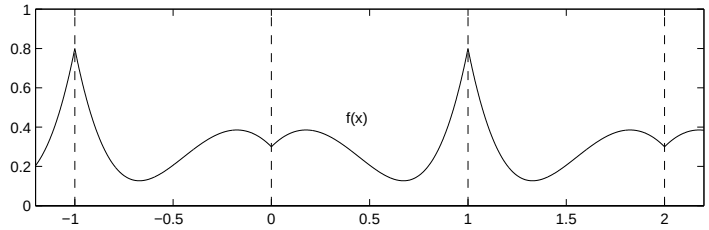


Figure 2.5: Symmetric extension of $f(x) \in L^2([0,1])$.

If $f(x)$ is continuously differentiable, then $f^{sym}(x)$ is at least C^0 continuous. This means that $\psi_{j,k}^{sym}(x)$ produces smaller wavelet coefficients than $\psi_{j,k}^{per}(x)$ near $x = 0$ and $x = 1$. However, because $f^{sym}(x)$ is not continuously differentiable at these end points, we may still obtain larger coefficients near the boundary than inside. Especially, if $f(x)$ is very smooth.

For $\psi_{j,k}^{sym}(x)$ to be used in the construction of a basis for $L^2([0,1])$, $\psi_{j,k}(x)$ must be (anti)symmetric around $x = 1/2$. In Section 2.2.1 it was mentioned that except for the Haar wavelet no symmetric real orthogonal wavelet with compact support exists. However, there exist biorthogonal wavelets with compact support that are (anti)symmetric.

2.6.4 Boundary Wavelets and Lifting

Given a smooth signal and a wavelet with enough vanishing moments the wavelet transform will result in small wavelet coefficients. The wavelets $\psi_{j,k}^{per}(x)$ and $\psi_{j,k}^{sym}(x)$ constructed in the two previous sections have respectively 0 and 1 vanishing moments. Therefore these wavelets may generate large coefficients near the boundary as if the signal or its derivative was discontinuous. In order to take full advantage of the signals regularity near the border special boundary wavelets that have as many vanishing moments as the original wavelets $\psi_{j,k}$ have to be constructed. Several methods for constructing such wavelets exist. In this section we briefly mention two such methods. In [15], Cohen et al. show how to modify the Daubechies wavelets to the interval and provide fast algorithms for the wavelet transform. As the algorithm switches filters near the boundary it is more complicated than the FWT given in Theorem 2.1 and therefore not as fast. Also the construction of the boundary filters is quite technical.

A more intuitive construction is given by Fernández et al. in [26] using the lifting scheme of Sweldens [80]. The wavelets constructed there are biorthogonal spline wavelets. These wavelets are symmetric, have compact support, and may have an arbitrary number of vanishing moments. The algorithm also changes behavior near the boundary. Furthermore, note that thresholding as described in Section 2.5 is no longer optimal when biorthogonal wavelets are used. In [32] Gross proposes a method for determining the most significant coefficients for biorthogonal transforms. This method suffers from its complexity, and often the magnitude based thresholding is used despite being suboptimal, e.g., see the JPEG2000 standard [41].

2.7 Chapter Summary

This chapter has explained the theory of wavelets in terms of multiresolution analysis and we presented the fast discrete wavelet transform of Mallat. We looked at the properties that have made wavelet so successful in image compression. Especially, the energy compacting property combined with thresholding is very important in generating very sparse representations for lossy compression. We also briefly explained how to extend wavelets to multiple dimensions and we discussed issues with boundary wavelets.

Chapter 3

Introduction to Dictionaries

The *dictionary* data structure is ubiquitous in computer science. A dictionary is used to maintain a set S under insertion and deletion of elements, referred to as *keys*, from a universe U. Membership queries, like "$x \in S$?", provide access to the data. In case of a positive answer the dictionary also provides a piece of *satellite data* that was associated with x when it was inserted.

A large literature has grown around the problem of constructing efficient dictionaries, and theoretically satisfying solutions have been found. Often a slightly easier problem has been considered, namely the *membership* problem, which is the dictionary problem without associated information. It is usually easy to derive a dictionary from a solution to the membership problem, using extra space corresponding to the associated information. E.g., in all methods discussed in this dissertation, with the exception of Bloom filtering and similar methods, the associated information of $x \in S$ can be stored together with x in a hash table. In this dissertation we are particularly interested in dictionary and membership schemes using little memory. In the following we let n denote $|S|$.

The most efficient dictionaries, in theory and in practice, are based on hashing techniques. The main performance parameters are of course lookup time, update time, and space. In theory, there is no trade-off between these. One can simultaneously achieve constant lookup time, expected amortized constant update time, and space within a constant factor of the information theoretical minimum of $B = \log \binom{|U|}{n}$ bits [10]. In practice, however, the various constant factors are crucial in many applications. In particular, lookup time is a critical parameter. It is well known that the expected time for all operations can be made within a factor of $(1 + \rho)$ from optimal (one universal hash function evaluation, one memory lookup) if space $O(n/\rho)$ is allowed. Therefore the challenge is to combine speed with a reasonable space usage. In particular, we only consider schemes using $O(n)$ words of space.

In Chapter 4 and Chapter 5 we present two new dictionary schemes based on hashing. The first scheme, which is the focus of Chapter 4, is a new dynamic hashing scheme called Cuckoo hashing. We show that the new scheme has worst case constant lookup time and amortized expected constant time for updates. The space usage equals that of binary search trees. Furthermore, we

show through extensive experiments that CUCKOO HASHING is quite competitive with the most commonly used dictionary methods, having nontrivial worst case lookup time. A large literature, surveyed in Chapter 4.1.1, is devoted to practical and theoretical aspects of dictionaries.

The second method, presented in Chapter 5.2, was partly developed as a tool for lossy compression with fast random access. In this setting space and lookup speed are of the essence. Current dictionary methods that are (nearly) optimal with respect to space are not practical to implement and rather slow in practice. However, if one relaxes the requirements to the membership data structure, allowing it to store a slightly different key set than intended, new possibilities arise. Since the data structure is allowed some errors with regards to the elements it reports as being in the set, we refer to it as a *lossy dictionary*. In Chapter 5.2 we present such a data structure and examine its properties both theoretically and experimentally. We find that it has very good experimental behavior which is partly explained by theoretical results.

Chapter 4

Cuckoo Hashing

4.1 Introduction

The contribution of this chapter is a new, simple to implement hashing scheme called Cuckoo hashing. A description and analysis of the scheme is given in Section 4.3, showing that it possesses the same theoretical properties as the dynamic dictionary of Dietzfelbinger et al. [21]. That is, it has worst case constant lookup time and amortized expected constant time for updates. A special feature of the lookup procedure is that (disregarding accesses to a small hash function description) there are just two memory accesses, which are independent and can be done in parallel if this is supported by the hardware. Our scheme works for space similar to that of binary search trees, i.e., three words per key in S on average.

Using weaker hash functions (simpler and easier to implement) than those required for our analysis, Cuckoo hashing is very simple to implement. Section 4.4 describes such an implementation, and reports on extensive experiments and comparisons with the most commonly used methods, having no worst case guarantee on lookup time. To our knowledge an experiment comparing the most commonly used methods on a modern multi-level memory architecture has not previously been described in the literature. Our experiments show Cuckoo hashing to be quite competitive, especially when the dictionary is small enough to fit in cache. We thus believe it to be attractive in practice, when a worst case guarantee on lookups is desired.

4.1.1 Previous Work on Linear Space Dictionaries

Hashing, first described in public literature by Dumey [23], emerged in the 1950s as a space efficient heuristic for fast retrieval of keys in sparse tables. Knuth surveys the most important classical hashing methods in [47, Section 6.4]. These methods also seem to prevail in practice. The most prominent ones, and the basis for our experiments in Section 4.4, are Chained Hashing (with separate chaining), Linear Probing, and Double Hashing. A more recent scheme called Two-Way Chaining [3] will also be investigated. We detail our implementation in Section 4.4.

Theoretical Work.

Early theoretical analysis of hashing schemes was typically done under the assumption that hash function values were uniformly random and independent. Precise analyses of the average and expected worst case behaviors of the above-mentioned schemes have been made, see e.g. [30, 47]. We mention just a few facts, disregarding asymptotically vanishing terms.

For Linear Probing the expected number of memory probes for successful and unsuccessful lookups are $\frac{1}{2}(1+\frac{1}{1-\alpha})$ and $\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$, respectively, where α denotes the fraction of the table occupied by keys, $0 < \alpha < 1$. The longest probe sequence is of expected length $\Omega(\log n)$. In Double Hashing the expected costs of successful and unsuccessful lookups are, respectively, $\ln(\frac{1}{1-\alpha})/\alpha$ and $\frac{1}{1-\alpha}$. The longest successful probe sequence is expected to be of length $\Omega(\log n)$, and there is *no bound* on the length of unsuccessful searches . For Chained Hashing, lookups have expected costs $1+\alpha/2$ and $1+\alpha^2/2$, respectively, for hash table size n/α. The expected maximum chain length is $\Theta(\log n/\log\log n)$. In terms of the number of probes, the above implies that Chained Hashing is better than Double Hashing, which is again better than Linear Probing. Note that for these three schemes, an insertion corresponds to an unsuccessful lookup, and that a deletion corresponds to a successful lookup. However, because of excellent cache usage we will see in Section 4.4 that on average Linear Probing is the best performer.

Two-Way Chaining is an alternative to Chained Hashing that offers $O(\log\log n)$ expected maximal lookup time. The implementation that we consider represents the lists by arrays of size $O(\log\log n)$. To achieve linear space usage, one must then use a hash table of size $O(n/\log\log n)$, implying that the average chain length is $\Omega(\log\log n)$. Another scheme with expected $O(\log\log n)$ time per operation is *Multilevel Adaptive Hashing* [8]. However, lookups can be performed in $O(1)$ worst case time if $O(\log\log n)$ hash function evaluations, memory probes and comparisons are possible in parallel. This is similar to the scheme described in this paper, though we use only two hash function evaluations, two memory probes, and two comparisons.

Though the results seem to agree with practice, the randomness assumptions used for the above analyses are questionable in applications. Carter and Wegman [14] succeeded in removing such assumptions from the analysis of chained hashing, introducing the concept of *universal* hash function families. When implemented with a random function from Carter and Wegman's universal family, chained Hashing has constant expected time per dictionary operation (plus an amortized expected constant cost for resizing the table). Constructions of universal hash function families with very efficient evaluation have since appeared [18, 20, 85].

A dictionary with worst case constant lookup time was first obtained by Fredman et al. [27], though it was *static*, i.e., it did not support updates. It was later augmented with insertions and deletions in amortized expected constant time by Dietzfelbinger et al. [21]. Dietzfelbinger and Meyer auf der Heide [22] improved the update performance by exhibiting a dictionary in which operations are done in constant time with high probability, namely at least $1-n^{-c}$, where

c is any constant of our choice. A simpler dictionary with the same properties was later developed [19]. When $n = |U|^{1-o(1)}$ a space usage of $O(n)$ words is not within a constant factor of the information theoretical minimum B. The earlier mentioned dictionary of Brodnik and Munro [10] offers the same performance as [21], using $O(B)$ bits in all cases.

Experimental Work.

Although the above results leave little to improve from a theoretical point of view, large constant factors and complicated implementation hinder direct practical use. For example, in the "dynamic perfect hashing" scheme of [21] the upper bound on space is $35n$ words. The authors of [21] refer to a more practical variant due to Wenzel [90] that uses space comparable to that of binary search trees.

According to [45] the implementation of this variant in the LEDA library [56], described in [90], has average insertion time larger than that of AVL trees for $n \leq 2^{17}$, and more than four times slower than insertions in chained hashing[1]. The experimental results listed in [56, Table 5.2] show a gap of more than a factor of 6 between the update performance of chained hashing and dynamic perfect hashing, and a factor of more than 2 for lookups[2].

Silverstein [79] reports that the space upper bound of the dynamic perfect hashing scheme of [21] is quite pessimistic compared to what can be observed when run on a subset of the DIMACS dictionary tests [55]. He goes on to explore ways of improving space as well as time, improving both the observed time and space by a factor of roughly three. Still, the improved scheme needs 2 to 3 times more space than an implementation of linear probing to achieve similar time per operation. Silverstein also considers versions of the data structures with packed representations of the hash tables. In this setting the dynamic perfect hashing scheme was more than 50% slower than linear probing, using roughly the same amount of space.

Is seems that recent experimental work on "classical" dictionaries (that do not have worst case constant lookup time) is quite limited. In [45] it is reported that chained hashing is superior to an implementation of dynamic perfect hashing in terms of both memory usage and speed. Judging from leading textbooks on algorithms, Knuth's selection of algorithms is in agreement with current practice for implementation of general purpose dictionaries. In particular, the excellent cache usage of Linear Probing makes it a prime choice on modern architectures.

4.2 Preliminaries

It is common to study the case where keys are bit strings in $U = \{0,1\}^w$ and w is the word length of the computer (for theoretical purposes modeled as a RAM). This restriction is discussed below. A special value $\perp \in U$ is reserved

[1] On a Linux PC with an Intel® Pentium® 120 MHz processor.
[2] On a 300 MHz SUN ULTRA SPARC.

to signal an empty cell in hash tables. For DOUBLE HASHING an additional special value is used to indicate a deleted key.

Our algorithm uses hash functions from a *universal* family.

Definition 4.1 *A family* $\{h_i\}_{i\in I}$, $h_i : U \to R$, *is* (c,k)-universal *if, for any* k *distinct elements* $x_1,\ldots,x_k \in U$, *any* $y_1,\ldots,y_k \in R$, *and uniformly random* $i \in I$, $\Pr[h_i(x_1) = y_1,\ldots,h_i(x_k) = y_k] \le c/|R|^k$.

A standard construction of a $(2,k)$-universal family for prime $p > 2^w$ and range $R = \{0,\ldots,r/2-1\}$ is

$$\{x \mapsto ((\sum_{l=0}^{k-1} a_l x^l) \bmod p) \bmod r/2 \quad | \quad 0 \le a_0, a_1, \ldots, a_{k-1} < p\} \ . \tag{4.1}$$

If U is not too large compared to k, there exists a space-efficient $(2,k)$-universal family due to Siegel [78] that has *constant* evaluation time. However, the constant factor of the evaluation time is rather high.

Theorem 4.1 (Siegel) *There is a constant* c *such that, for* $k = 2^{\Omega(w)}$, *there exists a* $(2,k)$*-universal family, using space and initialization time* k^c, *that can be evaluated in* constant *time.*

The restriction that keys are single words is not a serious one. Longer keys can be mapped to keys of $O(1)$ words by applying a random function from a $(O(1),2)$-universal family. There is such a family whose functions can be evaluated in time linear in the number of input words [14]. It works by evaluating a function from a $(O(1),2)$-universal family on each word, computing the bitwise exclusive or of the function values. See [85] for an efficient implementation. Such a function with range $\{0,1\}^{2\log(n)+c}$ will, with probability $1 - O(2^c)$, be injective on S. In fact, with constant probability the function is injective on a given sequence of $\Omega(2^{c/2}n)$ consecutive sets in a dictionary of initial size n (see [21]). When a collision between two elements of S occurs, everything is rehashed. If a rehash can be done in expected $O(n)$ time, the amortized expected cost of this is $O(2^{-c/2})$ per insertion. In this way we can effectively reduce the universe size to $O(n^2)$, though the full keys still need to be stored to decide membership. For $c = O(\log n)$ the new keys are of length $2\log n + O(1)$ bits. For any $\delta > 0$, Theorem 4.1 then provides a family of constant time evaluable $(2, n^{\Omega(1)})$-universal hash functions, whose functions can be stored using space n^δ.

4.3 Algorithm – Cuckoo Hashing

CUCKOO HASHING is a dynamization of a static dictionary described in [62]. The dictionary uses two hash tables, T_1 and T_2, each of length $r/2$ and two hash functions $h_1, h_2 : U \to \{0,\ldots,r/2-1\}$. Every key $x \in S$ is stored in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2, but never in both. Our lookup function is

```
function lookup(x)
  return T1[h1(x)] = x ∨ T2[h2(x)] = x
end
```

Two table accesses are in fact (worst case) optimal among all data structures using linear space, except for special cases, see [62].

Remark: The idea of storing keys in one out of two places given by hash functions previously appeared in [44] in the context of PRAM simulation, and in [3] for Two-Way Chaining, mentioned in Section 4.1.1.

It is shown in [62] that if $r/2 \geq (1+\rho)\,n$ for some constant $\rho > 0$ (i.e., the tables are to be a bit less than half full), and h_1, h_2 are picked uniformly at random from an $(O(1), O(\log n))$-universal family, the probability that there is no way of arranging the keys of S according to h_1 and h_2 is $O(1/n)$. By the discussion in Section 4.2 we may assume without loss of generality that there is such a family, with constant evaluation time and negligible space usage. A suitable arrangement of the keys in the two hash tables was shown in [62] to be computable in linear time by a reduction to 2-SAT.

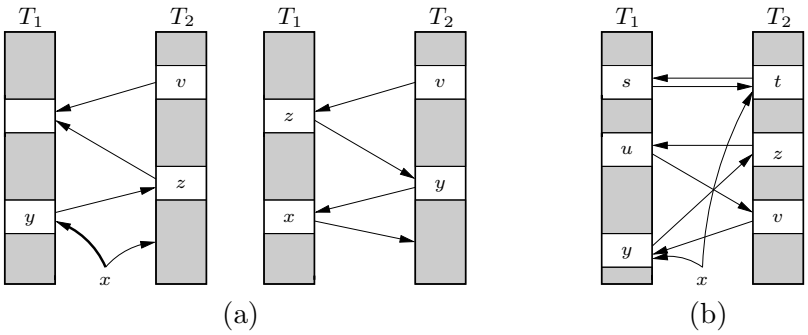


Figure 4.1: (a) Key x is successfully inserted by moving keys y and z to the other table. (b) Key x cannot be accommodated and a rehash is necessary. Arrows point to the other possible location of the keys.

We now consider a dynamization of the above. Deletion is of course simple to perform in constant time, not counting the possible cost of shrinking the tables if they are becoming too sparse. As for insertion, it turns out that the "cuckoo approach", kicking other keys away until every key has its own "nest", works very well. Specifically, if x is to be inserted we first see if cell $h_1(x)$ of T_1 is occupied. If not, we are done. Otherwise we set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "nestless". This key is then inserted in T_2 in the same way, and so forth, see Figure 4.1(a). As it may happen this process loops, see Figure 4.1(b), the number of iterations is bounded by a value "MaxLoop" to be specified in Section 4.3.1. If this number of iterations is reached everything is rehashed with new hash functions, and we try once again to accommodate the nestless key. Using the notation $x \leftrightarrow y$ to express that the values of variables x and y are swapped, the following code summarizes the insertion procedure.

```
procedure insert(x)
  if lookup(x) then return
  loop MaxLoop times
    if T1[h1(x)] = ⊥ then { T1[h1(x)] ← x; return }
    x ↔ T1[h1(x)]
    if T2[h2(x)] = ⊥ then { T2[h2(x)] ← x; return }
    x ↔ T2[h2(x)]
  end loop
  rehash(); insert(x)
end
```

The above procedure assumes that the tables remain larger than $(1+\rho)\,n$ cells. When no such bound is known, a test must be done to find out when a rehash to larger tables is needed.

The lookup call preceding the insertion in the procedure ensures robustness if the key to be inserted is already in the dictionary. A slightly faster implementation can be obtained if this is known not to occur.

Note that the insertion procedure is biased towards inserting keys in T_1. As will be seen in Section 4.4 this leads to faster successful lookups, due to more keys being found in T_1. The insertion time is only slightly worse than that of a more symmetric implementation. This effect is even more pronounced if one uses an asymmetric scheme where T_1 is larger than T_2. Another variant is to use a single table for both hash functions, but this requires keeping track of the hash function according to which each key is placed. In the following we consider just the basic two-table scheme.

4.3.1 Analysis

Our analysis has two main parts:

- First we consider what happens if one tries arbitrarily long to insert the new key, i.e., MaxLoop $= \infty$. We show that if the insertion procedure does not terminate, it is not possible to accommodate all the keys of the new set using the present hash functions, and a rehash is necessary. In conjunction with the result from [62], this shows that the insertion procedure loops without limit with probability $O(1/n)$.
- Next we turn to the analysis for the case where insertion is possible, showing that the insertion procedure terminates in $O(1)$ iterations, in the expected sense.

This accounts for the claimed time bound, except for the cost of rehashing. A rehash has no failed insertions with probability $1 - O(1/n)$. In this case, the expected time per insertion is constant, so the expected time is $O(n)$. Because the probability of having to start over with new hash functions is bounded away from 1, the total expected time for a rehash is $O(n)$. This implies that the expected time for insertion is constant if $r/2 \geq (1+\rho)(n+1)$. Resizing of tables can be done in amortized expected constant time per update by the usual doubling/halving technique, e.g. see [21].

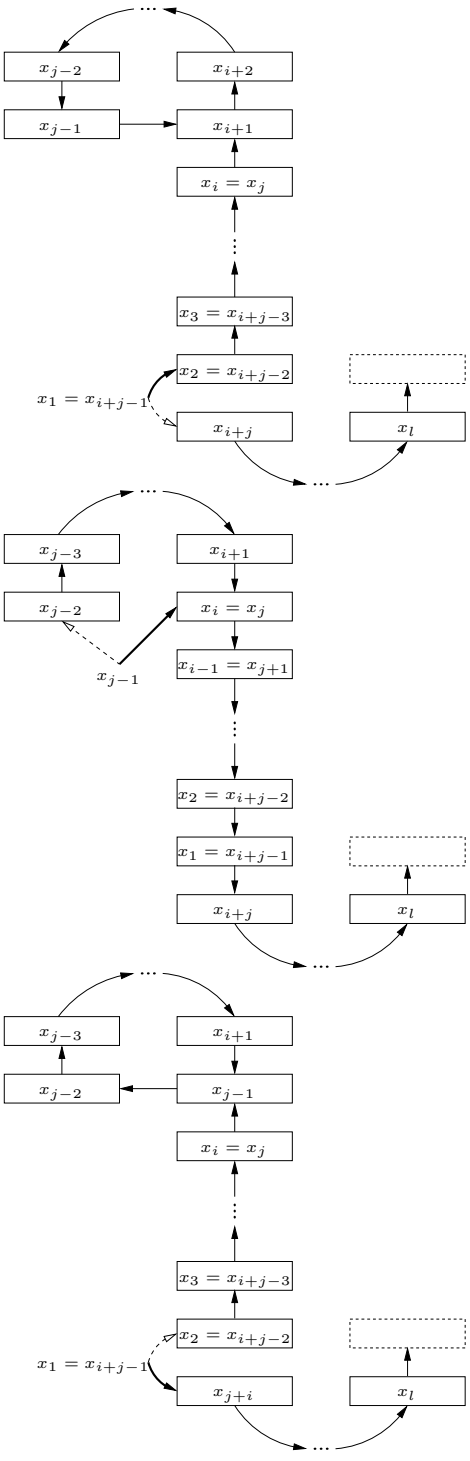


Figure 4.2: Stages of an insertion of key x_1. Boxes correspond to cells in either of the two tables, and arcs indicate the other possible position of a key. Bold arcs show where the nestless key is to be inserted.

The Insertion Procedure Loops

Consider the sequence $x_1, x_2, \ldots$ of nestless keys in the infinite loop. For $i, j \geq 1$ we define $X_{i,j} = \{x_i, \ldots, x_j\}$. Let j be the smallest index such that $x_j \in X_{1,j-1}$, and let l be the minimum index such that $l + 1 > j$ and $x_{l+1} \in X_{1,l}$.

We now argue that the first l steps of the insertion proceed as depicted in Figure 4.2. The topmost configuration in the figure is the one preceeding the insertion of x_1. The configuration just before x_j becomes nestless for the second time is shown in the middle of the figure. One step later we have that x_k is now in the previous location of x_{k+1}, for $1 \leq k < j$. Let $i < j$ be the index such that $x_i = x_j$. We now consider what happens towards the third stage. If $i > 1$ then x_j reclaims its previous location, occupied by x_{i-1}. If $i > 2$ then x_{i-1} subsequently reclaims its previous position, which is occupied by x_{i-2}, and so forth. Thus we have $x_{j+z} = x_{i-z}$ for $z = 0, 1, \ldots, i-1$, and end up with x_1 occurring again as x_{i+j-1}. This is shown in the third stage of the figure. Note that the dotted cell must, by definition of l, be identical to one of the other cells in the figure.

It is easy to see that the number of cells is not sufficient to accommodate $X_{i,l}$ for the current choice of hash functions. For a formal proof, we define $s_k = |h_1[X_{1,k}]| + |h_2[X_{1,k}]|$, i.e., s_k is the number of table cells available to $X_{1,k}$. Obviously $s_k \leq s_{k-1} + 1$, as every key x_i, $i > 1$, has either $h_1(x_i) = h_1(x_{i-1})$ or $h_2(x_i) = h_2(x_{i-1})$. In fact, $s_{j-1} = s_{j-2} \leq j - 1$, because the key x_j found in $T_1[h_1(x_{j-1})]$ or $T_2[h_2(x_{j-1})]$ occurred earlier in the sequence. As all of the keys $x_j, \ldots, x_{j+i-1}$ appeared earlier in the sequence, we have $s_{j+i-2} = s_{j-2}$. Similar to before we have $s_l = s_{l-1}$. In conclusion, $|X_{1,l}| = l + 1 - i$ and $s_l = s_{l-1} \leq s_{j+i-2} + (l-1) - (j+i-2) = s_{j-2} + l + 1 - j - i < l + 1 - i$.

Successful Insertion

Consider a prefix $x_1, x_2, \ldots, x_l$ of the sequence of nestless keys. The crucial fact is that there must be a subsequence of at least $l/3$ keys without repetitions, starting with an occurrence of the key x_1, i.e., the inserted key. If there is no repetition this is obvious. Otherwise the first l steps of the insertion proceed as in Figure 4.2. In particular, one of the sequences $x_1, \ldots, x_{j-1}$ and $x_{j+i-1}, \ldots, x_l$ is the desired one of length at least $l/3$.

Suppose that the insertion loop runs for at least t iterations. By the above there is a sequence of distinct keys $b_1, \ldots, b_m$, $m \geq (2t-1)/3$, such that b_1 is the key to be inserted, and such that for some $\beta \in \{0, 1\}$

$$h_{2-\beta}(b_1) = h_{2-\beta}(b_2),\ h_{1+\beta}(b_2) = h_{1+\beta}(b_3),\ h_{2-\beta}(b_3) = h_{2-\beta}(b_4), \ldots \qquad (4.2)$$

Given b_1 there are at most n^{m-1} sequences of m distinct keys. For any such sequence and any $\beta \in \{0, 1\}$, if the hash functions were chosen from a (c, m)-universal family, the probability that (4.2) holds is bounded by $c\,(r/2)^{-(m-1)}$. Thus, the probability that there is *any* sequence of length m satisfying (4.2) is bounded by $2c\,(2n/r)^{m-1} \leq 2c\,(1+\rho)^{-(2t-1)/3+1}$. Suppose we are using a $(c, 6\log_{1+\rho} n)$-universal family, for some constant c. Then the probability of more than $3\log_{1+\rho} n$ iterations is $O(1/n^2)$. Thus, we can set MaxLoop =

$3\log_{1+\rho} n$ with a negligible increase in the probability of a rehash. When there is no rehash the expected number of iterations is at most

$$\begin{aligned} & 1+\sum_{t=2}^{\infty} 2c\,(1+\epsilon)^{-(2t-1)/3+1} \\ & = 1 + 2c(1+\epsilon)^{4/3} \sum_{t=0}^{\infty}((1+\epsilon)^{-2/3})^t \\ & = 1 + O(\tfrac{1}{1-(1+\epsilon)^{-2/3}}) \\ & = O(1+1/\epsilon)\ . \end{aligned} \tag{4.3}$$

4.4 Experiments

To examine the practicality of Cuckoo Hashing we experimentally compare it to three well known hashing methods, described by Knuth in [47, Section 6.4]: Chained Hashing (with separate chaining), Linear Probing and Double Hashing. We also consider Two-Way Chaining [3].

The first three methods all attempt to store a key x at position $h(x)$ in a hash table. They differ in the way collisions are resolved, i.e., what happens when two or more keys hash to the same location.

Chained Hashing. A chained list is used to store all keys hashing to a given location.

Linear Probing. A key is stored in the next empty table entry. Lookup of key x is done by scanning the table beginning at $h(x)$ and ending when either x or an empty table entry is found. When deleting a key, some keys may have to be moved back in order to fill the hole in the lookup sequence, see [47, Algorithm R] for details.

Double Hashing. Insertion and lookup are similar to Linear Probing, but instead of searching for the next position one step at a time, a second hash function value is used to determine the step size. Deletions are handled by putting a "deleted" marker in the cell of the deleted key. Lookups skip over deleted cells, while insertions overwrite them.

The fourth method, Two-Way Chaining, can be described as two instances of Chained Hashing. A key is inserted in one of the two hash tables, namely the one where it hashes to the shortest chain. A cache-friendly implementation, as recently suggested in [9], is to simply make each chained list a short, fixed size array. If a longer list is needed, a rehash must be performed.

4.4.1 Data Structure Design and Implementation

We consider positive 32 bit signed integer keys and use 0 as $\perp$. The data structures are *robust* in that they correctly handle attempts to insert an element already in the set, and attempts to delete an element not in the set. During a

rehash, this is known not to occur and slightly faster version of the insertion procedure is used.

Our focus is on achieving high performance dictionary operations with a reasonable space usage. By the *load factor* of a dictionary we will understand the size of the set relative to the memory used[3]. As seen in [47, Figure 44] the speed of Linear Probing and of Double Hashing degrades rapidly for load factors above 1/2. On the other hand, none of the schemes improve much for load factors below 1/4. As Cuckoo Hashing only works when the size of each table is larger than the size of the set, we can only perform a comparison for load factors less than 1/2. To allow for doubling and halving of the table size, we let the load factor to vary between 1/5 and 1/2, focusing especially on the typical load factor of 1/3. For Cuckoo Hashing and Two-Way Chaining there is a chance that an insertion may fail, causing a forced rehash. If the load factor is larger than a certain threshold, somewhat arbitrarily set to 5/12, we use the opportunity to double the table size. By our experiments this only slightly decreases the average load factor.

Apart from Chained Hashing, the schemes considered have in common the fact that they have only been analyzed under randomness assumptions that are currently, or inherently, unpractical to implement, i.e., $O(\log n)$-wise independence or n-wise independence. However, experience shows that rather simple and efficient hash function families yield performance close to that, predicted under stronger randomness assumptions. We use a function family from [20] with range $\{0,1\}^q$ for positive integer q. For every odd a, $0 < a < 2^w$, the family contains the function $h_a(x) = (ax \bmod 2^w) \operatorname{div} 2^{w-q}$. Note that evaluation can be done very efficiently by a 32 bit multiplication and a shift. However, this choice of hash function restricts us to consider hash tables whose sizes are powers of two. A random function from the family (chosen using C's `rand` function) appears to work fine with all schemes except Cuckoo Hashing. For Cuckoo Hashing we experimented with various hash functions and found that Cuckoo Hashing was rather sensitive to the choice of hash function. It turned out that the exclusive or of three independently chosen functions from the family of [20] was fast and worked well. We have no good explanation for this phenomenon. For all schemes, various alternative hash families were tried, with a decrease in performance.

All methods have been implemented in C. We have striven to obtain the fastest possible implementation of each scheme. Specific choices made and details differing from the references are:

Chained Hashing. C's `malloc` and `free` functions were found to be a performance bottleneck, so a simple "freelist" memory allocation scheme is used. Half of the allocated memory is used for the hash table, and half for list elements. If the data structure runs out of free list elements, its size is doubled. We store the first element of each linked list directly in the hash table. This often saves one cache miss. It also slightly improves memory utilization, in the expected sense. This is because every

[3]For Chained Hashing, the notion of load factor traditionally disregards the space used for chained lists, but we desire equal load factors to imply equal memory usage.

nonempty chained list is one element shorter and because we expect more than half of the hash table cells to contain a linked list for the load factors considered here.

DOUBLE HASHING. To prevent the tables from clogging up with deleted cells, resulting in poor performance for unsuccessful lookups, all keys are rehashed when 2/3 of the hash table are occupied by keys and "deleted" markers. The fraction 2/3 was found to give a good tradeoff between the time for insertion and unsuccessful lookups.

LINEAR PROBING. Our first implementation, like that in [79], employed deletion markers. However, we found that using the deletion method described in [47, Algorithm R] was considerably faster, as far fewer rehashes were needed. Note that the consecutive search for a key or an empty cell makes excellent use of the cache on most computer architectures.

TWO-WAY CHAINING. We allow four keys in each bucket. This is enough to keep the probability of a forced rehash low for hundreds of thousands of keys, by the results in [9]. For larger collections of keys one should allow more keys in each bucket, resulting in general performance degradation.

CUCKOO HASHING. The architecture on which we experimented could not parallelize the two memory accesses in lookups. Therefore we only evaluate the second hash function after the first memory lookup has shown unsuccessful.

Some experiments were done with variants of CUCKOO HASHING. In particular, we considered ASYMMETRIC CUCKOO, in which the first table is twice the size of the second one. This results in more keys residing in the first table, thus giving a slightly better average performance for successful lookups. For example, after a long sequence of alternate insertions and deletions at load factor 1/3, we found that about 76% of the elements resided in the first table of ASYMMETRIC CUCKOO, as opposed to 63% for CUCKOO HASHING. There is no significant slowdown for other operations. We will describe the results for ASYMMETRIC CUCKOO when they differ significantly from those of CUCKOO HASHING.

4.4.2 Setup

Our experiments were performed on a PC running Linux (kernel version 2.2) with an 800 MHz Intel® Pentium® III processor, and 256 MB of memory (PC100 RAM). The processor has a 16 KB level 1 data cache and a 256 KB level 2 "advanced transfer" cache. As will be seen, our results nicely fit a simple model parameterized by the cost of a cache miss and the expected number of probes to "random" locations. The results are thus believed to have significance for other hardware configurations. An advantage of using the Pentium® III processor for timing experiments is its `rdtsc` instruction which can be used to measure time in clock cycles. This gives access to very precise data on the behavior of functions. In our case it also supplies a way of discarding

measurements significantly disturbed by interrupts from hardware devices or the process scheduler, as these show up as a small group of timings significantly separated from all other timings. Programs were compiled using the `gcc` compiler version 2.95.2, using optimization flags `-O9 -DCPU=586 -march=i586 -fomit-frame-pointer -finline-functions -fforce-mem -funroll-loops -fno-rtti`. As mentioned earlier, we use a global clock cycle counter to time operations. If the number of clock cycles spent exceeds 5000, and there was no rehash, we conclude that the call was interrupted, and disregard the result (it was empirically observed that no operation ever took between 2000 and 5000 clock cycles). If a rehash is made, we have no way of filtering away time spent during interrupts. However, all tests were made on a machine with no irrelevant user processes, so disturbances should be minimal. On our machine it took 32 clock cycles to call the `rdtsc` instruction. These clock cycles have been subtracted from the results.

4.4.3 Results

Dictionaries of Stable Size

Our first test was designed to model the situation in which the size of the dictionary is not changing too much. It considers a sequence of mixed operations with keys generated at random. We constructed the test operation sequences from a collection of high quality random bits publicly available on the Internet [54]. The sequences start by insertion of n distinct random keys, followed by $3n$ times four operations: A random unsuccessful lookup, a random successful lookup, a random deletion, and a random insertion. We timed the operations for several n in the equilibrium, where the number of elements is stable. For load factor 1/3 our results appear in Figure 4.3 and Figure 4.4, which shows an average over 10 runs. As Linear Probing was consistently faster than Double Hashing, we chose it as the sole open addressing scheme in the plots. Time for forced rehashes was added to the insertion time. Results had a large variance, over the 10 runs, for sets of size 2^{12} to 2^{16}. Outside this range the extreme values deviated from the average by less than about 7%. The large variance sets in when the data structure starts to fill up the level 2 cache. We believe it is due to other processes evicting parts of the data structure from cache.

As can be seen, the time for lookups is almost identical for all schemes as long as the entire data structure fits in level 2 cache, i.e., for $n < 2^{16}/3$. After this the average number of random memory accesses (with the probability of a cache miss approaching 1) shows up. This makes linear probing an average case winner, with Cuckoo Hashing and Two-Way Chaining following about 40 clock cycles behind. For insertion the number of random memory accesses again dominates for large sets, while a higher number of in-cache accesses and more computation makes Cuckoo Hashing, and in particular Two-Way chaining, relatively slow for small sets. The cost of forced rehashes sets in for Two-Way Chaining for sets of more than a million elements, at which point better results may have been obtained by a larger bucket size. For deletion

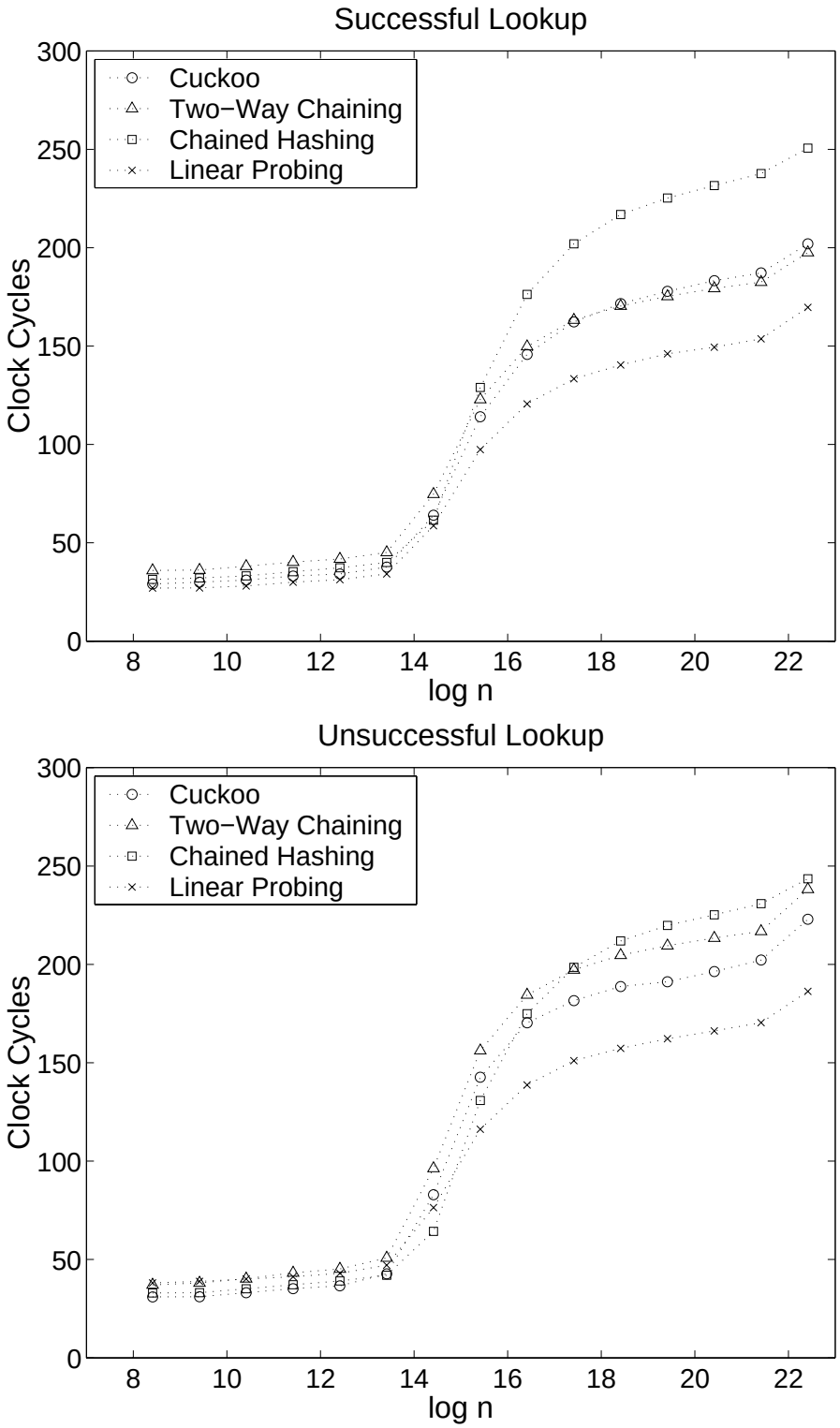


Figure 4.3: The average time per operation in equilibrium for load factor 1/3.

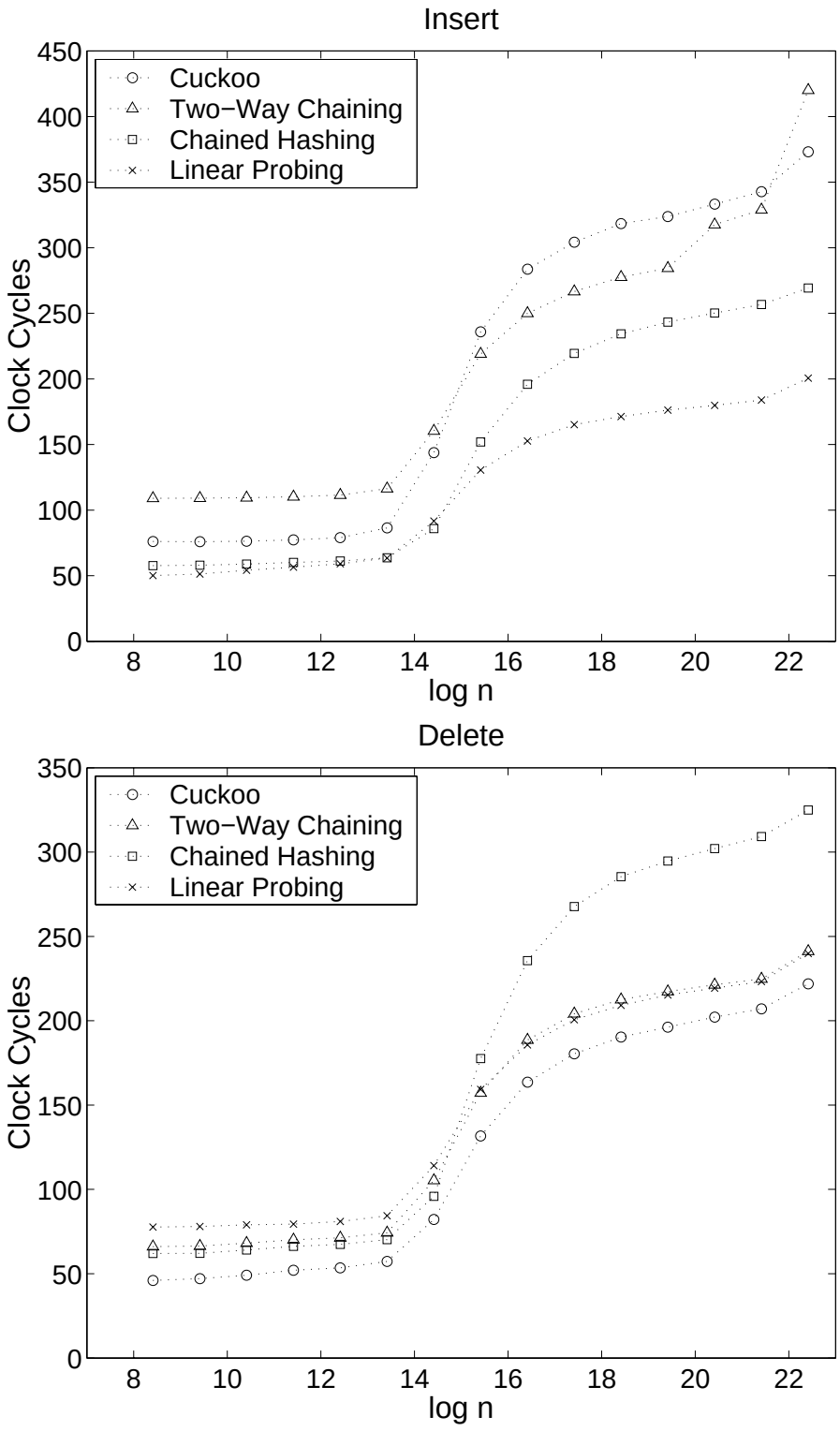


Figure 4.4: The average time per operation in equilibrium for load factor 1/3.

Chained Hashing lags behind for large sets due to random memory accesses when freeing list elements, while the simplicity of Cuckoo Hashing makes it the fastest scheme. We conjecture the slight rise in time for the largest sets is due to saturation of the bus, as the machine runs out of memory and begins swapping.

It is interesting to note that according to the theoretical results mentioned in Section 4.1.1 Double Hashing and Chained Hashing should perform better than Linear Probing.

Growing and Shrinking Dictionaries

The second test concerns the cost of insertions in growing dictionaries and deletions in shrinking dictionaries. This will be different from the above due to the cost of rehashes. Together with Figure 4.3 and Figure 4.4 this should give a fairly complete picture of the performance of the data structures under general sequences of operations. The first operation sequence inserts n distinct random keys, while the second one deletes them. The plot is shown in Figure 4.5. For small sets the time per operation seems unstable, and dominated by memory allocation overhead (The start table size was 4. If minimum table size 2^{10} is used, the curves become monotone). For sets of more than 2^{12} elements the largest deviation from the averages over 10 runs was about 6%. Disregarding the constant minimum amount of memory used by any dictionary, the average load factor during insertions was within 2% of 1/3 for all schemes except Chained Hashing whose average load factor was about 0.31. During deletions all schemes had average load factor 0.28. Again the fastest method is Linear Probing, followed by Chained Hashing and Cuckoo Hashing. This is largely due to the cost of rehashes.

DIMACS Tests

Access to data in a dictionary is rarely random in practice. In particular, the cache is more helpful than in the above random tests, for example due to repeated lookups of the same key, and quick deletions. As a rule of thumb, the time for such operations will be similar to the time when all of the data structure is in cache. To perform actual tests of the dictionaries on more realistic data, we chose a representative subset of the dictionary tests of the 5th DIMACS implementation challenge [55]. The tests involving string keys were preprocessed by hashing strings to 32 bit integers, as described in Section 4.2. This preserves, with high probability, the access pattern to keys. For each test we recorded the average time per operation. The minimum and maximum of six runs can be found in Tables 4.1 and 4.2, which also list the average load factor. Linear probing is again the fastest, but mostly just 20-30% faster than the Cuckoo schemes.

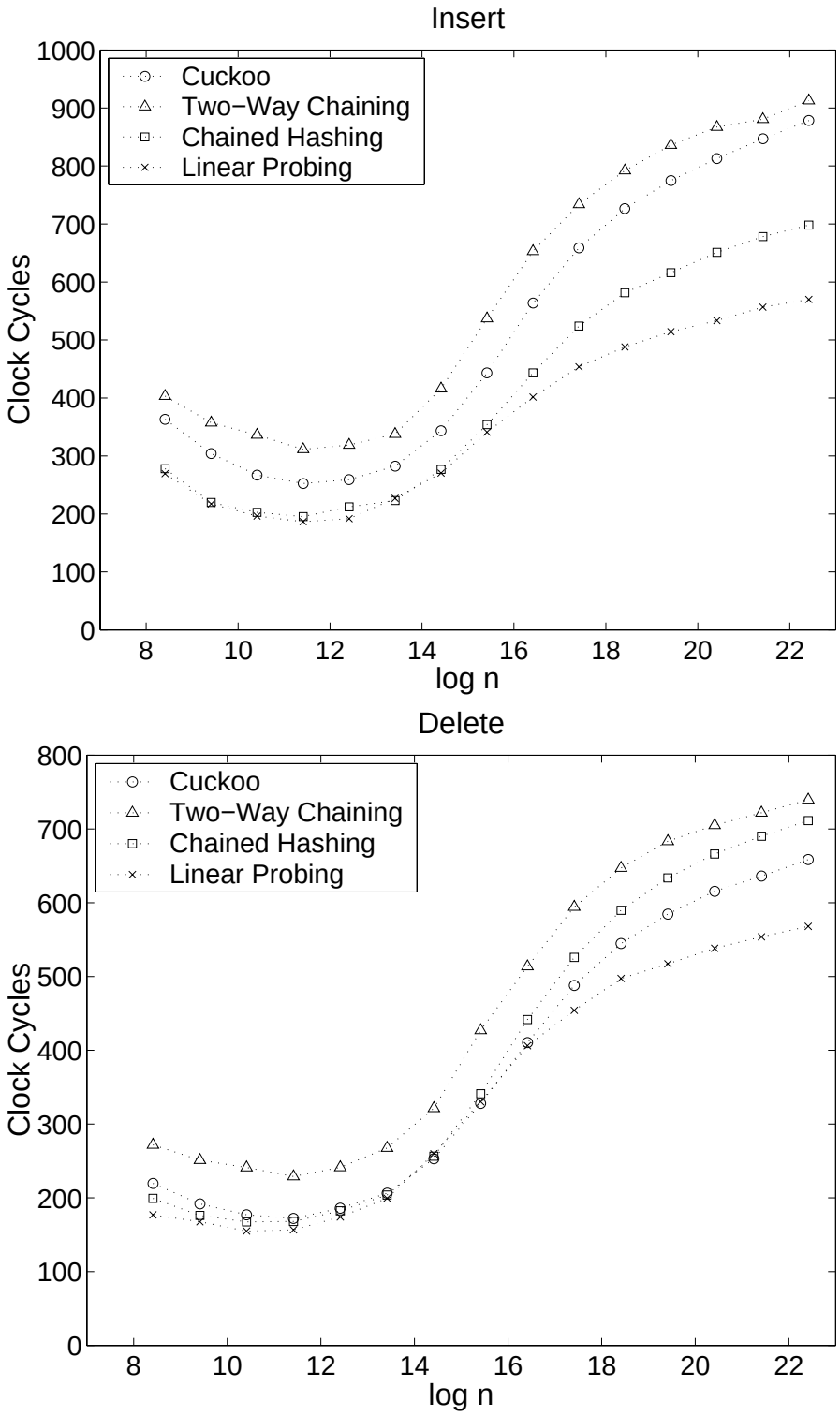


Figure 4.5: The average time per insertion/deletion in a growing/shrinking dictionary for average load factor $\approx 1/3$.

	Joyce		Eddington	
LINEAR	42 - 45	(.35)	26 - 27	(.40)
DOUBLE	48 - 53	(.35)	32 - 35	(.40)
CHAINED	49 - 52	(.31)	36 - 38	(.28)
A.CUCKOO	47 - 50	(.33)	37 - 39	(.32)
CUCKOO	57 - 63	(.35)	41 - 45	(.40)
TWO-WAY	82 - 84	(.34)	51 - 53	(.40)

Table 4.1: Average clock cycles per operation and load factors for two DIMACS string tests.

	3.11-Q-1		Smalltalk-2		3.2-Y-1	
LINEAR	99 - 103	(.30)	68 - 72	(.29)	85 - 88	(.32)
DOUBLE	116 - 142	(.30)	77 - 79	(.29)	98 - 102	(.32)
CHAINED	113 - 121	(.30)	78 - 82	(.29)	90 - 93	(.31)
A.CUCKOO	166 - 168	(.29)	87 - 95	(.29)	95 - 96	(.32)
CUCKOO	139 - 143	(.30)	90 - 96	(.29)	104 - 108	(.32)
TWO-WAY	159 - 199	(.30)	111 - 113	(.29)	133 - 138	(.32)

Table 4.2: Average clock cycles per operation and load factors for three DIMACS integer tests.

The Number of Cache Misses During Insertion

We have seen that the number of random memory accesses (i.e., cache misses) is critical to the performance of ashing schemes. Whereas, there is a very precise understanding of the probe behavior of the classic schemes (under suitable randomness assumptions), the analysis of the expected time for insertions in Section 4.3.1 is rather crude, establishing just a constant upper bound. One reason that our calculation does not give a very tight bound is that we use a pessimistic estimate on the number of key moves needed to accommodate a new element in the dictionary. Often a free cell will be found even though it could have been occupied by another key in the dictionary. We also pessimistically assume that a large fraction of key moves will be spent backtracking from an unsuccessful attempt to place the new key in the first table.

Figure 4.6 shows experimentally determined values for the average number of probes during insertion for various schemes and load factors below 1/2. We disregard reads and writes to locations known to be in cache, and the cost of rehashes. Measurements were made in equilibrium after 10^5 insertions and deletions, using tables of size 2^{15} and truly random hash function values. It is believed that this curve is independent of the table size (up to vanishing terms). The curve for LINEAR PROBING does not appear, as the number of non-cached memory accesses depends on cache architecture (length of the cache line), but it is typically very close to 1. The curve for CUCKOO HASHING seems to be $2 + 1/(4 + 8\alpha) \approx 2 + 1/(4\rho)$. This is in good correspondence with (4.3) of the analysis in Section 4.3.1. As noted in Section 4.3 the insertion algorithm of CUCKOO HASHING is biased towards inserting keys in T_1. If we instead of

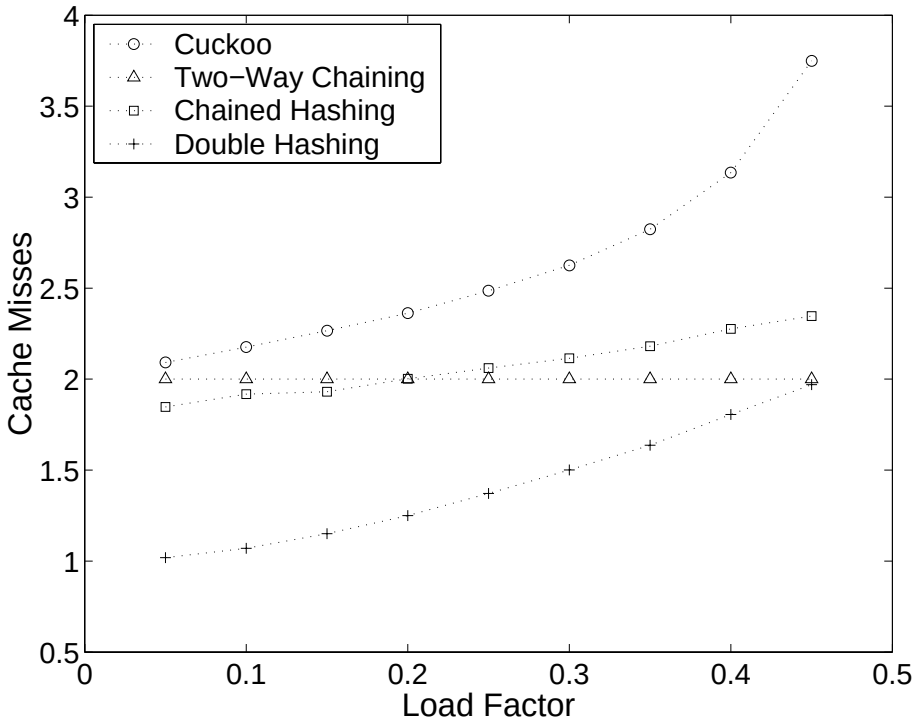


Figure 4.6: The average number of random memory accesses for insertion.

starting the insertion in T_1 choose the start table at random, the number of cache misses decreases slightly for insertion. This is because the number of free cells in T_1 increases as the load balance becomes even. However, this also means a slight increase in lookup time. Also, note that since insertion checks if the element is already inserted Cuckoo Hashing uses at least two cache misses. It should be remarked that the highest load factor for Two-Way Chaining is $O(1/\log\log n)$.

Since lookups are very similar to insertion in Chained Hashing, one could think that the number of cache misses would be equal for the two operations. However, in our implementation, obtaining a free cell from the freelist may result in an extra cache miss. This is the reason why the curve for Chained Hashing in the figure differs from a similar plot in Knuth [47, Figure 44].

4.5 Model

In this section, we look at a simple model of the time it takes to perform a dictionary operation, and note that our results can be explained in terms of this model. On a modern computer, memory speed is often the bottleneck. Since the operations of the investigated hashing methods mainly perform reads and writes to memory, we will assume that cache misses constitute the dominant part of the time needed to execute a dictionary operation. This leads to the

following model of the time per operation.

$$\text{Time} = O + N \cdot R \cdot (1 - C/T) \ , \tag{4.4}$$

where the parameters of the model are described by

- O – Constant overhead of the operation.
- R – Average number of random memory accesses.
- C – Cache size.
- T – Size of the hash tables.
- N – Cost of a non-cache read.

The term $R \cdot (1 - C/T)$ is the expected number of cache misses for the operations with $(1 - C/T)$ being the probability that a random probe into the tables results in a cache miss. Note that the model is not valid when the table size T is smaller than the cache size C. The size C of the cache and the size T of the dictionary are well known. From Figure 4.6 we can, for the various hashing schemes and for a load factor of 1/3, read the average number R of random memory accesses needed for inserting an element. Note that several accesses to consecutive elements in the hash table is counted as one random access, since the other accesses are then in cache. The overhead of an operation, O, and the cost of a cache miss, N, are unknown factors that we will estimate.

Performing experiments, reading and writing to and from memory, we observed that the time for a read or a write to a location known not to be in cache could vary dramatically depending on the state of the cache. For example, a read resulting in a cache miss will cause the cache line to be filled with the newly read value plus values from the surrounding memory locations. However, if the old contents of the cache line have been written to, the old contents must first be written back to memory, resulting in significant longer timings for the read. For this reason, we expect parameter N to depend slightly on both the particular dictionary methods and the combination of dictionary operations. This means that R and T are the only parameters not dependent on the methods used.

Method	N	O
Cuckoo	71	142
Two-Way	66	157
Chained	79	78
Linear	88	89
Average	76	-

Table 4.3: Estimated parameters according to the model for insertion.

Using the timings for insert from Figure 4.4 and the average number of cache misses observed in Figure 4.6, we estimated N and O for insertion for the four

hashing schemes. As mentioned, we believe the slight rise in time for the largest sets in the tests of Figure 4.4 to be caused by other non-cache related factors. So since the model is only valid for $T \geq 2^{16}$, the two parameters were estimated for time timings with $2^{16} \leq T \leq 2^{23}$. The results are shown in Table 4.3. As can be seen from the table, the cost of a cache miss varies slightly from method to method. The largest deviation from the average is about 15%.

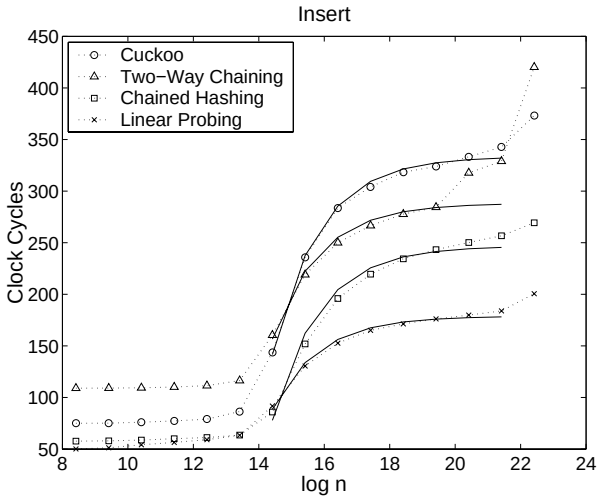


Figure 4.7: Model versus observed data.

To investigate the accuracy of our model we plotted in Figure 4.7 the estimated curves for insertion together with the observed curves used for estimating the parameters. As can be seen, the simple model explains the observed values quite nicely.

Having said this, we must admit that the values of N and O estimated for the schemes cannot be accounted for. In particular, it is clear that the true behavior of the schemes is more complicated than suggested by the model.

4.6 Chapter Summary

We have presented a new dictionary with worst case constant lookup time. It is simple to implement, and has average case performance comparable to the best previous dictionaries. Earlier schemes with worst case constant lookup time were more complicated to implement and had worse average case performance.

Despite having a reasonable space usage compared to other fast dictionaries the space requirements are too large to be of use in lossy compression. In Chapter 5 we develop a relaxation of Cuckoo Hashing that is readily turned into an efficient tool for lossy compression with fast random access.

Chapter 5

Lossy Dictionaries

5.1 Introduction

In the previous chapter it was demonstrated that efficient dictionaries with a reasonable space usage exist. However, the space usage is still too large to be of use in lossy compression. We now turn to a relaxation of dictionaries that results in a space efficient data structure that can be used for lossy compression with fast random access.

If one relaxes the requirements to the membership data structure, allowing it to store a slightly different key set than intended, new possibilities arise. A technique finding many applications in practice is *Bloom filtering* [7]. This technique allows space-efficient storage of a superset S' of the key set S, such that $S' \setminus S$ is no more than an ϵ fraction of the universe $U = \{0,1\}^w$. For $n \ll 2^w$, about $\log(1/\epsilon)$ bits per key in S are necessary and sufficient for this [13]. This is a significant savings compared to a membership data structure using $B = \log\binom{2^w}{n} \approx n\log(\frac{2^w e}{n})$ bits. Lookup of a key using Bloom filtering requires $O(\log(1/\epsilon))$ memory accesses and is thus relatively slow compared to other hashing schemes when ϵ is small. Also, Bloom filtering differs from most other hashing techniques in that it does not yield a solution to the dictionary problem.

5.1.1 This Chapter

In this chapter we introduce the concept of *lossy dictionaries* that can have not only false positives (like Bloom filters), but also false negatives. That is, some keys in S (with associated information) are thrown away when constructing the dictionary. For false positives there is no guarantee on the associated information returned. We let each key in S have a weight, and try to maximize the sum of weights of keys in the dictionary under a given space constraint.

We study this problem on a unit cost RAM, in the case where keys are machine words of w bits, examining a very simple and efficient data structure from a theoretical as well as an experimental point of view. Experimentally, we find that our data structure has surprisingly good behavior with respect to keeping the keys of largest weight. The experimental results are partially explained by our theoretical considerations, under strong assumptions on the hash functions involved. Specifically, we assume that in our RAM model, for

a number of random functions, arbitrary function values can be returned in constant time by an oracle. We also show that our data structure is nearly optimal with respect to space usage.

5.1.2 Applications

Recently, interest in lossy (volume) data compression with fast random access to decoded data has arisen [37, 46, 33, 71, 72, 4]. In Chapter 8 we show that lossy dictionaries are well suited for this purpose, providing lossy storage of the coefficients of wavelet transformed data. Compared to the previously best methods in the literature [37, 71, 4], our lossy dictionary based volumetric compression scheme performs about 50%-80% better in terms of compression ratio, while offering quite competitive random access times.

A *cache* can be seen as a dictionary that stores a small subset of a large key set, plus associated information. It is thus inherently lossy. A lossy dictionary allowed to discard a small fraction of a key set may thus in many cases be a quite acceptable implementation. If no wrong information is to be returned, we can allow no false positives. Our lossy dictionary seems best suited for applications where the cache only changes periodically, as for example in Web caching.

Web cache sharing [25] is a technique for implementing cooperating caches, for example Web proxies. When a request arrives at a proxy, it first checks whether it can answer the request. If not, it can forward the request to other proxies in the network. However, this increases traffic and is rather expensive. In cooperative caching each proxy keeps a summary of the content of all relevant proxies available to it. To reduce space requirements, this summary is stored with a small fraction of error using Bloom filtering. Often this reduces network traffic dramatically, since there is no more than a small chance that an expensive request forwarding is performed in vain. Lossy dictionaries with two-sided error could be used as a summary rather than a Bloom filter, since a small fraction of false negatives (cache misses) is tolerable.

In fact, the general idea of using in-memory summaries to reduce the number of expensive operations, such as I/O's, is well known in the database community. It dates at least back to [75], which uses Bloom filtering for efficient management of different versions of databases.

5.1.3 Related Work

Most previous work related to static dictionaries has considered the membership problem on a unit cost RAM with word size w. As mentioned in Chapter 4.1.1 the first membership data structure with worst case constant lookup time using $O(n)$ words of space was constructed by Fredman et al. [27]. For constant $\delta > 0$, the space usage is $O(B)$ when $2^w > n^{1+\delta}$, but in general the data structure may use $\Omega(Bw)$ bits of space. The space usage has been lowered to $B + o(B)$ bits by Brodnik and Munro [10]. The lower order term was subsequently improved to $o(n) + O(\log w)$ bits by Pagh [61]. The main concept used in [61] is that of a *quotient function* q of a hash function h, defined to be a function such that the mapping $x \mapsto (h(x), q(x))$ is injective.

The membership problem with false positives only was first considered by Bloom [7]. He described a technique, now known as *Bloom filtering*, where lookups return the conjunction of a number of bits from a bit vector. The locations of the bit probes are the values of a series of hash functions on the element to be looked up. Apart from Bloom filtering [7] presents a less space efficient data structure that is readily turned into a lossy dictionary with only false positives. However, the space usage of the derived lossy dictionary is not optimal. Carter et al. [13] provided a lower bound of $n \log(1/\epsilon)$ bits on the space needed to solve membership with an ϵ fraction false positives, for $n \ll 2^w$, and gave data structures with various lookup times matching or nearly matching this bound. Though none of their membership data structures have constant lookup time, such a data structure follows by plugging the abovementioned results on space optimal membership data structures [10, 61] into a general reduction provided in [13]. In fact, the dictionary of [61] can be easily modified to a lossy dictionary with false positives, thus also supporting associated information, using $O(n + \log w)$ bits more than the lower bound.

Another relaxation of the membership problem was recently considered by Buhrman et al. [11]. They store the set S exactly, but allow the lookup procedure to use randomization and to have some probability of error. For two-sided error δ they show that there exists a data structure of $O(nw/\delta^2)$ bits in which lookups can be done using just *one* bit probe. To do the same without false negatives it is shown that $O(n^2 w/\delta^2)$ bits suffice and that this is essentially optimal. Schemes using more bit probes and less space are also investigated. If one fixes the random bits of the lookup procedure appropriately, the result is a lossy dictionary with error δ. However, it is not clear how to efficiently guarantee the δ fraction of false positives in a reasonable model of computation, so this does not immediately give rise to a lossy dictionary.

5.2 Theory – Lossy Dictionaries

Consider a set S containing keys $x_1, \ldots, x_n$ with associated data or information $d_1, \ldots, d_n$ and positive weights $v_1, \ldots, v_n$. Suppose we are given an upper bound m on available space and an error parameter $\epsilon > 0$. The *lossy dictionary problem* for $\epsilon = 0$ is to store a subset of the keys in S and corresponding associated information in a data structure of m bits, trying to optimize the sum of weights of included keys. This corresponds to only allowing false negatives. For general ϵ we also allow the dictionary to contain $2^w \epsilon$ keys from the complement of S, corresponding to false positives. In this section we show the following theorem.

Theorem 5.1 *Let a sequence of keys $x_1, \ldots, x_n \in \{0,1\}^w$, associated information $d_1, \ldots, d_n \in \{0,1\}^l$, and weights $v_1 \geq \cdots \geq v_n > 0$ be given. Let $r > 0$ be an even integer, and $b \geq 0$ an integer. Suppose we have oracle access to random functions $h_1, h_2 : \{0,1\}^w \to \{1, \ldots, r/2\}$ and corresponding quotient functions $q_1, q_2 : \{0,1\}^w \to \{0,1\}^s \setminus 0^s$. There is a lossy dictionary with the following properties:*

1. *The space usage is $r(s-b+l)$ bits (two tables with $r/2$ cells of $s-b+l$ bits).*
2. *The fraction of false positives is bounded by $\epsilon \leq (2^b-1)r/2^w$.*
3. *The expected weight of the keys in the set stored is $\sum_{i=1}^{n} p_{r,i}\, v_i$ where*

$$p_{r,i} \geq \begin{cases} 1 - 52\, r^{-1}/(\frac{r}{i} - 2), & \text{for } i < r/2 \\ 2\,(1-2/r)^{i-1} - (1-2/r)^{2(i-1)}, & \text{for } i \geq r/2 \end{cases}$$

   is the probability that x_i is included in the set (which is independent of v_i).
4. *Lookups are done using at most two (independent) accesses to the tables.*
5. *The construction time is $O(n \log^* n + rl/w)$.*

As discussed in Section 5.2.1 there exist quotient functions for $s = w - \log(r/2) + O(1)$ if the hash functions map approximately the same number of elements to each value in $\{1, \ldots, r/2\}$. The inequality in item 2 is satisfied for $b = \lfloor \log(2^w \epsilon/r + 1) \rfloor$, so for $s = w - \log r + O(1)$ an ϵ fraction of false positives can be achieved using space $r\,(\log(\frac{1}{\epsilon + r/2^w}) + l + O(1))$. As can be seen from item 3, almost all of the keys $\{x_1, \ldots, x_{r/2}\}$ are expected to be included in the set represented by the lossy dictionary. For $i \geq r/2$ our bound on $p_{i,r}$ is shown in Figure 5.6 of Section 5.3, together with experimentally observed probabilities. If $n \geq r$ and r is large enough it can be shown by integration that, in the expected sense, more than 70% of the keys from $\{x_1, \ldots, x_r\}$ are included in the set (our experiments indicate 84%). We show in Section 5.2.5 that the amount of space that we use to achieve this is within a small constant factor of optimal.

Note that by setting $b = 0$ we obtain a lossy dictionary with no false positives. Another point is that given a desired maximum space usage m and false positive fraction ϵ, the largest possible size r of the tables can be chosen efficiently. Assume, for example, that we have quotient function with range $\lceil \log(2^{w+1}/r) \rceil$ and consider the case $b = 0$. The memory usage is $r(\lceil \log(2^{w+1}/r) \rceil + l)$. Whenever r is doubled, the number of bits per cell becomes one less. This means that the memory usage increases piecewise linearly in r, with jumps when r is a power of two. By setting $r = 2^i$ for $i = 1, 2, 3, \ldots$ we find the i for which m is first exceeded. The correct value of r can now easily be found in the interval $2^{i-1} < r < 2^i$. For general b this becomes more complicated, as we need to investigate more intervals, but finding r is still implementable in $O(\log m)$ time.

5.2.1 Preliminaries

The starting point for the design of our data structure is the CUCKOO HASHING scheme described in Chapter 4. In CUCKOO HASHING, two hash tables T_1 and T_2 are used together with two hash functions $h_1, h_2 : \{0,1\}^w \to \{1, \ldots, r/2\}$, where r denotes the combined size of the hash tables, assumed to be even. A

key $x \in S$ is stored in either cell $h_1(x)$ of T_1 or cell $h_2(x)$ of T_2. It was shown, in [62] that if $r \geq (2+\rho)\,n$, for $\rho > 0$, and h_1, h_2 are random functions, there exists a way of arranging the keys in the tables according to the hash functions with probability at least $1 - \frac{52}{\rho r}$. For small ρ this gives a dictionary utilizing about 50% of the hash table cells. The arrangement of keys was shown to be computable in expected linear time.

Another central concept is that of quotient functions. Recall that a quotient function q of a hash function h is a function such that the mapping $x \mapsto (h(x), q(x))$ is injective [61]. When storing a key x in cell $h(x)$ of a hash table, it is sufficient to store $q(x)$ to uniquely identify x among all other elements hashing to $h(x)$. To mark empty cells one needs a bit string not mapped to by the quotient function, e.g., 0^s for the quotient functions of Theorem 5.1. The idea of using quotient functions is that storing $q(x)$ may require fewer bits than storing x itself. If a fraction $O(1/r)$ of all possible keys hashes to each of r hash table cells, there is a quotient function whose function values can be stored in $w - \log r + O(1)$ bits.

Example We consider the hash function family from [20] mapping from $\{0,1\}^w$ to $\{0,1\}^t$, i.e., with $r = 2^t$. It contains functions of the form $h_a(x) = (ax \mod 2^w) \operatorname{div} 2^{w-t}$ for a odd and $0 < a < 2^w$. Letting bit masks and shifts replace modulo and division, these hash functions can be evaluated very efficiently. A corresponding family of quotient functions is given by $q_a(x) = (ax \mod 2^w) \bmod 2^{w-t}$, whose function values can be stored in $w - \log r$ bits.

The idea behind our lossy dictionary, compared to CUCKOO HASHING described in Chapter 4, is to try to fill the hash tables almost completely, working with key sets of size similar to or larger than r. Each key has two hash table cells to which it can be matched.

Thus, given a pair of hash functions, the problem of finding a maximum weight subset of S that can be arranged into the hash tables is a maximum weight matching problem that can be solved in polynomial time, see e.g. [16]. In Section 5.2.3 we will present an algorithm that finds such an optimal solution in time $O(n \log^* n)$, exploiting structural properties of our data structure. The term $O(rl/w)$ in the time bound of Theorem 5.1 is the time needed to copy associated information to the tables. Assume for now that we know which keys are to be represented in which hash table cells.

5.2.2 Our Data Structure

For $b = 0$ we simply store quotient function values in nonempty hash table cells and the value 0^s in empty hash table cells, using s bits per cell, as shown in Figure 5.1. For general b, we store only the first $s - b$ bits. Observe that no more than 2^b keys with the same hash function value can share the first $s - b$ bits of the quotient function value. This means that there are at most $2^b - 1$ false positives for each nonempty cell. Since 0^s is not in the range, this is also true for empty cells. In addition to the $s - b$ bits, we use l bits per cell to store associated information.

We now proceed to fill in the remaining details on items 3 and 5 of Theorem 5.1.

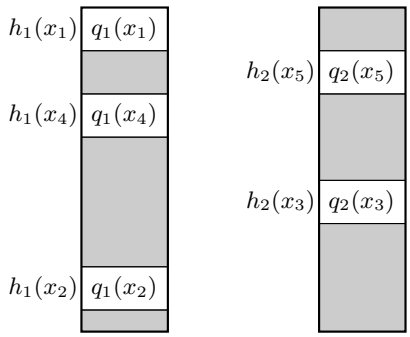


Figure 5.1: Data structure.

5.2.3 Construction Algorithm

Recall that the task of constructing our data structure boils down to finding the largest weight arrangement of keys in the tables. Given hash functions h_1 and h_2 and a key set K, we define the bipartite graph $G(K)$ with vertex set $\{1,2\} \times \{1,\ldots,r/2\}$, corresponding in a natural way to hash table cells, and the multiset of edges $\{\{(1,h_1(x)),(2,h_2(x))\} \mid x \in K\}$, corresponding to keys. Note that there may be parallel edges if several keys have the same pair of hash function values. We will use the terms keys/edges and cells/vertices synonymously. A connected component of $G(K)$ is defined to be *saturated* if the number of edges is greater than or equal to the number of vertices, i.e., if it is not a tree. We have the following characterization of the key sets that can be placed in the tables according to the given hash functions.

Lemma 5.1 *The key set K can be placed in the tables if and only if each connected component of $G(K)$ is a tree, plus possibly an extra edge.*

Proof. By Hall's theorem, K can be placed in the tables if and only if every subset $K' \subseteq K$ satisfies $|h_1(K')| + |h_2(K')| \geq |K'|$. This is true if and only if every subset K' of edges in $G(K)$ covers at least $|K'|$ vertices. Since it is equivalent to quantify only over subsets of edges within a connected component, the lemma follows. □

By an *optimal solution* for a key set K we will understand a maximum weight subset of K that can be placed in the tables.

Lemma 5.2 *There is an optimal solution for $\{x_1,\ldots,x_i\}$ including key x_i if and only if for any optimal solution K' for $\{x_1,\ldots,x_{i-1}\}$, the set $K' \cup \{x_i\}$ can be placed in the tables.*

Proof. If $K' \cup \{x_i\}$ can be placed in the tables for some solution K' optimal for $\{x_1,\ldots,x_{i-1}\}$, then $K' \cup \{x_i\}$ must be optimal for $\{x_1,\ldots,x_i\}$.

On the other hand, suppose that for some $K \subseteq \{x_1,\ldots,x_{i-1}\}$, the key set $K \cup \{x_i\}$ can be placed in the tables and has optimal weight, and let K' be an optimal solution for $\{x_1,\ldots,x_{i-1}\}$. Consider the connected components of

$(1, h_1(x_i))$ and $(2, h_2(x_i))$ in $G(K)$. By Lemma 5.1 and since $K \cup \{x_i\}$ can be placed in the tables, at least one of the (possibly identical) connected components must be a tree, without loss of generality assume it is the component of $(1, h_1(x_i))$. Since $K \cup \{x_i\}$ is optimal, the connected component of $(1, h_1(x_i))$ in $G(\{x_1, \ldots, x_{i-1}\})$ must also be a tree. (If there was a cycle, a key of higher weight could be substituted for x_i, contradicting the optimality of $K \cup \{x_i\}$.) In particular, the connected component of $(1, h_1(x_i))$ in $G(K')$ is a tree. Thus, by Lemma 5.1 the set $K' \cup \{x_i\}$ can be placed in the tables. □

The lemma implies that the following greedy algorithm finds an optimal key set S' given keys sorted according to nonincreasing weight.

1. Initialize a union-find data structure for the cells of the hash tables.
2. For each equivalence class, set a "saturated" flag to `false`.
3. For $i = 1, \ldots, n$:
   (a) Find the equivalence classes c_1 of cell $h_1(x_i)$ in T_1, and c_2 of cell $h_2(x_i)$ in T_2.
   (b) If c_1 or c_2 is not saturated:
      i. Include x_i in the solution.
      ii. Join c_1 and c_2 to form an equivalence class c.
      iii. Set the saturated flag of c if $c_1 = c_2$ or if the saturated flag is set for c_1 or c_2.

In the loop, equivalence classes correspond to the connected components of the graph $G(\{x_1, \ldots, x_{i-1}\})$. There is a simple implementation of a union-find data structure for which operations take $O(\log^* n)$ amortized time; see [81] which actually gives an even better time bound. Figures 5.2 to 5.5 show the four possible cases in step 3b of the algorithm.

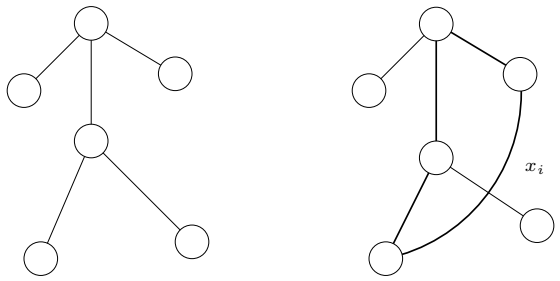


Figure 5.2: The case where $c_1 = c_2$ and the component is nonsaturated. The component becomes saturated.

What remains is arranging the optimal key set S' in the tables. Consider a vertex in $G(S')$ of degree one. It is clear that there must be an arrangement such that the corresponding cell contains the key of the incident edge. Thus, one can iteratively handle edges incident to vertices of degree one and (conceptually) delete them. As we remove the same number of edges and vertices from each

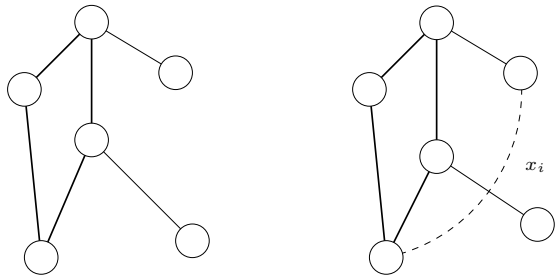


Figure 5.3: The case where $c_1 = c_2$ and the component is saturated. The new element is not included.

Figure 5.4: The case with one saturated and one nonsaturated component. The new component becomes saturated.

Figure 5.5: The case with two saturated components. The new element is not included.

connected component, the remaining graph consists of connected components with no more edges than vertices and no vertices of degree one, i.e., cycles. The arrangement of edges in a cycle follows as soon as one key has been put (arbitrarily) into one of the tables. The above steps are easily implemented to run in linear time. This establishes item 5 of Theorem 5.1.

5.2.4 Quality of Solution

We now turn to the problem of estimating the quality of the solution. Note that the optimal key set returned by our algorithm does not depend on the actual weights, but only on the sequence of hash function values. Thus, the expected weight of our optimal solution is $\sum_{i=1}^{n} p_{r,i}\, v_i$, where $p_{r,i}$ is the probability that the ith key is included in the returned optimal set of keys, which is independent of the weights.

Our algorithm includes all keys $\{k_1, \ldots, k_i\}$ in the optimal solution returned if they can all be accommodated under the given hash functions. Using the result of [62] mentioned in Section 5.2.1 on $\{k_1, \ldots, k_i\}$ with $\delta = r/i - 2$, we have that for $i < r/2$ this happens with probability at least $1 - 52\, r^{-1}/(r/i - 2)$. In particular, $p_{r,i}$ is at least this big.

For $i \geq r/2$ we derive a lower bound on $p_{r,i}$ as follows. If one of the

vertices $(1, h_1(k_i))$ and $(2, h_2(k_i))$ in $G(\{k_1, \ldots, k_{i-1}\})$ is isolated, then k_i is in the optimal solution returned. The randomness assumption on our hash functions implies that $G(\{k_1, \ldots, k_{i-1}\})$ has $i-1$ randomly and independently chosen edges. Thus, we have the bound $p_{r,i} \geq 1 - (1 - (1 - 2/r)^{i-1}))^2 = 2(1 - 2/r)^{i-1} - (1 - 2/r)^{2(i-1)} \approx 2e^{-i/r} - e^{-2i/r}$. This establishes item 3 of Theorem 5.1 and concludes the proof.

5.2.5 A Lower Bound

This section gives a lower bound on the amount of memory needed by a lossy dictionary with an ϵ fraction of false positives and γn false negatives. Our proof technique is similar to that used for the lower bound in [13] for the case $\gamma = 0$.

Proposition 5.1 *For $0 < \epsilon < 1/2$ and $0 < \gamma < 1$, a lossy dictionary representing a set $S \subseteq \{0,1\}^w$ of n keys, $120 < n \leq 2^{w-1}$, with at most $2^w\epsilon$ false positives and at most γn false negatives must use space of at least*

$$(1-\gamma)\, n \log\left(\frac{1}{\epsilon + n/2^w}\right) - \tfrac{5}{2}n \text{ bits.}$$

Proof. We can assume without loss of generality that γn is integer (this only gives a stronger space lower bound), and that $2^w\epsilon$ is integer. Consider the set of all data structures used for the various subsets of n elements from $\{0,1\}^w$. Any of these data structures must represent a set of at most $2^w\epsilon + n$ keys, in order to meet the requirement on the number of false positives. Thus, the number of n-element sets having up to γn keys outside the set represented by a given data structure is at most $\sum_{i=0}^{\gamma n} \binom{2^w\epsilon+n}{n-i}\binom{2^w}{i}$. Since $\epsilon < 1/2$ and $n \leq 2^{w-1}$ we have $2^w\epsilon + n \leq 2^w$, and so the largest term in the summation is $\binom{2^w\epsilon+n}{n-\gamma n}\binom{2^w}{\gamma n}$. Thus we have the upper bound $n\binom{2^w\epsilon+n}{n-\gamma n}\binom{2^w}{\gamma n}$.

We will use the inequalities $(\frac{a}{b})^b \leq \binom{a}{b} < (\frac{ae}{b})^b$, see e.g. [43, Proposition 1.3]. By the upper bound on the number of sets representable by each data structure, we need, in order to represent all $\binom{2^w}{n}$ key sets, space at least

$$\begin{aligned}
&\log\binom{2^w}{n} - \log\left(n\binom{2^w\epsilon+n}{(1-\gamma)n}\binom{2^w}{\gamma n}\right) \\
&\quad\geq \log\left(\frac{2^w}{n}\right)^n - \log\left(n\left(\frac{(2^w\epsilon+n)e}{(1-\gamma)n}\right)^{(1-\gamma)n}\left(\frac{2^w e}{\gamma n}\right)^{\gamma n}\right) \\
&\quad= n\log\left(\frac{2^w}{n}\frac{(1-\gamma)n}{(2^w\epsilon+n)e}\right) - \gamma n\log\left(\frac{(1-\gamma)n}{(2^w\epsilon+n)e}\frac{2^w e}{\gamma n}\right) - \log n \\
&\quad= n\log\left(\frac{(1-\gamma)/e}{\epsilon+n/2^w}\right) - \gamma n\log\left(\frac{1-\gamma}{\gamma(\epsilon+n/2^w)}\right) - \log n \\
&\quad= (1-\gamma)\, n\log\left(\frac{1}{\epsilon+n/2^w}\right) - (H(\gamma)+\log e)\, n - \log n
\end{aligned}$$

where $H(\gamma) = -\gamma\log\gamma - (1-\gamma)\log(1-\gamma) \leq 1$ is the binary entropy function. For $n > 120$ the sum of the last two terms is smaller than $\frac{5}{2}n$. □

In the discussion following Theorem 5.1 we noted that if there are quotient functions with optimal range, the space usage of our scheme is $n \log(\frac{1}{\epsilon+n/2^w}) + O(n)$ when tables of combined size n are used. The expected fraction γ of false negatives is less than 3/10 by Theorem 5.1. This means that our data structure uses within $O(n)$ bits of 10/7 times the lower bound. The experiments described in Section 5.3 indicate that the true factor is less than 6/5.

5.2.6 Using More Tables

We now briefly look at a generalization of the two-table scheme to schemes with more tables. Unfortunately the algorithm described in Section 5.2.3 does not seem to generalize to more than two tables. An optimal solution can again be found using maximum weight matching, but the time complexity of this solution is not attractive. Instead we can use a variant of the cuckoo scheme described in Chapter 4, greedily attempting to insert keys in order $x_1, \ldots, x_n$.

For two tables an insertion attempt for x_i works as follows: We store x_i in cell $h_1(x_i)$ of T_1 pushing the previous occupant, if any, away and thus making it nestless. If cell $h_1(x_i)$ was free we are done. Otherwise we insert the new nestless element in T_2, possibly pushing out another element. This continues until we either find a free cell or loop around unable to find a free cell, in which case x_i is discarded. It follows from Chapter 4 and the analysis in Section 5.2.3 that this algorithm finds an optimal solution, though, not as efficiently as the algorithm given in Section 5.2.2. When using three or more tables it is not obvious in which of the tables one should attempt placing the nestless key. One heuristic that works well is to simply pick one of the two possible tables at random. It is interesting to compare this heuristic to a random walk on an expander graph, which will provably cross any large subset of the vertices with high probability.

The main drawback of using three tables is, of course, that another memory probe is needed for lookups. Furthermore, as the range of the hash functions must be smaller than when using two tables, the smallest possible range of quotient functions is larger, so more space may be needed for each cell.

5.3 Experiments

An important performance parameter of our lossy dictionaries is the ability to store many keys with high weight. We tested this ability for lossy dictionaries using two and three tables. For comparison, we also tested the simple one-table scheme that stores in each cell the key of greatest weight hashing to it. The tests were done using truly random hash function values, obtained from a high quality collection of random bits freely available on the Internet [54]. Figure 5.6 shows experimentally determined values of $p_{r,\alpha r}$, the probability that the key with index $i = \alpha r$ is stored in the dictionary, determined from 10^4 trials. For the experiments with one and two tables we used table size $r = 2048$ while for the experiment with three tables we used $r = 1536$. We also tried various other table sizes, but the graphs were almost indistinguishable from the ones

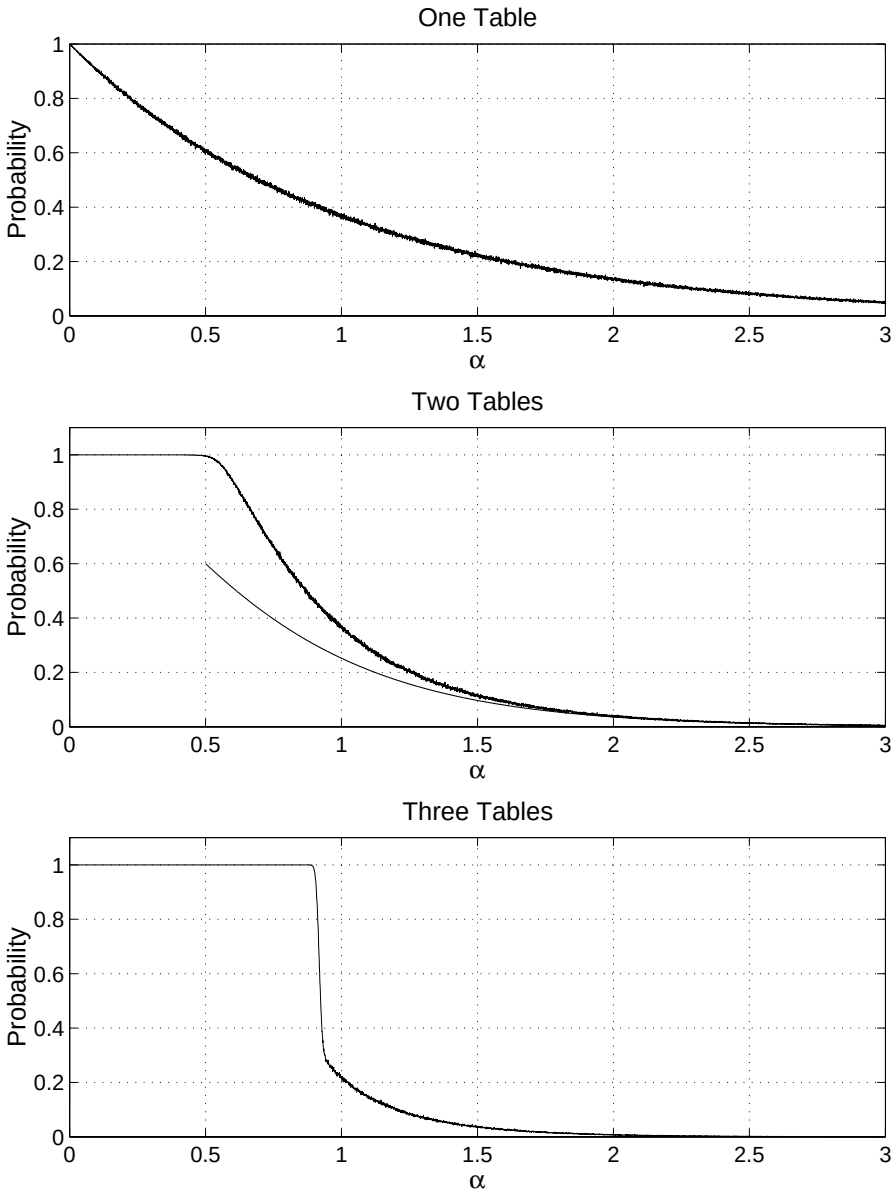


Figure 5.6: The observed probability that the element with (αr)th highest weight is stored when using one, two, and three tables. For two tables our lower bound is shown.

shown. From Figure 5.6 we see the significant improvement of moving from one to more tables. As predicted, nearly all of the $r/2$ heaviest keys are stored when using two tables. For three tables this number increases to about $.88r$. Of the r heaviest keys, about 84% are stored when using two tables, and 95% are stored when using three tables.

Apart from asymptotically vanishing differences around the point where the curves start falling from 1, the graphs of Figure 5.6 seem independent of r. For two tables the observed value of $p_{r,\alpha r}$ for $\alpha > 1/2$ is approximately $3.5/9.6^{\alpha}$

and for three tables it is approximately $8/33^{\alpha}$ for $\alpha > 0.95$.

The gap to the two-table lower bound of Theorem 5.1 can be explained by the fact that this lower bound considers only two cells of the hash tables, whereas opportunities for storing keys may appear when considering more cells.

5.3.1 Application

To give a flavor of the practicality of our lossy dictionary we turn to the real world example of lossy image compression using wavelets. In Chapter 8 a method for compressing volumetric data with fast random access is described in detail. Today most state-of-the-art image coders, such as JPEG2000 [41], are based on wavelets. As discussed in Chapter 2 the wavelet transform has the ability to efficiently approximate nonlinear and nonstationary signals with coefficients whose magnitudes, in sorted order, decay rapidly towards zero. This is illustrated in Figure 5.7. The figure shows the sorted magnitudes of the wavelet coefficients for the Lena image, a standard benchmark in image processing illustrated in Figure 5.8, computed using Daubechies second order wavelets. Thresholding the wavelet coefficients by a small threshold, i.e., setting small valued coefficients to zero, introduces only a small Mean Square Error while leading to a sparse representation that can be exploited for compression purposes. The main idea of most wavelet based compression schemes is to keep the value and position of the r coefficients of largest magnitude. To this end many advanced schemes, such as zerotree coding, have been developed. None of these schemes support access to a single pixel without decoding significant portions of the image.

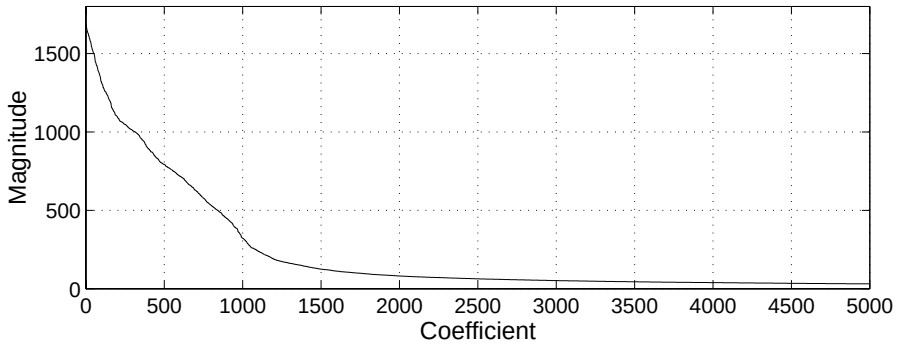


Figure 5.7: Largest 5000 magnitudes of 67615 wavelet coefficients of the Lena image.

Recently, interest in fast random access to decoded data, accessing only a few wavelet coefficients, has arisen. A survey of methods dealing with this problem is given in Chapter 6. In this section, however, we use the Lena image to give a flavor of the usefulness of lossy dictionaries on real world data and refer to Chapter 8 for a much more elaborate example. We store the coefficients of Figure 5.7 in a two-table lossy dictionary of total table size $r = 2^{11}$, using a simple family of hash functions. Specifically, we use hash functions of the form

$$h(x) = ((a_2x^2 + a_1a_2x + a_0) \bmod p) \bmod r/2,$$

Figure 5.8: The Lena image. A standard image benchmark in compression.

where p is a prime larger than any key, $0 < a_0, a_1, a_2 < p$ and a_1 is even. A corresponding quotient function is

$$q(x) = 2(((a_2x^2 + a_1a_2x + a_0) \bmod p) \operatorname{div} r/2) + x \bmod 2 \ .$$

Again, 10^4 iterations were made, selecting random functions from the above family using C's `rand` function. The graph of $p_{r,\alpha r}$ is indistinguishable from that in Figure 5.6. For our application, we obtain an MSE of 200, which is 27% more than the MSE when storing the r coefficients of largest magnitude. This difference would be difficult at best to detect in the reconstructed image. The previously mentioned family of [20] had somewhat worse performance. Using three tables reduces the increase in MSE to a mere 1%.

5.4 Chapter Summary

We have introduced the concept of lossy dictionaries and presented a simple and efficient data structure implementing a lossy dictionary. Our data structure combines very efficient lookups and near-optimal space utilization, and thus seems a promising alternative to previously known data structures when a small percentage of false negatives is tolerable, such as the examples mentioned in Section 5.1.2.

In Chapter 8 we show that our lossy dictionary can be used as an effective tool for volumetric compression with fast random access.

Chapter 6

Volumetric Compression with Fast Random Access

We will consider volumetric or volume data as discrete collections of scalar or vector values sampled over a uniform grid in n-dimensional space ($n \geq 3$). Volumetric data sets occur naturally in areas such as medical imaging and scientific visualization. Volumes produced by medical scanners such as CT, MR or PET are examples of three-dimensional data. Other examples are the output of physical simulations, and concentric mosaics used in image based rendering [77]. Sampled light fields [50] or lumigraphs [31] are examples of four-dimensional volumes also used in image based rendering. Four-dimensional volumes also appear as time varying three-dimensional volumes in, for example, computational fluid dynamics [59, pp. 84-87, pp. 125-143]. In this dissertation we focus on three-dimensional data only.

The management and processing of such massive data sets present many challenges to developers and researchers. One problem is that these data sets are often too large to keep in internal memory in uncompressed form. For example, the Visible Man [86] CT scanned data set takes roughly 1 Gbyte of storage, and the most detailed anatomical photo data sets are orders of magnitude larger. Thus, we are faced with memory requirements far exceeding the typical memory available on ordinary PCs and workstations. Even when taking the rapid development of larger memory and storage capabilities into account. On the other hand, it is desirable and becoming increasingly important, for example in interactive visualization, that any part of the data set can be rapidly retrieved. This implicitly assumes that the data can be loaded into memory for efficient processing. A solution to these apparently conflicting goals is to consider using compressed representations. Needed are methods allowing the user to load a compressed version of the volume into a small amount of memory and enable him to access and visualize it as if the whole uncompressed volume was present. Such a compression scheme must necessarily allow fast random access to individual voxels, decoded from the compressed volume. As we point out in Section 6.1 lossless compression schemes only provide very low compression ratios. Therefore lossy techniques have to be considered. Since lossy compression removes information from the data, lossless compression has often been preferred in medical imaging. However, in [68] Perlmutter et al. demonstrate that

lossy compression algorithms can be designed such that diagnostic accuracy is preserved. They use an embedded wavelet coding scheme with compression ratios of up to 80:1 for digital mammograms.

When designing lossy volume compression methods, certain properties (inspired from [4]) are desirable and should be considered during design:

1. *Fast decoding for random access.* As described above, fast decoding is a necessity for use in real-time or interactive applications. Also, applications often access data in unpredictable ways.

2. *Good visual fidelity at high compression ratios.* This seems obvious but it requires techniques for exploiting data redundancies in all n dimensions.

3. *Scalable or multiresolution decoding* is a desirable property which allows applications to process data at different levels of detail. For example, it could allow a rendering algorithm to render the data in low resolution at interactive frame-rates. When the user is satisfied with the setup, the renderer switches to full resolution.

4. *Selective block-wise decoding.* Even though our motivation is fast random access to individual voxels, some applications access data locally. In such cases it is useful if the compressed data can be decoded block-wise efficiently.

5. *Online compression.* By online compression we mean that the encoder is able to compress the volume while it is being downloaded or transferred to the system by other means. This is important in cases where there is not enough space to temporally store the data before compressing it.

In Chapter 7 and Chapter 8 we will present two methods for volumetric compression with fast decoding of randomly selected voxels. Before this a survey of previous related work is given in Section 6.1. Chapter 7 presents the first method which we will refer to as `Method 1` while the second method, referred to as `Method 2`, is described in Chapter 8. A comparison between the two methods is given in Chapter 9.

6.1 Previous Work

Research in lossy compression has mainly focused on lossy compression of still images or time sequences of images such as movies. The aim of these methods is to obtain the best compression ratio while minimizing the distortion in the reconstructed images. Often this limits the random accessibility. The reason being that most compression schemes employ variable bitrate techniques such as Huffman (used in JPEG [39] and MPEG [40]) and Arithmetic coders [74], or differential encoders such as the Adaptive Differential Pulse Code coder [74]. On the other hand, such methods provide fast sequential decoding which is important in, for example, compression of images and video sequences.

However, techniques dealing with the issue of random access in volumetric data have been emerging. In [57, 58] Muraki introduced the idea of using wavelets to efficiently approximate three-dimensional data. The two-dimensional wavelet transform was extended to three dimensions and it was shown how to compute the wavelet coefficients. By setting small coefficients to zero, Muraki showed that the shape of volumetric objects could be described in a relatively small number of three-dimensional functions. The motivation for the work was to obtain shape descriptions to be used in, for example, object recognition. No results on storage savings were reported. The potential of wavelets to reduce storage is evident, though.

Motivated by the need for faster visualization, a method for both compressing and visualizing three-dimensional data based on vector quantization was given by Ning and Hesselink [60]. The volume is divided into blocks of small size and the voxels in each block are collected into vectors. The vectors are then quantized into a codebook. Rendering by parallel projection is accelerated by preshading the vectors in the codebook and reusing precomputed block projections. Since the accessing of a single voxel is reduced to a simple table lookup in the codebook, fast random access is supported. Compressing two volumes both of size $128 \times 128 \times 128$, a compression factor of 5 was obtained with blocking and contouring artifacts being reported.

Burt and Adelson proposed the Laplacian Pyramid [12] as a compact hierarchical image code. This technique was extended to three dimensions by Ghavamnia and Yang [29] and applied to volumetric data. Voxel values can be accessed randomly by traversing the pyramid structure on the fly. Since there is high computational overhead connected with the reconstruction, the authors suggest a cache structure to speed up reconstructions during ray casting. They achieve a compression factor of about 10 with the rendered images from the compressed volume being virtually indistinguishable from images rendered from the original data.

Several compression methods, both lossless and lossy, for compressing and transmitting Visible Human images were presented and compared by Thoma and Long [84]. Among the lossy schemes, which as expected outperformed the lossless ones in terms of compression ratio, the scheme based on wavelets performed best. The wavelet method suggested by the authors compresses the images individually and consists of three steps comprised of a wavelet transform followed by vector quantization with Huffman coding of the quantization indices. This makes it a traditional two-dimensional subband coder, and compression factors of 20, 40 and 60 were reported with almost no visible artifacts for a factor of 20. The coder does not allow for fast random access to individual voxels as it was mainly designed for storage and transmission purposes. Also, there is no exploitation of inter-pixel redundancies between adjacent slices.

Motivated by the need for efficient compression of medical data, several methods that exploit correlation in all three dimensions have been proposed in recent years [6, 2, 1, 51, 69, 89, 5, 49]. Three of these methods are lossless methods based on predictive coding [2, 49, 1] and provide compression ratios of about 2:1. A compression ratio of 2:1 does not allow large volumes to be loaded into main memory. The other methods are lossy and utilize a three-

dimensional wavelet transform. The most recent of these are due to Bilgin et al. [6]. The method is a straightforward extension of zerotree coding proposed by Shapiro [76] and produces an embedded bit-stream which allows progressive reconstruction of data, i.e., it is possible to decode a lossy version of the data by using any prefix of the bit-stream. If the complete bit-stream is available, lossless reconstruction is possible. In order to improve performance, the zerotree symbols are further compressed using a context based adaptive arithmetic coder. Compression ratios of up to 80 were reported, still with good fidelity. However, none of the methods support fast random access.

The methods mentioned above all have in common that they support either high compression ratios or fast random access, but not both. Recently, methods dealing with issues of both fast random access and high compression ratios have emerged. In [37, 38], Ihm and Park present an algorithm achieving compression ratios of up to 28 for CT slices of the Visible Man, still with good fidelity of the reconstructed slices for a ratio of about 15:1. The algorithm divides the volume into blocks and decomposes each block using a three-dimensional wavelet decomposition. The wavelet coefficients in each block are then adaptively thresholded, which produces a sparse representation. Each block is coded separately using a data structure that takes the spatial coherency of the wavelet coefficients into consideration. The method also supports selective block-wise decoding and online compression.

Very recently, Bajaj et al. [4] have described an extension of [38] to RGB color volumes. Again the volume is divided into blocks, and the wavelet transform is performed three times on each block, once for each color component. The three corresponding wavelet coefficients are then vector quantized. The quantization indices are encoded block-wise in a manner similar to the technique used in [38]. To speed up voxel reconstruction, they introduce what they call zerobit encoding, which essentially corresponds to using a significance map signaling if certain wavelet coefficients are zero, or if they have to be retrieved. This is similar to the significance map we describe in Chapter 8. Compared to an extension of [38] to color volumes (without zerobit encoding), Bajaj et al. reported a 10% to 15% increase in compression ratio and a speedup in voxel access of a factor of 2.5. Selective block-wise decoding time is improved by a factor of about 4 to 5.5. depending on the compression ratio.

Not focusing directly on fast random access, Grosso et al. describe in [33] a wavelet based compression method for volumetric data suited for volume rendering. Each subband of the wavelet transformed data is independently runlength encoded. The problem with runlength encoding is that it does not allow for efficient retrieval of a single coefficient. In order to accelerate rendering speed, a method that avoids starting from the beginning of the runlength sequence is suggested. For a triple (i, j, k) a lookup table indexed by (i, j) is generated. Each entry in the table points to a runlength encoding of the coefficients corresponding to k. If the renderer accesses the data in the k-direction, efficient rendering can be achieved. However, it is obvious that fast random access is not supported.

In [46] Kim and Shin present a similar approach for fast volume rendering. The main difference is the way coefficients are arranged into runs. They divide the volume into blocks that are coded separately. Each block is decomposed in a three-dimensional wavelet basis. The thresholded coefficients are then ordered according to the reconstruction order and run-length encoded. Compression ratios of up to 40:1 were reported with good visual fidelity at ratios of up to about 30:1. Using the Shear Warp rendering algorithm of Lacroute and Levoy [48], Kim and Shin demonstrated that their algorithm using the compressed volume is only roughly two times slower, compared to using the uncompressed volume. Because of the run-length encoding, the algorithm does not support fast random access to individual voxels. Selective block-wise decoding is supported, though no results are reported.

The three methods [38, 4, 46] described above all make use of the Haar wavelet, which is convenient because of its simplicity. Since the mentioned methods all divide the data into blocks it is not clear, because of boundary conditions, how they generalize to wavelets with more vanishing moments, i.e., using wavelets with longer filters.

We conclude this section by noting that many of the results on volumetric compression in the literature are difficult to compare to one another since there is little consensus as to which data sets to use in presenting results. However, in volumetric compression with fast random access it seems that the CT images of the Visible Man are becoming a standard benchmark.

Chapter 7

Coding with Motion Estimation and Blocking

As one of the three spatial dimensions can be considered as similar to time, three-dimensional compression can borrow good ideas from research in video compression. Inspired by video coding we propose a coding scheme for wavelet based compression of very large volume data with fast random access. The main ideas are to use motion estimation to link sequences of two-dimensional slices rather than to use three-dimensional coding directly and to utilize a data structure, supporting fast random access, for encoding the wavelet coefficients and their significance map using a recursively block indexing scheme to locate the significant coefficients. Compared to the methods that support fast random access described in Chapter 6.1 our algorithm supports much higher compression ratios (up to 60:1 depending on quality) with an acceptable loss in decoding speed.

7.1 General Overview of Coder

First we present a black-box overview of both the encoder and the decoder, depicted in Figure 7.1. Detailed explanation of each stage will be given afterwards. A new observation that makes our coding strategy significantly different from earlier methods is that even though the data is genuinely volumetric in nature, we are not confined to treating it as such. Instead we can treat it as a sequence of two-dimensional slices in position or time and draw on results developed in the area of video coding. A major issue in video coding is the removal or exploitation of temporal redundancy or correlation.

7.1.1 Volume Encoder

Figure 7.1 depicts the four basic stages of our encoder together with the corresponding stages of the decoder. When designing our new coding scheme we constantly have to consider the trade-off between compression rate, distortion and decoding speed. When coding the volume we assume that we have divided it into two-dimensional slices in the *z-direction*. The first stage of the encoder will then be the removal of correlation along this z-direction. We call this stage

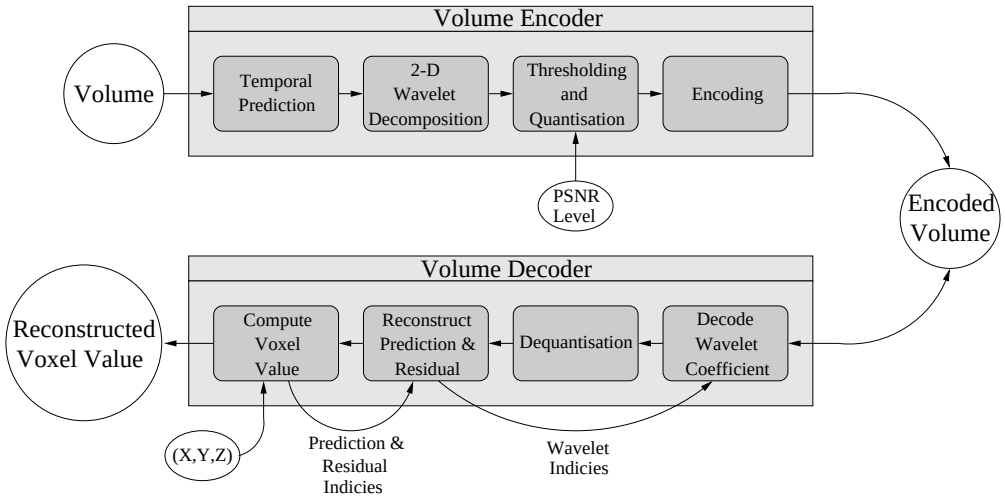


Figure 7.1: Overview of the encoder and the decoder.

temporal prediction. Ideally the next step should then according to [83], perform a three-dimensional wavelet decomposition to further remove correlations in both the spatial and temporal directions. Since a three-dimensional transform is computationally more expensive than a two-dimensional transform and under the assumption that the prediction stage has removed enough of the temporal correlation, we adopt a two-dimensional wavelet transform to handle the spatial redundancy as the next stage. The third stage, which is typical for a lossy subband coder, removes insignificant coefficients to make the representation even sparser. This step also quantizes the remaining coefficients restricting these to a small number of possibilities. Finally, in the last stage, the wavelet transformed data is encoded efficiently in a way that allows fast retrieval of the data needed to reconstruct individual voxels.

7.1.2 Volume Decoder

In general the decoder consists of the inverse stages of the encoder but in reverse order. However, there is a small but significant difference. Since the decoder acts on input from the user or another program some stages of the decoder have to communicate information back to other stages in order to retrieve the desired voxels from the encoded data. This is not necessary in the encoder since it encodes the whole volume at once. For example, the indices of the wavelet coefficients needed for reconstruction must be passed to the stage that extracted them from the compressed bit-stream.

7.2 Description of the Encoding Stages

7.2.1 Test Data

The data that we use for our experiments is the same as in [37] and was kindly made available by Professor Insung Ihm. The data set is a volume of size $512 \times 512 \times 512$ rebuilt from the fresh[1] CT slices of the Visible Man. Rebuilding the volume was necessary since the fresh CT slices have varying pixel sizes and spacings. Each voxel is represented as a 12 bit grayscale value in the interval [0,4095] and is stored in 16 bits resulting in a total volume size of 256 Mbytes. In the rest of the dissertation we will refer to this data set as the Visible Man data set.

7.2.2 Temporal Prediction

Many different methods for motion estimation and motion compensation have been reported and investigated in the literature. Among the most popular are block-matching algorithms [28, 82], parametric motion models [36, 67, 83], optical flow [82] and pel-recursive methods [82].

The scheme that we have chosen to adopt, depicted in Figure 7.2, is the simplest one possible, yet very effective. It corresponds to a best matching neighbor scheme. Every n-th slice, called an *I-slice*, is used as a reference slice

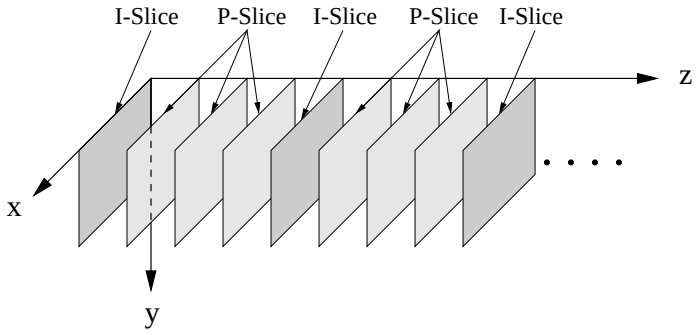


Figure 7.2: Ordering of I-slices and P-slices.

and is coded without temporal prediction. This corresponds to intra frames in MPEG and H.263 [40, 42]. The prediction of a slice in between two I-slices, called *P-slices*, is then chosen to be the neighboring I-slice giving the best match in terms of the Mean Square Error. Finally, the residual is constructed and sent to the next stage of the coder for further processing. The voxel values of the original data reside in the integer interval [0,4095] so the possible values of the residual lie in the interval [-4095,4095]. By using both backward and forward I-slices in the prediction, we effectively half the longest distance between an I-slice and a P-slice. Experiments have shown this to give significantly better compression results than just using forward prediction. The distance n between

[1] There exist both fresh and frozen CT and MRI images.

two I-slices is chosen to be $n = 2^k$ for faster processing. The issue of selecting a good value for k will be discussed in Section 7.4.

7.2.3 Wavelet Decomposition, Thresholding and Quantization

An important objective for our compression scheme is to code the volume with high accuracy in a minimum number of bits. Wavelets have proven to be very effective at achieving this goal. As mentioned in Chapter 2.2.1 wavelets have the ability to efficiently concentrate the energy of a signal in a few wavelet coefficients with large magnitude creating a compact or sparse representation and thus making it easier to code in a smaller amount of bits.

When implementing the wavelet transform we have to decide which wavelet to use and how many levels of decomposition to perform. Different wavelets correspond to filters of different length as described in Chapter 2.3.1. Table 7.1 shows how the choice of wavelet and the number of decomposition levels affect the number of wavelet coefficients that are used in the two-dimensional reconstruction.

		Filter length			
		2	4	6	8
Level	1	4	16	36	64
	2	7	31	71	127
	3	10	46	106	190
	4	13	61	121	253

Table 7.1: The influence of filter length and decomposition level on the number of coefficients needed for reconstruction in two dimensions.

Even though, the Haar wavelet does not perform so well in terms of quality as other wavelets it certainly allows for faster reconstruction, only being a two-tap filter, and thus it becomes our choice of wavelet. For the number of decomposition levels, we restrict ourselves to two levels, despite the fact that in wavelet based compression it is common to perform three or four levels. For two levels of decomposition more than 93% of all the coefficients are already decomposed into wavelet coefficients and according to Table 7.1 we only need 7 coefficients to reconstruct a voxel in an I-slice and 14 coefficients in a P-slice.

After applying the wavelet transform to each slice, all wavelet coefficients below a certain threshold are set to zero in order to make the representation even sparser. The threshold is determined such that the PSNR[2] does not drop below a user specified value, see Figure 7.1. According to Chapter 2.5 the coefficients from all the slices should be sorted and then in ascending order be set to zero until the desired PSNR is obtained. Sorting the wavelet coefficients is only practical if enough disk space is available to store all the coefficients. Considering the available harddisk space on modern computers this is feasible. However, in this chapter we will consider a compression algorithm where

[2] $\text{PSNR} = 10\log_{10}(\frac{\max(x_i^2)}{MSE})$, x_i being the samples of the original data.

only a few slices are necessary at a time. Because the algorithm reads data sequentially and only needs to buffer a small number of images it can be used online where slices are compressed as they are downloaded. Instead of using a global threshold computed by sorting all wavelet coefficients we use a different threshold for each slice. This of course is not globally optimal for the volume but it provides a reasonable solution to the online problem. When calculating the PSNR of each slice we use the global maximum of the whole volume,which is set to 2^b where b is the number of bits used to represent a voxel.

After thresholding we rescale the wavelet coefficients as mentioned in Example 2.4 and round the remaining coefficients to the nearest integer. Since few coefficients are remaining, rounding the coefficients is negligible with respect to the Mean Square Error. The rescaling insures that the coefficients now lie in the interval $[-4095, 4095]$, i.e., use (2.42) with division by 4 to see this.

7.2.4 Encoding Wavelet Coefficients – Data Structure

The last stage of the compression process is the encoding of the remaining wavelet coefficients. In addition, we also need to encode the positional information about the coefficients – this is often referred to as the significance map. Shapiro [76] showed that the bits needed to code the significance map are likely to consume a large part of the bit-budget with the situation becoming worse at low bitrates. In his paper Shapiro proposed a technique called zerotree coding for storing the significance map. Although effective, this method does not lend itself to fast retrieval of individual wavelet coefficients. The same problem is true for run-length coding, Huffman coding or arithmetic coding since they produce variable length codes [74].

We now propose a method for storing the significant coefficients and their significance map. The method is illustrated in Figure 7.3 and is designed to fit the preprocessed CT images of the Visible Man data set described in Section 7.2.1 but is easily extended to other volumes.

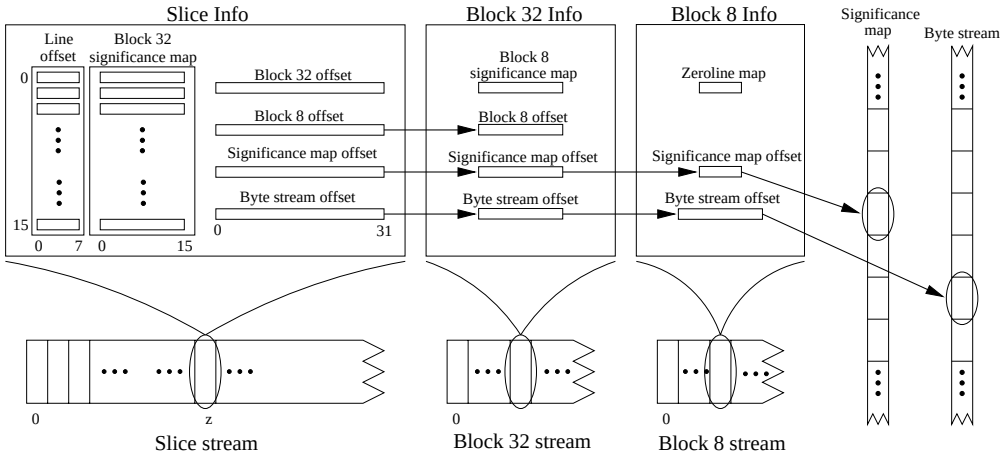


Figure 7.3: Data structure used for encoding significant wavelet coefficients.

Given that most of the wavelet coefficients have been set to zero and the

usually spatial coherence of a wavelet decomposed image it is likely that the zero coefficients appear in dense clusters. This hypothesis is backed by experiments such as the one illustrated in Figure 7.4, which depicts the significance map for an I-slice and a P-slice for the Visible Man data set.

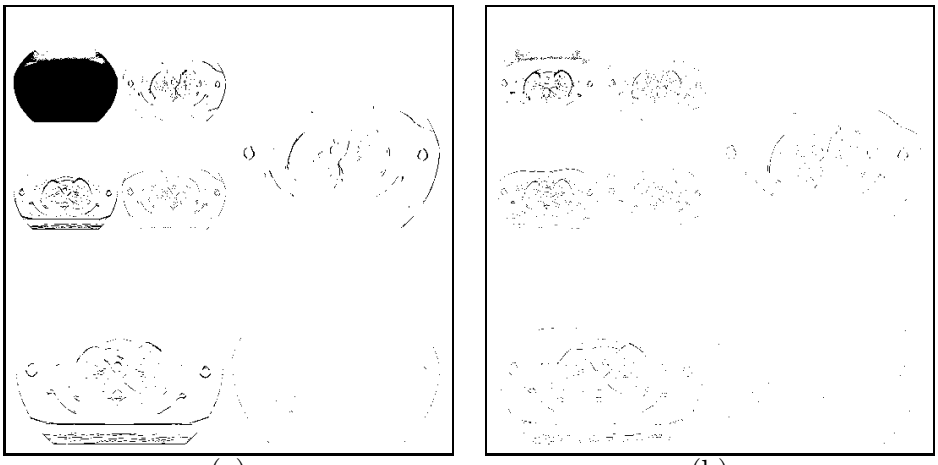


Figure 7.4: Significance map of thresholded wavelet decomposition with PSNR level 46. A black dot in the map indicates the position of a nonzero coefficient. (a) Slice no. 344 (I-slice) and (b) slice no. 343 (P-slice).

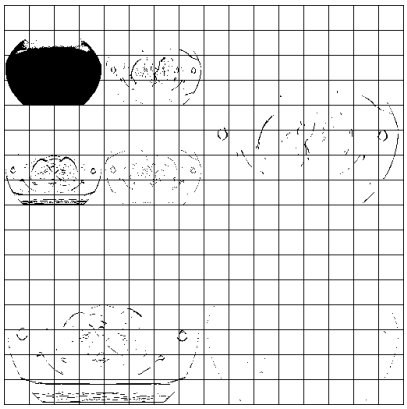


Figure 7.5: The significance map is initially divided into blocks. Note that many of the blocks are empty. Later nonempty blocks will be divided further.

The initial idea will be to split each slice into quadratic blocks of length b and then use a bitmap (*block b significance map*) to differentiate between the blocks containing at least one nonzero coefficient (*nonzero b block*) and the blocks having all coefficients equal to zero (*zero b block*). This is illustrated in Figure 7.5. Spending one bit on each block in the bitmap is optimal with respect to first order entropy if the block size b is chosen such that the ratio between zero blocks and nonzero blocks is one half. Table 7.2 shows the percentage

of nonzero blocks for different sized blocks and PSNR levels, with a fixed I-slice spacing of $n = 4$. For each PSNR level we have marked the values, in the table, closest to 50%. From the table we notice that $b = 32$ is the best choice.[3] The bitmaps (*block 32 significance map*) for each slice are collected

		PSNR Level				
		43	46	49	52	53
Block Size	4	5.5%	7.9%	10.6%	15.5%	22.9%
	8	10.8%	14.7%	18.4%	24.5%	32.6%
	16	19.8%	24.8%	29.3%	35.8%	**43.7%**
	32	32.8%	**38.4%**	**43.3%**	**50.0%**	58.2%
	64	**55.8%**	61.7%	66.0%	70.6%	78.0%

Table 7.2: Number of nonzero blocks in percent for different block sizes and PSNR levels.

into records called *slice info* records, together with other information which we explain shortly, and kept in an array. Retrieval of a slice info record is then a simple array lookup since each record has a fixed size. For each of the nonzero 32 blocks, further information needs to be stored and we allocate a new record (*block 32 info*) for each of them. These records are also stored in a stream. Each slice contains a variable number of nonzero 32 blocks, so in order to quickly find a given block 32 info record in the stream we add the following offsets to the slice info record. *Block 32 offset* holds the index in the block 32 stream of the first block 32 info record for that particular slice. The *line offset* array contains counts of all nonzero 32 blocks that precede a given line in the block 32 significance map. In order to find the position of a given block 32 record in the stream, we first compute where it is in the bitmap. Then we count the number of bits that precedes it in the respective bitmap-line. This count is then added to the block 32 offset together with the correct line offset and the result is used for lookup. Counting the number of bits in a bitmap line can be done efficiently by using a precomputed table with $2^{16} = 65536$ entries.

		PSNR Level				
		43	46	49	52	53
Block Size	4	16.7%	20.6%	24.5%	31.0%	39.3%
	8	33.1%	**38.1%**	**42.4%**	**48.9%**	**56.0%**
	16	**60.2%**	64.5%	67.6%	71.5%	75.1%

Table 7.3: Empirical probability of a block being nonzero given that it is contained in a nonzero block of size 32.

Following the same idea, the nonzero 32 blocks can be split into sub-blocks. Table 7.3 shows the empirical conditional probabilities of a block with size 4, 8 and 16 being nonzero given that it is contained in a nonzero block of size 32. It follows that a sub-block of size 8 is a good choice. The information about these

[3] b is a power of two for efficient processing.

sub-blocks is stored in records (*block 8 info*) and kept in a new stream (*block 8 stream*). Similar to the problem of addressing the 32 blocks we need offsets to quickly access a block 8 info record. To this end we add a *block 8 significance map* and a *block 8 offset* to the block 32 info record. There are at most $\frac{512^3}{8^2}$ nonzero sub-blocks of size 8 in the volume. To offset all of these we need at least 21 bits. We observe that there are at most $\frac{512^2}{8^2}$ nonzero blocks of size 8 in each slice so instead of using 32 bits in the block 32 info record[4] we divide the offset in two using 32 bits in the slice info record and only 16 bits in the block 32 record. That way the total overhead of storing all the offsets is reduced. Finally each nonzero 8 block is divided into lines keeping a bitmap (*zeroline map*) in the block 8 info record marking the lines which contain all zeros. For all other lines containing at least one nonzero coefficient we keep one byte as a *significance map*. This way we need between 0 and 8 bytes for the map, hence we store them in a stream (*significance map stream*) introducing a new offset, the *significance map offset*, which is similarly divided between the three info records for efficient storage. Table 7.4 shows that dividing the sub-blocks into lines is a good idea since about half of the lines in an 8 sub-block are zero. The

	PSNR Level				
	43	46	49	52	53
Zerolines	60.7%	57.2%	53.6%	47.6%	40.1%

Table 7.4: Empirical probability of a line in a nonzero sub-block of size 8 being all zero.

significance maps give the positions of the significant coefficients.

In the following, it will be explained how the coefficients are stored. As stated in Section 7.2.3 the wavelet coefficients have been scaled and rounded to the integer interval [-4095,4095]. Rather than using 13 bits to represent the values in this interval, we quantize the nonzero coefficients, as described below, to the interval [0,255] so they fit into one byte and we store them in a stream (*byte stream*) pointed to by offsets (*significance offset*) located in all three info records. Inspired from [37, 38] we observe that the coefficients within a size 8 sub-block are likely to be numerically close. So whenever the coefficients in a sub-block all belong to either the interval $[\theta, \theta + 127]$ or the interval $[-\theta - 128, -\theta]$, where θ is a two-byte offset, we code them by storing θ in the byte stream followed by a signed displacement (1 byte) for each coefficient. For all other 8×8 sub-blocks not satisfying this property, we quantize the coefficients in the sub-block using a uniform scalar quantizer given by

$$\tilde{C} = \left\lfloor \frac{A - B}{255} \times x + 0.5 \right\rfloor + B \ , \tag{7.1}$$

where A and B are 16 bit integer parameters to the quantizer, the byte x is the quantization value and $\tilde{C}$ is the reconstructed value. For each 8×8 sub-block

[4] Offsets are 8, 16 or 32 bits to ease programming.

G: A, B and x can be determined as

$$A = \max_{C \in G}(C) \quad , \quad B = \min_{C \in G}(C) \quad , \quad x = \left\lfloor 255 \times \frac{C - B}{A - B} + 0.5 \right\rfloor \quad . \tag{7.2}$$

We only quantize the significant coefficients. If there are both negative and positive coefficients in the same block quantization steps are unnecessarily spent spanning the interval $]-\tau, \tau[$ making (7.2) ineffective, see Figure 7.6.

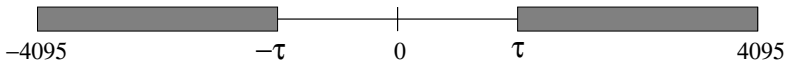


Figure 7.6: Situation after thresholding. Coefficients are either zero or in the shaded region.

We improve (7.2) by using

$$A = \max_{C \in G}(|C|) \quad , \quad B = \min_{C \in G}(|C|) \ , \tag{7.3}$$

$$C \geq 0 : x = \left\lfloor 127 \times \frac{C - B}{A - B} + 0.5 \right\rfloor \ , \ C < 0 : x = \left\lfloor 127 \times \frac{C + B}{A - B} - 1.5 \right\rfloor , \tag{7.4}$$

with reconstruction for $x \geq 0$ and $x < 0$ respectively being

$$\tilde{C} = \left\lfloor \frac{A - B}{127} \times x + 0.5 \right\rfloor + B \quad , \quad \tilde{C} = \left\lfloor \frac{A - B}{127} \times (x + 1) - 0.5 \right\rfloor - B \ . \tag{7.5}$$

After quantization A, B and the quantized coefficients are stored in the byte stream.

How a coefficient is stored can be coded in 1 bit and placed in the most significant bit (MSB) of the byte stream offset in the block 8 info record. This bit is free since there are only $32^2 = 1024$ coefficients in each 32 block and we use 16 bits for the offset. Similarly for all P-slices we use the MSB of the block 32 offset in the slice info record to indicate the direction of the prediction. Looking at the sizes of the offsets used throughout the data structure, it is quite easy to verify that they are large enough.

The design of the data structure has been based on the Visible Man data set. However, from Figure 7.3 it is obvious that the data structure can easily be modified to accommodate volumes of varying sizes. All that is needed to compress larger volumes is to make the offsets and tables in the Slice Info record larger. This can be done without sacrificing compression ratio since this record only constitute a very small fraction of the total bit-budget.

7.3 Analysis of Performance

In this section we analyze the work needed for decoding a single voxel. The reconstruction filter for the Haar wavelet is a two-tap filter. Performing a two level reconstruction we therefore need 7 wavelet coefficients (four for the

first level and three for the second). If we are decoding a voxel in a P-slice we must also decode the same number of coefficients for the I-slice resulting in the retrieval of 14 wavelet coefficients. Table 7.5 shows average values for the number of wavelet coefficients that must be retrieved and the number of additions performed for different I-slice spacings. It is observed that the amount of information that needs to be retrieved and the number of additions that must be performed are not critical with respect to the slice spacing.

			Weighted average for a spacing of				
	I-slice	P-slice	2	4	8	16	32
Coefficients	7	14	10.50	12.25	13.13	13.56	13.78
Additions	6	12	9.00	10.50	11.25	11.63	11.81

Table 7.5: Number of coefficients and additions needed for reconstruction of a voxel given different I-slice spacings.

For accessing a wavelet coefficient the approximate workload can be assessed. First we look at the algorithm for decoding the wavelet coefficient corresponding to the voxel at location (x, y, z).

function Wavelet_coefficient(x, y, z)
    S := Lookup_Slice_Info(z);
(1) **if** is_in_zero_block(x, y, **S**) **then return** 0;
    B32 := Lookup_Block32_Info(**S**, block 32 offset);
    Calculate relative index (x′, y′) in **B32**;
(2) **if** is_in_zero_block(x′, y′, **B32**) **then return** 0;
    B8 := Lookup_Block8_Info(**B32**, **S**, block 8 offset, bitmap);
    Calculate relative index (x″, y″) in **B8**;
(3) **if** is_zeroline(y″, **B8**) **then return** 0;
    M := significance_map(M, offsets);
(4) **if** bit(x″, M) = 0 **then return** 0;
    Access bytestream using offsets;
(5) **return** value;
end;

The division of each slice into blocks and lines was designed so the empirical probability of going from (1)→(2), (2)→(3), (3)→(4), or (4)→(5) in the reconstruction algorithm is roughly one half. Combining these probabilities with the fact that less than 6 percent of the wavelet coefficients are nonzero (see Section 7.4) we obtain the numbers in Table 7.6.

From this table we estimate that on average $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \approx 2$ lookups in the encoded data have to be performed in order to retrieve one wavelet coefficient. In addition, one lookup in the bitcount table is performed on average.

According to Table 7.5 roughly 13 wavelet coefficients must be extracted from the compressed data in order to reconstruct a single voxel. Observing

Stage	1	2	3	4	5
Probability	100%	50%	25%	12.5%	6%

Table 7.6: Empirical probability of reaching a certain stage in the algorithm.

from (2.43) that 4 values can be reconstructed from 4 wavelet coefficients this seems inefficient. Figure 7.7 shows how the wavelet coefficients on different levels are related to the reconstructed coefficients. If the voxels are accessed in some regular pattern, increased efficiency can be achieved by reconstructing all voxels in the 4×4 neighborhood reusing the extracted coefficients. Furthermore, the grouping of the first level detail coefficients ensures that the information about each group resides in the same Block 8 Info record resulting in fewer lookups in the data structure.

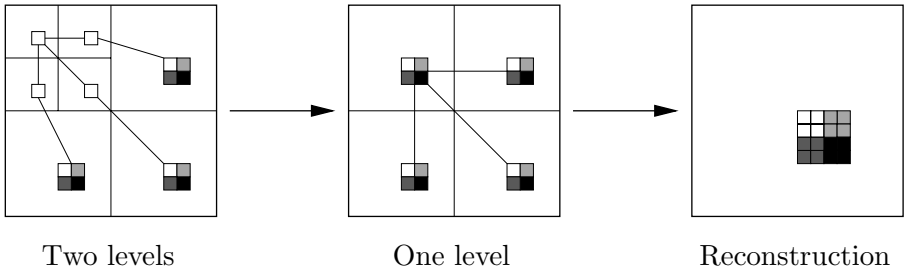


Figure 7.7: Relationship between average, detail and reconstructed coefficients.

7.4 Experiments

In this section we present results based on the Visible Man data set as described in Section 7.2.1. Before presenting the main results we first deal with the issue of selecting a k such that the distance $n = 2^k$ between two I-slices is best possible. Table 7.7 illustrates the size of the compressed volume for different choices of k with a desired PSNR level fixed at 46. According to the table selecting $k = 2$, resulting in I-slice spacings of 4, produces the best result. For PSNR levels in the range 43–56 we obtained similar results proving $k = 2$ to be a good choice.

Spacing $n = 2^k$	Original	2	4	8	16	32
Compressed size (Mb)	256	7.11	**6.01**	6.28	7.14	8.24

Table 7.7: Compressed size for different I-slice spacings. The desired PSNR level is 46.

7.4.1 Compression Ratio and Distortion

The main results of our coding scheme (`Method 1`) are presented in Table 7.8. We tested with desired PSNR levels of 43.0, 46.0, 48.03, 51.6, and 55.8 achieving compression ratios between 60.2:1 and 14.5:1. On average this is about a 50% reduction in size compared to the results in [37]. This can be seen from Figure 7.8 which shows the PSNR as a function of the compression ratio for our method and [37]. One of the main reasons for achieving an increased compression rate is that our method succeeds in setting more wavelet coefficients to zero for a given PSNR level, see Table 7.9. As mentioned in Chapter 6.1 an improvement to [37] has very recently been proposed in [4]. The reported improvement was a 10%-15% increase in compression ratio for the RGB color volume of the Visible Man. Even if a similar increase holds for the Visible Man data set used in our experiments, our method still performs best, rate-distortion wise.

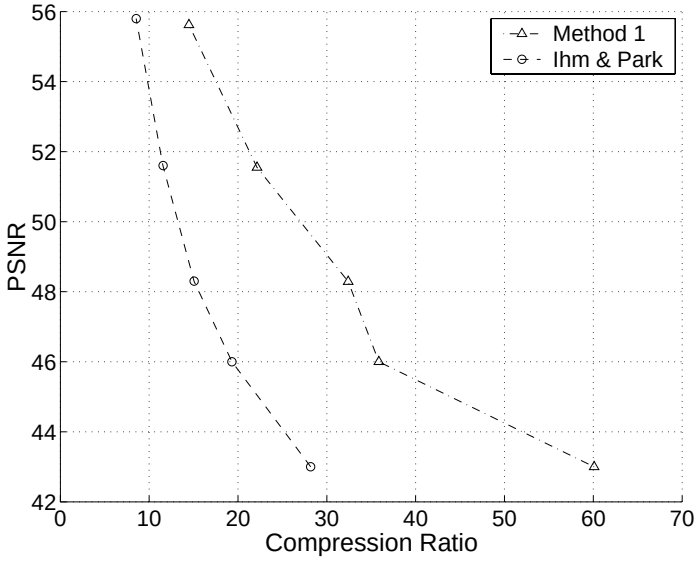


Figure 7.8: Rate-distortion curve for the Visible Man data set.

We note, in Table 7.8, that the actual PSNR is not equal to the one desired, but close. This is because thresholding stops before the desired PSNR level is exceeded. Also rounding and quantization of the coefficients introduce additional errors.

Figure 7.12 and Figure 7.13 at the end of this chapter show sample slices from the original volume and from compressed volumes at different compression ratios. It is observed that for the highest compression ratios some blockiness and loss of small level details occur. In all cases edges are preserved extremely well. We have also generated ray-cast rendered images from the original and compressed volumes. Figures 7.14 to 7.17 at the end of this chapter depict rendered images of skin and bone. To generate the images we used Volvis

		Desired lower bound on PNSR				
		43.0	46.0	48.3	51.6	55.8
Compression	Ratio	60.2:1	42.6:1	32.4:1	22.2:1	14.5:1
	Size (Mb)	4.26	6.01	7.90	11.56	17.68
Errors	Actual PSNR	43.00	46.00	48.29	51.55	55.62
	SNR	24.98	27.98	30.27	33.53	37.60
	MSE	839.59	421.39	248.83	117.34	45.92

Table 7.8: Results on compression ratio and quality.

	PNSR levels				
	43.0	46.0	48.3	51.6	55.8
Method 1	1.44%	2.14%	2.94%	4.62%	7.73%
Ihm & Park	2.78%	4.57%	6.32%	8.94%	13.30%

Table 7.9: The percent of nonzero coefficients after thresholding for different PSNR levels.

2.1 [88][5]. The images are essentially indistinguishable from the original except for the volumes with PSNR level of 43 and 46. For these two cases the introduced artifacts are not significantly disturbing the image and most of the details are well preserved.

7.4.2 Timing Results

To evaluate the time for accessing voxels from the compressed data, we measured the time it took to access 1,000,000 randomly selected voxels in the compressed volumes, using C's `rand()` function. These experiments were conducted on two different computers. The first computer used was a SGI Octane workstation with a 175 MHz R10000 CPU with a 32 Kbyte level 1 data cache and a 1 Mbyte level 2 cache. This machine is very similar (except maybe differences in the amount of installed memory and cache) to the one used in [37], so direct comparison is possible. Programs were compiled with the `cc` compiler version 7.3.1.1m, using optimization flags `-Ofast=IP30`. The second computer used was a PC with an 800 MHz Intel Pentium III processor with 16 Kbyte level 1 data cache and 256 Kb level 2 cache. On this machine programs were compiled with `gcc` version 2.95.2, using optimization flags `-O9 -fomit-frame-pointer -fno-rtti`. Both machines were equipped with 256 Mbytes of main memory.

Results are listed in Tables 7.10. As a reference, we also list in the tables the time it takes to access 1,000,000 randomly selected voxels from the uncompressed data. However, because the Visible Man data set could not fit into main memory on the two computers used we simulated accessing voxels in uncompressed data by allocating a piece of memory accessing entries of this piece at random. For the PC we noted that the time it took to randomly ac-

[5]The rendering system that we used only supports byte data, so the volumes were uniformly quantized from 12 bits to 8 bits before rendering.

CPU	Uncompressed	Compression Ratio				
		60.2:1	42.6:1	32.4:1	22.2:1	14.5:1
PC	0.50	3.68	4.04	4.36	4.87	5.54
SGI	1.65–3.03	8.07	9.04	9.81	10.93	13.02

Table 7.10: Voxel reconstruction times in seconds for accessing 1,000,000 randomly selected voxels for the Visible Man data set.

cess 1,000,000 elements varied very little with the size of the allocated piece of memory, as long as it was significantly larger than the cache. However, for the SGI we found that there is a linear dependence between the size of the allocated memory and the time it takes to access the random voxels. For a piece of memory varying between 20%–80% of the size of the Visible Man volume, the time it took to randomly access 1,000,000 voxels varied between 1.65 and 3.03 seconds.

From table 7.10, we observe that accessing random voxels from the Visible Man data set on a PC is about 7 to 11 times slower than accessing uncompressed data. Running on the SGI, it is more difficult to evaluate how much slower it is to access compressed data because of the mentioned variation when accessing uncompressed data. Figure 7.9 shows the time for accessing 1,000,000 random voxels in compressed data on the SGI as a function of the PSNR level for our method and the method by Ihm and Park [37].

From the figure, we observe that at low PSNR levels the two methods perform about the same, while for higher PSNR levels [37] is about 30% faster than our method. Considering the time it would take to access the voxels from a harddisk or over a network this is a small and acceptable slowdown. It should be noted that a speedup of about a factor of 2.5 compared to Ihm and Parks method [37] has recently been obtained in [4] for color volumes. This speedup, if it holds for the Visible Man data set, results in a method faster than ours, especially at high PSNR levels. However, our method achieves significantly higher compression ratios for the same PSNR level.

Observe that accessing voxels from highly compressed data is faster than accessing voxels from less compressed data. The reason for this is that at high compression ratios more wavelet coefficients are zero resulting in faster termination of the reconstruction algorithm in Section 7.3.

7.4.3 Selective Block-wise Decoding

Applications working with volumetric data hardly make accesses purely at random. Instead accesses are made in some regular way. As explained in Section 7.3 this might lead to more efficient decoding. We have performed an experiment where voxels are reconstructed in $4 \times 4 \times 4$ blocks in the following way. A $4 \times 4 \times 4$ block is considered to contain voxels from 3 P-slices followed by an I-slice. We start by decoding the 4×4 voxels in the I-slice. Assuming that we have kept the I-slice voxel values from the block above the one we are re-

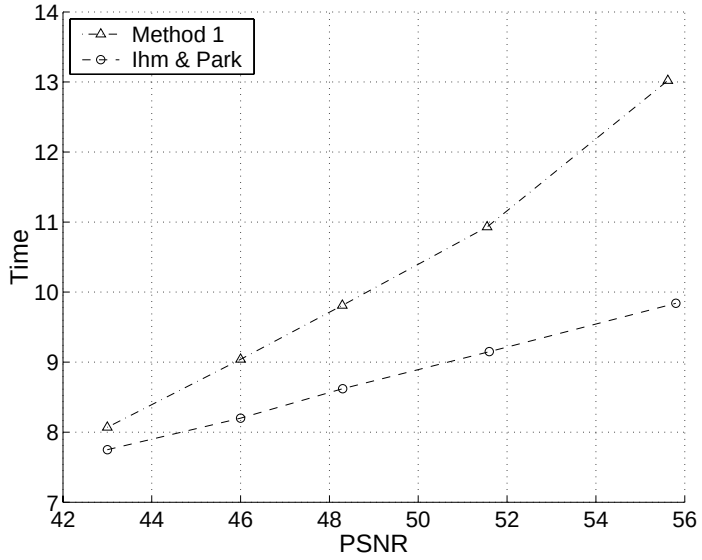


Figure 7.9: Time for accessing 1,000,000 voxels in compressed data for the Visible Man data set on the SGI.

constructing we compute the 4×4 voxels of each P-slice reusing the already computed I-slice values. Results on reconstruction time for the whole volume are shown in Table 7.10. We also experimented with how long it takes to access the uncompressed volumes block by block. This was done by using 6 nested loops. The first three loops indexed the blocks, while the last three loops accessed voxels within each block. As the Visible Man data set could not fit into the memory of the computers used, we simulated block-wise access by accessing a $256 \times 512 \times 512$ array (half the size of the data set), multiplying the time it took by two. On the PC we obtained different times for the Visible Man data set depending on whether the loops were arranged in a cache friendly way or not. This effect was not observed for the SGI. We believe that the SGI compiler optimizes for such effects, as using lesser optimization options resulted in significantly longer timings.

CPU	Uncompressed	Compression Ratio				
		60.2:1	42.6:1	32.4:1	22.2:1	14.5:1
PC	1.48–2.20	19.22	19.87	20.48	21.45	22.93
SGI	3.04	41.60	43.00	44.3	46.35	50.09

Table 7.11: Selective block-wise reconstruction times in seconds for the Visible Man data set.

Roughly an extra 18-20 seconds on the PC are spent on the compressed volumes than on the original. On SGI about 38-47 extra seconds are needed. Considering that the images in Figure 7.14 and Figure 7.16 shown at the end of this chapter each took about 4 minutes to render on the SGI we find this

acceptable. Figure 7.10 compares our method to [37]. We note that our method is about 35% slower. We find this quite acceptable considering the higher compression ratio offered. However, if the significance map improvement in [4] is able to improve the block-wise decoding results of [37] by a factor of 4-5.5 this becomes by far the fastest method at high PSNR levels.

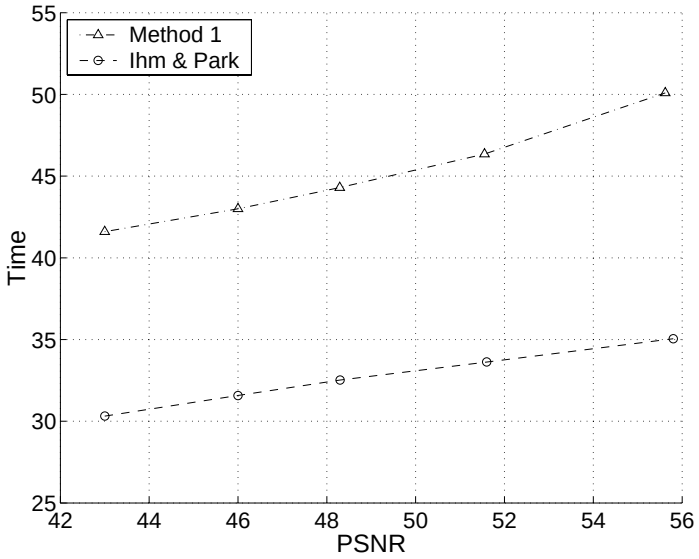


Figure 7.10: Selective block-wise decoding for the Visible Man data set on the SGI.

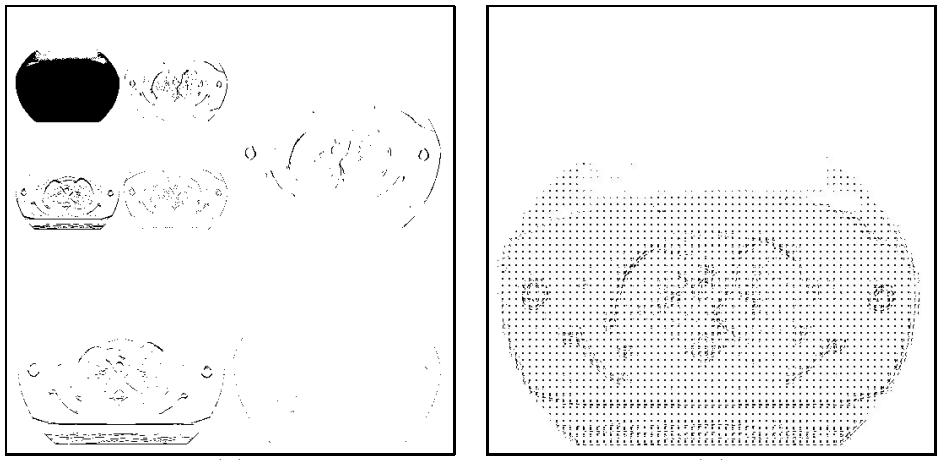
7.5 Issues with Block Indexing Methods

We will understand *block indexing* as a technique used to efficiently identify and locate significant coefficients in a compressed bit-stream. When using block indexing the decomposed volume is divided into blocks, either two- or three-dimensional, and a bitmap is used to signal which blocks contain significant coefficients. To save bits for the significance map the blocks are initially chosen large. To further locate significant coefficients the significant blocks are then recursively split resulting in new smaller blocks with corresponding significant maps. To facilitate fast random access, pointers are used to efficiently locate the begin of the code stream of each block. Both the method presented in this chapter and the methods [38, 4] described in Chapter 6.1 can be classified as block indexing techniques.

On average about 13 wavelet coefficients must be retrieved in order to reconstruct a voxel for the method described in this chapter, see Table 7.5. Since each of the 13 coefficients is located in different blocks, the data necessary to reconstruct a voxel might reside far apart in memory. However, for the three-dimensional scheme described in [38], 15 coefficients must be decoded from the compressed bit-stream. One reason why the method in [38] is slightly faster than our method is that even though more wavelet coefficients must be decoded in order to reconstruct a voxel, it is more cache efficient. Initially the

volume is divided into blocks and each block is wavelet transformed and coded separately. To reconstruct a voxel, only data from one block must be accessed and processed.

However, there are two problems with computing the wavelet transform block-wise. First of all, it does not fully exploit the clustering of zeros. This is illustrated in Figure 7.11. This figure shows the difference in the significance map between decomposing the volume block-wise and decomposing the whole volume. Figure 7.11(a) is the same as Figure 7.4(a). Comparing Figure 7.11(a) with the slices in Figure 7.12 we note that except for the average part, significant coefficients occur mostly at edges. In Figure 7.11(b), however, the significant coefficients are much more uniformly distributed and it is clear that large blocks are less useful.



(a) (b)

Figure 7.11: Significance maps of thresholded wavelet decompositions with PSNR level 46. A black dot in the maps indicates the position of a nonzero coefficient. (a) Decomposition of a whole slice. (b) Block-wise decomposition of a slice.

Secondly, blocking does not generalize well to higher order wavelets with corresponding longer filters. The reason for this, as discussed in Chapter 2.6, is boundary conditions. Because of blocking, boundary voxels constitute a much larger fraction of the volume than when no blocking is used.

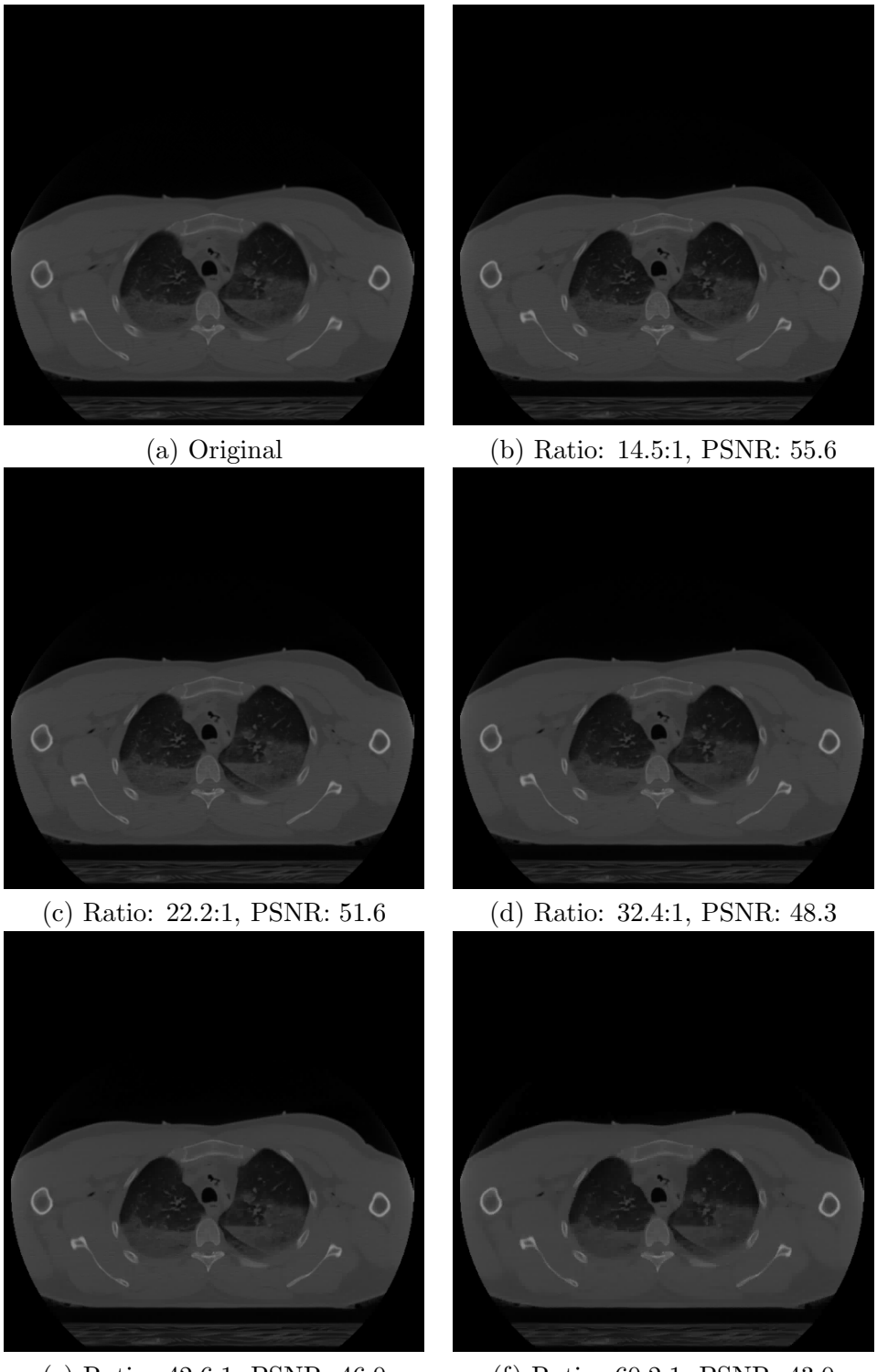
Considering the two above mentioned issues it is clear, from a compression ratio point of view, that it is an advantage to compute the wavelet transform on the whole volume instead of using a block-wise decomposition. Unfortunately, the two-dimensional method described in this chapter does not support scalable decoding in all three dimensions. For that we need a three-dimensional multiresolution analysis. Also, as discussed, the method does not utilize CPU cache as efficiently as the other methods, which becomes even worse in three dimensions as more wavelet coefficients have to be decoded.

In Chapter 8 we present a different approach (`Method 2`) to lossy compression with fast random access that does not use block indexing to code the significance map. The approach generalizes well to other wavelets and it sup-

ports scalable decoding as it is based on a three-dimensional wavelet transform on the whole volume. Also, it clusters coefficients of the same resolution level and the same spatial position together for cache efficient retrieval. It is superior with respect to compression ratio and access time. Unfortunately, it lacks the ability to compress data in an online setting since it makes multiple passes over the data. A comparison and further discussion of `Method 1` given in this chapter and `Method 2` presented in Chapter 8 are the topics of Chapter 9.

7.6 Chapter Summary

We have presented a wavelet based 3D compression scheme for very large volume data supporting fast random access to individual voxels within the volume. Experiments on the CT data of the Visible Man have proven that our method is capable of providing high compression rates with fairly fast decoding of random voxels. Our intension is to provide a method that allows a wider range of users the ability of working with and visualize very large volume data.

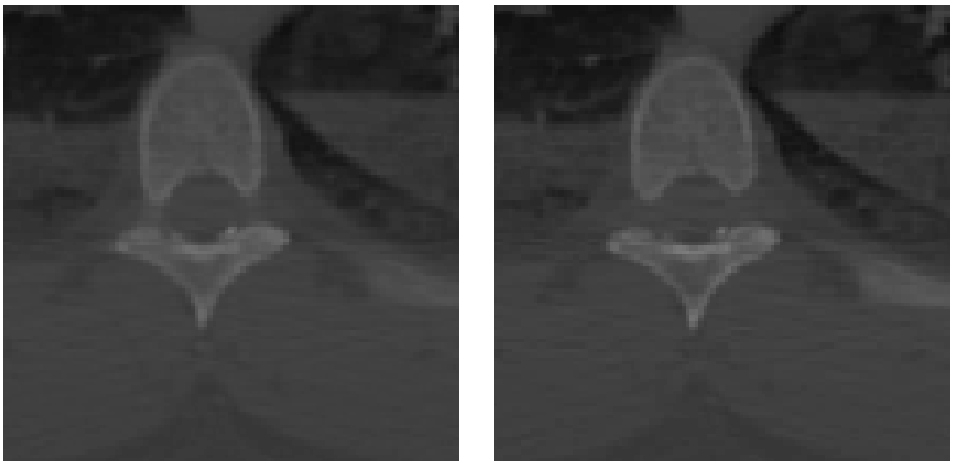


(a) Original (b) Ratio: 14.5:1, PSNR: 55.6

(c) Ratio: 22.2:1, PSNR: 51.6 (d) Ratio: 32.4:1, PSNR: 48.3

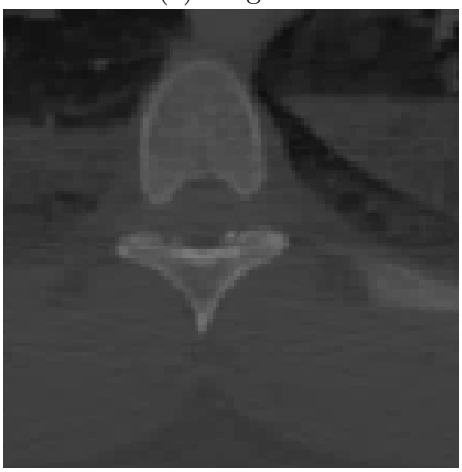
(e) Ratio: 42.6:1, PSNR: 46.0 (f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.12: Sample slice (no. 345) of the Visible Man data set for various compression ratios.

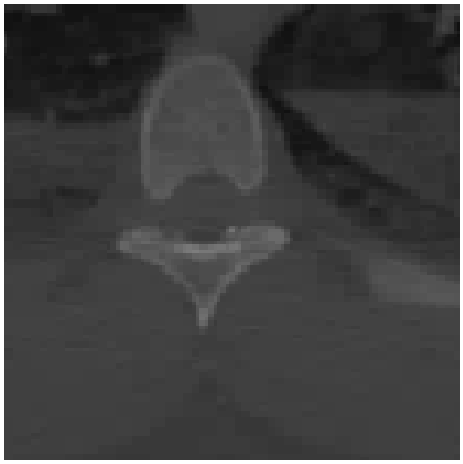


(a) Original

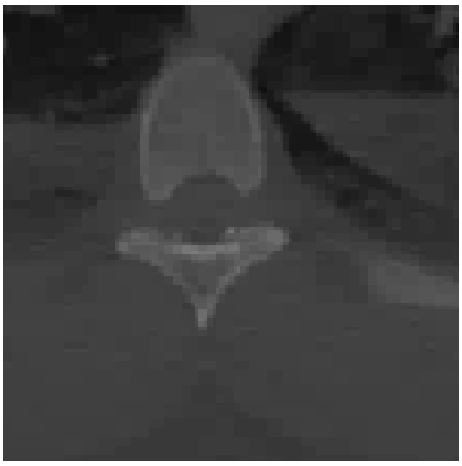
(b) Ratio: 14.5:1, PSNR: 55.6



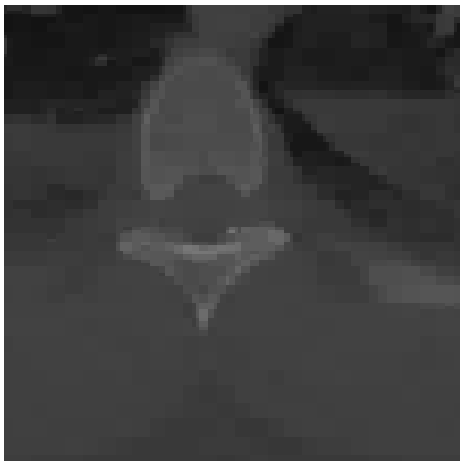
(c) Ratio: 22.2:1, PSNR: 51.6



(d) Ratio: 32.4:1, PSNR: 48.3

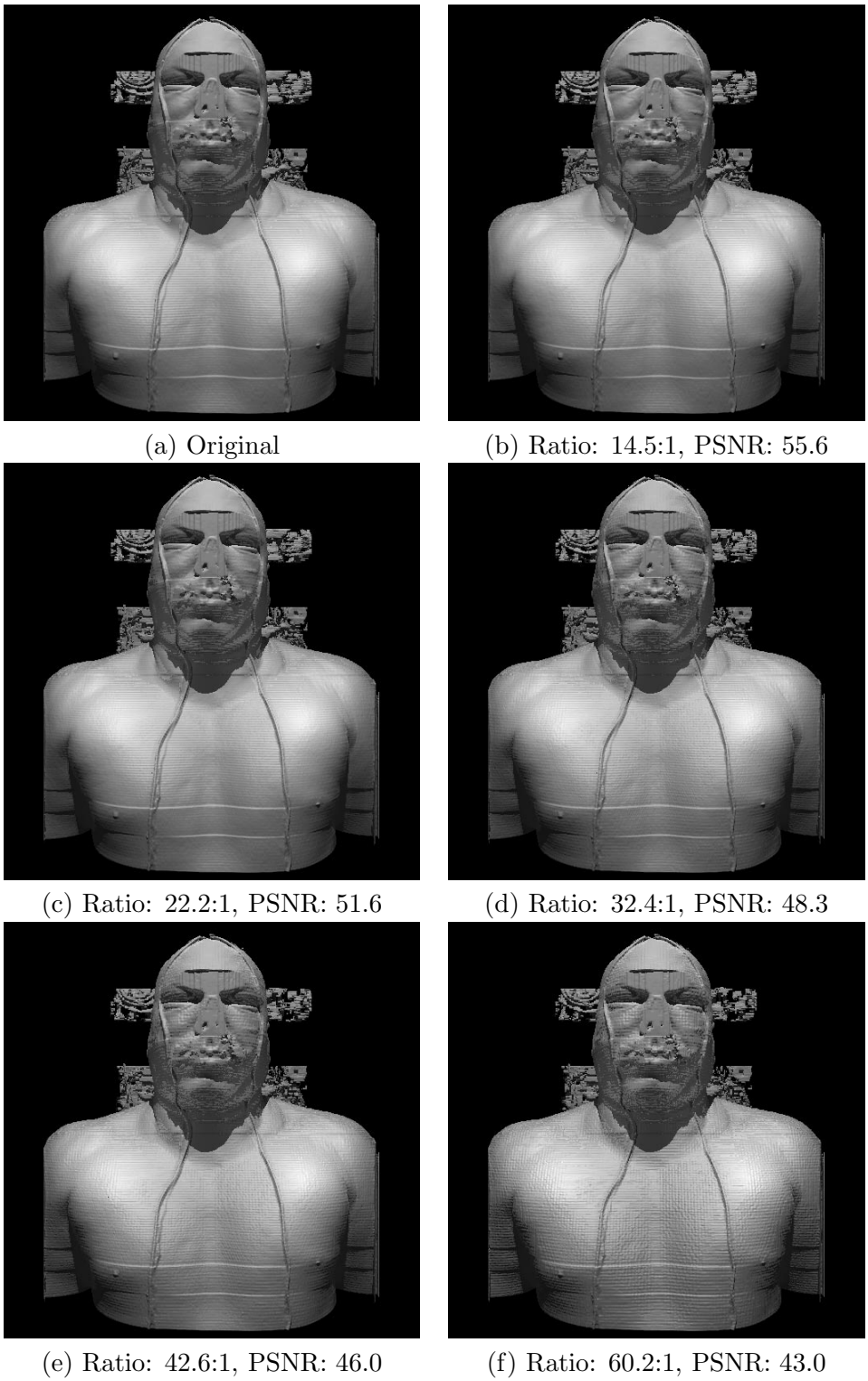


(e) Ratio: 42.6:1, PSNR: 46.0



(f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.13: Zoom of sample slice (no. 345) of the Visible Man data set for various compression ratios.

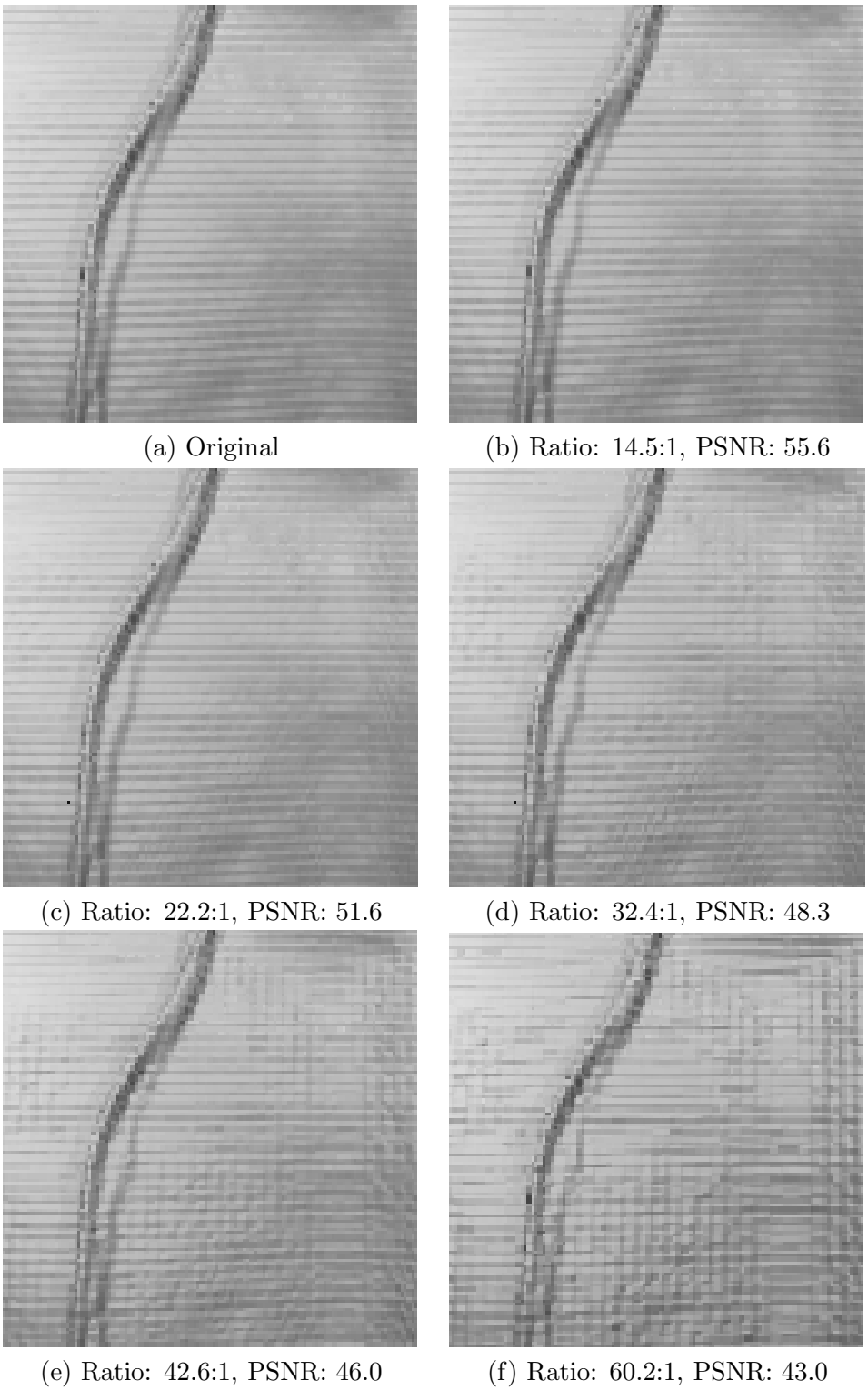


(a) Original (b) Ratio: 14.5:1, PSNR: 55.6

(c) Ratio: 22.2:1, PSNR: 51.6 (d) Ratio: 32.4:1, PSNR: 48.3

(e) Ratio: 42.6:1, PSNR: 46.0 (f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.14: Rendered images of the Visible Man data set for various compression ratios.



(a) Original

(b) Ratio: 14.5:1, PSNR: 55.6

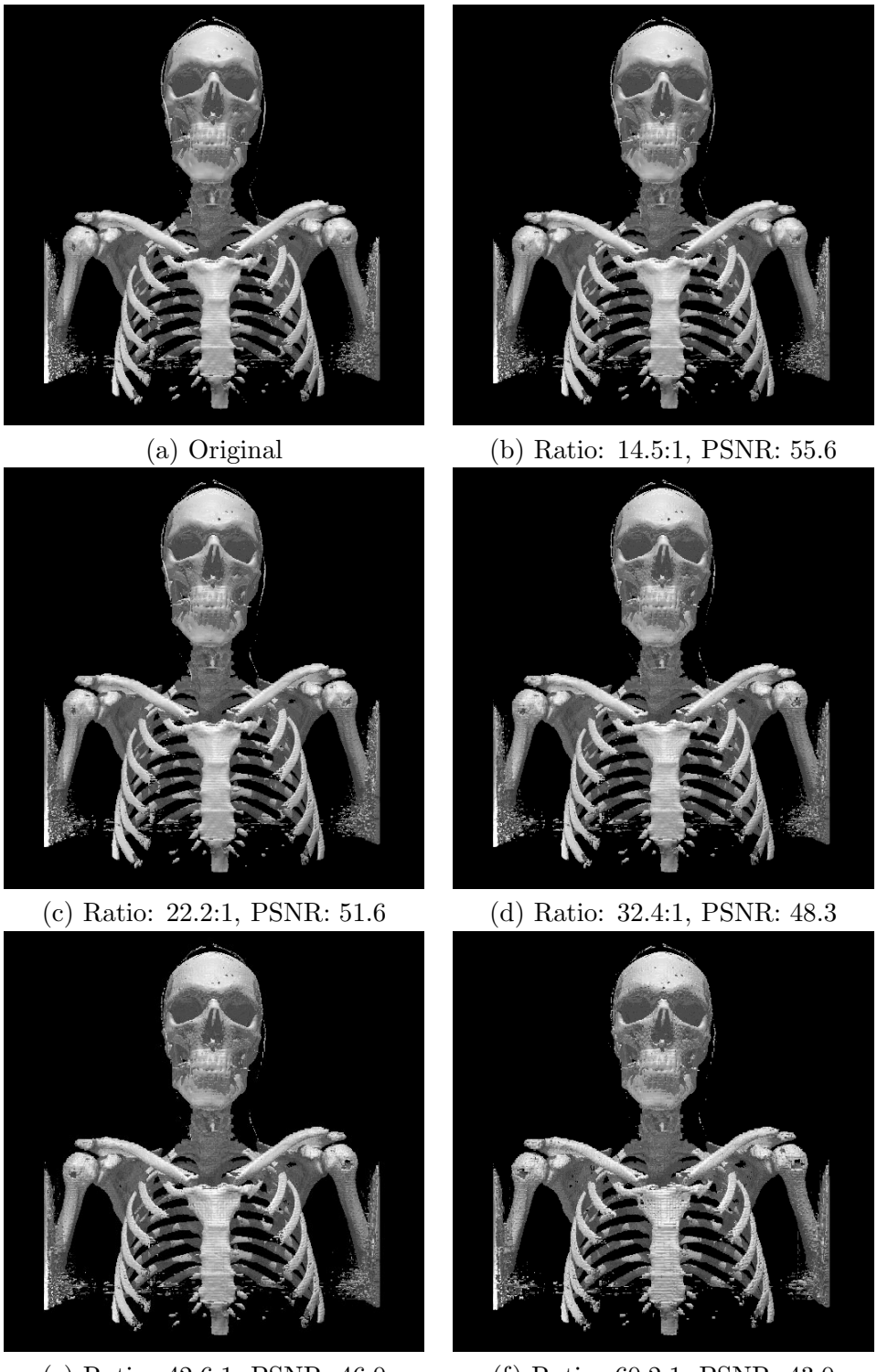
(c) Ratio: 22.2:1, PSNR: 51.6

(d) Ratio: 32.4:1, PSNR: 48.3

(e) Ratio: 42.6:1, PSNR: 46.0

(f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.15: Zoom of rendered images of the Visible Man data set for various compression ratios.

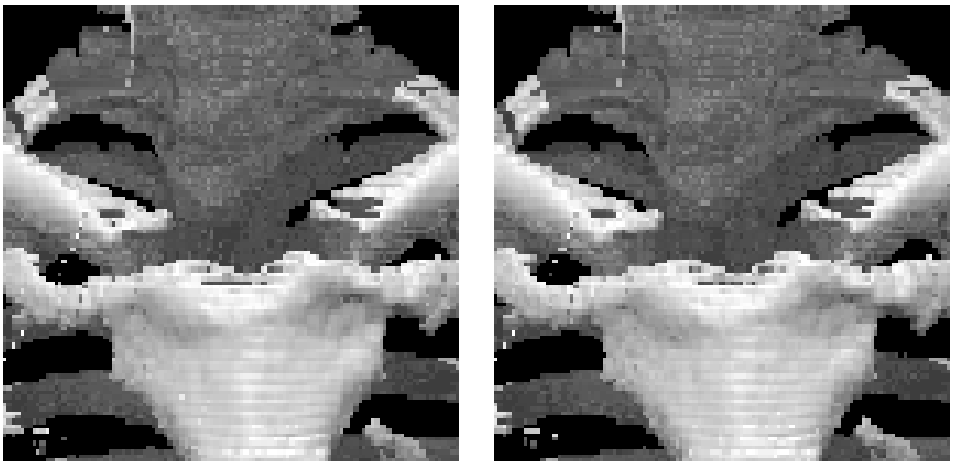


(a) Original (b) Ratio: 14.5:1, PSNR: 55.6

(c) Ratio: 22.2:1, PSNR: 51.6 (d) Ratio: 32.4:1, PSNR: 48.3

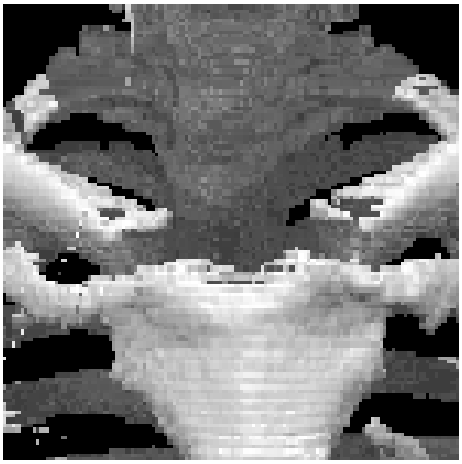
(e) Ratio: 42.6:1, PSNR: 46.0 (f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.16: Rendered images of bone from the Visible Man data set for various compression ratios.

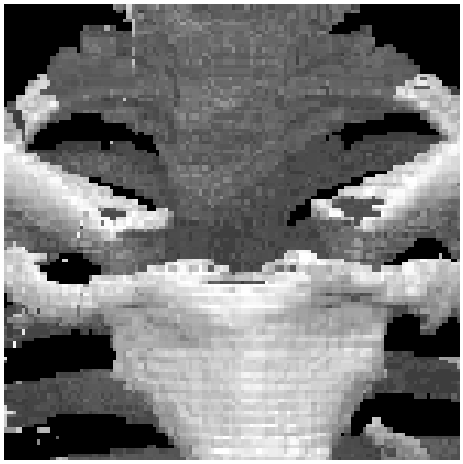


(a) Original

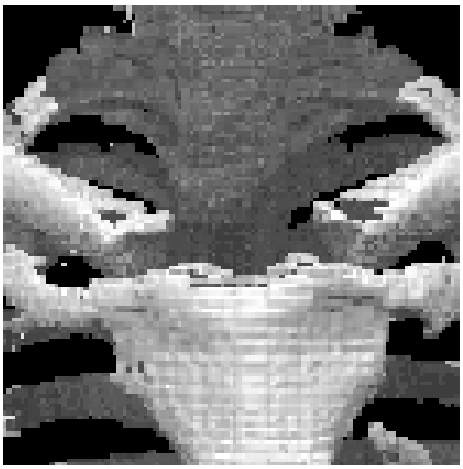
(b) Ratio: 14.5:1, PSNR: 55.6



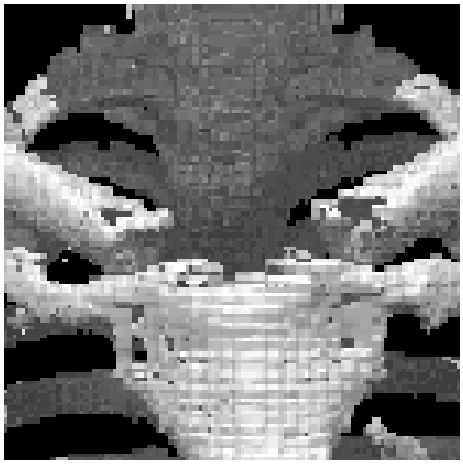
(c) Ratio: 22.2:1, PSNR: 51.6



(d) Ratio: 32.4:1, PSNR: 48.3



(e) Ratio: 42.6:1, PSNR: 46.0



(f) Ratio: 60.2:1, PSNR: 43.0

Figure 7.17: Zoom of rendered images of bone from the Visible Man data set for various compression ratios.

Chapter 8

Coding with Three-dimensional Wavelets and Hashing

In this chapter we present a coding scheme for wavelet based compression of very large volume data with fast random access. It is based on a new approach to lossy storage of the coefficients of wavelet transformed data. Instead of storing the set of coefficients with largest magnitude we will allow a slightly different set of coefficients to be stored. To this end we use the lossy dictionary data structure described in Chapter 5. As mentioned in Chapter 5 this data structure provides space efficient storage and very efficient retrieval of the coefficients. Experiments on the Visible Man data set show an up to 80% improvement in compression ratio over previously suggested schemes including the approach presented in Chapter 7. Furthermore, the time for accessing a random voxel is competitive to existing schemes.

8.1 Coding Method

In designing our coding method we have to consider the trade-off between compression ratio, distortion, and speed of retrieval. As mentioned the Haar wavelet decomposition is simple to compute. In order to reconstruct a single data value, only one coefficient from each subband must be retrieved. Higher order wavelets, while promising better compression ratios, require the retrieval of significantly more coefficients. Focusing on fast retrieval of voxels, we choose to use the Haar wavelet. However, our method easily generalizes to other wavelets. As we shall see in Section 8.2 the Haar wavelet provides good compression results.

We have chosen to use 3 decomposition levels. This provides a good trade-off between how sparse a representation the wavelet transform produces, and the number of coefficients needed to reconstruct a single voxel. We thus have one level of average coefficients and three levels of wavelet coefficients, each level consisting of seven subbands, as depicted in Figure 8.1. In order to reconstruct a voxel, we need 22 coefficients: 1 average coefficient, and 7 wavelet coefficients from each of the three detail levels. Most wavelet based compression schemes exploit the fact, mentioned in Chapter 2.5, that a very sparse representation can be obtained by thresholding the wavelet representation. After thresholding, the challenge is to efficiently store positions and values of the nonzero coefficients.

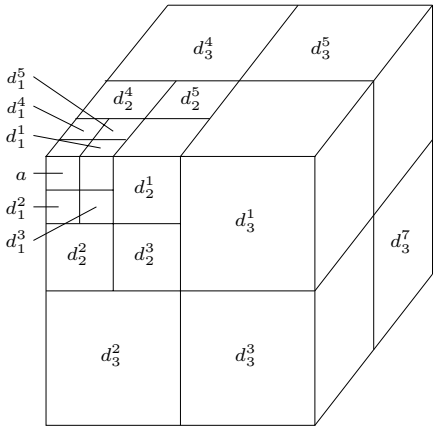


Figure 8.1: Subbands of a three-level three-dimensional wavelet transform. The subbands are numbered d_j^i, where j denotes the resolution level while i denotes the subband number within the level.

This corresponds to storing the set W containing the n most important wavelet coefficients. Using the hashing scheme described in Chapter 5, we will take a slightly different approach. We noted in Chapter 5 that the most space efficient methods for storing sparse tables have a rather high overhead for lookups, but that an index set W' slightly different from a given set W can be stored in a way allowing much more efficient lookups. By not storing exactly the n coefficients of largest magnitude, we get a larger MSE, but we will see that the increase is quite acceptable compared to the gain in compression ratio.

Since the sparseness of the wavelet coefficients change between subband levels, we will use a different set of hash tables for each level. For the moment, we will in fact pretend that each subband is stored as a separate lossy dictionary. We later merge the hash tables of subbands at the same level. The main reason for choosing a three-level (instead of a two-level) wavelet transform is that the average coefficients take up a fraction of just 2^{-9} of the wavelet transformed volume, and can therefore with negligible overhead be stored in uncompressed form.

To use the lossy dictionary data structure, we map (x, y, z) positions of a subband B to indices in the range $0, \ldots, |B| - 1$ in a straightforward way. Denote, by $i_{x,y,z}$ the index corresponding to (x, y, z). We use pairwise independent pseudo-random permutations of the following form:

$$\phi(x, y, z) = (a\, i_{x,y,z} + b) \bmod p \ , \tag{8.1}$$

where p is the smallest prime greater than $|B|$, and a, b are chosen at random such that $0 < a < p$, and $0 \leq b < p$. For each hash table, we have two such permutations ϕ_1 and ϕ_2, and take as our hash functions $h_1(x, y, z) = \phi_1(x, y, z) \bmod r/2$ and $h_2(x, y, z) = \phi_2(x, y, z) \bmod r/2$, where r denotes the combined size of the two tables. To store the coefficient with index (x, y, z) in entry $h_1(x, y, z)$ of hash table one, we only need to specify the quotient $\phi_1(x, y, z)$ div $r/2$ and the uniformly quantized value of the coefficient (the same

holds for hash table two). We choose the number of quantization levels to be a value such that the distortion introduced by quantization constitutes a minor part of the total distortion. This desired value is then adjusted slightly, such that the quantization index and the quotient fit into an integer number of bits. This defines the size of each entry in the hash tables. In case of an empty cell, we store the quotient $((a\,(p-1)+b) \bmod p) \operatorname{div} r/2$, which does not match the quotient of any index (x, y, z). This is in correspondence with the definition of quotient functions in Theorem 5.1 of Chapter 5.

For performance reasons, we do not store the significant coefficients exactly as described above where each subband is stored in a separate lossy dictionary. Instead, we store all coefficients of a subband level in a single lossy dictionary of correspondingly larger size. This is done in such a way that the wavelet coefficients of a subband level that are needed for the same voxel reconstruction are more or less adjacent in the hash tables. This significantly improves the time for retrieving the coefficients, because of better cache usage. The merging of the tables is illustrated in Figure 8.2. At each subband level, we number the seven subbands $0, \ldots, 6$. The way in which wavelet coefficients are accessed at each subband level during a voxel reconstruction is as follows: The same index within each subband is needed. When hashing index (x, y, z) of the subband with number s we use the functions

$$h'_i(x, y, z, s) = (\phi_i(x, y, z) + s) \bmod r/2, \tag{8.2}$$

for $i = 1, 2$. This gives corresponding indices in different subbands at the same subband level consecutive hash function values (modulo $r/2$). There was no significant degradation in MSE as a result of making hash function values correlated in this way. To be able to distinguish coefficients in different subbands, we pack the subband number together with the quotient.

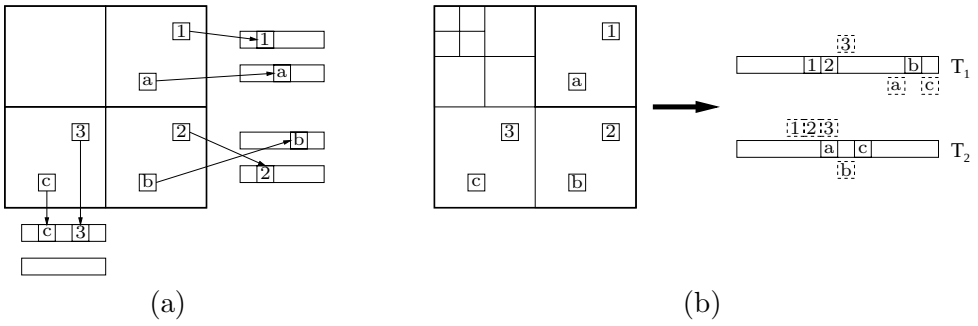
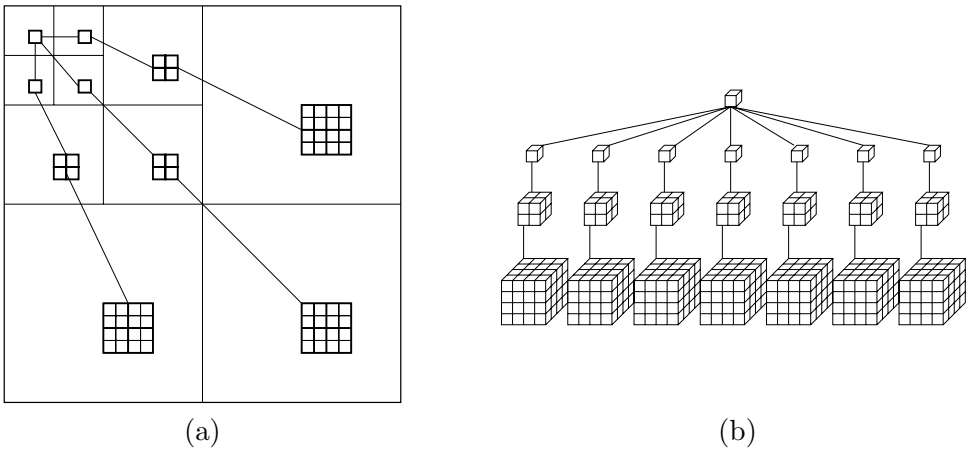


Figure 8.2: How tables are merged. (a) No merging – a separate lossy dictionary is used for each subband. (b) All coefficients of a subband level are stored in one lossy dictionary. Note that coefficients at the same spatial position get consecutive hash values as shown by the dashed boxes.

The above describes how we store the wavelet coefficients, but not how to determine the size of the hash tables at different subband levels. This is

nontrivial, as the best ratio between tables at different levels depends on the volume and the desired distortion. We use a simple heuristic, namely to perform global thresholding and use the number of nonzero coefficients at each level as the total size of the hash tables at that level. As we nearly store the coefficients remaining after thresholding, the MSE will be similar to the MSE introduced by the thresholding.



(a) (b)

Figure 8.3: Wavelet coefficients corresponding to the same spatial location. (a) Illustrated in two dimensions. (b) In three dimensions we obtain an octree like structure. Note that, except for the top voxel, all voxels point to eight children. For clarity this is not shown.

While the scheme described this far is reasonably fast, a significant speedup can be obtained by exploiting the fundamental property of the stored wavelet coefficients: A very large fraction of coefficients is zero. Thus, in many cases only zero coefficients are found in the detail levels. To avoid such expensive lookups we store, together with each average coefficient, a small significance map that can be consulted to avoid some lookups of detail coefficients that are zero. In particular, we divide subband levels 3 and 2 into $2 \times 2 \times 2$ blocks. If all seven blocks at the same spatial location in the seven different subbands of a level only have zero coefficients, we set a bit in the significance map of the corresponding average coefficient. This uses 8 bits for subband level 3 (at level 3 there are 8 blocks of size $2 \times 2 \times 2$ that correspond to the same spatial location as the average coefficient), and a single bit for subband level 2. Figure 8.3(a) shows, in two dimensions, the spatial relationship between the blocks at different subband levels. Finally, we set a bit to indicate whether all seven corresponding wavelet coefficients at subband level 1 are zero. In total, we use 10 bits for the significance map per average coefficient. Since the average coefficients only constitute a fraction of 2^{-9} of all wavelet coefficients this is negligible. The complete data structure used is illustrated in Figure 8.4. The illustration is in two dimensions for simplicity.

When reconstructing a voxel we first look up the average coefficient and the significance map. For each subband level, where the significance map does not declare all coefficients to be zero, we must then look up the seven coef-

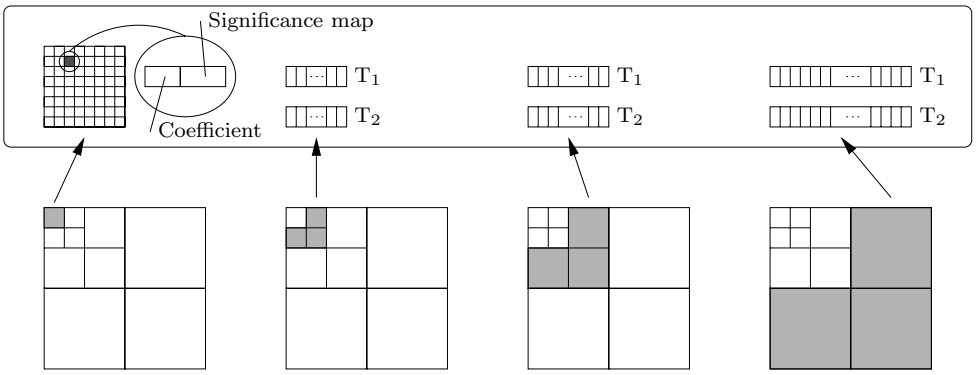


Figure 8.4: How the coefficients at different levels are stored in the data structure.

ficients in the hash tables of that level. Let r denote the combined size of the two hash tables used for the subband level, and let ϕ_1 and ϕ_2 denote the pseudo-random permutations used. If the position within the subbands, at a given subband level, is (x, y, z), we evaluate $\phi_1(x, y, z)$ and $\phi_2(x, y, z)$. Then, starting at position $\phi_1(x, y, z) \bmod r/2$ of hash table one and position $\phi_2(x, y, z) \bmod r/2$ of hash table two, we extract two sequences of seven consecutive quotients and subband numbers. If a quotient in the sequence from table i equals $\phi_i(x, y, z)$ div $r/2$, for some i, and the subband number equals the position in the sequence, the coefficient in that subband is nonzero, and its value is calculated from the quantized value in the table. Otherwise the coefficient is zero. When all the necessary wavelet coefficients have been extracted, the inverse Haar wavelet transform is applied. Using the same notation as in Chapter 2.4.3, the reconstruction formulas for one level of the inverse Haar wavelet transform are

$$\begin{aligned}
c_1 &= (a + d^1 + d^2 + d^3 + d^4 + d^5 + d^6 + d^7)/2\sqrt{2} \\
c_2 &= (a + d^1 + d^2 + d^3 - d^4 - d^5 - d^6 - d^7)/2\sqrt{2} \\
c_3 &= (a + d^1 - d^2 - d^3 + d^4 + d^5 - d^6 - d^7)/2\sqrt{2} \\
c_4 &= (a + d^1 - d^2 - d^3 - d^4 - d^5 + d^6 + d^7)/2\sqrt{2} \\
c_5 &= (a - d^1 + d^2 - d^3 + d^4 - d^5 + d^6 - d^7)/2\sqrt{2} \\
c_6 &= (a - d^1 + d^2 - d^3 - d^4 + d^5 - d^6 + d^7)/2\sqrt{2} \\
c_7 &= (a - d^1 - d^2 + d^3 + d^4 - d^5 - d^6 + d^7)/2\sqrt{2} \\
c_8 &= (a - d^1 - d^2 + d^3 - d^4 + d^5 + d^6 - d^7)/2\sqrt{2}
\end{aligned} \tag{8.3}$$

Depending on the voxel that is to be reconstructed we must traverse an octree like structure from top to bottom, as illustrated in Figure 8.3(b), using one of the eight formulas at each node in the octree.

As discussed in Chapter 6 applications do not always access data totally at random. To enhance decoding efficiency when some locality in the access patterns exists, our decoding algorithm also supports selective block-wise de-

coding. When reconstructing a single voxel, seven wavelet coefficients must be retrieved for each subband level. These wavelet coefficients can be reused to compute neighboring voxels. In selective block-wise decoding, we retrieve all 512 wavelet coefficients necessary to compute all voxels in a $8 \times 8 \times 8$ block. To do this we must decode one average coefficient and all its associated detail coefficients as illustrated in Figure 8.3(b). From (8.3) it appears that 56 additions/subtractions must be performed to reconstruct eight voxels. However, by reusing computations this can be reduced to 24 operations.

8.2 Experiments

In this section we present results based on two data sets. The first data set is the $512 \times 512 \times 512$ CT data set of the upper body of the Visible Man used in Chapter 7. Each voxel is stored in 2 bytes, in which 12 bits are used, and the whole volume takes up 256 Mbytes. The second data set is a CT scan of an engine block, resampled to size $256 \times 256 \times 112$ to make all dimensions a multiple of 8. Each voxel is stored in 1 byte and the volume uses about 7 Mbytes of space. The original Engine Block data set was obtained from [24].

	Experiment			
Subband level	1	2	3	Best
3	166.59	**162.49**	163.02	162.49
2	54.10	**53.23**	53.98	53.23
1	24.29	15.23	**13.57**	13.57
Total MSE	244.98	230.95	230.57	229.29
Total PSNR	48.84	48.61	48.62	48.64

Table 8.1: MSE for the three subband levels when using three different pseudo-random permutations.

The exact set of wavelet coefficients that is stored during compression depends somewhat on the particular parameters of the pseudo-random permutations in (8.1). For each compression ratio, we have compressed the volume three times, and for each subband level selected the permutation yielding the best results. Table 8.1 shows the results of compressing the Visible Man data set three times at a compression ratio of 44:1. The highlighted values in the table show the best MSE for each subband. Except for subband level 1 there is little variation in the MSE between the three runs. For higher compression ratios we observed a decrease in this variation. For lower compression ratios the variation decreased in subband level 1 and increased in subband level 2. We believe this behavior can be explained in terms of Figure 5.6 in Chapter 5. This figure shows the observed probability that a given coefficient is included in the dictionary when using truly random and independent hash functions. However, we use weaker hash functions in our implementation. This fact, together with the particular decay of the wavelet coefficients (see Figure 5.7 for an example) for the subband level in relation to the hash table sizes, result in the observed variation of the MSE.

As mentioned in Section 8.1 we uniformly quantize the wavelet coefficients. The number of quantization levels is chosen such that the quantization induced distortion is a minor part of the total distortion. The number of bits used for coefficients at subband level 3 varied between 4.0 and 7.5 bits depending on the compression ratio. The exact number was chosen such that the hash quotient and the quantization level would fit into an integer number of bits. For the other subband levels we increase the desired number of bits used by 1.5 bits per level. This is necessary because of the $2\sqrt{2}$ scaling of the wavelet coefficients in (2.45) of Chapter 2.

8.2.1 Compression Ratio and Distortion

It is important that our compression scheme provides high compression ratios at low distortion, while maintaining fast random access. In our first experiment we compressed the two test volumes at various ratios. The results are summarized in Tables 8.2 and 8.3. Figures 8.5 and 8.6 show the PSNR as a function of the compression ratio. The PSNR was calculated from the voxel differences between the original voxel values and the voxel values reconstructed from the compressed data. In Figure 8.5 we also show the rate-distortion performance of `Method 1` described in Chapter 7 and the method by Ihm and Park given in [37]. As can be seen from the figure, our method gives strictly better performance in the distortion range investigated, and it significantly outperforms the previous best methods, nearly doubling the compression ratio compared to `method 1` in Chapter 7 at high ratios. For low compression ratios it seems that the three methods converge. As in Chapter 7 we note that a 10%–15% improvement of [37] was reported in [4].

Comparing the fraction of nonzero wavelet coefficients, shown in Table 8.2, with the same fraction of `method 1`, shown in Table 7.9, we find one explanation for the higher compression ratios. The method in this chapter simply succeeds in setting more coefficients to zero for the same distortion.

Figure 8.6 shows that the achieved compression ratios for the Engine Block data set are not as large as for the Visible Man data set. This can be explained by observing the percent of nonzero wavelet coefficients in Tables 8.2 and 8.3. We observe that the wavelet decomposition performs better on the Visible Man data set, in terms of obtaining a sparse representation for a given distortion, than on the Engine Block data set. This can partially be explained by noting that large areas of the Visible Man data set contains only zero valued voxels.

Even though the PSNR measure is an accepted measure in lossy compression, visual inspection of the quality is very important. To that end Figures 8.9 and 8.10 shows sample slices for the various compression ratios. We observe that for the best PSNRs the reconstructed slices are virtually indistinguishable from the original. However, for a PSNR of 43 blocking artifacts and loss of small level detail occur.

We also generated rendered images of the original and compressed volumes using the VolPack rendering engine [87]. The rendered images are depicted in Figures 8.11 to 8.14. For the Visible Man data set, the images at compression ratios of up to 44:1 are essentially indistinguishable from the original. However,

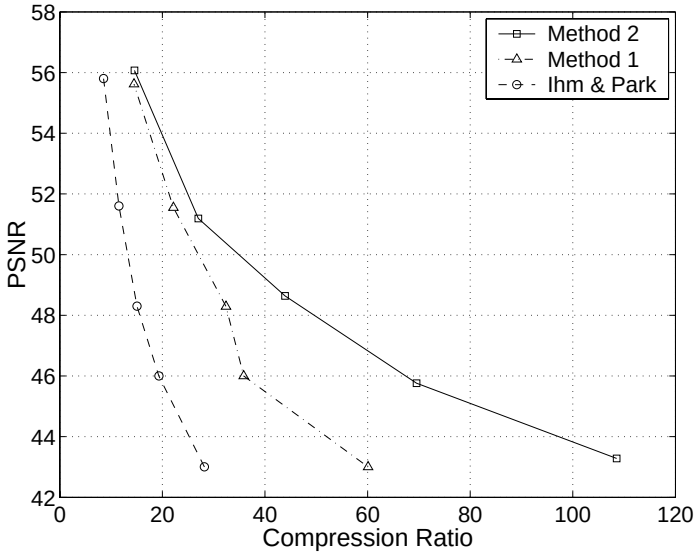


Figure 8.5: Rate-distortion curve for the Visible Man data set.

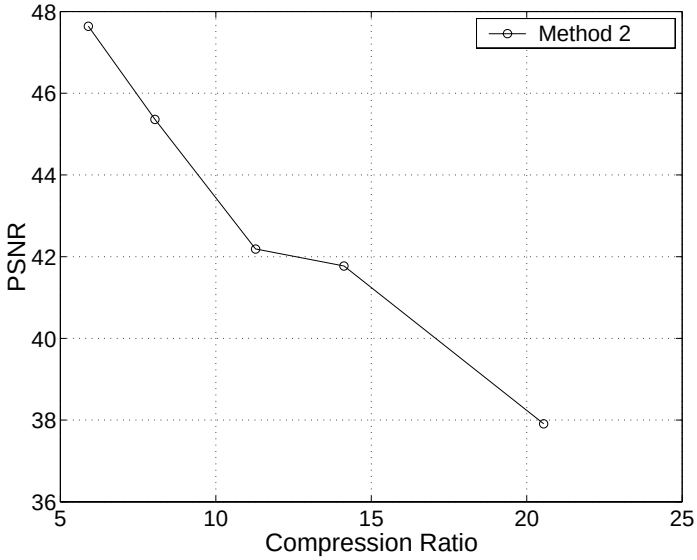


Figure 8.6: Rate-distortion curve for the Engine Block data set.

the close ups in Figure 8.12 reveal beginning blocking artifacts for compression ratios of 27:1 and 44:1. For the images at ratio 80:1 and 109:1 reconstruction artifacts become noticeable, however most of the features and details are still preserved. For the Engine Block data set, artifacts begin to appear at compression ratio 11:1 but not until a ratio of 21:1 do they become significantly noticeable. Again, when zooming the blocking artifacts appear sooner.

Compression	Ratio	109:1	80:1	44:1	27:1	15:1
	Size (Mb)	2.36	3.68	5.83	9.48	17.57
Errors	PSNR	43.28	45.76	48.64	51.19	56.07
	SNR	25.25	27.74	30.62	33.17	38.05
	MSE	788.7	445.4	229.3	127.5	41.4
Fraction of nonzero wavelet coefficients		0.96%	1.56%	2.60%	4.68%	8.24%

Table 8.2: Results on compression ratio and quality for the Visible Man data set.

Compression	Ratio	20.6:1	14.1:1	11.3:1	8.0:1	5.9:1
	Size (Mb)	0.34	0.50	0.62	0.87	1.19
Errors	PSNR	37.91	41.77	42.19	45.36	47.64
	SNR	22.29	26.15	26.54	29.74	32.02
	MSE	10.53	4.32	3.95	1.89	1.12
Fraction of nonzero wavelet coefficients		3.60%	4.89%	6.79%	8.78%	11.06%

Table 8.3: Results on compression ratio and quality for the Engine Block data set.

8.2.2 Timing Results

The time for accessing 1,000,000 randomly selected voxels was evaluated as described in Chapter 7.4.2 using the same hardware and compile options. Results are given in Tables 8.4 and 8.5 with and without the significance map optimization discussed in Section 8.1. As in Chapter 7.4.2 we also list in the tables timings for accessing the uncompressed data.

CPU	Significance map	Uncompressed	Compression Ratio				
			109:1	80:1	44:1	27:1	15:1
PC	Yes	0.50	1.61	1.82	2.09	2.44	2.86
	No		4.73	4.84	4.94	4.82	4.95
SGI	Yes	1.65 – 3.03	3.79	4.63	5.68	7.13	8.67
	No		14.88	15.78	16.30	16.70	17.00

Table 8.4: Voxel reconstruction times in seconds for accessing 1,000,000 randomly selected voxels for the Visible Man data set.

CPU	Significance map	Uncompressed	Compression Ratio				
			20.6:1	14.1:1	11.3:1	8.0:1	5.9:1
PC	Yes	0.49	2.10	2.47	2.70	3.14	3.45
	No		4.51	4.64	4.73	4.76	4.97
SGI	Yes	1.17	5.28	6.25	7.09	8.26	8.86
	No		12.93	12.97	13.06	13.12	13.27

Table 8.5: Voxel reconstruction times in seconds for accessing 1,000,000 randomly selected voxels for the Engine Block data set.

Observe from Table 8.4 that on the PC accessing voxels from the compressed Visible Man data set using the significance map is about 3 to 6 times slower than accessing uncompressed data. On the SGI it is, as discussed in Chapter 7.4.2,

difficult to assess the slowdown when accessing compressed data. For the Engine Block data set, accessing compressed data on both computers is about 4.5 to 7.5 times slower. In Figure 8.7 we show the time for accessing 1,000,000 random voxels in compressed data on the SGI as a function of the PSNR level. From the figure, we observe that `Method 2` outperforms `Method 1` in Chapter 7 and the method given in [37]. For high distortion (small PSNR), `Method 2` is about twice as fast as [37]. Note however, that if the speedup of [37] by a factor of about 2.5 reported in [4] also holds for the CT data of the Visible Human this will result in a faster method than `Method 2`, especially at high PSNR levels. Even if this is the case, we believe that `Method 2` is still highly relevant as it produces compression ratios that are up to three times larger than the method in [37].

Similar to observations in Chapter 7 it is observed that accessing voxels from highly compressed data is faster than accessing voxels from less compressed data. The main reason for this is that at high compression ratios fewer wavelet coefficients are kept, resulting in a more efficient significance map because of more zero coefficients. Furthermore, as the compression ratio increases, more of the compressed data might fit into data cache of the computer, which again results in a speedup. This last type of speedup can be noticed in the timings where the significance map was disabled. From Table 8.4 and Table 8.5 we see that enabling the use of the significance map gives a speedup of about 2-4 times for the Visible Man data set compared to having it disabled. The improvement is largest at high compression ratios where the significance map is more effective because of more zeros. For the Engine Block data set, the speedup is slightly smaller. The reason for this, as discussed above, is that the Visible Man data set contains more voxels with value zero.

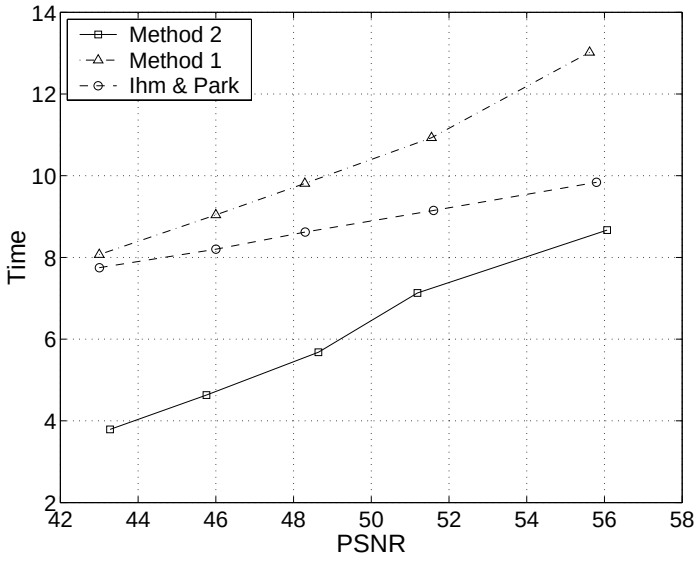


Figure 8.7: Time for accessing 1,000,000 voxels in compressed data for the Visible Man data set on the SGI.

8.2.3 Selective Block-wise Decoding

As mentioned in Chapter 6 not all applications access data completely at random. To improve decoding efficiency when data is accessed locally `Method 2` supports selective block-wise decoding. As mentioned in Section 8.1 the voxels in an $8 \times 8 \times 8$ block can be decoded efficiently by reusing both wavelet coefficients and computations in the reconstruction formulas. The effectiveness of selective block-wise decoding was evaluated by decoding the entire volume, block by block. The timings are shown for the two data sets in Tables 8.6 and 8.7. The timings for accessing the uncompressed volume were done as described in Chapter 7.4.3.

From tables 8.6 and 8.7, we observe that decoding the whole volume blockwise is significantly slower than accessing uncompressed data. However, considering the memory requirements needed for large uncompressed volumes we think this overhead is quite acceptable. Also, applications such as some volume renderers require orders of magnitudes longer processing time. From the tables, we also note that as the compression ratio decreases the decoding time increases. This can be explained by the fact that the significance map is less effective at low ratios. Finally, in Figure 8.8, we compare our method to `Method 1` of Chapter 7 and the method in [37] for the Visible Man data set. For a PSNR level below 50 our method performs best. Again, if the improvement in [4] is able to improve the results of [37] by a factor of 4-5.5, the improved method would be about 7–42 seconds faster than `Method 2` depending on the PSNR level.

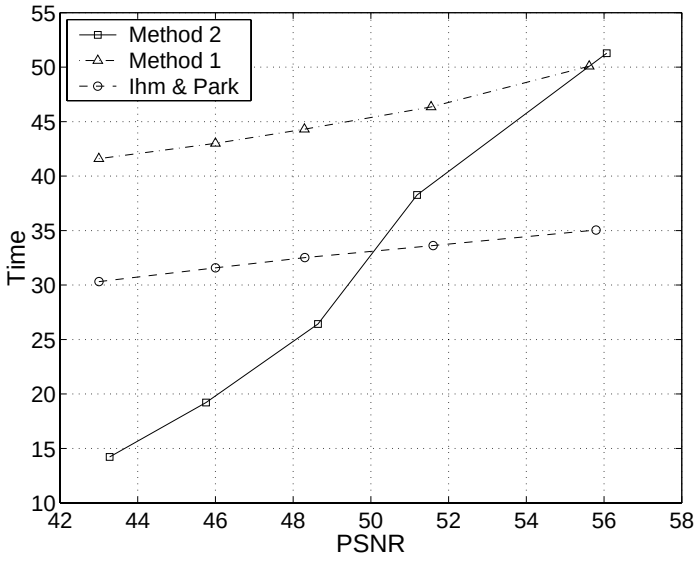


Figure 8.8: Selective block-wise decoding for the Visible Man data set on the SGI.

CPU	Uncompressed	Compression Ratio				
		109:1	80:1	44:1	27:1	15:1
PC	1.48–2.20	6.24	7.75	10.06	13.12	17.17
SGI	3.04	14.22	19.22	26.42	38.27	51.28

Table 8.6: Selective block-wise reconstruction times in seconds for the Visible Man data set.

CPU	Uncompressed	Compression Ratio				
		20.6:1	14.1:1	11.3:1	8.0:1	5.9:1
PC	0.08	0.59	0.71	0.85	1.08	1.33
SGI	0.15	1.46	1.67	2.08	2.65	3.28

Table 8.7: Selective block-wise reconstruction times in seconds for the Engine Block data set.

CPU	Downsampling factor	Uncompressed	Compression Ratio				
			109:1	80:1	44:1	27:1	15:1
PC	1	0.50	1.61	1.82	2.09	2.44	2.86
	2		1.49	1.63	1.82	2.07	2.34
	4		1.18	1.26	1.37	1.50	1.59
	8		0.81	0.81	0.81	0.81	0.81
SGI	1	1.65 – 3.03	3.79	4.63	5.68	7.13	8.67
	2		3.32	3.93	4.68	5.55	6.50
	4		1.18	1.26	1.37	1.50	1.59
	8		1.20	1.20	1.20	1.20	1.20

Table 8.8: Voxel reconstruction times in seconds for the Visible Man data set when accessing 1,000,000 randomly selected voxels at lower resolutions.

CPU	Downsampling factor	Uncompressed	Compression Ratio				
			20.6:1	14.1:1	11.3:1	8.0:1	5.9:1
PC	1	0.50	2.10	2.47	2.70	3.14	3.45
	2		1.80	2.08	2.20	2.47	2.61
	4		1.30	1.42	1.44	1.42	1.44
	8		0.73	0.73	0.73	0.73	0.73
SGI	2	1.65 – 3.03	5.28	6.25	7.09	8.26	8.86
	2		4.48	5.28	5.83	6.54	6.55
	4		2.82	3.23	3.20	3.20	3.22
	8		1.15	1.15	1.15	1.15	1.15

Table 8.9: Voxel reconstruction times in seconds for the Engine Block data set when accessing 1,000,000 randomly selected voxels at lower resolutions.

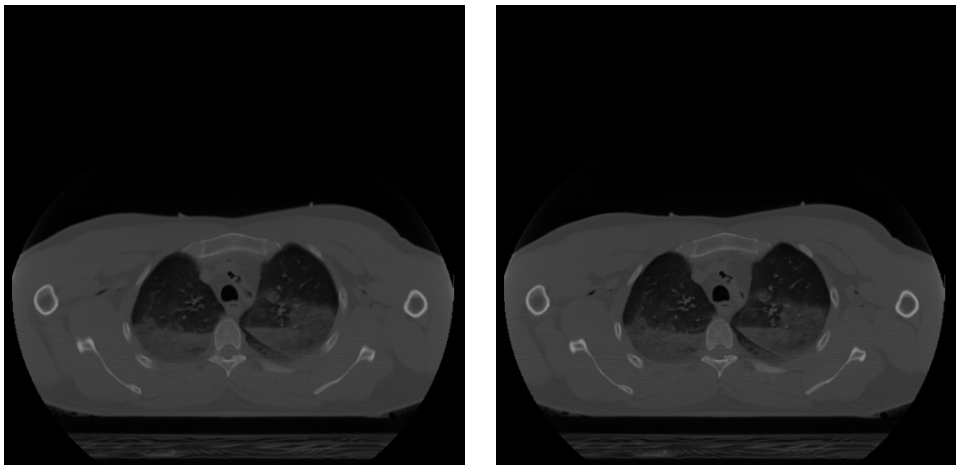
8.3 Scalable or Multiresolution Decoding

While relatively fast, the decoding process might still be too slow for interactive applications. In some applications it is possible to work with a low resolution representation of the volume. Volume rendering is such an example. When the user is satisfied with the setup at the low resolution, the application switches to full resolution producing the final result. Since our method uses

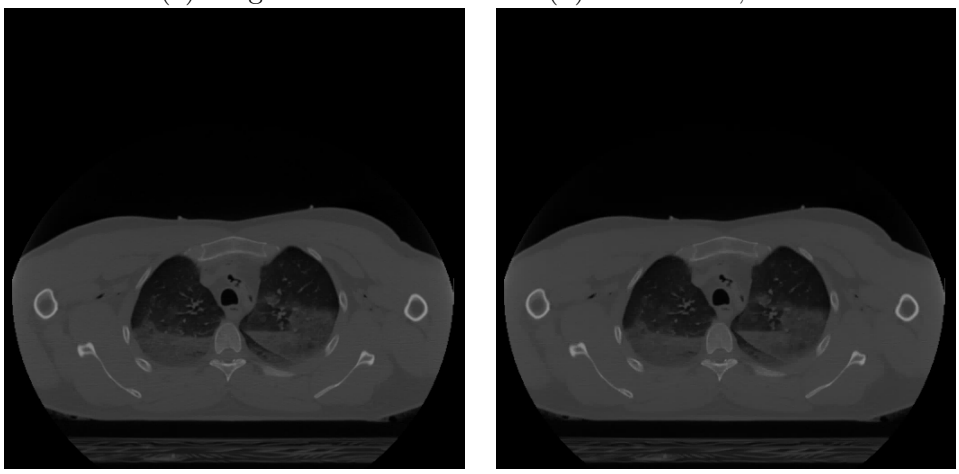
a three-dimensional wavelet transform, it supports multiresolutional decoding in a natural way. For example, by assuming that the wavelet coefficients at subband level 3 are all zero, we can omit the last reconstruction step. This results in a volume half the size of the original volume in each dimension. By omitting coefficients at subband levels 1 and 2, our algorithm can return voxels from a volume that is downsampled by a factor of 2, 4, or 8 in each direction. Since the algorithm does not need to decode coefficients assumed to be zero this results in a speedup. In Table 8.8 and Table 8.9 we present results on decoding 1,000,000 randomly selected voxels from the two compressed volumes at different resolutions. As expected, the reconstruction time of a voxel decreases as the downsampling factor increases. When downsampling 8 times we notice that the access time does not depend on the compression ratio. This is because we access only the uncompressed average coefficients, and these do not change with the compression ratio. Accessing and decoding the average coefficients take slightly longer than accessing uncompressed data since we still need to apply the scaling in the reconstruction formulas (8.3).

8.4 Chapter Summary

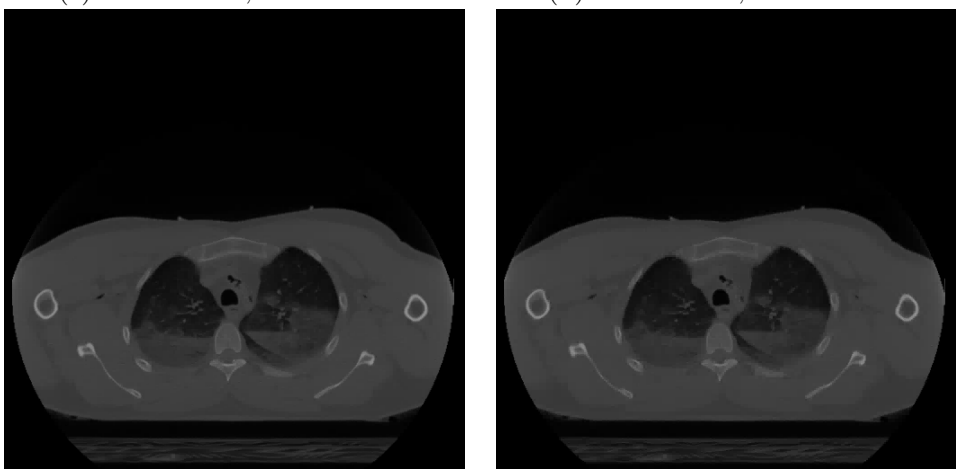
We have presented a new method for compression of volumetric data that supports fast random access to individual voxels within the compressed data. The method is based on a wavelet decomposition of the data, and a new technique for storing the significant coefficients and their significance map using hashing. The new hashing technique may be of independent use in other compression applications. Results comparing our new compression method to other techniques show an improvement over the entire distortion range investigated. For high ratios, the improvement in ratio is up to 80%. We also reported competitive access times for our method.



(a) Original (b) Ratio: 15:1, PSNR: 56.1

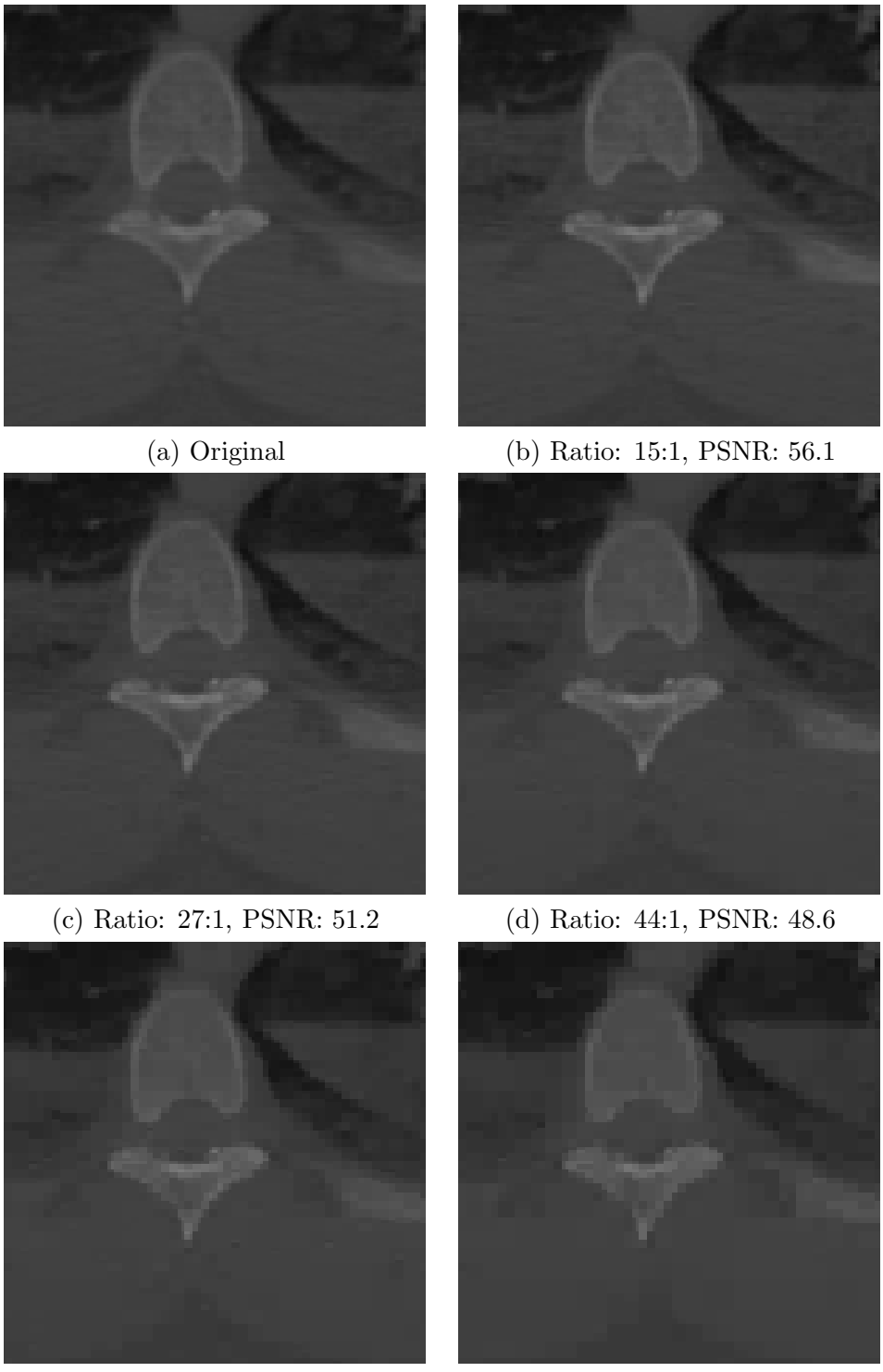


(c) Ratio: 27:1, PSNR: 51.2 (d) Ratio: 44:1, PSNR: 48.6



(e) Ratio: 80:1, PSNR: 45.8 (f) Ratio: 109:1, PSNR: 43.3

Figure 8.9: Sample slice (no. 345) of the Visible Man data set for various compression ratios.

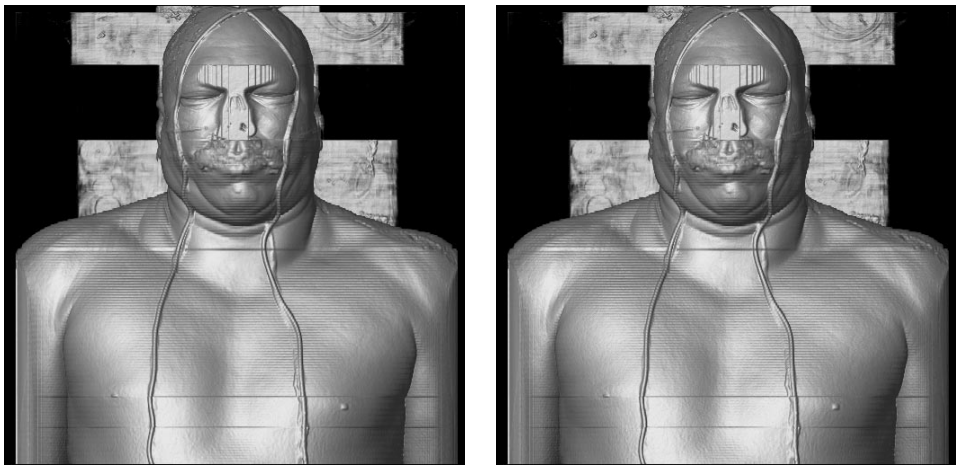


(a) Original (b) Ratio: 15:1, PSNR: 56.1

(c) Ratio: 27:1, PSNR: 51.2 (d) Ratio: 44:1, PSNR: 48.6

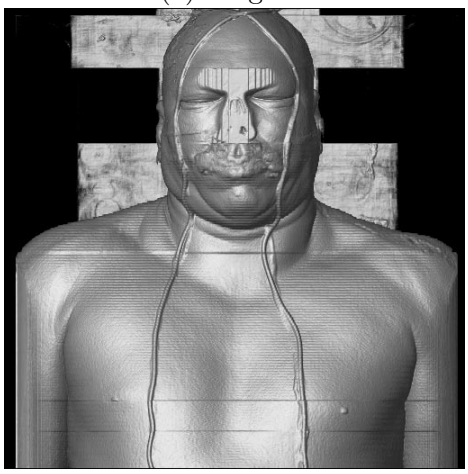
(e) Ratio: 80:1, PSNR: 45.8 (f) Ratio: 109:1, PSNR: 43.3

Figure 8.10: Zoom of sample slice (no. 345) of the Visible Man data set for various compression ratios.

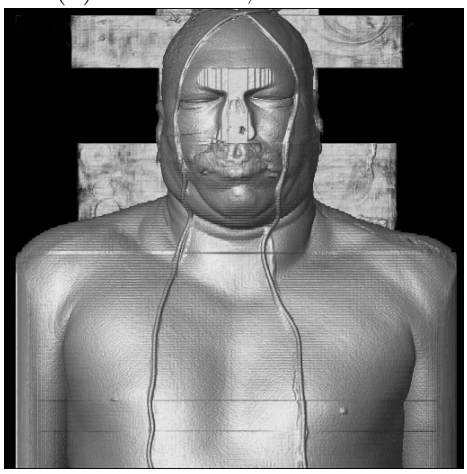


(a) Original

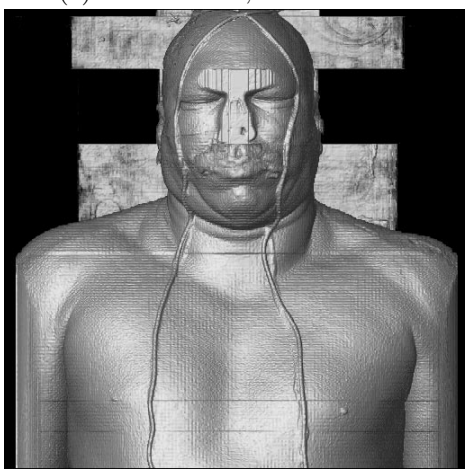
(b) Ratio: 15:1, PSNR: 56.1



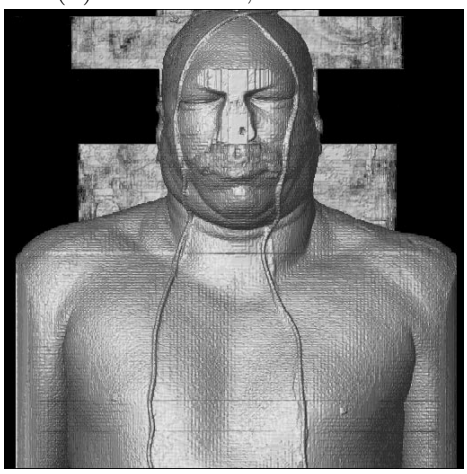
(c) Ratio: 27:1, PSNR: 51.2



(d) Ratio: 44:1, PSNR: 48.6

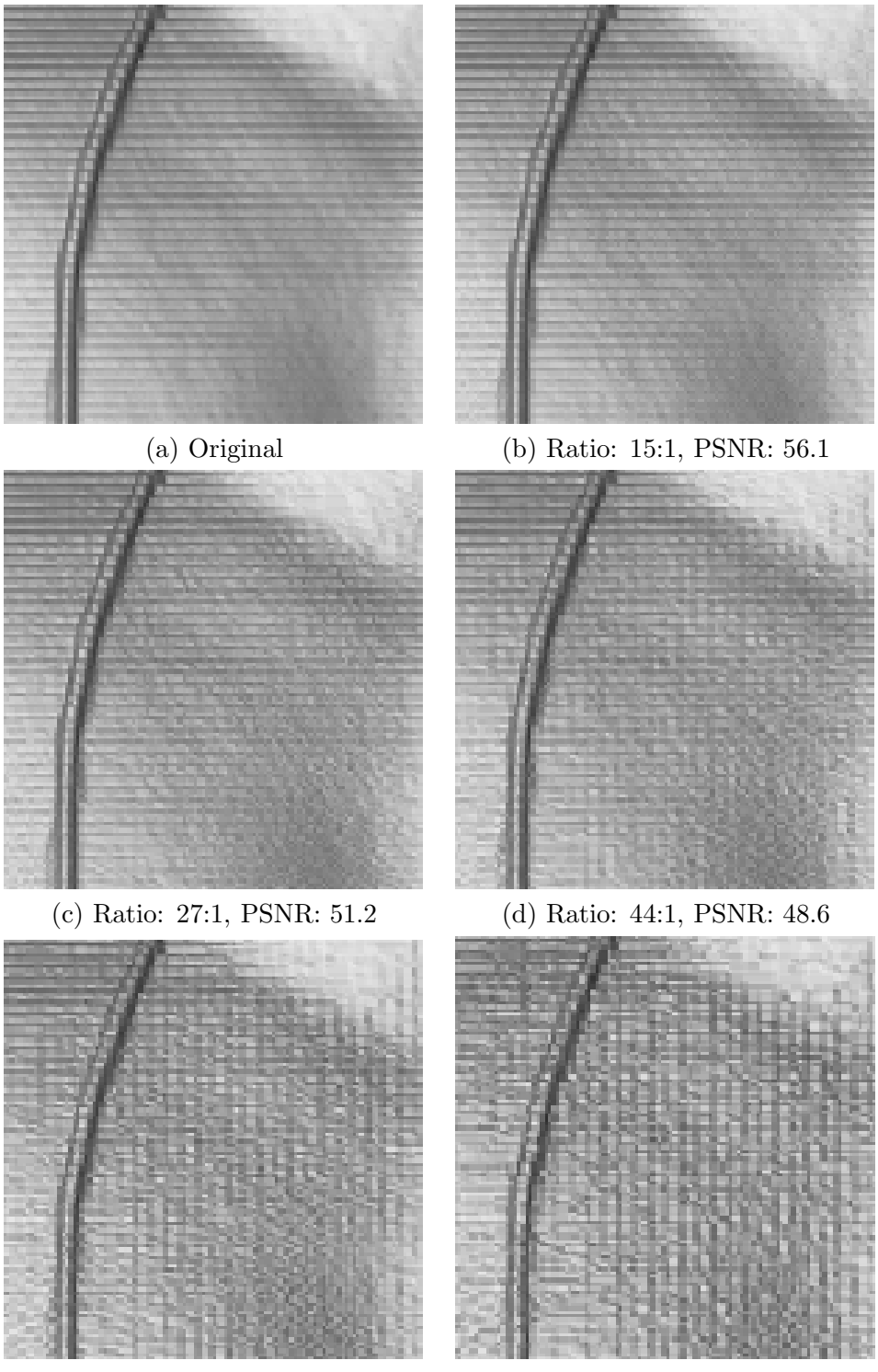


(e) Ratio: 80:1, PSNR: 45.8



(f) Ratio: 109:1, PSNR: 43.3

Figure 8.11: Rendered images of the Visible Man data set for various compression ratios.

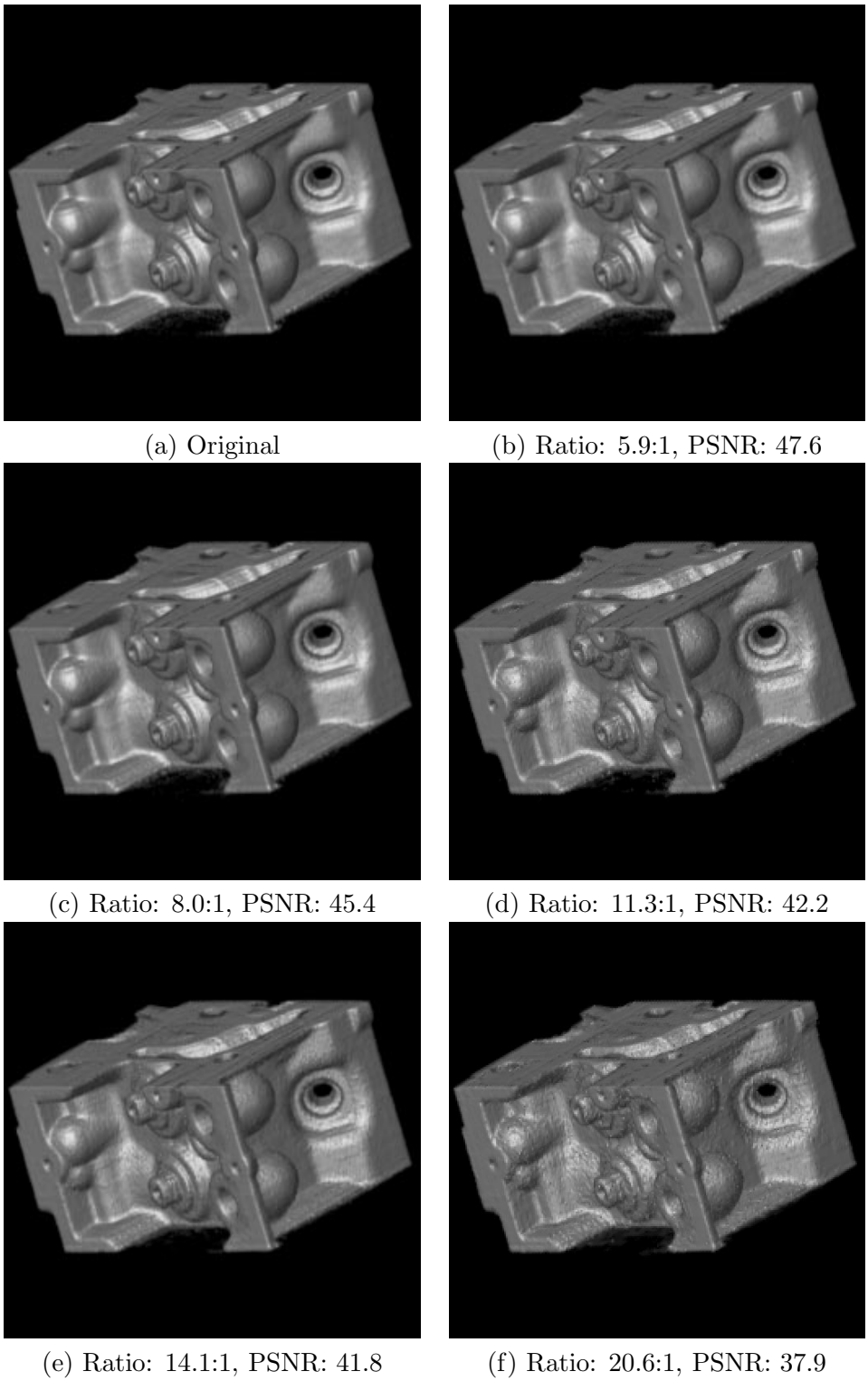


(a) Original (b) Ratio: 15:1, PSNR: 56.1

(c) Ratio: 27:1, PSNR: 51.2 (d) Ratio: 44:1, PSNR: 48.6

(e) Ratio: 80:1, PSNR: 45.8 (f) Ratio: 109:1, PSNR: 43.3

Figure 8.12: Zoom of rendered images of the Visible Man data set for various compression ratios.

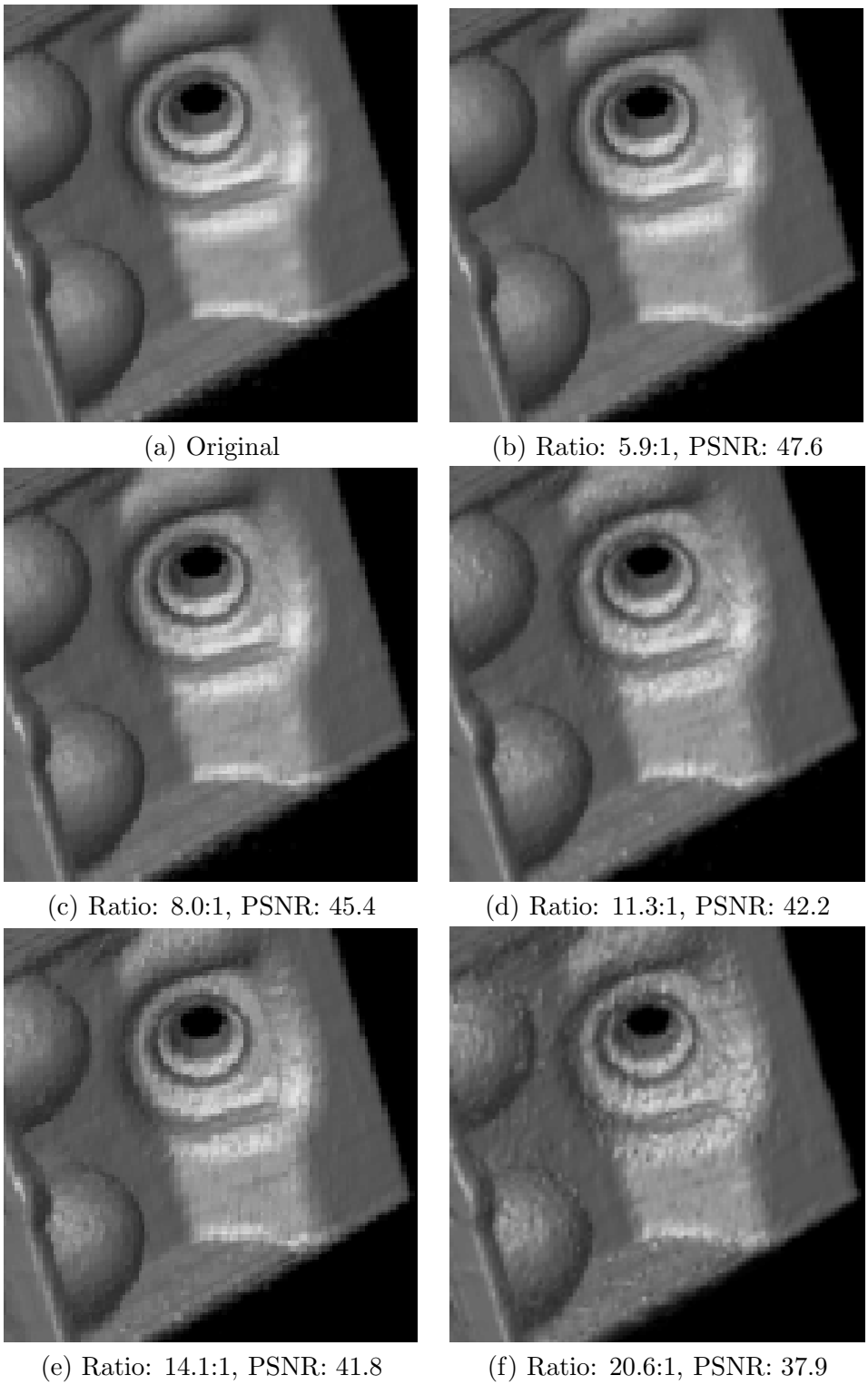


(a) Original (b) Ratio: 5.9:1, PSNR: 47.6

(c) Ratio: 8.0:1, PSNR: 45.4 (d) Ratio: 11.3:1, PSNR: 42.2

(e) Ratio: 14.1:1, PSNR: 41.8 (f) Ratio: 20.6:1, PSNR: 37.9

Figure 8.13: Rendered images of the Engine Block data set for various compression ratios.



(a) Original (b) Ratio: 5.9:1, PSNR: 47.6

(c) Ratio: 8.0:1, PSNR: 45.4 (d) Ratio: 11.3:1, PSNR: 42.2

(e) Ratio: 14.1:1, PSNR: 41.8 (f) Ratio: 20.6:1, PSNR: 37.9

Figure 8.14: Zoom of rendered images of the Engine Block data set for various compression ratios.

Chapter 9

Comparison

In this chapter we briefly compare the two volumetric compression methods with fast random access, presented in Chapter 7 and Chapter 8, to each other. Many aspects of the two methods, such as compression ratio and access speed, have already been compared in Chapter 8. However, Chapter 8 focused on comparing the two methods to methods in the published literature. In this chapter we focus on comparing the two methods with each other in relation to the design criteria mentioned in Chapter 6.

9.1 Fast Decoding for Random Access

In order to be of use in interactive applications our compression methods have to provide fast random access to voxels. Of the two methods presented, `Method 2` was the fastest. Depending on the compression ratios `Method 2` was about 3–6 times slower than accessing uncompressed data on the PC for the Visible Man data set. Considering that the Visible Man data set was too large to fit into memory on the computers used, we think this is a reasonable slowdown.

9.2 Good Visual Fidelity at High Compression Ratios

Achieving high compression ratios while maintaining good visual fidelity is very important. As volumes increase in size the memory requirements increase cubically. Therefore high compression ratios are vital, if volumetric applications and algorithms are to be run on computers with limited memory. On the other hand in many applications only a small degradation in the reconstruction quality is tolerable, e.g., care must be taken in medical imaging.

For the Visible Man data set `Method 1` and `Method 2` obtained compression ratios of up to 60:1 and 109:1 respectively for the same PSNR. The reconstruction error at these ratios, though noticeable in rendered images, might be acceptable for some applications. However, if only a small degradation in quality is allowed, the compression ratios have to be lowered. We found that for the same visual fidelity `Method 2` produced the best compression ratio, e.g, see Figure 7.13 and Figure 8.10 (the rendered images are not readily compared,

as two different volume rendering techniques were used). This is in good correspondence with the results obtained by using the PSNR as a measure of quality. In fact, since all methods for volumetric compression with fast random access currently use the Haar wavelet we believe that the PSNR is a good way of measuring and comparing the quality of the reconstructed volumes.

9.3 Multiresolutional Decoding

The timing results in Chapter 7.4.2 and Chapter 8.2.2 indicate that current methods might not be fast enough for real-time applications. One solution to this problem is to use coarser representations of the volume. For example, in volume rendering one could decode and render a downsampled version of the volume at interactive frame-rates and when time permits the renderer and decoding algorithm could switch to full resolution.

Because `Method 1` compresses volumes by employing temporal prediction it does not support scalable decoding in a natural way. However, since it uses a two-dimensional wavelet transform it scales better with respect to the number of wavelet coefficients necessary for reconstruction. The three-dimensional wavelet transform used by `Method 2` makes scalable decoding straightforward. Timing results show that for a downsampling factor of 8, accessing compressed data is almost as fast as accessing uncompressed data on the PC.

9.4 Selective Block-wise Decoding

The motivation throughout this dissertation has been fast random access to individual voxels. However, some applications access data locally. To enhance decoding efficiency in such cases both `Method 1` and `Method 2` support selective block-wise decoding.

At low PSNR levels `Method 2` is significantly faster than `Method 1` when decoding block-wise, while at high PSNR levels the two methods perform more equally. From the timing results in Chapter 8.2.2 we observe that accessing the complete volume block-wise is much faster than accessing all voxels in random order.

9.5 Online Compression

In systems with limited storage it is desirable if the volume data can be compressed while it is being downloaded or transferred to the system by other means. This kind of online compression puts limitations on the resources available to the encoder.

In order to compress a volume, `Method 1` only accesses a few slices at a time. This only requires a small buffer and `Method 1` therefore supports online compression. `Method 2`, on the other hand, makes multiple passes over the data. First a pass to compute the wavelet coefficients, then a pass to compute the table sizes by thresholding, and finally three passes to find the best hash functions for each subband. For this reason `Method 2` does not support online

compression. However, if the table sizes and hash functions are known a priory online compression is possible.

Chapter 10

Conclusions

10.1 Final Summary

Many algorithms for compressing volumetric data have been proposed in the last decade. However, most of the research focus has been on efficient transmission and storage of volumes. This only in part solves the problems associated with handling such large data sets. For one thing, keeping the volumes uncompressed in memory when they are used as input to other applications may only be possible on expensive high end computers with extensive amounts of memory. Recently though, the need for keeping the volumes compressed while they are accessed by other applications has been recognized, thereby introducing the challenge of supporting fast random access to the compressed data.

In this dissertation two methods (`Method 1` and `Method 2`) for volumetric compression with fast random access have been proposed. The methods improve on existing methods either by employing ideas from video coding or by combining techniques from subband coding with the new concept of lossy dictionaries introduced in this dissertation.

The two-dimensional "motion" compensating coder (`Method 1`) presented in Chapter 7 benefits from being able to code data in an online setting while offering high compression ratios and moderate access times. The second method (`Method 2`), presented in Chapter 8, which is based on a three-dimensional wavelet transform and a lossy dictionary data structure significantly improves the compression ratio compared to existing schemes while providing quite competitive random access times.

Performance tests show that `Method 2` yields an increase in compression ratio of about 80% for the same amount of distortion compared to `Method 1` at high ratios. Comparing to other existing methods the compression ratio is up to three times better. Furthermore, the time to access a random voxel is competitive with existing schemes. As the compression algorithm of `Method 2` makes several passes over the data, enough storage capacity must be available to store the whole volume uncompressed. However, `Method 1` can be used in an online setting using only a small buffer, where data is compressed as it is

downloaded or generated by, for example, a MR scanner.

In the subband coding literature the prevailing practice is to store the set of coefficients with largest magnitude. At low bitrates coding the positions of these coefficients, denoted the significance map, constitutes a significant portion of the total bitbuget. Many advanced schemes have been suggested to reduce this cost. One major reason for the success of `Method 2` is the idea that instead of storing the exact set of wavelet coefficients with largest magnitude, significant bit savings can be achieved by storing a slightly different set. This has also been noted in [91]. To that end we introduced the concept of lossy dictionaries as a tool for achieving efficient compression with fast random access.

The concept of lossy dictionaries is presented in Chapter 5. In the setting of lossy dictionaries, we are given a set of keys, each key associated with a weight. The task of a lossy dictionary is then to store as many keys as possible while maximizing the sum of weights of the keys stored under a given space constraint. In Chapter 5 we presented a lossy dictionary data structure and showed it to have very good behavior with respect to the set of keys stored. We furthermore proved our data structure to be nearly optimal with respect to the space used. Also, we have shown that our lossy dictionary is very well suited for lossy compression with fast random access. We believe that lossy dictionaries might find use in other applications as well. For example, they might be used in web cache sharing and differential files.

The efficient lossy dictionary data structure that we described in Chapter 5 was built on the CUCKOO HASHING scheme presented in Chapter 4. CUCKOO HASHING is a new dynamic dictionary with worst case constant lookup time and amortized expected constant time for updates. Besides being simple to implement, it has average case running times comparable to several well known and often used dictionary methods. These methods have nontrivial worst case lookup time, which makes CUCKOO HASHING useful in time critical applications where such a guarantee is necessary. The typical space usage of CUCKOO HASHING is three words per key. This is similar to binary search trees.

10.2 Future Directions

In this work we have presented methods for improving both compression ratio and random access time for wavelet compressed volumes. This is an important step both towards making volumetric data processing less memory demanding, but also it paves the way for generating and using even larger volumes. However, several challenges remain. In the following, we list some open problems for the four major chapters (Chapter 4, 5, 7, and 8) of this dissertation. Finally, we provide some interesting ideas for future research in lossy compression with fast random access including possible improvements to the methods given in this dissertation.

10.2.1 Cuckoo Hashing

First of all, an explicit practical hash function family that is provably good for the dictionary scheme has yet to be found. Secondly, we lack a precise understanding of why the scheme exhibits low constant factors. In particular, the curve of Figure 4.6 and the fact that forced rehashes are rare for load factors quite close to 1/2 need to be explained. Another point to investigate is whether using more than two tables yields practical dictionaries. Experiments in Chapter 5 suggest that space utilization could be improved to more than 80% using three tables. However, it remains to be seen how this would affect insertion performance.

10.2.2 Lossy Dictionaries

For lossy dictionaries some of the same challenges as for CUCKOO HASHING remain. Though simple and efficient hash functions seem to work well in practice with our lossy dictionary, the challenge of finding such families that provably work well remains. Furthermore, the last two graphs in Figure 5.6 are not completely understood. Especially, the theoretical lower bound shown in the figure must be improved to match the statistically determined bound. Also, it is not completely understood why the insertion heuristic for three or more tables works so well.

10.2.3 Coding using Motion Estimation and Blocking

Some aspects of `Method 1` need further research and attention. For example, we only employ a very simple prediction scheme. By using a more advanced prediction schemes we might be able to reduce the number of slices that is coded without temporal prediction and thereby increase the compression ratio. Block matching might at first seem attractive because of its simplicity when decoding. However, we believe that the correlation of volumetric data will be captured better by, for example, a parametric motion model that captures zoom well. Also, how other wavelet types will affect compression ratio and quality must be evaluated. Since our method only performs a two-dimensional wavelet transform it scales better than using a three-dimensional transform, with respect to the number of coefficients that is needed for reconstruction when the wavelet filters become longer. This is attractive since the number of coefficients that must be extracted directly affects decoding speed.

10.2.4 Coding using Three-Dimensional Wavelets and Hashing

One interesting idea for future research in lossy compression is the concept of not storing the most significant wavelet coefficients but a slightly different set of coefficients. This might yield a combined rate-distortion improvement when used in combination with existing techniques. The idea has been mentioned before by Xiong et al. [91] but it has still not attracted the attention we think it deserves.

10.2.5 General Challenges in Compression with Fast Random Access.

Of course, a general challenge in compression with fast random access is to further improve compression ratio and access time. Another important step forward would be to consider the compression of color volumes. The first attempt at compressing color volumes with fast random access was reported by Bajaj et al. in [4]. Basically, the authors suggest to vector quantize the wavelet coefficients of the individual RGB color components. The quantization indices are then coded using a method very similar to the method given in [38]. Compression ratios of up to 80:1 were reported. Since our `Method 2` already outperforms [38] providing compression ratios of up to 109:1, we believe it to be a good platform for further color compression experiments.

Finally, to make our compression method useful in an interactive visualization environment, decoding speed must be improved, either by developing an efficient cache structure temporary holding voxel values or by adding redundancy to the data structure to decrease lookup overhead. However, to be of real value this would require special purpose designed data structures for the particular rendering method, which has not been the focus of this dissertation.

Bibliography

[1] Bruno Aiazzi, Pasquale Alba, Luciano Alparone, and Stefano Baronti. Lossless compression of multi/hyper-spectral imagery based on a 3-D fuzzy prediction. *IEEE Transactions on Geoscience and Remote Sensing*, 37(5):2287–2294, 1999.

[2] Bruno Aiazzi, Pasquale Alba, Stefano Baronti, and Luciano Alparone. Three-dimensional lossless compression based on a separable generalized recursive interpolation. *Proceedings of the 1996 IEEE International Conference on Image Processing*, pages 85–88, 1996.

[3] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200 (electronic), 1999.

[4] Chandrajit Bajaj, Insung Ihm, and Sanghun Park. 3D RGB image compression for interactive applications. *ACM Transactions on Graphics*, 20, March 2001. To appear.

[5] Atilla M. Baskurt, Hugues Benoit-Cattin, and Christophe Odet. A 3-D medical image coding method using a separable 3-D wavelet transform. In Yongmin Kim, editor, *Medical Imaging 1995: Image Display*, pages 184–194. Proceedings of SPIE 2431, 1995.

[6] A. Bilgin, G. Zweig, and M. W. Marcellin. Three-dimensional image compression using integer wavelet transforms. *Applied Optics: Information Processing, Special Issue on Information Theory in Optoelectronic Systems*, 39:1799–1814, April 2000.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 43–53. ACM Press, New York, 2000.

[9] Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. To appear in INFOCOM, 2001.

[10] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640 (electronic), 1999.

[11] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, New York, 2000.

[12] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31(4):532–540, April 1983.

[13] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, New York, 1978.

[14] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[15] A. Cohen, I. Daubechies, and P. Vial. Wavelet bases on the interval and fast algorithms. *Journal of Applied and Computational Harmonic Analysis*, 1(12):54–81, 1993.

[16] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial optimization*. John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.

[17] Ingrid Daubechies. *Ten Lectures on Wavelets*. SIAM, Philadelphia, Pennsylvania, 1992.

[18] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS '96)*, pages 569–580. Springer-Verlag, Berlin, 1996.

[19] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, Berlin, 1992.

[20] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[21] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[22] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of*

the 17th International Colloquium on Automata, Languages and Programming (ICALP '90), volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, Berlin, 1990.

[23] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[24] *The Engine Block data set.* http://www9.informatik.uni-erlangen.de/~cfrezksa/volren/.

[25] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[26] G. Fernández, S. Periaswamy, and Wim Sweldens. LIFTPACK: A software package for wavelet transforms using lifting. In M. Unser, A. Aldroubi, and A. F. Laine, editors, *Wavelet Applications in Signal and Image Processing IV*, pages 396–408. Proceedings of SPIE 2825, 1996.

[27] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the Association for Computing Machinery*, 31(3):538–544, 1984.

[28] Didier J. Le Gall. The MPEG video compression algorithm. *Signal Processing: Image Communication*, 4:129–140, 1992.

[29] Mohammad H. Ghavamnia and Xue D. Yang. Direct rendering of laplacian pyramid compressed volume data. *Proceedings of Visualization '95*, pages 192–199, October 1995.

[30] Gaston Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley Publishing Co., London, 1984.

[31] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Computer Graphics, Proceedings of SIGGRAPH '96*, pages 43–54, 1996.

[32] Markus H. Gross. L^2 optimal oracles and compression strategies for semiorthogonal wavelets. Technical Report 254, Computer Science Department, ETH Zürich, December 1996.

[33] Roberto Grosso, Thomas Ertl, and Joachim Aschoff. Efficient data structures for volume rendering of wavelet-compressed data. *WSCG '96 – The Fourth International Conference in Central Europe on Computer Graphics and Visualization*, February 1996.

[34] A. Haar. Zur Theorie der orthogonalen Funktionensysteme. *Mathematische Annalen.*, 69:331–371, 1910.

[35] W. Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 43:172–198, 1927.

[36] Michael Hoetter. Differential estimation of the global motion parameters zoom and pan. *Signal Processing*, 16:249–265, 1989.

[37] Insung Ihm and Sanghun Park. Wavelet-based 3D compression scheme for very large volume data. In *Graphics Interface*, pages 107–116, June 1998.

[38] Insung Ihm and Sanghun Park. Wavelet-based 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, March 1999.

[39] ISO/IEC JTC1 and ITU-T. Digital compression and coding of continuous-tone still images. *ITU-T Recommendation T.81 – ISO/IEC 10918-1 (JPEG)*, 1992.

[40] ISO/IEC JTC1 and ITU-T. Information technology – generic coding of moving pictures and associated audio information – part 2: Video. *ITU-T Recommendation H.262 – ISO/IEC 13818-2 (MPEG-2)*, 1994.

[41] ISO/IEC JTC1 and ITU-T. Information technology – JPEG 2000 image coding system. *JPEG 2000 final committee draft version 1.0*, March 2000.

[42] ITU-T. Video coding for low bit rate communication. *ITU-T Recommendation H.263*, version 1, Nov 1995; version 2, Jan. 1998.

[43] Stasys Jukna. *Extremal Combinatorics with Applications in Computer Science*. Springer-Verlag, 2000.

[44] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

[45] Jyrki Katajainen and Michael Lykke. Experiments with universal hashing. Technical Report DIKU Report 96/8, University of Copenhagen, 1996.

[46] Tae-Young Kim and Yeong Gil Shin. An efficient wavelet-based compression method for volume rendering. In *Proceedings of the Seventh Pacific Conference on Computer Graphics and Applications*, pages 147–156, Seoul, Korea, October 1999.

[47] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Massachusetts, second edition, 1998.

[48] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH 94*, pages 451–458, 1994.

[49] K. Lau, W. Vong, and W. Ng. Lossless compression of 3-D images by variable predictive coding. *Proceedings of 2nd Singapore International Conference on Image Processing*, pages 161–165, 1992.

[50] Marc Levoy and Pat Hanrahan. Light field rendering. *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '96)*, pages 31–42, 1996.

[51] Jiebo Luo, Xiaohui Wang, Chang W. Chen, and Kevin J. Parker. Volumetric medical image compression with three-dimensional wavelet transform and octave zerotree coding. In Rashid Ansari and Mark J. Smith, editors, *Visual Communications and Image Processing '96*, pages 579–590. Proceedings of SPIE 2727, 1996.

[52] Stéphane Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans on Pattern Analysis and Machine Intelligens*, 11(7):674–693, July 1989.

[53] Stéphane Mallat. *A Wavelet Tour of Signal Processing 2nd Edition*. Academic Press, 1999.

[54] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[55] Catherine C. McGeoch. The fifth DIMACS challenge – dictionary tests. http://cs.amherst.edu/∼ccm/challenge5/dicto/.

[56] Kurt Mehlhorn and Stefan Näher. *LEDA. A platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, 1999.

[57] Shigeru Muraki. Approximation and rendering of volume data using wavelet transforms. *Proceedings of Visualization '92*, pages 21–28, October 1992.

[58] Shigeru Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics & Applications*, 13(4):50–56, July 1993.

[59] Gregory M. Nielson, Hans Hagen, and Heinrich Müller, editors. *Scientific Visualization: Overviews, Methodologies, and techniques*. IEEE Computer Society, 1997.

[60] Paul Ning and Lambertus Hesselink. Fast volume rendering of compressed data. *Proceedings of Visualization '93*, pages 11–18, October 1993.

[61] Rasmus Pagh. Low redundancy in static dictionaries with $O(1)$ lookup time. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604. Springer-Verlag, Berlin, 1999.

[62] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 425–432. ACM Press, New York, 2001.

[63] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Technical Report RS-01-32, BRICS, August 2001.

[64] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133, Berlin, 2001. Springer-Verlag.

[65] Rasmus Pagh and Flemming Friche Rodler. Lossy dictionaries. Technical Report RS-01-33, BRICS, August 2001.

[66] Rasmus Pagh and Flemming Friche Rodler. Lossy dictionaries. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 300–311, Berlin, 2001. Springer-Verlag.

[67] C. A. Papadopoulos and Trevor G. Clarkson. Motion compensation using second-order geometric transformations. *IEEE Transactions. on Circuits and Systems for Video Technology*, Vol. 5:319–331, August 1995.

[68] S.M. Perlmutter, P.C. Cosman, R.M. Gray, R.A. Olshen, D. Ikeda, C.N. Adams, B.J. Betts, M. Williams, K.O. Perlmutter, J. Li, A. Aiyer, L. Fajardo, R. Birdwell, and B.L. Daniel. Image quality in lossy compressed digital mammograms. *Signal Processing*, 52(2):189–210, 1997.

[69] Michael A. Pratt, Chee-Hung H. Chu, and Stephen T. Wong. Volume compression of MRI data using zerotrees of wavelet coefficients. In Michael A. Unser, Akram Aldroubi, and Andrew F. Laine, editors, *Wavelet Applications in Signal and Image Processing IV*, pages 752–763. Proceedings of SPIE 2431, 1996.

[70] Flemming Friche Rodler. Wavelet based 3D compression for very large volume data supporting fast random access. Technical Report RS-99-34, BRICS, October 1999.

[71] Flemming Friche Rodler. Wavelet based 3D compression with fast random access for very large volume data. In *Proceedings of the Seventh Pacific Conference on Computer Graphics and Applications*, pages 108–117, Seoul, Korea, 1999.

[72] Flemming Friche Rodler and Rasmus Pagh. Fast random access to wavelet compressed volumetric data using hashing. To Appear in ACM Transactions on Graphics.

[73] Flemming Friche Rodler and Rasmus Pagh. Fast random access to wavelet compressed volumetric data using hashing. Technical Report RS-01-34, BRICS, August 2001.

[74] Khalid Sayood. *Introduction to Data Compression.* Morgan Kaufmann Publishers, Inc., USA, 1996.

[75] Dennis S. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large data bases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

[76] Jerome M. Shapiro. Embedded image coding using zerotress of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[77] Heung-Yeung Shum and Li-Wei He. Rendering with concentric mosaics. *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH '99)*, pages 299–306, 1999.

[78] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20–25. IEEE Computer Society Press, Los Alamitos, CA, 1989.

[79] Craig Silverstein. A practical perfect hashing algorithm. Manuscript, 1998.

[80] Wim Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and Computational Harmonic Analysis*, 3(2):186–200, 1996.

[81] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22:215–225, 1975.

[82] A. M. Tekalp. Digital video processing. *Prentice Hall Signal Processing Series*, 1995.

[83] Jo Yew Tham, Surendra Ranganath, and Ashraf A. Kassim. Scalable very low bit-rate video compression using motion compensated 3-D wavelet decomposition. *IEEE ISPACS Workshop*, 3:38.7.1–38.7.5, November 1996.

[84] Geoge R. Thoma and L. Rodney Long. Compressing and transmitting visible human images. *IEEE Multimedia*, 4(2):36–45, 1997.

[85] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 496–497. ACM Press, New York, 2000.

[86] The visible human project. The National Library of Medicine (NLM) homepage – http://www.nlm.nih.gov/research/visible/visible_human.html.

[87] The volpack volume rendering library. Stanford Computer Graphics Laboratory. http://graphics.stanford.edu/software/volpack/.

[88] Volvis 2.1. Visualization Laboratory of the Department of Computer Science at the State University of New York at Stony Brook. http://www.cs.sunysb.edu/∼vislab/volvis_home.html.

[89] J. Wang and H. Huang. Medical image compression by using three-dimensional wavelet transformation. *IEEE Transactions on Medical Imaging*, 15:547–554, August 1996.

[90] M. Wenzel. Wörterbücher für ein beschränktes Universum. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992.

[91] Xixiang Xiong, Kannan Ramchandran, and Michael T. Orchard. Space-frequency quantization for wavelet image coding. *IEEE Transactions on Image Processing*, 6(5):677–693, May 1997.

Recent BRICS Dissertation Series Publications

DS-01-9 **Flemming Friche Rodler.** ***Compression with Fast Random Access*****. November 2001. PhD thesis. xiv+124 pp.**

DS-01-8 **Niels Damgaard.** ***Using Theory to Make Better Tools*****. October 2001. PhD thesis.**

DS-01-7 **Lasse R. Nielsen.** ***A Study of Defunctionalization and Continuation-Passing Style*****. August 2001. PhD thesis. iv+280 pp.**

DS-01-6 **Bernd Grobauer.** ***Topics in Semantics-based Program Manipulation*****. August 2001. PhD thesis. ii+x+186 pp.**

DS-01-5 **Daniel Damian.** ***On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*****. August 2001. PhD thesis. xii+111 pp.**

DS-01-4 **Morten Rhiger.** ***Higher-Order Program Generation.*** **August 2001. PhD thesis. xiv+144 pp.**

DS-01-3 **Thomas S. Hune.** ***Analyzing Real-Time Systems: Theory and Tools*****. March 2001. PhD thesis. xii+265 pp.**

DS-01-2 **Jakob Pagter.** ***Time-Space Trade-Offs*****. March 2001. PhD thesis. xii+83 pp.**

DS-01-1 **Stefan Dziembowski.** ***Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*****. January 2001. PhD thesis. 109 pp.**

DS-00-7 **Marcin Jurdziński.** ***Games for Verification: Algorithmic Issues*****. December 2000. PhD thesis. ii+112 pp.**

DS-00-6 **Jesper G. Henriksen.** ***Logics and Automata for Verification: Expressiveness and Decidability Issues.*** **May 2000. PhD thesis. xiv+229 pp.**

DS-00-5 **Rune B. Lyngsø.** ***Computational Biology*****. March 2000. PhD thesis. xii+173 pp.**

DS-00-4 **Christian N. S. Pedersen.** ***Algorithms in Computational Biology*****. March 2000. PhD thesis. xii+210 pp.**

DS-00-3 **Theis Rauhe.** ***Complexity of Data Structures (Unrevised)*****. March 2000. PhD thesis. xii+115 pp.**