31st Annual International Conference of the IEEE EMBS
Minneapolis, Minnesota, USA, September 2-6, 2009

Optimized bzip2 Compression for Reducing Diffraction Effects in Protein-based Computing: A Study of Feasibility

Dragoş Trincă and Sanguthevar Rajasekaran

Abstract— **One of the current research directions in biological nanotechnology is the use of bacteriorhodopsin in the fabrication of protein-based photonic devices. Bacteriorhodopsin, with its unique light-activated photocycle, nanoscale size, cyclicity (> 107), and natural resistance to harsh environmental conditions, provides for protein-based memories that have a comparative advantage over magnetic and optical data storage devices. However, the construction of protein-based memories has been severely limited by fundamental issues that exist with such devices, such as unwanted diffraction effects. In this paper, we propose an algorithm for coping with diffraction effects, thus providing a solution to a long-standing problem.**

I. INTRODUCTION

The field of nanotechnology has responded to the challenges posed by Moore's law by measuring, modeling, and fabricating materials no larger than one thousandth of a micron (nanometer). At this size, techniques in nanolithography must account for the thermodynamic effects that accompany complex molecular architectures. To cope with the high error rates associated with these thermal side effects, fault-tolerant designs are often used in fabricating such devices. The caveat to using these designs is that they rarely match the high level of functional complexity that is observed in biological machinery. As a result, biological nanotechnology has emerged as an appealing alternative to methods in macroscopic miniaturization.

Much of the current research effort in biological nanotechnology is directed toward self-assembled monolayers and thin films, biosensors, and protein-based photonic devices [1], [2], [6]. Although a number of proteins have been explored for device applications, bacteriorhodopsin [5] has received the most attention. This protein, with its unique light-activated photocycle, nanoscale size, cyclicity (> 107), and natural resistance to harsh environmental conditions, provides for protein-based memories that have a comparative advantage over magnetic and optical data storage devices. In addition, bacteriorhodopsin protein memory devices exhibit increased thermal, chemical and photochromic stability, and have the advantage of being portable, radiation-hardened, waterproof, and EMP-resistant. Such devices are capable of storing large amounts of data in a small volume. There are a number of potential applications of these systems which can leverage either the large memory or the associative memory capabilities. For instance, the photo or fingerprints of a suspect in a crime can be matched against a database of photos or fingerprints of known criminals. To accomplish this task quickly, a massive degree of parallelism is called for. Protein-based associative memory processors (PBAMPs) offer this parallelism and image (fingerprint) matching can potentially be done in real time. Indeed, prototype PBAMPs are currently being used for matching fingerprints and other images.

Although there are some prototype systems and preliminary effort to apply them, the potential of this promising technology is relatively unexplored. Research on protein-based memories started in the late 1980s with considerable anticipation, but enthusiasm decreased rapidly for several reasons. Commercial development of spatial light modulators (SLMs), that are an integral part of protein-based memories, was slow and there were fundamental issues, such as unwanted diffraction effects, that limited performance in three-dimensional memory applications. More recently, however, the development of high-definition television projection equipment has resulted in the commercial availability of high-resolution, high-performance and relatively inexpensive SLMs. Nonetheless, fundamental problems remained. Two such problems are diffraction effects and scaling.

A. Diffraction Effects

The branched-photocycle three-dimensional memory stores data by using a sequential two-photon process to convert bacteriorhodopsin (bR) in the activated region from the bR resting state to the Q state. The process involves using a paging beam to select a thin page of memory and a writing beam that is pixilated in those positions where data are to be written. The transition from the bR resting state to the Q state occurs via the intermediate states K, L, M, N, and O. Only the bR (bit 0) and the Q (bit 1) states are stable for extended periods of time. By using 32-level grey-scaling and two polarizations, each voxel can store 64 bits. Attempts to use higher levels of grey-scaling have failed due in large part to the problems of diffraction introduced by having pages with significant differences in the average refractive indices. To understand this problem, we note that protein representing bit 0 has a high refractive index and protein representing bit 1 has a low refractive index with reference to the red laser beam at $633\ nm$ that is used to read out the data. Prototypes made of the three-dimensional memory fail to operate at high storage densities when individual pages of memory have a preponderance of bits of a given state. Consider the worst case scenario – each page is either all

This work was supported by the NSF Grant EMT/NANO-0829916.

D. Trincă is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA dtrinca@engr.uconn.edu

S. Rajasekaran is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA rajasek@engr.uconn.edu

0's or all 1's. Then we create a refractive index grating that diffracts the laser beams quite efficiently because individual pages are stored with separations of $6-20\ \mu m$. While these separations are much larger than the diffraction limit would dictate, closer spacing is impossible due to beam steering inside the data cuvettes. The unwanted beam steering is due to refractive index gradients. Algorithms that can store data at high resolution and that maintain the number of 0's equal to the number of 1's would solve this problem.

B. Previous Work on Reducing Diffraction Effects

One way of ensuring an equal number of zeros and ones is to replace a zero with 01 and a one with 10. However, this would reduce the available memory by a factor of 2 (which means that the *utility factor* would be 50% in this case) and hence may not be preferred. In [10], the authors have proposed some methods with corresponding utility factors of between 95-99$\%$. In this paper, we are interested in methods that would achieve utility factors of 100% or even more.

II. Optimized bzip2 compression for reducing diffraction effects

Probably the best way to achieve utility factors of 100% or more would be to use a data compression algorithm. (However, simply applying a data compression algorithm may not solve the problem, since, in the resulting output, the number of 0's may not equal the number of 1's; so, the output would need to be further processed in such a way that the number of 0's would equal the number of 1's.) One of the fastest and most effective data compression algorithms currently in use is bzip2 [4]. The basic scheme under bzip2 is as follows: apply the Burrows-Wheeler transform [3], then apply the Move-to-Front (MTF) transform, and finally apply Huffman coding [7]. Based on this scheme, we consider the following algorithm for reducing diffraction effects.

Step 1.Apply the Burrows-Wheeler transform.
Step 2.Apply the Move-to-Front (MTF) transform.
Step 3.Apply Huffman coding.
Step 4.The corresponding output is further processed by appending a number of 0's (or a number of 1's) in such a way that, in the resulting final output, the number of 0's equals the number of 1's. (Definitely, we also need to specify in the final output how many 0's or 1's have been appended in this step.)

Denote this algorithm by ALG1. (The idea under ALG1 was already suggested in [9].) Since the first three steps are very effective in compression, we expect the additional 0's or 1's used in the last step of ALG1 to still allow for utility factors of 100% or more.

(Data is stored in protein-based memories in binary format. However, as input to ALG1, we may consider this binary format or its equivalent byte-by-byte format. No matter how the input to ALG1 is represented, the output of ALG1 is still binary.)

A. ALG2: Optimizing ALG1 by means of Quadratic Programming

The output in the Huffman algorithm (the third step in ALG1) is a binary string $H = H_1H_2$ (the concatenation of H_1 and H_2), where H_1 is a binary string representing the compressed input, and H_2 is the binary representation of the Huffman tables used during the Huffman algorithm. (H_2 is needed at decompression.)

For most files that are compressed in practice, the length of H_1 accounts for more than 99% of the length of H. The input to the fourth step in ALG1 is H. So, for most files that are compressed in practice, if H_1 consists of an average number of 0's and 1's, then we don't need to append too many bits in the fourth step of ALG1. Otherwise, the number of bits appended in the fourth step of ALG1 may be significant.

In this section, our aim is to optimize the H_1 part. (By an optimized H_1 part, we mean an H_1 part in which the number of 0's is equal to the number of 1's, or as close as possible to that in case that is not possible.) Note that the Huffman algorithm is a randomized algorithm, in the sense that some decisions during the algorithm are randomly taken. So, our aim is to see what decisions should be taken during Huffman coding in order for H_1 to have an equal number of 0's and 1's. Let us take an example.

Example 1: Let $X = (a, a, a, a, b, b, c, c, d, e)$ be an input to the Huffman algorithm. The vector of frequencies F for X is $F = (4, 2, 2, 1, 1)$. Applying the Huffman algorithm to X might lead to the Huffman tree shown in Fig. 1.

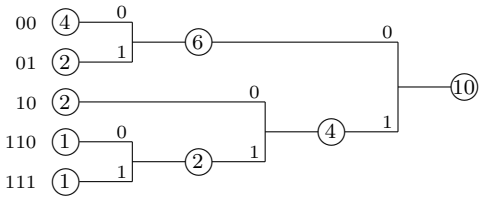

Fig. 1. A possible Huffman tree for X

Thus, if we use the Huffman tree shown in Fig. 1, then the codeword associated to a would be 00, the codeword associated to b would be 01, the codeword associated to c would be 10, the codeword associated to d would be 110, and the codeword associated to e would be 111. In such a case, H_1 would be 00000000010110101101111 (H_1 being obtained from X by replacing each symbol in X with its corresponding codeword). So, H_1 would have 13 0's and 9 1's. However, the Huffman tree that can be associated to X is not unique. Another possibility would be the Huffman tree shown in Fig. 2.

Note that the branch $6 \leftrightarrow 10$ in Fig. 1 has associated bit 0, while the branch $4 \leftrightarrow 10$ has associated bit 1. The Huffman tree shown in Fig. 2 is obtained from the Huffman tree shown in Fig. 1 by swapping these two bits. (Note that if N is a node in a Huffman tree and D_1, D_2 its immediate descendants, then we can assign bit 0 to the branch $D_1 \leftrightarrow N$ and bit 1 to the branch $D_2 \leftrightarrow N$, or, we can assign bit 1 to

Minimize: N_{zeros}^2+N_{ones}^2

S.t.: $4*(1\text{-}x_1)+4*(1\text{-}x_2)+2*x_1+2*(1\text{-}x_2)+2*(1\text{-}x_3)+2*x_2+1*(1\text{-}x_4)+1*x_3+1*x_2+1*x_4+1*x_3+1*x_2=N_{zeros}$

$4*x_1+4*x_2+2*(1\text{-}x_1)+2*x_2+2*x_3+2*(1\text{-}x_2)+1*x_4+1*(1\text{-}x_3)+1*(1\text{-}x_2)+1*(1\text{-}x_4)+1*(1\text{-}x_3)+1*(1\text{-}x_2)=N_{ones}$

$x_1,\ x_2,\ x_3,\ x_4 \in \{0,1\}$

$N_{zeros},\ N_{ones}$ being positive integers

Fig. 3. A quadratic programming formulation for finding the best assignation of variables (for Huffman trees whose structure is the one shown in Figs. 1 and 2)

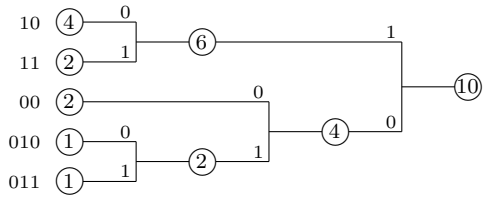

Fig. 2. Another possible Huffman tree for X

the branch $D_1 \leftrightarrow N$ and bit 0 to the branch $D_2 \leftrightarrow N$; the decision is randomly taken for each pair of branches once the tree is already built.) So, if we use the Huffman tree shown in Fig. 2, then the codeword associated to a would be 10, the codeword associated to b would be 11, the codeword associated to c would be 00, the codeword associated to d would be 010, and the codeword associated to e would be 011. In this case, H_1 would be 10101010111110000010011, and would have 11 0's and 11 1's, that is, the number of 0's would be equal to the number of 1's. Thus, if the Huffman tree shown in Fig. 2 is used to compress X, then H_1 would be optimized.

(end of *Example 1*.)

We have seen in Example 1 that for the same structure of the Huffman tree, an optimized H_1 might be obtained or not, depending on how bits are assigned at each pair of branches. For a Huffman tree with n pairs of branches, there are 2^n possibilities, which means that an exhaustive search for the best assignation of bits at the branches is not a good idea in practice.

We propose to solve this problem by means of quadratic programming, as follows. If N is a node in the Huffman tree and D_1, D_2 its immediate descendants, then the branches $D_1 \leftrightarrow N$ and $D_2 \leftrightarrow N$ forms a pair of branches, denoted by $(D_1 \leftrightarrow N, D_2 \leftrightarrow N)$. If the Huffman tree is represented as in Figs. 1 and 2, then each pair of branches consists of an upper branch and a lower branch. For example, the Huffman tree shown in Fig. 1 has four pairs of branches. For the pair of branches $(6 \leftrightarrow 10, 4 \leftrightarrow 10)$, $6 \leftrightarrow 10$ is the upper branch, while $4 \leftrightarrow 10$ is the lower branch. We can associate a binary variable to each pair of branches in the Huffman tree. A value of 0 for the binary variable would correspond to the case when the upper branch has been assigned bit 0 and the lower branch bit 1, while a value of 1 would correspond to the case when the upper branch has been assigned bit 1 and the lower branch bit 0. Thus, trying to optimize H_1 reduces to trying to find the best assignation of variables once the Huffman tree structure is already built.

For example, consider the Huffman tree shown in Fig. 1. If the pair of branches $(4 \leftrightarrow 6, 2 \leftrightarrow 6)$ has associated the binary variable x_1, the pair of branches $(6 \leftrightarrow 10, 4 \leftrightarrow 10)$ has associated the binary variable x_2, the pair of branches $(2 \leftrightarrow 4, 2 \leftrightarrow 4)$ has associated the binary variable x_3, and the pair of branches $(1 \leftrightarrow 2, 1 \leftrightarrow 2)$ has associated the binary variable x_4, as in Fig. 4, then if the bits are assigned to the branches as in Fig. 1, we would have $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$.

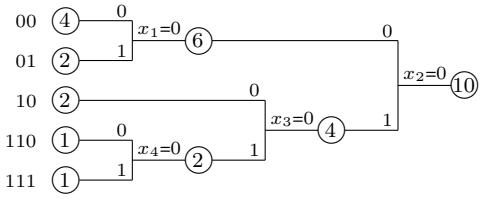

Fig. 4. The Huffman tree shown in Fig. 1 with the corresponding assignation of variables

For the Huffman tree shown in Fig. 2, the assignation of variables would be $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$, as in Fig. 5.

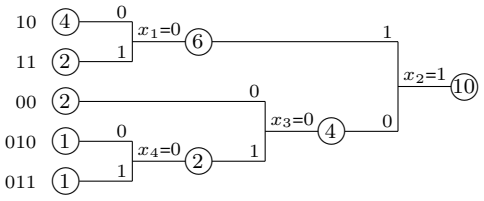

Fig. 5. The Huffman tree shown in Fig. 2 with the corresponding assignation of variables

Clearly, the assignation $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$ is better than $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$, since it leads to an H_1 with an equal number of 0's and 1's.

We propose the quadratic programming formulation shown in Fig. 3 for finding the best assignation of variables. Besides the binary variables x_1, x_2, x_3, x_4, we have two more variables: the number of zeros N_{zeros} and the number of ones N_{ones}. Clearly, the smaller the difference between N_{zeros} and N_{ones}, the smaller the quantity $N_{zeros}^2 + N_{ones}^2$. The constraints consist of two equations and some conditions regarding the values that can be assigned to the variables. The two equations correspond to the Huffman tree structure that is present in both Fig. 1 and Fig. 2. (Each of these two equations is obtained by following all the paths $T \leftrightarrow R$, where T is a terminal node, and R is the root; for example, in the first equation shown in Fig. 3, the first two terms correspond to the path between the terminal node 4 and the root 10.) Extending the formulation shown in Fig. 3 to any Huffman tree structure is straightforward. Solving the formulation shown in Fig. 3 using special programs like CPLEX [8], we would obtain the assignation $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$, that is, we would obtain the Huffman tree shown in Fig. 1, which is optimized. (We have $N_{zeros}^2 + N_{ones}^2 = 121 + 121 = 242$ in this case.)

Based on this quadratic programming formulation, we propose the following algorithm, denoted by ALG2.

Step 1.Apply the Burrows-Wheeler transform.

Step 2.Apply the Move-to-Front (MTF) transform.

Step 3.Apply Huffman coding. Once the Huffman tree structure is already built, solve the corresponding quadratic programming formulation using CPLEX [8], in order to optimize the H_1 part.

Step 4.The corresponding output is further processed by appending a number of 0's (or a number of 1's) in such a way that, in the resulting final output, the number of 0's equals the number of 1's. (Definitely, we also need to specify in the final output how many 0's or 1's have been appended in this step.)

So, ALG2 is actually ALG1, with the difference that the proposed quadratic programming formulation is incorporated into the Huffman coding step (the third step). The output of the third step in ALG2 is $H = H_1 H_2$, as in ALG1. But, since the H_1 part is optimized in ALG2 once the third step is finished, we expect that significantly less bits would be appended in the fourth step of ALG2 than in the fourth step of ALG1.

B. Experimental Results

We have implemented both ALG1 and ALG2 in order to observe the difference between the utility factors. Standard corpora used to compare data compression algorithms usually consist of files generated from an alphabet with not too many symbols. For such files, the bzip2 compressor (or equivalently, the first three steps in ALG1) is good enough that ALG1 will almost always produce outputs with an utility factor of more than 100%. The real difference between ALG1 and ALG2 can be seen in the case of files generated from an alphabet with many symbols, say 256 symbols. We have randomly generated three files, each of length $1,000,000$ bytes, as shown in Table I. The first file, file1, has been randomly generated using an alphabet with 236 symbols; file2 has been randomly generated using an alphabet with 246 symbols; and file3 has been randomly generated using an alphabet with 256 symbols.

TABLE I
FILES USED FOR COMPARING ALG1 AND ALG2

File	Size (in bytes)
file1	1,000,000
file2	1,000,000
file3	1,000,000

The results are reported in Table II. (The results given under the 'bzip2' column are actually the results obtained by applying the first three steps of ALG1; the actual steps used in the bzip2 utility are a bit more involved.) As one can see, the more symbols the source alphabet has, the smaller the utility factor. For file1 and file2, ALG2 provides utility factors of more than 100%, while ALG1 fails to do so. Even if there is not a big difference between the utility factors provided by ALG1 and the utility factors provided by ALG2 (at least for these three files), reaching an utility factor of 100% or even more seems to be of utmost importance in practice.

TABLE II
EXPERIMENTAL RESULTS

File	bzip2	ALG1	ALG2
file1	991,783	1,000,112 (util. factor: 99.90%)	991,892 (util. factor: 100.81%)
file2	998,903	1,002,676 (util. factor: 99.70%)	999,113 (util. factor: 100.08%)
file3	1,004,625	1,010,762 (util. factor: 98.90%)	1,004,838 (util. factor: 99.50%)

REFERENCES

[1] R.R. Birge, Photophysics and molecular electronic applications of the rhodopsins, *Annu. Rev. Phys. Chem.*, vol. 41, 1990, pp. 683–733.

[2] R.R. Birge, N.B. Gillespie, E.W. Izaguirre, A. Kusnetzow, A.F. Lawrence, D. Singh, Q.W. Song, E. Schmidt, J.A. Stuart, S. Seetharaman, and K.J. Wise, Biomolecular Electronics: Protein-Based Associative Processors and Volumetric Memories, *J. Phys. Chem. B*, vol. 103, 1999, pp. 10746–10766.

[3] M. Burrows and D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, 1994, Digital Equipment Corporation.

[4] bzip2: http://www.bzip.org.

[5] N.A. Hampp, Bacteriorhodopsin: Mutating a biomaterial into an optoelectronic material, *Applied Microbiology and Biotechnology*, vol. 53, 2000, pp. 633–639.

[6] J.R. Hillebrecht, J.F. Koscielecki, K.J. Wise, D.L. Marcy, W. Tetley, R. Rangarajan, J. Sullivan, M. Brideau, M.P. Krebs, J.A. Stuart, and R.R. Birge, Optimization of Protein-Based Volumetric Optical Memories and Associative Processors by Using Directed Evolution, *NanoBiotechnology*, vol. 1, 2005, pp. 141–151.

[7] D.A. Huffman, A Method for the Construction of Minimum-Redundancy Codes, in *Proceedings of the I.R.E.*, 1952, pp. 1098–1102.

[8] ILOG CPLEX: http://www.ilog.com/products/cplex.

[9] V. Kundeti, S. Rajasekaran, and R. Birge, Generalized Algorithms for Generating Balanced Modulation Codes in Protein-based Volumetric Memories, in *Proceedings of the IEEE Congress on Evolutionary Computation*, Trondheim, Norway, 2009, pp. 1585–1590.

[10] S. Rajasekaran, V. Kumar, S. Sahni, and R. Birge, Efficient Algorithms for Protein-Based Associative Processors and Volumetric Memories, in *Proceedings of the 8th IEEE Conference on Nanotechnology*, Arlington, TX, 2008, pp. 397–400.