ELSEVIER

Signal Processing: *Image Communication* 17 (2002) 277–292

SIGNAL PROCESSING: IMAGE COMMUNICATION

www.elsevier.com/locate/image

Design of an improved lossless halftone image compression codec

Koen Denecker*, Dimitri Van De Ville[1], Frederik Habils, Wim Meeus, Marnik Brunfaut, Ignace Lemahieu

Department of Electronics and Information Systems, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Received 6 June 2000; received in revised form 12 February 2001; accepted 15 June 2001

Abstract

The popularity of high-resolution digital printers and the growing computational requirements of new applications such as printing-on-demand and personalized printing have increased the need for fast and efficient lossless halftone image compression. In a previous paper, we have shown that the compression performance can be improved significantly by adapting the context template to the halftone parameters. Unfortunately, this variability in data dependency makes the modeling stage more complex and slows down the overall compression scheme. In this paper, we describe the design of an improved block-based software and hardware implementation. The software implementation uses complementary line-shifting to by-pass the adaptivity of the template. The hardware implementation is based on the automated construction of a microcoded program from a given template. Experimental results show that our improved implementation achieves approximately the same processing speed as when the fixed context template is applied. The proposed implementation is also of importance for the emerging JBIG2 standard which uses up to four adaptive template pixels. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Halftone image compression; Bilevel image compression; JBIG; JBIG2; Compression hardware

1. Introduction

Halftoning is the last stage in the prepress workflow before a document can be printed [16]. It transforms grayscale or color images into bilevel images which can be reproduced on a printing device. This is necessary because a printer is a binary medium: either there is ink or there is no ink. As a result, large bilevel images arise which put heavy constraints on the storage and transmission media.

New applications in the digitized prepress industry like telepublishing and printing-on-demand show a need for intermediate storage and fast transmission of the halftones. Therefore, halftone image compression will easily improve the flexibility and total cost of the workflow. Tasks that will benefit from compression are: temporary

*Corresponding author.

E-mail addresses: denecker@elis.rug.ac.be, kdenecke@cisco.com (K. Denecker), dvdevill@elis.rug.ac.be (D. Van De Ville), fhabils@elis.rug.ac.be (F. Habils), wmeeus@elis.rug.ac.be (W. Meeus), brunfaut@elis.rug.ac.be (M. Brunfaut), il@elis.rug.ac.be (I. Lemahieu).

[1] Research Assistant of the Fund for Scientific Research, Flanders, Belgium.

0923-5965/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.

PII: S0923-5965(01)00030-3

storage for buffering purposes, decreased transmission times for network-based printing in the case where the digital halftoning happens at a different place than the printing, and decreased computational time for the halftoning procedure in those cases where specific elements need to be halftoned multiple times.

On the other hand, compressing and decompressing the halftone are two new tasks that enter the process and they need to be done multiple times, preferably at very high speeds. As a result, there is a need for fast software and hardware implementations of very powerful halftone image compression algorithms.

The current bilevel image compression standard is JBIG (Joint Bilevel Image Experts Group) [5,12]. The compression technique is based on the combination of a context modeling block and an arithmetic coder. The context template comprises nine pixels at fixed locations and one at a variable place. The best results on halftones are achieved in sequential mode (i.e. when no multi-resolution decomposition is applied) and when the adaptive template pixel is chosen at the screening period in the case of classical halftones.

However, in the case of halftone images, much better compression can be achieved if a larger template is chosen and if, for each given halftone, an optimized context template is constructed [8]. An efficient way to approximate such an optimized template for classical halftones is by taking the best correlated pixels [3].

As a result, compression performance will improve but processing speed will generally decrease due to the increased complexity. Increasing the size of the template leads to an approximately linear increase of the context building time and an exponential increase of the modeling memory. Use of an adaptive template gives rise to highly unpredictable data dependencies. Since the context template is not known at compile-time and no assumptions about the template can be made, optimization is a non-trivial task. In this paper, we propose an efficient implementation both in software and in hardware of the context modeling stage with a free template.

The international standardization organization (ISO) has recently drafted JBIG2, the newest bilevel image compression standard [6]. Amongst other functionalities, it uses a method for lossless halftone image compression similar to JBIG1 [4]. However, the new standard will allow the context template to consist of twelve fixed template pixels and four adaptive template pixels [4]. These numbers are derived from the observation that using a standard implementation and compared to the fixed JBIG1-template, the algorithm is already four times slower if four adaptive template pixels are allowed [8]. This leads us to the conclusion that a completely free template did not make it into the standard for performance reasons.

The development of a context modeling implementation that allows a completely free template is not only beneficial for lossless bilevel image compression. The freedom one has to choose the template pixels creates a universal compression scheme which can be used successfully for all kinds of binary data. With only a few minor modifications and by carefully selecting the template, an efficient grayscale and color image compression scheme can be constructed [2]. Unfortunately, due to the bit-oriented processing of byte-oriented data, this approach is destined to be slower than algorithms processing the images pixel by pixel.

The previous arguments show that a fast implementation of a free template context modeling algorithm is useful. Because the template pixels are not known at the time of software compilation or hardware design, the context model is very general and flexible. This complicates the optimization of the implementation. Section 2 describes the compression algorithm and its building blocks in detail. Section 3 presents an optimized software implementation for the context modeling and the autocorrelation block. Section 4 describes the hardware design of the context modeling block and the interaction between the other building blocks such as the QM-coder and the memory blocks. Results from both the software and the hardware optimization on a realistic halftone image are given for both the software and the hardware implementation. Finally, the paper ends with a conclusion in Section 5.

2. Compression algorithm

The fast implementations presented in this paper do not cover the complete compression algorithm but only the context modeling as well as an optimized template generation algorithm. The halftoning, which happens prior to the compression and decompression, is described first.

2.1. Halftoning

The purpose of the halftoning algorithm is to render the illusion of contone (continuous-tone) pictures on a medium that is capable of producing only binary pixel elements: either the pixel is covered with ink or not [16]. This operation is a computationally intensive application [7].

The halftoning techniques for printing applications can be divided into two classes. In the case of "classical halftoning", the contone image is replaced by dots of variable size located on a fixed rotated rectangular grid (AM or amplitude modulation). Each dot is built using tiny laser spots. The resolution of the dots is typically about half the resolution of the contone image and is called the screen ruling (typically about 150 lpi or lines per inch). The resolution of the spots is much higher and is called the screening resolution (typically about 1270 spi or spots per inch). This type of halftoning is by far the most popular one in the conventional printing industry. On the other hand, in the case of "stochastic halftoning", the contone image is replaced by equally sized dots at varying places (FM or frequency modulation). The dots appear to be distributed in a stochastic or random way. As an illustration of the halftoning process, a detail of a classical and a stochastic halftone image is represented in Fig. 1.

2.2. Compression algorithm overview

Lossless image compression using context modeling processes the image, pixel by pixel, in rasterscan sequence. For each pixel x, the pixels already encoded are mapped onto a context c (or context class) by a "context mapping function" [13]. Only a finite number of different contexts are allowed. All pixels mapped to the same context are regarded as samples of the same stationary and ergodic source. To guarantee lossless reconstruction, the context mapping function must be deterministic and may use only pixels which have already been processed. Once the context c is determined, the conditional probability $p(x|c)$ is estimated based on all previously observed pixels given that particular context. The estimated probability $\hat{p}(x|c)$ together with the observed pixel value x is fed into an entropy coder which

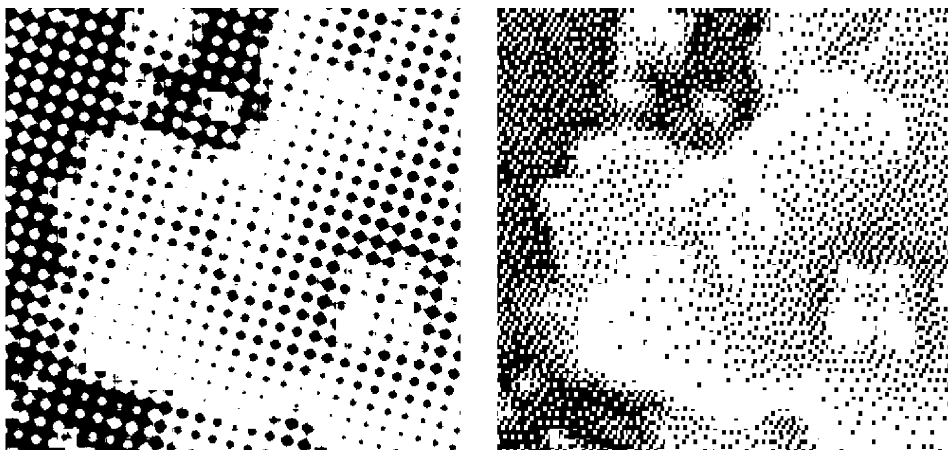


Fig. 1. Magnification of a detail of the "Musicians" image; (left) classical screening at 2540 spi and 75° and (right) stochastic screening at 2540 spi.

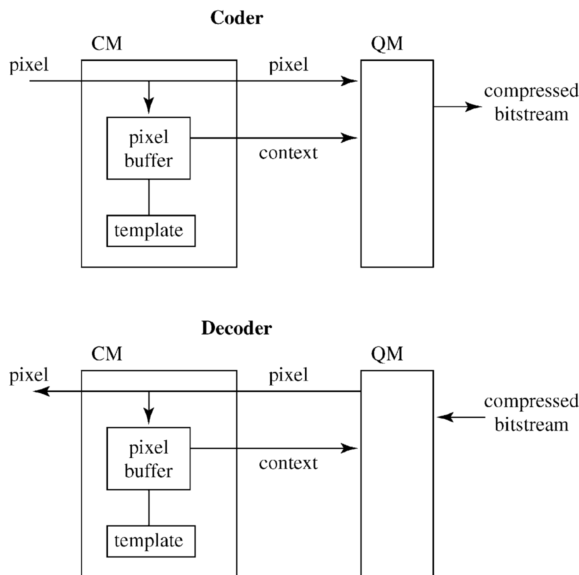


Fig. 2. The overall compression algorithm consists of a context modeling block (CM) and a QM-coder. The coder (top) and the decoder (bottom) perform approximately the same operations but their data paths differ.

produces the encoded bitstream. A detailed description about context modeling can be found in [1,14].

In the case of bilevel images, the QM-coder is often used as an entropy coder. However, it does more than that since it also fulfills the function of both probability estimator and entropy coder [5]. It implements the estimator

$$\hat{p}(x|c) = \frac{n(x|c) + \delta}{n(0|c) + n(1|c) + \delta},$$

where $n(x|c)$ indicates the number of observations x given the context c and $\delta = 0.45$. The QM-coder also has provisions to take care of slow variations in the observed probability estimates.

A general scheme of a bilevel image codec using the QM-coder is represented in Fig. 2. The decoder mimics the behavior of the coder, however it has a significantly different data path.

The context model is completely described by the context mapping function. In general terms, this function maps the already processed part of the image, the past, onto a finite number c called the "context". The choice of the context mapping function is very general and determines both the complexity and the efficiency of the model. The most popular technique that uses context modeling is template coding. This technique concatenates the values of all pixels belonging to a prespecified set of relative positions to form the context. This set of relative positions is called the template and the size q of the template is called the order of the context model. The template can be fixed or free: a fixed template uses the same set of relative pixel positions for every image, whereas a free template tunes the set for every image. If only a subset of the template is free, as in JBIG2, the template is called partially free.

Template coding with a unique and constant order is a good tradeoff between compression efficiency and model complexity. Better compression results can be achieved when templates of two different sizes are combined [10]. The context model only uses the large template if the corresponding small template has occurred frequently enough.

More complex models make use of a context tree, which can be regarded as a context model with templates of variable size [3,8,13]. Firstly, for each pixel, the two-dimensional image data from the past is mapped onto a one-dimensional sequence. Then, a context tree of all possible contexts along this sequence is constructed, where the depth of tree corresponds with the delay in the past. Finally, an appropriate context length is chosen based on a certain criterion. This approach is much slower, but is still feasible without using greedy algorithms.

In context trees, the relative position of a template pixel is the same for every node of the tree at the same depth. This is no longer in the case of free tree coding, a generalization of the tree-based context models where the location of the template pixel is dependent on the branch of the tree [8]. However, since a more general class of models is used, more degrees of freedom must be tuned for every image and this makes the parameter estimation very tedious. Often this parameter optimization takes too much time to be practical.

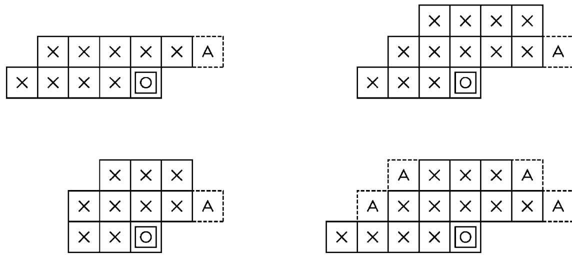


Fig. 3. Four templates can be used in the lowest-resolution layer of JBIG2. The two templates on the left were already part of JBIG1. The current pixel is marked "O", the fixed template pixels are marked "X" and the adaptive template pixels are marked "A".

2.3. Template coding

The context template defines the position of the context pixels relative to the pixel currently being encoded. The context is obtained by concatenating the pixel values according to the template. A fixed template is not dependent on the image. A free or semi-adaptive template can be different for every image but does not alter during the encoding of the image. A partially free template (or a partially fixed template) combines fixed and free template pixels. An adaptive template can change during the image encoding. For lossless compression, a free or adaptive template must be communicated to the decoder.

The sequential JBIG2 templates are presented in Fig. 3 [5,6]. All four are partially free. The adaptive pixel templates are introduced to improve compression performance on halftones. Note that although JBIG1 and JBIG2 were designed with multi-resolution decomposition, the best compression results are obtained if only one resolution layer is used.

2.4. Autocorrelation-based template

The use of adaptive template pixels significantly improves compression performance in the case of halftones, but at the same time it introduces additional degrees of freedom which need to be determined prior to compression. Normally, this parameter optimization only needs to be done at the encoding side. For classical halftones, a good approximation of the optimal free template can be obtained by taking the best correlated pixels [3]. Unfortunately, this approach does not produce good results for stochastic halftones. For both types of halftones, this autocorrelation-based template can be improved by performing a greedy template search [3,8]. The greedy search in [3] is performed on a representative part of the image. It is many orders of magnitude slower than the approach in [8], but it has less concession with respect to search optimization. A full optimal template search is not feasible on realistic images and context sizes. As an illustration, free templates for a representative classical and stochastic halftone are shown in Fig. 4. These templates clearly show that in the case of halftones, an optimized free template has only few pixels in common with the fixed template pixels from the JBIG proposed templates.

Table 1 shows compression ratios obtained on a representative classical and stochastic halftone image. In this paper, compression ratio is defined as the ratio of uncompressed file size to compressed file size. The "JBIG1*, 9F + 1A" results are obtained using the JBIGKIT v1.2 implementation with minimal requirements.[2] This means that the position of the adaptive template pixel is limited to the current line and the maximal offset is limited to 8. All other results are obtained using our implementation, and decompression and verification were performed afterwards to verify the implementation. The compression results marked "b" were obtained using a slight modification of the greedy search algorithm: since the greedy search was performed on a 2048×2048 part of the image, an optimum in compression ratio was achieved at the order of 15 and 14 for images "ro_10k" and "ro_20k", respectively. However, for the entire image, experiments have shown that the optimal order is higher. Unfortunately, performing the greedy search on the entire image is not an option due to time constraints. Therefore, the greedy templates of size 15 and 14, respectively, are enlarged to size 16 by adding extra

[2] ftp://ftp.informatik.uni-erlangen.de/pub/doc/ISO/JBIG/jbigkit-1.2.tar.gz

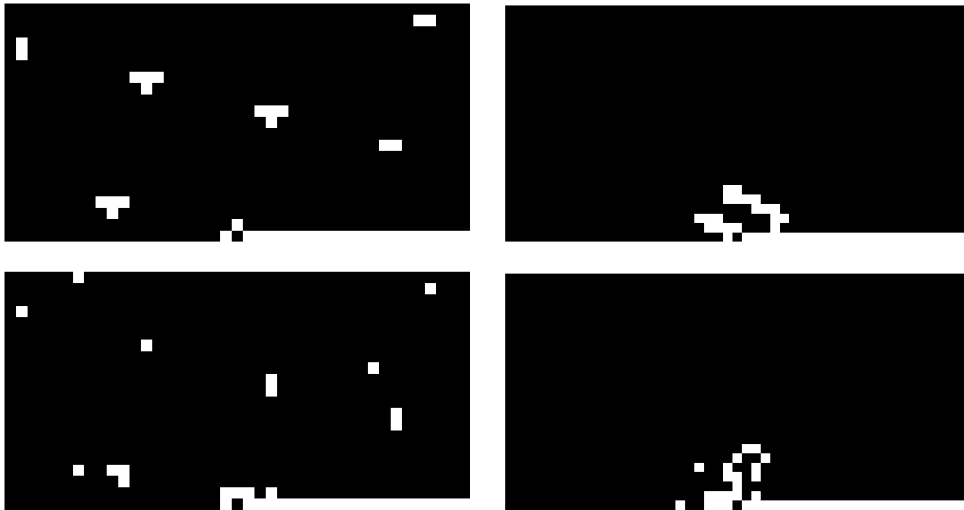


Fig. 4. Free templates obtained using auto-correlation (top) or using a greedy search (bottom) for a classical halftone (left) and a stochastic halftone (right). The last black pixel on the bottom line is the current pixel.

template pixels which are closest with respect to L_1-distance. The L_n-distance between two positions (x_1, y_1) and (x_2, y_2) is defined as $(|x_2 - x_1|^n + |y_2 - y_1|^n)^{1/n}$. The table clearly shows that much higher compression ratios can be achieved if multiple adaptive template pixels are being used.

3. Software implementation

The major building blocks of the encoder and the decoder are similar, but they show an important difference in the data dependencies. At the encoding side, it is possible to construct all contexts prior to transmitting even the very first pixel and context to the QM-coder. But at the decoding side, the value of each context may depend on the previously reconstructed pixel, and this creates a very short critical path.

The implementation presented in this section is optimized for a free or semi-adaptive template. By this, we mean that the template is fixed for the duration of the encoding of the entire image, but it varies from image to image. Hence, the template is not known at compile-time. If an elementary implementation is used, the overall compression speed is an order of magnitude lower compared to one of the closest-distance fixed templates.

The hardware-optimized Q-coder [9,11] served as a starting point when the JPEG/JBIG committee developed the QM-coder. The QM-coder serves as an adaptive binary arithmetic coder optimized for software. Later, the Q-coder and software-optimized QM-coder are unified into the Qx-coder [15]. The QM-coder was patented by IBM, but good implementations of the QM-coder are available from multiple sources.[3] It is designed to be multiplication-free and it uses only one byte of memory per possible context. If template coding is used and q is the size of the template, then 2^q bytes are needed for the context model.

For the above reasons, the QM-coder is not described in greater detail. The design considerations to improve the context modeling block are presented first, both for encoding and decoding. Secondly, an efficient implementation of the construction of a template based on autocorrelation is described.

[3] from ftp://nic.funet.fi/pub/graphics/misc/test-images/jbig.tar.gz or ftp://ftp.informatik.uni-erlangen.de/pub/doc/ISO/JBIG/jbigkit-1.2.tar.gz

Table 1
Image characteristics and compression ratios using different template types[a]

	ro_10k	ro_20k	mo_10k	mo_20k
Width	9525	19050	9525	19050
Height	10795	21590	10795	21590
Halftone	class.	class.	stoch.	stoch.
Angle	75°	75°	—	—
spi	1270	2540	1270	2540
L1, 10F	4.10	6.50	4.19	13.42
L2, 10F	4.32	7.35	4.18	12.96
JBIG1*, 9F + 1A	4.30	7.31	4.16	12.91
JBIG1, 9F + 1A (2L)	5.47	9.15	4.34	16.75
JBIG1, 9F + 1A (3L)	5.59	9.34	4.40	15.15
JBIG2, 12F + 1A	5.96	10.22	5.04	16.05
JBIG2, 12F + 4A	7.21	12.11	5.59	18.80
autocorr, 6A	5.10	9.93	2.45	11.49
autocorr, 10A	5.43	10.71	3.31	12.43
autocorr, 13A	5.54	11.35	3.63	12.95
autocorr, 16A	6.71	12.31	4.26	12.19
autocorr, 20A	6.68	12.56	4.88	13.10
greedy, 6A	6.34	10.66	3.44	16.33
greedy, 10A	7.08	11.85	4.30	19.97
greedy, 13A	7.40	12.58	5.42	22.89
greedy, 16A	7.55[b]	13.11[b]	5.87	24.37

[a] Key: "L1" and "L2" stand for L_1- and L_2-distance limited templates, "xF" and "yA" stand for x fixed and y adaptive template pixels, respectively. The JBIG1*-entry stands for the JBIG1 minimal requirements and makes use of the JBIGKIT implementation.

[b] The slight modification in the greedy search algorithm for order 16 is described in the text.

3.1. Context modeling for encoding

In software, we pose no restrictions on the size of the template or the position of the template pixels. The implementation we propose aims at minimizing memory requirements and maximizing the throughput. Also, once the template is determined, the image data should be accessed only once and buffered when necessary. This allows the encoding block to be pipelined on a higher level.

Let m and n be the width and the height, respectively, of the image being coded. Furthermore, let $x(i,j)$ be the pixel value at position (i,j) and let (δ_i^t, δ_j^t) be the relative position of template pixel t. By convention, j denotes the line number starting from the top, and i denotes the horizontal pixel position starting from the left, so $0 \leqslant i < m$ and $0 \leqslant j < n$. Since only pixels from the past are to be used, either $\delta_j^t < 0$ or both $\delta_j^t = 0$ and $\delta_i^t < 0$ must hold. The template T is defined by $\{(\delta_i^t, \delta_j^t) \mid 1 \leqslant t \leqslant q\}$. Finally, the context of pixel $x(i,j)$ is defined as

$$c(i,j) = x(i+\delta_i^1, j+\delta_j^1) \oplus x(i+\delta_i^2, j+\delta_j^2) \oplus \cdots \oplus x(i+\delta_i^q, j+\delta_j^q),$$

where $\oplus$ indicates the concatenation of bits. Furthermore, define δ_i^- and δ_i^+ as the minimum and maximum, respectively, of all δ_i^t, hence $\delta_i^- = \min\{\delta_i^t \mid 1 \leqslant t \leqslant q\}$ and $\delta_i^+ = \max\{\delta_i^t \mid 1 \leqslant t \leqslant q\}$. Equivalently, define δ_j^- as the minimum of all δ_j^t, so $\delta_j^- = \min\{\delta_j^t \mid 1 \leqslant t \leqslant q\}$. The proposed software implementation puts no restrictions on these values.

To minimize the memory requirements, two line buffers are implemented. These line buffers are carefully set up before the first pixel of a new line is encoded and they do not change during the encoding of an entire line.

The first buffer L is a buffer that holds the $-\delta_j^-$ lines above the current line j. The goal of this buffer is to guarantee that each line is read only once and buffered as long as necessary.

The second buffer S aims at speeding up the context determination by carefully shifting the appropriate previous lines according to the context template. This "shift buffer" does not access the image data directly, but gets them from the line buffer L or from the current line. Given the template $T = \{(\delta_i^t, \delta_j^t) \mid 1 \leqslant t \leqslant q\}$, the shift buffer S is organized as follows. It contains exactly q lines S_t, where $1 \leqslant t \leqslant q$. If the line with number j is to be encoded, then S_t contains the line numbered $j + \delta_j^t$, shifted to the left over a distance δ_i^t. If δ_i^t is negative, the shifting is to the right. This means that all previous lines containing context pixels are taken and that they are shifted in such a way that the context distance is compensated. The context of the pixel at position i of the current line is determined by the pixels $S_t(i)$, where $1 \leqslant t \leqslant q$. The shift buffer can be constructed completely, even if context pixels from the current line (i.e. for which $\delta_j^t = 0$) are used. Note that if multiple template

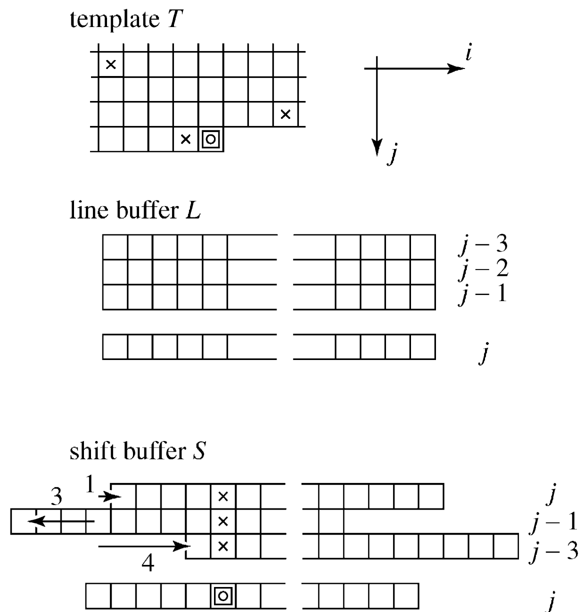


Fig. 5. Given the template T, two buffers are maintained for encoding. The line buffer L contains the previous δ_j^- lines. The shift buffer S contains the q lines given by $j + \delta_j^t$, each shifted to the left over a distance δ_i^t. As a consequence, the contexts show up as a vertical line in S.

pixels are on a same line, then that line will occur multiple times in the shift buffer S. These buffers remain unchanged during the encoding of an entire line.

As an example, take the template defined as $T = \{(-4,-3), (3,-1), (-1,0)\}$. Then $q = 3$, $\delta_i^- = -4$, $\delta_i^+ = 3$ and $\delta_j^- = -3$. The line buffer L contains the $-\delta_j^- = 3$ previous lines numbered $j-1$, $j-2$ and $j-3$. The shift buffer S contains $q = 3$ lines: line $j-3$ shifted to the right over a distance of 4 pixels, line $j-1$ shifted to the left over a distance of 3 pixels and line j shifted to the right over a distance of 1 pixel. This example is illustrated in Fig. 5.

To make optimal use of the 32-bit architecture of most processors, pixel values are represented using 1 bit per pixel and are grouped to form blocks of $B = 32$ bits. The shifting operation of an entire line is performed on a block-by-block basis. A single block is shifted by splitting it into a part that remains within the block boundaries and a part that crosses the block boundaries. Like this, the amount of time used to set up the shift buffer is negligible compared to the rest of the encoding process.

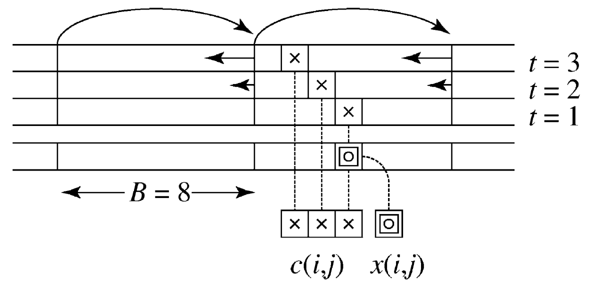


Fig. 6. Each block in S_t is cyclically shifted to the left over a distance $t-1$. For clarity, the block size is $B = 8$ in this example, but on normal processors, $B = 32$. The full arrows illustrate the cyclic shifts. The dashed lines illustrate the context bits dropping vertically to generate the context.

Finally, to determine the B contexts corresponding with the B pixels in a block, B vertical columns must be read from the shift buffer S. This operation is similar to the transposition of a matrix of $q \times B$ bits. Due to the integer-oriented nature of the instructions of a processor, this operation is not easy to optimize.

Experiments showed that this can be done faster by additionally performing a cyclic block-wise shift in the shift register S: each block of B pixels in S_t is cyclically shifted to the left over a distance of $t-1$, with $1 \leqslant t \leqslant q$. As a result of this additional shift, the context bits can drop down to construct the context. This is illustrated in Fig. 6. The "drop-down" operation of an individual bit is performed by an AND-operation with a bit mask containing only one '1'-bit and an OR-operation with the intermediate context result. The net improvement in speed induced by the additional block-wise shift operation can be explained by the replacement of B per-bit shift operations by one per-block shift operation.

3.2. Context modeling for decoding

Due to the fundamentally different data path between encoding and decoding, the context modeling approach used for encoding cannot be used without modifications. The only situation

when the previously described approach can be used is when the template contains no template pixels from the current line, i.e. $\delta_j^t < 0$ for all $1 \leqslant t$. In this section, we describe the necessary modifications to the encoding implementation. Since the line buffer L contains no information of the current line, it is the same as for the encoding.

Depending on the length of the data path, three types of template pixels can be discerned. Firstly, template pixels on previous lines, i.e. $\delta_j^t < 0$, are known for the whole line prior to the decoding of the first pixel of the current line. Secondly, there are the template pixels on the current line but at larger distance than the block-size B, i.e. $\delta_j^t = 0$ and $\delta_i^t \leqslant -B$. These pixels are not known for the entire line, but only for the next block. Thirdly, there are the template pixels on the current line and closer than the block-size, i.e. $\delta_j^t = 0$ and $\delta_i^t > -B$. Their values are responsible for the most time-critical data path in the decoding process. Based on these considerations, the template is divided into three subtemplates T_1, T_2 and T_3, respectively, each containing the template pixels of one of these types. If $B = 32$, then T_2 will usually contain no pixels, however it is necessary if one aims at a general implementation without linked lists. These subtemplates are depicted in Fig. 7.

The shift buffer S is maintained differently to accommodate the new subtemplates. Define q_k as the size of template T_k. The shift buffer now contains only $q_1 + q_2$ lines instead of q lines. As in the encoding case, the shifted lines corresponding to T_1 can be fully determined prior to the decoding of the first pixel of the current line. Moreover, these lines remain unchanged during the decoding of the current line. The shifted lines corresponding to T_2 are not known beforehand, but are filled in block by block as they are generated during the decoding process. The shifted lines corresponding to T_3 are omitted from the shift buffer S since the critical data path can be as short as one bit, meaning that part of the shift buffer would need extremely frequent updating.

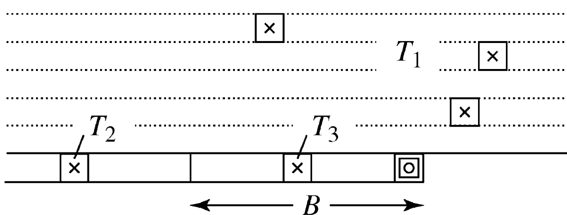


Fig. 7. Dependent on the data path, the template T is divided into three subtemplates T_1, T_2 and T_3.

The determination of the contexts remains the same for the template pixels belonging to T_1 and T_2. For the template pixels belonging to T_3, a special register of B bits called the "bypass block" is used. It contains the B most recently decoded pixels and the context pixels corresponding with T_3 are determined from this block. Each newly reconstructed pixel is immediately shifted into the bypass block. After each block, the B pixels from the bypass block are copied into the line currently reconstructed as well as into the part of the shift buffer corresponding with T_2.

In the previous example, $T_1 = \{(-4,-3), (3,-1)\}$, $T_3 = \{(-1,0)\}$ and T_2 is empty. The shift register contains two shifted lines and the bypass block is empty at the beginning of the line. A context is constructed in two phases. In the first phase, the same technique as during the encoding is used to determine the first two context bits. In the second phase, the most recent pixel from the bypass block is used to determine the third context bit. The reconstructed pixel is shifted into the bypass block immediately. Every B pixels, the bypass block is copied into the line currently reconstructed.

3.3. Autocorrelation

The template construction procedure for classical halftones presented in [3] constructs the binary autocorrelation function. For the whole image, this function is defined as

$$A(\delta_i, \delta_j) = \left(\sum_S (1 - x(i,j) \wedge x(i+\delta_i, j+\delta_j)) \right) \Bigg/ \left(\sum_S 1 \right),$$

where $a \wedge b$ is defined as 1 if $a \neq b$ and 0 otherwise, and where S is defined as that region of the image such that both $x(i,j)$ and $x(i+\delta_i, j+\delta_j)$ are well

defined. To increase speed, this function is only evaluated on a representative part of the image, S typically is many times smaller than the image. Note that $A(\delta_i, \delta_j)$ must be evaluated for many values of (δ_i, δ_j).

In the case of classical halftoning, a part of an image can be considered representative if the average gray level is between 25% and 75%, if the gray level is never $<5\%$ or $>95\%$ and if its size covers at least about 100 halftone dots. This definition is very "ad hoc", but has proven to be successful over a quite large set of halftoned photographic images, under the assumption that the original grayscale image is close to stationary.

The autocorrelation function can be implemented in multiple ways, but if it has to be determined on a large (δ_i, δ_j) range, it involves a lot of bit comparisons. For example, if the image part is m' pixels wide and n' pixels high and if furthermore $\delta_i^- \leqslant \delta_i \leqslant \delta_i^+$ and $\delta_j^- \leqslant \delta_j \leqslant 0$, then about $m'n'(\delta_i^+ - \delta_i^- + 1)(1 - \delta_j^-)$ bit comparisons need to be performed. For example, typical values could be $m' = n' = 1024$ and $-\delta_i^- = \delta_i^+ = -\delta_j^- = 32$, yielding about 2×10^9 bit comparisons.

The components each implementation must have is four nested loops (for i, j, δ_i and δ_j), a pixel comparison function and a summation function. Empirically, the following combination was found to be optimal: pixels are represented using 1 bit, the outer loop covers j, the middle loops cover δ_j and δ_i, and the inner loop covers i in blocks of size $B = 32$. As in the software encoding, a shift buffer is used which, for a given value of j, holds all rows $j + \delta_j$ shifted over a distance δ_i, for every possible value of δ_j and δ_i. The pixel comparison function is performed by a lookup table $X(d)$, which contains the number of 1-bits for every 16 bit word d. A 16 bit word corresponds to half a block. The optimized implementation looks as follows:

```
for every line j do:
   set up S(di,dj) for all (di,dj)
   for every dj do:
      for every di do:
         for every word i' do:
            a = word i' in line j
            b = word i' in line S(di,dj)
            add X(a∧b) to autocorr'(di,dj).
```

During our tests, we noticed that the optimal addressing of the lookup table was dependent on the platform and the size of the memory caches. However, differences between platforms are small.

3.4. Experimental results

The proposed software implementations are developed on IBM AIX using the native C-compiler and on Linux using the Pentium-optimized pgcc-compiler. The compiler optimization was set at its highest level. The AIX hardware platform is a PowerPC 604e running at 166 MHz with 64 Kbyte L1 cache and 512 Kbyte L2 cache, whereas the Linux hardware platforms comprise an Intel Pentium II Xeon 450 MHz with 512 Kbyte L2 cache and an Intel Pentium III 700 MHz. All results tend to scale with processor speed, except for the autocorrelation for which the optimal size of the lookup table is dependent on the size of the cache. All speeds reported hereafter are obtained on the Linux Pentium III 700 MHz platform. The other platforms were investigated to detect processor-specific or cache-dependent optimizations.

3.4.1. Autocorrelation

The autocorrelation function is only implemented in software. As expected, the processing speed scales linearly with each of the ranges of i, j, δ_i and δ_j. Evaluation of the autocorrelation function on a typical 1024×1024 part of the image for relative distances $-31 \leqslant \delta_i^t \leqslant 31$ and $-31 \leqslant \delta_j^t \leqslant 0$ requires about 2×10^9 bit comparisons. This takes about 1.95 s, so that about 10^9 bit comparisons can be processed per second on the Pentium III 700 MHz platform. The Pentium II Xeon 450 MHz takes about 2.25 s, and this increase in processing speed per clock cycle is due to the larger and faster L2 cache. A straightforward implementation using 1 byte per pixel is about 60–100 times slower, dependent on the platform. The fast implementation requires enough memory to buffer approximately $(1 - \delta_j^-)(\delta_i^+ - \delta_i^- + 1)$ pixel lines and autocorrelation counts.

Table 2
Encoding and decoding speeds in Mpix/s for typical images and using various templates and orders[a]

	ro_10k	ro_20k	mo_10k	mo_20k
Coding				
JBIG1*, 9F+1A	11.84	12.37	11.85	39.11
JBIG1, 9F+1A (3L)	13.61	14.42	13.31	14.98
JBIG2, 12F+1A	12.64	13.31	12.35	13.62
JBIG2, 12F+4A	11.70	12.12	11.52	12.62
autocorr, 6A	16.05	17.31	13.73	17.73
autocorr, 10A	13.53	14.47	12.64	14.78
autocorr, 13A	12.47	13.34	11.81	13.59
autocorr, 16A	11.57	12.19	11.00	12.37
greedy, 6A	16.51	17.43	15.06	18.11
greedy, 10A	13.92	14.56	13.36	15.10
greedy, 13A	12.84	13.41	12.48	13.89
greedy, 16A	11.43	11.64	11.52	12.69
Decoding				
JBIG1*, 9F+1A	15.04	16.60	15.00	49.04
BIGKIT JBIG1, 9F+1A (3L)	11.49	12.26	11.24	12.18
JBIG2, 12F+1A	10.10	10.63	9.97	11.44
JBIG2, 12F+4A	10.23	10.47	9.87	10.31
autocorr, 6A	15.17	14.76	11.30	15.14
autocorr, 10A	13.01	12.53	10.80	12.69
autocorr, 13A	11.84	11.75	10.28	11.67
autocorr, 16A	10.00	10.52	9.43	9.95
greedy, 6A	13.97	14.83	12.60	13.85
greedy, 10A	12.07	12.62	11.35	12.37
greedy, 13A	11.32	11.41	10.06	11.17
greedy, 16A	8.94	8.76	9.35	10.21

[a] The results marked* are obtained using the freely available JBIG-kit implementation, for which the adaptive template pixel offset is limited.

3.4.2. *Context modeling and compression*

The context modeling has been implemented both in software and in hardware. We have used the QM-coder from the JBIG-kit rather than have it reimplemented. The results obtained on the various platforms scale with the clock frequency of the respective processor. The speeds we present in this section are compression speeds rather than those of the context modeling block only. As a result, the observed coding and decoding times depend on the order of the template, the type of the template and on the predictive power of the template.

Table 2 represents the encoding and decoding speeds for the test images and some of the templates of Table 1. All results, except for the entries marked*, are obtained using the proposed software. For both encoding and decoding, the observed speeds do not differ very much as a function of the image, the order and the template. On average, the encoding speed for order 10 is 13.95 Mpix/s, which is about 15% faster than the JBIG-kit implementation if the exceptional "mo_20k" image is not taken into account. Apparently, the JBIG-kit encoding and decoding speed is much higher (up to 40 Mpix/s for encoding and 50 Mpix/s for decoding) if high compression ratios can be reached. For decoding, the average speed is 11.81 Mpix/s, which is about 25% slower than the JBIG-kit implementation. Decoding is about 10–15% slower than encoding, which is explained by the difference in data dependency. However, we find it strange that the JBIG-kit decoding is faster than the encoding. Using a profiler, we observed that about the QM-coder takes about 40% of the processing time.

The worst template with respect to data dependency for the improved software implementation we propose, consists entirely of template pixels immediately to the left of the current pixel. It is unlikely that a template like this will be used for image compression, and one could develop an exception for this particular type of template so that it can be optimized at compile-time. Our implementation tends to be generic and avoids introducing this type of exceptions. Keep in mind that the overall speed is a combination of the context modeling speed and the arithmetic coding speed. The results for the worst template and for the autocorrelation-based template for all orders q ranging from 0 to 24 for the "ro_10k" image are shown in Fig. 8. For encoding, the speed difference between typical and worst template is caused entirely by the increased load of the QM-block. Since the worst-case template is less optimal with respect to compression performance, a longer bitstream is produced. For decoding, the data dependency problem manifests itself and it outweighs the increased load of the QM-block. The figure also clearly shows how encoding and decoding speeds decrease with increasing order q. The compression ratios obtained using the worst template are not relevant.

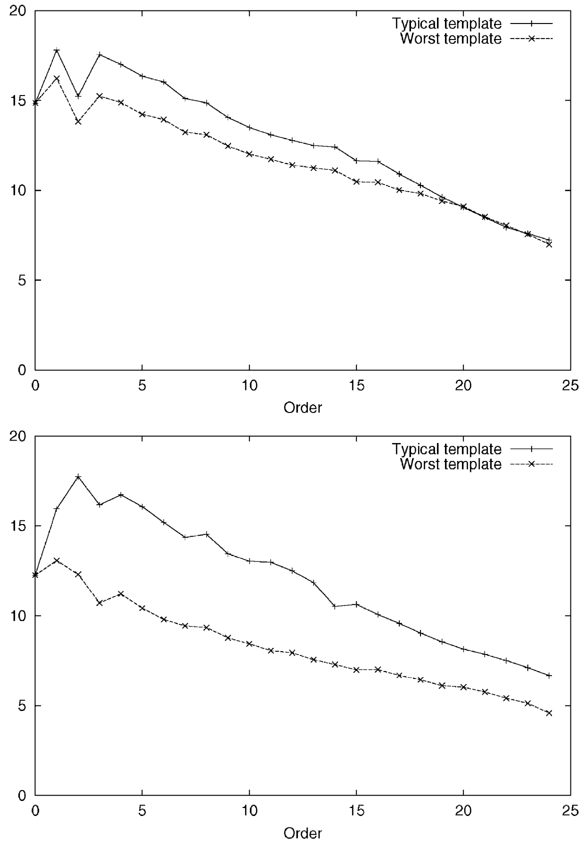


Fig. 8. Encoding (top) and decoding speeds (bottom) using worst and the autocorrelation-based template for the "ro_10k" halftone image, for the order q ranging from 0 to 24.

4. Hardware implementation

4.1. Introduction

The adaptivity of the context template poses the same heavy constraints on the design of an efficient hardware implementation as of the software implementation. The locations of the template pixels are not known at design time, so optimization is not trivial. A key difference between software and hardware implementation is that the maximal amount of memory for buffering must be determined at design time. This includes maximal order, line width and maximal offsets.

The hardware design focuses mainly on the context modeling (CM-block). The arithmetic coder and the accompanying statistical coder (the Qx-coder [15]) are already commercially available as an ASIC core. Therefore, we only implemented a behavioral VHDL-model of the QM-block. Because the amount of memory on the ASIC needs to be determined in advance, we imposed following additional constraints: (1) the maximum number of contextbits is $q = 12$; (2) the offsets (δ_i^t, δ_j^t) of a template pixel are restricted to $\delta_i^- = -31$, $\delta_i^+ = 31$, $\delta_j^- = -32$; (3) the maximal width of a line is 32 Kbit. The combination of these constraints limits the buffer to 128 Kbyte. These constraints are a global trade-off and the design process can be repeated for other combinations of parameters.

The design of the CM-block is based on a technique called software pipelining: it tries to take advantage of mixing operations needed for different context calculations. In particular, each program cycle loads the data from the next block while producing the contexts for the last line of the current block. Throughout this section we illustrate the hardware design by example for a small block size, i.e. $B = 4$. In reality, of course, the hardware design is carried out for a much larger blocksize in accordance to the constraints set above: the maximal offsets of (δ_i^t, δ_j^t). Fig. 9 illustrates the blockwise processing. The contexts corresponding to pixels of the last line of the middle block are generated during one program cycle. Meanwhile, the lines of the right block are read and stored. So one program cycle handles B contexts. For this purpose, a set of primitive instructions is defined and before the coding or decoding process starts, the context template is compiled into a program which is loaded into the hardware. Each program line (or program instruction) is a form of microcode, meaning that it can hold one of each of the primitive instructions which can be executed in parallel. The compiler is not implemented in hardware since that task needs to be done only once.

4.2. Architecture

The basic building blocks of the architecture are dedicated registers for each individual context bit. Fig. 10 shows such a 'context register' C_t, with

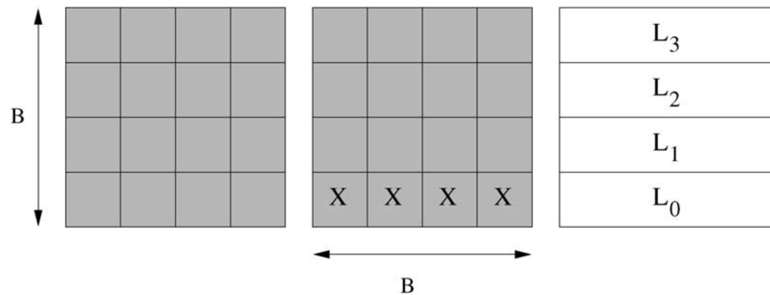


Fig. 9. The hardware design is based on processing the data block by block ($B \times B$ pixels per block). The contexts of last line of the middle block (crossed pixels) are generated, while reading the lines of the right block.

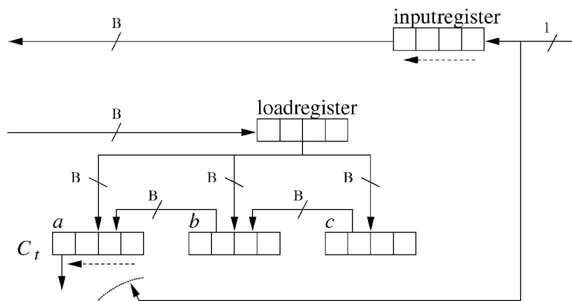


Fig. 10. The basic building block of the architecture is the context register. There is one context register for each context bit. The load- and inputregister are common for all context registers.

$1 \leqslant t \leqslant q$, where B equals 4. The context register consists of three parts: a shift register a, and two regular registers b and c. Part b can be dropped fully into a, and part c can be dropped fully into b. The first bit of shift register a of every context register forms the current context. All three parts can be loaded separately from a common single loadregister. The loadregister itself is filled up by the buffer with words of the block which is currently being read (the right block of Fig. 9), but possibly of previous lines. A common input-register is used for decoding only.

The number of parts a context register is constituted of, is a tradeoff between compiler complexity, coding speed and physical area. The complexity is determined by the total size of the regular registers which serve as a workspace for context preparation, by the border effects, and by the degree of optimization one wants to achieve. A complex compiler is to be avoided as it is very hard to verify whether it behaves correctly in all cases. It would be possible to use context registers of only two parts each: one shift register a and one regular register b. But this would make the compiler significantly more complex since the initial loading of the registers would be much harder to optimize. At the same time, it cannot give us as an increase in speed. Context registers of four or more parts are also possible as this is a straightforward extension of the current scheme. This would not give us a significant increase in speed but would make correct handling of border effects more cumbersome.

As mentioned before, decoding is more difficult than encoding if the template contains pixels from the current row. To take care of this data dependency, a special facility was provided to bring bits that are just decoded right into the context registers. For that purpose, a branch at the inputregister is able to transfer such bits into part a of the context registers. The branch is controlled and enabled by a (hidden) bypassregister, which is loaded at initialization time.

4.3. Microcode generation

We are now able to identify the basic instructions to make this architecture work. First of all, we need to load lines into the loadregister (`LOAD`) and next into one of the registers of a context-register (`LOADTO`). We also need to output the currently available context, this is the first bit of each context register (`READY`). Finally, when the shift register a of a context register is empty, we

need to refill it from register b (DROP). These instructions constitute together a program which generates $B \times q$ contextbits. We summarize the instructions and now also mention the arguments:

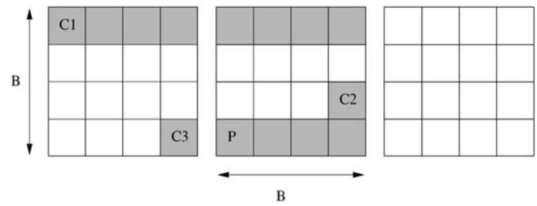
- DROP(t) moves part b of context register C_t into part a and part c into b;
- LOAD(l) loads part of row l (of the next block) from the buffer into the loadregister;
- LOADTO(p, t) copies the loadregister into part p of C_t;
- READY means the context composed by the outputs of all the context registers is ready and can be sent to the QM-block;
- STORE is for decoding purposes only. The inputregister, containing the decoded bitstream, is filled and ready to be stored externally.

It is very instructive to show how a program can easily be constructed by using these primitive instructions. Fig. 11(a) shows an example for the template $T = \{(-4,-3),(3,-1),(-1,0)\}$. The contextbits of the first context to be generated are marked by 'C1', 'C2' and 'C3'. Since the right block is going to be read, the gray pixels should already be available in the respective context registers. The block boundaries need to be conserved, so the initial filling of the context registers in this example is given by Fig. 11(b). More formally, define $a \uparrow b$ as $b\lceil (a+1)/b \rceil$, the smallest multiple of b larger than a. Then for every template pixel (δ_i^t, δ_j^t), part a is filled with bit $(i + \delta_i^t, j + \delta_j^t)$, $(i + \delta_i^t + 1, j + \delta_j^t)$, etc. up till bit $(i + \delta_i^t \uparrow B - 1, j + \delta_j^t)$. Then, if $\delta_i^t < 0$, part b is filled with the next B bits $(i + \delta_i^t \uparrow B, j + \delta_j^t)$ till $(i + \delta_i^t \uparrow B + B - 1, j + \delta_j^t)$. The program construction is now driven by the following two objectives:

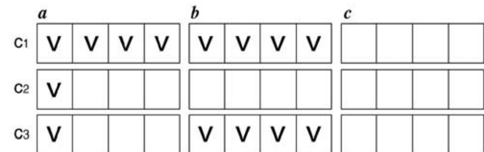
(1) output a context as soon as available,
(2) read a line for a context register that needs them first (i.e., in order of decreasing δ_i^t).

Of course, the precise program construction requires the template to be known. Each program cycle generates B contexts and hence contains at least B instruction cycles, but usually a few more than that. Each instruction cycle can contain at most one READY, B DROP's, one LOAD, one STORE and one LOADTO instruction, which are executed in parallel. For example, the content of the loadregister due to a LOAD instruction can be copied to a (part of a) context register by a LOADTO instruction in the next instruction cycle, but another LOAD instruction can again be executed in this second instruction cycle. These compound instructions allow us to simultaneously execute a large part of the architecture. For the given constraints, which determine the range of the arguments and hence the required number of bits, the instruction word length is 29 bits. An instruction is coded as follows: 1 bit for end-of-program, 2 bits for READY (encoding and decoding respectively), 1 bit for

(a) Template for the current pixel 'P'



(b) Context registers (v=valid)



(c) Microcoded program (encoding)

```
1: READY   DROP(3)  LOAD(1)
2:                  LOAD(3)  LOADTO(a,2)
3: READY            LOAD(0)  LOADTO(c,1)
4: READY                     LOADTO(b,3)
5: READY   DROP(1)
```

(d) Microcoded program (decoding)

```
1: READY            LOAD(1)
2:                  LOAD(3)  LOADTO(a,2)
3: READY                     LOADTO(c,1)
4: READY
5: READY   DROP(1)
6: STORE
```

Fig. 11. An example for a small template with blocksize $B = 4$. (a) The template is $T = \{(-4,-3),(3,-1),(-1,0)\}$. (b) The state of the context buffers at the beginning and the end of a program cycle. (c) The microcoded program for encoding needs 5 instruction cycles. Note that the LOAD instructions process the right block. (d) The microcoded program for decoding needs 6 instruction cycles. Note that the bypass forwards the decoded pixels right into the first bit of the context register C_1. The STORE instruction saves the decoded pixels.

STORE, 13 bits for DROP, 6 bits for LOAD and 6 bits for LOADTO. The CM-block is organized in a pipelined fashion to allow the fastest instruction fetching and execution.

The program is assembled externally and downloaded as a part of the initialization. A compiler is developed to convert an arbitrary template into a program by the construction explained above: firstly, it determines how the context registers are filled at the beginning of every program cycle; secondly, the program is built by the two objectives mentioned (output if ready, input as soon as possible). Note that the context registers are filled exactly the same way at as they were at the beginning of a program cycle.

Fig. 11(c) shows the assembled program for our small example. Each program cycle takes 5 instruction cycles to generate 4 contexts.

4.4. Experimental results

Our design was carried out to synthesis with Synopsys. The results of synthesis were also checked with a simulator. These simulations did not yet take into account exact information about placing and routing. We opted for SGS Thomson HCMOS7, a 0.25 μm standard cell technology supporting up to 6 metal layers, aiming at a clock frequency of 400 MHz. The total surface area for the CM-block is only 0.68 mm^2. This leaves us ample of space to insert the Qx-core and provide memory for the model and additional I/O-buffering. The Qx-coder is able to process on average 0.85 contexts per clock cycle [15]. We now discuss the performance of the hardware design of the CM-block and Qx-core together.

We first discuss the typical case for encoding and decoding. A typical template requires approximately 32–33 program instructions, i.e., every instruction cycle is able to output a context. Therefore, the Qx-core can be used at its full capabilities, generating 340 Mpix/s for encoding. For decoding, we must take into account the critical path of template pixels at the current line ($\delta_j^t = 0$). For example, the very useful template pixel $(-1, 0)$ requires the CM-block to stall on average one clock cycle in order to receive the decoded pixel, which is required for the next context. This data dependency slows down the decoder to 170 Mpix/s. For typical template sizes, where q is significantly less than B, the program length does not depend on the template size. Table 3 summarizes these results.

Table 3
Overview of the experimental results obtained by the hardware design[a]

	Encoding		Decoding	
	instr.	Mpix/s	instr.	Mpix/s
Worst case ($q = 6$)	39	328	39	164
Typical ($q = 6$)	32	340	33	170
Worst case ($q = 12$)	45	284	45	140
Typical ($q = 12$)	32	340	33	170

[a] The typical template for order 6 is $T = \{(-1,0), (-2,0), (0,1), (10,3), (-3,11), (-12,12)\}$, the typical template for order 12 is $T = \{(-1,0), (-2,0), (-3,0), (-1,-1), (10,-2), (11,-3), (10,-3), (-14,-7), (-14,-8), (-3,-11), (-12,-12)\}$.

Based on the considerations for the hardware design, we can also easily derive the worst case scenario for encoding and decoding. For encoding, if all template elements are at the most right column ($\delta_i^t = 31$, $\delta_j^t = -t$ for $1 \leqslant t \leqslant q$), then the program needs to fill every context register before the second context can be generated. This worst case template requires $33 + q$ instructions for encoding. The worst case encoder runs at 284 Mpix/s for order 12. For decoding, the worst case template contains $(-1, 0)$ combined with template elements of the most right column. The worst case decoder runs at 140 Mpix/s for order 12. Fortunately, the worst case is not very useful in practice because it has not much predicting capabilities.

5. Conclusion

Context modeling using an adaptive template allows for improved compression on halftone images. Both JBIG-standards allow for one or

four adaptive template pixels. Using a straightforward implementation for JBIG2, the adaptivity of these four pixels causes the coding and decoding speed to be about four times lower than when the template is fixed. In this paper, we propose an improved software and hardware implementation for adaptive template context modeling. The software implementation is based on efficient buffering, complementary line shifting and a fast bit transposition. Using the same QM-implementation, our software implementation achieves speeds in the order of 10–15 Mpix/s, which is comparable with the JBIG-kit implementation but the template pixels can all be adaptive in our implementation. We also propose a fast implementation of the binary autocorrelation function, which performs about 10^9 bit comparisons per second on our platform. The hardware implementation is based on a special-purpose register, i.e., the context register, which enables us to efficiently handle pixels belonging to different contexts. The hardware design makes use of an off-line compiler which translates the context template into a microcoded program. Using 0.25 μm standard cell technology, we are able to obtain encoding speeds of 340 Mpix/s, and decoding speeds of 170 Mpix/s.

Acknowledgements

This work was financially supported by the Fund for Scientific Research – Flanders (Belgium), through two mandates of Research Assistant and through the projects 39.0051.93 and 31.5831.95, and by the Flemish Institute for the Advancement of Scientific-Technological Research in Industry (IWT) through the projects Tele-Vision (IWT 950202) and Samset (IWT 950204). The authors would like to thank Hans De Stecker from Barco Graphics, and Jan Van Campenhout and Koen De Bosschere from Ghent University for their support. Our thanks are also due to the anonymous referees whose comments significantly helped to improve the manuscript.

References

[1] T.C. Bell et al., Text Compression, Advanced Reference Series Computer Science, Prentice-Hall, Englewood Cliffs, NJ, USA, 1990.

[2] K. Denecker et al., Bit-oriented context modeling of the prediction error residue for lossless image compression, in: M.H. Hamza (Ed.), Proceedings of the IASTED International Conference Computer Graphics and Imaging, IASTED, ACTA, Halifax, Nova Scotia, Canada, June 1998, pp. 138–141.

[3] K. Denecker et al., Context-based lossless halftone image compression, J. Electron. Imaging 8 (4) (1999) 404–414.

[4] P.G. Howard et al., The emerging JBIG2 standard, IEEE Trans. Circuits Systems Video Technol. 8 (7) (November 1998) 838–848.

[5] International Telecommunication Union, Telecommunication standardization sector (ITU-T), Information Technology—Coded Representation of Picture and Audio Information—Progressive Bi-Level Image Compression, 1993, ITU-T Recommendation T.82—ISO/IEC International Standard 11544, 1993.

[6] International Telecommunication Union, Telecommunication standardization sector (ITU-T), Information Technology—Coded Representation of Picture and Audio Information—Lossy/Lossless coding of Bi-Level Images, March 1999, ITU-T Recommendation T.82—ISO/IEC International Standard 14492 Final Committee Draft.

[7] H.R. Kang, Digital Color Halftoning, SPIE/IEEE Press, New York, 1999.

[8] B. Martins, S. Forchhammer, Tree coding of bilevel images, IEEE Trans. Image Process. 7 (4) (April 1998) 517–528.

[9] J.L. Mitchell, W.B. Pennebaker, Software implementations of the q-coder, IBM J. Res. Dev. 32 (6) (November 1988) 753–774.

[10] A. Moffat, Two-level context based compression of binary images, in: J.A. Storer, J.H. Reif (Eds.), Proceedings of the IEEE Data Compression Conference, IEEE Computer Society Press, Snowbird, UT, USA, April 1991, pp. 382–391.

[11] W.B. Pennebaker et al., An overview of the basic principles of the Q-coder adaptive binary arithmetic coder, IBM J. Res. Dev. 32 (6) (November 1988) 717–726.

[12] W.B. Pennebaker, J.L. Mitchell, JPEG Still Image Compression, Van Nostrand Reinhold, New York, NY, USA, 1993.

[13] J.J. Rissanen, A universal data compression system, IEEE Trans. Inform. Theory IT-29 (5) (September 1983) 656–664.

[14] K. Sayood, Introduction to Data Compression, Morgan Kaufmann Publishers, Los Altos, CA, 1996.

[15] M.J. Slatter, J.L. Mitchell, The Qx-coder, IBM J. Res. Dev. 42 (6) (November 1998) 717–784.

[16] R. Ulichney, Digital halftoning, MIT Press, Cambridge, MA, 1987.