GFWX: GOOD, FAST WAVELET CODEC
ICT TECH REPORT ICT-TR-01-2016

Graham Fyffe

University of Southern California Institute for Creative Technologies

ABSTRACT

Wavelet image compression is a popular paradigm for lossy and lossless image coding, and the wavelet transform, quantization, and entropy encoding steps are well studied. Efficient implementation is straightforward for the first two steps using e.g. lifting and uniform scalar deadzone quantization, but entropy encoding is typically carried out using complex context modeling and arithmetic coding. We propose a simple entropy encoding scheme for wavelet coefficients based on limited-length Golomb-Rice codes, and we propose two simple context schemes for selecting the Golomb parameter; one slightly faster than the other. We also propose a simple solution to the rounding problem in integer wavelet transforms, allowing the use of integer transforms for both lossless and lossy compression instead of resorting to floating point. If the input is Bayer patterned data, we include an additional lifting step allowing lossy compression without demosaicing. We demonstrate that a straightforward implementation of a complete codec in under 1000 lines of C++ is several times faster than JPEG 2000 while producing similar file sizes, without sacrificing certain desirable features such as downsampled decoding and progressive decoding of incomplete data streams.

***Index Terms*—** Wavelets, image compression, JPEG 2000, Golomb-Rice codes, Bayer pattern compression.

1. INTRODUCTION

Wavelet image compression generally follows three steps: wavelet transform, quantization, and entropy encoding [6]. Decompression consists of the reverse steps: entropy decoding, dequantization, and inverse transform. Many image compression-decompression libraries, or *codecs*, have been described in the literature using wavelets, e.g. EZW [11] and SPIHT [10]. By far the most popular standard is JPEG 2000 [14], which enjoys several open source implementations including OpenJPEG [2] and JasPer [1]. Though popular, the JPEG 2000 standard is complex, and some effort has been made to propose standards that are computationally simpler yet retain most of the benefits of JPEG 2000, for example the CCSDS recommendation [16]. In this work, we propose a simple wavelet image compression pipeline that is several times faster than JPEG 2000 while producing similar sizes. Our proposed pipeline uses only integer arithmetic, without the rounding problems typically associated with integer lossy wavelet encoding. We provide an implementation in under 1000 lines of C++ source code using only standard C++ libraries, with many of the features that make JPEG 2000 attractive including optional downsampled decoding, and progressive decoding of incomplete data streams. Sections 2 through 5 describe the steps of our proposed pipeline, and Section 6 discusses the implementation and evaluation.

2. INTEGER COLOR TRANSFORM

Our pipeline begins with an optional user-programmable color transform. Let the input data represent one or more non-interleaved image *layers*, each pixel having one or more *interleaved channels*. The memory layout of the data is hence ordered by layers, then rows, then columns, and finally interleaved channels. Conceptually, each spatial image location has one data value per layer per interleaved channel, which we collectively call *channels*. We support any transform that can be expressed as a sequence of steps of the following form:

$$C_i \leftarrow C_i + \lfloor \tfrac{1}{s} \sum_{t=1}^{n} k_t C_{j_t} \rceil_0^{\downarrow}, \tag{1}$$

where C_i is the value of channel i, s is an integer divisor, $j_t \neq i$ are summand channel indices, k_t are integer weights, and $\lfloor x \rceil_0^{\downarrow} = \mathrm{sign}(x)\lfloor |x| \rfloor$ represents rounding towards zero. This general form is trivially reversible, and allows many popular color transforms to be implemented. For example, the YUV transform can be implemented for RGB input as:

$$\begin{aligned} C_0 &\leftarrow C_0 + \lfloor -C_1 \rceil_0^{\downarrow}; \\ C_2 &\leftarrow C_2 + \lfloor -C_1 \rceil_0^{\downarrow}; \\ C_1 &\leftarrow C_1 + \lfloor \tfrac{1}{4}(C_0 + C_2) \rceil_0^{\downarrow}, \end{aligned} \tag{2}$$

and the $\mathbf{A}_{7,10}$ transform [12] can be implemented as:

$$\begin{aligned} C_0 &\leftarrow C_0 + \lfloor -C_1 \rceil_0^{\downarrow}; \\ C_2 &\leftarrow C_2 + \lfloor \tfrac{1}{2}(-C_0 - 2C_1) \rceil_0^{\downarrow}; \\ C_1 &\leftarrow C_1 + \lfloor \tfrac{1}{8}(3C_0 + 2C_2) \rceil_0^{\downarrow}. \end{aligned} \tag{3}$$

During encoding, the result of the color transform is stored in a temporary buffer with a larger integer precision than the input data, which facilitates later steps that produce intermediate values having additional bits of dynamic range. For example, 8-bit data is transformed into a 16-bit temporary buffer, and 16-bit data is transformed into a 32-bit temporary buffer.

3. INTEGER WAVELET TRANSFORM

After the color transform, we apply an in-place lifting scheme [13] to compute the wavelet transform per channel. Like JPEG 2000, we recommend using the 5/3 wavelet for lossless encoding, and the 9/7 wavelet for lossy encoding, however we support both modes for both encoding types. Normally, the 9/7 wavelet transform is implemented for lossy encoding using floating-point arithmetic, because integer wavelet transforms based on lifting suffer from a rounding problem, where rounding of residuals prevents proper smoothing of low frequency values. We solve this using what amounts to fixed point arithmetic, without any impact on encoding or decoding speed. We simply multiply the input values by 8, which we fold into the color transform step. The extra 3 bits of precision is enough to smooth the low frequency values. Using a value greater than 8 is also possible, but may cause integer overflow in our implementation of the 9/7 wavelet transform. After the transform, we divide the output by 8, which we fold into the quantization step. We employ this scheme for both 5/3 and 9/7 wavelets, but only for lossy compression, as it is not strictly reversible. We additionally clamp the cubic predictions of the 9/7 wavelet transform to lie within the range defined by the two center samples of the cubic, which reduces ringing artifacts and slightly improves compression. If the input is Bayer patterned data, it is known that the first level of a wavelet transform effectively decorrelates the color channels into the LL, LH, HL, and HH subbands [17, 5], however the LH, HL, and HH subbands still exhibit significant spatial correlation. Therefore in the case of Bayer data we repeat the entire wavelet transform on the first-level LH, HL and HH subbands, improving decorrelation and hence compression.

4. QUANTIZATION

After the wavelet transform, lossy compression may be achieved by *quantizing* the coefficients. We employ uniform scalar deadzone quantization (USDQ) [8]:

$$c_{x,y} \leftarrow \lfloor Q_{x,y} c_{x,y} \rceil_0^{\downarrow}, \tag{4}$$

where $Q_{x,y} = \max(q_{\min}, \min(q_{\max}, 2^{L_{x,y}} q))/q_{\max}$, $c_{x,y}$ is the coefficient at location (x, y), $q_{\min}$ is the minimum quality, $q_{\max}$ is the maximum quality, q is the user-selected quality parameter, and $L_{x,y}$ is the wavelet transform level at location (x, y), where the finest level has $L = 0$ and coarser levels have $L > 0$. A form of chroma downsampling can be effected by using a smaller value of q for channels that are flagged as chroma, for example $q_{\text{chroma}} = \lfloor q_{\text{luma}}/d \rfloor$ for some $d \geq 1$. In the case of Bayer data, the LL, LH, HL, and HH subbands of the first transform level are treated as four separate channels, allowing lossy compression without demosaicing. The LH, HL, and HH subbands are considered chroma, but as they contain some residual luma we let $q_{\min} = q_{\text{luma}}$.

5. ENTROPY ENCODING

After quantization, we encode the wavelet coefficients in level order, starting from the DC coefficient. Within each level, we divide the pixels into blocks that are 2^b pixels on each side for some user-selected b. We encode blocks in scanline order, and we encode the channels within a block sequentially and independently. This allows the encoding task to be divided into a number of parallel encoding subtasks equal to the number of blocks times the number of channels, called *block-channels*. Since the length of the encoded block-channels is unknown beforehand, we divide the remaining encoding buffer into equal parts on word boundaries and encode each block-channel into its own part. For each block-channel, we encode wavelet coefficients in scanline order using limited-length Golomb-Rice codes as described in Subsections 5.1 and 5.2. After encoding, we tightly pack the encoded block-channels in sequence, exploiting the word alignment of the encoded buffers for efficiency. For speed and simplicity, we encode whole coefficients instead of multiple bitplane passes, which means we lose the *embedded* encoding property prevalent in most wavelet image codecs. While embedded encoders are progressive in quality (where each transmitted bit optimally improves a quality measure such as PSNR), our proposed encoder is progressive in resolution (where each transmitted level doubles the resolution of the decodable image). We argue that this is a desirable feature for today's so-called responsive media applications, where a variety of devices may request different resolutions from a single image resource.

5.1. Limited-Length Golomb-Rice Codes

We encode the wavelet coefficients in a block-channel using limited-length Golomb-Rice codes [9], or zero run codes when the probability of zero is high. Golomb-Rice coding with zero runs is popular for encoding prediction residuals in e.g. JPEG-LS [15], and we find them equally effective for encoding wavelet coefficients. Rather than constructing a custom code table based on symbol statistics as in Huffman coding, Golomb coding implements a computationally simple code with just one free integer parameter, called the *Golomb parameter*. An integer $x \geq 0$ is encoded as the unary representation of $\lfloor x/m \rfloor$ (so many zeros followed by a one) followed by the binary representation of $x \bmod m$, where m is the Golomb parameter. Golomb-Rice codes are Golomb codes where m is a power of two, which admit extremely sim-

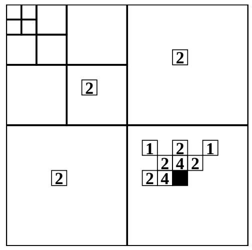


Fig. 1. Schematic of the neighborhood used to compute the context. The coefficient is shown as a black square, and the neighbors are shown as numbered squares (numbers represent relative weight), including eight neighbors in the same subband, two neighbors in the neighboring subbands, and one neighbor in the parent band. Only neighbors that precede the coefficient in scanline order may be used. For clarity, we show the wavelet decomposition in Mallat configuration, though our implementation uses an in-place transform.

ple implementations using bit operations. Due to the unary part, the length of a Golomb code can be quite large if m is small relative to x. In such a case, it is customary to choose a limit l on the value of $\lfloor x/m \rfloor$, and use this limit to signal an escape to raw binary encoded data. For example, suppose x were a 32-bit value, and $\lfloor x/m \rfloor \geq l$, then one would emit l zeros signifying an escape, and then emit x in binary using 32 additional bits. This limiting scheme has several deficiencies: it requires advance knowledge of the range of x; it is redundant as any $x < lm$ has two representations; and the implied probability distribution changes abruptly from geometric to flat at the limiting threshold. We propose instead a more gradual limiting scheme. Simply, if $\lfloor x/m \rfloor \geq l$, then we emit l zeros to signify an escape, and then we recursively encode $x - lm$ using a larger modulus (we found $16m$ works well). The advantages of this technique are that the range of x need not be known; every x has exactly one representation; and the implied probability distribution flattens gradually as x increases, which improves compression in our tests. For signed data, such as wavelet coefficients, two schemes are common. The first is to interleave positive and negative values in sequence, i.e. $0, 1, -1, 2, -2, \ldots$ and encode the index of x in this sequence, which we call *interleaved* codes. The second is to encode $|x|$ followed by a sign bit if $x \neq 0$, which we call *signed* codes. We note that the implied probability distribution differs between these two schemes, and so rather than choosing one over the other, we use both. We encode zero run lengths using Golomb-Rice codes, and we encode wavelet coefficients using either interleaved Golomb-Rice codes or signed Golomb-Rice codes. Switching between these three methods and selecting the Golomb parameter in each case is described in the next subsection.

5.2. Selecting the Golomb Parameter

Optimal selection of the Golomb parameter m is possible if the data comes from a geometrically distributed source with

---

Algorithm 1 Map $(\hat{\mu}_1, \hat{\mu}_2)$ to coding method, with $B_0 = 2\hat{\mu}_2 + 100$, $B_1 = 2\hat{\mu}_2 + 250$, $B_2 = 2\hat{\mu}_2 + 950$, $B_3 = 3\hat{\mu}_2 + 3000$, $B_4 = 5\hat{\mu}_2 + 400$, $B_5 = 3\hat{\mu}_2 + 12000$, $B_6 = 5\hat{\mu}_2 + 3000$, $B_7 = 4\hat{\mu}_2 + 44000$, and $B_8 = 6\hat{\mu}_2 + 12000$.

---

if (is luma **and** $\hat{\mu}_1^2 < B_0$) **or** (is chroma **and** $\hat{\mu}_1^2 < B_1$) **then**
  use interleaved code with $m = 1$
else if $\hat{\mu}_1^2 < B_2$ **then** use interleaved code with $m = 2$
else if $\hat{\mu}_1^2 < B_3$ **and** $\hat{\mu}_1^2 < B_4$ **then** use signed code with $m = 2$
else if $\hat{\mu}_1^2 < B_3$ **then** use interleaved code with $m = 4$
else if $\hat{\mu}_1^2 < B_5$ **and** $\hat{\mu}_1^2 < B_6$ **then** use signed code with $m = 4$
else if $\hat{\mu}_1^2 < B_5$ **then** use interleaved code with $m = 8$
else if $\hat{\mu}_1^2 < B_7$ **and** $\hat{\mu}_1^2 < B_8$ **then** use signed code with $m = 8$
else if $\hat{\mu}_1^2 < B_7$ **then** use interleaved code with $m = 16$
else use signed code with $m = 16$

---

known mean μ [7]. A popular selection strategy that is always close to optimal is to let $m = 2^k$ where $2^k \leq \mu < 2^{k+1}$. However, the distribution of wavelet coefficients in real-world images is not strictly geometric, and therefore we propose a scheme based on the first moment $\mu_1 = \mathrm{E}[|x|]$ and second moment $\mu_2 = \mathrm{E}[x^2]$, to capture richer distribution characteristics. We propose two methods to estimate (μ_1, μ_2). The first method is to examine a neighborhood around a coefficient shown in Fig. 1, and compute the following estimates:

$$\hat{\mu}_1 = \mathrm{round}\left(\frac{16}{\sum_{i \in N} w_i} \sum_{i \in N} w_i |y_i|\right);$$
$$\hat{\mu}_2 = \mathrm{round}\left(\frac{16}{\sum_{i \in N} w_i} \sum_{i \in N} w_i \min(4096, |y_i|)^2\right), \quad (5)$$

where N is the set of neighbors that exist considering image boundaries and scanline order precedence, y_i is the coefficient at the ith neighbor, and w_i is its weight. Scaling the moments by 16 lets us use integer arithmetic, and limiting the values to 4096 avoids overflow later on. The second method, which is somewhat faster but less effective, is to update a running estimate of the two moments after each value x is encoded:

$$\hat{\mu}_1 \leftarrow \mathrm{round}\left(\tfrac{15}{16}\hat{\mu}_1\right) + |x|;$$
$$\hat{\mu}_2 \leftarrow \mathrm{round}\left(\tfrac{15}{16}\hat{\mu}_2\right) + \min(4096, |x|)^2. \quad (6)$$

We call $(\hat{\mu}_1, \hat{\mu}_2)$ the *context* of the current coefficient. To design a mapping from context to coding method, we collected all the wavelet coefficients belonging to each context using (5) over the Kodak lossless true color image suite [3] and several photographs of cats collected from the Internet. We then found the best method for each context by brute force, compressing the coefficients using both interleaved and signed codes with power-of-two Golomb parameters ranging from 1 through 16. After inspecting the distribution of best coding methods over the context landscape, we designed a partitioning of the landscape using quadratic boundaries, described in Algorithm 1. We also use quadratic boundaries on $(\hat{\mu}_1, \hat{\mu}_2)$ to

Table 1. Lossless compression results (bytes) for our method vs. JPEG 2000. (Y) indicates a grayscale version of the image, using only the luma channel. Best sizes are highlighted in bold.

Image	Ours	JP2K	Ours(Y)	JP2K(Y)
Kodak01	513,752	**510,526**	**263,412**	267,235
Kodak02	465,084	**450,482**	**202,704**	207,203
Kodak03	398,544	**397,835**	**173,972**	175,538
Kodak04	463,872	**460,159**	**202,332**	206,711
Kodak05	532,040	**531,776**	**255,936**	260,523
Kodak06	475,884	**471,536**	**231,788**	233,317
Kodak07	419,540	**418,095**	**184,804**	185,476
Kodak08	554,260	**547,613**	**269,668**	271,490
Kodak09	454,500	**445,098**	**195,764**	196,662
Kodak10	459,748	**453,211**	**198,564**	201,113
Kodak11	460,076	**456,826**	**220,736**	223,955
Kodak12	**423,244**	425,654	**192,124**	193,588
Kodak13	**581,164**	583,094	**293,760**	300,050
Kodak14	500,804	**499,524**	**243,224**	247,866
Kodak15	**440,072**	442,318	**192,404**	195,095
Kodak16	433,860	**431,410**	**203,664**	205,678
Kodak17	453,940	**451,947**	**202,220**	207,277
Kodak18	549,572	**546,925**	**247,108**	252,670
Kodak19	493,036	**482,870**	**221,572**	223,442
Kodak20	405,004	**397,111**	**180,740**	181,948
Kodak21	487,132	**479,938**	**225,624**	228,514
Kodak22	503,516	**496,250**	**223,064**	226,958
Kodak23	421,288	**418,135**	**171,516**	173,523
Kodak24	**497,996**	499,855	**231,464**	236,302
Cats	**17,838,708**	18,440,076	**7,818,056**	7,875,114

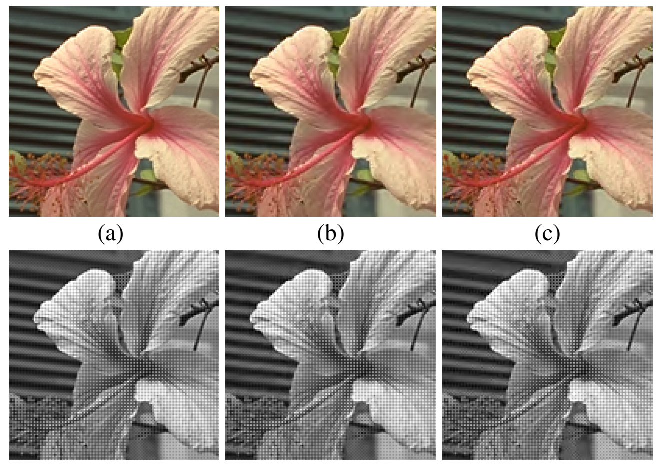
decide when to use zero run coding, and to select the Golomb parameter for encoding the zero run length. (Please see the source code for details.) If we encode a zero run, the next coefficient cannot be zero, and so we add one to the next coefficient if it is negative. This slightly improves compression.

6. EVALUATION AND DISCUSSION

Though we describe only the compression pipeline, constructing the decompression pipeline is straightforward, by reversing the steps. We implemented the proposed compression pipeline and corresponding decompression pipeline using C++11 templates, to facilitate customizing the data source and to support 8- and 16-bit signed and unsigned data without extra code. The entire implementation is under 1000 lines of C++11 in a single header file with no dependencies besides standard C++ libraries. We invite the reader to view the code at `http://www.gfwx.org`, and indeed to use the code as it is released under the 3 clause BSD license. We evaluated our method on the Kodak lossless true color image suite [3] and a large image comprising several photographs of cats collected from the Internet [4]. Table 1 lists lossless compression results for our method vs. the JasPer JPEG 2000 library [1] for the original color images and for grayscale versions of the same. For all color images, we employed the $\mathbf{A}_{7,10}$ color transform [12] as it produced smaller overall size than other transforms we tried. We employed the somewhat slower context method from (5), which is still substantially faster than JPEG 2000. Despite the simplicity of our encoding scheme, it outperforms the EBCOT arithmetic encoder used in JPEG 2000 for lossless encoding of all grayscale images tested, and is competitive for color images. Evidently our proposed encoder is superior for luma yet inferior for chroma, which suggests an avenue for future work tuning for chroma. Table 2 lists encoding and decoding times for our method on the large image of cats, showing both context methods and large block size vs. small block size to take advantage of multithreading, vs. the OpenJPEG [2] and JasPer [1] JPEG 2000 libraries, running on a dual quad-core 2.4 GHz Intel Xeon E5620 CPU with hyperthreading. All configurations of our proposed method are faster than JPEG 2000 on this image (up to 17 times faster), while still producing a smaller size. Fig. 2 shows a region of an image compressed to 1 bit per pixel, comparing our proposed method vs. JPEG 2000 (best viewed in high resolution electronic format). JPEG 2000 preserves more fine detail, but also has more ringing artifacts, and introduces problematic artifacts on Bayer pattern data, for which it is not designed. In future work, we could employ advanced quantization schemes such as Trellis coding to further improve quality at low bit rates.

Table 2. Lossless timings on the large image of cats.

Method	Size (bytes)	Encode Time	Decode Time
Ours (slow, large block)	17,838,708	5.1s	6.1s
Ours (slow, small block)	17,931,684	1.0s	1.2s
Ours (fast, large block)	18,276,476	2.2s	3.1s
Ours (fast, small block)	18,371,604	0.6s	0.7s
OpenJPEG	18,440,016	14.0s	12.0s
JasPer	18,440,076	10.6s	9.1s



(a) (b) (c)
(d) (e) (f)

Fig. 2. A region of an image compressed to 1 bit per pixel. (a) Original. (b) Proposed method. (c) JPEG 2000. (d) Artificial Bayer pattern image. (e) Proposed method on (d). (f) JPEG 2000 on (d).

7. ACKNOWLEDGEMENTS

This work was sponsored by the U.S. Army Research Laboratory (ARL) under contract number W911NF-14-D-0005, and the USC Shoah Foundation.

8. REFERENCES

[1] Jasper: An open-source JPEG 2000 codec. `https://www.ece.uvic.ca/~frodo/jasper`. Accessed: 2015-11-1.

[2] Openjpeg: An open-source JPEG 2000 codec written in C. `http://www.openjpeg.org`. Accessed: 2015-11-1.

[3] Kodak lossless true color image suite. `http://r0k.us/graphics/kodak`, 1999. Accessed: 2015-11-1.

[4] Cat poster 1. `https://commons.wikimedia.org/wiki/File%3ACat_poster_1.jpg`, 2012. Accessed via Wikimedia Commons: 2015-11-1.

[5] K.-H. Chung and Y.-H. Chan. A fast reversible compression algorithm for Bayer color filter array images. In *APSIPA ASC 2009*. Asia-Pacific Signal and Information Processing Association, October 2009.

[6] M. Hilton, B. D. Jawerth, and A. Sengupta. Compressing still and moving images with wavelets. *Multimedia Systems*, 2:218–227, 1994.

[7] A. Kiely. Selecting the Golomb parameter in Rice coding. IPN Progress Report 42-159, November 2004.

[8] M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner. An overview of quantization in JPEG 2000. *Signal Processing: Image Communication*, 17(1):73 – 84, 2002. JPEG 2000.

[9] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *Communication Technology, IEEE Transactions on*, 19(6):889–897, December 1971.

[10] A. Said and W. Pearlman. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. *Circuits and Systems for Video Technology, IEEE Transactions on*, 6(3):243–250, Jun 1996.

[11] J. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *Signal Processing, IEEE Transactions on*, 41(12):3445–3462, Dec 1993.

[12] T. Strutz. Multiplierless reversible color transforms and their automatic selection for image data compression. *Circuits and Systems for Video Technology, IEEE Transactions on*, 23(7):1249–1259, July 2013.

[13] W. Sweldens. The lifting scheme: A new philosophy in biorthogonal wavelet constructions. In *Wavelet Applications in Signal and Image Processing III*, pages 68–79, 1995.

[14] D. S. Taubman and M. W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[15] M. J. Weinberger, G. Seroussi, and G. Sapiro. From LOCO-I to the JPEG-LS standard. In *Proc. of the 1999 Intl Conference on Image Processing*, pages 68–72, 1999.

[16] P.-S. Yeh, P. Armbruster, A. Kiely, B. Masschelein, G. Moury, C. Schaefer, and C. Thiebaut. The new CCSDS image compression recommendation. In *Aerospace Conference, 2005 IEEE*, pages 4138–4145, March 2005.

[17] N. Zhang and X. Wu. Lossless compression of color mosaic images. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, volume 1, pages 517–520 Vol. 1, Oct 2004.