# SOFTWARE ARCHITECTURES FOR JPEG2000

*David Taubman*

The University of New South Wales, Sydney, Australia

d.taubman@unsw.edu.au

**Abstract:** This paper is concerned with software architectures for JPEG2000. The paper is informed by the author's own work on the JPEG2000 VM (Verification Model) and the Kakadu implementation of the standard. The paper describes non-obvious implementation strategies, state machines, predictive algorithms and internal representation techniques which are used to minimize memory consumption and maximize the throughput of these implementations.

## 1. INTRODUCTION

JPEG2000 [1] is the most recent image compression standard to emerge from the ISO working group commonly known as JPEG (Joint Photographic Experts Group). The new standard emphasizes scalable image representations. Portions of the compressed code-stream may be extracted and decompressed independently, to recover the image at a reduced resolution, at a reduced quality (SNR) within any given resolution, or within a reduced spatial region, at the desired resolution and quality. JPEG2000 also supports entirely lossless compression of images without sacrificing scalability. This means that an application is able to extract progressively higher quality representations of any given spatial region, leading eventually to a lossless representation of that region. These features provide applications and users with new paradigms for interacting with compressed imagery.

JPEG2000 is inherently more complex than its predecessor, JPEG. Its reliance on the DWT (Discrete Wavelet Transform), and coding of wavelet coefficients in blocks, together imply a significantly higher cost in memory consumption than the baseline JPEG algorithm. The embedded entropy coding algorithm is far from trivial to implement and the code-stream parsing rules involve considerable conceptual complexity.

On the other hand, whereas the JPEG standard describes a collection of related algorithms which often producing incompatible bit-streams, JPEG2000 is designed as a "monolithic" algorithm, to avoid interoperability problems between code-streams and implementations which exploit different features. A well-designed decoder, should be able to utilize its limited computational or memory resources to recover a meaningful image, possibly at a reduced resolution or quality, regardless of the size or bit-depth of the original image.

The challenge for a good implementation is to exploit and expose as many of the standard's beneficial features as possible, while consuming as little memory and computational resources as possible. Both hardware and software architectures are of interest, but this paper focuses exclusively on software solutions. Our goal is to offer insights gained by the author in creating the VM (Verification Model) and Kakadu[1] implementations of JPEG2000. In particular, the paper addresses three central considerations for any implementation of the standard: 1) efficient implementation of the DWT; 2) efficient implementation of the block coder; and 3) efficient management and exploitation of the compressed code-stream. It is worth noting that efficient hardware architectures may look very different from their software counterparts. Some useful hardware design strategies may be found in [2, §17].

[1] Visit *www.kakadusoftware.com* for documents and downloads.

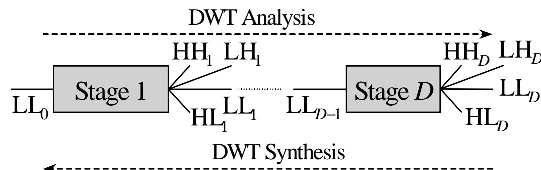


**Fig. 1**. *$D$ stage DWT producing $3D+1$ subbands.*

## 2. EFFICIENT ARCHITECTURES FOR THE DWT

For the sake of this discussion, we consider only one image component (e.g., one colour component), noting that image components are compressed independently in JPEG2000. We also restrict our attention to a single image tile, again noting that JPEG2000 can decompose an image into disjoint rectangular tiles, each compressed independently.

The DWT is most easily described in terms of the iterative application of $D$ processing stages. The $d^{\text{th}}$ stage, decomposes (analysis) its input image $\text{LL}_{d-1}$, into four "subband" images, denoted $\text{LL}_d$, $\text{HL}_d$, $\text{LH}_d$ and $\text{LL}_d$. Each subband image has essentially half the height and width of the input image. We label the original image $\text{LL}_0$ and note that $\text{LL}_1$ through $\text{LL}_{D-1}$ are intermediate results, each serving as input to a subsequent stage. Figure 1 represents this staged view of the DWT. Each stage may be inverted (synthesis), so the image is fully represented by $3D+1$ subbands. If the last $r$ synthesis stages are omitted, $\text{LL}_r$ is recovered; this is a low-resolution image having dimensions $2^r$ times smaller than those of the original. This is the origin of resolution scalability in JPEG2000, since it is possible to reconstruct a reduced resolution version of the image using only a subset of the compressed subbands.

Although the staged view in Figure 1 is convenient as a description of the DWT, it belies the fact that there is no need to finish processing one stage before moving on to the next. A memory efficient implementation need not buffer the original image or any of the intermediate $\text{LL}_d$ images in memory. By processing each stage incrementally, and pipelining the processing stages, it is possible to reduce the working memory requirements to a fixed multiple of either the image height or the image width.

Each DWT stage is formed by the separable composi-

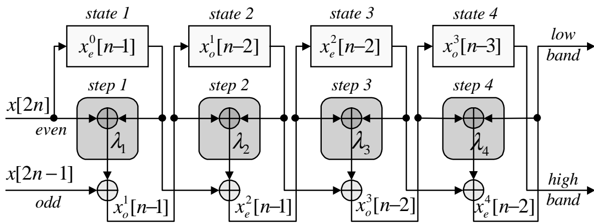


**Fig. 2**. *Lifting state machine for* $9/7$ *wavelet transform.*

tion of two identical 1D operators. The first operator processes each column of the input image, producing columns of a vertical low and a vertical high pass subband image. The second operator processes each row of these vertical subbands, dividing each into two horizontal subbands. It is instructive to consider a trivial case (sometimes called the "lazy wavelet transform"), in which the 1D analysis operator simply separates its input sequence $x\left[n\right]$ (column or row) into even and odd sub-sequences, $x_e\left[n\right] = x\left[2n\right]$ and $x_0\left[n\right] = x\left[2n+1\right]$. In this case, the vertical "low pass" subband consists of all even indexed rows from the input image and the vertical "high pass" subband consists of all odd indexed rows. The horizontal operator then divides each vertical subband into its even and odd indexed columns.

The lazy transform is trivially invertible, but the subbands cannot really be interpreted as low and high pass image components. Interestingly, however, every wavelet transform involving finite support operators may be obtained by applying a sequence of "lifting steps" to the lazy transform [3, 4]. In the case of wavelet transforms involving symmetric filters of odd, least dissimilar lengths, the lifting steps may be compactly described and efficiently implemented using the state machine depicted in Figure 2. The $5/3$ and $9/7$ wavelet transforms described by Part 1 of the JPEG2000 standard [1] both have this form [2, §6.4.4], with 2 and 4 lifting steps, respectively.

Input samples are pushed into the machine in pairs, producing subband samples in pairs. The $9/7$ transform used for the example of Figure 2 requires only 4 sample memories (the states) and 4 multiplications ($\lambda_1$ through $\lambda_4$), to form each new pair of subband samples, even though the subband filtering operations which the machine implements involve filters with 9 and 7 taps for the low and high pass subbands, respectively.

The Kakadu implementation of JPEG2000 takes advantage of this lifting state machine. For DWT analysis, image lines are pushed in pairs to a vertical analysis object, producing one new output line for each vertical subband. Each state in the vertical lifting machine corresponds to single line of buffered samples. The output lines are subjected to horizontal DWT analysis. Once a pair of LL subband lines have been collected, they are recursively passed to the next DWT analysis stage object. Including the cost of buffering image lines into pairs, and noting that the width of the image lines decreases by 2 from stage to stage, the total working memory required to implement the DWT is bounded by $W(2+L)(1+\frac{1}{2}+\frac{1}{4}+\cdots) = (4+2L)W$ samples, where $L$ is the number of lifting steps (2 or 4, depending on the transform) and $W$ is the width of the image.

16-bit fixed-point representations for the intermediate state variables are quite sufficient for processing images with bit-depths up to about 12 bits per sample. DWT synthesis is the reverse of DWT analysis, having an almost identical lifting state machine and identical memory and computational requirements. As noted in the next section, subband samples must be arranged into blocks for coding. This requires significantly more memory than the DWT itself. Including this cost, the Kakadu implementation's processing memory requirements may be expressed as $(4+2L+3J)W$ samples, where $J$ is the code-block height (typically $J = 32$ or $J = 64$) and each sample is 16 bits wide. The Kakadu implementation can also process images column-wise, in which case the memory consumption depends on the height of the image and the width of the code-blocks.

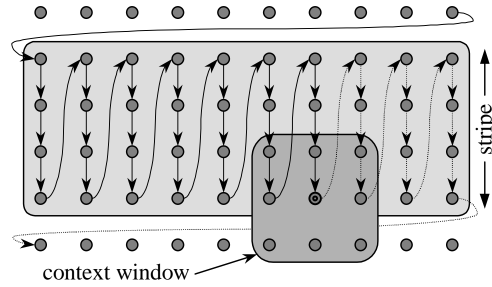


**Fig. 3**. *Stripe-oriented scan through code-block samples.*

## 3. EFFICIENT BLOCK CODING/DECODING

The subband images produced by the DWT are each partitioned into smaller blocks, each of which is independently coded. Typical block sizes are $32 \times 32$ or $64 \times 64$ and each block bit-stream holds an embedded representation of the block's samples. Embedded representations can be truncated to obtain more compact descriptions of the block, with correspondingly larger distortion. Many of JPEG2000's features derive from the efficient block coding algorithm, which allows image quality and spatial regions of interest to be manipulated by appropriately truncating the individual block bit-streams.

Without a doubt, the block coder is the most computationally demanding element in the standard. Our purpose in this brief treatment is to suggest certain non-obvious implementation strategies. While it is not possible to review the algorithm in any detail here, we note that it involves a potentially large number of coding passes, each of which consists of a regular scan through the samples in the block, following the stripe-oriented pattern shown in Figure 3. At each point in the scan, the behaviour of the encoder (or decoder) depends upon context information, which is derived from the significance and sign of the sample under consideration and its eight immediate neighbours, as shown in the figure. A sample is said to become significant when the coding first identifies it as having a non-zero value.

### 3.1. Context State Broadcasting

Since each block requires many coding passes, and each pass requires the collection of context information from a 9 sample window about each location, the implementation cost of the algorithm may at first appear to be quite daunting. Fortunately, however, we may take advantage of the fact that the context associated with any given sample

changes only infrequently. At moderate bit-rates where JPEG2000 exhibits good visual image quality, fewer than 15% of the samples typically ever become significant. For this reason, an efficient software implementation strategy is to associate a "context state word" with each sample in the block. The context word is updated when the sample or any of its immediate neighbours first becomes significant, but this occurs only rarely.

In this way, the coder need not retrieve state information from the 9 samples which affect the coding context at any given location. Instead, the coding process may be described and implemented in terms of operations on the context word. By carefully arranging the state bits in the context word, the context adaptive arithmetic coding steps required to generate or decode the embedded bit-stream may be implemented using only a few small lookup tables and primitive logic operations. On the rare occasions when a sample becomes significant, the changes must be "broadcast" to the context words associated with each of its eight immediate neighbours.

The JPEG2000 Verification Model (from VM3A onwards) employ context broadcasting with a 16-bit context word for each sample. The stripe oriented scan shown in Figure 3, however, lends itself to even more efficient context broadcasting strategies. The Kakadu implementation gains at least 20% speed improvement over the VM, by packing the context information for an entire stripe column (4 samples) into a single 32-bit word.

### 3.2. Predictive Truncation

The JPEG2000 standard draws heavily on the EBCOT (Embedded Block Coding with Optimal Compression) compression paradigm described in [5]. The idea is to generate embedded bit-streams for each code-block in the image in a first pass. A second pass then generates the final code-stream, truncating the individual block bit-streams in an optimal fashion, so as to minimize the reconstructed image distortion for a given overall compressed size. In contrast to JPEG, this allows precise rate control to be achieved without compressing the image multiple times. In fact, the code-block truncation pass generally fragments the code-block bit-streams into multiple "quality layers", such that any leading set of layers will hold optimally truncated block bit-streams corresponding to some bit-rate. Individual layer bit-rates or qualities may be adjusted to satisfy the needs of a wide variety of different applications simultaneously.

One drawback of the EBCOT paradigm is that the block encoder does not know at what point its bit-stream will be truncated by the rate-control pass. For this reason, the encoder generally produces many more compressed bits than will actually be included in the final code-stream. This tends to make JPEG2000 encoders much slower than decoders.

Kakadu partially overcomes this difficulty by implementing a truncation prediction algorithm, consisting of two parts. The first collects statistics from code-blocks which have already been compressed, to form a conservative estimate of the rate-distortion slope threshold which will be used to truncate each block bit-stream. The second part is executed at the end of each coding pass performed by the block encoder, to determine whether any future coding passes are likely to yield a rate-distortion slope which exceeds the threshold determined by the first part. Coding passes whose bits are likely to be truncated are not performed at all.

**Table 1**. *Encode/decode $\mu s$/sample on a 400MHz PII.*

| | 0.25 bpp | 0.5 bpp | 1.0 bpp | 2.0 bpp |
|---|---|---|---|---|
| Full encode | $0.69\mu s$ | $0.69\mu s$ | $0.69\mu s$ | $0.69\mu s$ |
| Slope predict | $0.13\mu s$ | $0.19\mu s$ | $0.29\mu s$ | $0.42\mu s$ |
| Slope known | $0.10\mu s$ | $0.12\mu s$ | $0.19\mu s$ | $0.30\mu s$ |
| Decode | $0.03\mu s$ | $0.05\mu s$ | $0.08\mu s$ | $0.14\mu s$ |

### 3.3. Experimental Observations

Table 1 reports block encoder and decoder processing times, obtained using Kakadu, with a 400 MHz PII processor. CPU times are expressed in $\mu s$ per image sample, and are reported at a variety of compressed bit-rates, measured in bits per image pixel. The ISO/IEC colour test image "Bike," is used for these tests, measuring $2560 \times 2048$ pixels, with 3 colour samples per pixel. Image quality is very good at 1.0 bpp, fair at 0.5 bpp and poor at 0.25 bpp.

The first row in the table indicates the time required to encode all information in the code-blocks, using Kakadu's default encoding parameters. These parameters have no dependence on the target bit-rate; they are selected to ensure that sufficiently information is coded to allow near lossless performance with most 8 bit/sample images. Truly lossless compression requires approximately twice as much processing time. The second row reveals the effect of the predictive truncation algorithm described above. Note the dependence of encoding times on the target bit-rate; this is a consequence of effective prediction, allowing coding passes to be skipped if their bits are likely to be truncated during code-stream formation. Prediction has no impact on the final image quality.

Recall that the prediction algorithm involves two steps: one to estimate a rate-distortion slope threshold; and another to estimate the point at which further code bits are likely to be discarded based on the slope threshold. In some cases, it is possible to determine the slope threshold ahead of time by other means. Applications requiring uniform image quality over a set of images, for example, generally do much better to fix the slope threshold for all images, rather than targeting identical bit-rates for all images. In fact, the slope threshold plays a similar role to a "quality factor" in JPEG. The third row of the table indicates the reduced encoding times which can be achieved by using prior knowledge of the slope threshold, rather than conservative estimates derived from image statistics. Kakadu's Motion JPEG2000 compressor exhibits performance virtually identical to that reported in the third row of the table, by using results obtained from previous frames in a video sequence to estimate good slope bounds for subsequent frames.

The last row in the table reveals the fact that decoding is generally significantly faster than encoding. In applications where rate-distortion slope thresholds can be predicted accurately, the encoding time need not be much more than twice the decoding time. There at least two reasons for the remaining discrepancy: 1) encoding is inherently more complex, involving more memory accesses than decoding; and 2) to avoid loss of image quality, the encoder's stopping condition must be somewhat conservative.

To put the numbers in Table 1 into perspective, we note that the full end-to-end compression process, including colour transforms, DWT, quantization, block truncation and code-stream generation requires roughly an additional $0.1\mu s$ per sample. Thus, at 1.0 bpp, even the most efficient implementation of the block encoder, with perfect slope threshold prediction, represents about $66\%$ of the total computational burden. At higher bit-rates, other costs pale into insignificance. During decompression, block decoding represents only about $50\%$ of the total computational burden, at bit-rates of around $0.5$ to $1.0$ bpp, but its cost dominates at higher bit-rates.

## 4. CODE-STREAM MANAGEMENT

Both the VM and Kakadu implementations employ a "pull" model for decompression. The application requests image lines one by one. These requests propagate through the pipelined DWT synthesis engine described in Section 2, which requests new lines from individual subband images, as required. These requests in turn, are satisfied by decoding and buffering code-blocks on demand. Kakadu extends the pull model all the way through to the code-stream parser itself. Parsing occurs on-demand as new code-block bit-streams are required by the decoder. The code-stream management machinery also unloads from memory all parsed quantities which will not be required for further processing. In interactive applications, such unloading may not be possible, since an interactive user may need to re-process previously decompressed material.

As a general rule, every compliant decompressor must be prepared to walk through the code-stream in sequential fashion to extract the information of interest to the application. However, it is possible to include optional pointer information in the code-stream which may be used to seek over unwanted data. Of particular interest to the present discussion are the optional packet length marker segments.

Apart from header information, a JPEG2000 code-stream is basically a concatenated list of "packets," each of which holds the contributions to one quality layer, from the code-blocks belonging to a single "precinct." A precinct may contain all code-blocks required to double the resolution at which the image can be reconstructed. However, a precinct may also contain smaller groupings of code-blocks. For spatial random access, it is desirable to select small precinct dimensions so that each packet represents an incremental quality contribution to a given image resolution, within a small spatial region. If the code-stream contains packet length marker segments, these may be used to derive the locations of each packet, allowing the code-stream parser to seek to the packets containing information requested by the block decoder.

Kakadu takes advantage of packet length and other pointer information embedded in the code-stream whenever it is available. Importantly, however, the Kakadu architecture will not attempt to use this information unless all packets belonging to a given precinct appear contiguously in the code-stream. JPEG2000 is intended to serve applications involving huge images, such as those produced by geo-spatial imaging systems with dimensions upwards of $64K \times 64K$ pixels. The block bit-streams belonging to any given precinct may be scattered throughout many quality layers (many packets), leading to large packet length tables and requiring numerous disjoint seeking operations to piece together the code-block bit-streams of interest. To avoid these difficulties, large images should be compressed using a packet sequencing convention which places all packets of a given precinct together. Kakadu internally condenses the packet length information for such code-streams into a single seek location for each precinct, rather than each packet.

Once the location of a precinct is known within the code-stream, the code-stream management machinery can unload that precinct's block bit-streams from memory, knowing that they can be loaded back again on demand. Using these techniques an interactive viewer can navigate quickly and efficiently within a huge image, consuming very little memory. For example, an interactive browser based on the Kakadu implementation requires less than 300 kBytes of memory to maintain an $800 \times 800$ view port into a $13K \times 13K$ colour image, compressed nearly losslessly to 90 MBytes.

It is worth noting that precincts are an excellent structure on which to base the implementation of compressed data caches, which are able to store fragments of a JPEG2000 code-stream. It is a simple matter to generalize the concept of a precinct seek address to that of a unique precinct identifier, which may be used to retrieve compressed data from a cache. The Kakadu implementation uses exactly this strategy to construct compressed data cache. In client-server applications, portions of a JPEG2000 code-stream are sent to a remotely located client, in response to a user's requests. The client caches these code-stream fragments, identified as byte ranges from each precinct's packet stream, and retrieves them on demand to render the image to a user-defined view port.

## 5. SUMMARY

In this paper, we have described some of the more important and less obvious design features of an efficient implementation of the JPEG2000 standard. The appeal of this new compression standard derives from its coupling of the DWT with embedded block coding. These allow compressed images to be accessed, processed and disseminated incrementally, in accordance with the image resolution, the image quality and the spatial region of interest to an application or end user. It is important that implementations of the standard expose these capabilities, while keeping memory and computational costs at bay.

## 6. REFERENCES


[1] I. 15444-1, "JPEG2000 image coding system," 2000.

[2] D. Taubman and M. Marcellin, *JPEG2000: Image compression fundamentals, standards and practice*. Boston: Kluwer Academic Publishers, 2001.

[3] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Applied and Computational Harmonic Analysis*, vol. 3, pp. 186–200, April 996.

[4] R. Calderbank, I. Daubechies, W. Sweldens, and B. Yeo, "Wavelet transforms that map integers to integers," *Applied and Computational Harmonic Analysis*, vol. 5, pp. 332–369, July 1998.

[5] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Proc.*, vol. 9, pp. 1158–1170, July 2000.