Cache issues with JPEG-2000 wavelet lifting

Peter Meerwald, Roland Norcen and Andreas Uhl

Dept. of Scientific Computing, University of Salzburg,
Jakob-Haringer-Str. 2, A-5020 Salzburg, Austria

ABSTRACT

In this paper, we have a close look at the runtime performance of the intra-component transform employed in the reference implementations of the JPEG2000 image coding standard. Typically, wavelet lifting is used to obtain a wavelet decomposition of the source image in a computationally efficient way. However, so far no attention has been paid to the impact of the CPU's memory cache on the overall performance. We propose two simple techniques that dramatically reduce the number of cache misses and cut column filtering runtime by a factor of 10. Theoretical estimates as well as experimental results on a number of hardware platforms show the effectivity of our approach.

Keywords: JPEG2000, wavelets, lifting, caching

1. INTRODUCTION

JPEG2000[1] is designed to supplement and enhance the existing JPEG standard[2] for still image coding. It provides advanced features such as low bit-rate compression, lossless and lossy coding, resolution and quality scalability, progressive transmission, region-of-interest (ROI) coding, error-resilience, spatial random access in a unified framework.[3,4] Of course, all these coding possibilities are not obtained for free. The runtime requirements for JPEG2000 are typically a factor of 5 - 12 higher than for JPEG and 1 - 2 higher than for SPIHT,[5] see figure 2 and compare with.[6]

The JPEG2000 image coding standard[6,7] is based on a coding scheme originally proposed by Taubman and known as EBCOT ("Embedded Block Coding with Optimized Truncation").[8] The major difference between previously proposed wavelet-based image compression algorithms such as EZW[9] or SPIHT[5] is that EBCOT as well as JPEG2000 operate on independent, non-overlapping blocks whose bit-planes are coded in several passes to create an embedded, scalable bitstream. Instead of zerotrees, the JPEG2000 scheme only exploits intra-subband correlation via context-adaptive arithmetic coding since the strictly independent block coding strategy precludes structures across subbands or even code-blocks.

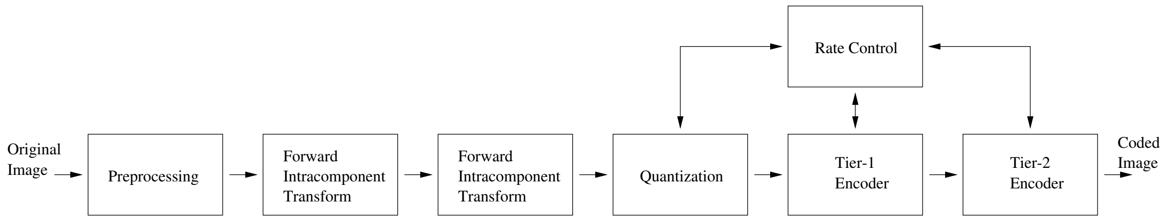


Figure 1: The basic structure of the JPEG2000 coder.

Looking at figure 1, we can identify the following coding stages: Preprocessing and the inter-component transform which are only needed for dealing with sub-sampled, multi-component (color) images. The intra-component transform is a wavelet decomposition. Tier-1 coding comprises three coding passes, context-selection and arithmetic coding. R/D allocation and bitstream formation is performed in tier-2 coding, followed by bitstream creation.

E-mail: {pmeerw,rnorcen,uhl}@cosy.sbg.ac.at

The main design goals behind JPEG2000 are versatility and flexibility which are achieved to a large extent by the independent processing and coding of image blocks. The default setting for JPEG2000 is to perform a five-level wavelet decomposition with 7/9-biorthogonal filters and then segment the transformed image into non-overlapping code-blocks of no more than 4096 coefficients which can be coded independently.

Part 5 of the JPEG2000 standard document provides two reference implementations, namely Jasper* and jj2000†. While Jasper is implemented in C, jj2000 is an object-oriented design written in Java. Surprisingly, the Java implementation does not run much slower in a just-in-time (JIT) translated Java runtime environment (JRE) compared to the compiled Jasper code – the performance difference is within 20% (see figure 2).

Runtime analysis reveals that typically 60 % of the encoder's runtime is spent in the intra-component transform (figure 3). Clearly, if we want to improve execution speed we have to target the wavelet decomposition code. JPEG2000 offers two modes of operation, lossless and lossy coding. In lossless mode, the reversible 5/3 integer transform[10] is used while the irreversible 9/7 real-to-real transform[11] gives better compression results in lossy mode. Lifting-based schemes of the wavelet transform are favored in the JPEG2000 reference implementations because they require less computational effort compared to convolution-based implementations.[12, 13]

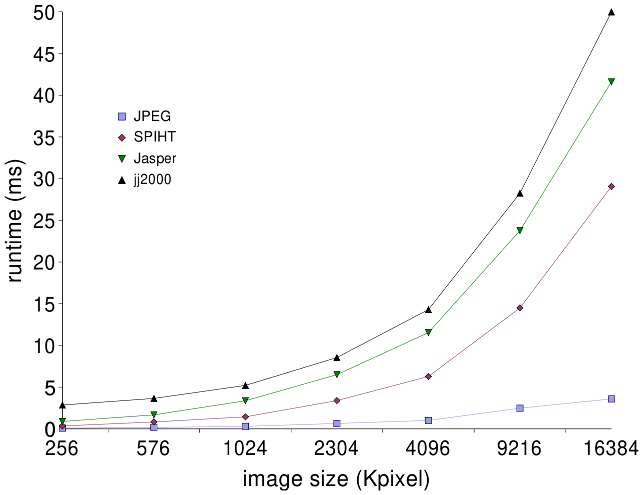


Figure 2. Comparing the runtime of JPEG, SPIHT and JPEG2000 coding gray-scale images of different sizes on a Intel Pentium II 400 Mhz.

The rest of the paper is roughly organized along the sequence of discoveries that led to the surprising runtime results. Section 2 summarizes interesting runtime values of the two reference implementations Jasper and jj2000. The overall performance for the different coding stages are given, as well as detailed results for the intra-component phase. Section 3 discusses the problem of cache inefficiency, which turns out to reduce the performance of vertical lifting significantly. Section 4 proposes 2 different solutions to enhance the access to column image data via optimizing the caching strategy. Experimental results are given in this section, too.

All results presented in this paper stem from 3 different architectures: An Intel Pentium-II Xeon 400 Mhz, an Alpha EV67 running at 667 Mhz, and a SGI MIPS R10000 system with 195 Mhz. Within the text, we use the terms Intel, Alpha, and MIPS. When doing this, we always refer to the 3 mentioned systems.

2. RUNTIME ANALYSIS

Figure 3 shows the runtime of the Jasper and jj2000 coder split up in the main processing stages. We are using gray-scale images of increasing size, lossy coding mode (and thus the 7/9 real-to-real lifting transform)

*By Michael D. Adams, available at `http://www.ece.ubc.ca/~mdadams`.

†Developed at EPFL, can be downloaded from `http://jj2000.epfl.ch`.

and perform coding with full bit-rate. Across all analyzed architectures, the wavelet decomposition in the intra-component transform stage is clearly the most demanding part.

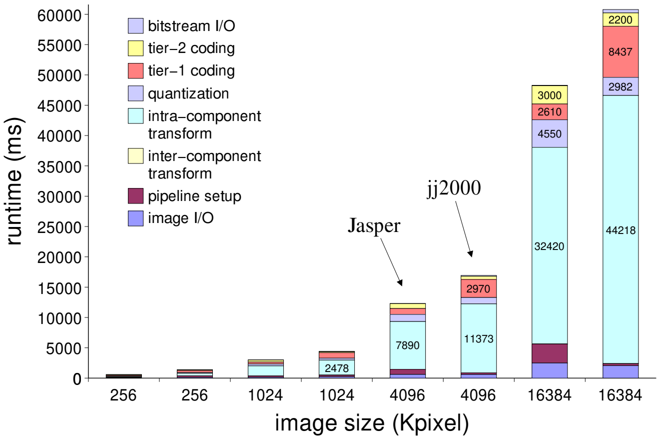


Figure 3. Runtime analysis of Jasper and jj2000; lossy coding of gray-scale images with increasing size on the Intel platform.

In order to perform a 2-D wavelet decomposition for each resolution level, first the image rows are filtered in horizontal direction followed by a decomposition in vertical direction. Filtering along both directions is done via lifting.[12] The lifting scheme is an alternative technique of computing the wavelet coefficients. Depending on the applied filter bank, lifting often requires less computations, as well as less memory when compared to the classical convolution based filtering. Anyhow, figure 4 shows the timing for the filtering procedure, broken down into the vertical and horizontal parts, respectively. The jj2000 vertical filtering step requires more than 10 times the execution time of the horizontal counterpart on the Intel CPU. The Jasper decomposition differs by a factor of roughly 10, 14, and 18 on the Intel, Alpha, and MIPS CPUs. The runtime difference is increasing with the size of the image.

JPEG2000 offers the possibility to limit the wavelet transform to tiles of a selectable size. Per default, the coder uses one tile for each component. To reduce the observed runtime difference for bigger images, the coder could select smaller tiles, which can be coded independently. The drawback, however, is that tiling typically introduces boundary effects at medium and high compression rates. Part 2 of the JPEG2000 standard document provides extensions to perform the wavelet transform with an overlap between different tiles (single sample overlap technique, SSO). This almost completely removes the artifacts and reduces memory usage and the observed difference between horizontal and vertical filtering. Nevertheless, tile sizes can be selected manually, making it necessary to analyze the vertical/horizontal filtering in more detail, especially for large image data.

3. CACHE INEFFICIENCY

The unexpected behavior suggests severe cache-miss problems (see[14–16] and also[17] for similar effects in an MPI implementation of a 3-D wavelet decomposition). In fact, it turns out that when using large images with a width equal to a power-of-two and filter length longer than 4, which corresponds to a 4-way associative cache, an entire image column is mapped onto a single cache set (see figure 5). Consequently, during the execution of the vertical wavelet filtering procedure, an enormous amount of cache misses occurs. A cache miss forces the required data to be fetched from the next higher cache level or the main memory itself. This way, we get very poor performance for the vertical filtering phase.

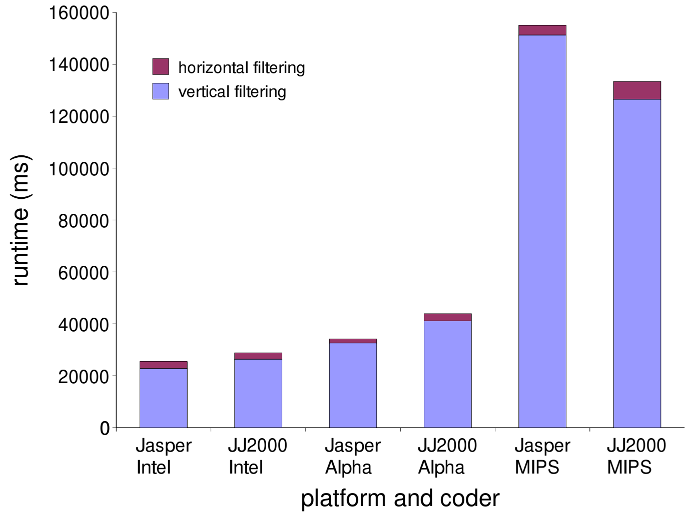
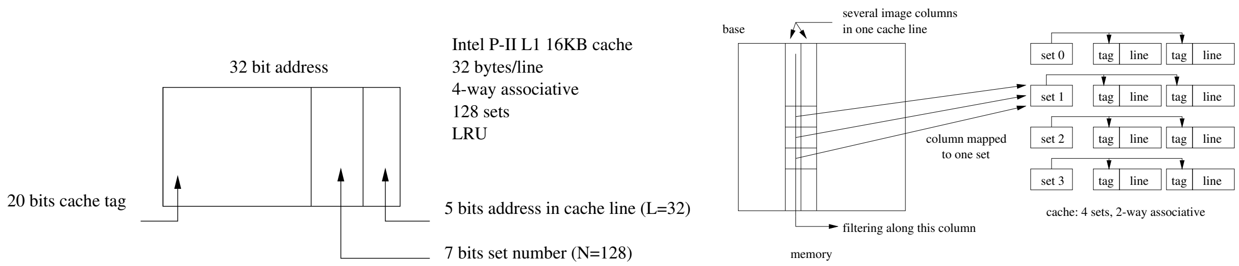


Figure 4. Zooming into the intra-component transform: horizontal versus vertical filtering for a 4096×4096 pixels image.



(a) Typical classification of a 32 bit adress for caching.

(b) Illustrative mapping of column image data onto cache sets.

Figure 5: Caching basics.

4. PROPOSED SOLUTION

In order to overcome this problem, we have considered two approaches to improve the cache hit rate. The first approach is called *'row extension'* and forces the image width to be not a power of two (eg. by allocating the next prime number of pixels greater than the image width for each row). This trivial technique does not require any modification to the filter code and results in the use of more cache sets and consequently allows cache hits when vertically adjacent pixels are processed. We can observe a significant enhancement of the vertical filtering routine on all tested platforms. The improvement is illustrated in figure 6. We cut the runtime by a factor of 3.29, 2.67, and 4.4 for the listed machines (Intel, Alpha, MIPS). The Jasper coder shows a similar behavior when incorporating row extension (1.86, 1.5, 4.35 speedup of vertical filtering through row extension on Intel, Alpha, and MIPS). Although this technique to enhance the caching efficiency is very simple, it has clearly the disadvantage, that we have to modify the original input image, inserting some dummy data thus changing the final coded bitstream.

The second approach, dubbed *'aggregation'*, involves filtering a number of adjacent columns concurrently, instead of doing the filtering sequentially column by column. As an advantage, this technique doesn't require changing the image data. When loading the pixels of an image column into the cache, the corresponding data of adjacent columns get fetched into the cache as well. This is because a single cache line spans several pixel

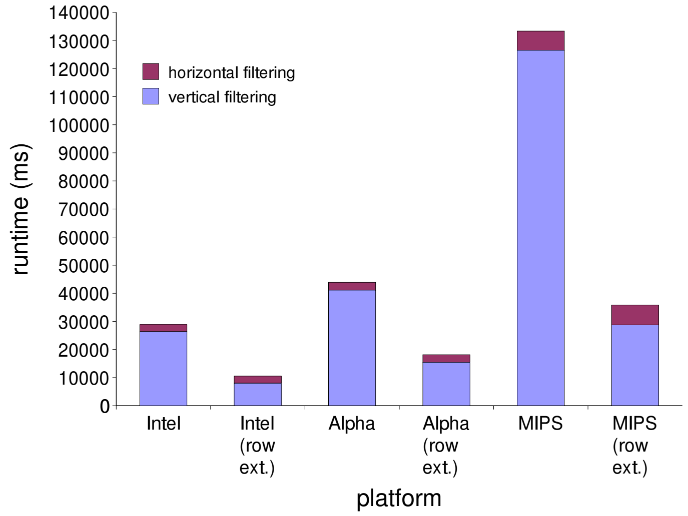


Figure 6. The effect of row extension on jj2000 vertical wavelet lifting runtime on different CPUs, image size 4096×4096.

columns. Therefore, computing the product of pixels and filter coefficients within a single cache line can be performed without any cache miss (except the initial access which triggers the 'cold' cache load). We take advantage of this caching feature and filter several columns concurrently.

To make aggregation possible, the filtering code has to be modified, the intermediate results for a certain number of different columns have to be buffered. The achievable speedup depends on the size of a cache line, i.e. 32 bytes for the Intel Pentium-II processor which corresponds to 4 pixels stored as single-precision real numbers, and the aggregation factor, which is the number of adjacent columns processed at once. Applying aggregation when filtering large images, we get surprising improvements for the vertical filtering part. Figure 7 illustrates the effect of increasing aggregation factors on the execution speed. Up to an aggregation factor of 2^5 and 2^6 (which gives the number of columns filtered concurrently), we see an improving performance for the vertical filtering part, closing the gap to the caching optimal horizontal filtering. At best, we speedup with factor 4.06, 5.82, and 8.85 (Jasper vertical lifting on Intel, Alpha, and MIPS). At this point, we are only 1.95, 5.65, and 4.67 times slower than horizontal filtering. Originally, we were 8.41, 37.07, and 41.41 times 'away' from horizontal filtering.

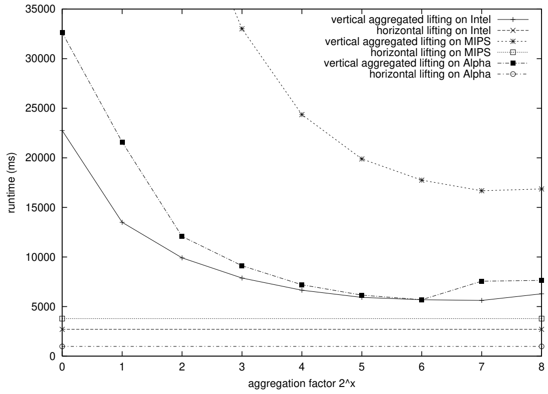
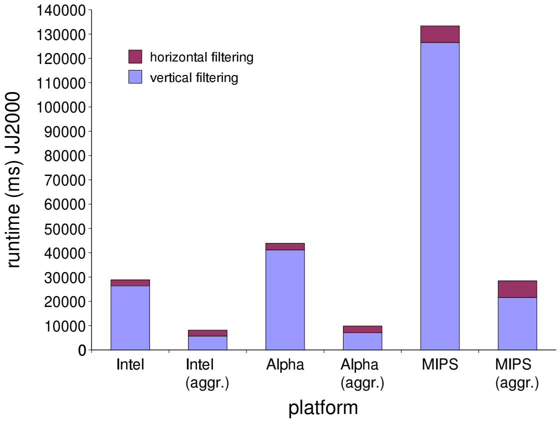
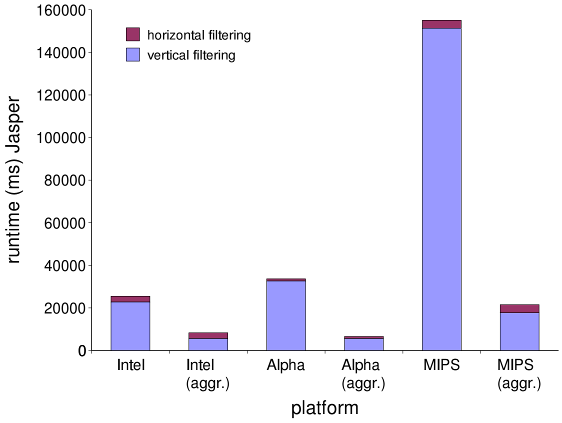


Figure 7. Vertical wavelet lifting runtime on an Intel Pentium-II 400 Mhz CPU with increasing aggregation factor; image size 4096×4096, 9/7-biorthogonal filter, C implementation.

Furthermore, we can see that for bigger aggregation factors the runtimes increase again. This should be clear,



(a) jj2000



(b) Jasper

Figure 8. The effect of aggregation on jj2000 vertical wavelet lifting runtime on different CPUs, image size 4096×4096, aggregation factor 16.

since the advantage of a better cache usage is destroyed through increased memory demands and administration overhead for the concurrent lifting. Note, that an upper and lower bound on the achievable speedup can be established: at best we achieve the performance of horizontal filtering, at worst, at a aggregation factor of 1, we achieve the performance of unmodified vertical filtering plus some extra overhead. Figure 9 correlates 3 parameters: The runtime of the vertical filtering part, the image size, as well as the aggregation factor. Along the x-axis, every line gives the runtime of the vertical filtering part for a given image size at a specific aggregation factor. Within one aggregation factor, the image size is increasing steadily from the most left line up to the most right line with 4096×4096 pixels. The peaks wee see at aggregation factor 1 prove the poor performance when filtering images equal to power of two. Applying more and more aggregation, we see these peaks diminishing, reducing the cache problem significantly.

The second approach has turned out to be more effective than row extension – see figure 8 and compare figure 8 with 6, although there is a trade-off between speedup and extra memory consumption for intermediate buffering.

The 'row extension' and 'aggregation' techniques can be combined, further reducing the runtime as shown in figure 10.

Overall, for jj2000 vertical wavelet lifting of a 4096×4096 pixels image, we have achieved a speedup factor of 6.27, 7.14 and 13.06 on the Intel, Alpha and MIPS platform, respectively. The results for the Jasper coder are similar (see figure 11 a). The numerical results for vertical and horizontal filtering employing one or both of the proposed techniques are summarized in table 1.

Considering the overall coding time for large images, we cut the runtime significantly on all architectures. On the Intel, Alpha, and MIPS, the Jasper coding time for a 4096×4096 pixels image using aggregation and row extension is reduced by a factor of 1.66, 4.31, and 3.07 (see figure 11 b). The Java implementation performs similarly.

It is also worth to note, that we don't have a big runtime gap between the Jasper C - and the Java implemantation. Especially the time demanding processing stages show similar performance. The used just-in-time compiler makes the usage of the Java implementation very competitive, different as one might suspect initially.

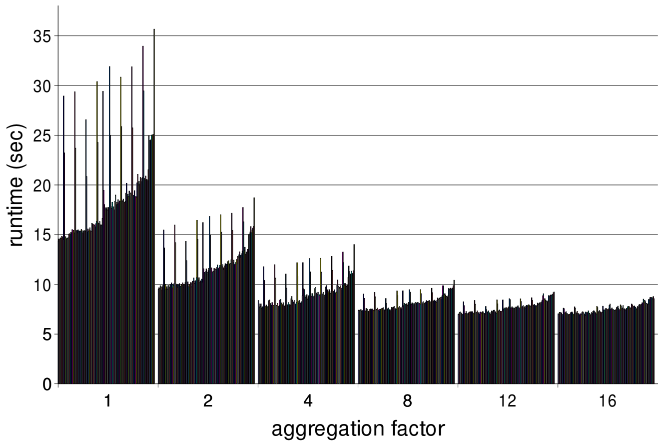
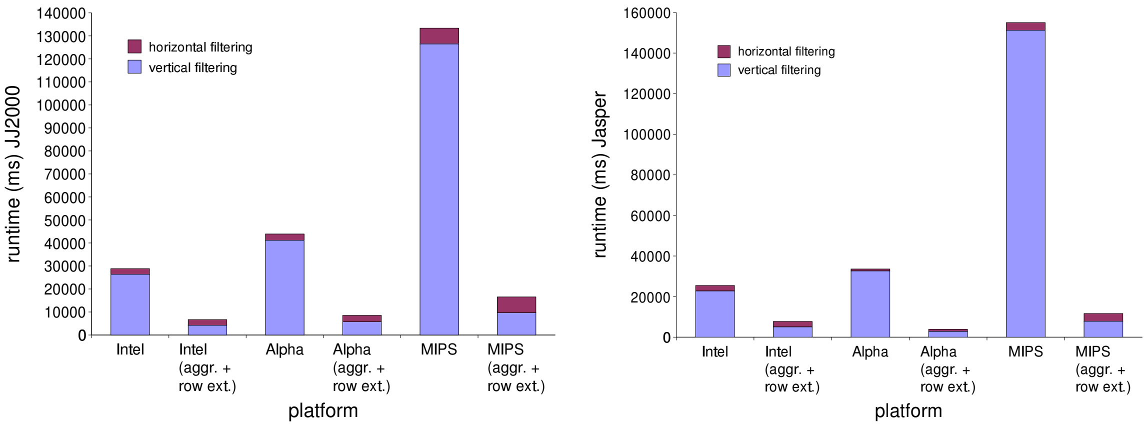


Figure 9: Increasing the performance of vertical filtering through aggregation.



(a) jj2000

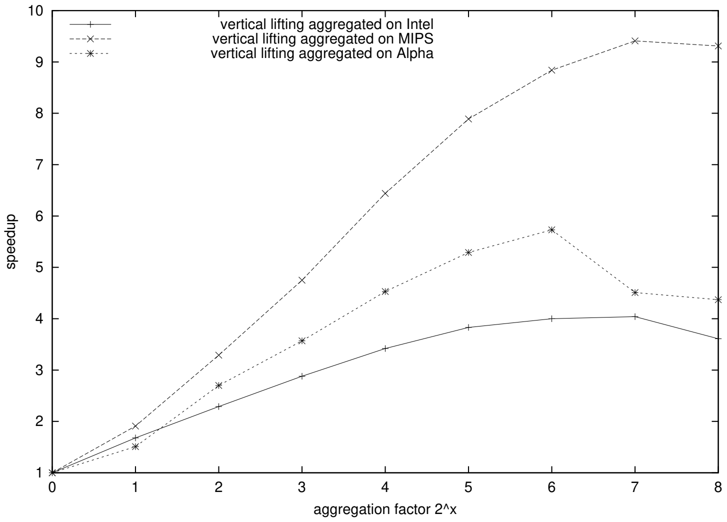
(b) Jasper

Figure 10. Combining the proposed speedup techniques (row extension + aggregation); runtime of jj2000 vertical wavelet lifting on different CPUs with image size 4096×4096, aggregation factor 16.

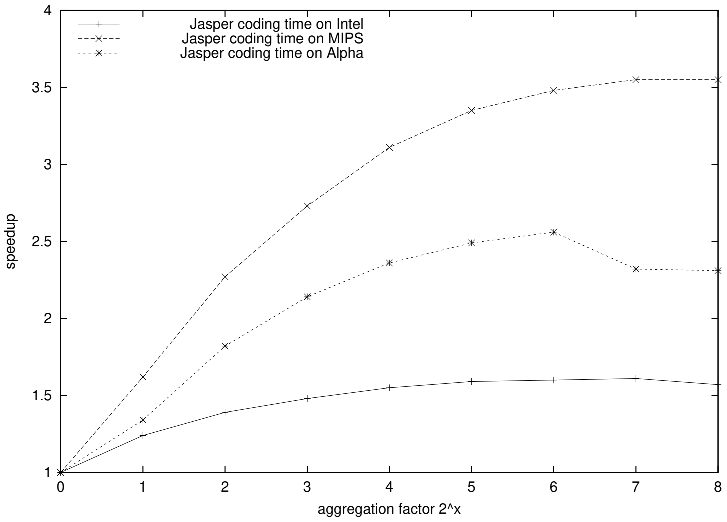
5. CONCLUSION

He have proposed two simple speedup techniques that tremendously improve the cache hit rate when performing vertical wavelet lifting as implemented in the two JPEG2000 reference implementations. Preliminary results indicate that due to the reduced memory-bus traffic, the scalability of a parallel version of the JPEG2000 coder on SMP systems is also increased. In a forthcoming paper we will combine the techniques described here with a parallelization approach to further decrease the runtime of JPEG2000 coding.

So far we have only looked at the impact of the first level (L1) cache performance. The interplay between subsequent cache hierarchy levels and multi-processor shared-memory access patterns[18] are promising areas for further runtime improvements.



(a) Speedup of vertical lifting



(b) Speedup of overall coding time

Figure 11. Performance for the modified Jasper coder on different architectures: Speedup reached with increasing aggregation factors.

ACKNOWLEDGMENTS

This work has been partially supported by the Austrian Science Fund (project FWF-13903).

REFERENCES

1. "JPEG2000 part 1 final committee draft version 1.0," tech. rep., ISO/IEC FCD15444-1, March 2000.
2. W. Pennebaker and J. Mitchell, *JPEG – Still image compression standard*, Van Nostrand Reinhold, New York, 1993.

runtime (ms)	jj2000	row ext.	aggr.	comb.	Jasper	row ext.	aggr.	comb.
Intel vert.	26348	7998	5670	4203	22760	12190	5600	5045
Intel hor.	2485	2552	2870	2868	2705	2770	2860	2790
Alpha vert.	41132	15363	7077	5759	32624	21665	5600	2854
Alpha hor.	2776	2773	2755	3078	880	983	990	984
MIPS vert.	126484	28744	21550	9683	156970	36060	17730	7860
MIPS hor.	6845	7076	6970	7125	3790	4250	3796	4240

Table 1. Numerical results on different platforms for jj2000 and Jasper; the speedup techniques are 'row extension', 'aggregation' and a combination of both.

3. M. Charrier, D. S. Cruz, and M. Larsson, "JPEG2000, the next millenium compression standard for still images," in *Proceedings of the IEEE International Conference on Multimedia & Computing Systems, ICMCS '99*, **1**, pp. 131–132, (Florence, Italy), June 1999.
4. D. Santa-Cruz and T. Ebrahimi, "A study of JPEG 2000 still image coding versus other standards," in *Proceedings of the 10th European Signal Processing Conference, EUSIPCO '00*, pp. 673–676, (Tampere, Finland), September 2000.
5. A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology* **6**, pp. 243–249, June 1996.
6. M. D. Adams, H. Man, F. Kossentini, and T. Ebrahimi, "JPEG2000: The next generation still image compression standard," tech. rep., Image Power Inc., Vancouver, BC, Canada, 2000.
7. C. Christopoulos, T. Ebrahimi, and A. N. Skodras, "JPEG2000: The new still picture compression standard," in *ACM Multimedia 2000*, pp. 45–49, (Los Angeles, CA, USA), November 2000.
8. D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Transactions on Image Processing* **9**, pp. 1158–1170, July 2000.
9. J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. on Signal Process.* **41**, pp. 3445–3462, December 1993.
10. M. D. Adams and F. Kossentini, "Reversible integer-to-integer wavelet transforms for image compression: Performance evaluation and analysis," *IEEE Transactions on Image Processing* **9**, pp. 1010–1024, June 2000.
11. M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Transactions on Image Processing* **1**(2), pp. 205–220, 1992.
12. W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM Journal of Mathemetical Analysis* **29**, pp. 511–546, March 1998.
13. A. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Lossless image compression using integer to integer wavelet transforms," in *Proceedings of the IEEE International Conference on Image Processing, ICIP '97*, **1**, pp. 596–599, (Santa Barbara, California, USA), October 1997.
14. S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," *ACM SIG-PLAN Notices* **30**(6), pp. 279–290, 1995.
15. P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Improving cache performance through tiling and data alignment," in *Workshop on Parallel Algorithms for Irregularly Structured Problems, IRREGULAR '97*, G. Bilardi, ed., *Lecture Notes in Computer Science* **1253**, pp. 167–185, Springer, 1997.
16. A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the impact of memory system performance on software prefetching and locality optimizations," in *Proceedings of the 2001 International Conference on Supercomputing, ICS '01*, (Sorrento, Italy), June 2001.
17. R. Kutil and A. Uhl, "Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers," in *Parallel Computation. Proceedings of ACPC'99*, P. Zinterhof, M. Vajtersic, and A. Uhl, eds., *Lecture Notes on Computer Science* **1557**, pp. 347–356, Springer-Verlag, 1999.
18. A. Milenkovic, "Achieving high performance in bus-based shared-memory multiprocessors," *IEEE Concurrency* **8**, pp. 36–44, July 2000.