Improving the Memory Behavior of Vertical Filtering in the Discrete Wavelet Transform

Asadollah Shahbahrami shahbahrami@ce.et.tudelft.nl
Ben Juurlink benj@ce.et.tudelft.nl
Stamatis Vassiliadis stamatis@ce.et.tudelft.nl

Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, The Netherlands
Phone: +31 15 2787362, Fax: +31 15 2784898.

ABSTRACT

The discrete wavelet transform (DWT) is used in several image and video compression standards, in particular JPEG2000. A 2D DWT consists of horizontal filtering along the rows followed by vertical filtering along the columns. It is well-known that a straightforward implementation of vertical filtering (assuming a row-major layout) induces many cache misses, due to lack of spatial locality. This can be avoided by interchanging the loops. This paper shows, however, that the resulting implementation suffers significantly from 64K aliasing, which occurs in the Pentium 4 when two data blocks are accessed that are a multiple of 64K apart, and we propose two techniques to avoid it. In addition, if the filter length is longer than four, the number of ways of the L1 data cache of the Pentium 4 is insufficient to avoid cache conflict misses. Consequently, we propose two methods for reducing conflict misses. Although experimental results have been collected on the Pentium 4, the techniques are general and can be applied to other processors with different cache organizations as well. The proposed techniques improve the performance of vertical filtering compared to already optimized baseline implementations by a factor of 3.11 for the $(5,3)$ lifting scheme, 3.11 for Daubechies' transform of four coefficients, and by a factor of 1.99 for the Cohen, Daubechies, and Feauveau 9/7 transform.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Measurement Techniques.

General Terms: Algorithms, Performance.

Keywords: Discrete Wavelet Transform, memory hierarchy, cache, performance.

1. INTRODUCTION

The wavelet transform is mainly used for image and video compression. Standards such as MPEG-4 and JPEG2000 [13, 15] are based on the 2D discrete wavelet transform (DWT).

CF'06, May 3–5, 2006, Ischia, Italy.

The JPEG2000 compression standard has been created to provide higher compression ratios than JPEG. It can be very time-consuming, however. For example, simulation results presented in [18] show that JPEG2000 encoding can take up to 34 times longer than JPEG encoding. Furthermore, results presented in [3, 14] show that the DWT consumes 40 to 60% of the JPEG2000 encoding time.

One way to reduce the execution time of the DWT is by developing special-purpose hardware. Programmable processors, however, are preferable because they are more flexible and enable different transforms, various filter bank lengths, and various transform levels. In this paper we, therefore, focus on the implementation of the 2D DWT on general-purpose processors, in particular the Pentium 4. Three different filters are considered in this paper, namely the $(5,3)$ lifting scheme [9, 21], Daubechies' transform with four coefficients [22] (Daub-4), and the Cohen, Daubechies and Feauveau 9/7 transform [8] (CDF-9/7). The reasons for considering these filters are (1) the lifting and CDF-9/7 transforms are included in Part 1 of the JPEG2000 standard [15], and (2) these transforms have been considered in many recent papers (e.g., [1, 10, 16, 5, 22, 6]).

A 2D DWT consists of horizontal filtering along the rows followed by vertical filtering along the columns. It is well-known that a straightforward implementation of vertical filtering (assuming a row-major layout) generates many cache misses, due to lack of spatial locality. This can be avoided by interchanging the loops. Loop interchange, however, does not solve all cache and memory problems. This is illustrated in Figure 1, which depicts the speedup of horizontal filtering over vertical filtering (with interchanged loops) on the Pentium 4, which is equipped with an 8KB 4-way set-associative L1 data cache with a line size of 64 bytes.

Figure 1 shows that for some image sizes vertical filtering is significantly slower than horizontal filtering while for other image sizes there is hardly any difference. This behavior should *not* be attributed to cache misses, however, since the associativity of the L1 data cache is sufficient to eliminate most conflict misses for the lifting and Daub-4 transforms. This is illustrated in Figure 2, which shows the ratio of the number of cache misses incurred by vertical filtering to the number of cache misses incurred by horizontal filtering. These results have been obtained using a trace-driven cache simulator.

It can be seen that for small images, vertical filtering with the lifting and Daub-4 transform does not produce more cache misses than horizontal filtering. For larger images,

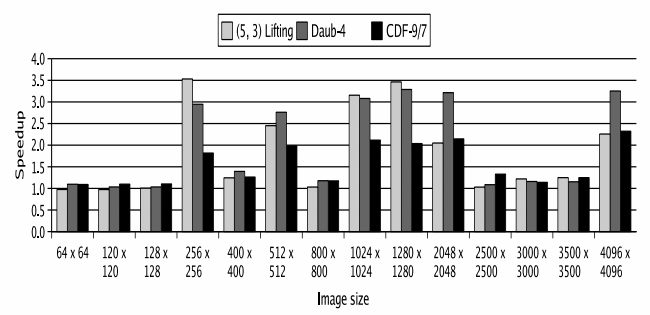

Figure 1: Speedup of horizontal filtering over vertical filtering on the Pentium 4 for various image sizes and filters

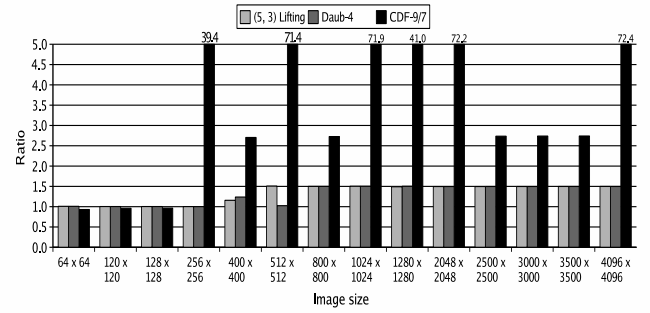

Figure 2: Ratio of the number of cache misses incurred by vertical filtering (with loops interchanged) to the number of cache misses incurred by horizontal filtering for an 8KB 4-way set-associative L1 data cache with a line size of 64 bytes.

vertical filtering generates approximately 50% more cache misses than horizontal filtering. This trend is not seen in Figure 1, however. For example, when the image dimension is 2048, vertical filtering using the lifting transform is slower by a factor of approximately 2.1 than horizontal filtering while when the image dimension is 2500 the slowdown factor is about 1.03. In both cases, however, vertical filtering generates approximately 50% more cache misses than horizontal filtering. On the other hand, if CDF-9/7 is employed the cache performance of vertical filtering is much worse than the cache performance of horizontal filtering, in particular when the image size is a power of two or a multiple of a large power of two. But also for this transform the cache behavior does not match the runtime behavior on the Pentium 4. Consequently, poor cache performance does *not* fully explain why vertical filtering is much slower than horizontal filtering for some image sizes on the Pentium 4. Instead, this behavior is due to *64K aliasing* [11], which occurs in the Pentium 4 when two data blocks whose addresses differ by a multiple of 64K need to be cached simultaneously.

The objectives addressed in this paper are to propose and evaluate techniques to avoid 64 aliasing and to improve the cache performance for transforms that suffer from many cache conflict misses. Our main contributions can be summarized as follows:

- We propose and evaluate two techniques to avoid 64K aliasing. The first technique provides a speedup of up to 3.3, but for image sizes that do not suffer from 64K aliasing this technique reduces performance by up to 20%. The second technique improves performance by up to a factor of 3.1 and incurs no performance penalty for image sizes that do not suffer from 64K aliasing.

- For those transforms (such as the CDF-9/7 on a 4-way set-associative cache) that experience many cache conflict misses, we propose and evaluate two techniques to improve cache performance. The first technique improves performance by up to 80% and the second technique by up to 99%. For image sizes that do not generate many cache conflict misses both techniques decrease performance slightly, due to the overhead (loop overhead, address calculations) introduced by applying these techniques.

This paper is organized as follows. Related work is discussed in Section 2. In Section 3 we have collected various background information. It describes the wavelet transform in detail, the experimental environment, and the evaluation methodology. Section 4 proposes and evaluates two techniques to circumvent 64K aliasing. Section 5 addresses the cache behavior of transforms such as CDF-9/7 and proposes and evaluates techniques to avoid conflict misses. Finally, conclusions are drawn in Section 6.

2. RELATED WORK

Meerwald et al. [14] also observed that caching inefficiency reduces the performance of vertical lifting significantly and proposed two techniques, called row extension and aggregation, to overcome this problem. Row extension adds some dummy elements to each row so that the image width is no longer a power of two but co-prime with the number of cache sets. According to [14], a disadvantage of this method is that the final coded bitstream is changed. Aggregation filters a number of adjacent columns consecutively before moving to the next row, instead of performing vertical filtering column by column. If the number of columns filtered consecutively is equal to the image width, aggregation is identical to loop interchange. However, if the length of the filters is larger than the number of cache ways, aggregation does not eliminate all conflict misses. In other words, it does not remove the conflicts that may exist between the input coefficients needed to compute one output coefficient. This occurs, for example, in the CDF-9/7 transform (filter length 9) for a 4-way set-associative cache. Meerwald et al. employed the CDF-9/7 transform.

Chatterjee and Brooks [3] proposed to employ an auxiliary matrix to store the results of horizontal filtering. We do the same, because this auxiliary matrix avoids an expensive rearrangement step. In addition, they proposed two optimizations: strip-mining and recursive data layout. Strip-mining is identical to aggregation. The second optimization modifies the layout of the image data so that each sub-band is stored contiguously. This increases the locality for subsequent decomposition levels, but only the execution time of the first decomposition level is reported. Both strip-mining and recursive data layout do not remove the conflicts that may exist between the input coefficients needed to compute one output coefficient.

Chaver et al. [4, 5] also considered the memory hierarchy issue but also vectorized the 2D DWT using an SIMD extension. They proposed combining aggregation with a line-based approach [7], which starts vertical filtering as soon as a

sufficient number of lines (determined by the filter lengths) has been filtered horizontally. This approach reduces the amount of memory required. In addition, they considered different layouts. Although they considered images with a width equal to a power of two and measured performance on a Pentium 4 (as well as a Pentium III), they did not mention 64K aliasing. Moreover, their approach also does not eliminate all cache conflict misses.

There are three main differences between our work and the works mentioned above. First, the techniques we propose have not been applied before to optimize vertical filtering. Second, previous work did not mention nor address 64K aliasing. Third, our techniques eliminate the conflicts that may exist between the input coefficients needed to compute one output coefficient if the filter length is larger than the number of cache ways. Previous work has focused mainly on improving spatial locality.

In [17] we considered various ways to implement the 2D DWT using MMX instructions. In this work we observed that the performance improvement provided by MMX varies significantly depending on the image size. Specifically, if 64K aliasing occurs, the speedup is much higher than when it does not occur. This observation is the main motivation for the current work.

3. BACKGROUND

In this section we describe the discrete wavelet transform in detail, the experimental setup, and the reference implementation with which we will compare our results.

3.1 Discrete Wavelet Transform

The wavelet representation of a discrete signal X consisting of N samples can be computed by convolving X with the low-pass and high-pass filters and down-sampling the output signal by 2, so that the two frequency bands each contain $N/2$ samples. With the correct choice of filters, this operation is reversible. This process decomposes the original image into two sub-bands: the lower and the higher band [20]. This transform can be extended to multiple dimensions by using separable filters. A 2D DWT can be performed by first performing a 1D DWT on each row (*horizontal filtering*) of the image followed by a 1D DWT on each column (*vertical filtering*).

Figure 3 illustrates the first decomposition level ($d = 1$). In this level the original image is decomposed into four sub-bands that carry the frequency information in both the horizontal and vertical directions. In order to form multiple decomposition levels, the algorithm is applied recursively on the LL sub-band. Figure 4 illustrates the second ($d = 2$) and third ($d = 3$) decomposition levels as well as the layout of the different bands. As mentioned before, three transforms have been employed: the integer-to-integer $(5, 3)$ lifting scheme, Daubechies' real-to-real transform with four coefficients, and the Cohen, Daubechies and Feauveau 9/7 transform. The $(5, 3)$ lifting scheme requires four memory accesses (three loads and one store) to compute each coefficient, Daub-4 requires five, and CDF-9/7 requires 10.

3.2 Experimental Setup

All programs have been implemented in C and were compiled using gcc with optimization level *-O2*. As experimental platform we have employed a 3.0GHz Pentium 4 processor. The main architectural parameters of our system are summarized in Table 1.

Processor	Intel Pentium 4
CPU Clock Speed	3.0GHz
L1 Data Cache	8 KBytes, 4-way set associative, 64 Bytes line size
L2 Cache	512 KBytes, 8-way set associative, 64 Bytes line size, On Chip

Table 1: Parameters of the experimental platform.

All programs were executed on a lightly loaded system. Performance was measured using the IA-32 cycle counter [12]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program [2, 19]. In order to eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache have been used [2]. That means the function is repeatedly (K times) executed and the fastest time is recorded. Executing the function at least once before starting the measurement minimizes the effects of both instruction and data cache misses.

3.3 Reference Implementation

It is possible to compute the DWT in place. However, in order to do so, the wavelet coefficients have to be rearranged in the order expected by the quantization step. To avoid this rearrangement step, we have employed an auxiliary matrix that stores the results of horizontal filtering, as proposed in [3].

The straightforward way of performing vertical filtering is by processing each column entirely before advancing to the next column. This method, however, results in excessive cache misses because it is unable to exploit spatial locality, since the cache blocks corresponding to the first rows will have been replaced when the algorithm advances to the next column. In order to improve spatial locality we have applied *loop interchange*, which is a well-known compiler technique. Figure 5 depicts the effectiveness of loop interchange for vertical filtering. It depicts the speedup of vertical filtering with interchanged loops over the straightforward implementation which processes each column entirely before advancing to the next column for the $(5, 3)$ lifting and Daub-4 transforms. Clearly, the implementations with interchanged loops are much more efficient than the straightforward implementations, especially when the image is large. For this reason we will compare the performance of our methods to the performance attained by the algorithms after loop interchange. In other words, the implementations with interchanged loops will be used as reference implementations.

We finally remark that the reason why the speedup for $N = 2500, 3000, 3500$ is much larger than for $N = 2048, 4096$ (for example) is that for $N = 2500, 3000, 3500$, 64K aliasing does not occur while for $N = 2048, 4096$ it does. Consequently, although loop interchange improves spatial locality, for those image sizes that suffer from 64K aliasing the performance improvement is smaller than for those image sizes that do not suffer from 64K aliasing.

4. AVOIDING 64K ALIASING

In the Pentium 4 there is a phenomenon known as *64K aliasing*. It occurs if two or more data blocks whose addresses differ by a multiple of 64K need to be cached simultaneously. If this occurs, the associativity of the cache is useless and the effectiveness of the cache is greatly reduced. For some image sizes, the 2D DWT suffers from 64K

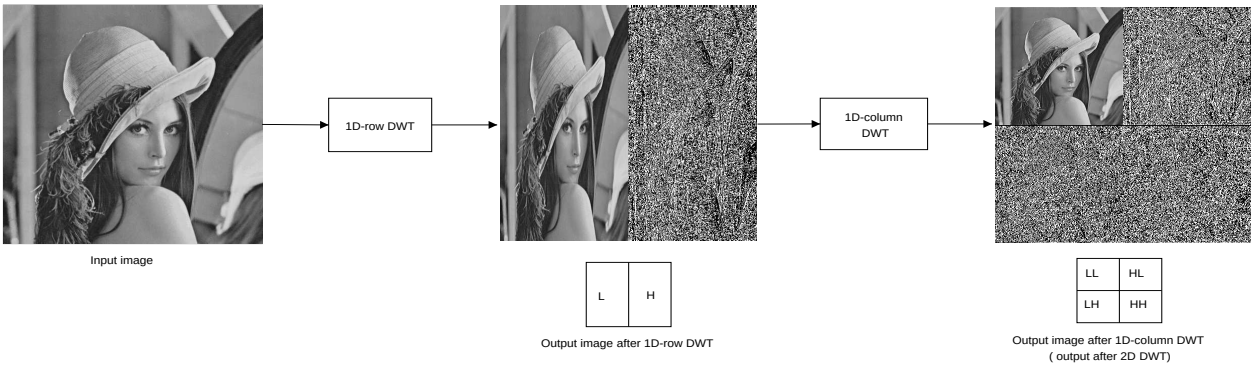

Figure 3: Different sub-bands after first decomposition level.

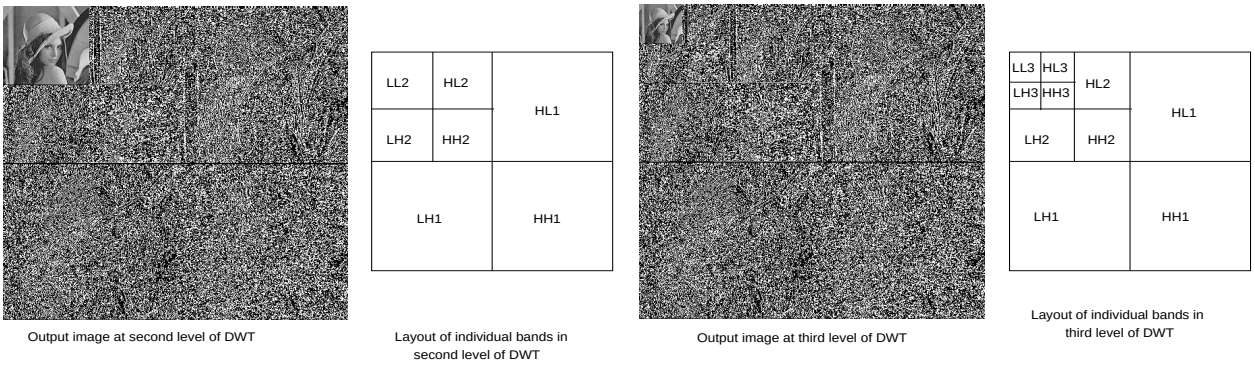

Figure 4: Sub-bands after second and third decomposition level.

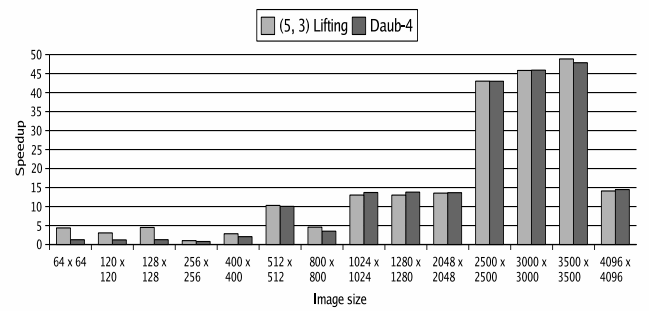

Figure 5: Effectiveness of loop interchange on the Pentium 4. This figure depicts the speedup of vertical filtering with interchanged loops over the straightforward implementation which processes each column entirely before advancing to the next column for the lifting and Daub-4 transforms.

aliasing. For example, Figure 6 depicts the C implementation of vertical filtering using the Daub-4 transform for an $N \times M$ image. It can be seen that one iteration of the inner loop accesses `input_img[i][j]` and `input_img[i+N/2][j]`. Consequently, since the matrices are stored in row-major order, 64K aliasing occurs if $cNM/2$ is a multiple of 64K bytes, where c is the number of bytes needed to represent one wavelet coefficient. In this section we propose and evaluate two techniques to circumvent 64K aliasing.

The first idea we explore is *loop splitting*. By calculating the low-pass (`input_img[i][j]`) and high-pass values (`input_img[i+N/2][j]`) in separate loops, the 64K alias between them is removed.

```
void Daub_4_vertical() {
int i, j, jj;
float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
float high[]={-0.4830, 0.8365, -0.2241, -0.1294};
for (i=0, ii=0; ii<N; i++, ii +=2)
  for(j=0; j<M; j++) {
    input_img[i][j] = output_img[ii][j]  *low[0]
                    + output_img[ii+1][j]*low[1]
                    + output_img[ii+2][j]*low[2]
                    + output_img[ii+3][j]*low[3];

    input_img[i+N/2][j]=output_img[ii][j]*high[0]
                    + output_img[ii+1][j]*high[1]
                    + output_img[ii+2][j]*high[2]
                    + output_img[ii+3][j]*high[3];
      }
  }
```

Figure 6: C implementation of vertical filtering using the Daub-4 transform. Note that the loops have been interchanged w.r.t. the straightforward implementation.

Figure 7 depicts the speedup resulting from this program transformation. For those image sizes that suffer from 64K aliasing (powers of two larger than 256×256 and 1280×1280), loop splitting indeed improves performance significantly. In these cases the speedup ranges from 1.97 to 2.94 for the lifting transform, from 2.36 to 3.31 for Daub-4, and from 1.27 to 1.75 for CDF-9/7. For CDF-9/7, the performance improvements are smaller than for the other two transforms, because it also suffers from many cache conflict misses while the other two transforms do not. Furthermore, for those image sizes that do not suffer from 64K alias-

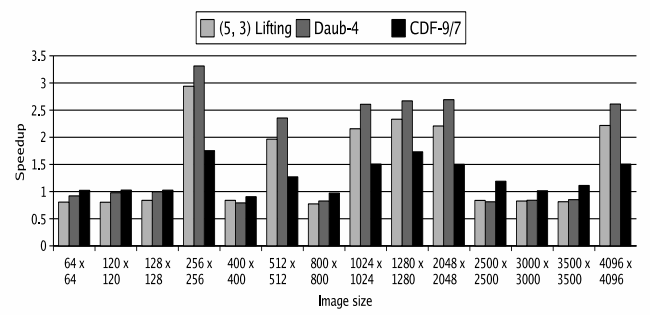

Figure 7: Speedup resulting from loop splitting.

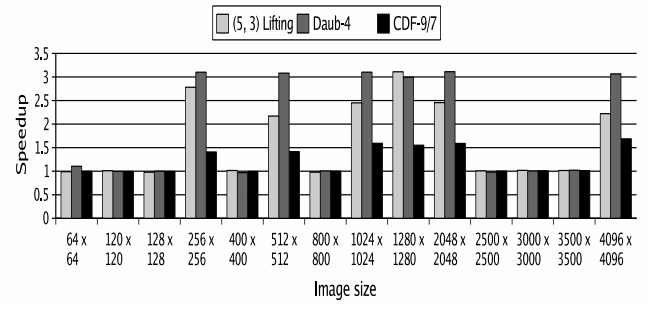

Figure 8: Performance improvement achieved by the offsetting technique.

ing, loop splitting reduces performance by up to 20%. This is because this transformation removes the temporal reuse that exists between the calculation of the high-pass and low-pass values. For example, as Figure 6 shows the coefficient `output_img[ii][j]` is needed to compute `input_img[i][j]` as well as `input_img[i+N/2][j]`. When the loop is split, this temporal reuse is no longer exploited. In addition, loop splitting slightly increases loop overhead.

The second technique we propose is to offset the memory address of the high-pass value by one row (or, equivalently, by cM bytes, where c is the number of bytes per wavelet coefficient and M is the number of columns). In other words, instead of storing the high-pass value in `input_img[i+N/2][j]`, it is stored in `input_img[i+N/2+1][j]`. By applying this offsetting technique, the distance between the two addresses is no longer a multiple of 64K, but in order to apply this method, the matrices have to be extended with one row.

Figure 8 depicts the speedup achieved by the offsetting technique over the reference implementation. For those image sizes ($N = 256, 512, 1024, 1280, 2048, 4096$) that suffer from 64K aliasing, offsetting improves performance by a factor ranging from 2.17x to 3.11x for the lifting transform, from 2.99x to 3.11x for Daub-4, and from 1.41x to 1.69x for CDF-9/7. Furthermore, offsetting technique does *not* incur a performance penalty for image sizes that do not suffer from 64K aliasing ($N = 64, 120, 128, 400, 800, 2500, 3000, 3500$). This is because this technique does not destroy the temporal locality between the calculation of the low and high-pass values. Concluding, the offsetting technique is better than loop splitting.

Figure 8 also shows that the speedups are higher for the lifting and Daub-4 transforms than for the CDF-9/7 transform, as was the case for the loop splitting technique. Again this is due to the fact that vertical filtering using the CDF-9/7 transform not only suffers from 64K aliasing but also from many cache conflict misses. For the other two transforms, on the other hand, the associativity of the L1 data cache is sufficient to eliminate most conflict misses. The following section discusses this problem in detail and presents two solutions.

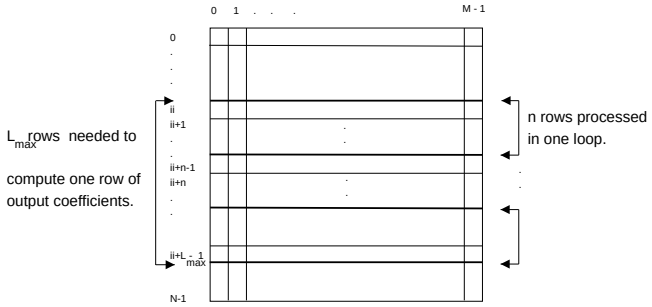

Figure 9: Associativity-conscious loop splitting.

5. AVOIDING CACHE CONFLICTS

The Pentium 4 microprocessor is equipped with a relatively small (8KB) 4-way set-associative cache with a line size of 64B. As shown in the introduction (Figure 2) and as can be seen analytically, the associativity is sufficient to remove most cache conflict misses incurred by vertical filtering for the $(5, 3)$ lifting and Daub-4 transforms, since in the lifting transform three input coefficients are needed to compute one output coefficient and in Daub-4 four input coefficients are required to compute one output coefficient. When using the CDF-9/7 as well as other transforms, however, the number of input coefficients (9) needed to compute one output coefficient exceeds the number of ways. This leads to excessive cache misses in vertical filtering if the rows needed to compute one row of wavelet coefficients map to the same cache set(s). In this section we present and evaluate two techniques to avoid such cache conflicts. Although experimental results are presented for the CDF-9/7 transform and the Pentium 4, both techniques are general and architecture independent. By this we mean that they can also be applied to other transforms and to other cache organizations, since they take the associativity of the cache and the lengths of the low- and high-pass filters into account.

The first method is referred to as *associativity-conscious loop splitting* (ACLS). The idea is to split the loop that computes one row of wavelet coefficients into multiple loops so that each loop accesses at most n rows, where n is the number of ways. The first loop computes the partial results that can be computed by accessing the first n rows of input coefficients. The remaining loops add their results to these partial results. Specifically, let $L_{\max} = \max\{L_{\text{low}}, L_{\text{high}}\}$, where L_{low} and L_{high} are the lengths of the low- and high-pass filters of the DWT. We calculate one row of wavelet coefficients using $L_{\max}/n$ loops, and each loop accesses n rows of input coefficients. This transformation is illustrated in Figure 9 and pseudo-code that illustrates the transformation is depicted in Figure 10. For simplicity, we have assumed that n divides $L_{\max}$.

In the second scheme, which is called *lookahead*, the rows of input coefficients are processed in a skewed manner. There is only one loop for vertical filtering, as in the original algorithm. In each loop iteration we process n rows of in-

```
for (i=0, ii=0; ii<N; i++, ii+=2)
   for(j=0; j<M; j++) {
      input_img[i][j] = output_img[ii][j]*low[0] + output_img[ii+1][j]*low[1]
                      + . . .  + output_img[ii+L_max-1][j]*low[L_max-1];
      . . .
   }
```

(a)

```
for (i=0, ii=0; ii<N; i++, ii+=2) {
   for(j=0; j<M; j++)
      input_img[i][j] = 0.0;
   for(L=0; L<L_max ; L+=n)
      for(j=0; j<M; j++) {
          input_img[i][j] += output_img[ii+L][j]*low[L]
                          +  output_img[ii+L+1][j]*low[L+1]
                          +  . . . + output_img[ii+L+n-1]*low[L+n-1];
    . . .
       }
 }
```

(b)

Figure 10: (a) reference implementation and (b) associativity-conscious loop splitting technique.

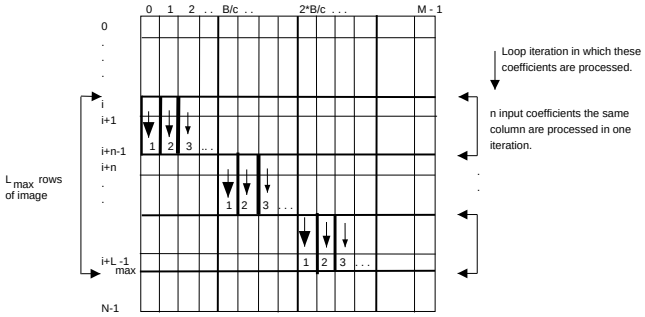

Figure 11: Illustration of the lookahead algorithm for vertical filtering.

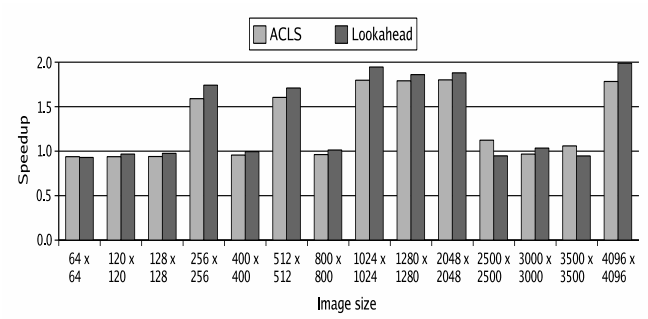

Figure 13: Speedup obtained by applying associativity-conscious loop splitting and the lookahead technique as well as offsetting technique over the reference implementation in the CDF-9/7 transform.

put coefficients for one particular output coefficient (say `input_img[i][j]`) but, in the same iteration, we process the next n rows for the output coefficient that is located B/c columns ahead (`input_img[i][j+B/c]`), where B is the cache line size in bytes and `c` is the number of bytes per coefficient, and so on. So $L_{\max}/n$ partial results are computed in one loop iteration. In later iterations, partial results that correspond to the same column are added together. This scheme ensures that no more than n input coefficients accessed in one loop iteration map to the same cache set. This algorithm is illustrated in Figure 11 and pseudo-code that illustrates the transformation is given in Figure 12. For brevity and simplicity, start-up and clean-up code has been omitted.

Figure 13 depicts and compares the performance obtained by the associativity-conscious loop splitting and lookahead techniques. It depicts the speedup obtained by applying both techniques compared to the reference implementation. To avoid 64K aliasing, the offsetting technique has been applied. For image sizes that suffer from excessive conflict misses ($2^m \times 2^m$ where $m \geq 8$ and 1280×1280), both techniques indeed improve performance significantly. For these image sizes the performance improvement provided by ACLS ranges from 59% to 80% and the improvement achieved by the lookahead technique ranges from 71% to 99%. In general, except for two image sizes ($N = 2500$ and $N = 3500$), the lookahead technique performs slightly better than ACLS. This is because it incurs less loop overhead than ACLS. For image sizes that do not generate many conflict misses, both schemes generally slightly decrease performance. This is due to overhead needed for managing loop and index variables and address calculations.

As remarked before, both ACLS and the lookahead technique are general and architecture independent. By this we mean that, although results have been measured for the CDF-9/7 transform and on the Pentium 4, they can also be applied to other transforms and processors with different cache configurations. For example, for certain image sizes, the $(5, 3)$ lifting and Daub-4 transforms would incur many cache conflict misses for a 2-way set-associative cache. But in these cases the same techniques can be applied with the parameters $L_{\max} = 4$ and $n = 2$.

To validate this claim, Figure 14 depicts the speedup obtained by applying ACLS on the Intel Pentium III (Katmai) and AMD Opteron processors. The Pentium 3 is equipped with a 16KB L1 data cache with a line size of 32 bytes, and the Opteron with a 64KB L1 data cache with a line size of 64 bytes. Both caches are 2-way set-associative. We

```
     for (i=0, ii=0; ii<N; i++, ii+=2)
        for (j=0; j<M; j++) {
           input_img[i][j] = output_img[ii][j]*low[0] + output_img[ii+1][j]*low[1]
                           + . . .  + output_img[ii+L_max-1][j]*low[L_max-1];
              . . .
        }
```

(a)


```
for (i=0, ii=0; ii<N; i++, ii+=2)
  for (j=0; j<M - L_max/n*B/c; j++) {
    input_img[i][j] += output_img[ii][j]*low[0]
                    +  output_img[ii+1][j]*low[1]
                    + . . . + output_img[ii+n-1]*low[n-1];

    input_img[i][j+B/c] += output_img[ii+n][j+B/c]   * low[n]
                        +  output_img[ii+n+1][j+B/c]* low[n+1]
                        + . . . + output_img[ii+2*n-1][j+B/c]*low[2*n-1];
    . . .
    input_img[i][j+L_max/n*B/c] += output_img[ii+L_max-1-n][j+L_max/n*B/c]*low[L_max-1-n]
                                +  output_img[ii+L_max-1-n+1][j+L_max/n*B/c]*low[L_max-1-n+1]
                                + . . . + output_img[ii+L_max-1][j+L_max/n*B/c]*low[L_max-1];
  }
```

(b)

Figure 12: (a) reference implementation and (b) lookahead technique.

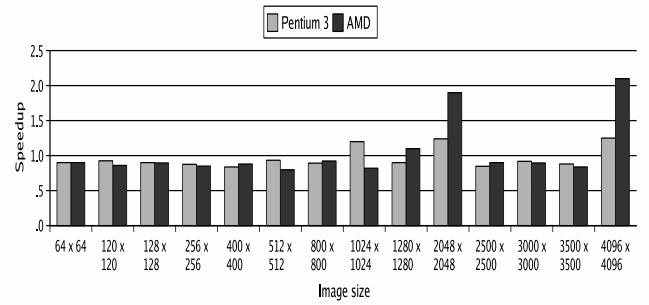

Figure 14: Speedup obtained by applying associativity-conscious loop splitting technique over the reference implementation in the CDF-9/7 transform on the Pentium 3 and AMD processors.

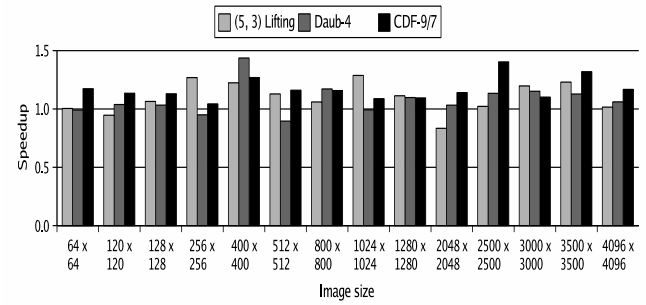

Figure 15: Speedup of horizontal filtering over vertical filtering after avoiding both 64K aliasing and cache conflict misses problems.

have determined experimentally that the Pentium III suffers from many conflict misses for $N = 1024, 2048, 4096$ and the Opteron for $N = 2048, 4096$ and, to a lesser extent, for $N = 1280$. Figure 14 shows that for these image sizes ACLS provides a performance improvement ranging from 20% to 25% on the Pentium III and from 10% to 110% on the Opteron. For all other image sizes, however, it reduces performance by up to 20%, due to the overhead introduced. This shows that it is necessary to provide different versions of the code and, depending on the image size and the cache configuration of the target platform, to branch to the most efficient version.

Finally, Figure 15 depicts the speedup of horizontal filtering over vertical filtering after the offsetting technique has been applied to avoid 64K aliasing (for all three transforms) and the lookahead technique has been applied to reduce cache conflicts (for the CDF-9/7 transform). In most cases vertical filtering is at most 20% slower than horizontal filtering, which indicates that there is not much room for further improvements. In a few cases, after applying the proposed techniques, vertical filtering is even faster than horizontal filtering. Comparing Figure 15 to Figure 1 shows that the proposed techniques improve the performance of vertical filtering tremendously compared to the often reported implementation with interchanged loops.

6. CONCLUSIONS

In this paper several techniques have been proposed to improve the memory behavior of the vertical filtering phase of the 2D DWT on the widespread Pentium 4 processor. It has been shown that although loop interchange improves performance significantly, the resulting implementation still suffers from 64K aliasing for certain image sizes. Furthermore, depending on the transform employed, the associativity of the cache, and the image size, vertical filtering can generate many cache conflict misses, even when the loops are interchanged. Previous work did not address 64K aliasing and mainly focused on improving spatial locality, while our caching techniques eliminate the conflicts between the input coefficients needed to compute one output coefficient.

To avoid 64K aliasing two techniques have been applied: loop splitting and offsetting. For image sizes that suffer from

64K aliasing, loop splitting provides a speedup that ranges from 1.97 to 2.94 for the lifting transform, from 2.36 to 3.31 for Daub-4, and from 1.27 to 1.75 for CDF-9/7. Loop splitting, however, destroys the temporal locality between the calculation of the low- and high-pass values. Consequently, for those image sizes that do not suffer from 64K aliasing it reduces performance by up to 20%. For image sizes that suffer from 64K aliasing, offsetting achieves speedups between 2.17 and 3.11 for the lifting transform, between 2.99 and 3.11 for Daub-4, and between 1.41 and 1.69 for CDF-9/7. Furthermore, because it does not destroy the temporal reuse, it does not incur a performance penalty for image sizes that do not suffer from 64K aliasing. We conclude that offsetting is better than loop splitting.

If the filter length exceeds the number of cache ways, conflicts may occur if the input coefficients needed to compute one output coefficient map to the same cache set. For the 4-way set-associative cache of the Pentium 4, this only occurs for the CDF-9/7 transform. To avoid these conflicts we have proposed and evaluated two techniques: associativity-conscious loop splitting (ACLS) and lookahead. For image sizes that experience many cache conflict misses ACLS improves performance by a factor that ranges from 1.59 to 1.80, while the lookahead technique provides a speedup between 1.71 and 1.99. For image sizes that do not generate many conflict misses both schemes generally decrease performance slightly, due to the loop overhead needed for managing index variables and address calculations. With the exception of two image sizes, the lookahead technique performs slightly better than ACLS, because it incurs less loop overhead. Both ACLS and lookahead are general because they can also be applied to other cache organizations and/or filter lengths.

We are currently vectorizing the 2D DWT implementations discussed in this paper using the MMX and SSE instruction set extensions. Our final goal is to obtain a high-performance, parameterizable implementation of this important kernel.

7. REFERENCES

[1] G. Bernabe, J. M. Garcia, and J. Gonzales. Reducing 3D Wavelet Transform Execution Time Through the Streaming SIMD Extensions. In *Proc. 11th Euromicro Conf. on Parallel Distributed and Network based Processing*, February 2003.

[2] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[3] S. Chatterjee and C. D. Brooks. Cache-Efficient Wavelet Lifting in JPEG 2000. In *Proc. IEEE Int. Conf. on Multimedia*, pages 797–800, August 2002.

[4] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. 2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation. In *Proc. Int. Conf. on the High Performance Computing*, December 2002.

[5] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. Vectorization of the 2D Wavelet Lifting Transform Using SIMD Extensions. In *Proc. 17th IEEE Int. Symp. on Parallel and Distributed Image Processing and Multimedia*, 2003.

[6] B. D. Choi, K. S. Choi, M. C. Hwang, J. K. Cho, and S. J. Ko. Real-time DSP Implementation of Motion-JPEG2000 Using Overlapped Block Transferring and Parallel-Pass Methods. *Real-Time Imaging*, 10:277–284, 2004.

[7] C. Chrysafis and A. Ortega. Line-Based, Reduced Memory, Wavelet Image Compression. *IEEE Trans. on Image Processing*, 9(3):378–389, March 2000.

[8] A. Cohen, I. Daubechies, and J. C. F. Eauveau. Biorthogonal Bases of Compactly Supported Wavelets. *Communications on Pure and Appl. Math.*, 45(5):485–560, June 1992.

[9] I. Daubechies and W. Sweldens. Factoring Wavelet Transforms into Lifting Steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[10] D. He and W. Zhang. The Parallel Algorithm of 2-D Discrete Wavelet Transform. In *Proc. 4th IEEE Int. Conf. on Parallel and Distributed Computing Applications and Technologies*, pages 738–741, August 2003.

[11] Intel Corporation. *IA-32 Intel Architecture Optimization*, 2004. Order Number: 248966-011.

[12] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual Volume 3 System Programming Guide*, 2004. Order Number: 253668.

[13] M. W. Marcellin, M. J. Gormish, A. Bilgin, and M. P. Boliek. An Overview of JPEG 2000. In *Proc. Data Compression Conf.*, March 2000.

[14] P. Meerwald, R. Norcen, and A. Uhl. Cache Issues with JPEG2000 Wavelet Lifting. In *Proc. of Visual Communications and Image Processing*, January 2002.

[15] M. Rabbani and R. Joshi. An Overview of the JPEG2000 Still Image Compression Standard. *Signal Processing: Image Communication*, 17(1):3–48, January 2002.

[16] J. A. Shafer. Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression. Master's thesis, Department of Electrical and Computer Eng. University of Dayton, 2004.

[17] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Performance Comparison of SIMD Implementations of the Discrete Wavelet Transform. In *Proc. 16th IEEE Int. Conf. on Application Specific Systems Architectures and Processors (ASAP)*, pages 393–398, July 2005.

[18] A. N. Skodras, C. A. Christopoulos, and T. Ebrahimi. JPEG 2000: The Upcoming Still Image Compression Standard. In *Proc. 11th Portugues Conf. on Pattern Recongnition*, pages 359–366, May 2000.

[19] D. B. Stewart. Measuring Execution Time and Real-Time Performance. In *Embedded Systems Conf.*, pages 1–15, April 2001.

[20] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1996.

[21] W. Sweldens. The Lifting Scheme: A Custom-Design Construction of Biorthogonal Wavelets. *Journal of Applied and Computational Harmonic Analysis*, 3(2):186–200, 1996.

[22] M. A. Trenas, J. Lopez, E. L. Zapata, and F. Arguello. A Memory System Supporting the Efficient SIMD Computation of the Two Dimensional DWT. In *IEEE Int. Conf. on Acoustics Speech and Signal Processing*, volume 3, pages 1521–1524, May 1998.