Implementing the 2-D Wavelet Transform on SIMD-Enhanced General-Purpose Processors

Asadollah Shahbahrami, *Student Member, IEEE*, Ben Juurlink, *Senior Member, IEEE*, and Stamatis Vassiliadis, *Fellow, IEEE*

***Abstract*—The 2-D Discrete Wavelet Transform (DWT) consumes up to 68% of the JPEG2000 encoding time. In this paper, we develop efficient implementations of this important kernel on general-purpose processors (GPPs), in particular the Pentium 4 (P4). Efficient implementations of the 2-D DWT on the P4 must address three issues. First, the P4 suffers from a problem known as 64K aliasing, which can degrade performance by an order of magnitude. We propose two techniques to avoid 64K aliasing which improve performance by a factor of up to 4.20. Second, a straightforward implementation of vertical filtering incurs many cache misses. Cache performance can be improved by applying loop interchange, but there will still be many conflict misses if the filter length exceeds the cache associativity. Two methods are proposed to reduce the number of conflict misses which provide an additional performance improvement of up to 1.24. To show that these methods are general, results for the P3 and Opteron are also provided. Third, efficient implementations of the 2-D DWT must exploit the SIMD instructions supported by most GPPs, including the P4, and we present MMX and SSE implementations of horizontal and vertical filtering which provide a maximum speedup of 3.39 and 6.72, respectively.**

***Index Terms*—Cache, Discrete Wavelet Transform, memory hierarchy, multimedia extensions, SIMD.**

I. Introduction

THE Discrete Wavelet Transform (DWT) is a highly effective tool in image and video compression standards such as JPEG2000 and MPEG-4, since it achieves higher compression ratios than other transforms such as the DCT. A potential problem, however, is the high computational complexity of the DWT. We have measured the total execution time consumed by the DWT using the JasPer software tool kit [1]. The results show that the DWT consumes on average 46% of the total encoding time for lossless compression and even 68% for lossy compression. Results presented by other researchers [2] also show that the DWT consumes a significant fraction of the total JPEG2000 encoding time.

Manuscript received March 2, 2007; revised August 1, 2007. This work was supported by the Netherlands Organization for Scientific Research (NWO). The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Anna Hac.

A. Shahbahrami is with the Computer Engineering Laboratory, EEMCS, Delft University of Technology, 2628 CD Delft, The Netherlands, and also with the Department of Electrical Engineering, Faculty of Engineering, The University of Guilan, Rasht, Iran (e-mail: shahbahrami@ce.et.tudelft.nl).

B. Juurlink is with the Computer Engineering Laboratory, EEMCS, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: benj@ce.et.tudelft.nl).

S. Vassiliadis, deceased, was with the Computer Engineering Laboratory, EEMCS, Delft University of Technology, 2628 CD Delft, The Netherlands.

Digital Object Identifier 10.1109/TMM.2007.911195

In order to reduce complexity and improve performance, several researchers [3], [4] have proposed hardware implementations of the DWT. Programmable processors, however, are preferable to special-purpose hardware because they are more flexible, enable different transforms to be employed, and allow various filter bank lengths and various transform levels. Hence, in this paper we focus on general-purpose, programmable processors, in particular the Pentium 4 (P4).

A 2-D DWT consists of horizontal filtering along the rows followed by vertical filtering along the columns. In order to develop high-performance implementations of the 2-D DWT on general-purpose processors (GPPs) in general and the P4 in particular, the following issues need to be addressed.

First, the P4 suffers from a problem known as 64K aliasing, which can degrade performance by an order of magnitude. It occurs when two data blocks need to be cached simultaneously whose addresses differ by a multiple of 64K. In a straightforward implementation of the 2-D DWT, there is a 64K alias between the low- and high-pass values when the image size is a large power of two. We propose two techniques to avoid 64K aliasing. The first technique provides a speedup of up to 3.31, but for image sizes that do not suffer from 64K aliasing it reduces performance by up to 20%. The second technique achieves a speedup of up to 4.20 and incurs no performance penalty for image sizes that do not suffer from 64K aliasing.

Second, a plain implementation of vertical filtering incurs many cache misses. As shown in this paper, most of them can be avoided by applying loop interchange, but there will still be many conflict misses if the filter length exceeds the number of cache ways, in particular if the image size is a multiple of the cache size. We, therefore, propose two methods to reduce the number of conflict misses. The first technique improves performance by up to 80% and the second by up to 99%. For image sizes that do not generate many conflict misses, both techniques slightly decrease performance due to the overhead (loop overhead, address calculations) introduced by applying these techniques. Although 64K aliasing is a problem specific to the P4, the conflict avoidance methods are general and can be applied to other processors as well. To show this, results are also presented for the P3 and AMD Opteron processors.

Third, high-performance implementations of the 2-D DWT must exploit the data-level parallelism using the SIMD instructions supported by most GPPs. We describe how the 2-D DWT can be vectorized using MMX and SSE instructions. Vertical filtering is relatively straightforward to vectorize. Horizontal filtering is more difficult to vectorize, since it requires the data to be reorganized. The maximum speedups of the SIMD im-

plementations of horizontal and vertical filtering over the corresponding scalar versions are 3.39 and 6.72, respectively.

We make the following contributions. First, although implementations of the 2-D DWT on the P4 have been presented before, these works did not mention nor address 64K aliasing. Second, previous work has focused mainly on improving the spatial locality of vertical filtering. In contrast, our techniques eliminate the conflicts that may exist between the input values needed to compute one output value if the filter length is larger than the number of cache ways. Third, while related works have focused mainly on improving cache performance or on SIMD vectorization, we address both these issues.

This paper is organized as follows. Section II describes the discrete wavelet transform, the experimental environment, and the evaluation methodology. Section III proposes and evaluates two techniques to circumvent 64K aliasing. Section IV addresses the cache behavior of transforms with long filters and presents two techniques to avoid conflict misses. The SIMD implementations of the 2-D DWT are described in Section V. Finally, conclusions and future work are given in Section VI.

II. Preliminaries

In this section we describe the discrete wavelet transform, the experimental setup, and the reference implementation.

A. Discrete Wavelet Transform

A 2-D DWT consists of *horizontal filtering* along the rows followed by *vertical filtering* along the columns. Traditionally, both are implemented by convolution methods such as finite impulse response (FIR) filters. In this paper we consider two convolution methods, namely Daubechies' transform with four coefficients [5] (Daub-4), and the Cohen, Daubechies and Feauveau 9/7 filter [6] (CDF-9/7), and one lifting scheme, namely the (5, 3) lifting scheme [7]. The (5, 3) lifting scheme is included because it has low computational complexity and performs reasonably well for lossy as well as lossless compression compared to other filters [8]. We remark that the (5, 3) transform and CDF-9/7 are included in Part 1 of the JPEG2000 standard. Furthermore, these transforms have been considered in many recent papers (e.g., [2]–[5]). Although we have used these three transforms, the proposed techniques are general and equally applicable to other transforms.

B. Experimental Setup

Most programs were implemented in C and compiled using gcc with optimization level *-O2*. As is usual in C, the matrices are stored in row-major order. The SIMD implementations were implemented in assembly using the MMX and SSE instruction sets.

As experimental platforms we employed the Pentium 3, Pentium 4, and AMD Opteron processors. The main architectural parameters of our systems are summarized in Table I.

All programs were executed on a lightly loaded system. Performance was measured using the IA-32 cycle counter [10], which provides a very precise tool for measuring the time that elapses between two different points in the execution of a program. To eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache have been used. That means that the function is repeatedly (K times) executed and the fastest time is recorded. This minimizes the effects of both instruction and data cache misses.

TABLE I
Parameters of the Experimental Platforms

Processor	Intel P3	AMD Opteron	Intel P4
CPU Frequency	451MHz	2.0GHz	3.0GHz
L1 Data Cache	16 KBytes, 2-way SA, 32B line size	64 KBytes, 2-way SA, 64B line size	8 KBytes, 4-way SA, 64B line size
L2 Cache	512 KBytes, 8-way SA, 32B line size	1 MBytes, 8-way SA, 32B line size	512 KBytes, 8-way SA, 64B line size

C. Reference Implementation

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 evicted from the cache 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. Our results [11], [12] have clearly shown that this improves performance significantly. 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.

To avoid a rearrangement step, our implementations employ an auxiliary matrix as proposed in [2]. The horizontal filtering phase reads from the input image `img` and stores the results in the auxiliary matrix `tmp`. Thereafter, the vertical filtering phase reads from `tmp` and stores the results back to `img`.

III. Avoiding 64K Aliasing

In the Pentium 4 there is a phenomenon known as *64K aliasing* [13]. It occurs if two or more data blocks whose addresses differ by a multiple of 64K need to be cached simultaneously. If it occurs, the associativity of the cache is useless and the effectiveness of the cache is greatly reduced. The reasons for the 64K aliasing problem are not well documented. Some sources [14] say it is due to incomplete tag encoding. More precisely, only 16 bits are used for the cache lookup: 6 for the block offset, 5 for the index, and bits 11 to 15 for the tag [15]. The remaining tag bits come from the DTLB. Because of this, references to addresses with the same 16 lower-order bits (i.e., addresses that are 2^{16} bytes or a multiple thereof apart) are not resolvable in the L1 data cache. According to Intel documentation [13], the instruction that accesses the second 64K-aliasing data item has to wait until the first one is written from the cache. This clearly obstructs out-of-order processing.

For some image sizes, the 2-D DWT suffers from 64K aliasing. To illustrate this, Fig. 1 depicts the slowdown of the reference implementation of vertical filtering over horizontal filtering on the P4. Even though they perform the same number

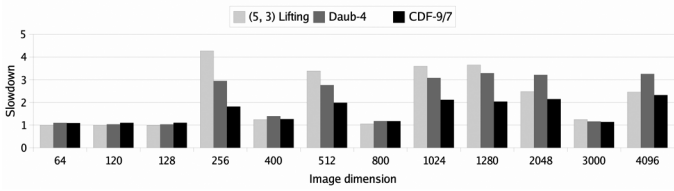

Fig. 1. Slowdown of vertical filtering over horizontal filtering on the P4.

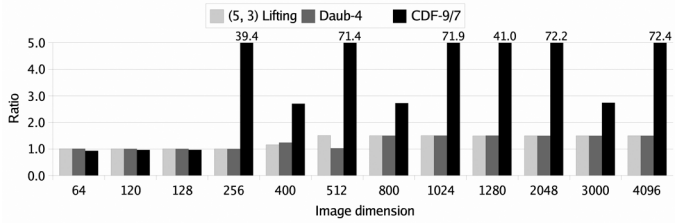

Fig. 2. Ratio of the number of cache misses incurred by vertical filtering to the number of cache misses incurred by horizontal filtering for an 8 KB 4-way set-associative L1 data cache with a line size of 64 bytes.

```
void Lifting53_vertical(){
  for (i=0, ii=1; ii<N; i++, ii+=2)
    for (j=0; j<N; j++){
      img[i+N/2][j] = tmp[ii][j] -
               ((tmp[ii-1][j]+tmp[ii+1][j])>>1);
      img[i][j] = tmp[ii-1][j] +
               ((img[i+N/2][j]+img[i+N/2-1][j]+2)>>2);
    }
}
```

Fig. 3. C implementation of vertical filtering using the (5, 3) lifting scheme.

of operations, for some image sizes vertical filtering is substantially slower (up to a factor of 4.3) than horizontal filtering. One reason for this could be the cache behavior. To analyze if this is the case, Fig. 2 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 with the cache configured as the L1 data cache of the P4.

It can be seen that the slowdown of vertical filtering over horizontal filtering cannot be explained by the cache miss behavior. For example, when the image size is 256×256, vertical filtering is slower than horizontal filtering by a factor of 4.27 for the lifting transform, and by a factor of 2.95 for the Daub-4 transform. For both transforms, however, they incur about the same number of cache misses. For the CDF-9/7 transform, on the other hand, vertical filtering is slower by a factor of 1.82 but generates more than 39 times as many cache misses as horizontal filtering. Similar behavior can be observed for other image sizes. Hence, the large slowdown of vertical versus horizontal filtering should not (only) be attributed to cache misses but mainly to 64K aliasing.

To further explain why and when 64K aliasing occurs, Fig. 3 depicts a C implementation of vertical filtering using the lifting scheme. It can be seen that one iteration of the inner loop accesses `img[i][j]` and `img[i+N/2][j]`. Hence, 64K aliasing occurs if $cN^2/2$ is a multiple of 2^{16}, where c is the number of bytes needed to represent one wavelet coefficient. Since c is 2 for the lifting and 4 for the Daub-4 and CDF-9/7

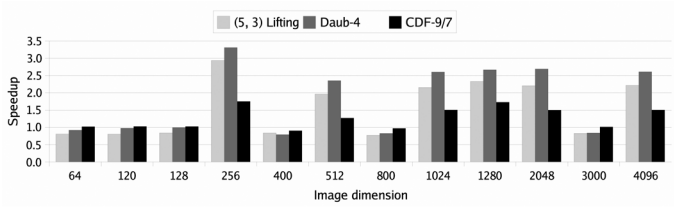

Fig. 4. Speedup of vertical filtering over the reference implementation achieved by loop fission.

transforms, for square $N \times N$ images 64K aliasing occurs if $N = 256$ (since $2 \cdot 256^2/2 = 2^{16}$), for powers of 2 larger than 256, and for $N = 1280$ (since $2 \cdot 1280^2/2 = 25 \cdot 2^{16}$). We remark that although we focus on square images, 64K aliasing may also occur for non-square images. For $N \times M$ images, it occurs when $cNM/2$ is a multiple of 2^{16}.

To circumvent 64K aliasing, we propose and evaluate two techniques. The first idea is to split the inner loop so that the low-pass (`img[i][j]`) and high-pass values (`img[i+N/2][j]`) are calculated in separate loops. In this way the 64K alias between them is removed. This is actually a well-known compiler technique called *loop fission*. Loop fission, however, is usually applied to enable other transformations such as loop interchange and vectorization, while we apply it to avoid 64K aliasing.

Fig. 4 depicts the speedup resulting from this program transformation. For those image sizes that suffer from 64K aliasing (as explained above, powers of two larger than 256×256 and 1280×1280), loop fission 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. However, for those image sizes that do not suffer from 64K aliasing, loop fission reduces performance by up to 20%. This is due to the following reasons. First and most importantly, loop fission removes the temporal reuse that exists between the calculation of the high-pass and low-pass values. As can be seen in Fig. 3, the first statement in the loop body uses `tmp[ii-1][j]` and so does the second statement. After loop fission has been applied, the two statements are in different loops and this temporal reuse has been removed. Second, loop fission increases loop overhead but this overhead could be reduced by unrolling the loop.

The second technique we propose is to offset the memory address of the high-pass value by one or two rows depending on the transform. In other words, instead of storing the value in `img[i+N/2][j]`, it is stored in `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 to apply this method, the matrices have to be extended with one row.

Fig. 5 depicts the speedup achieved by the offsetting technique. For those image sizes that suffer from 64K aliasing, it improves performance by a factor ranging from 3.07 to 4.20 for the lifting transform, from 2.99 to 3.11 for Daub-4, and from 1.41 to 1.69 for CDF-9/7. Moreover, the offsetting technique does *not* incur a performance penalty for image sizes that do

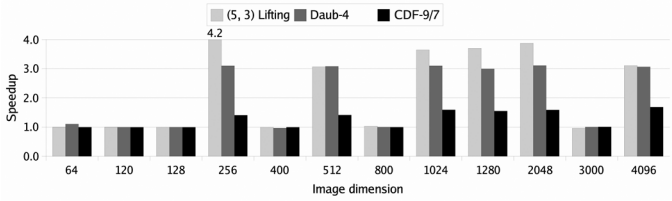

Fig. 5. Performance improvement achieved by the offsetting technique.

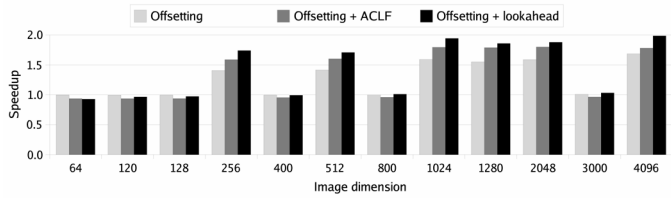

Fig. 6. Comparison of the speedups obtained by applying offsetting alone to the speedups achieved by applying associativity-conscious loop fission or lookahead in addition to offsetting for the CDF-9/7 transform.

not suffer from 64K aliasing. 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 fission.

IV. Cache Optimization

Fig. 2 shows that for small images (up to 128×128), vertical filtering does not produce more cache misses than horizontal filtering. For images larger than 800×800, however, vertical filtering generates about 50% more misses than horizontal filtering for the lifting and Daub-4 transforms. For the Daub-4 transform this can be explained as follows. To compute row i of `img`, it uses rows $2i, 2i+1, 2i+2, 2i+3$ of `tmp`. Hence, to compute row $i+1$, it uses rows $2i+2, 2i+3, 2i+4, 2i+5$. This implies that rows $2i+2$ and $2i+3$ are reused, provided 4 rows can be kept in cache. When $N \geq 800$, however, they cannot (since $4 \times 4 \times 800 > 8$ KB), which is why vertical filtering generates more misses than horizontal filtering. For the lifting transform this actually already occurs for $N = 512$, because this transform does not access four consecutive input rows.

More serious behavior, however, is exhibited by the CDF-9/7 transform. When N is (a multiple of) a large power of two, vertical filtering generates up to 72.4 times as many cache misses as horizontal filtering. This can be explained as follows. As noted before, the L1 data cache of the P4 is 4-way set associative. Each block maps to a unique set and can be placed in any of the four elements of that set. When $N = 512$ or a multiple thereof, however, the nine blocks that are needed to compute one block of output data map to the same set and, hence, evict each other from the cache. Because of this, many conflict misses are generated and the reuse that exist between the computation of `img[i][j]` and `img[i][j+1]` (provided they are in the same cache block) is destroyed. When $N = 256$ or $N = 1280$, five input blocks map to the same set, causing also many cache misses but fewer than when all nine blocks map to the same set. For the lifting and Daub-4 transforms, this problem does not exist because their filter lengths is equal to the number of cache ways.

A. Proposed Techniques

The first method is referred to as *associativity-conscious loop fission* (ACLF). The idea is to split the loop that computes one row of wavelet output into multiple loops so that each loop accesses at most n rows. Each loop computes the partial results that can be computed by accessing the first n rows of input data. The remaining loops add their results to these partial results.

In the second scheme, which is called *lookahead*, the rows are processed in a skewed manner. There is only one loop, as in the original algorithm. In iteration j $(0 \leq j < N)$ we compute a partial results for the output element `img[i][j]` but, in the same iteration, we compute a partial result for the output element `img[i][(j+B/c) mod N]` that is located B/c columns ahead, for the element `img[i][(j+2*B/c) mod N]`, and so on. Here B is the cache line size in bytes and c is the number of bytes per wavelet coefficient, as before. To compute a partial result for `img[i][j]`, we process n input elements `tmp[ii][j]`, `tmp[ii+1][j]!`, ..., `tmp[ii+n-1][j]!`,. A partial result for `img[i][(j+B/c) mod N]` is computed using the elements `tmp[ii+n][(j+B/c) mod N]!...`, `tmp[ii+2n-1][(j+B/c) mod N]`, and so on. So in each iteration, L/n partial results are computed, where L is the filter length. In later iterations, partial results corresponding to the same column are accumulated. This scheme ensures that no more than n input elements accessed in one loop iteration map to the same cache set.

Fig. 6 compares the performance improvements obtained by applying ACLF or lookahead in addition to offsetting to the speedup achieved by applying offsetting alone. Results are presented only for CDF-9/7, since only this transform suffers from both 64K aliasing as well as excessive cache misses. For image sizes that experience many cache conflicts ($N = 256$ and multiples thereof), avoiding them provides additional performance improvements. For example, applying offsetting alone provides a speedup of up to 1.69, while combining it with the lookahead technique yields a speedup of up to 1.99. In general, the lookahead technique performs slightly better than ACLF. This is because it incurs less loop overhead than ACLF. For image sizes that do not generate many conflict misses, both schemes generally slightly decrease performance (by at most 7%), due to overhead needed for managing loop and index variables and address calculations.

As mentioned before, both ACLF and the lookahead technique are general and architecture independent. By this we mean that, 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 $L = 4$ and $n = 2$. To validate this claim, Fig. 7 depicts the speedup obtained by applying ACLF and the lookahead techniques on the P3 and Opteron processors. Analytically it can be determined that on the P3 many conflict misses occur when $N = 1024, 2048, 4096$ and on the Opteron when $N = 2048, 4096$ and, to a lesser extent, for $N = 1280$. Fig. 7

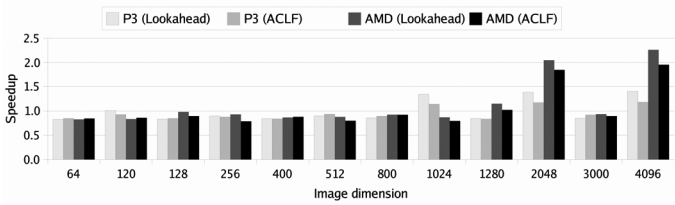

Fig. 7. Speedups obtained by applying ACLF and the lookahead technique over the reference implementation of the CDF-9/7 transform on the P3 and Opteron.

shows that for these image sizes ACLF provides a performance improvement ranging from 20% to 25% on the P3 and from 10% to 110% on the Opteron. On the other hand, the lookahead technique improves performance by 35% to 45% on the P3 and by 15% to 125% on the Opteron.

B. Related Work

Meerwald *et al.* [16] also observed that a straightforward implementation of vertical filtering that processes the elements along the columns can generate many cache misses and proposed two techniques, row extension and aggregation, to avoid 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 [16], 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. The number of columns filtered consecutively is called the "aggregation factor". If the aggregation factor is equal to the image width, aggregation is identical to loop interchange, which we have used as the baseline implementation.

A potential advantage of aggregation over loop interchange is that it can exploit the reuse between the rows of input values needed to compute consecutive rows of output values, while loop interchange cannot for large images. To investigate this, we have also implemented aggregation. Contrary to our initial expectations, aggregation performs worse than loop interchange. This is due to three reasons. First, aggregation incurs more loop overhead than loop interchange, since it consists of three loop nests instead of two. Second, on a cache miss the P4 prefetches the next block. Loop interchange takes advantage of this but aggregation does not when it reaches the end of a group of columns. Third, cache performance is not critical in these scalar implementations.

Chatterjee and Brooks [2] proposed two optimizations: strip-mining (which is identical to aggregation) and recursive data layout. 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 level is reported. Furthermore, the first decomposition level takes more time than all subsequent decomposition levels together.

Chaver *et al.* [17] combined aggregation with a line-based approach [18], which starts vertical filtering as soon as a sufficient number of rows (determined by the filter length) 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 P4 (as well as a P3), they did not mention the 64K aliasing problem.

Komi *et al.* [19] as well as Lee *et al.* [20] proposed block-based approaches to improve the cache efficiency for the 2-D DWT. In [19] equations are presented which allow to find the optimal block size, assuming a fully associative data cache. In [20] the block size is equal to one way of the two-way set-associative L1 data cache.

Andreopoulos *et al.* [21] identified three categories of DWT implementations: strictly breadth-first (SBF), roughly depth-first (RDF), and strictly depth-first (SDF). SBF implementations filter all rows horizontally before filtering all columns vertically. Our DWT implementations belong to this category. RDF implementations interleave periods of horizontal filtering with periods of vertical filtering. The line-based and block-based approaches belong to this category. SDF corresponds to RDF with minimal interleaving period. Andreopoulos *et al.* have shown that RDF implementations incur fewer misses than SBF implementations. However, from the exposition in [21] it is unclear if loop interchange has been applied to the presented SBF implementations. Furthermore, although they have presented results for the Pentium, only cache miss rate results are presented and no real execution times.

There are two main differences between these related works and our work on improving the memory behavior of the 2-D DWT. First, previous work did not mention nor address 64K aliasing. Second, previous work did not remove the conflicts that may exist between the input coefficients needed to compute one output coefficient. As was shown in Section IV (cf. Fig. 2), if the filter length is larger than the number of cache ways and the image width is a large power of two, then many cache misses are generated. The ACLF and lookahead techniques remove these conflicts.

V. SIMD Vectorization

An efficient implementation of the DWT on the P4 as well as other GPPs must exploit the SIMD extensions provided by these processors. In this section we present MMX/SSE implementations of the DWT and present performance results on the Intel P4 processor. This section is organized as follows. In Section V-A we discuss the SIMD implementations of the convolutional methods (Daub-4 and CDF-9/7). Thereafter, in Section V-B we present an SIMD implementation of the lifting scheme. Experimental results are provided in Section V-C.

A. SIMD Implementations of Convolutional Methods

The SIMD implementations of Daub-4 and CDF-9/7 are very similar. Both process single-precision floating-point values and apply filtering by multiplying the filter coefficients with the input samples and accumulating the results. They will therefore be discussed together.

Under a row-major image layout, it is relatively straightforward to vectorize vertical filtering using SSE instructions. This is because the elements that can be processed simultaneously are stored consecutively in memory. Consider, for example, the Daub-4 transform and let $x_{i,j}$ be the input samples, let $c_0, \ldots, c_3$ denote the low-pass filter coefficients, and let $L_{i,j}$

be the low-pass values. Then vertical filtering of the low-pass values is given by

$$\begin{aligned}(L_{i,j}\ L_{i,j+1}\ L_{i,j+2}\ L_{i,j+3}) &= (c_0\ c_0\ c_0\ c_0) \times (x_{2i,j}\ x_{2i,j+1}\ x_{2i,j+2}\ x_{2i,j+3}) \\ &+ (c_1\ c_1\ c_1\ c_1) \times (x_{2i+1,j}\ x_{2i+1,j+1}\ x_{2i+1,j+2}\ x_{2i+1,j+3}) \\ &+ (c_2\ c_2\ c_2\ c_2) \times (x_{2i+2,j}\ x_{2i+2,j+1}\ x_{2i+2,j+2}\ x_{2i+2,j+3}) \\ &+ (c_3\ c_3\ c_3\ c_3) \times (x_{2i+3,j}\ x_{2i+3,j+1}\ x_{2i+3,j+2}\ x_{2i+3,j+3}).\end{aligned} \tag{1}$$

In this equation, the operator $\times$ denotes elementwise vector multiplication. Similar equations can be drawn for the high-pass values and other convolutional filters. This equation can be mapped almost one-to-one to SSE instructions. The only technical detail is that each coefficient needs to be replicated four times. Some SIMD extensions provide instructions for this (splat instructions), but since SSE does not, we have replicated each coefficient four times in memory.

Horizontal filtering is more difficult to vectorize, however. In this case, the low-pass values can be calculated using the equation

$$\begin{aligned}(L_{i,j}\ L_{i,j+1}\ L_{i,j+2}\ L_{i,j+3}) &= (c_0\ c_0\ c_0\ c_0) \times (x_{i,2j}\ x_{i,2j+2}\ x_{2i,2j+4}\ x_{2i,2j+6}) \\ &+ (c_1\ c_1\ c_1\ c_1) \times (x_{i,2j+1}\ x_{i,2j+3}\ x_{2i,2j+5}\ x_{2i,2j+7}) \\ &+ (c_2\ c_2\ c_2\ c_2) \times (x_{i,2j+2}\ x_{i,2j+4}\ x_{2i,2j+6}\ x_{2i,2j+8}) \\ &+ (c_3\ c_3\ c_3\ c_3) \times (x_{i,2j+3}\ x_{i,2j+5}\ x_{2i,2j+7}\ x_{2i,2j+9}).\end{aligned} \tag{2}$$

To map this equation to SIMD instructions, a vector-vector multiplication (dot product) instruction would have been useful, but since SSE does not provide such an instruction, we have to rearrange the elements so that, for example, the input samples $x_{i,2j}, x_{i,2j+2}, x_{2i,2j+4}$, and $x_{2i,2j+6}$ are stored consecutively in an SSE register. Fig. 8 shows the SSE code that computes four low-pass values. It can be seen that many overhead (unpack) instructions are needed.

`movaps xmm0, (esi)` ; xmm0 =	a_3	a_2	a_1	a_0
`movaps xmm1,16(esi)` ; xmm1 =	a_7	a_6	a_5	a_4
`movaps xmm2, xmm0` ; xmm2 =	a_3	a_2	a_1	a_0
`unpcklps xmm0, xmm1` ; xmm0 =	a_5	a_1	a_4	a_0
`unpckhps xmm2, xmm1` ; xmm2 =	a_7	a_3	a_6	a_2
`movaps xmm1, xmm0` ; xmm1 =	a_5	a_1	a_4	a_0
`unpcklps xmm0, xmm2` ; xmm0 =	a_6	a_4	a_2	a_0
`unpckhps xmm1, xmm2` ; xmm1 =	a_7	a_5	a_3	a_1
`movups xmm2,8(esi)` ; xmm2 =	a_5	a_4	a_3	a_2
`movups xmm3,24(esi)` ; xmm3 =	a_9	a_8	a_7	a_6
`movaps xmm4, xmm2` ; xmm4 =	a_5	a_4	a_3	a_2
`unpcklps xmm2, xmm3` ; xmm2 =	a_7	a_3	a_6	a_2
`unpckhps xmm4, xmm3` ; xmm4 =	a_9	a_5	a_8	a_4
`movaps xmm3, xmm2` ; xmm3 =	a_7	a_3	a_6	a_2
`unpcklps xmm2, xmm4` ; xmm2 =	a_8	a_6	a_4	a_2
`unpckhps xmm3, xmm4` ; xmm3 =	a_9	a_7	a_5	a_3
`movaps xmm4, xmm0` ; xmm4 =	a_6	a_4	a_2	a_0
`movaps xmm5, xmm1` ; xmm5 =	a_7	a_5	a_3	a_1
`movaps xmm6, xmm2` ; xmm6 =	a_8	a_6	a_4	a_2
`movaps xmm7, xmm3` ; xmm7 =	a_9	a_7	a_5	a_3

Fig. 8. Computing four low-pass values for horizontal filtering using SSE instructions (Daub-4 transform).

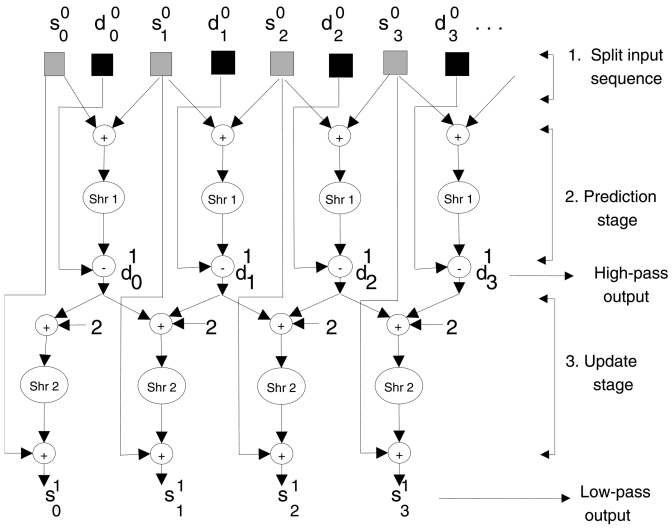

Fig. 9. Part of the data flow graph of the forward integer-to-integer lifting transform using the (5, 3) filter bank (Shr = Shift right).

B. MMX Implementation of the Lifting Scheme

The SIMD implementation of the (5, 3) lifting scheme is significantly different from the SSE implementations of Daub-4 and CDF-9/7 for the following reasons. First, the (5, 3) lifting transform uses integer arithmetic and hence its SIMD implementation employs MMX instructions. Second, in the MMX implementation there are no multiplication operations, since the input values need to be divided by powers of 2 which can be accomplished using shift operations. Third, because of its structure, the (5, 3) lifting scheme is vectorized in a completely different way than the convolutional transforms.

The lifting operation consists of several stages. First, the original 1-D input signal is split into a subsequence consisting of the even-numbered input values $\{s_i^0\}$ and a subsequence containing the odd-numbered input values $\{d_i^0\}$. Thereafter, the prediction stage produces the high-pass output values $\{d_i^1\}$ and the update stage generates the low-pass output values $\{s_i^1\}$. Specifically, the forward transform of the (5, 3) filter bank used in this paper is given by [8]

$$d_i^1 = d_i^0 - \left\lfloor \frac{s_i^0 + s_{i+1}^0}{2} \right\rfloor \tag{3}$$

$$s_i^1 = s_i^0 + \left\lfloor \frac{d_{i-1}^1 + d_i^1 + 2}{4} \right\rfloor. \tag{4}$$

Fig. 9 depicts a part of the data flow graph of the (5, 3) lifting scheme based on (3) and (4). In order to vectorize horizontal filtering we need to rearrange the data so that the even and odd subsequences are placed in different registers. Furthermore, because s_i^0 and s_{i+1}^0 have to be added, two copies of the even subsequence are required, one that starts with s_0^0 and one that starts

`movq`	`mm0, (esi)`	; **mm0** =	s_0^0	d_0^0	s_1^0	d_1^0	s_2^0	d_2^0	s_3^0	d_3^0
`movq`	`mm1,8(esi)`	; **mm1** =	s_4^0	d_4^0	s_5^0	d_5^0	s_6^0	d_6^0	s_7^0	d_7^0
`pxor`	`mm7, mm7`	; **mm7** =	0	0	0	0	0	0	0	0
`movq`	`mm2, mm0`	; **mm2** =	s_0^0	d_0^0	s_1^0	d_1^0	s_2^0	d_2^0	s_3^0	d_3^0
`punpcklbw`	`mm0, mm7`	; **mm0** =	s_0^0		d_0^0		s_1^0		d_1^0	
`punpckhbw`	`mm2, mm7`	; **mm2** =	s_2^0		d_2^0		s_3^0		d_3^0	
`punpcklbw`	`mm1, mm7`	; **mm1** =	s_4^0		d_4^0		s_5^0		d_5^0	
`movq`	`mm3, mm0`	; **mm3** =	s_0^0		d_0^0		s_1^0		d_1^0	
`punpcklwd`	`mm0, mm2`	; **mm0** =	s_0^0		s_2^0		d_0^0		d_2^0	
`punpckhwd`	`mm3, mm2`	; **mm3** =	s_1^0		s_3^0		d_1^0		d_3^0	
`movq`	`mm4, mm0`	; **mm4** =	s_0^0		s_2^0		d_0^0		d_2^0	
`punpcklwd`	`mm0, mm3`	; **mm0** =	s_0^0		s_1^0		s_2^0		s_3^0	
`punpckhwd`	`mm4, mm3`	; **mm4** =	d_0^0		d_1^0		d_2^0		d_3^0	
`punpcklwd`	`mm2, mm1`	; **mm2** =	s_2^0		s_4^0		d_2^0		d_4^0	
`punpcklwd`	`mm3, mm2`	; **mm3** =	s_1^0		s_2^0		s_3^0		s_4^0	

Fig. 10. MMX instructions needed to rearrange the elements for the (5,3) lifting scheme.

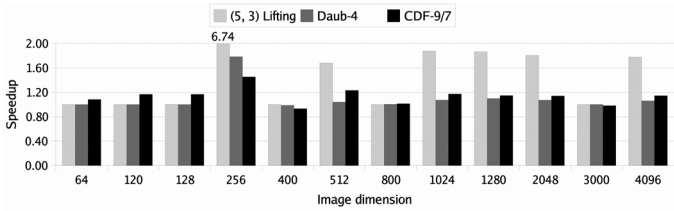

Fig. 11. Performance improvements achieved by applying the offsetting technique to the SIMD implementations of all three transforms and, in addition, the lookahead technique to CDF-9/7.

with s_1^0. Fig. 10 shows the MMX code that achieves this rearrangement. It can be seen that many unpack instructions are required to achieve this. After the code has been executed, the first four high-pass output values can be computed by adding `mm0` with `mm3`, shifting the results to the right by 1 bit position, and adding these results to `mm4`.

As was the case for the convolutional transforms, vertical filtering is easier to vectorize. In this case, the even and odd subsequences do not have to be split because they correspond to different rows. A drawback of vertical filtering compared to horizontal filtering is, however, that the previous high-pass output values which are needed for the update stage cannot be kept in a register, while in horizontal filtering they can. For example, in vertical filtering, after calculating the high-pass values $\{d_{i+1,j}^1, d_{i+1,j+1}^1, d_{i+1,j+2}^1, d_{i+1,j+3}^1\}$, the computation of the low-pass values $\{s_{i+1,j}^1, s_{i+1,j+1}^1, s_{i+1,j+2}^1, s_{i+1,j+3}^1\}$ should start. For this, access to the previous row to load the four calculated high-pass values $\{d_{i,j}^1, d_{i,j+1}^1, d_{i,j+2}^1, d_{i,j+3}^1\}$ is necessary. Consequently, the access pattern in vertical filtering is more complex than the access pattern in horizontal filtering.

C. Performance Results

First, we have applied the offsetting technique to the SIMD implementations of all three transforms and, in addition, the lookahead technique to CDF-9/7. The resulting speedups are depicted in Fig. 11. It can be seen that applying offsetting to the MMX implementation of the (5,3) lifting transform improves performance significantly. For those image sizes

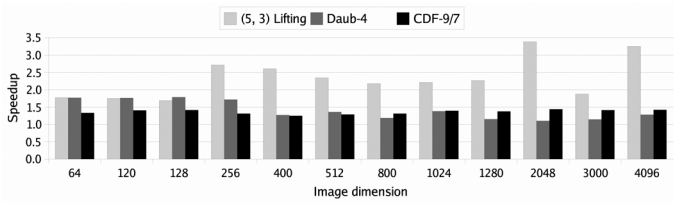

Fig. 12. Speedup of the SIMD implementations of horizontal filtering over the scalar versions.

that suffer from 64K aliasing, it improves performance by factors ranging from 1.68 to 6.74. For the Daub-4 and CDF-9/7 transforms, however, the attained speedups are comparatively small. Applying offsetting to Daub-4 provides a speedup of 1.78 for images of size 256×256. For the other image sizes that suffer from 64K aliasing, the speedups are smaller than 1.10. Applying both offsetting and lookahead to CDF-9/7 improves performance by factors ranging from 1.14 to 1.45 when 64K aliasing as well as excessive cache conflict misses occur. The reason for this behavior is that vectorization already (partially) eliminates 64K aliasing. The SSE implementations of the convolutional methods, for example, load 32 bytes of data (half a cache line) into registers before accessing a different cache line that could conflict with the current one. The MMX implementation of the (5,3) lifting scheme, on the other hand, loads 16 bytes of data into registers before accessing a different cache line (see the first two lines of the code given in Fig. 8). Hence, in this implementation 64K aliasing still occurs, but to a lesser extent than in the scalar version.

Fig. 12 depicts the speedups of the SIMD implementations of horizontal filtering over the corresponding scalar version. The largest speedups are obtained for the (5,3) lifting scheme. For this transform the speedup ranges from 1.69 to 3.39, while the speedups for Daub-4 and CDF-9/7 range from 1.10 to 1.79 and from 1.25 to 1.44, respectively. There are three main reasons why the speedups for the (5,3) lifting scheme are higher than for the Daub-4 and CDF-9/7 transforms. First, there are no misaligned memory accesses in the MMX implementation of horizontal filtering using the (5,3) lifting scheme, while in the SSE implementations there are. Although SSE permits misaligned memory accesses, they are much slower than aligned memory accesses. Second, there are more MMX execution units than SSE units. This implies that more MMX instructions can be executed in parallel. Third, the MMX implementation of the (5,3) lifting scheme performs more arithmetic operations per wavelet sample than the SSE implementations of Daub-4 and CDF-9/7. Because SIMD vectorization significantly reduces the CPU component of the execution time, horizontal filtering using Daub-4 and CDF-9/7 has become memory-bound.

Two other important observations can be drawn from Fig. 12. First, the speedups for the (5,3) lifting scheme are lower for small images ($N \leq 128$) than for larger images, while for Daub-4 the opposite behavior can be observed. Second, since all SIMD implementations perform four operations in one instruction, the expected maximum speedup is 4, but the attained speedups are smaller. The first behavior can be explained as follows. When $N \leq 128$, almost all reads hit the L1 data cache

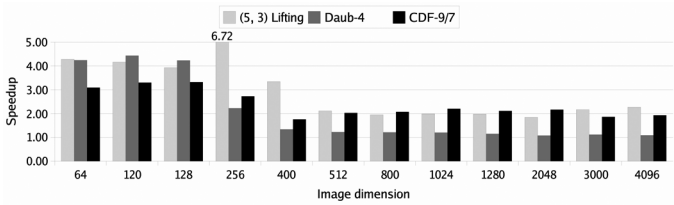

Fig. 13. Speedup of the SIMD implementations of vertical filtering over scalar versions.

(except for compulsory misses). Hence, the speedups obtained for these image sizes are the speedups resulting from SIMD vectorization. When $N > 128$, however, other factors start to play a role. For (5,3) lifting, the speedup increases because the MMX implementation incurs fewer cache conflicts and hence fewer memory stall cycles than the scalar implementation. For Daub-4, on the other hand, the speedup decreases because this implementation has become memory-bound. The fact that the obtained speedups are smaller than 4 even when $N \leq 128$ is mainly due to the overhead instructions required to vectorize horizontal filtering.

Fig. 13 depicts the speedups for vertical filtering. As explained in the previous sections, vertical filtering is easier to vectorize and incurs less overhead than horizontal filtering. This explains why the obtained speedups are about the maximum speedup of 4 for images smaller than 256×256. For the lifting and Daub-4 transforms the speedups for these image sizes are even larger than 4 in all but one case due to reduction of loop overhead. For CDF-9/7, the speedups are slightly smaller (around 3.32), because due to the small number of SSE registers, this implementation needs to spill registers to memory. When $N > 256$, however, the obtained speedups are smaller. For the lifting and CDF-9/7 transforms they are around 2 in most cases, but for Daub-4 the average speedup for images larger than 256×256 is only 1.26. Again this should be attributed to a memory bandwidth bottleneck.

D. Related Work

SIMD vectorization of the 2-D DWT has been considered in [17]–[22]. Chaver *et al.* [17] used SSE and the CDF-9/7 filter. They focused on automatic vectorization and did not consider assembly-level programming. The Intel compiler, however, can only vectorize simple loops, and therefore some manual code modifications had to be performed. Furthermore, only horizontal filtering could be automatically vectorized (they assumed column-major order). They also combined aggregation with a line-based approach for their SIMD implementation. In [9] they have vectorized vertical filtering of CDF-9/7 by hand using built-in SSE functions. In order to do so, however, an additional data transposition stage was required, which reduces the benefits of SIMD vectorization.

Kutil [22] has implemented the (9,7) lifting scheme using built-in SSE functions. He proposed a single loop approach to SIMD vectorization. In this approach horizontal and vertical filtering are combined into a single loop. This is called line-based computation in [18] and pipeline computation in [17], where it has been used to vectorize the CDF-9/7 transform. The single-loop approach requires a buffer whose size is equal to 16 rows of data. If this buffer does not fit in the cache, the temporal locality will be reduced.

VI. Conclusions and Future Work

In this paper we have focused on developing efficient implementations of the 2-D DWT on general-purpose, programmable processors, in particular the Pentium 4. Our contributions can be summarized as follows.

First, a simple and effective technique to improve the cache locality of vertical filtering is loop interchange. We have identified, however, that for certain image sizes the resulting implementation suffers from a phenomenon known as 64K aliasing. To avoid this problem, two techniques have been applied: loop fission and offsetting. For image sizes that suffer from 64K aliasing, loop fission provides a speedup that ranges from 1.27 to 3.31, depending on the applied transform, while offsetting achieves speedups between 1.41 and 4.20. Loop fission, however, incurs more loop overhead and, more seriously, destroys the temporal locality between the low- and high-pass values. Consequently, for image sizes that do not suffer from 64K aliasing it reduces performance by up to 20%. Because offsetting does not destroy the temporal reuse, we conclude that it is better than loop fission.

We have also shown that for certain image sizes, vertical filtering (with interchanged loops) still generates many more misses than horizontal filtering. On the P4, this happens in particular for the CDF-9/7 transform. The reason is that the filter length exceeds the number of cache ways. Because of this, conflicts occur if the input coefficients needed to compute one output coefficient map to the same cache set. To avoid these conflicts two techniques have been applied: associativity-conscious loop fission (ACLF) and lookahead. For image sizes that experience many cache conflict misses ACLF 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 other image sizes, both schemes generally decrease performance slightly, due to the increased loop overhead. Except for two image sizes, the lookahead technique performs slightly better than ACLF, because it incurs less loop overhead. Both schemes are general because they can also be applied to other cache organizations and/or filter lengths. To show this, results for the P3 and Opteron have also been presented.

To further enhance performance, the SIMD instructions provided by most general-purpose, programmable processors must be exploited. We have presented MMX implementations of the lifting transform and SSE implementations of the convolutional transforms. While vertical filtering is relatively straightforward to vectorize, horizontal filtering requires to rearrange the elements (sub-words) within a register. Mainly because of this overhead, the speedups obtained for horizontal filtering are relatively small, ranging from 1.69 to 3.39 for the lifting transform and from 1.10 to 1.79 for the convolutional transforms. Because vertical filtering does not incur this overhead, the speedups approach the ideal speedup of 4 when most reads hit the L1 data cache. For larger images, however, the obtained speedups are smaller, because the computation becomes memory-bound. This is especially the case for Daub-4

which has a smaller computation-to-communication ratio than the other two transforms.

Amongst others, our work has shown that it is difficult to obtain a single implementation of the 2-D DWT that works well for all image sizes, because most techniques incur some overhead. This indicates that in order to obtain the fastest implementation of this important kernel, a parameterizable implementation is needed that takes into account factors such as the cache organization of the target processor, the image size, the filter length, etc. Specifically, focusing on the cache conflict problem, if the cache organization, image size, and filter length are such that the number of input blocks needed to compute one output block exceeds the number of cache ways, then the lookahead technique should be applied. Otherwise, the reference implementation should be called.

We remark that the proposed methods can also be applied to other kernels and applications. For example, the convolutional methods are similar to Finite Impulse Response (FIR) filters and the proposed methods can also be used for FIR applications.

As future work we consider investigating (micro-)architectural techniques to accelerate horizontal filtering. This phase of the 2-D DWT is difficult to vectorize efficiently because the elements within a register need to be rearranged, incurring substantial overhead. Techniques we are considering include providing support for packed multiply-accumulate instructions for floating-point values (MMX/SSE provides such instructions but only for integer data) and the matrix register file (MRF) [23], which is a (micro-)architectural technique to efficiently support matrix transposition.

REFERENCES

[1] M. D. Adams and R. K. Ward, "JasPer: A portable flexible open-source software tool kit for image coding/processing," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, May 2004, vol. 5, pp. 241–244.

[2] S. Chatterjee and C. D. Brooks, "Cache-efficient wavelet lifting in JPEG 2000," in *Proc. IEEE Int. Conf. Multimedia*, Aug. 2002, pp. 797–800.

[3] P. P. Dang and P. M. Chau, "Reduce complexity hardware implementation of discrete wavelet transform for JPEG 2000 standard," in *Proc. IEEE Int. Conf. Multimedia Expo*, Aug. 2002, pp. 321–324.

[4] M. Ferretti and D. Rizzo, "A parallel architecture for the 2-D discrete wavelet transform with integer lifting scheme," *J. VLSI Signal Process.*, vol. 28, pp. 165–185, 2001.

[5] 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 *Proc. 1998 IEEE Int. Conf. Acoustics, Speech, Signal Process.*, Seattle, WA, May 1998, vol. 3, pp. 1521–1524.

[6] A. Cohen, I. Daubechies, and J. C. F. Eauveau, "Biorthogonal bases of compactly supported wavelets," *Commun. Pure Appl. Math.*, vol. 45, no. 5, pp. 485–560, Jun. 1992.

[7] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, vol. 3, no. 2, pp. 186–200, Apr. 1996.

[8] D. M. Adams and F. Kossentini, "Reversible integer-to-integer wavelet transforms for image compression: Performance evaluation and analysis," *IEEE Trans. Image Process.*, vol. 9, pp. 1010–1024, Jun. 2000.

[9] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado, "Vectorization of the 2-D wavelet lifting transform using SIMD extensions," in *Proc. 17th IEEE Int. Symp. Parallel Distributed Image Process. Multimedia*, 2003.

[10] The IA-32 Intel architecture software developer's Manual Volume 3 system programming guide 2004.

[11] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Performance comparison of SIMD implementations of the discrete wavelet transform," in *Proc. 16th IEEE Int. Conf. Applicat.-Specific Syst. Architectures Processors*, Jul. 2005.

[12] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Improving the memory behavior of vertical filtering in the discrete wavelet transform," in *Proc. 3rd ACM Int. Conf. Comput. Frontiers*, May 2007, pp. 253–260.

[13] *IA-32 Intel architecture optimization* 2004.

[14] *Does Hyperthreading Technol. Speed Up VirtualDub*, [Online]. Available: http://www.virtualdub.org/blog/pivot/entry.php?id=18

[15] A. A. Lopez-Estrada, Reduction of address aliasing Aug. 25, 2005, U.S. Patent 20050188172.

[16] P. Meerwald, R. Norcen, and A. Uhl, "Cache issues with JPEG2000 wavelet lifting," *Proc. Visual Commun. Image Process.*, Jan. 2002.

[17] 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. High Performance Comput.*, Dec. 2002.

[18] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Trans. Image Process.*, vol. 9, no. 3, pp. 378–389, Mar. 2000.

[19] H. Komi and A. Ortega, "Analysis of cache efficiency in 2-D wavelet transform," in *Proc. IEEE Int. Conf. Multimedia Expo*, 2001, pp. 465–468.

[20] Y. D. Lee, B. D. Choi, J. K. Cho, and S. J. Ko, "Cache management for wavelet lifting in JPEG 2000 running on DSP," *Electron. Lett.*, vol. 40, no. 6, Mar. 2004.

[21] Y. Andreopoulos, K. Masselos, P. Schelkens, G. Lafruit, and J. Cornelis, "Cache misses and energy-dissipation results for JPEG-2000 filtering," in *Proc. 14th IEEE Int. Conf. Digital Signal Process.*, 2002, pp. 201–209.

[22] R. Kutil, "A single-loop approach to SIMD parallelization of 2-D wavelet lifting," in *Proc. 14th Euromicro Int. Conf. Parallel, Distributed, Netwrok-Based Process.*, Feb. 2007, pp. 413–420.

[23] A. Shahbahrami, B. Juurlink, D. Borodin, and S. Vassiliadis, "Avoiding conversion and rearrangement overhead in SIMD architectures," *Int. J. Parallel Progr.*, vol. 34, no. 3, pp. 237–260, Jun. 2007.

Asadollah Shahbahrami (S'04) received the M.Sc. degree in computer engineering/machine intelligence from Shiraz University, Shiraz, Iran, in 1996. In January 2004, he joined the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS), Delft University of Technology, Delft, The Netherlands, as a full-time Ph.D. student under advisors Prof. Stamatis Vassiliadis and Dr. Ben Juurlink.

He was a Member of Faculty Staff in electrical engineering with the University of Guilan, Guilan, Iran, from 1996 to 2003. His research interests include computer architecture, image and video processing, multimedia instructions set design, and SIMD programming.

Ben Juurlink (M'01–SM'04) received the M.S. degree in computer science from Utrecht University, Utrecht, The Netherlands, in 1992, and the Ph.D. degree in computer science from Leiden University, Leiden, The Netherlands, in 1997.

In 1998, he joined the Faculty of EEMCS, Delft University of Technology, Delft, The Netherlands, where he is currently an Associate Professor. His research interests include instruction-level parallel processors, application-specific ISA extensions, low power techniques, and hierarchical memory systems.

Stamatis Vassiliadis (M'87–SM' 92–F'97) was born in Samos, Greece, in 1951.

He was a Chair Professor in the Faculty of EEMCS, Delft University of Technology, Delft, The Netherlands. He was previously with Cornell University, Ithaca, NY, and the State University of New York, Binghamton. He was also with IBM, where he was involved in a number of advanced research and development projects.

Dr. Vassiliadis received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engineering/scientific hardware design. His 72 U.S. patents rank him as the top all-time IBM inventor. He passed away in April 2007.