Optimizing Cache Performance of the Discrete Wavelet Transform Using a Visualization Tool

Jie Tao[1], Asadollah Shahbahrami[2, 3], Ben Juurlink[2]
Rainer Buchty[4], Wolfgang Karl[4], and Stamatis Vassiliadis[2]

[1]Institut fuer
Wissenschaftliches Rechnen
Forschungszentrum Karlsruhe
jie.tao@iwr.fzk.de

[2]Computer Engineering Laboratory
Delft University of Technology
2628 CD Delft, The Netherlands
A.Shahbahrami@TUDelft.nl

[4]Institut für Technische
Informatik Universität
Karlsruhe (TH) 76128
Karlsruhe, Germany

Abstract

The 2D DWT consists of two 1D DWT in both directions: horizontal filtering processes the rows followed by vertical filtering processes the columns. It is well known that a straightforward implementation of the vertical filtering shows quite different performance with various working set sizes. The only reasonable explanation for this has to be the access behavior of the cache memory. As known, vertical filtering has mapping conflicts in the cache with a working set size that is power of two. However, it is not clear how this conflict forms and whether cache problems exist with other data sizes. Such knowledge is the base for efficient code optimization. In order to acquire this knowledge and to achieve more accurate optimization potentials, we apply a cache visualization tool to examine the runtime cache activities of the vertical implementation. We find that besides mapping conflicts, vertical filtering also shows a large number of capacity misses. More specifically, the visualization tool allows us to detect the parameters related to the strategies. This guarantees the feasibility of the optimization. Our initial experimental results on several different architectures show an up to 215% gain in execution time compared to an already optimized baseline implementation.

Keywords: Discrete wavelet transform, memory performance, visualization tool, code optimization.

1. Introduction

Discrete Wavelet Transform (DWT) tool is one of the most important multimedia algorithm to process multimedia data. Standards such as MPEG-4 and JPEG2000 are based on the 2D DWT. The 2D DWT is much more memory-intensive and time consuming transform than other transforms such as Discrete Cosine Transform (DCT). We have measured the total execution time consumed by the DWT using the JasPer software tool kit [2]. 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 [4, 8] also show that the DWT consumes a significant fraction of the total JPEG2000 encoding time. Therefore, one way to reduce the execution time of the JPEG2000 standard is to optimize the implementation of 2D DWT.

The 2D DWT consists of two 1D DWT in both directions: *horizontal filtering* processes the rows followed by *vertical filtering* processes the columns. As well-known, 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) for the Cohen, Daubechies and Feauveau 9/7 transform (CDF-9/7) on three different platforms: Pentium 3 (P3), Pentium 4 (P4), and AMD Opteron processors.

Figure 1 shows that for some image sizes, vertical filtering is significantly slower than horizontal filtering, while for other image sizes there is a slight difference. This scenario has been explained by the mapping conflicts in the cache memory [8, 12]. This explanation, however, is merely based on a static analysis of the source code without any runtime information. This analysis can neither exhibit the concrete conflict nor cover all cache problems. For example, it is not known how the conflict is caused and whether other issues, e.g. capacity problem, exist which are also responsible for

[3]This research was supported in part by the Netherlands Organisation for Scientific Research (NWO). Asadollah Shahbahrami is also with the Department of Electrical Engineering, Faculty of Engineering, The University of Guilan, Rasht, Iran.

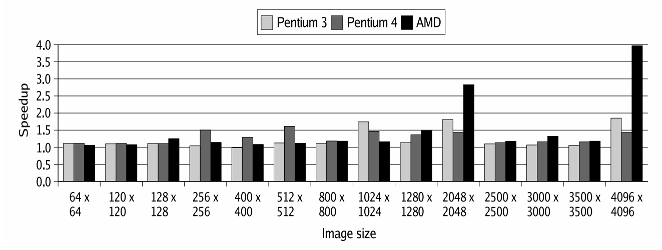


Figure 1. Speedup of horizontal filtering over vertical filtering for CDF-9/7 transform on three different platforms, P3, P4, and AMD Opteron processors, for various image sizes.

the cache inefficiency of the vertical filtering.

In order to exactly understand the access pattern and the runtime cache behavior of the DWT implementation and to detect potential further optimizations, we apply the visualization tool YACO [9]. YACO is specifically designed and implemented for understanding and optimizing the cache access behavior of sequential as well as parallel applications. Using this tool, we examined the runtime data operations of the DWT and found that besides the well-known cache conflict capacity miss is also a primary reason for the poor performance of vertical filtering. Moreover, we detected the cause for the conflict misses.

Based on this finding, we first created an optimized version using loop tiling, an efficient strategy for tackling capacity miss. This optimization scheme enables reused data to be maintained in the cache by reducing the size of the working set in a single loop. Again based on the presented access pattern, we then achieved two optimized versions for tackling conflict misses, one using loop fission and the other with array padding. Loop fission removes the conflict misses by distributing the overmapping data into several individual loops, while array padding changes the mapping behavior of data structures by inserting buffers between them or inside a single data array.

Our optimizations differ from existing approaches in that the deployed schemes and their related parameters such as tiling and padding size are selected according to the access pattern and the runtime cache behavior of the program. Hence, they exactly address the problem and are therefore more efficient.

The optimizations were tested on several target architectures, including P3, P4, AMD Opteron, and Intel Itanium II. In most cases we achieved a performance gain of more than a factor of two. The best performance is obtained on the Opteron processor, where an improvement of a factor of 3.15 in execution time has been attained.

The remainder of the paper is organized as follows. Section 2 provides a brief explanation of the DWT and the CDF-9/7 implementation of the vertical filtering. This is followed by a description of some related work in the area of DWT optimization in Section 3. In Section 4 the visualization tool is introduced, followed by a detailed description of how this tool is applied to find the bottlenecks, the reason for them and the strategies for tackling the problems in Section 5. Initial experimental results are presented in Section 6. The paper concludes in Section 7 with a short summary.

2. Background

In this section we describe the DWT and the implementation of vertical filtering using loop interchange.

2.1. Discrete Wavelet Transform (DWT)

As mentioned, several image standards use 2D DWT to compress image data. The DWT transforms the wavelet by sampling it discretely. With this algorithm, the wavelet representation of a discrete signal X consisting of N samples is computed by convolving X with the low-pass and high-pass filters and down-sampling the output signal by a factor of 2. This process decomposes the original image into two sub-bands: the lower and the higher band [13], each containing $N/2$ samples.

A 2D DWT is performed by first performing an 1D DWT on each row (*horizontal filtering*) of the image followed by an 1D DWT on each column (*vertical filtering*). The horizontal filtering filters whole rows of an image and stores the intermediate coefficients in an output matrix. Then, the vertical filtering filters these intermediate results and stores the final results in the input matrix in the order expected by the quantization step.

2.2. Vertical Filtering

Vertical filtering processes the matrix in column order. The original implementation of vertical filtering computes each column entirely before advancing to the next column. This approach, 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*, a well-known compiler technique. Figure 2 depicts the C implementation of vertical filtering using the CDF-9/7 transform for an $N \times M$ image with loop interchange.

In this work we consider the CDF-9/7 transform, because this filter is included in Part 1 of the JPEG2000 standard [10]. However, the proposed techniques can be applied to other filters as well, as an example Daubechies' transform with four coefficients. The CDF-9/7 transform has 9 low-pass filter coefficients and 7 high-pass filter coefficients, both filters are symmetric. Array *low* and *high* in

```
void CDF_97_vertical() {
int i, j, ii;
float low[] ={0.6029, 0.2669, -0.07822, -0.0169, 0.0267};
float high[]={-0.5913, -0.0575, 0.09127, 1.1150};
for (i=0, ii=4; ii<N-4; i++, ii +=2)
  for(j=0; j<M; j++) {
    in_image[i][j]  = ou_image[ii-4][j] * low[4] + ou_image[ii-3][j] * low[3]
                    + ou_image[ii-2][j] * low[2] + ou_image[ii-1][j] * low[1]
                    + ou_image[ii][j]   * low[0] + ou_image[ii+1][j] * low[1]
                    + ou_image[ii+2][j] * low[2] + ou_image[ii+3][j] * low[3]
                    + ou_image[ii+4][j] * low[4];
    in_image[i+ N/2][j]  = ou_image[ii-4][j] * high[2] + ou_image[ii-3][j] * high[1]
                         + ou_image[ii-2][j] * high[0] + ou_image[ii-1][j] * high[3]
                         + ou_image[ii][j]   * high[0] + ou_image[ii+1][j] * high[1]
                         + ou_image[ii+2][j] * high[2];
  }
```

Figure 2. C implementation of vertical filtering using the CDF-9/7 transform. Note that the loops have been interchanged w.r.t. the straightforward implementation.

the code shown in Figure 2 store these values. The experimental results that have been presented in [11, 12] clearly show that the implementations with interchanged loops are much more efficient than the straightforward implementations. For this reason we will compare the performance of our improved implementations to the performance attained by the algorithms after loop interchange.

As illustrated in Figure 1, however, performance problem still exists with the vertical filtering. This motivated us to apply the visualization tool to acquire a deeper insight into the runtime cache access behavior of this code.

3. Related Work

Meerwald et al. [8] have proposed two techniques to improve cache utilization, called row extension and aggregation. 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. 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.

Chaver et al. [5] also considered the memory hierarchy issue and made a further step towards understanding the cache problem exactly. They detected that the main problem of the DWT is caused by the discrepancies between the memory access pattern of horizontal and vertical filtering. The main bottleneck of the DWT is caused by the vertical filtering. They proposed combining aggregation with a line-based approach [6], 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. They chose images with a width equal to a power of two and measured performance on a P4 (as well as a P3). The experimental results show, however, that this approach does not eliminate all cache conflict misses.

Adams [1] also observed that processing in the vertical direction can be extremely inefficient, due to the large stride accesses in memory. Adams noted that many misses occur during vertical filtering when the image width is power of two. He proposed two optimization techniques, called modified split/join (MSJ) and pipelined filtering (PF). The MSJ method affects the way in which split/join operations are performed in the lifting-based implementation. It tries to reduce the number of read and write operations compared to the traditional method. In the PF technique the computation for all lifting operations are simultaneously performed as soon as the necessary data dependencies are satisfied.

In summary, existing research work has targeted on the cache problem of the DWT (primarily the conflict misses), achieved statistics on cache misses, and tried different optimizations with the goal of improving the data locality. However, these researchers could not completely solve the cache inefficiency with DWT. This is because of the following reasons. First, the cache miss statistics only give an overview of the global cache performance and hence do not expose the actual problem. Second, optimizations are based on assumptions or static analysis without any knowledge of the runtime cache behavior, and hence cannot fully remove conflict misses. Finally, optimization strategies are not specifically chosen. For achieving optimal optimization, however, suitable techniques have to be appropriately deployed because each scheme is only efficient for a certain kind of cache misses.

Therefore, we perform optimizations based on the knowledge about cache miss reason and the runtime access feature. Our work differs from previous work in the following ways. First, we detect all possible cache misses. Second, we acquire the knowledge of cache miss characteristics like the reason for each kind of miss. Third, we locate the concrete object to optimize. Finally, the optimization scheme is specifically selected for tackling the detected miss type.

4. YACO: A Cache Visualization Tool

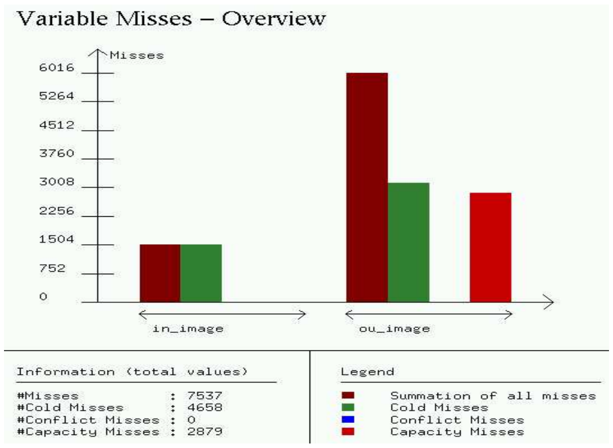
YACO [9] is developed for exhibiting the runtime cache accesses. It is specially designed with the goal of efficiently helping the users in their task of cache optimization. It provides a variety of graphical views to display the various aspects of the cache access behavior. The presentation is at a high level with respect to user-visible data structures. More specifically, YACO shows the cache miss reason, which allows the user to detect optimization strategies.

Following the conventional optimization process, YACO uses a top-down approach to direct the user step-by-step to detect the problem and the solution. Users first acquire an overview about the cache access behavior shown by the observed program. Based on this overview, users can determine whether an optimization is essential. In the next step, the access hot spots, i.e., the data structures and code regions which are responsible for the poor cache performance, can be located with the help of both YACO's view about the miss statistics on variables and its three dimensional view showing the miss behavior of individual functions. After that, the reason for the cache miss and the access pattern of data structures can be detected. This information also allows the user to decide an appropriate optimization scheme and related parameters, or to design novel algorithms to eliminate the detected cache problem. The impact of the optimization can be observed with YACO after running the optimized code. This process can be repeated until an acceptable cache performance is achieved. Some views and this step-by-step analysis process will be shown and explained in the following section by examining the cache behavior of the DWT algorithm.

5. Analyzing the Cache Problem

In order to remove the effects of conflict misses for detecting other cache problems, we applied a data set size, randomly chosen as 160×160, which is not a power of 2. The target cache is a 4-way set associative one with a line size of 64 bytes and a total capacity of 4KB.

For detecting the cache problems with the vertical filtering, we fist examined YACO's Variable Miss Overview, which gives an overview of cache misses with the main data structures of a program or a function. As illustrated in Figure 3, this view displays the miss statistics on each data structure in four bars. They represent from left to right the total misses, cold (compulsory) misses, conflict misses, and capacity misses, where the total misses is the sum of the other three classifications. Absolute numbers of the misses are depicted at the left bottom corner, while the meaning of each column is explained at the right bottom corner.



```
Information (total values)
#Misses            : 7537
#Cold Misses       : 4658
#Conflict Misses   : 0
#Capacity Misses   : 2879
```

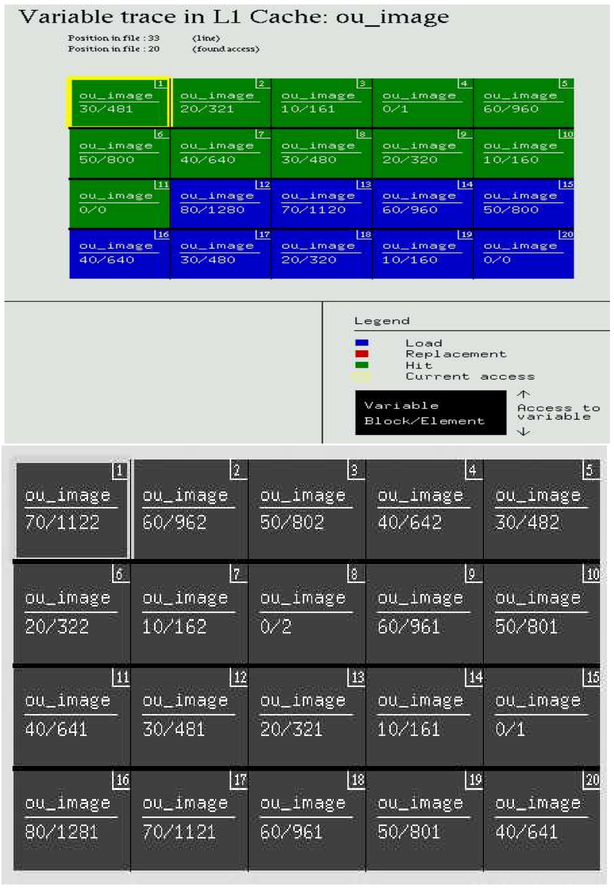
Legend	
🟥	Summation of all misses
🟩	Cold Misses
🟦	Conflict Misses
🟥	Capacity Misses

Figure 3. Variable Miss Overview: 4KB-cache.

The figure clearly show that for the two primary data structures in the vertical filtering, `ou_image` introduces a significant number of cache misses, 80% more than `in_image`. In addition, it can be seen that all misses with `in_image` are compulsory misses caused by the first references. This kind of miss can be reduced using prefetching.

Having found the optimization solution for `in_image`, we now focus on the matrix `ou_image`. For this data structure, in addition to compulsory misses, the limited cache capacity is another primary reason for the misses. For tackling capacity miss, a common approach is loop tiling. This technique reduces cache misses by decreasing the number of iterations within the innermost loop so that the data can be reused by the outer loops. For this, however, we need to know the tiling size, i.e., the number of iterations in the innermost loop. This parameter decides the data size within a loop and the data size determines whether the execution introduces capacity misses.

For acquiring this parameter we further examine YACO's Variable Trace and Cache Set view, which present the access pattern and the runtime cache updates individually. The Variable Trace view of YACO shows the references to data blocks/elements in a data structure in the order as they are accessed. As depicted in the sample view in the upper diagram of Figure 4, 20 fields are used to demonstrate the successive references to the selected data structure. These fields are filled from left to right, top to bottom as the references going on. In case that all fields are filled, the earliest access, i.e. the reference in field 20, is removed.



Variable trace in L1 Cache: ou_image

Position in file : 33 (line)
Position in file : 20 (found access)

1 ou_image 30/481	2 ou_image 20/321	3 ou_image 10/161	4 ou_image 0/1	5 ou_image 60/960
6 ou_image 50/800	7 ou_image 40/640	8 ou_image 30/480	9 ou_image 20/320	10 ou_image 10/160
11 ou_image 0/0	12 ou_image 80/1280	13 ou_image 70/1120	14 ou_image 60/960	15 ou_image 50/800
16 ou_image 40/640	17 ou_image 30/480	18 ou_image 20/320	19 ou_image 10/160	20 ou_image 0/0

Legend

🟦 Load
🟥 Replacement
🟩 Hit
Current access

Variable Block/Element — Access to variable

1 ou_image 70/1122	2 ou_image 60/962	3 ou_image 50/802	4 ou_image 40/642	5 ou_image 30/482
6 ou_image 20/322	7 ou_image 10/162	8 ou_image 0/2	9 ou_image 60/961	10 ou_image 50/801
11 ou_image 40/641	12 ou_image 30/481	13 ou_image 20/321	14 ou_image 10/161	15 ou_image 0/1
16 ou_image 80/1281	17 ou_image 70/1121	18 ou_image 60/961	19 ou_image 50/801	20 ou_image 40/641

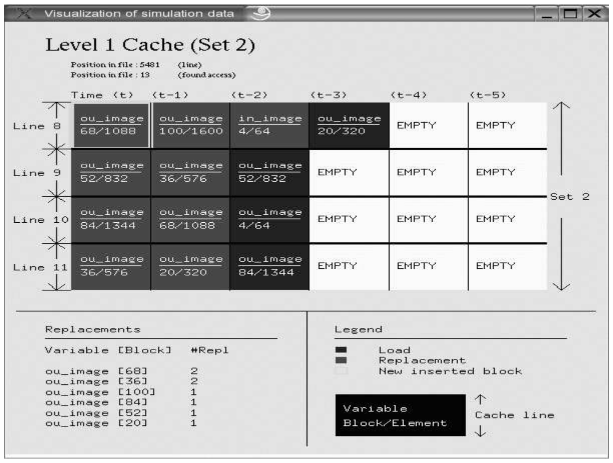
Figure 4. Variable Trace with hit turned-on (upper: first 20 references, lower: the continuous 20 references).

Figure 4 depicts the initial 40 accesses to matrix `ou_image` in two views. Each field in the diagram contains complete information about the corresponding reference: the access type (indicated by the color), name of the referenced data structure (upper text), the concrete data element (lower right number), and the data block (lower left number) holding this element. The access type can be a load operation indicating a first reference, a replacement indicating the access to this matrix element replacing another data block in the cache, or a hit event.

First, the accesses show a regular pattern, where nine data blocks are loaded into the cache, seven of them reused, and then fully reused one after another. For example, the first access, illustrated in field 20 of the upper diagram of Figure 4, is a reference to element 0 stored in data block 0. The color indicates it is a load operation. After another eight loads of different data blocks, block 0 is re-accessed for the same element. According to the color of field 11, this access is a cache hit. Observing field 4 it can be seen that block 0 is referred again but this time the access is to element 1.

Examining further references we see that this behavior repeats till the last elements in these nine blocks, i.e., 0, 10, ..., 80, are referenced with a cache hit (unfortunately, we could not show all accesses in a single picture). After that, nine successive blocks, i.e., 1, 11, 21, ..., 81, are loaded into the cache and reused like the previous blocks. This behavior repeats till blocks 9, 19, 29, ..., 89 are loaded into the cache. We call all of these an access phase. In the following, block 20, 30, ..., 100 are accessed, similarly to the blocks 0, 10, ..., 80, then 21, 31, ..., 101, and so on, forming the second access phase.

This behavior shows us two kind of data locality: intra-phase and inter-phase. The former exists within an individual access phase, where the same data block is reused. The latter exists across phases, like block 20 is accessed in the first phase and then in the second phase again. According to the YACO views, the intra-phase data reuse all introduces a cache hit, however, the inter-phase reuse all shows cache miss. YACO shows the reason: after loading twenty data blocks the cache is full and the inter-phase reused block is evicted from the cache before it could be reused.



Visualization of simulation data

Level 1 Cache (Set 2)

Position in file : 5481 (line)
Position in file : 13 (found access)

	Time (t)	(t-1)	(t-2)	(t-3)	(t-4)	(t-5)
Line 8	ou_image 68/1088	ou_image 100/1600	in_image 4/64	ou_image 20/320	EMPTY	EMPTY
Line 9	ou_image 52/832	ou_image 36/576	ou_image 52/832	EMPTY	EMPTY	EMPTY
Line 10	ou_image 84/1344	ou_image 68/1088	ou_image 4/64	EMPTY	EMPTY	EMPTY
Line 11	ou_image 36/576	ou_image 20/320	ou_image 84/1344	EMPTY	EMPTY	EMPTY

Set 2

Replacements

Variable [Block]	#Repl
ou_image [68]	2
ou_image [36]	2
ou_image [100]	1
ou_image [84]	1
ou_image [52]	1
ou_image [20]	1

Legend

■ Load
■ Replacement
□ New inserted block

Variable Block/Element — Cache line

Figure 5. The Cache Set view.

For holding the blocks in the cache for inter-phase reuse, it needs to know which data block is responsible for the eviction. For this, we examine the cache set view of YACO, shown in Figure 5. The cache set view is designed to exhibit the runtime activities in a single cache set, in this case set 2 where the first reused block, i.e. block 20, is mapped. As shown in the figure, a cache set contains four lines of horizontal blocks which correspond to the four cache lines in a set. These blocks demonstrate the operations and content update in an individual cache set. The operations are presented in chronological order with the right followed by the left. Using the arrow keys, further or earlier operations can be presented or represented.

Observing the first line it can be seen that block 20

Cache size	1K	2K	4K	8K	16K	32K	64K
Tiling size	16	32	64	128	256	512	1024

Table 1. Tiling size for different cache size.

is replaced by block 4 while accessing element 64 of `in_image`. As the source code in Figure 2 shows, `in_image` is computed element by element. This means that if the inner loop only processes 64 elements of `in_image`, from element 0 to element 63, and then the outer loop starts, the access to element 64 would not be issued and block 20 would be maintained in the cache. Therefore, the number of iterations in the innermost loop, here loop `j`, has to be less than or equal to 64 for the goal of data reuse. We then added another loop for maintaining the program semantic and achieved an optimized version. This optimization technique is called tiling.

Similarly, we examined the case with 2KB caches and detected a tiling size of 32 for this configuration. Based on these results with both 4KB and 2KB caches, we assume that the tiling size can be calculated for other cache sizes with the same associativity. Actually, this is also theoretically correct. For example, a 4KB cache has a double capacity as a 2KB cache, can hold twice as much data, and hence the innermost loop is allowed to contain twice as many iterations. Table 1 gives this information for several cache capacities.

Having detected the cache problem with image size which is not a power of two, we further examined the feature of conflict misses with power two image sizes. For this, we performed a simulation with an image size of 512×512. First, we found that in this case even the intra-phase reuse causes cache miss. Using the cache set view we further detected that the reason for this poor behavior lies in the fact that all nine `ou_image` blocks needed for processing a single element of `in_image` are mapped in the same cache set. This means that if a cache set cannot hold all these blocks, e.g. in case of a 2-, 4-, and 8-way cache with LRU replacement policy, each of the blocks has to be moved from the cache before reuse. Therefore, all accesses to them are cache misses.

The solution is either to perform only partial calculations in the innermost loop so that fewer data blocks are requested or to insert at least a cache line between the rows of matrix `ou_image` to map these nine blocks in different cache sets. The former technique is called loop fission and the later array padding. Again, we achieved two optimized versions of the DWT program using theses schemes.

6. Experimental Results

The optimized versions were evaluated on several different architectures. For a comparative study the original code was also tested.

6.1. Experimental Setup

Four platforms were used for executing the resulted codes: P3, P4, AMD Opteron, and Itanium II. The main architectural parameters of these systems are summarized in Table 2.

All versions were compiled using gcc with optimization level *-O2* and executed on a lightly loaded system. Performance was measured using the IA-32 cycle counter [7]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program. 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 [3]. That means that 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.

6.2. Performance

We first examine the implementation of the vertical filtering using the fission technique which is used to tackle conflict miss by working set size of a power of two. Figure 6 depicts the speedup resulting from this technique. The speedup is calculated by dividing the execution time of the reference implementation by the execution time of the optimized version.

It can be seen that for those image sizes that suffer from many cache misses, loop fission improves performance significantly. For example, the speedup ranges from 1.01 to 1.29 for P3, from 1.09 to 1.24 for P4, and from 1.13 to 2.38 for the AMD Opteron processors. However, only a slight performance improvement with a speedup of up to 1.1 has been gained on the Itanium architectures. Even a slowdown can be observed with smaller and no power of two image sizes. The slowdown accounts for up to 13%, 4%, 17%, and 12% on P3, P4, AMD Opteron, and the Itanium processors, respectively.

The reason that why the loop fission reduces the performance for some image sizes is that this technique partitions a single loop into several loops. This means that the processor has to do additional work for managing these loops and their related index variables. More importantly, the loops introduce a number of conditional instructions. Depending on the processor architecture the overhead for managing theses loops can vary significantly. For example, processors with better branch predictor requires less time for handling the loops than those that are poorer in this feature. Hence, the overhead could be larger than the gain through reduced cache misses, causing thereby slowdown rather than speedup. However, for large power of two image sizes, which suffer from considerable cache misses, the

Processor	Intel P3	AMD Opteron	Intel P4	Intel Itanium II
CPU Clock Speed	451MHz	2.0GHz	3.0GHz	1.3GHz
L1 Data Cache	16 KBytes 2-way set asso., 32 Bytes line size	64 KBytes 2-way set asso., 64 Bytes line size	8 KBytes 4-way set asso., 64 Bytes line size	16 KBytes 4-way set asso., 64 Bytes line size
L2 Cache	512 KBytes 8-way set asso., 32 Bytes line size	1 MBytes 8-way set asso., 32 Bytes line size	512 KBytes 8-way set asso., 64 Bytes line size	256 KBytes 8-way set asso., 128 Bytes line size
Memory	384 MBytes	1 GBytes	1 GBytes	8 GBytes

Table 2. Parameters of the experimental platforms.

performance gain through cache optimization is significant and therefore speedup has been achieved.

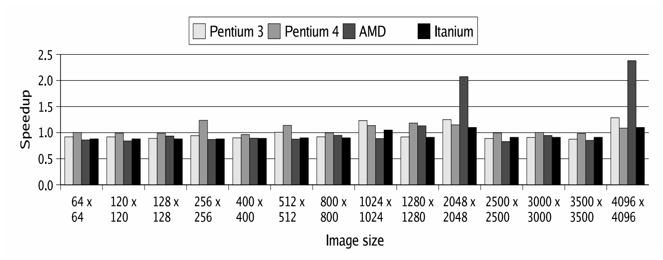


Figure 6. Speedup by using loop fission.

For the padding scheme we do not see such performance penalty. This is because the padding technique relies on enlarging data structures to change the mapping behavior and thereby eliminate conflict misses. Hence, this optimization scheme does not result in any runtime execution overhead. Figure 7 depicts the speedup achieved by this technique.

Again, it can be seen that considerable improvements are achieved with power of 2 image sizes. The padding technique improves performance by factors ranging from 1.01 to 1.68 for P3, from 1.01 to 1.27 for P4, from 1.01 to 3.19 for AMD Opteron, and from 1.05 to 1.34 for the Itanium processors. This result is better than loop fission.

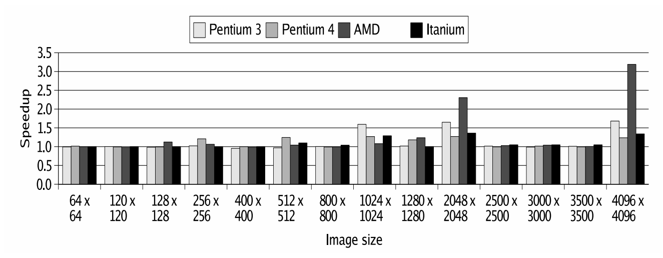


Figure 7. Speedup achieved by the padding technique.

The high performance improvement obtained by both optimized versions, loop fission and padding, on P3, P4, and the AMD Opteron processors compared to Itanium processor. The reason is that the applied Itanium processor has an L3 cache of 3MB with a latency of 15 cycles. This cache can hold most of the data evicted from the L1/L2 caches. Therefore, the programs do not suffer as significantly as other machines from the long access latency of the main memory.

Figure 8 depicts the speedup of the horizontal filtering over vertical filtering with padding technique. It can be observed the vertical filtering presents a good performances, much better than its behavior with the original code depicted in Figure 1.

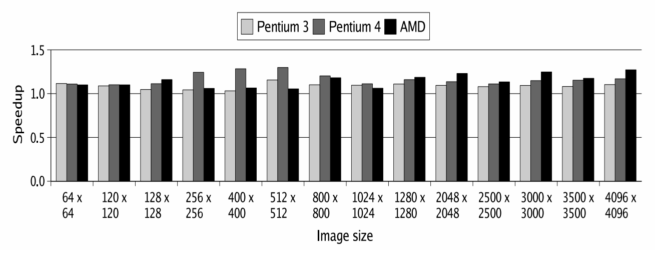


Figure 8. Speedup of horizontal filtering over vertical filtering with padding technique.

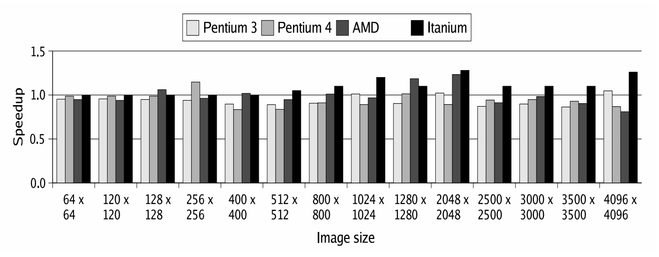


Figure 9. Speedup of the vertical filtering using tiling technique over the reference implementation.

Finally, we examine the tiling technique. Figure 9 depicts the speedup achieved by this scheme in the implementation of vertical filtering over the reference implementation. Observing the overall performance gain across the image sizes, it can be seen that this technique does not yield speedups as high as the fission and padding schemes do. This can be expected because the latter two techniques are used to tackle conflict misses, and there are several orders of magnitude more conflict misses than capacity misses. For example, our simulation with an image size of 512×512 showed 120,960 conflict misses and 5020 capacity misses (24-fold).

Despite the relatively small number of capacity misses, we have achieved performance gains. As can be seen in Figure 9, the best performance is obtained on the Itanium and AMD Opteron, where a speedup of close to 1.3 has been achieved. P4 has shown an up to 1.15 speedup, while on P3 the maximal improvement counts for 8%.

Our tiling algorithm is similar to the aggregation technique. The reason that why this technique improve performance for some image sizes is as follows. The rows of input image data needed to compute one row of output image data, all except the first two can be reused to compute the next row of output data. In particular, if the rows of input image needed to compute row i of output data are the rows j to $j+L-1$, where L is the filter length, then rows $j+2$ to $j+L-1$ are reused to compute the next row $i+1$, provided L rows can be kept in cache.

On the other hand, the reason that why this technique reduce the performance of some image sizes is as follows. First, this algorithm does not improve cache conflict misses. Second, this algorithm incurs more loop overhead than reference implementation. Finally, it destroy spatial locality because it does not entirely processes the elements of rows consecutively.

7. Conclusions

In this paper, we show an approach of using a visualization tool to analyze and optimize the cache performance of the DWT algorithm. We actually have addressed two kind of cache misses: the capacity and the conflict miss. For capacity misses, the proposed approach is general with respect to image size and target systems. This is because capacity misses are caused by the limited cache size. Hence the miss behavior only depends on the cache size. For conflict misses we could find general solution for various image sizes and architectures based on one analysis like we have done in this paper. This solution, however, may not be ideal for all scenarios. This is because the runtime conflict misses depend on the cache associativity, the data size, and also on the compiler. The former two factors determine the mapping of data in the cache and thereby influence the number of conflict misses. For the latter, modern compilers perform code transformations that modify the data accesses. Hence, for an accurate optimization it is better to observe the cache behavior with individual data size and cache architecture.

Overall, in contrast to conventional approaches based on static analysis of the source code, our approach is better in terms of accuracy because it considers the runtime dynamic behavior of the cache. Hence, a higher performance gain can be achieved. Although with some optimization techniques it is needed to deal with individual cases, such as different cache organization and data size, for optimal solutions, however, approaches based on static analysis also have to handle these cases separately.

References

[1] M. D. Adams. Efficient Breadth-First Implementation of the Wavelet Transform. In *Proc. IEEE Int. Symp. on Signal Processing and Information Technology*, August 2006.

[2] 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. on Acoustics, Speech, and Signal Processing*, volume 5, pages 241–244, May 2004.

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

[4] 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.

[5] 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*, Dec. 2002.

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

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

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

[9] B. Quaing, J. Tao, and W. Karl. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization. In *Proc. 1st Int. Conf. on High Performance Computing and Communcations*, volume 3726 of Lecture Notes in Computer Science, pages 694–703, September 2005.

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

[11] 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.

[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. on Computing Frontiers*, pages 253–260, May 2006.

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