Data Locality Optimization based on Comprehensive Knowledge of the Cache Miss Reason: A Case Study with DWT

Jie Tao
[1]Department of Computer Science and Technology
Jilin University, P.R China
[2]Steinbuch Centre for Computing, Forschungszentrum Karlsruhe
Karlsruhe Institute of Technology, Germany
jie.tao@iwr.fzk.de

Asadollah Shahbahrami
Computer Engineering Laboratory
Delft University of Technology
2628 CD Delft, The Netherlands
shahbahrami@ce.et.tudelft.nl

Abstract

The overall performance of a computing system increasingly depends on the efficient use of the cache memories. Traditional approaches for cache tuning deploy performance tools to help the user optimize the source program towards a better runtime data locality. Following this conventional way, we developed a set of such toolkits including data profiling, pattern analysis, and performance visualization tools.

This paper demonstrates how the toolset can be used step-by-step to understand the cache access behavior of the applications and then achieve optimized program code. The Discrete Wavelet Transform, a common used algorithm for image and video compression, is applied as an example. Our initial experimental results with this sample application show an up to 3.19 speedup in execution time compared to the original implementation.

Keywords: Memory performance, program optimization, performance tools, data profiling, performance analysis and visualization

1 Introduction

As the gap between the processor and memory speed is growing, cache performance becomes increasingly important for hiding the memory access latency of modern processors. However, due to the complexity of both the memory hierarchy and the applications, still excessive cache misses have been measured. These misses cause significant performance degradation because an access to the main memory takes hundreds of cycles while an access in the cache needs only several ones. According to [9], the SPEC2000 benchmarks running on a modern, high-performance microprocessor spend over half of the time stalling for loads that miss in the last level cache.

A conventional way to improve cache locality is to provide the users with a performance analyzer or a visualization tool that help them detect the critical regions to optimize.

Following the traditional approach, we developed a cache visualizer to depict the runtime access behavior of the cache and the application. This visualization tool relies on a data profiler, a pattern analyzer, and a cache simulator for source data. In addition, a program development environment was implemented for users to operate the tools. Together, we provide a complete toolset that covers the whole feedback loop of code optimization, from data acquisition to pattern analysis, problem location, solution exploring, up to the optimization of the source program.

In comparison with existing systems, which are only capable of showing the bottlenecks, our toolset goes a step further: it presents the reason of the bottlenecks and thereby help the user determine an adequate scheme to solve the problems.

This paper focuses on demonstrating how the toolset helps programmers step-by-step overcome the cache problem of their applications. We use the Discrete Wavelet Transform (DWT) as an example to depict this process.

The DWT algorithm is applied by standards, such as MPEG-4 and JPEG2000, for image and video compression. The reason to choose this application is its simple code structure and wide use in both multimedia and embedded areas. Using the toolset we found that this application suffers from considerable conflict misses. Additionally, capacity misses and compulsory misses are also observed with this program. Guided by the tools, we applied several cache optimization techniques to individually tackle differ-

[1]This work was conducted as Dr. Tao worked at the Institut für Technische Informatik, Universität Karlsruhe.

978-0-7695-3352-0/08 $25.00 © 2008 IEEE
DOI 10.1109/HPCC.2008.7

ent cache misses. Initial experimental results show a high achievement with the optimization.

The remainder of the paper is organized as follows. Section 2 presents several related work in the research area of performance analysis tools. Section 3 gives a brief introduction of the developed toolset, followed by a detailed description in Section 4 of how it can be used to find and correct the cache problems of the DWT algorithm. Experimental results are depicted in Section 5. The paper concludes in Section 6 with a short summary.

2 Related Work

Over the last years, performance tools have been applied to support users in the task of code analysis and optimization. In the specific research area of cache locality optimization, a few analysis systems have been developed.

The Intel VTune Performance Analyzer [8] is regarded as a useful tool for performance analysis. It provides several views, like “Sampling” and “Call Graph”, to help programmers identify bottlenecks. For cache metrics it shows the absolute number of cache misses with respect to the code region allowing the user to detect functions and even code lines that introduce most cache misses.

The Tuning and Analysis Utilities (TAU) [11] is a profiling and tracing toolkit for performance analysis. It consists of a visualization component providing graphical displays of analysis results. This allows the user to identify the source of performance bottlenecks in the programs.

Vampir [3] is another well-known toolkit for performance visualization. It was originally developed to show the communication issues of MPI applications, but extended to depict the cache performance. It can show the number of cache misses in a time-line view as well as in the source code.

KCachegrind [14] is a visualization tool for presenting the profile data gathered by a profiling tool based on the Valgrind [5] framework for simulation. The focus of this tool is on individual functions. As a result, it shows the function call graph, the time needed for executing each function, and the number of cache misses introduced by the functions.

Overall, existing tools hold the feature of helping the programmers understand the overall cache performance and further find the bottlenecks. However, they do not offer hints about how to optimize the code. Our toolset compensates for this with its comprehensive data acquisition system and specific design of the visualization component.

3 Performance Tools for Cache Optimization

The developed toolset aims at efficiently supporting the user in locality optimization with detailed information

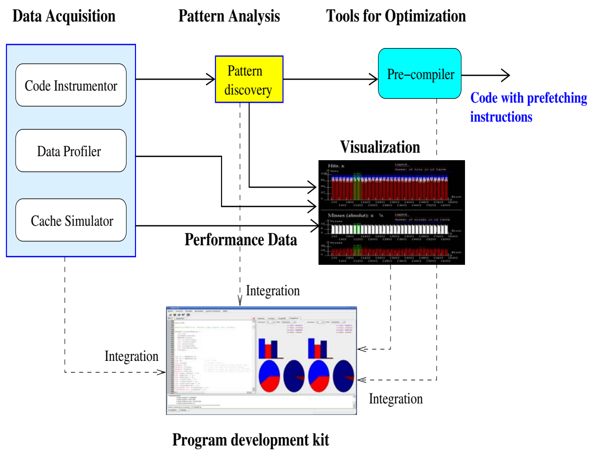


Figure 1. The infrastructure of the provided tools and their connections for cache performance analysis.

about the access feature, cache behavior, miss reason, and optimization objects. The following tools are provided:

- A visualization tool that displays the performance data in a user-understandable way.
- A profiling tool that deploys hardware counters to collect information about global cache events like hit/miss and memory reference.
- A pattern discovery system that detects repeated address sequence and access stride.
- A pre-compiler that automatically inserts prefetching instructions in the source program. The prefetching is guided by the access pattern.
- A cache simulator that models the runtime cache activities and analyzes the character of cache misses. It delivers a cache operation histogram and an application access histogram that allow the visualization tool to present the access property and the miss reason.
- A program development kit that establishes a platform for application design.

Figure 1 shows the infrastructure of the tool environment. Core of the toolset is the program development kit that not only provides a platform for users to modify, build, and run their applications, but also integrates all tools into a single interface. From there, programmers can start the code instrumentor, the data profiler, the simulator, and the discovery system to acquire performance data. After that the visualization tool can be called to show the runtime activities of both the cache and the application. In case of high compulsory misses, the pre-compiler could be used to perform prefeching.

The data profiler is developed to collect the cache performance data using hardware counters. Actually, existing counter interfaces such as PAPI [4] and *pfmon* [7] can be used for this purpose, however, they only allow the programmers to find critical code regions but not data structures. Our profiler [12] shows the access hotspots in terms of both the code and the variable.

The performance data acquired by the profiler show the access bottlenecks. For tracking the cache miss reason, however, more detailed information about the runtime cache and application access behavior is needed. We gather such information with a cache simulator. This component models the functionality of a complete cache hierarchy on a shared memory multiprocessor system. It also contains mechanisms for analyzing the feature of the cache misses, tracing the cache event, performing data prefetching, and detecting inefficiency of cache coherence protocols.

In addition, a pattern discovery system [13] is developed for detecting simple patterns like repeated access sequence and access stride. The former shows which memory addresses are frequently accessed together, helping users group them into a single cache block. The latter depicts which element in a data array is the next access target. This information is often used to guide the prefetching [2, 6]. However, it is a tedious work for the user to insert prefetching instructions in the source code, even with help of a visualization tool. Therefore, we developed a pre-compiler to give the user an additional help: it automatically extends the source program with prefetching.

Performance data from the pattern discovery system, the data profiler, and the cache simulator are delivered, as depicted in Figure 1, to the visualization tool [10]. This component is specifically designed to present the acquired performance data in a way that bottlenecks, miss reason, and optimization schemes can be observed. Some sample views will be depicted in the following section together with the presentation of the DWT runtime access behavior.

The program development kit provides a graphical interface for code development. This interface contains mainly two windows and a menu bar. The first window is a code editor where the users can modify, compile, and execute their applications. The other window is a display area where the execution time is visualized. This allows the user to compare the performance of different runs. The menu bar includes items for user to start the tools and to specify the parameters that control their behavior.

4 DWT Cache Performance Analysis

In this section, we demonstrate how the developed tools can be applied to understand the cache performance and the access behavior, to find the access hotspots and the cause for that, and finally to conclude an appropriate optimization scheme.

4.1 Discrete Wavelet Transform

The Discrete Wavelet Transform performs wavelet transformation by sampling the wavelet discretely. In a DWT algorithm, the wavelet representation of a discrete signal X consisting of N samples can be computed by convolving X with the low-pass and high-pass filters and down-sampling the output signal by 2. This process decomposes the original image into two sub-bands: the lower and the higher band. The transform can be extended to multiple dimensions by using separable filters.

A two-dimensional DWT is conducted 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 most straightforward implementation of the 2D DWT is *Row-Column Wavelet Transform (RCWT)* [1]. First, it performs horizontal filtering and stores the intermediate coefficients in an output matrix. Then, it performs vertical filtering with these intermediate coefficients and stores the final results in the input matrix in the order expected by the quantization step.

The main data structures in a DWT implementation are the input and output image array. Figure 2 gives a code segment in the vertical filtering procedure. This segment produces the second half of the final results which is, as mentioned, stored in the input image *in_image* with N rows and M columns. The processing follows a way that each row is entirely computed before advancing to the next row. The computation handles a summarization of seven multiplication operations.

4.2 The Results of the Visualization Tool

To examine the runtime cache performance of DWT, we simulated its execution with the cache architecture of an existing Intel processor as a sample configuration. This processor has a two level cache hierarchy with a 4-way L1 cache and an 8-way L2 cache. Both caches use a line size of 64Bytes. We use a small image size of 64*64 to avoid producing very large performance data and correspondingly reduce the cache size for matching to the image size.

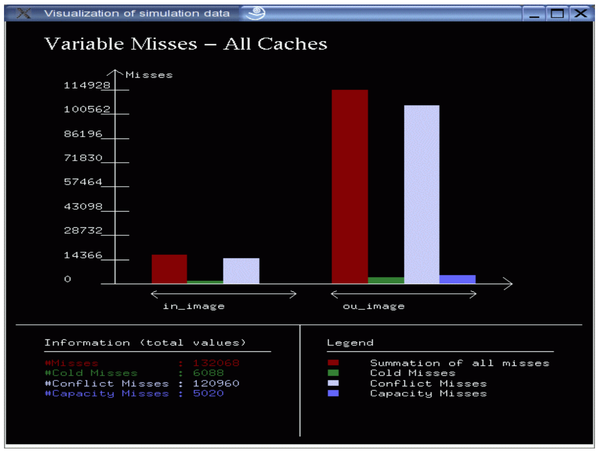
Overall Performance. Now we start the process of analyzing and tuning the cache access behavior of the given DWT implementation. The first step is to observe the overall runtime cache performance in order to know whether

```
for (i=0, ii=4; ii<N-4; i++, ii +=2)
  for(j=0; j<M; j++) {
     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. The original implementation of a code segment in the vertical filtering.

an optimization is needed. The Performance Overview diagram of the visualization tool shows us: while 97% of the accesses to the L2 cache is a cache hit, 70% of the L1 references results in cache misses. Hence, we go a step further to locate the bottlenecks with the L1 cache.

Access Hotspots. It is important for programmers to know where the access hotspots lie because they are the optimization targets. Associated with the source code, hotspots are the critical data structures and their location in the program.



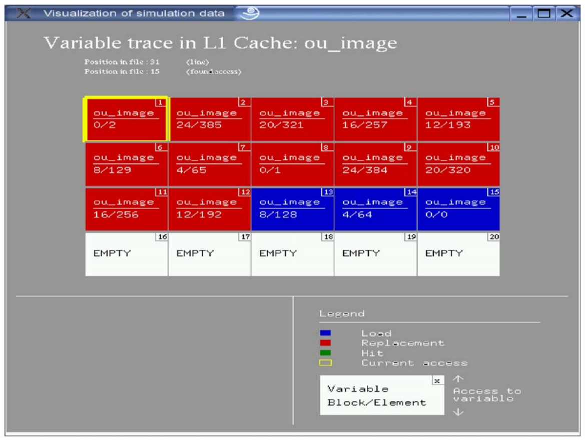
```
Information (total values)
#Misses            : 132068
#Cold Misses       : 6088
#Conflict Misses : 120960
#Capacity Misses : 5020
```

Figure 3. Variable Miss Overview of the 2D DWT implementation.

Figure 3 shows the Variable Miss Overview of the visualization tool. We use this view to find the access bottlenecks. The colored bars in the figure present the number of cache misses with each variable in the program. Since few cache misses have been observed with other variables, we only include the input and output image in the visualization.

From left to right, each bar depicts the total miss, the compulsory miss, the conflict miss, and the capacity miss. It can be seen that both matrices introduce cache misses, however, more than 80% of the misses is caused by accessing *ou_image*.

Having the bottleneck found, we now need the code region where the hotspots arise and the optimization shall be



```
Position in file : 31     (line)
Position in file : 15     (found access)
```

1 ou_image 0/2	2 ou_image 24/385	3 ou_image 20/321	4 ou_image 16/257	5 ou_image 12/193
6 ou_image 8/129	7 ou_image 4/65	8 ou_image 0/1	9 ou_image 24/384	10 ou_image 20/320
11 ou_image 16/256	12 ou_image 12/192	13 ou_image 8/128	14 ou_image 4/64	15 ou_image 0/0
16 EMPTY	17 EMPTY	18 EMPTY	19 EMPTY	20 EMPTY

Figure 4. Variable Trace: the first 15 accesses to the *ou_image* array in the vertical filtering.

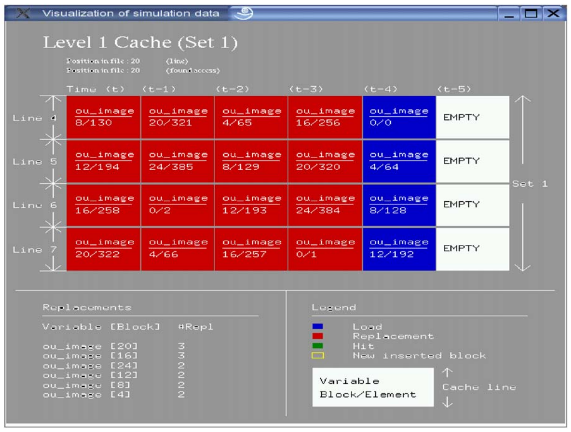
conducted. The 3D view of the visualization tool provides such information. Using this view, we found that a significant number of cache misses exist with the vertical filtering (the left one), while the horizontal filtering shows only a small number of misses. This leads to a conclusion: the access to matrix *ou_image* in the vertical filtering has to be optimized.

Miss Reason. Cache miss is caused by several factors: referencing a data block at the first time results in compulsory miss; mapping conflict introduces conflict miss; insufficient cache blocks cause capacity miss. Figure 3 has indicated that mapping conflict is responsible for the cache misses, however, for optimization programmers require the knowledge about how the conflict is generated. The Variable Trace and Cache Set views can be used to acquire this knowledge.

The Variable Trace view aims at demonstrating the access pattern of an array by showing the references to all of its data blocks/elements. This allows the programmer to find the reused data and the reuse feature. As depicted in Figure 4, twenty fields are applied to depict the access trace. Theses fields are filled from left to right, i.e. the first reference is initially illustrated in the first field and then moved to the next field after the second reference occurs. A display field gives the information about the access type (indicated by the color), name of the array (text in the upper line), the number of the accessed data block (left number), and the concrete array element (right number). The access type can be a load operation, a replacement, or a cache hit.

Now we use the Variable Trace to detect the access pattern of *ou_image*. First, a reuse behavior can be seen because the same data block is repeatedly accessed, while the references target on different array elements. For example, the first access, presented in field 15, is performed on element 0 in block 0 and the eighth, shown in field 8, addresses on element one which is stored in the same block. However, the color shows that the second access is a cache miss, indicating that block 0 has been evicted from the cache before the second access is issued. The same behavior can be seen with all other data blocks.

Based on the Cache Set view we discovered how the conflicts were formed. Figure 5 depicts a sample view which demonstrates the runtime activities of Set 1.


Visualization of simulation data

Level 1 Cache (Set 1)

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

	Time (t)	(t-1)	(t-2)	(t-3)	(t-4)	(t-5)
Line 4	ou_image 8/130	ou_image 20/321	ou_image 4/65	ou_image 16/256	ou_image 0/0	EMPTY
Line 5	ou_image 12/194	ou_image 24/385	ou_image 8/129	ou_image 20/320	ou_image 4/64	EMPTY
Line 6	ou_image 16/258	ou_image 0/2	ou_image 12/193	ou_image 24/384	ou_image 8/128	EMPTY
Line 7	ou_image 20/322	ou_image 4/66	ou_image 16/257	ou_image 0/1	ou_image 12/192	EMPTY

Replacements

Variable [Block]	#Repl
ou_image [20]	3
ou_image [16]	3
ou_image [24]	2
ou_image [12]	2
ou_image [8]	2
ou_image [4]	2

Legend

🟦	Load
🟥	Replacement
🟩	Hit
▭	New inserted block

Figure 5. The Cache Set view of Set 1 in L1.

The four lines in the figure correspond to the four data blocks in the cache set. Each line contains colored fields that depict the cache operations and the content update. The operations are combined with the color of the field, while the content update is presented by the chronological order of the fields with the left one replacing the right one. This means, the most right non-empty fields are the very initial value of each line in the cache set.

It can be seen that the presented cache set is first filled with block 0, block 4, block 8, and block 12. According to the Variable Trace view in Figure 4, theses blocks are reused. However, other required and reused blocks, i.e. block 16, 20, and 24, are also mapped in the same cache set and the access to them removes the previously used ones. As a consequence, the second access to the same data block is a miss and this access replaces another reused block. Therefore, all accesses to the same data block is a cache miss.

After examining the entire references to *ou_image*, we found the same behavior with all of its data blocks: they are reused but the reuse does not bring cache hits due to mapping conflict.

4.3 Code Optimization

The conflict miss with vertical filtering is caused by the fact that reused data have mapping conflict. To eliminate this conflict, we can either change the mapping behavior or distribute the conflicting data into separate calculations.

For the former we add a buffer of one cache line between the rows of array *ou_image*. In this case, it is defined as $ou\_image[N][M + LineSize]$ rather than $ou\_image[N][M]$. This results in a different mapping of its data blocks in the cache. Such technique is called array padding. The other scheme is called loop fission, where the computation is partitioned into several loops, each processing a part of the whole task. The idea is to reduce the amount of data needed for a single iteration so that required data can be held in the cache for the next iterations. Figure 6 shows the optimized version of the sample code segment in Figure 2.

In contrast to the original implementation, the optimized code distributes the same computation into two single loops. In this case, the data blocks in conflict are separated; hence no eviction will occur at the runtime. However, the additional loop also introduces extra instructions. Therefore, the padding scheme shall be better.

Besides conflict miss, DWT also depicts some compulsory and capacity misses. For the former, our pattern analyzer and pre-compiler take care of the prefetching task. For the latter, an explicit code optimization is required.

Capacity miss usually occurs with hierarchical loops, where data of the inner loop is reused by the iterations in the outer loop. Our visualization tool has shown that DWT is such an example. Actually, this feature is also depicted by the program. Examining the code segment in Figure 2, it can be seen that the neighbouring iterations of loop *i* share a part of the data accessed within the same *j* iteration. However, if the cache is not sufficient for holding the complete working set of the inner loop, the shared data must be removed from the cache before the next outer loop iteration is performed. According to the Variable Trace view, we found that the capacity miss with DWT is formed in this way.

Loop tiling is the traditional scheme for tackling capacity misses. With this approach, the inner loop is tailored to a size that reused data are not removed. An additional loop is inserted for maintaining the original work load.

```
  for (i=0, ii=4; ii<N-4; i++, ii +=2) {
    for(j=0; j<M; j++) {
       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]
    }
    for(j=0; j<M; j++) {
       in_image[i+ N/2][j] += ou_image[ii][j]   * high[0] + ou_image[ii+1][j] * high[1]
                           + ou_image[ii+2][j] * high[2];
    }
}
```

Figure 6. Optimization with loop fission.

```
  for (k=0; k<M; k+=blocksize)
    for (i=0,  ii=4; ii<N - 4; i++, ii +=2)
      for(j=k; j<min(M, k+blocksize); j++) {
       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 7. Optimization with loop tiling.

A key issue with this technique is the tiling size, i.e. the maximal number of inner loop iterations which guarantees that the cache does not remove reused blocks. It is clear that this parameter can not be simply acquired by analyzing the code. However, the visualization tool can provide this information. First, programmers can use Variable Trace to find the first data block reused in the outer loop. Relying on the Cache Set view, they can further detect which block replaces this reused one. For the case of a 2K L1 cache, for example, we found that the first reused data block is replaced while the loop variable *j* has a value of 32. This means if the inner loop stops at this point and the outer loop starts, the reused data would be maintained in the cache. Hence, for this cache the tiling size is 32.

Actually, the tiling size can be calculated for other cache sizes. This is also theoretically correct. For example, a 4K cache has a double capacity as a 2K cache, can hold twice as much data, and hence the inner loop is allowed to contain twice as many iterations.

Figure 7 shows the optimized code segment with loop tiling. In contrast to the original implementation, a loop with index k is added to the code. This loop divides the computation in the j loop into smaller blocks (noting the change of loop condition with j). In this case, loop j processes only `blocksize` elements rather than the whole row.

5 Experimental Results

Based on the performance tools, we have found the cache problems with DWT and applied padding, fission, and tiling techniques to optimize the program. Optimization with compulsory miss is conducted by the pre-compiler automatically.

In order to observe the impact of individual techniques without their interaction involved, we apply only a single optimization scheme for an optimized version. In this case, we achieved five variants of the same program: the original implementation, the padding version, the fission version, the tiling version, and the code with prefetching.

Actually, the optimization is based on a single run with certain data size and cache configuration. For conflict miss the resulted code is probably not optimal for other working sets and cache organizations because the mapping behavior changes. However, in order to examine the applicability of this approach, we run the same optimized version on different machines and using various data size. For capacity miss, since the tiling size can be calculated, the corresponding variable in the program is adapted to the target architecture. The prefetching technique is independent of both working load and cache configuration. Currently, prefetching is guided by access stride and access chains. The former specifies the distance between accessed elements in an array, while the latter is a group of accesses which often occur together. It is clear that this kind of access pattern does not rely on the data size or the cache organization.

The programs are run on four different machines: a Pentium 3 with an L1 cache of 16K, 2-way, and 32B of cache line, a Pentium 4 with an L1 of 8K, 4-way, and 64B of line size, an AMD Opteron with an L1 of 64K, 2-way, and 64B of cache line, and an Itanium II with an L1 of 16K, 4-way, and 64B of line size. Figure 8 to 10 depicts the experimental results.

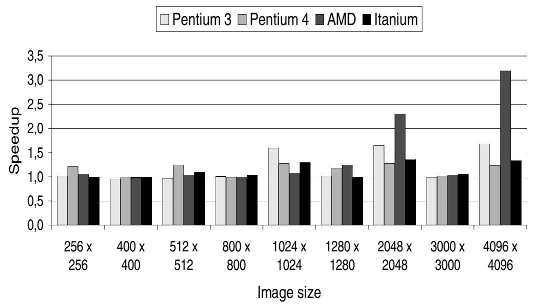


Figure 8. Performance improvement achieved by the padding technique.

Figure 8 shows the speedup of running the padding version for tackling conflict miss. Here, speedup is calculated by the execution time of the original implementation divided by the execution time of the optimized version. It can be seen that for image sizes of a power of two, a clear speedup has been acquired. For other images the improvement is not high. Using the toolset, we found that these images have no or less mapping conflict and as a consequence the optimization is not required. Overall, this result indicates that DWT has a regular access pattern and for such applications the optimization based on a single run also works for other configurations.

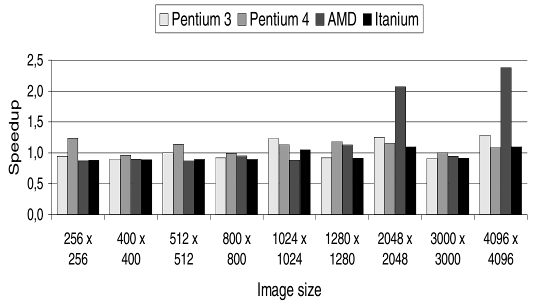


Figure 9. Performance improvement by using loop fission.

As presented in Figure 9, similar result has been obtained with the fission technique where images with a size of a power of two shows better speedup. In contrast to the padding scheme, however, slowdown can be seen with this technique.

The slowdown comes from the additional instructions introduced by the fission strategy. It is clear that running these instructions causes a performance loss. If the gain by reduced memory access penalty can not compensate for this loss, the optimization does not bring any speedup.

Figure 10 depicts the result with the tiling technique. It is not surprising that the speedup is low because DWT does not show much capacity misses. Similar to loop fission, tiling also introduces additional instructions. Due to the same reason, slowdown has been observed.

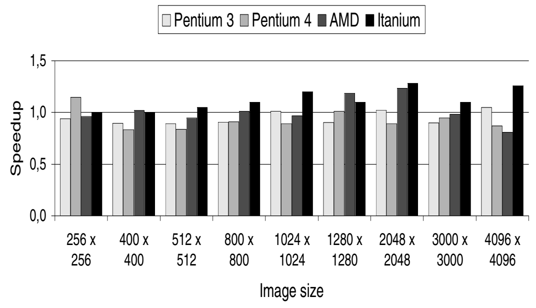


Figure 10. Speedup with loop tiling.

The improvement in execution time is directly contributed by the reduction of cache misses. Figure 11 depicts the statistics on total L1 misses delivered by our data profiler.

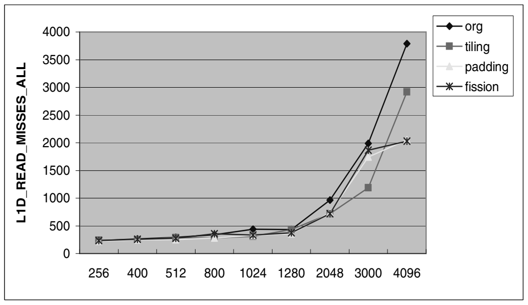


Figure 11. Statistics on L1 read miss with different image sizes.

For small image sizes up to 512, the picture can not clearly show the actual behavior because the number of misses is low. Hence, we observe the points with large images, for example, the last two of size 3000 and 4096. For the first case, the tiling scheme reduces cache misses significantly. Since conflict miss is not critical with this image size, the other schemes reduce the cache misses only slightly. For an image size of 4096, both mapping conflict and capacity problem exist. Therefore, padding and fission considerably decrease the miss number, while the tiling strategy also works well. This better cache behavior directly contributes to the reduction in execution time.

Finally, we take a look at the prefetching result. As DWT does not show much compulsory misses, we only achieved

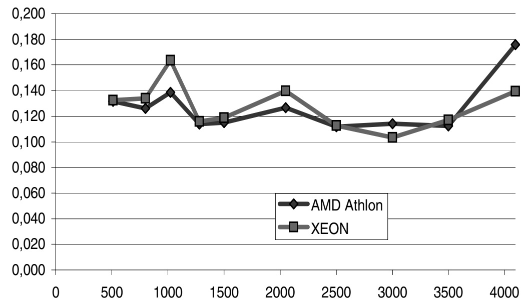


Figure 12. Improvement with prefetching.

a slight performance improvement on some machines due to the prefetching overhead. Here, we choose the two best cases for demonstration.

Figure 12 depicts the improvement in execution time of the prefetching version against the original implementation. First, several peak points can be seen with both curves. They are actually the results with image sizes of a power of two. This can be explained by the fact that more cache misses occur with theses cases due to mapping conflict and as a result more misses can be eliminated with optimization. Overall, we achieved the best improvement of 18% with an image size of 4096.

6 Conclusions

In this paper, we demonstrate how to apply performance tools to optimize the cache access behavior of a Discrete Wavelet Transform implementation. Based on the presented cache problems and the cause for them we perform optimizations using different techniques. These schemes and the related parameters are concluded from the observed runtime cache and application access pattern, hence are efficient and concrete. The experimental results show improvement in overall performance with all kind of optimizations.

References

[1] Y. Andreopoulos, P. Schelkens, and J. Cornelis. Analysis of Wavelet Transform Implementations for Image and Texture Coding Applications in Programmable Platforms. In *Proc. IEEE Signal Processing Systems*, pages 273–284, 2001.

[2] J.-L. Baer and T.-F. Chen. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[3] H. Brunst, H.-Ch. Hoppe, W. E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Proceedings of the ICCS 2001*, pages 751–760, 2001.

[4] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI For Hardware Performance Monitoring On Linux Systems. In *Linux Clusters: The HPC Revolution*, June 2001.

[5] Nicholas Nethercote et al. Valgrind. available at http://www.valgrind.org/.

[6] T. Inagaki et al. Stride Prefetching by Dynamically Inspecting Objects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–277, June 2003.

[7] HP. Perfmon Project Web Site. available at http://www.hpl.hp.com/research/linux/perfmon/.

[8] Intel Corporation. Intel VTune Performance Analyzer, 2004. available at http://www.cts.com.au/vt.html.

[9] W. Lin, S. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the 7th International symposium on High-Performance Computer Architecture*, pages 301–312, January 2001.

[10] B. Quaing, J. Tao, and W. Karl. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization. In *High Performance Computing and Communcations: First International Conference, HPCC 2005. Proceedings*, volume 3726 of Lecture Notes in Computer Science, pages 694–703, Sorrento, Italy, September 2005.

[11] S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[12] J. Tao, T. Gaugler, and W. Karl. A Profiling Tool for Detecting Cache-critical Data Structures. In *Proceedings of Euro-Par 2007 Parallel Processing (Lecture Notes in Computer Science)*, pages 52–61, August 2007.

[13] J. Tao, S. Schloissnig, and W. Karl. Analysis of the Spatial and Temporal Locality in Data Accesses. In *Proceedings of ICCS 2006*, number 3992 in Lecture Notes in Computer Science, pages 502–509, May 2006.

[14] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proceedings of the ICCS 2004*, pages 440–447, 2004.