# Cache Impact on Image Processing Performance

Tiago Chaves Bezerra

*Departamento de Informática, Universidade do Minho*
*4710 - 057 Braga, Portugal*
*mi7300@mestrado.di.uminho.pt*

**Abstract.** This communication analyses a non-linear relationship between the processing time and the size of an image. Research methodologies were applied to explain this behaviour, such as verifying the "memory mountain", debugging the code and using a cache profiling tool. The main cause of this non-linear behaviour is shown to be the merge of the cache size, their associate degree and the image location address.

## 1 Introduction

High performance has always been a concern in several computer areas. In the domain of image processing, where the time needed to analyse an image may be high, it is possible to make some optimizations to obtain improved processing times. However, some optimization techniques may depend on the configuration of the hardware elements, and it may not be easy to achieve the application's best performance.

With this goal, this communication will discuss one of the several situations where an application code may present a performance anomaly due to the cache organization. A specific case study of image processing was used to understand the cache hierarchy and apply the knowledge to avoid such undesirable behaviour. The case under study is an image processing algorithm that converts images from a 256 grey scale to 2 colours (black and white).

A reported problem occurs when the size of the image grows: the time to process it did not always grow proportionally with the size, with periodic glitches. This periodic behaviour suggests that the cause could lie in the cache hierarchy.

To study this particular case, it was followed the methodology described below. The first step is to verify the problem reported by executing the algorithm in other computers with other environments. After recognizing the problem, it is necessary to inspect the problem, verifying the code implemented. The third step is to make an analysis about the situation, looking for similar problems and causes. The fourth step is to explain the reported problem with a scientific foundation. Once the problem and its causes are identified, try to adjust the code to solve it, without modifying the algorithm.

This communication has the following structure: Section 2 - Basic concepts, a review of concepts required to understand the problem; Section 3 – The image processing problem, a description of the problem and the beginning of the code inspection; Section 4 – Analysis of the cache impact, which shows the data structures and the cache performance; Section 5 – A possible explanation, presenting a possible explanation of the problem; Section 6 – Conclusion, closes the communication by reporting some interesting aspects of the case study.

## 2 Basic Concepts

This section presents some concepts related to memory hierarchy, namely memory cache, to better understand the problem to be analysed.

## 2.1 The Memory Hierarchy

To obtain a better relation of cost, capacities and performance, the memory system is a hierarchy of storage devices with different capacities, costs, and access times.

The registers, in the CPU, are the faster memory and hold the most frequently used data, but is also must expensive memory, that's the motive of only have a few of them in the CPU. After the register there are a subset of memories, small, fast cache memories near the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

This structure may produce a variance of the access time by factors of ten, or one hundred, or even one million, and that may result on significant and inexplicable performance slowdowns in programs execution [1].

## 2.2 Cache Memory

In general, a cache is a small, fast storage device that acts as a temporary area for data objects, which was recently requested (principle of locality [1]), stored in a larger, slower device.

But, as know the cache near the CPU is small, it is necessary a technique of replacement. There are three possibilities:

- direct mapped: a specific block can only go in one place in the cache, usually, at *address MOD number of blocks in cache*;
- fully associative: a specific block can go anywhere in cache;
- set associative: a specific block can go in one of a set of places in the cache.

A *set* is a group of blocks in the cache. In a *set associative* cache, a block is first mapped to a specific *set* by using *block address MOD number of sets* in the cache, after that the block may then be placed anywhere in that set. If sets have *n* blocks, the cache is said to be *n-way set associative*.

When a block is requested and there is no place to put it in the cache, it is necessary a replacement rule. This only applies to fully associative and set associative caches. For direct mapped, each block can only be placed in one location. The replacement rules can be:

- random: choose a block from the set at random;
- LRU: choose the least- recently used block; this approach has been unused for the longest time; it requires extra bits in the cache to keep track of accesses, it turns out that LRU is not much better than random replacement.

About the cache it is also important to know some measurement, which allows to verify a good use of the cache:

- cache hit: occurs when a particular data object is requested from a level *k* to a level *k + 1*, and the level *k + 1* has the object data to respond;
- cache miss: occurs when a particular data object is requested from a level *k*, and the level k+1 has not the object data, needing to request to a higher level on the hierarchy (k + 1+ n), where $n >= 1$ .

The cache performance is evaluated with a number of metrics [1]:

- miss rate: the fraction of memory references during the execution of a program, or a part of a program, that misses. It is computed as *number of misses / number of references* ;
- hit rate: the fraction of memory references that hit. It is computed as *1 - miss rate* .
- Hit time: the time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is typically 1 to 2 clock cycle for L1 caches.
- miss penalty: any additional time required because of a miss. The penalty for L1 misses served from L2 is typically 5 to 10 cycles. The penalty for L1 misses served from main memory is typically 25 to 100 cycles [1].

There are reasons for a cache miss, known as the three C [2]:

- compulsory: the first access to a block cannot be in the cache, so there must be a compulsory miss;
- capacity: if the cache is too small to hold all of the blocks needed during execution of a program, misses occur on blocks that were discarded earlier. In other words, this is the difference between the compulsory miss rate and the miss rate of a finite size fully associative cache;
- conflict: if the cache has sufficient space for the data, but the block cannot be kept because the set is full, a conflict miss will occur. This is the difference between the miss rate of a non- fully associative cache and a fully- associative cache. These misses are also called collision or interference misses.

A special case of conflict miss is the *trashing*, it is the situation where a cache is repeatedly loading and evicting the same sets of cache blocks. There are many types of optimizations, some of that are, try to reduce cache misses, increase cache hit and predict future request of data, putting it before be asked into the cache.

About the write policy, this determines what happens when a block is written to the cache, and when the write is communicated to the lower level (main memory):

- write-through: in this scheme, the block is written both to the cache and main memory.
- write back (also copy back): in this scheme, only the block in cache is modified; main memory is modified when the block must be replaced in the cache; this requires the use of a dirty bit to keep track of which blocks have been modified.

## 3 The Image Processing Problem

Several tests were performed with the algorithm, modifying the images sizes (in pixel), from 64x64 pixels up to 4096x4096 pixels, every 64 pixels, on each axis (i.e. 64x64, 64x128, … 64x4096, 128x64, … 4096x64, 4096x128, …).

The tests were run on two computers: Mickey - Intel Centrino at 1.6 GHz, 512MB of RAM, 32KB data cache on L1 and 1MB unified cache on L2 - and Alfa - Intel Pentium 4 at 2.6 GHz, 1GB of RAM, 8KB data cache on L1 and 512KB unified cache on L2 [3].

As shown in the graph bellow, it is possible to see some peaks according to the image size. The two computers showed similar results after several execution of the algorithm. This behaviour was unexpected, once the most logic was a constant growing up time, this is the question here placed.

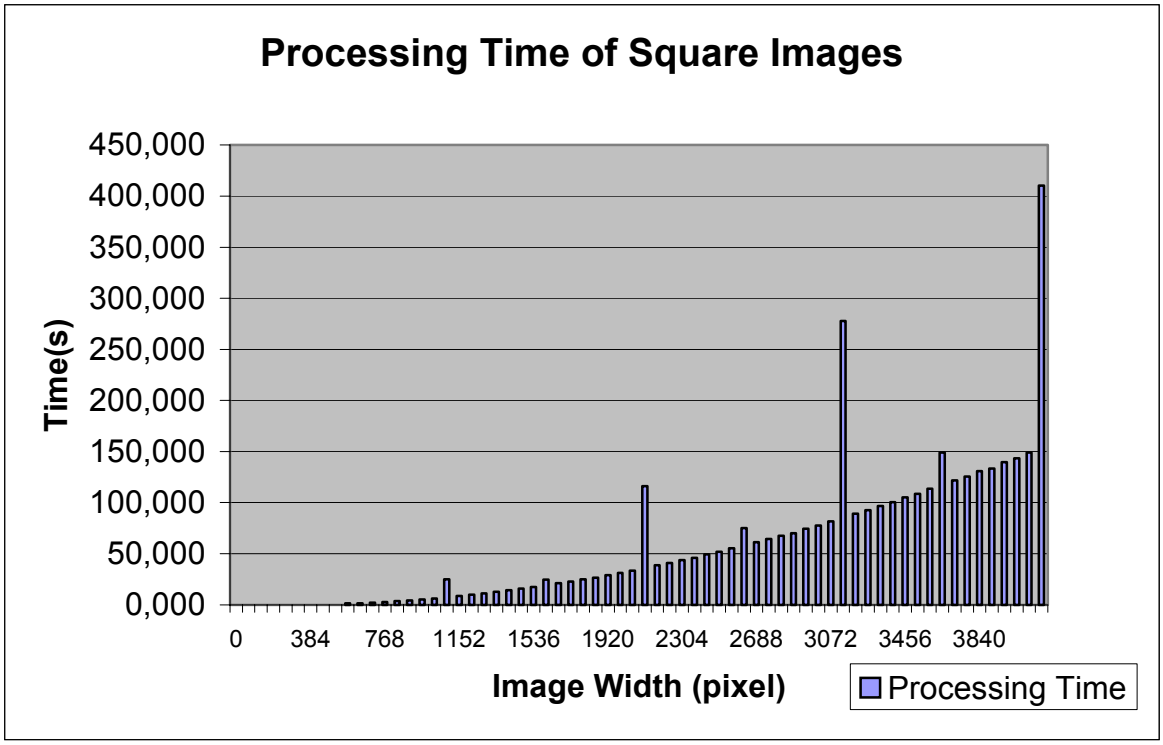


**Figure 1**: Result of the time spent to process different image sizes on Alfa

### 3.1 Code Inspection

The code was inspected aiming to find the code portion whose processing execution time was longer with a 1024x2024 pixels image than with a 1026x1026 pixels image. That part of the code would be responsible for the peak showed before. To inspect the code Linux tools were used, such as *gdump*, *ddd*, *kcachegrind-0.3b*. These are debugging and profiling utilities. The code was disassembled to better analyse its behaviour. The entire assembly code is on Appendix B of the full report[1].

Table 1: Cost taken to process the functions in images with different sizes, percent of the entire processes cost. Extracted from *kcachegrind-0.3b*, report

| Function / size of the image | 1024x1024 pixels | 1026x1026 pixels |
|---|---|---|
| bina() | 99,27% | 99,25% |
| getXY() | 45,85% | 43,60% |

The table shows that the bigger cost spent on the functions *bina* and *getXY* are inversely proportional to the image size. The function *bina* is the function that decides if the pixel will go black or white. The function *getXY* is called inside the function *bina*, which is the function that returns the value of the pixel in the coordinate *x, y*.

The code implements the algorithm that transforms grey images in black and white images using a window median limit [4]. The principal part of the *bina* code function is showed bellow. The entire code is in Appendix A of the full report[2].

[1] Available in http://gec.di.uminho.pt/micei/ac0304/icca04/proc/w2-vision.zip
[2] Available in http://gec.di.uminho.pt/micei/ac0304/icca04/proc/w2-vision.zip

```
...
/* Loop Y*/
for (y=sob; y<pAltura-sob; y++){
      // Loop X
      for (x=sob; x<pLargura-sob; x++){
            // put the pixel at Black or White
            if (getXY(x,y)>200){
               setXY(x,y,255);
            }else{
               if ( getXY(x,y) >= media - K*desvio)
                     setXY(x,y,0);
               else
                     setXY(x,y,255);
         }// end if
   }// end for x
}// end for y
...
```

This part of the code represents two inner loops that make a verification of each element value, by testing if it will be black or white.

### 3.2 Analysis of the Cache Impact

The research has converged into two situations: the impact of caches on program performance and the effects of array size in cache performance. Similar problems were found with different computers [2].

The first theme is about a study of the cache hierarch and an evaluation of the time taken to get object data on the different cache level. As proposed by the authors of [1], the result obtained after run the application *mountain.c*, on Mickey computer is showed below.

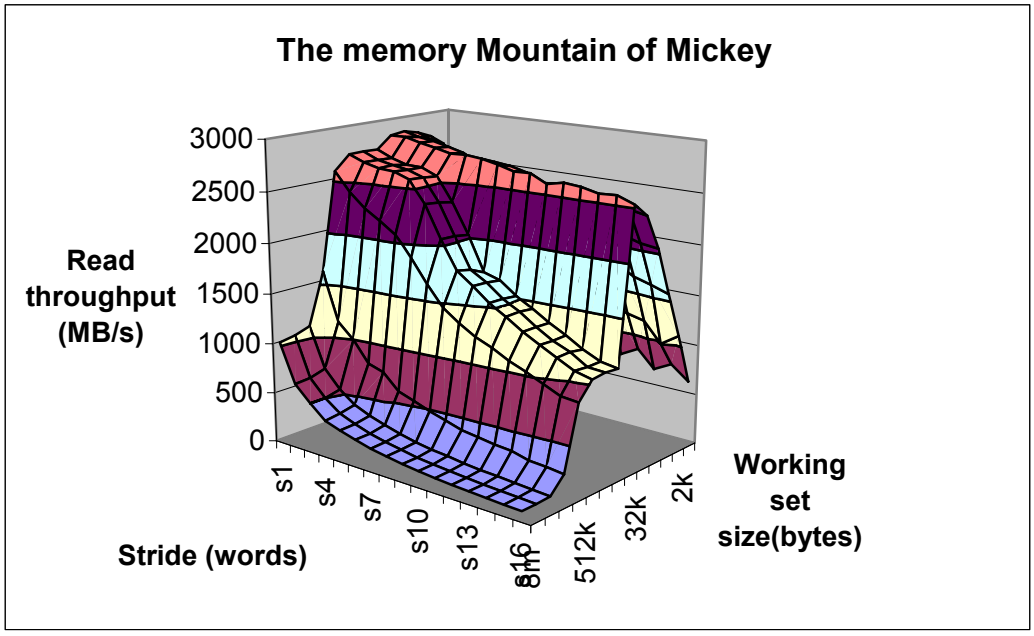


**Figure 2**: The memory mountain of Mickey

The second is a study cache misses. Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of two. To verify more precisely if the problem reported is really a problem about rate miss in different level

of cache, it was necessary to exam the cache hit and misses in the application runtime. To this it was used the Kcachegrind-0.3b. It is an application that can show the process time, memories and cache uses.

The graph bellow shows the cache miss rate on the two levels, L1 and L2. Complete results are in Appendix D of the full report[3].

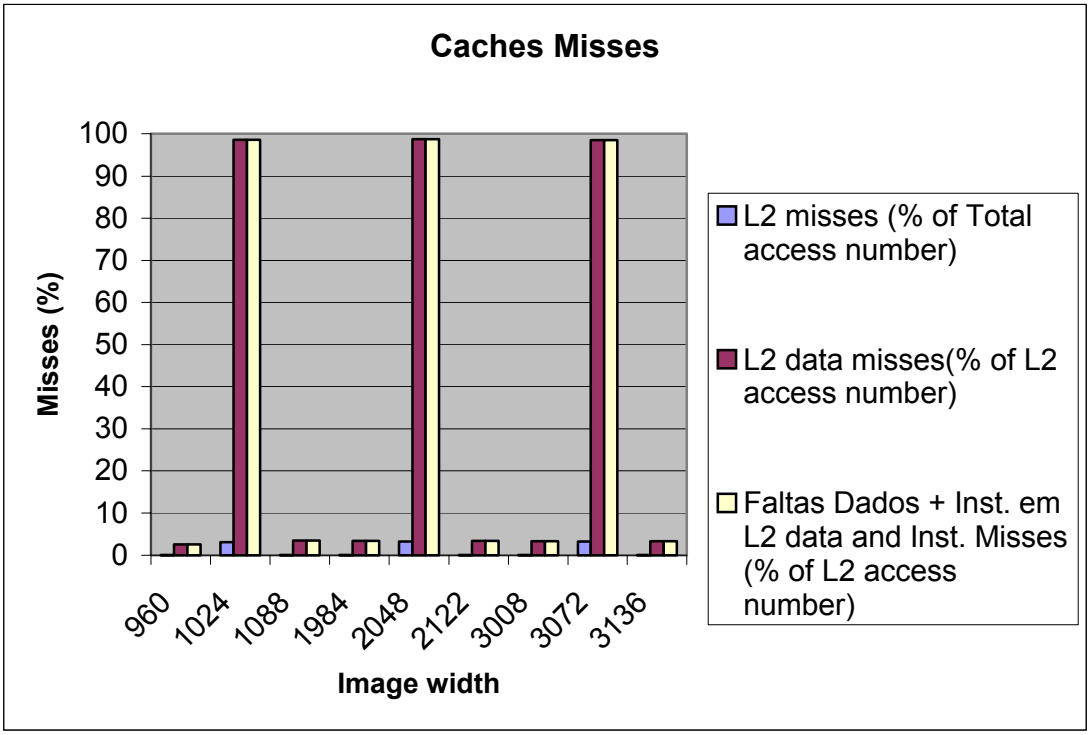


**Figure 3**: Graph of cache misses of the execution with specific images width

### 3.3 The Explanation

As already described in the basic concepts section, the cache may optimize the process time. But, it is still a question about the peaks of processing time for some image sizes. The response is a relation between the cache size, its associative degree and the address of the memory allocated for the image data. In order to demonstrate the problem, an exemplification is necessary.

Imagine two images, *inImage* and *outImage*, represented by two consecutives arrays in the memory. The algorithm is simple; just sum a constant value to the elements of the first array, putting it on the second array at the same position. To do that, the Alfa computer was used: previously described, with 8KB L1 data cache (4-way set associative) and L1 instruction cache for 12K µops, 512KB L2 cache (8-way set associative), with 1024 sets of 8 data lines in L1 and xxx sets of 8 lines in L2. Usually, to map a block onto a set, the *block Address MOD Number of sets* equation is used.

With image sizes up to 512x512 pixels everything works fine since both images fit into L2 cache. Once they are initialized, they may stay there for the whole executing time.

However, it works differently for image sizes having 1024 or 2048 pixels. The array data are not only concurring for a cache space, but correspondent elements are also concurring to the same set. The description step by step is (if implements write-back):

[3] Available in http://gec.di.uminho.pt/micei/ac0304/icca04/proc/w2-vision.zip

1) Load a block from inImage
    1a) Cache misses in L1 cache: block needs to be caught
    1b) L1 cache line must be replaced, it may be a line with outImage block
    1c) Write back dirty cache line to L2 cache [out]
    1d) Go to L2 cache, cache miss there as well, block still need to be caught
    1e) L2 cache line must be replaced, it may be a line with outImage block
    1f) Write the dirty L2 cache line back to memory [out]
    1g) Read 128-byte, L2 cache line from inImage, into L2 cache.
    1h) Read the appropriate quarter (32 bytes) into L1 cache.
    1i) Complete the read of one byte.
2) Do the computation (a few clock cycles).
3) Do the store of one byte to outImage
    3a) Cache misses in L1 cache (since inImage is there)
    3b) Go to L2 cache, cache miss there as well
    3c) Read, 128-byte L2 cache line from outImage, into L2 cache.
    3d) Read the appropriate quarter (32 bytes) into L1 cache.
    3e) Do the store byte.

Then, do all this again for the next byte.

As may be perceived, the element to be read and the element to write are directly mapped in the same set. So the *outImage* entry (most of the time) may have just replaced the *inImage* entry, so there is constant thrashing. Therefore, in the worst case, each byte in the input array causes two 128-byte reads from memory, and one 128-byte write-back to memory. If the block size is smaller than 128 bytes, it also prevents a cache hit.

For the other sizes, there are still cache misses (since the two 512KB arrays do not fit into the 512KB cache together), but the caches work as intended, i.e., at least get to read 128 bytes of *inImage* (with one memory read) and write 128 bytes to *outImage* (with one 128-byte read, and later a 128-byte write-back).

The figure bellow shows different blocks of memory, requested by the CPU, concurring to the same space on the lower cache memory level

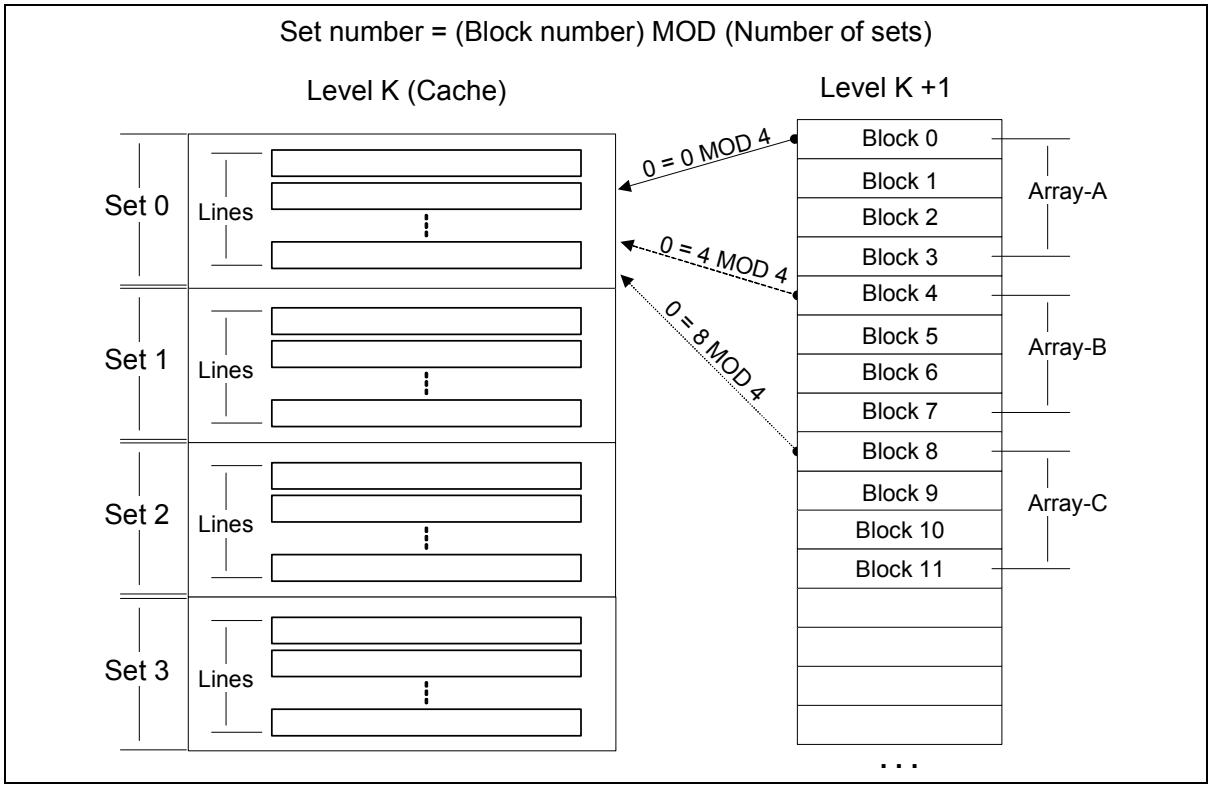


Figure 4: Blocks mapping between different memory levels

## 4 Conclusions

To obtain higher performance an adequate use of cache hierarchy has always been a good technique. However, these techniques may generate unexpected results. This particular case study shows an unexpected behaviour with interesting image sizes. The time taken to process some images was significantly higher than the time taken to process other larger images.

According to the research made, the most possible cause of the problem is a coincidence of hardware memory cache sizes, their associative degree and the address of the allocated array elements. This is a peculiar example of cache misses conflict. This behaviour was more significant due to the number of cycles necessary to process the entire image. One easy solution is to place some blocks of bytes at the end of each array.

There are several ways to improve performance, but not all of them are the best technique for all the problems. Even wise programmers may not understand all impacts of a memory hierarchy. It is relatively simple to produce efficient programs with fast average memory access times, but it is harder to understand why expected behaviours happen or not.

## References


[1] Bryant, Randal and O'Hallaron, David: Computer Systems - A Programmer's Perspective, Prentice Hall, (2002) 277-347

[2] Hennessy, John L. and Patterson, David A.: Computer Architecture – A quantitative approach, Third edition, Elsevier Science USA (2003) 390-513

[3] Intel: IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide, 2002, p 10-3.

[4] L. Eikvil & T. Taxt & K. Moen: "A Fast Adaptive Method for Binarization of Document Images", Proc First Int´l Conf Document Analysis and Recognition, France, 1991, p. 435-443.