IF4-ARCH (Architecture)

TP2 – practical session 2

Report - evaluation of cache memory performances

Authors
Stepan Blaha (SB)
Kubicka Matej (MK)
Iram Laurencio Gallegos Tobías (IGT)

Class
IMC4

Professor
Eva Dokladalova

Date
December 15th, 2011

Objectives

1. To be able to estimate the performances of cache memory
2. To understand the cache memory parameters (size, line, associativity);
3. To understand the programming style consequences on the efficiency the cache memory;
4. To be able to execute the profiling of the run time of an application.

Exercise 1

(programmation done by Blaha and Kubicka, analysis by Kubicka, results revision by Iram GT)

Aim of this exercise was to write a program in C programming language which simulates working with large working sets (copying of matrix with size 1000x1000 cells). We were supposed to write program in three distinct ways – first with reading column-by-column, then line-by-line and finally with optimized function `memcpy()` from standard library which can copy the data in somewhat optimized way.

The three programs were added to this report, please see enclosed source files `arch-tp2ex1.c`, `arch-tp2ex1-2.c` and `arch-tp2ex1-3.c`. Source codes are also included in the appendixes F, G and H.

Implementation notes

- Testing programs doesn't print anything into the standard output or more generally interact with the user in any way – those actions would spoil results of the profiling tool `valgrind`.
- Working set size can be affected by input argument – there is only one parameter which describe the size of a square matrix side. Total working set size is then computed as a square of given argument. For example if we call `./arch-tp2ex1.c 1000,` working set of size `1000*1000*sizeof(int)` will be allocated for each of two matrices. This behavior was implemented for greater scalability of our tests, however, we have used only matrices with size of 1000 to 1000 elements.
- No checking for errors was implemented – we suppose the environment can provide enough resources and the program was called properly. Purpose of the program is to conclude the test, not to be foolproof.
- For memory initialization we use in all cases `stdlib` function `calloc()` which automatically sets whole allocated memory block to zeros, which simplifies the source code (although not required).

Parameters of cache memory

Early computer designs used memory with speed not that different from MIPS performance of the CPU. In this system the memory access was not slowing down the system. In early 90's CPU performance have risen dramatically, but access time to DRAM memories have not fallen proportionally to it. The gap created between the speed of memories and the CPU speed became large and because most of CPU activity is related to memory access, it was crucial to find a way to at least

partially bypass this problem [1].

The solution was CPU cache memory, based on a principle discovered in 1960's. CPU cache stores commonly used data to memory with the same speed as CPU (an SRAM made of RS flip-flops). This cache have relatively small size, but it can provide it's data quickly.

The cache is generally hidden from programming point of view, it works as a transparent block between the CPU and the slow memory. Also there are separated caches for instructions and for data. Furthermore multilevel caches are typically implemented to achieve good cost-performance trade-off [1]. With higher level cache usually comes larger memory and lower speed (larger access time).

As of today we have I1 (sometimes called as “L1i”) and D1 (sometimes referenced as “L1d”) cache for each processor core (level-1 caches for instructions and for data), L2 cache (L2i + L2d) and in modern processors also an L3 cache common for all cores.

Note.: Cache implementation design differs from manufacturer to manufacturer – so does the terminology. In literature we can meet different naming conventions.

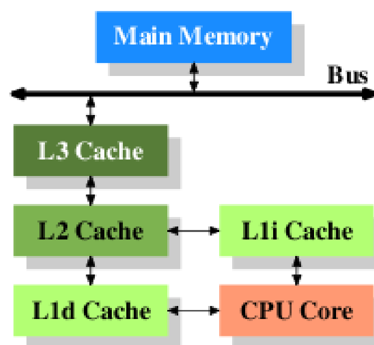

Figure 1 – Processor with L3 cache memory (taken from [1])

Some rules have to be applied in order for the cache memory to be quickly accessible – particularly to decide where should a copy of the memory be placed. There may be cache where copy of a memory can be stored anywhere (**fully associative** cache [1]), or a memory where copy of a memory can be stored on exactly one address (**direct mapped** cache [1]). Those are extreme cases – for fully associative cache we have to go through whole content to find cached entry (it takes time), for directly mapped cache we know where exactly are cached data stored, but we need a cache with size of the original memory to be able to map any block of data to any block of cached data (it takes memory).

Solution is a trade-off in form of 1-way, 2-way, 4-way and 8-way associative memories. For example, with 8-way associative cache we can place data into one of eight applicable addresses and to read we need to check up to 8 addresses for our data. This trade-off allows us to use reasonably small cache memories with limited number of places where data with specific address can be stored in the cache.

Measurement results – cache misses

We measure how many times accessed data/instructions were not in the cache memory.

	Total executed instructions	*Total data accesses*	*L1i*	*L2i*	*L1d*		*L2d*	
					read	*write*	*read*	*write*
column-by-column	26,196,356	18,099,039	0.00%	0.00%	6.20%	52.40%	0.10%	6.20%
line-by-line	202,494	106,128	0.34%	0.34%	2.20%	1.10%	1.80%	1.00%
`memcpy()`	190,644	95,608	0.36%	0.36%	2.60%	1.00%	2.00%	0.90%

Table 1 – Cache misses for all three cases

Note: misses are represented in percentage, for exact results see appendix A.

Evaluation

Case 1 (colum-by-column) provides far worse results than the others. It seem logical – we approach data in memory column by column, but actual data in memory are stored row by row (by program/test design). In order to read next cell in the column we need to skip in memory one whole line (a 1000 integer numbers – 4kbytes). From the results in Appendix A we can see, that this approach need 125x longer time to finish, 180 times more memory accesses and even cache L1d did many misses because data were accessed irradically (with jump between any two matrix cell) and so practical cache efficiency got low.

Case 2 (line-by-line) is manual copying of data from one place in the memory to another. Data are accessed in the same manner as they are represented in memory. From the measurement result we can see that this solution is almost as good as using optimized functions from standard library.

Case 3 is using optimized function from standard library - `memcpy()` - to copy the data from one matrix to another. Measurement results are slightly better than in Case 2 (line-by-line) due to internal optimizations of standard library. In general, this case provides the best results. Data are not accessed row-by-row, or column-by-column, `memcpy()` works with a block of memory to be copied from one place to another.

Exercise 2

(analysis done by Iram GT, results revision by Blaha and Iram GT, report text by Kubicka)

Aim of this exercise is to measure number of cache “misses” with different cache D1 parameters. We were manually changing associativity and cache size (for detailed information see exercise 1 or [1]) in order to see how number of misses varies for case 2 and case 3 from previous exercise.

We measured both cases for 1-way, 2-way, 4-way and 8-way associative memories. Furthermore each measurement have been be done for memory with size 1kB, 2kB, 4kB, 8kB, 16kB, 32kB, 64kB and 128kB. See Table 2 and Table 3 for the results.

For the purposes of this exercise we wrote automated script which does all the measurement. In order to run the test on case 2 we have to call “`./ex2-automate.sh ./arch-tp2-ex1-2 1000`”, for case 3 it is: “`./ex2-automate.sh ./arch-tp2-ex1-3 1000`”. For the script see Appendix B.

Cache misses	*1-way associativity*	*2-way associativity*	*4-way associativity*	*8-way associativity*
1kB	16,402	13,362	12,533	12,682
2kB	13,095	9,971	8,848	8,585
4kB	9,358	8,166	7,701	7,539
8kB	8,126	7,202	7,072	7,053
16kB	7,554	6,982	6,918	6,923
32kB	7,014	6,677	6,663	6,658
64kB	6,493	6,351	6,309	6,280
128kB	6,429	6,190	6,164	6,161

Table 2 – Cache misses for case 2 (line-by-line) in various configurations

Cache misses	*1-way associativity*	*2-way associativity*	*4-way associativity*	*8-way associativity*
1kB	16,295	13,019	12,267	12,614
2kB	12,759	9,563	8,589	8,035
4kB	9,133	7,960	7,385	7,256
8kB	8,041	7,033	6,912	6,882
16kB	7,232	6,834	6,767	6,772
32kB	6,791	6,577	6,512	6,505
64kB	6,365	6,181	6,145	6,106
128kB	6,279	6,013	5,987	5,982

Table 3 – Cache misses for case 3 (`memcpy`) in various configurations

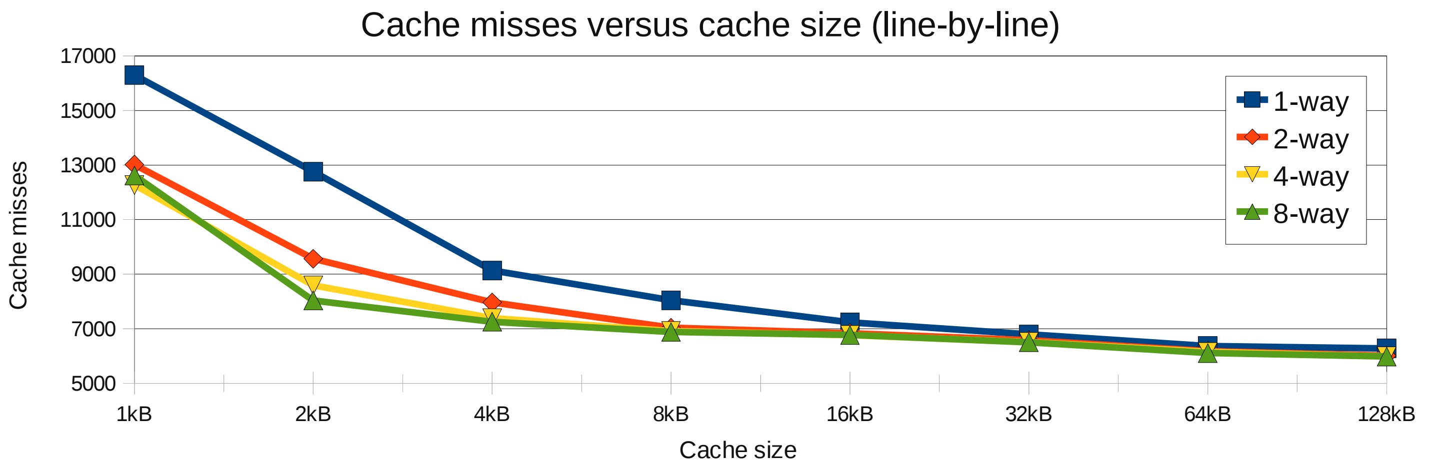

Figure 2 – Cache misses versus cache size (case 2 - line-by-line)

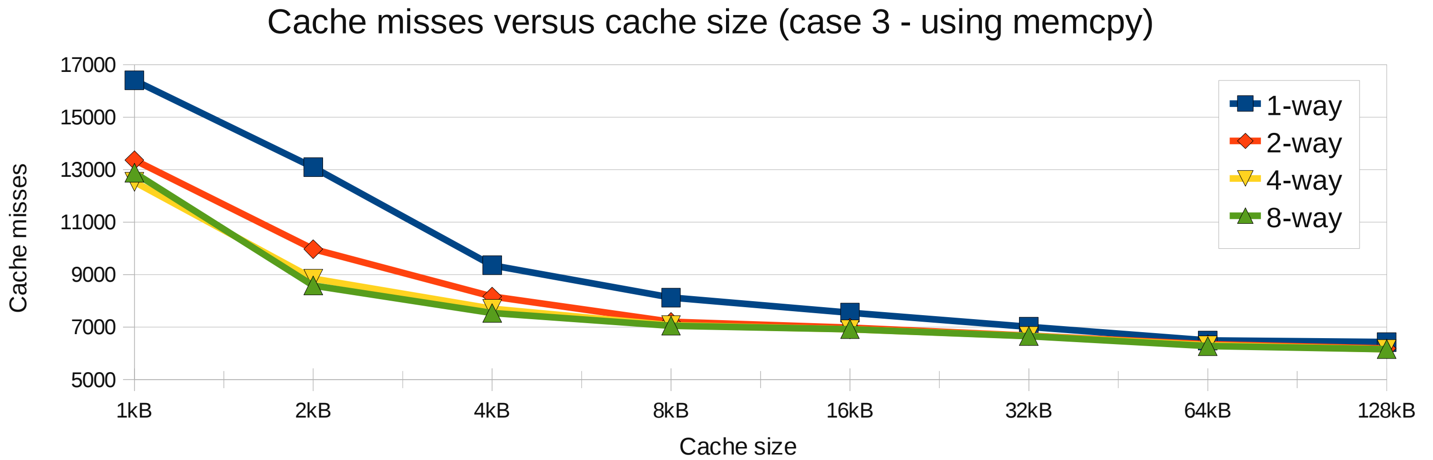

Figure 3 – Cache misses versus cache size (case 3 - memcpy)

Results show that using `memcpy()` gives always a smaller amount of misses no matter the cache size. The difference gets lower as cache size rises. Somewhere around 64kB cache size the difference between method with `memcpy()` and manual copying line-by-line stabilizes around 150 misses. Also, number of cache misses is no longer improving for both cases after going over 64kB of cache size. See Figure 4 for compared results of both methods.

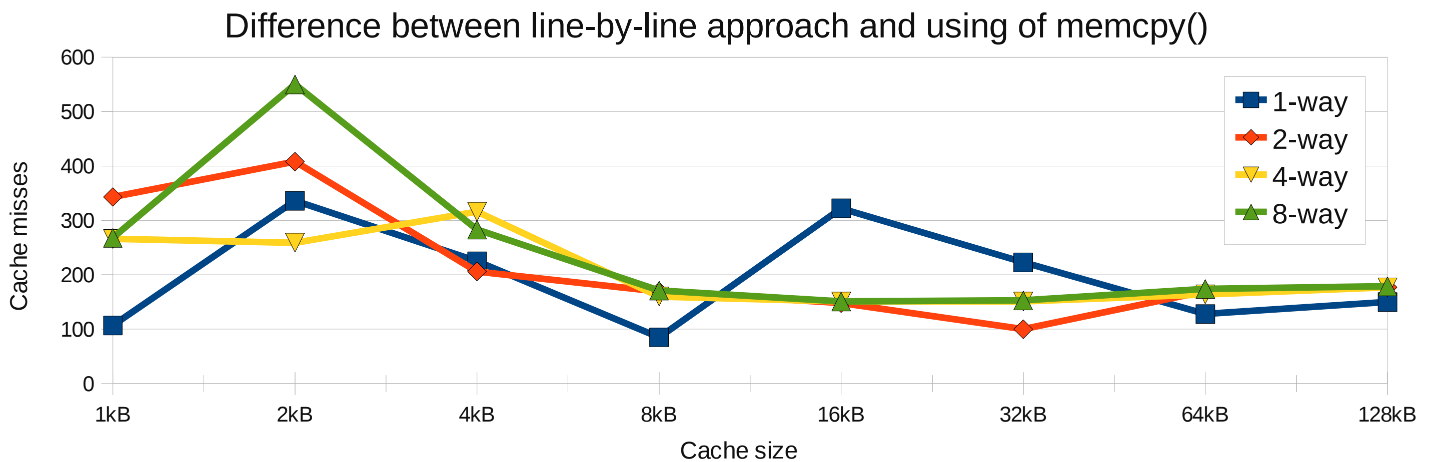

Figure 4 – Difference in cache misses between line-by-line copying (case 2) and optimized `memcpy` function (case 3)

Exercise 3

(analysis done by Kubicka+Blaha, results revision by Iram GT, report text by Kubicka)

Task given in exercise 3 is to analyze given program and profile it. The program is implementing a mean operator on an image – simply said, it smoothes the image. Basic program flow is like this:

1) Load given image (pgm file format)
2) Allocate memory for result data
3) Process mean operator on input data
4) Store the result

Given library is able to represent the image in the memory in 8 different ways – in our particular case we use no compression and 8-bits per pixel (as grayscale image). In this configuration one byte contain one pixel, stored in the memory bottom-up, and line-by-line.

For the purpose of profiling of this program we created a shell script which measures the application seven times. In order to run the script call `./profile-mean.sh 5 <resultfile>`. First argument specifies kernel size for the mean operator and second argument is a file where should be result stored. For the full script see Appendix C.

Time [%]	Time [s]	No. calls	Time per call	Function name
93.18%	0.41s	7	58.57 ms	`mean(...)`
6.82%	0.03s	7	4.29 ms	`readimage(...)`
0.00%	0.00s	14	0.00ms	`allocimage (...)`
0.00%	0.00s	7	0.00ms	`freeimage(...)`
0.00%	0.00s	7	0.00ms	`writeimage(...)`
0.00%	0.00s	7	0.00ms	`writerawimage(...)`

Table 4 – Non-optimized profiling results

The profiling results of non-optimized program is listed in Table 4. It shows how much time was spent in each function inside the program. Test was taken under following conditions:

- 7 independent executions
- `im2.pgm` picture was used for testing purposes (640x780 / 8-bit / grayscale)
- Kernel of size 5x5 pixels was used
- Testing suite:
  - CPU: Intel Core2 Duo CPU (P8400) 2.4Ghz
  - OS: Debian 6.0.3 (with Xfce) **running in virtual machine**.

We can see that most of the time (93.18%) is spent on calculating mean function.

Cache evaluation

Memory accesses were evaluated by cachegrind tool with default cache parameters. Complete results can be seen in Appendix D. Amount of D1 misses is quite small compared to total amount of memory accesses (0.20%). When talking about absolute numbers, amount of misses of D1 cache is 636,780. To give an example, 636 thousand misses is number 6 times higher than total number of memory accesses to copy 1000x1000 matrix of integers (4B each number).

Mean operator optimization

Algorithm itself can be optimized too, but before optimizing algorithm we decided to try gcc compiler-specific optimizations . They allow us to lower processing time without changing the algorithm. We said to the gcc compiler to optimize as much as he can for better speed results (option `-O3`). Please note that the compiler probably haven't applied loop unrolling techniques, because data set size (number of iterations) is unknown at a compilation time. We have tried to force the compiler to do the loop unrolling anyway (with option `-funroll-all-loops`), but the profiling results were even worse than for original non-optimized program.

Optimized results can be seen in Appendix D. Compiler optimizations lowered number of processed instructions from 474 million to 311 millions (processing time decreased supposedly by 33%). When comes to accessed data, the compiler significantly lowered reading from memory (from 180×10^6 to 107×10^6), although cache misses have not changed much. Actually there was slightly more of D1 cache misses than with non-optimized version. *Note: test was committed on given picture (`im2.pgm`) with kernel size of 5x5 pixels.*

To optimize D1 cache misses, we have rewritten the way in which algorithm accesses data – original algorithm accesses memory column-by-column and that is not effective as we have seen in exercise one. Proposed algorithm accesses data row-by-row – the same way as matrix is stored in the memory. See following Listing 1 for the proposed algorithm.

```
// mean filter with variable kernel size
void mean (struct xvimage *image, struct xvimage *result, int med_size)
{
  int i,j,pi, pj, pos;
  double mean=0;

  for (i=0; i < image->col_size; i++)
    for (j=0; j < image->row_size; j++) {
      pos = 0;
      mean = 0;
      for(pi=max(0,i-(int)(med_size/2));pi<min(image->col_size-1,i+
                  (int)(med_size/2));pi++) {
        for(pj=max(0,j-(int)(med_size/2)); pj<min(image->row_size-1,j+
                  (int)(med_size/2)); pj++)     {
          mean += image->imagedata[ADRESSE(pj,pi)];
          pos += 1;
        }
        result->imagedata[ADRESSE(j,i)] = (int)(mean/pos);
      }
}
```

Listing 1 – proposed mean operator algorithm

Proposed algorithm should significantly lower cache misses because it accesses memory with lesser number of jumps between two distant addresses, which allows the cache memory to work more effectively. From measurement results (see Appendix D) we can see that D1 cache misses have decreased significantly from 635,411 to 16,882. Level 2 cache misses haven't changed at all.

Testing of optimized version

Testing of cache misses and of total executing time on various data set sizes. Data set size is controlled via kernel size, each test was done on the same image (`im2.pgm` – 640x480 / 8 bppx). Test was designed for 8 different kernel sizes – 3x3, 6x6, 9x9, 12x12, 15x15, 18x18, 21x21 and 24x24. A Bash script for automated execution was written, see Appendix E.

Note: results for L2 cache misses were constant – each test shown 11,340 misses on L2 (level 2) cache.

Figure 5 – D1 cache misses and execution time for different kernel sizes

Cache misses time - penalization

Processing of image `im2.pgm` for kernel size of 24x24 took 344.29ms (1,913,465,124 instruction cycles), data were accessed 706,737,165 times from which 731,169 times requested data were not in the D1 cache memory and 11,340 times data were not even in L2d memory. We suppose that accessing data in D1 cache take 1 instruction cycle, accessing data in D2 cache takes 10 instruction cycles and data from main memory are retrieved from in 100 instruction cycles.

Level 1 - Instruction cycles penalization:

$$L1pen = D1\,cache\,misses * (10-1) = 731{,}169 * 9 = 6{,}580{,}520$$

Level 2 – instruction cycles penalization:

$$L2pen = L2d\,cache\,misses * (100 - 1) = 11{,}340 * 99 = 1{,}122{,}660$$

Total instruction cycles penalization:

$$Penalty = L1pen + L2pen = 7{,}703{,}180$$

Total number of instruction cycles without penalty:

$$Cycles\,without\,penalization = Total\,ins.\,cycles - Penalty = 1{,}913{,}465{,}124 - 7{,}703{,}180 = 1{,}905{,}761{,}944$$

Total time without penalty:

$$Time\,without\,penalization = \frac{Cycles\,without\,penalization}{Total\,ins.\,cycles} * Exec.\,time = \frac{1{,}905{,}761{,}944}{1{,}913{,}465{,}124} * 344.29 = 342.90\,ms$$

Time difference:

$$Time\,diff = Exec.\,time - Time\,without\,penalization = 344.29 - 342.90 = 1.29\,ms$$

Time penalization is 1.29ms long waiting of processor for data from memory.

References

[1] *What every programmer should know about memory;* Ulrich Drepper; 10/2007 (available online at http://www.akkadia.org/drepper/cpumemory.pdf)

Appendix A – Exercise 1 – Measurement results

Case 1 (column-by-column)

```
pc5103e:~> valgrind --tool=cachegrind ./arch-tp2ex1 1000
==18859== Cachegrind, a cache and branch-prediction profiler
==18859== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==18859== Using Valgrind-3.6.0.SVN-Debian and LibVEX;
==18859== Command: ./arch-tp2ex1-2 1000
==18859==
==18859==
==18859== I   refs:      26,196,356
==18859== I1  misses:           685
==18859== L2i misses:           683
==18859== I1  miss rate:       0.00%
==18859== L2i miss rate:       0.00%
==18859==
==18859== D   refs:      18,099,039  (16,072,783 rd   + 2,026,256 wr)
==18859== D1  misses:     2,064,475  ( 1,001,772 rd   + 1,062,703 wr)
==18859== L2d misses:       148,327  (    22,139 rd   +   126,188 wr)
==18859== D1  miss rate:       11.4% (       6.2%     +      52.4%  )
==18859== L2d miss rate:        0.8% (       0.1%     +       6.2%  )
==18859==
==18859== L2 refs:        2,065,160  ( 1,002,457 rd   + 1,062,703 wr)
==18859== L2 misses:        149,010  (    22,822 rd   +   126,188 wr)
==18859== L2 miss rate:         0.3% (       0.0%     +       6.2%  )
```

Case 2: (line-by-line)

```
pc5103e:~> valgrind --tool=cachegrind ./arch-tp2ex1-2 1000

==18856== Cachegrind, a cache and branch-prediction profiler
==18856== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==18856== Using Valgrind-3.6.0.SVN-Debian and LibVEX;
==18856== Command: ./arch-tp2ex1 1000
==18856==
==18856==
==18856== I   refs:      202,494
==18856== I1  misses:        697
==18856== L2i misses:        695
==18856== I1  miss rate:    0.34%
==18856== L2i miss rate:    0.34%
==18856==
==18856== D   refs:      106,128  (78,846 rd   + 27,282 wr)
==18856== D1  misses:      2,090  ( 1,767 rd   +    323 wr)
==18856== L2d misses:      1,729  ( 1,433 rd   +    296 wr)
==18856== D1  miss rate:     1.9% (   2.2%     +    1.1%  )
==18856== L2d miss rate:     1.6% (   1.8%     +    1.0%  )
==18856==
==18856== L2 refs:         2,787  ( 2,464 rd   +    323 wr)
==18856== L2 misses:       2,424  ( 2,128 rd   +    296 wr)
==18856== L2 miss rate:      0.7% (   0.7%     +    1.0%  )
```

Case 3 (memcpy)

```
pc5103e:~> valgrind --tool=cachegrind ./arch-tp2ex1-3 1000
==18862== Cachegrind, a cache and branch-prediction profiler
==18862== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==18862== Using Valgrind-3.6.0.SVN-Debian and LibVEX;
==18862== Command: ./arch-tp2ex1-3 1000
==18862==
==18862==
==18862== I   refs:      190,644
==18862== I1  misses:        705
==18862== L2i misses:        702
==18862== I1  miss rate:    0.36%
==18862== L2i miss rate:    0.36%
==18862==
==18862== D   refs:       95,608  (68,759 rd   + 26,849 wr)
==18862== D1  misses:      2,074  ( 1,799 rd   +    275 wr)
==18862== L2d misses:      1,685  ( 1,435 rd   +    250 wr)
==18862== D1  miss rate:     2.1% (   2.6%     +    1.0%  )
==18862== L2d miss rate:     1.7% (   2.0%     +    0.9%  )
==18862==
==18862== L2 refs:         2,779  ( 2,504 rd   +    275 wr)
==18862== L2 misses:       2,387  ( 2,137 rd   +    250 wr)
==18862== L2 miss rate:      0.8% (   0.8%     +    0.9%  )
```

Appendix B – Exercise 2 – Cache measurement script

```
valgrind --tool=cachegrind --D1=1024,1,16 $1 $2
valgrind --tool=cachegrind --D1=2048,1,16 $1 $2
valgrind --tool=cachegrind --D1=4096,1,16 $1 $2
valgrind --tool=cachegrind --D1=8192,1,16 $1 $2
valgrind --tool=cachegrind --D1=16384,1,16 $1 $2
valgrind --tool=cachegrind --D1=32768,1,16 $1 $2
valgrind --tool=cachegrind --D1=65536,1,16 $1 $2
valgrind --tool=cachegrind --D1=131072,1,16 $1 $2

valgrind --tool=cachegrind --D1=1024,2,16 $1 $2
valgrind --tool=cachegrind --D1=2048,2,16 $1 $2
valgrind --tool=cachegrind --D1=4096,2,16 $1 $2
valgrind --tool=cachegrind --D1=8192,2,16 $1 $2
valgrind --tool=cachegrind --D1=16384,2,16 $1 $2
valgrind --tool=cachegrind --D1=32768,2,16 $1 $2
valgrind --tool=cachegrind --D1=65536,2,16 $1 $2
valgrind --tool=cachegrind --D1=131072,2,16 $1 $2

valgrind --tool=cachegrind --D1=1024,4,16 $1 $2
valgrind --tool=cachegrind --D1=2048,4,16 $1 $2
valgrind --tool=cachegrind --D1=4096,4,16 $1 $2
valgrind --tool=cachegrind --D1=8192,4,16 $1 $2
valgrind --tool=cachegrind --D1=16384,4,16 $1 $2
valgrind --tool=cachegrind --D1=32768,4,16 $1 $2
valgrind --tool=cachegrind --D1=65536,4,16 $1 $2
valgrind --tool=cachegrind --D1=131072,4,16 $1 $2

valgrind --tool=cachegrind --D1=1024,8,16 $1 $2
valgrind --tool=cachegrind --D1=2048,8,16 $1 $2
valgrind --tool=cachegrind --D1=4096,8,16 $1 $2
valgrind --tool=cachegrind --D1=8192,8,16 $1 $2
valgrind --tool=cachegrind --D1=16384,8,16 $1 $2
valgrind --tool=cachegrind --D1=32768,8,16 $1 $2
valgrind --tool=cachegrind --D1=65536,8,16 $1 $2
valgrind --tool=cachegrind --D1=131072,8,16 $1 $2
```

Appendix C – Exercise 3 - Profiling script

```
rm a.out
gcc -pg function.c main.c mccodimage.c mcimage.c
rm gmon.sum
mv gmon.out gmon.sum

./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum
./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum
 ./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum
./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum
./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum
 ./a.out im2.pgm $1
gprof -s a.out gmon.out gmon.sum

gprof a.out gmon.sum > $2
```

Appendix D – Exercise 3 – cachegrind results

Non-optimized version

```
root@debian:/home/matej/Downloads/tp-cache# gcc *.c -o a.out
root@debian:/home/matej/Downloads/tp-cache# valgrind --tool=cachegrind ./a.out
im2.pgm 5
==25659== Cachegrind, a cache and branch-prediction profiler
==25659== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==25659== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright
info
==25659== Command: ./a.out im2.pgm 5
==25659==
==25659==
==25659== I   refs:      474,947,531
==25659== I1  misses:          1,165
==25659== L2i misses:          1,137
==25659== I1  miss rate:        0.00%
==25659== L2i miss rate:        0.00%
==25659==
==25659== D   refs:      239,747,859  (182,126,268 rd   + 57,621,591 wr)
==25659== D1  misses:        636,780  (    323,597 rd   +    313,183 wr)
==25659== L2d misses:         12,120  (      1,567 rd   +     10,553 wr)
==25659== D1  miss rate:         0.2% (        0.1%     +        0.5%  )
==25659== L2d miss rate:         0.0% (        0.0%     +        0.0%  )
==25659==
==25659== L2 refs:           637,945  (    324,762 rd   +    313,183 wr)
==25659== L2 misses:          13,257  (      2,704 rd   +     10,553 wr)
==25659== L2 miss rate:          0.0% (        0.0%     +        0.0%  )
Profiling timer expired
```

Optimized version (-O3)

```
root@debian:/home/matej/Downloads/tp-cache# gcc -O3 *.c -o a.out
root@debian:/home/matej/Downloads/tp-cache# valgrind --tool=cachegrind ./a.out
im2.pgm 5
==3936== Cachegrind, a cache and branch-prediction profiler
==3936== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==3936== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright
==3936== Command: ./a.out im2.pgm 5
==3936==
==3936==
==3936== I   refs:      311,374,384
==3936== I1  misses:          1,108
==3936== L2i misses:          1,094
==3936== I1  miss rate:        0.00%
==3936== L2i miss rate:        0.00%
==3936==
==3936== D   refs:      160,568,259  (107,849,140 rd   + 52,719,119 wr)
==3936== D1  misses:        635,411  (    323,088 rd   +    312,323 wr)
==3936== L2d misses:         11,341  (      1,541 rd   +      9,800 wr)
==3936== D1  miss rate:         0.3% (        0.2%     +        0.5%  )
==3936== L2d miss rate:         0.0% (        0.0%     +        0.0%  )
==3936==
==3936== L2 refs:           636,519  (    324,196 rd   +    312,323 wr)
==3936== L2 misses:          12,435  (      2,635 rd   +      9,800 wr)
==3936== L2 miss rate:          0.0% (        0.0%     +        0.0%  )
```

Optimized version (-O3) with newly proposed row-by-row algorithm

```
root@debian:/home/matej/Downloads/exercise3# gcc -O3 *.c
root@debian:/home/matej/Downloads/exercise3# valgrind --tool=cachegrind ./a.out
im2.pgm im3.pgm 5
==18006== Cachegrind, a cache and branch-prediction profiler
==18006== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote et al.
==18006== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright
info
==18006== Command: ./a.out im2.pgm im3.pgm 5
==18006==
==18006==
==18006== I   refs:      305,548,171
==18006== I1  misses:          1,103
==18006== L2i misses:          1,089
==18006== I1  miss rate:        0.00%
==18006== L2i miss rate:        0.00%
==18006==
==18006== D   refs:      159,644,647  (106,926,350 rd   + 52,718,297 wr)
==18006== D1  misses:         16,882  (      6,960 rd   +      9,922 wr)
==18006== L2d misses:         11,342  (      1,543 rd   +      9,799 wr)
==18006== D1  miss rate:         0.0% (        0.0%     +        0.0%  )
==18006== L2d miss rate:         0.0% (        0.0%     +        0.0%  )
==18006==
==18006== L2 refs:            17,985  (      8,063 rd   +      9,922 wr)
==18006== L2 misses:          12,431  (      2,632 rd   +      9,799 wr)
==18006== L2 miss rate:          0.0% (        0.0%     +        0.0%  )
```

Appendix E – Exercise 3 - Automation script

```
rm mean-opt.out
gcc -O3 function.c  main.c  mccodimage.c  mcimage.c -o mean-opt.out

mkdir mean-prof-results

# cache measurement
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 1 2>&1 | tee mean-prof-
results/mean-opt.cache1.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 3 2>&1 | tee mean-prof-
results/mean-opt.cache3.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 6 2>&1 | tee mean-prof-
results/mean-opt.cache6.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 9 2>&1 | tee mean-prof-
results/mean-opt.cache9.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 12 2>&1 | tee mean-prof-
results/mean-opt.cache12.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 15 2>&1 | tee mean-prof-
results/mean-opt.cache15.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 18 2>&1 | tee mean-prof-
results/mean-opt.cache18.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 21 2>&1 | tee mean-prof-
results/mean-opt.cache21.log
valgrind --tool=cachegrind ./mean-opt.out im2.pgm 24 2>&1 | tee mean-prof-
results/mean-opt.cache24.log

# calculate running times
./profile-mean.sh 1 mean-prof-results/mean-opt.1.result
./profile-mean.sh 3 mean-prof-results/mean-opt.3.result
./profile-mean.sh 6 mean-prof-results/mean-opt.6.result
./profile-mean.sh 9 mean-prof-results/mean-opt.9.result
./profile-mean.sh 12 mean-prof-results/mean-opt.12.result
./profile-mean.sh 15 mean-prof-results/mean-opt.15.result
./profile-mean.sh 18 mean-prof-results/mean-opt.18.result
./profile-mean.sh 21 mean-prof-results/mean-opt.21.result
./profile-mean.sh 24 mean-prof-results/mean-opt.24.result

rm cachegrind.out.*
```

Appendix F – Exercise 1 – Test line-by-line

```
#include "stdio.h"
#include "stdlib.h"

void main(int argc, unsigned char ** argv) {
  int *a, *b;
  int noel = atoi(argv[1])^2;

  // Get arr
  a = calloc(noel,sizeof(int));
  b = malloc(noel*sizeof(int));

  // Copy the matrix A to B
  int *pb = b, *pa, *pc = (a+noel);
  for(pa=a;pa<pc;pa++) { pb++; *pb = *pa; }

}
```

Appendix G – Exercise 1 – Test column-by-column

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

void main(int argc, unsigned char ** argv) {
  int *a, *b;
  int noel = atoi(argv[1]);

  // Get arr
  a = malloc(noel*noel*sizeof(int));
  b = malloc(noel*noel*sizeof(int));

  //printf("Filling with zeros... \n");
  // Fill the A matrix with zeros
  int *p = a, *pc = (a + noel*noel);
  for(;p<pc;p++) *p = 0;

  //printf("Copying the stuff manually... \n");
  // Copy the matrix A to B
  int x,y;
  for(x=0;x<noel;x++)
    for(y=0;y<noel;y++)
       *(b + y*noel+x) = *(a + y*noel+x);
}
```

Appendix H – Exercise 1 – Test with memcpy()

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

void main(int argc, unsigned char ** argv) {
  int *a, *b;
  int noel = atoi(argv[1])^2;

  // Get arr
  a = calloc(noel,sizeof(int));
  b = malloc(noel*sizeof(int));

  // Copy the matrix A to B
  memcpy(b,a,noel);

}
```