Roofline and Matrix Multiplication PAPI Analysis

Brito, Rui
PG22781
Department of Informatics
University of Minho
ruibrito666@gmail.com

Alves, José
PG22765
Department of Informatics
University of Minho
zealves.080@gmail.com

Abstract

This paper describes the roofline[1] analysis of two different machines. The roofline consists in a theoretical performance model that provides visual insights on floating point computing performance. This model relates processor performance to off-chip memory traffic, giving a upper bound on performance of a kernel depending on its operational intensity. There is also a study of a matrix dot product using PAPI(Performance Application Programming Interface)[2]. With PAPI we are able to use the low-level hardware counters to identify the most accessed areas of our memory hierarchy as well as the operations made by running software.

1 Introduction

The importance of a system's characterization is due to the need of understanding the limitations of a system, in order to make the necessary optimizations to obtain the maximum performance of a system. In order to provide a easy-to-understand performance model for floating-point programs, the Roofline Model was created. The Roofline indentifies a theoretical upper bound on performance on a kernel that relates if a certain opetional intensity is memory-bound, dependant on accesses of memory, or computational-bound, dependant on floating-point operational units. To comprehend the influence of different components and certain architecture characteristics a number of ceilings, both computational and memory driven, may be added. Since the Roofline Model is better understood with a case study, a matrix multiplication problem was proposed to present the theoretical limitations of the algorithm through the model. To specify the problem in the Roofline model, a series of values were obtained from the hardware counters through PAPI(Performance Application Programming Interface). PAPI allow us to reach the low-level hardware counters and obtain information from components accessed by the software running.

2 Roofline

2.1 Machine Profiles

The machines used for this study were an Apple MacBook Pro late 2008 and a 2010 HP dv6-2190ep. Information about the machines was gathered from various *NIX tools (e.g */proc/cpuinfo*, */proc/meminfo*, *dmidecode* and *sysctl*) and from the web (e.g. *Intel Ark* and *Crucial*). To calculate cache and main memory bandwidth we used the tool *bandwidth* [1].

2.1.1 Performance Peaks

In order to calculate the rooflines, we needed the Floating-Point(FP) Performance Peak and the Memory Bandwidth's Peak. To attain the FP Performance Peak we solve the following formula:

$$\text{GFlop/s}_{\max} = \#_{\text{cores}} \times f_{\text{clock}} \times \#_{\text{SIMD}}$$

MacBook Pro FP Performance Peak:

$$\text{GFlop/s}_{\max} = 2 \times 2.8 \times 8 = 44.8 GFLOPSs$$

HP Pavillion FP Performance Peak:

$$\text{GFlop/s}_{\max} = 4 \times 1.6 \times 8 = 51.2 GFLOPSs$$

To calculate de Memory Bandwidth Peak we solve the following formula:

$$\text{BW}_{\max} = \#_{\text{channels}} \times mem_{\text{clock}} \times bus_{\text{bandwidth}}$$

[1] http://zsmith.co/bandwidth.html

MacBook Pro Memory Bandwidth Peak:

$$\text{GFlop/s}_{\max} = 2 \times 1067 \times 64 = 17.072 GBbyte$$

HP Pavillion Memory Bandwidth Peak:

$$\text{GFlop/s}_{\max} = 2 \times 1333 \times 64 = 21.328 GBbyte$$

2.1.2 Specifications

The specifications for the MacBook Pro are displayed on Table 1.
It should be noted that L1 Access Bandwidth refers to 128-bit sequential reads

3 sdjfdsfhsdk

Manufacter:	Apple
Model:	MacBook Pro late 2008
Processor	
Manufacturer:	Intel
Arch:	Core
Model:	Core 2 Duo T9600
Cores:	2
Clock Frequency:	2.80 GHz
FP Performance's Peak:	44.8 GFlops/s
Cache	
Level:	1
Size:	32KB + 32KB
Line Size:	64 B
Associative:	8-way
L1 Access Bandwidth:	40 GB/s
Level:	2
Size:	6 MB
Line Size:	64 B
Associative:	24-way
RAM	
Type:	SDRAM DDR3 PC3-8500
Frequency:	1067 MHz
Size:	4 GB
Num. Channels:	2
Latency:	13.13 ns

Table 1: MacBook Pro late 2008 specifications

The specifications for the HP dv6-2190ep are displayed on Table 2.

Manufacter:	HP
Model:	Pavillion dv6-2190ep
Processor	
Manufacturer:	Intel
Arch:	Nehalem
Model:	i7-720QM
Cores:	4
Clock Frequency:	1.60 GHz
FP Performance's Peak:	51.2 GFlops/s
Cache	
Level:	1
Size:	32KB + 32KB
Line Size:	64 B
Associative:	4/8-way
L1 Access Bandwidth:	22 GB/s
Level:	2
Size:	256 KB
Line Size:	64 B
Associative:	8-way
Level:	3
Size:	6 MB
Line Size:	64 B
Associative:	12-way
RAM	
Type:	SDRAM DDR3 PC3-10600
Frequency:	1333 MHz
Size:	4GB
Num. Channels:	2
Latency:	13.5 ns

Table 2: HP Pavillion dv6-2190ep specifications

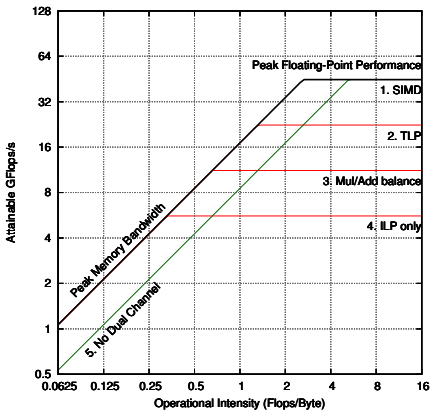

Figure 1: Mackbook Pro late 2008

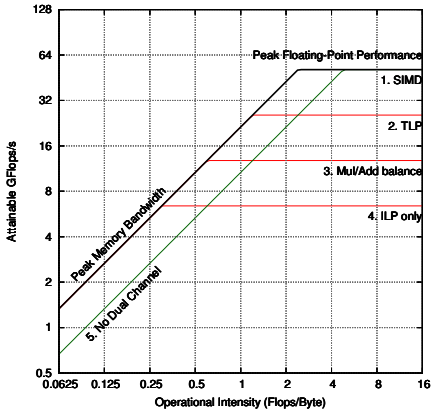

Figure 2: HP Pavillion dv6-2190ep Roofline

3.1 Roofline Model

3.1.1 MacBook Pro Roofline

3.1.2 HP Pavillion Roofline

4 PAPI Case Study

4.1 Problem

The case study of this report, is to analyse the performance of a **matrix dot product** algorithm,

$$MatrixA * MatrixB = MatrixC \tag{1}$$

wich contains a triple nested loop with the indexes i,j and k(line,column and position). A naive implementation of this algorithm will provide, at best, very weak performance, since for every iteration of the j loop the whole current line of the matrix will be brought to cache (if it fits), however, since the j loop sweeps columns, this proves to be very inefficient. Our aim was to minimize this inefficiency.

4.2 Algorithm Analysis

The implementation produced to calculate the matrix multiplication was coded in C and compiled with Optimization level 3 (-O3, so the compiler can explore SIMD extensions). The naive algorithm of matrix multiplication is presented here, in order to better understand the problem at hand.

```
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        for(k = 0; k < size; k++) {
                acc += matrixA[i][k] *
                          matrixB[k][j];
            }
            matrixC[i][j] = acc;
            acc = 0;
        }
    }
```

And, for completeness' sake, here's the optimized version. Note that the matrix *tMatrix* is transposed so we can take advantage of a unit stride.

```
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        for(k = 0; k < size; k++) {
            acc += matrixA[i][k] *
                        tMatrix[j][k];
        }
        matrixC[i][j] = acc;
        acc = 0;
    }
}
```

Two versions of the program were run, one with the original matrixB and other where matrixB is transposed. With this second version, it is expected for the algorithm to access a continous memory space, thus leveraging unit stride, increasing overall performance while reducing memory accesses (due to the reduced miss rate).

4.3 Tests

4.3.1 Methodology

To measure the algorithm's perfomance, hardware counters were used. To gather information from these counters we used *PAPI*(Performance API). This tool alowed us to measure (among others) the following counters:

PAPI _TOT_CYC Total number of cycles;

PAPI _TOT_INS Instructions completed;

PAPI _LD_INS number of load instructions;

PAPI _SR_INS number of store instructions;

PAPI _FP_OPS Floating point operations;

PAPI _FP_INS Floating point instructions;

PAPI _L1_DCA L1 data cache accesses;

PAPI _L1_DCM L1 data cache misses;

PAPI _L2_DCA L2 data cache accesses;

PAPI _L2_DCM L2 data cache misses;

PAPI _L3_DCA L3 data cache accesses;

Moreover, all tests were run on a dedicated execution of the algorithm with process niceness set to -20 (*nice -n -20*) to ensure that essential machine time wasn't being spent on some other task. Also, to minimize overhead, OS Widgets and network were disabled. To decrease the chance of human error, the execution of all tests was "commanded" by a bash script. This script was also responsible for checking the 5% error margin, and, in case it wasn't satisfied, re-running the tests.

4.3.2 Test Cases

All four tests, presented below, were chosen to run in the two different version(normal and transpose). Each test was run four times, with the best execution time being selected within a margin of 5%.

Memory	Size	Matrix Size
L1	30 KB	50
L2	255 KB	146
L3	3 MB	500
RAM	7.68 MB	800

Table 3: Test cases

4.4 Results

4.4.1 Cache Analysis

To measure the data cache misses the counters PAPI_L1_DCM and PAPI_L2_DCM were used. Each test fits in a different memory hierarchy level(L1, L2, L3 and RAM). (Note that test suffixed with a 0 mean unoptimized version, while those suffixed with a 1 mean, optimized version).

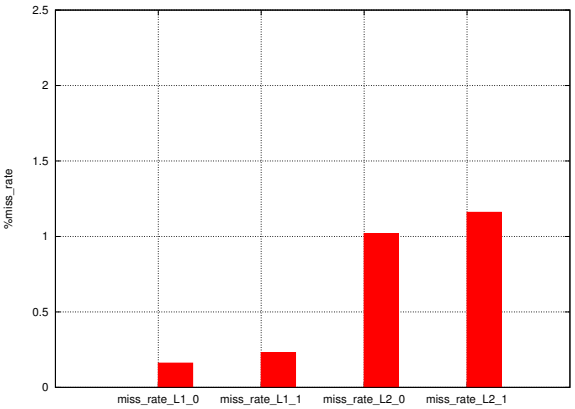

Figure 3: Percentage of Cache Misses

The above graphic shows an increase of percentage of misses with the optimized version, they are misleading. The next graph shows that although the percentage of misses increased, the total of misses didn't because the number of accesses also droped.

Usage of both levels of cache was estimated with specific counters. PAPI_L1_DCA and PAPI_L2_DCA provided the number of data accesses to the caches.

Before the results were out, it was expected a decrease of cache accesses from version one to version two of the algorithm.

As we can see, the number of access to cache drops significantly from the first version to the second version

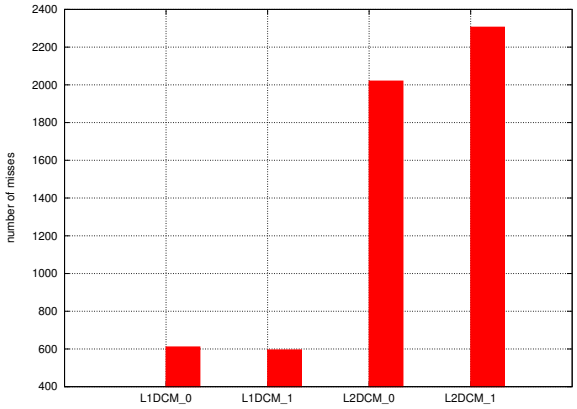

Figure 4: Number of Cache Misses

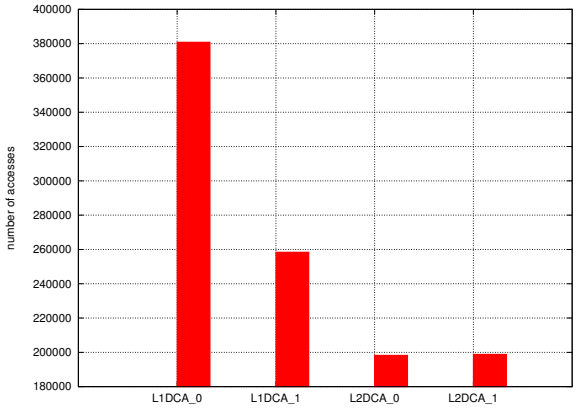

Figure 5: Number of Memory Accesses

while running with the L1 Cache Test. Though in the second test, the L2 Cache, the number of accesses slightly increased.

For more palpable values, the following table shows the execution times.

Test	Time (μs)
L1_0	271.091
L1_1	254.000
L2_0	6983.450
L2_1	6629.000
L3_0	521429.0
L3_1	260535.0
RAM_0	6.60003e+06
RAM_1	3 1.08117e+06

Table 4: Execution Times

As it can be seen, the best improvements are seen in cache level 3 and in the main memory.

4.5 Going Even Further

The transposed matrix optimization, while showing improvements, it still is far away from the desired peak performance. Apart from enabling and/or taking advantage of all the features shown in the roofline model, implementing the algorithm in a way so that it uses blocks, would dramatically improve performace[3]. But that isn't this paper's goal, we just wanted to let the readers know that there is still much to improve on.

References

[1] `Roofline: An insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures`
Samuel Webb Williams, Andre Waterman, David A. Patterson
23th November 2012

[2] `Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters`
Jack Dongarra, Kevin London, Shirley Moore, Philip Mucci, Daniel Terpstra, Haihang You, Min Zhou

[3] `Computer Architecture: A Quantitative Aproach, 5th Ed.`
John L. Hennesy, David A. Patterson

[4] `http://ark.intel.com/`

[5] `http://www.crucial.com`

[6] `http://www.wikipedia.com`

Appendices

A CUDA Implementation

Furthermore, an implementation of this algorithm was done using the GPU. We used the MacBook Pro referred earlier which has an NVIDIA GeForce 9600M GT with a compute capability of 1.1. The GPU kernel is has follows:

```
__global__ void dotProduct( float* matrixA, float* matrixB,
                            float* matrixC, int n) {

   int j = blockIdx.x * blockDim.x + threadIdx.x;
   int i = blockIdx.y * blockDim.y + threadIdx.y;

   float value = 0;
   int k;
   for (k = 0; k < n; k++) {
      value += matrixA[i * n + k] * matrixB[k * n + j];
   }

   matrixC[i * n + j] = value;
}
```

To measure it's performance we used the NVIDIA command line profiler which read hardware counters from the GPU. Table 5 shows those results for a matrix of size (20x20). Using NVIDIA's own *matrixMulCUBLAS*, we got the peak performance for this particular GPU which is 14.48 Gflops. With our naive algorithm we achieved 3.44 Gflops. There should probably be another Roofline taking the GPU into account, with ceilings such as SW Prefetching, Shared Memory, Memory Coalescing, NUMA allocation and usage, etc...

method	gputime	cputime	instructions
memcpyHtoD	6.912	26.669	-
memcpyHtoD	6.048	17.427	-
dotProduct	64.960	94.172	2237
memcpyDtoH	5.920	42.166	-

Table 5: Cuda Profile

B Images

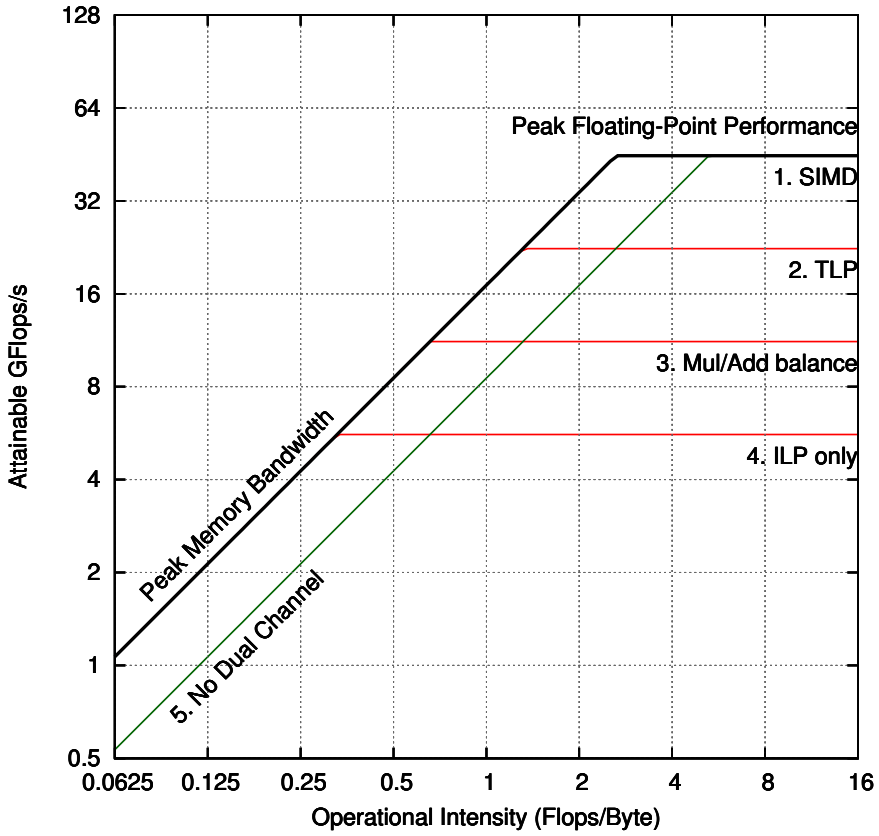

Figure 6: Mackbook Pro late 2008

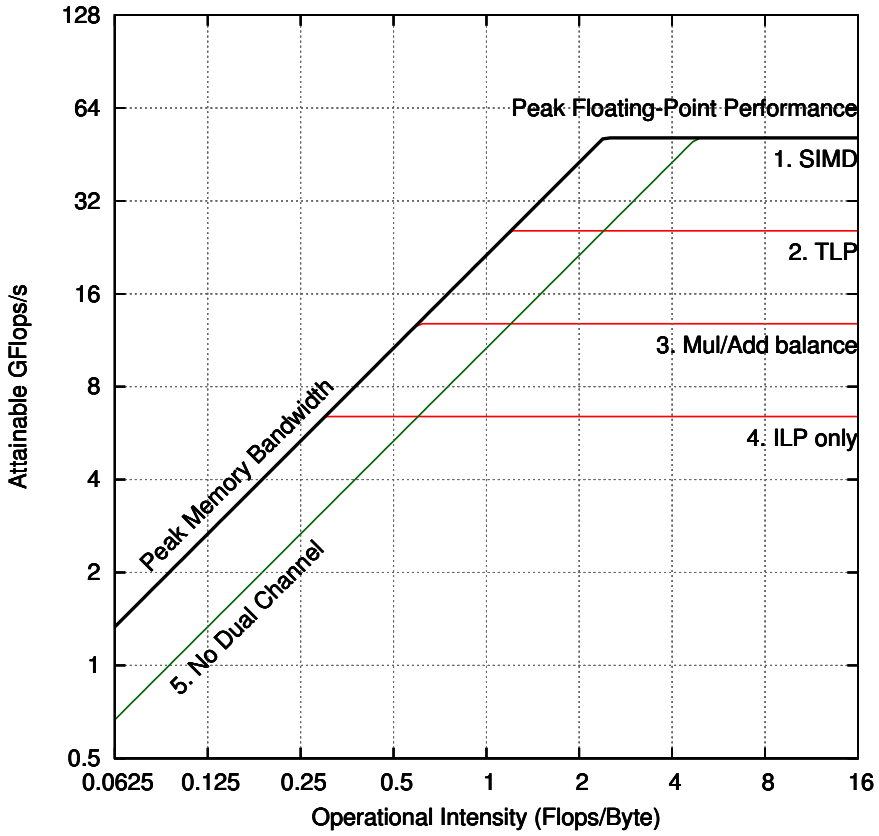

Figure 7: HP Pavillion dv6-2190ep Roofline

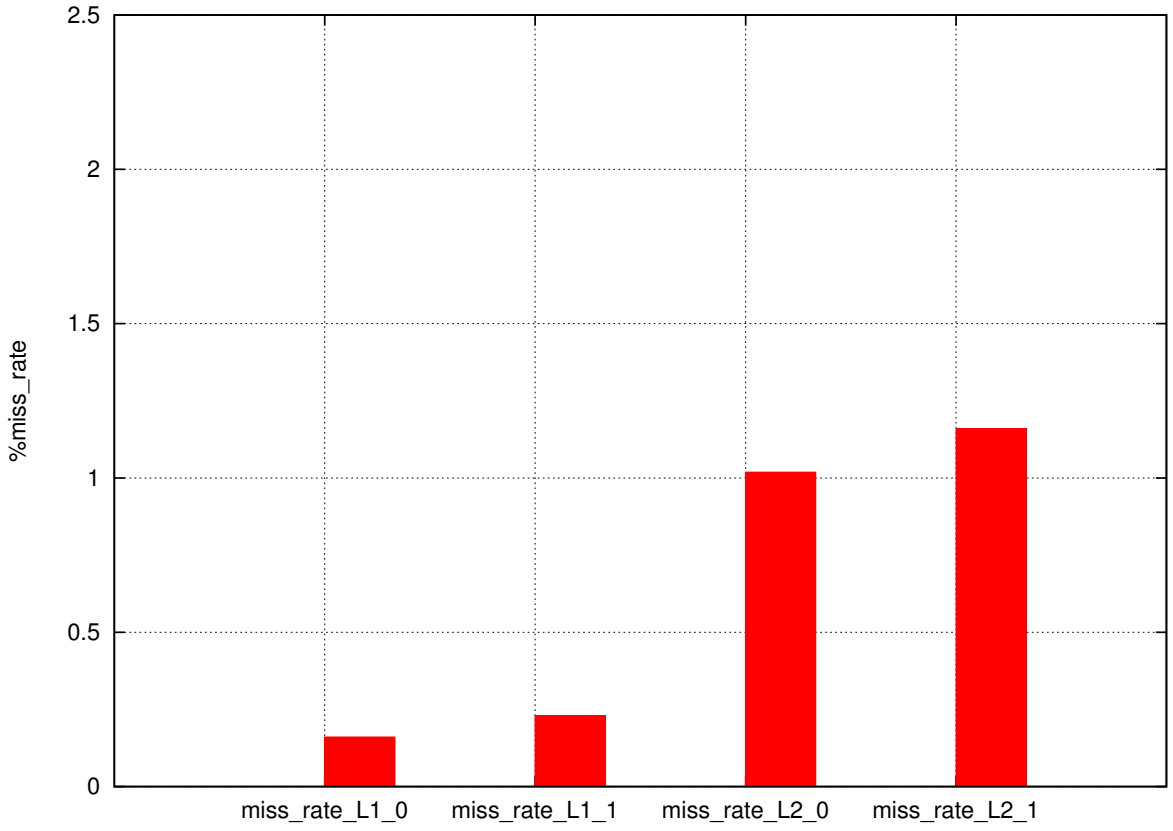

Figure 8: Percentage of Cache Misses

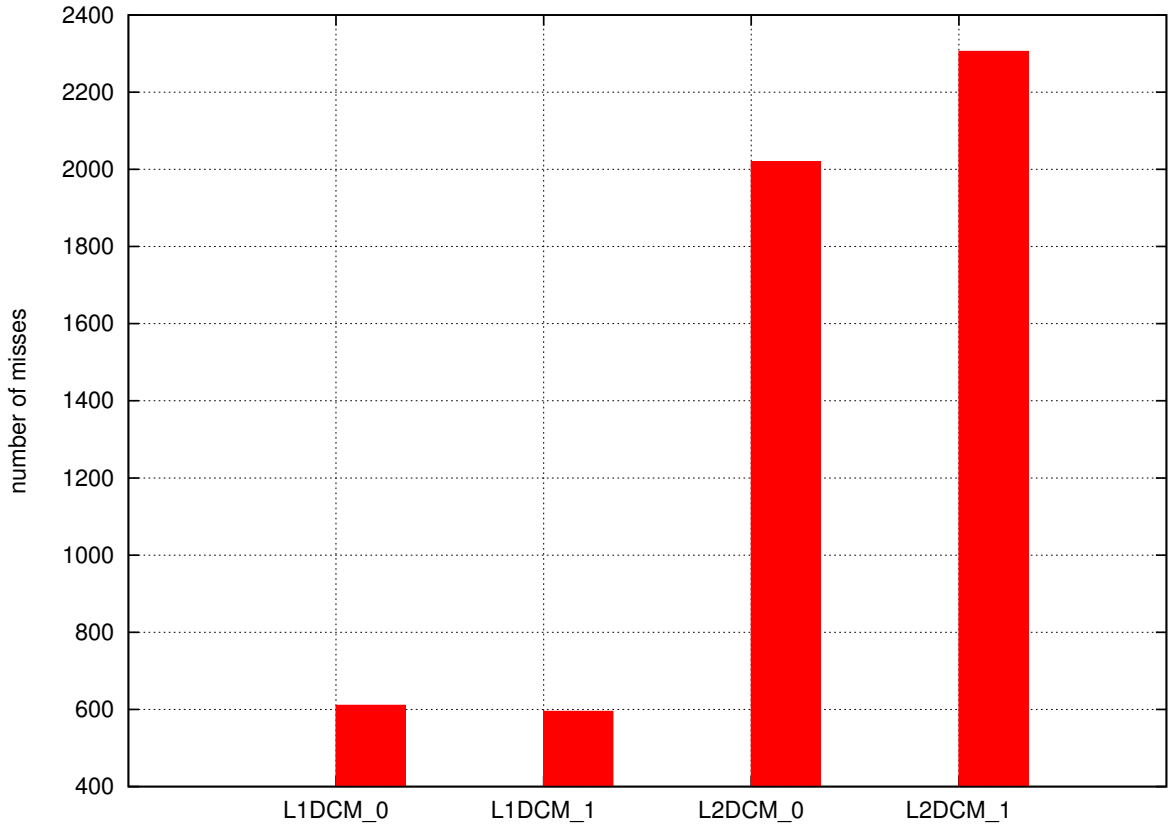

Figure 9: Number of Cache Misses

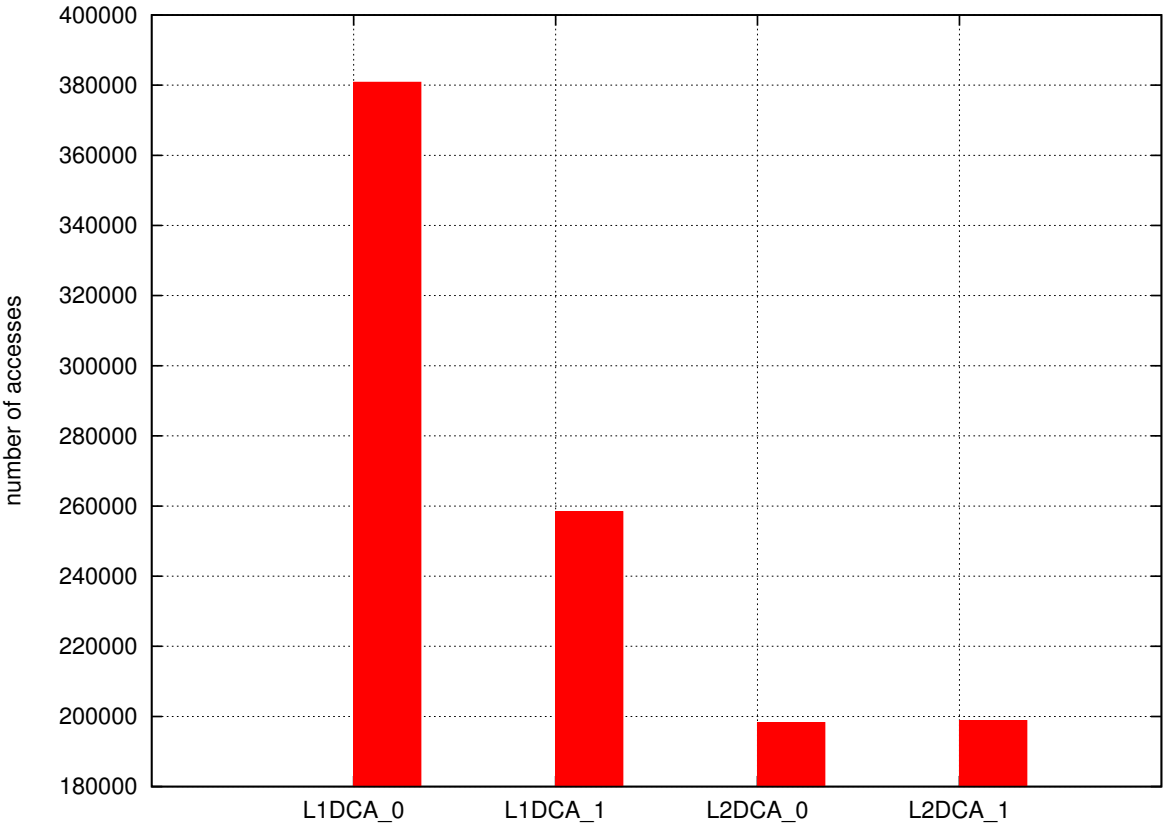

Figure 10: Number of Memory Accesses