INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

ORGANIZAÇÃO DE COMPUTADORES

LEIC

# Second Lab Assignment: System Modeling and Proﬁling

Version 1.1.0

2021/2022

## Introduction

The goal of this assignment is twofold: (i) to determine the characteristics of a computer’s caches, and

(ii) to leverage the obtained knowledge about the caches in order to optimize the performance of a given program. For this task, the students will make use of a performance analysis tool to have direct access to hardware performance counters available on most modern microprocessors. The tool that will be used is the standard Application Programming Interface (API): PAPI [1].

In the rest of this section, we make a brief introduction to PAPI, and describe the targeted computer platform and the development environment. In Section 3, we describe the procedure for modeling the L1 and L2 caches of the targeted platform (Subsection 3.1), and provide a guide for analyzing the perfor- mance of a matrix-multiply code segment and optimizing it based on the characteristics of the L2 cache of the target architecture (Subsection 3.2).

### Targeted Platform and Development Environment

**IMPORTANT: This assignment must be performed on the computers of your lab classes room.** These computers have similar hardware characteristics, and any of them can be used as a target platform. Note that, since this work is hardware-dependent, conducting it on an computer with different hardware characteristics could produce unexpected results, and hence invalidating your work. This means you should always use the same lab. If you are an Alameda student, you can access the speciﬁc lab computer you want (see https://welcome.rnl.tecnico.ulisboa.pt/#labs-access).

To properly setup the development environment, it is necessary to obtain the PAPI library and a set of auxiliary program ﬁles. This material can be found in the package lab2\_kit.zip, which can be downloaded from the course website. After downloading and uncompressing this package on any of the lab classes’ computers, PAPI must be built. To this end, change directories to the location of the PAPI source code: folder papi-X.X.X/src. Compile the code by issuing the commands: ./configure, and make. This operation will produce a set of helper tools located in directory src/utils/ and create the PAPI library papilib.a. The tool papi\_avail, in particular, is useful to determine the PAPI events supported on the target platform. The library will be linked to the auxiliary programs presented in the following sections.

## Exercise

To help determining the characteristics of the labs computer’s caches, the following exercises will help you estimate cache parameters from small C applications.

The ﬁrst step to get acquainted with the procedure is to determine only the size of the cache using a small C application on a (known) machine, such as the code you have analyzed on lab exercise VI.3. This C code, is a simpliﬁed version of the following programs in this assignment. Basically, it iterates over an array to determine the cache size.

To guarantee that you measure the time accurately, please use the source code available in the labkit (ﬁle spark.c).

In order to perform the evaluation you should go to your lab in order to access the cache size by running the application there. You may want to repeat the evaluation of the elapsed time a few times to achieve statistical signiﬁcance. You should table the relevant results for different cache sizes on the response sheet and make a conclusion regarding the cache size. You can calculate more measures before the output, examine the ﬁnal part of the source code ﬁle.

1. What is the cache capacity of the computer you tested? Please justify.

To discover the other cache parameters, you’re going to modify the C application, so that it generates different data access patterns. Please spend a few minutes analyzing the modiﬁcations to the source code.

for(size\_t cache\_size = CACHE\_MIN; cache\_size < CACHE\_MAX; cache\_size = 2\*cache\_size) { for(size\_t stride = 1; stride <= cache\_size/2; stride = 2\*stride){

limit = cache\_size - stride + 1;

for(ssize\_t i = 10 \* stride; i > 0; i--) {

for(index = 0; index < limit; index += stride) {

array[index] = array[index] + 1;

}

}

}

}

The meaning of each variable is the following:

**array[]** an arbitrary large array that will be repeatedly accessed to measure the cache miss pattern;

**cache\_size** value of the cache size under test; all cache sizes given by integer powers of 2, between

CACHE\_MIN = 8kB and CACHE\_MAX = 64kB should be considered;

**stride** states how many entries are being skipped at each access; for example, if the stride is 4, entries 0, 4, 8, 12, ... in the array are being accessed, while entries 1, 2, 3, 5, 6, 7, 9, 10, 11, ... are skipped;

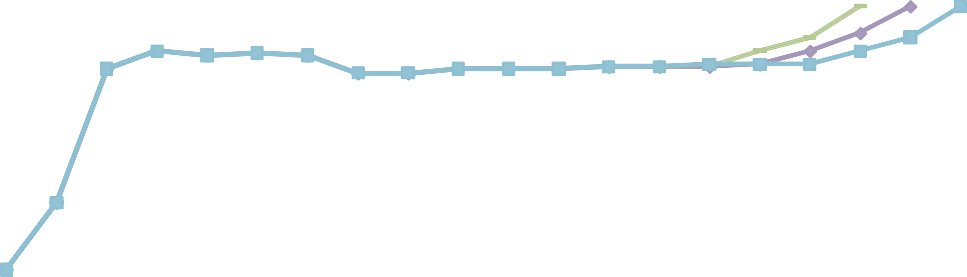
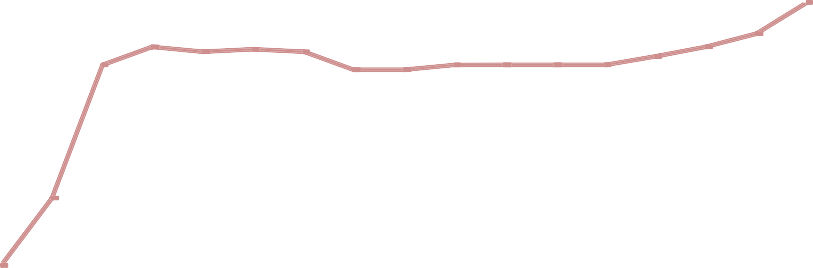
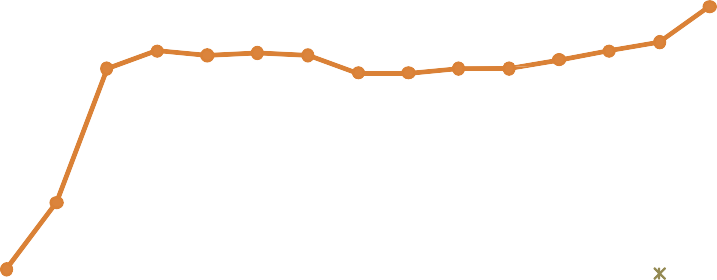
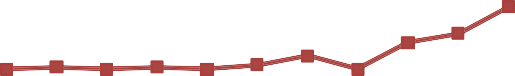
**limit** the largest address that will be accessed for the cache size and access pattern under test;

**repeat** denotes the number of times that each access pattern will be repeated in array.

The execution time for this code segment on this machine yield the chart depicted in Figure 1, by varying the adopted value for the *stride* parameter and for different array sizes, deﬁned between ARRAY\_MIN = 4kB and ARRAY\_MAX = 4MB.

1. What is the cache capacity of the computer?
2. What is the size of each cache block?
3. What is the L1 cache miss penalty time?

1100



1000

900

800

Time for read + write (ns)

700

600

500

400

300

200



Stride

4k 8k 16k 32k 64k 128k 256k 512k 1M 2M 4M

**Figure 1:** Variation of the cache access time with the adopted *stride* value for different array sizes.

## Procedure

### Modeling Computer Caches

In the ﬁrst part of this assignment, the goal is to model the characteristics of the L1 data cache and L2 cache of the targeted computer platform. Next, we provide instructions for performing this analysis.

Use the forms at the end to answer the questions below.

#### Modeling the L1 Data Cache

The methodology to experimentally model the L1 data cache consists in considering the total amount of data cache misses during the execution of the following code sequence of program cm1.c, similar to the program in Section 2. This program can be found in the package lab2\_kit.zip.

for(array\_size=ARRAY\_MIN; array\_size < ARRAY\_MAX; array\_size=array\_size\*2) for(stride=1; stride <= array\_size/2; stride=stride\*2){

limit = array\_size - stride + 1;

for(repeat=0; repeat<=200\*stride; repeat++) for(index=0; index<limit; index+=stride)

x[index] = x[index] + 1;

}

* + - 1. Change to directory cm1/ and analyze de code of the program cm1.c. Identify its source code with the program described above.

What are the processor events that will be analyzed during its execution? Explain their meaning.

* + - 1. Compile the program cm1.c using the provided Makefile and execute cm1. Plot the variation of the average number of misses (*Avg Misses*) with the stride size, for each considered dimension of the L1 data cache (8kB, 16kB, 32kB and 64kB).

NOTE: A fast sketch of these plots can be drawn in your computer by running the following commands:

./cm1 > cm1.out

./cm1\_proc.sh

NOTE 2: You can draw these tables and plots on your computer, print, and attach to the report. You do not have to

ﬁll them by hand on the printed report.

NOTE 3: You may need to mark the script as executable before being able to run it.

* + - 1. By analyzing the obtained results:
         * Determine the **size** of the L1 data cache. Justify your answer.
         * Determine the **block size** adopted in this cache. Justify your answer.
         * Characterize the **associativity set size** adopted in this cache. Justify your answer.

#### Modeling the L2 Cache

In this part of the assignment, the goal is to experimentally model the characteristics of the L2 cache of the targeted computer platform. To analyze the computer’s L2 cache, we will use the same methodology that was introduced in the previous section to model the L1 data cache.

* + - 1. Modify the program cm1.c in order to analyze the characteristics of the L2 cache. (Hint: use the event PAPI\_L2\_DCM.) Describe and justify the changes introduced in this program.
      2. Compile the program cm1.c, execute cm1, and plot the variation of the average number of misses (*Avg Misses*) with the stride size, for each considered dimension of the L2 cache.
      3. By analyzing the obtained results:
         * Determine the **size** of the L2 cache. Justify your answer.
         * Determine the **block size** adopted in this cache. Justify your answer.
         * Characterize the **associativity set size** adopted in this cache. Justify your answer.

### Proﬁling and Optimizing Data Cache Accesses

Often, programmers wishing to improve their programs’ performance focus their attention on how the programs affect the computer’s caches. In the following, it will be analyzed how simple code changes can help to improve that performance for a matrix multiplication application.

Consider a simple matrix multiplication application, operating on two square matrices of *N N*

*×*

16-bit integer elements, with *N* = 1024. From a mathematical point of view, given two matrices **A** and

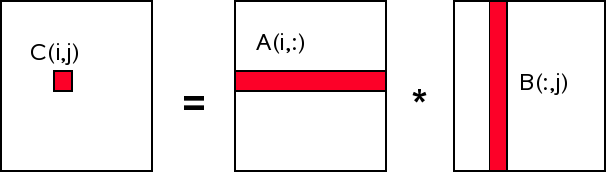
**B**, with elements *aij* and *bij* such that 0 *≤ i, j < N* , the product matrix **C** is deﬁned as:

*N−*1

∑

*cij* = *aikbkj* = *ai*1*b*1*j* + *ai*2*b*2*j* + *. . .* + *ai*(*N−*1)*b*(*N−*1)*j* (1)

*k*=0



**Figure 2:** Straightforward matrix multiplication.

#### Straightforward implementation

A straight-forward C implementation of Eq. 1 can look like this:

for (i = 0; i < N; ++i) { for (j = 0; j < N; ++j) {

for (k = 0; k < N; ++k) {

res[i][j] += mul1[i][k] \* mul2[k][j];

}

}

}

The two input matrices are mul1 and mul2. The result matrix res is assumed to be initialized to all zeroes.

The provided program mm1.c includes this code sequence and all the necessary initialization steps, as well as the set of statements that are required in order to proﬁle its execution using the PAPI toolbox.

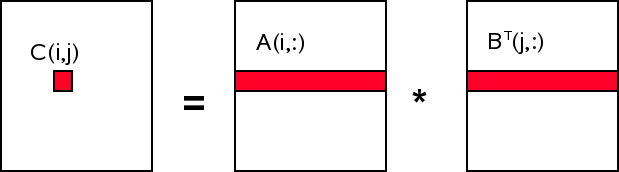
* + - 1. Change to directory mm1/ and analyze de code of the program mm1.c. Identify its source code with the program described above.

What is the total amount of memory that is required to accommodate each of these matrices?

* + - 1. Compile the source ﬁle mm1.c using the provided Makefile and execute it. Fill the table with the obtained data.
      2. Evaluate the resulting L1 data cache *Hit-Rate*.

#### First Optimization: Matrix transpose before multiplication [2]

By analyzing the obtained results, it can be observed that such a straightforward implementation suffers from a severe penalty in what concerns the amount of L2 cache misses resulting from its access pattern. In fact, while mul1 matrix is accessed sequentially, the inner loop advances the row number of mul2 (see Fig. 2), meaning successive accesses to far away memory positions.



**Figure 3:** Transposed matrix multiplication.

One possible remedy to attenuate such problem is based on matrix transposition. In fact, since each matrix element is accessed multiple times, it might be worthwhile to rearrange (“transpose,” in mathematical terms) the second matrix mul2 before using it (see Fig. 3):

*N−*1

*cij* = ∑ *aikbT*

*jk*

*j*1

*j*2

*j*(*N−*1)

= *ai*1*bT*

+ *ai*2*bT*

+ *. . .* + *ai*(*N−*1)*bT*

(2)

*k*=0

After the preliminary transposition step, both matrices may be iterated sequentially. As far as the C code is concerned, it now looks like this:

int16\_t tmp[N][N];

// transposition

for (i = 0; i < N; ++i) { for (j = 0; j < N; ++j) {

tmp[i][j] = mul2[j][i];

}

}

// multiplication

for (i = 0; i < N; ++i) { for (j = 0; j < N; ++j) {

for (k = 0; k < N; ++k) { res[i][j] += mul1[i][k] \* tmp[j][k];

}

}

}

Variable tmp is a temporary array to store the transposed matrix.

One direct consequence of this optimization is that it now requires additional accesses to the data memory. Hopefully, this extra cost can be easily recovered, since the 1024 non-sequential accesses per column are usually much more expensive.

* + - 1. Change to directory mm2/ and analyze the code of the program mm2.c. Identify its source code with the program described above. Compile this program using the provided Makefile and exe- cute it.

Fill the table with the obtained data.

* + - 1. Evaluate the resulting L1 data cache *Hit-Rate*.
      2. Change the code in the program mm2.c in order to include the matrix transposition in the execution time. Compile this program using the provided Makefile and execute it.

Fill the table with the obtained data.

Comment on the obtained results when including the matrix transposition in the execution time.

* + - 1. Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedups.

#### Second Optimization: Blocked (tiled) matrix multiply [2]

Despite the good results that may be obtained with the matrix transposition method, in many applications this approach can not be applied, either because the matrix is too large or the available memory is too small. Hence, other alternatives, which do not require the extra copy procedure, should be studied.

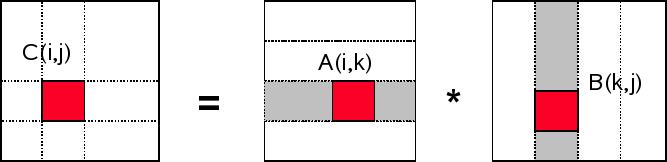
The search for an alternative processing scheme should start with a close examination of the involved math and the operations performed by the original implementation. Trivial math knowledge shows that the order of the several additions to obtain each element of the result matrix is irrelevant, as long as

each addend appears exactly once. This understanding will lead to solutions which reorder the additions performed in the inner loop of the original code.

According to the original algorithm, the adopted order to access the elements of matrix mul2 is: (0,0), (1,0), ... , (N -1,0), (0,1), (1,1), Although the elements (0,0) and (0,1) are in the same cache

line, by the time the inner loop completes one round, this cache line has long been evicted. For this example, each round of the inner loop requires, for each of the three matrices, 1024 cache lines, which is much more than what is available in most processors’ caches.

One possible solution is to simultaneously handle more than one iteration of the middle loop, while executing the inner loop. In this case, several values which are guaranteed to be in cache will be used, thus contributing to a reduction of the L2 cache miss-rate. Hence, to maximize the speedup provided by this technique, it is necessary to adapt the dimension of the sub-matrix under processing to the cache block size, by taking into account the size of each matrix element. As a hypothetical example, considering that a short operand occupies 2-Bytes, this means that a 64-Byte cache block will accommodate 32 matrix elements, thus deﬁning the optimal size for the sub-matrix line to be 32 (see Fig. 4).



**Figure 4:** Blocked matrix multiplication.

As far as the C code is concerned, it now looks like this:

#define SUB\_MATRIX\_SIZE (CACHE\_LINE\_SIZE / sizeof (short))

for (i = 0; i < N; i += SUB\_MATRIX\_SIZE) {

for (j = 0; j < N; j += SUB\_MATRIX\_SIZE) {

for (k = 0; k < N; k += SUB\_MATRIX\_SIZE) {

for (i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k]; i2 < SUB\_MATRIX\_SIZE;

++i2, rres += N, rmul1 += N) {

for (k2 = 0, rmul2 = &mul2[k][j]; k2 < SUB\_MATRIX\_SIZE; ++k2, rmul2 += N) { for (j2 = 0; j2 < SUB\_MATRIX\_SIZE; ++j2) {

rres[j2] += rmul1[k2] \* rmul2[j2];

}

}

}

}

}

}

The most visible change is that the code has six nested loops now. The outer loops iterate with intervals of SUB\_MATRIX\_SIZE (the cache line size CACHE\_LINE\_SIZE divided by sizeof(short)). This divides the matrix multiplication in several smaller problems which can be handled with more cache locality. The inner loops iterate over the missing indexes of the outer loops. There are, once again, three loops. The k2 and j2 loops are in a different order. This is done because, in the actual computation, only one expression depends on k2 but two depend on j2.

* + - 1. Change to directory mm3/ and analyze the code of the program mm3.c. Identify its source code with the program described above.

Change the program source code in order to comply the algorithm parameterization (sub-matrix line size) with the block size (CLS) that was determined in Section 3.1.

How many matrix elements can be accommodated in each cache line?

* + - 1. Compile this program using the provided Makefile and execute it. Fill the table with the obtained data.
      2. Evaluate the resulting L1 data cache *Hit-Rate*.
      3. Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedup.
      4. Compare the obtained results with those that were obtained for the matrix transpose implementa- tion by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedup. If the obtained speedup is positive, but the difference of the resulting hit-rates is negative, how do you explain the performance improvement? (Hint: study the hit-rates of the L2 cache for both implementations; You may use the following PAPI events PAPI\_L2\_DCH (or PAPI\_L2\_DCM) and PAPI\_L2\_DCA. Run papi\_avail to check for available events and understand their meaning.)

## References

1. Performance Application Programming Interface (PAPI). Webpage. "[http://icl.cs.utk.](http://icl.cs.utk/) edu/papi", December 2008.
2. Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.
3. *PAPI User’s Guide*.
4. *PAPI Programmer’s Reference*.

# Second Lab Assignment: System Modeling and Proﬁling

STUDENTS IDENTIFICATION:

|  |  |
| --- | --- |
| **Number:** | **Name:** |
|  |  |
|  |  |
|  |  |

## Exercise

Please justify all your answers with values from the experiments.

* 1. What is the cache capacity of the computer you used (please write the workstation name)?

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Array Size** |  |  |  |  |  |  |
| **t2-t1** |  |  |  |  |  |  |
| **# accesses a[i]** |  |  |  |  |  |  |
| **# mean access time** |  |  |  |  |  |  |

|  |
| --- |
|  |

Consider the data presented in Figure 1. Answer the following questions (2, 3, 4) about the machine used to generate that data.

* 1. What is the cache capacity?

|  |
| --- |
|  |

* 1. What is the size of each cache block?

|  |
| --- |
|  |

* 1. What is the L1 cache miss penalty time?

|  |
| --- |
|  |

## Procedure

#### Modeling the L1 Data Cache

* + - 1. What are the processor events that will be analyzed during its execution? Explain their meaning.

|  |
| --- |
|  |

* + - 1. Plot the variation of the average number of misses (*Avg Misses*) with the stride size, for each considered dimension of the L1 data cache (8kB, 16kB, 32kB and 64kB).

#### Note that, you may ﬁll these tables and graphics (as well as the following ones in this report) on your computer and submit the printed version.

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Stride** | **Avg Misses** | **Avg Cycl Time** |
| 8kBytes | 1 | 0,000104 | 0,002193 |
| 2 | 0,0000765 | 0,00221125 |
| 4 | 0,00002575 | 0,00218575 |
| 8 | 0,0000165 | 0,002136 |
| 16 | 0,000012 | 0,0022025 |
| 32 | 0,00002 | 0,00215025 |
| 64 | 0,00002025 | 0,00211275 |
| 128 | 0,00001025 | 0,002007 |
| 256 | 0,00000525 | 0,00194425 |
| 512 | 0,0000035 | 0,00194225 |
| 1024 | 0,000002 | 0,00191775 |
| 2048 | 0,00000325 | 0,00201525 |
| 4096 | 0,0000035 | 0,0021 |
| 16kBytes | 1 | 0,000079 | 0,00219975 |
| 2 | 0,00019175 | 0,00218875 |
| 4 | 0,00034325 | 0,00219575 |
| 8 | 0,00065325 | 0,00213225 |
| 16 | 0,00126025 | 0,00218075 |
| 32 | 0,00250025 | 0,00217875 |
| 64 | 0,00003675 | 0,00215725 |
| 128 | 0,0000225 | 0,0021015 |
| 256 | 0,00001225 | 0,00199275 |
| 512 | 0,0000045 | 0,001966 |
| 1024 | 0,00000225 | 0,00194025 |
| 2048 | 0,000003 | 0,00206125 |
| 4096 | 0,00000275 | 0,00222175 |
| 8102 | 0,00000225 | 0,00216925 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Stride** | **Avg Misses** | **Avg Cycl Time** |
| 32kBytes | 1 | 0,000965 | 0,00221025 |
| 2 | 0,00154425 | 0,00218425 |
| 4 | 0,00277975 | 0,00216175 |
| 8 | 0,00506875 | 0,0021075 |
| 16 | 0,009466 | 0,002023 |
| 32 | 0,01301825 | 0,00210375 |
| 64 | 0,01395325 | 0,00204875 |
| 128 | 0,04142575 | 0,002097 |
| 256 | 0,08156675 | 0,0021115 |
| 512 | 0,030923 | 0,00202175 |
| 1024 | 0,04595125 | 0,00197575 |
| 2048 | 0,047435 | 0,00202 |
| 4096 | 0,0733255 | 0,00229775 |
| 8102 | 0,000003 | 0,0022 |
| 16384 | 0,0000015 | 0,002691 |
| 64kBytes | 1 | 0,0156485 | 0,00194075 |
| 2 | 0,03129975 | 0,0018575 |
| 4 | 0,06265175 | 0,00210025 |
| 8 | 0,12533875 | 0,00217825 |
| 16 | 0,25059675 | 0,00220275 |
| 32 | 0,501075 | 0,0021085 |
| 64 | 1,000364 | 0,0018075 |
| 128 | 1,0272305 | 0,00183925 |
| 256 | 1,05449325 | 0,00188475 |
| 512 | 1,0469055 | 0,00187775 |
| 1024 | 1,00980275 | 0,001862 |
| 2048 | 1,81308225 | 0,005062 |
| 4096 | 1,203477 | 0,00665525 |
| 8102 | 0,0754395 | 0,002315 |
| 16384 | 0,00000275 | 0,00223 |
| 32768 | 0,000001 | 0,00210975 |

* + - 1. By analyzing the obtained results:
         * Determine the **size** of the L1 data cache. Justify your answer.
         * Determine the **block size** adopted in this cache. Justify your answer.
         * Characterize the **associativity set size** adopted in this cache. Justify your answer.

#### Modeling the L2 Cache

* + - 1. Describe and justify the changes introduced in this program.

|  |
| --- |
|  |

* + - 1. Plot the variation of the average number of misses (*Avg Misses*) with the stride size, for each considered dimension of the L2 cache.

1*.*0

0*.*8

0*.*6

0*.*4

0*.*2

0

20 21 22 23 24 25 26 27 28 29 210 211 212 213 214 215 216 217 218 219 220

* + - 1. By analyzing the obtained results:
         * Determine the **size** of the L2 cache. Justify your answer.
         * Determine the **block size** adopted in this cache. Justify your answer.
         * Characterize the **associativity set size** adopted in this cache. Justify your answer.

### Proﬁling and Optimizing Data Cache Accesses

#### Straightforward implementation

* + - 1. What is the total amount of memory that is required to accommodate each of these matrices?

|  |
| --- |
|  |

* + - 1. Fill the following table with the obtained data.

|  |  |
| --- | --- |
| Total number of L1 data cache misses | *×*106 |
| Total number of load / store instructions completed | *×*106 |
| Total number of clock cycles | *×*106 |
| Elapsed time | seconds |

* + - 1. Evaluate the resulting L1 data cache *Hit-Rate*:

|  |
| --- |
|  |

#### First Optimization: Matrix transpose before multiplication [2]

* + - 1. Fill the following table with the obtained data.

|  |  |
| --- | --- |
| Total number of L1 data cache misses | *×*106 |
| Total number of load / store instructions completed | *×*106 |
| Total number of clock cycles | *×*106 |
| Elapsed time | seconds |

* + - 1. Evaluate the resulting L1 data cache *Hit-Rate*:

|  |
| --- |
|  |

* + - 1. Fill the following table with the obtained data.

|  |  |
| --- | --- |
| Total number of L1 data cache misses | *×*106 |
| Total number of load / store instructions completed | *×*106 |
| Total number of clock cycles | *×*106 |
| Elapsed time | seconds |

Comment on the obtained results when including the matrix transposition in the execution time:

|  |
| --- |
|  |

* + - 1. Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedups.

|  |
| --- |
| ∆HitRate = HitRatemm2 *−* HitRatemm1: |
| Speedup(#Clocks) = #Clocksmm1*/*#Clocksmm2: |
| Speedup(Time) = Timemm1*/*Timemm2: |
| Comment: |

#### Second Optimization: Blocked (tiled) matrix multiply [2]

* + - 1. How many matrix elements can be accommodated in each cache line?

|  |
| --- |
|  |

* + - 1. Fill the following table with the obtained data.

|  |  |
| --- | --- |
| Total number of L1 data cache misses | *×*106 |
| Total number of load / store instructions completed | *×*106 |
| Total number of clock cycles | *×*106 |
| Elapsed time | seconds |

* + - 1. Evaluate the resulting L1 data cache *Hit-Rate*:

|  |
| --- |
|  |

* + - 1. Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedup.

|  |
| --- |
| ∆HitRate = HitRatemm3 *−* HitRatemm1: |
| Speedup(#Clocks) = #Clocksmm1*/*#Clocksmm3: |
| Comment: |

* + - 1. Compare the obtained results with those that were obtained for the matrix transpose implementa- tion by calculating the difference of the resulting hit-rates (∆HitRate) and the obtained speedup. If the obtained speedup is positive, but the difference of the resulting hit-rates is negative, how do you explain the performance improvement? (Hint: study the hit-rates of the L2 cache for both implementations;)

|  |
| --- |
| ∆HitRate = HitRatemm3 *−* HitRatemm2: |
| Speedup(#Clocks) = #Clocksmm2*/*#Clocksmm3: |
| Comment: |

#### 3.2.3 Comparing results against the CPU speciﬁcations

Now that you have characterized the cache on your lab computer, you are going to compare it against the manufacturer’s speciﬁcation. For this you can check the device’s datasheet, or make use of the command lscpu. Comment the results.

|  |
| --- |
|  |

## A PAPI - Performance Application Programming Interface

The PAPI project [1] speciﬁes a standard Application Programming Interface (API) for accessing hard- ware performance counters available in most modern microprocessors. These counters exist as a small set of registers that count *Events*, deﬁned as occurrences of speciﬁc signals related to the processor’s function (such as cache misses and ﬂoating point operations), while the program executes on the pro- cessor. Monitoring these events may have a variety of uses in the performance analysis and tuning of an application, since it facilitates the correlation between the source/object code structure and the efﬁ- ciency of the actual mapping of such code to the underlying architecture. Besides performance analysis, and hand tuning, this information may also be used in compiler optimization, debugging, benchmarking, monitoring and performance modeling.

PAPI has been implemented on a number of different platforms, including: Alpha; MIPS R10K and R12K; AMD Athlon and Opteron; Intel Pentium II, Pentium III, Pentium M, Pentium IV, Itanium 1 and Itanium 2; IBM Power 3, 4 and 5; Cell; Sun UltraSparc I, II and II, etc.

Although each processor has a number of events that are native to that speciﬁc architecture, PAPI provides a software abstraction of these architecture-dependent *Native Events* into a collection of *Preset Events*, also known as *predeﬁned events*, that deﬁne a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide per- formance counters. They give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Hence, preset events may be regarded as mappings from symbolic names (PAPI preset name) to machine speciﬁc deﬁnitions (native countable events) for a particular hardware resource. For example, Total Cycles (in user mode) is mapped into PAPI\_TOT\_CYC. Some presets are derived from the underlying hardware metrics. For example, Total L1 Cache Misses (PAPI\_L1\_TCM) is the sum of L1 Data Misses and L1 Instruction Misses on a given platform. The list of preset and native events that are available on a speciﬁc platform can be obtained by running the commands papi\_avail and papi\_native\_avail, both provided by the papi source distribution.

Besides the standard set of events for application performance tuning, the PAPI speciﬁcation also includes both a high-level and a low-level sets of routines for accessing the counters. The high level interface consists of eight functions that make it easy to get started with PAPI, by simply providing the ability to start, stop, and read sets of events. This interface is intended for the acquisition of simple but accurate measurement by application engineers [3, 4]:

PAPI\_num\_counters – get the number of hardware counters available on the system; PAPI\_flops – simpliﬁed call to get Mﬂops/s (ﬂoating point operation rate), real and processor time;

*•*

*•*

* PAPI\_ipc – gets instructions per cycle, real and processor time;
* PAPI\_accum\_counters – add current counts to array and reset counters;
* PAPI\_read\_counters – copy current counts to array and reset counters;
* PAPI\_start\_counters – start counting hardware events;
* PAPI\_stop\_counters – stop counters and return current counts.

The following is a simple code example of using the high-level API [3, 4]:

#include <papi.h>

#define NUM\_FLOPS 10000

#define NUM\_EVENTS 1

int main(){

int Events[NUM\_EVENTS] = {PAPI\_TOT\_INS};

long\_long values[NUM\_EVENTS];

/\* Start counting events \*/

if (PAPI\_start\_counters(Events, NUM\_EVENTS) != PAPI\_OK)

handle\_error(1); do\_some\_work();

/\* Read the counters \*/

if (PAPI\_read\_counters(values, NUM\_EVENTS) != PAPI\_OK)

handle\_error(1);

printf("After reading the counters: %lld\n",values[0]); do\_some\_work();

/\* Add the counters \*/

if (PAPI\_accum\_counters(values, NUM\_EVENTS) != PAPI\_OK)

handle\_error(1);

printf("After adding the counters: %lld\n", values[0]); do\_some\_work();

/\* Stop counting events \*/

if (PAPI\_stop\_counters(values, NUM\_EVENTS) != PAPI\_OK)

handle\_error(1);

printf("After stopping the counters: %lld\n", values[0]);

}

Possible output:

After reading the counters: 441027 After adding the counters: 891959 After stopping the counters: 443994

The fully programmable low-level interface provides more sophisticated options for controlling the counters, such as setting thresholds for interrupt on overﬂow, as well as access to all native counting modes and events. Such interface is intended for third-party tool writers or users with more sophisticated needs.

The PAPI speciﬁcation also provides access to the most accurate timers available on the platform in use. These timers can be used to obtain both real and virtual time on each supported platform: the real time clock runs all the time (e.g., a wall clock), while the virtual time clock runs only when the processor is running in user mode.

In the following code example, PAPI\_get\_real\_cyc() and PAPI\_get\_real\_usec() are used to obtain the real time it takes to create an event set in clock cycles and in microseconds, respectively [3, 4]:

#include <papi.h>

int main(){

long long start\_cycles, end\_cycles, start\_usec, end\_usec; int EventSet = PAPI\_NULL;

if (PAPI\_library\_init(PAPI\_VER\_CURRENT) != PAPI\_VER\_CURRENT) exit(1);

/\*Create an EventSet \*/

if (PAPI\_create\_eventset(&EventSet) != PAPI\_OK)

exit(1);

/\* Gets the starting time in clock cycles \*/ start\_cycles = PAPI\_get\_real\_cyc();

/\* Gets the starting time in microseconds \*/ start\_usec = PAPI\_get\_real\_usec();

do\_some\_work();

/\* Gets the ending time in clock cycles \*/ end\_cycles = PAPI\_get\_real\_cyc();

/\* Gets the ending time in microseconds \*/ end\_usec = PAPI\_get\_real\_usec();

printf("Wall clock cycles: %lld\n", end\_cycles - start\_cycles); prinf(“Wall clock time in microseconds: %lld\n”, end\_usec - start\_usec);

}

Possible output:

Wall clock cycles: 100173

Wall clock time in microseconds: 136