Second Lab Assignment: System Modeling and Profiling

STUDENTS IDENTIFICATION:

Number:	Name:
105875	Maria Ramos
106336	Enzo Nunes
106909	Guilherme Campos

2 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	8kB	16kB	32kB	64kB	128kB	256kB
t2-t1	0,002204s	0,004010s	0,008247s	0,019693s	0,042943s	0,092911s
# accesses a[i]	819200	1638400	3276800	6553600	13107200	409600
# mean access time	2,69043ns	2,44751ns	2,51678ns	3,00491ns	3,27629ns	3,31055ns

The cache capacity is 32kB. For this exercise, we used a computer from classroom 0-14. Looking at the values, we observe that up until 32kB, the mean access time stays relatively low, suggesting that the data fits well within the cache. However, starting at 64kB, there is a noticeable increase in mean access time to 3.00 ns for 64k, 3.27 ns for 128k and 3,31 for 256k. This indicates that cache misses are becoming more frequent as the array size is bigger than the cache capacity.

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

2. What is the cache capacity?

The cache capacity is 64k $\times$ 1B = 64kB. Looking at the graph, the performance for small arrays remains relatively flat across various strides, indicating that the array fits into the cache. However, a significant performance degradation occurs from size 64k to 128k. This means that the amount of data being accessed no longer fits within the cache, causing more cache lines to be replaced, resulting in cache misses and expensive memory accesses. We multiply 64k by 1B since we are working with 64k integers, and each occupying 1 byte of memory (uint8_t).

3. What is the size of each cache block?

The size of each cache block is 16B. When the stride size is small, the program accesses memory in a sequential pattern, so the entire block of data is utilised efficiently, resulting in fewer cache misses. As the stride value increases beyond 16B, which corresponds to larger jumps in memory access, you can see an increase in performance degradation. This happens because the stride becomes larger than the cache block size, resulting in more frequent cache misses.

4. What is the L1 cache miss penalty time?

The L1 cache miss penalty time is approximately 600ns. Looking at the graph, the access time for a 64k array or smaller is around 400ns, while for a 128k array, it increases to about 1000ns. The difference between these times reflects the miss penalty: 1000 – 400 = 600ns, representing the additional time required to access data once the array size exceeds the cache capacity.

3 Procedure

3.1.1 Modeling the L1 Data Cache

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

The code in cm1.c uses the PAPI_L1_DCM event from the PAPI library to track L1 data cache misses. This event measures the number of times data is accessed but not found in the cache. The code loops through different array sizes and strides to evaluate how changing memory access patterns impacts cache performance. It also analyzes the number of clock cycles and execution time, by capturing the starting time in clock cycles using PAPI_get_real_cyc() and the starting time in microseconds using PAPI_get_real_usec(). This allows for a more comprehensive analysis of both cache efficiency and timing.

b) 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	0.000239	0.003813
	2	0.000826	0.003761
	4	0.000274	0.003350
	8	0.000076	0.003122
	16	0.000528	0.003068
	32	0.000225	0.002830
	64	0.000053	0.002854
	128	0.000131	0.002801
	256	0.000220	0.002732
	512	0.000103	0.002447
	1024	0.000016	0.002342
	2048	0.000041	0.002614
	4096	0.000070	0.002903
16kBytes	1	0.000659	0.002677
	2	0.000744	0.002845
	4	0.000465	0.002658
	8	0.000670	0.002941
	16	0.000334	0.002733
	32	0.001099	0.002563
	64	0.000207	0.002586
	128	0.000119	0.002532
	256	0.000030	0.002615
	512	0.000019	0.002373
	1024	0.000018	0.002298
	2048	0.000003	0.002347
	4096	0.000003	0.002453
	8192	0.000004	0.002363

Array Size	Stride	Avg Misses	Avg Cycl Time
32kBytes	1	0.001993	0.002478
	2	0.002802	0.002485
	4	0.004169	0.002526
	8	0.007133	0.002552
	16	0.013420	0.002527
	32	0.023663	0.002634
	64	0.027893	0.002683
	128	0.049160	0.002692
	256	0.000485	0.002739
	512	0.000481	0.002906
	1024	0.000078	0.002428
	2048	0.000037	0.002457
	4096	0.000014	0.002671
	8192	0.000004	0.002453
	16384	0.000002	0.002375
64kBytes	1	0.015641	0.002476
	2	0.031260	0.002483
	4	0.062523	0.002488
	8	0.125045	0.002501
	16	0.250077	0.002535
	32	0.500087	0.002494
	64	0.999098	0.002772
	128	0.998949	0.002786
	256	1.001812	0.002770
	512	1.003597	0.002776
	1024	1.000002	0.002940
	2048	1.506382	0.004458
	4096	1.222427	0.008196
	8192	0.000020	0.002739
	16384	0.000002	0.002459
	32768	0.000014	0.002495

c) By analyzing the obtained results:

- Determine the **size** of the L1 data cache. Justify your answer.

The size of the L1 data cache is 32kB. Looking at the plot, we see a sharp increase in cache misses starting in the 64k array size as the access time raises in the graph, which suggests that the array size exceeds the capacity of the L1 cache at that point. Before this, the cache efficiently handles the accesses, implying that it can store enough data to avoid significant cache misses.

- Determine the **block size** adopted in this cache. Justify your answer.

When using an array of 64kB, which is the double of the cache size, the graph shows a noticeable shift in behaviour at the stride size of 64 bytes. In this case, the cache consistently loads new blocks from the L2 cache or main memory, which indicates the block size is 64 bytes. This is because, at a 64 byte stride, each memory access corresponds to the start of a new cache block, causing a consistent cache miss for each access and leading to a stable access times.
At a 64 byte stride, we can notice that the plot flattens due to the fact that we have broken out of the bounds of the block, that is, sequential accesses are to different blocks.

- Characterize the **associativity set size** adopted in this cache. Justify your answer.

When analysing the miss rate for a 64kB array, we can observe that for a stride size of 32kB (half the array size), the miss rate is nearly zero. The stride size determines how many distinct blocks of array data we access during each iteration. With a 32kB stride, we access only 2 different blocks, which the cache can hold easily. If the cache is at least 2-way set associative, it can map both blocks without conflict, resulting in almost zero misses, which is consistent with the observed data.

This pattern holds for smaller strides as well:
- With a 16kB stride (a quarter of the array size), we access 4 blocks, which still fit within the cache's associativity.
- Similarly, with an 8kB stride (one-eighth of the array size), we access 8 blocks. Given that the miss rate remains low, this suggests the cache is 8-way set associative, as it can hold all 8 blocks without conflicts.

However, with a 4kB stride, the miss rate increases significantly. This is because the stride results in 16 blocks being accessed, which likely exceed the cache's number of ways (or sets). Since multiple blocks map to the same cache location, they start to evict each other, resulting in higher cache misses. This increase in misses suggests that the cache's associativity (number of ways) is no longer sufficient to handle the increased block accesses.

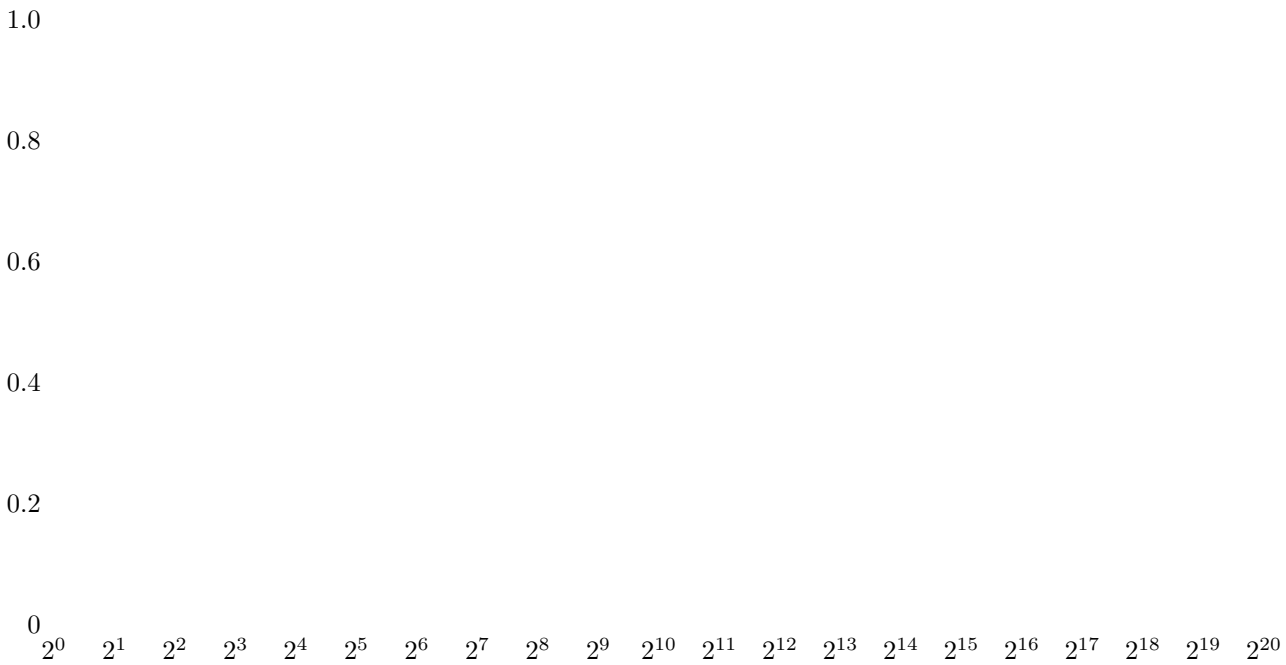
For a 64kB array divided by a 8kB stride, the total number of sets accessed can be calculated as: 64kB / 8kB = 2^6 / 2^3 = 2^3 = 8 sets.

3.1.2 Modeling the L2 Cache

a) Describe and justify the changes introduced in this program.

First, we modified the event being monitored from PAPI_L1_DCM to PAPI_L2_DCM, allowing us to measure L2 data cache misses instead of L1. Consequently, to accommodate the larger size of the L2 cache, we adjusted the minimum (CACHE_MIN) and maximum (CACHE_MAX) array sizes to 128 kB and 1 MiB, respectively. These larger array sizes are necessary to exceed the L2 cache size, enabling us to analyse its performance characteristics effectively.

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



c) By analyzing the obtained results:

- Determine the **size** of the L2 cache. Justify your answer.

The size of the L2 cache is 256kB. Looking at the plot, we can observe a noticeable increase in the miss rate as the array size jumps from 128 kB to 256 kB, followed by a more significant jump from 256 kB to 512 kB. The larger increase in miss rate at the second jump suggests that the L2 cache size has been exceeded. This is likely because, at 512 kB, the array can no longer fully fit within the 256 kB L2 cache, resulting in more frequent cache misses.

- Determine the **block size** adopted in this cache. Justify your answer.

The size of each cache block is 64B. Just like in the L1 cache, we noticed that when the stride is less than 64 bytes, the miss rate keeps increasing, meaning some sequential accesses are still within the same block. However, the graph shows a noticeable shift in behaviour at a stride size of 64 bytes. In this case, the cache consistently loads new blocks from the main memory, which indicates the block size is 64 bytes. This is because, at a 64 byte stride, each memory access corresponds to the start of a new cache block, causing a consistent cache miss for each access and leading the graph to stabilise.
At a 64 byte stride, we can notice that the plot flattens due to the fact that we have broken out of the bounds of the block, that is, sequential accesses are to different blocks.

- Characterize the **associativity set size** adopted in this cache. Justify your answer.

When analysing the miss rate for the largest array size, 1MiB, we observe that with a stride size of 512kB (half the array size), the miss rate is nearly zero. This is because the stride size influences the number of distinct blocks we access. With a 512kB stride, we only access 2 distinct blocks in total. If the cache is at least 2-way set associative, it can accommodate both blocks without conflicts, resulting in a near-zero miss rate. This outcome aligns with the observed data.

This low miss rate pattern extends to smaller strides:

With strides ranging from 32kB to 256kB, the miss rate remains near zero, suggesting that the cache's associativity can handle all accessed blocks without excessive evictions.
Given the low miss rate across these strides, we can infer that the cache is 32-way set associative. This is because the cache can hold 32 distinct blocks at these stride sizes.
When the stride decreases to 16kB, the miss rate increases noticeably. This happens because the stride size now results in more than 32 blocks being accessed, exceeding the cache's associativity level. In this scenario, blocks start to evict each other within the same cache set, leading to a higher miss rate. The increase in misses indicates that the cache's 32-way associativity is insufficient to handle the larger number of accessed blocks, which map to the same sets repeatedly.

For a 1MiB array divided by a 32kB stride, the total number of sets accessed can be calculated as: 1MiB / 32kB = 2^20 / 2^15 = 2^5 = 32 sets.

3.2 Profiling and Optimizing Data Cache Accesses

3.2.1 Straightforward implementation

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

Each matrix is a N x N square matrix with N = 512. Each element in the matrix is a int16_t, meaning each element is represented by 16 bits, 2 bytes. The size of each matrix is N^2 elements times 2 bytes per element. 512^2 * 2 B = 262144 * 2 B = 512 kB.

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	134.264337	$\times 10^6$
Total number of load / store instructions completed	2551.470328	$\times 10^6$
Total number of clock cycles	1918.169135	$\times 10^6$
Elapsed time	0.582607	seconds

c) Evaluate the resulting L1 data cache *Hit-Rate*:

To calculate the hit rate we can compute 1 - (L1_miss_count / load_and_store_instruction_count).
1 - (134.264337 / 2551.470328) = 94.7%

3.2.2 First Optimization: Matrix transpose before multiplication [2]

a) Fill the following table with the obtained data.

Total number of L1 data cache misses	4.217411	$\times 10^6$
Total number of load / store instructions completed	2551.456832	$\times 10^6$
Total number of clock cycles	1249.987679	$\times 10^6$
Elapsed time	0.379660	seconds

b) Evaluate the resulting L1 data cache *Hit-Rate*:

Applying the same logic, to calculate the hit rate we can compute 1 - (L1_miss_count / load_and_store_instruction_count).
1 - (4.217411 / 2551.456832) = 99.83%

c) Fill the following table with the obtained data.

Total number of L1 data cache misses	4.750208	$\times 10^6$
Total number of load / store instructions completed	2557.227127	$\times 10^6$
Total number of clock cycles	1247.627334	$\times 10^6$
Elapsed time	0.378942	seconds

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

The new hit rate with the transpose, applying the same logic, is :
1 - (4.750208 / 2557.227127) = 99.81%.
The values did not change significantly. This is because the operation of matrix multiplication is more computationally expensive ($O(N^3)$) than doing a matrix transpose ($O(N^2)$). As a result, the number of clock cycles and elapsed time remains nearly the same with or without the transpose, since the computation dominates the performance rather than memory access efficiency.

d) Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates ($\Delta\mathrm{HitRate}$) and the obtained speedups.

$\Delta\mathrm{HitRate} = \mathrm{HitRate}_{\mathrm{mm2}} - \mathrm{HitRate}_{\mathrm{mm1}}$:	99.81% - 94.7% = 5.11%
$\mathrm{Speedup}(\#\mathrm{Clocks}) = \#\mathrm{Clocks}_{\mathrm{mm1}}/\#\mathrm{Clocks}_{\mathrm{mm2}}$:	1918.169135 / 1247.627334 = 1.537
$\mathrm{Speedup}(\mathrm{Time}) = \mathrm{Time}_{\mathrm{mm1}}/\mathrm{Time}_{\mathrm{mm2}}$:	0.582607 / 0.378942 = 1.537

Comment:

Even though the straightforward implementation yielded a very high hit rate, we still can see a significant speedup with the matrix transposition method. This speedup can be attributed to the improved spatial locality and better cache utilisation. Although matrix transposition incurs some overhead, the reduction in cache misses during multiplication more than compensates for this, leading to the observed speedup.

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

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

Given that each block has size 64 B, and the cache is 8-way associative, we can conclude that each cache line contains 8 * 64 B = 512 B. Each element in the matrix is represented by a int16_t, which takes up 2 B each. 512 B / 2 B = 256 elements.

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	4.790098	$\times 10^6$
Total number of load / store instructions completed	3645.53453	$\times 10^6$
Total number of clock cycles	1866.926729	$\times 10^6$
Elapsed time	0.567042	seconds

c) Evaluate the resulting L1 data cache *Hit-Rate*:

Applying the same logic, to calculate the hit rate we can compute 1 - (L1_miss_count / load_and_store_instruction_count).
1 - (4.790098 / 3645.53453) = 99.87%

d) Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates ($\Delta\mathrm{HitRate}$) and the obtained speedup.

$\Delta\mathrm{HitRate} = \mathrm{HitRate}_{\mathrm{mm3}} - \mathrm{HitRate}_{\mathrm{mm1}}$:	99.87% - 94.7% = 5.17%
$\mathrm{Speedup}(\#\mathrm{Clocks}) = \#\mathrm{Clocks}_{\mathrm{mm1}}/\#\mathrm{Clocks}_{\mathrm{mm3}}$:	1918.169135 / 1866.926729 = 1.027%

Comment:

The blocked matrix multiplication approach improves the straightforward method by ensuring that more data remains in the cache during computation. This leads to reduced cache misses, higher hit rates, and improved performance. This approach is particularly effective when the matrix size exceeds the cache size, making it a highly scalable and efficient optimization for matrix operations.

e) Compare the obtained results with those that were obtained for the matrix transpose implementation by calculating the difference of the resulting hit-rates ($\Delta\mathrm{HitRate}$) 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;)

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm3}} - \text{HitRate}_{\text{mm2}}$:	99.87% - 99.81% = 0.06%
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm2}}/\#\text{Clocks}_{\text{mm3}}$:	1247.627334 / 1866.926729 = 0,66827
Comment: Both matrix transposition and blocked matrix multiplication yielded similar results because they enhance cache efficiency by optimising data locality, though in different ways. The blocked matrix multiplication method reduces cache misses by improving spatial and temporal locality. However, this approach incurs more clock cycles due to added loop complexity and sub-matrix management overhead. If the obtained speedup had been positive while the hit-rate difference was negative, it would indicate more efficient usage of cache that could compensate for a higher miss rate in L1. This didn't happen likely because both methods result in similar memory access patterns, making them highly efficient in cache utilisation. Although blocking is designed to improve cache usage, the transposition might have optimised memory accesses sufficiently, leading to only slight gains from blocking. *	

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.

We ran the commands "lscpu -C" and "getconf -a | grep CACHE" to obtain the computer's specifications and compare them to our results. We confirmed that our findings for the cache size, block size, and associativity of L1 were accurate. However, for L2, althought the data cache size and the block size we calculated are the same as the real ones, we may have calculated the wrong number of sets, 8 instead of the 32 we expected. This discrepancy could be due to incorrect data, misinterpretation of the graphs, or the presence of CPU optimizations, such as prefetching, that we're not accounting for. These optimizations could artificially reduce the miss rate by loading data into cache ahead of time.

*The speedup being below 1 suggests that the matrix multiplication with blocking took longer than the transposed version. This can occur if the overhead of managing blocks—such as additional loops and boundary conditions—outweighed the cache benefits. Essentially, the computational cost from block management may have surpassed the performance gains from improved cache efficiency. In a typical scenario, blocking should lead to fewer clock cycles than matrix transposition, as it facilitates better data reuse.