INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

ORGANIZAÇÃO DE COMPUTADORES

LEIC

First Lab Assignment: System Modeling and Profiling

Version 1.1.0

2023/2024

1 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 performance of a matrix-multiply code segment and optimizing it based on the characteristics of the L2 cache of the target architecture (Subsection 3.2).

1.1 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 `lab1_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.

2 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 lab kit (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.

2. What is the cache capacity of the computer?

3. What is the size of each cache block?

4. What is the L1 cache miss penalty time?

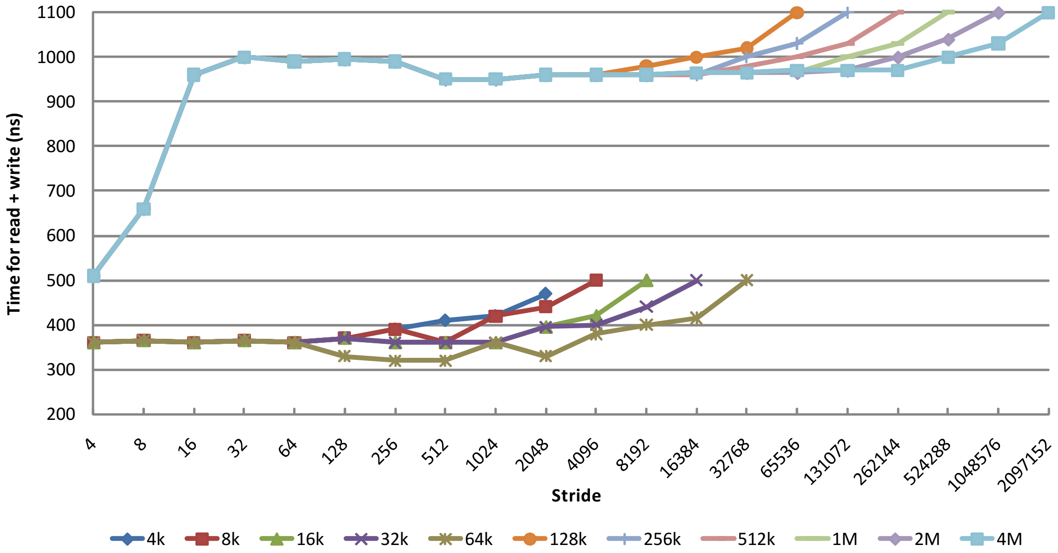

Figure 1: Variation of the cache access time with the adopted *stride* value for different array sizes.

3 Procedure

3.1 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.

3.1.1 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 `lab1_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;
  }
```

a) Change to directory `cm1/`, in the package `lab1_kit.zip`, 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.

b) 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.

c) 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.

3.1.2 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.

a) 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.

b) 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.

c) 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.

3.2 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 $N \times N$ 16-bit integer elements, with $N = 1024$. From a mathematical point of view, given two matrices $\mathbf{A}$ and $\mathbf{B}$, with elements a_{ij} and b_{ij} such that $0 \leq i, j < N$, the product matrix $\mathbf{C}$ is defined as:

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{i(N-1)} b_{(N-1)j} \tag{1}$$

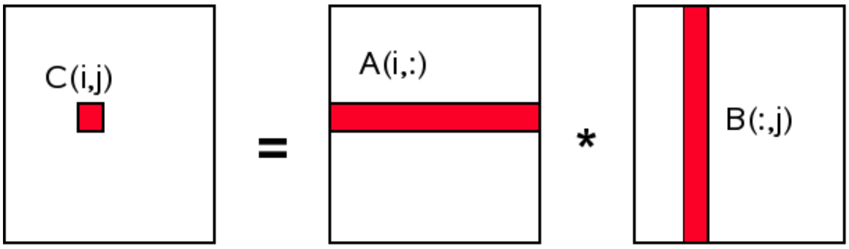

Figure 2: Straightforward matrix multiplication.

3.2.1 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.

a) 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?

b) Compile the source file `mm1.c` using the provided `Makefile` and execute it. Fill the table with the obtained data.

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

3.2.2 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.

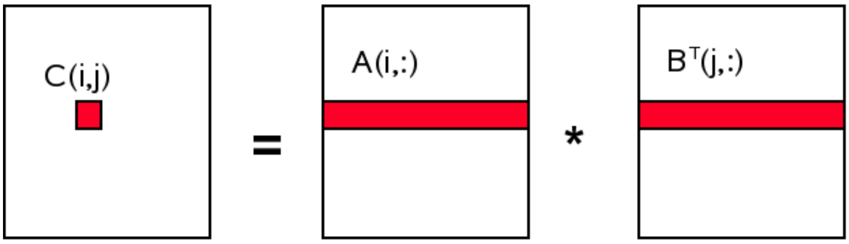

Figure 3: Transposed matrix multiplication.

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):

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b^T_{jk} = a_{i1} b^T_{j1} + a_{i2} b^T_{j2} + \ldots + a_{i(N-1)} b^T_{j(N-1)} \tag{2}$$

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.

a) 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 execute it.

Fill the table with the obtained data.

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

c) 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.

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.

3.2.3 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).

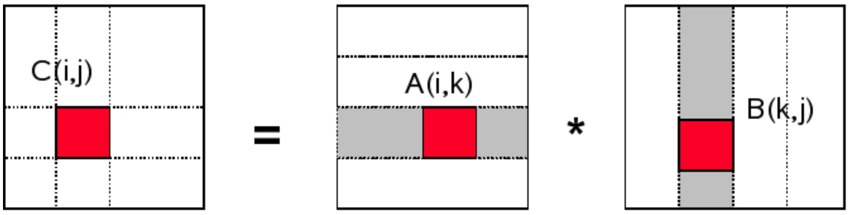

Figure 4: Blocked matrix multiplication.

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`.

a) 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?

b) Compile this program using the provided `Makefile` and execute it. Fill the table with the obtained data.

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

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.

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; 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.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*.

A PAPI - Performance Application Programming Interface

The PAPI project [1] specifies a standard Application Programming Interface (API) for accessing hardware 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 processor. 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 efficiency 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 performance 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
```