Computer Structure

Computadors – Grau en Ciència i Enginyeria de Dades – 2021-2022 Q2

Facultat d'Informàtica de Barcelona

The Computer Structure topic of this course introduces multiple hardware related concepts. Even though GCED is not hardware oriented, it is important you get experience on collecting information from computer devices, as well as understand the impact of them on the performance of apps.

During this Lab Session you will analyse some components of your computer. Afterwards, we present a collection of codes that you have to compile and run to analyse the impact of different execution settings on the system performance. Finally, there is another code that will let you better understand the performance of the memory hierarchy.

In this Lab Session you only have to edit a single file (**"answers.txt"**) to write your findings.

The Processor and Components Layout

Depending on the Operating System (OS) you are using, the information about the processor can be collected in different ways.

Exercise 1

Edit an ***"answers.txt"*** *file and indicate the data you can collect about the processor of your computer. In particular, the model of the processor, number of cores, number of hardware threads, and cache hierarchy. Nevertheless, we encourage you to identify other details we have introduced in Theory lectures. If you are using a virtual machine (VM), show either the information of the VM and the physical computer. We strongly suggest you to follow the indications to find such data as well as to go to Internet and find the number of cores and threads of the CPU model. For example introduce the name of the CPU model, and the keywords "cores" and "threads" in your search.*

Linux-like Operating Systems comprise a special folder, "/proc" (process information pseudo-filesystem), that contains many data about environment characteristics as well as the current status of what is going on in the system. One of them collects information about the processor, which can be found in "/proc/cpuinfo". Among the specs, you will see "siblings" that refers to the total number of hardware threads (also known as processing units in Linux) and "cores" of the processor. Sometimes, siblings is twice the number of cpu cores, meaning that the architecture has two hardware threads per CPU. You can also collect the information using the "lscpu" command. It is very likely you first have to install a given package. Please execute the following two command lines (introduce the same password that you use to access to your accoung when it is asked):

```
#> sudo apt-get update
#> sudo apt-get install util-linux
```

The "sudo" command is used to temporally elevate your permissions as "root" user, and the "apt-get" command is used to install a given required software. Afterwards, execute the command line "#>lscpu" to collect generic information about the processor and "#> lscpu -C" to only collect information about the cache hierarchy (only check the level, type, and size). If you are using Linux in a VM, some data depends on the configuration of the VM itself. If you are using a Mac M1 computer, there are some incompatibilities that prevent to properly collect the data.

If you are using a Mac (either you are using Mac M1 or any other Mac), you can find information in **MacOS** going to the "Apple" symbol (top-left corner of the screen)→About This Mac→System Report...

In **Windows** you can collect data in different ways. One of the easiest ways to collect this information is: open the file explorer, press the right-click with the mouse in the "My PC" icon. Then select "properties" and you will see the model of the CPU. Another approach that presents more detail is doing to the search slot of Windows OS to open an app and write "dispositivos" to open the "Device Manager" (a.k.a. "administrador de dispositivos"). You will find, among others, the processor device. If you click on it, you will see the model and as many entries as hardware threads are in the system. Finally, another approach is to open the "Power Shell" of Windows (in the search slot of the OS to open apps). It is an advanced terminal, in which we suggest you to write the following command lines (take care with capital characters, since it is case-sensitive):

```
#> Get-CimInstance  -class  CIM_Processor  |  Select-Object  *  |  more
#> Get-CimInstance  -class  CIM_CacheMemory  |  Select-Object  *  |  more
#> Get-CimInstance  -class  CIM_Chip  |  Select-Object  *  |  more
```

These command lines present details about the processor, cache hierarchy, and the memory. We suggest you to check the size of the cache hierarchy and compare it with the one shown in the VM.

What if...

In this section we will play with different environment settings. To do this, you have to download a file attached to the following link:
https://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/S5/FilesS5.tar.gz
Once you uncomprise the file, as indicated in the slot, you will see three C codes, the Makefile (remember to compile C files before doing any experiment), and another file, named "launch.sh" (there is a "matrix" subfolder that will be used in the second part of the lab). In the following exercises, if you are using a VM, try different Hw configurations (e.g. changing the number of cores or hardware threads).

The "launch.sh" file is a special group of command lines (also known as shell script) that can be interpreted by the Bash Shell in your terminal. This file needs pairs of input parameters. The first parameter in each pair indicates the n-th processing unit that will run the program indicated by the second parameter in the pair. For example, if you type:

```
#> ./launch.sh  0  ./integers  1  ./floats  1  ./mems
```

it will execute the program "./integers" in the processing unit (this is the name that Linux-like OSes use to refer to hardware thread) "0", whilst "./floats" and "./mems" will run in the processing unit "1".

We strongly suggest you open another terminal and run a command called "top". This command shows the current status of the system and all processes running on it. The output is automatically updated every 2 seconds, by default (although it can be configured to other frequencies). To quit from it, press the "q" key. You should focus on the column "%CPU" that means percentage of the CPU that is consuming every process. Once you write the command line to call "launch.sh" you should automatically see in the output of "top" the name of the programs you indicated.

Finally, after the end of every program execution, there is another output similar to:

```
3.20user  0.00system  0:04.12elapsed  99%CPU ...
```

You ONLY have to take note of the "elapsed" time (that is, "4,12" in this example) and "%CPU". The former value indicates how much time (in minutes, seconds and decimals) the program took to finish the execution (both fields “user” and “system” will be explained in a future session). The latter value is the average consumption of the CPU by this particular program execution. PLEASE, do not execute other heavy programs while you are waiting for the processes to finish. At the end you will get one measurement output per program. If you load the computer with other heavy processes, the measurements may be less accurate.

Exercise 2

*Edit the **“answers.txt”** file and write down the following measurements about the execution of every single app alone:*

a) The command line to run a SINGLE instance of “./integers” program in a processing unit
b) The elapsed and %CPU will be the baseline performance of this program
c) The command line to run a SINGLE instance of “./floats” program in a processing unit
d) The elapsed and %CPU will be the baseline performance of this program
e) The command line to run a SINGLE instance of “./mems” program in a processing unit
f) The elapsed and %CPU will be the baseline performance of this program

If you are using a high-performance computer and you see the elapsed time is too short (e.g. less than 10 seconds), modify the corresponding C code to increase the number of iterations of the main loop (j loop) of that code in order to increase the execution time.

Contention in hardware resources

Let’s analyse what happens when you run multiple independent instances of the same program in the same processing unit. That is, we will bind instances of the same program in the same hardware thread.

Exercise 3

*Edit the **“answers.txt”** file and write down the following pair of points per every experiment:*

a) The command line to run the experiment for a given program type
b) Average elapsed and %CPU

Finally, analyze and briefly explain whether there is any impact on average time or average %CPU compared to baseline measurements. Also indicate if you find any behavioral pattern in the “%CPU” value provided by the “top” command.

Finally, there is a final VOLUNTARY step. Execute experiments binding same types of programs to different processing units. In case your Virtual Machine has a single CPU (go to properties→system→processor), you can reconfigure the system to increase the number of hardware threads. However, you previously have to definitely shutdown the VM without saving the state, reconfigure the setting, and boot again the VM. If you are using Mac M1 with UTM, it is better not to modify the settings of the VM. In this you should measure a performance in the range of the ones obtained in Exercise 2 and 3.

The Memory Hierarchy

There is a subfolder called “matrix” where you can find a sourcode called “matinit.c” that simply initializes three matrixes multiple times and present the time taken by every initialization. Briefly analyse the code and the Makefile. The compilation command lines include a define (“-DROW” and “-DCOLUMN”) to select row-wise versus column-wise access. You can check the different access in the sourcecode and compare it with the slides that introduce row-wise or colum-wise accesses in theory slides of “Data Representation” and “Computer Structure”. In this section we will analyse the impact of using one or the other approach.

To do so, you have to install a suite of tools to profile and debug programs, called “Valgrind” (more information can be found in https://wiki.ubuntu.com/Valgrind), executing the following command line:

```
#> sudo apt-get install valgrind
```

You can modify the matrix size by introducing an input parameter in the command line when you launch the binary, or there is a default size in case you introduce no size. In the line 14th there is a “define” that let’s you modify the data type of the matrixes (by default, it is a “double”).

Take into account this matrix application just initializes matrixes. Thus, the impact on performance time is only in assigning a value during the initialization of the matrixes, but not doing further computations among matrixes.

Exercise 4

*Edit the **“answers.txt”** file and write down your findings when you compare:*

a) the average execution time of both access approaches.

b) the execution using “valgrind”. To use it, you have to execute the command as follows: “#>valgrind --tool=cachegrind PROGRAM” where “PROGRAM” is the name of the binary. Pay special attention to L1 refs, L1 misses, LL refs, LLd misses (data-L1 and L2 caches, respectively).

c) according to the three different types of cache misses (cold, conflict, capacity) introduced in theory lectures, what is the majority type of misses you think that occurs during the different initializations?

Exercise 5

*Edit the **“answers.txt”** file and write down your findings when you modify the type of data (“float” instead of “double”) and compare:*

a) the average execution time of both access approaches.

b) the execution using “valgrind”.

c) according to the three different types of cache misses (cold, conflict, capacity) introduced in theory lectures, what is the reason there is a difference of number of misses compared to the ones obtained in the previous exercise?

Upload the Deliverable

To save the changes you can use the tar command as follows:

```
#tar czvf session5.tar.gz answers.txt
```

Now go to RACO and upload this recently created file to the corresponding session slot.