**LAB 1. Understanding the Basics on   
Cache Hierarchy Performance and System Performance**

**LEARNING GOALS**

* Know multiple events that can be monitored at runtime with the hardware performance counters of current processors, and identify the performance metrics that can be calculated with them.
* Familiarize with cache performance metrics such as cache misses per kilo-instruction (MPKI), bandwidth (BW), and hit ratio (HR).
* Understand the role of each cache level on the system performance.
* Identify the relationship between cache and system performance.

1. **Theoretical Concepts**

**Memory hierarchy organization**

The memory system aims to provide a fast, huge and cheap memory storage space. For this purpose, this system is implemented with multiple technologies in hierarchical way, as depicted in Figure 1. The higher levels of the memory hierarchy, the closest to the processor, provide faster access to the stored information than the lower ones, but they need to be small for speed so few information can be kept. As we move away from the processor, by descending the levels of the memory hierarchy, the memory structures of each level of hierarchy become larger and slower but the cost per bit is much cheaper.

Thanks to the temporal and spatial locality exhibited by the data used by the applications, that is, the data referenced by the processor and stored in the closest levels is likely to be referenced again, this memory organization allows to provide reduced memory data access time as perceived by the processor, so giving the illusion of having a bigger and fast memory system.



Figure 1. Diagram of the memory hierarchy organization.

The quad-core Intel i5 4590 processor available in our labs disposes of 32 KB L1 caches and 256 KB L2 caches, both private for each individual core, and a single shared 6 MB L3 cache.

In this lab session we will familiarize with the different levels of the memory hierarchy studying different memory performance metrics and analyzing the connection between the memory performance and the performance of the applications.

**Hardware performance counters**

Hardware performance counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. Current microprocessors usually implement between four and eight performance counters and support monitoring several hundreds of events related with different structures of the processor core (functional units, queues, branch prediction logic, etc.) and the memory hierarchy. In addition, performance counters can be used without modifying the source code or program binaries and their use do not imply a significant overhead. Performance counters are currently used, among others, to analyze and optimize both hardware and software, to carry out many architecture research studies, and to provide interesting information to smartly schedule the applications.

Many tools (e.g., *perf*) and libraries (e.g. *libpfm*) have been developed to configure and read performance counters. These libraries offer to the users the different functions required to easily set and read performance counters from a user application, allowing them to implement their own applications that make use of performance counters. One of the main issues with performance counters is that he events that can be monitored differ not only among processor manufacturers, but also among different processors of the same company. Despite most processors offer similar measures about many structures such as the memory hierarchy or certain processor structures (even though they might have different event names), the user must ensure the events that a given architecture can monitor to determine whether a particular study can be performed or not on the experiment platform.

**Performance metrics**

The performance of the applications can be characterized according to multiple indexes. In this lab session, we focus on the performance of the memory subsystem and on its impacts on the overall performance. For this purpose, we will evaluate three performance indexes related with the memory subsystem (misses per kilo instruction, hit ratio, and bus transaction rate) along with the IPC of the applications.

The performance indexes evaluated in this lab are:

* Instructions per cycle (IPC). This is one of the reference performance indexes. IPC is calculated as the number of committed instructions divided by the number of execution cycles. It is usually used to compare the performance of different processors. Provided that two systems run at the same frequency, the application or system with the higher IPC provide superior performance.
* *Misses per Kilo Instruction* at L1, L2, and the LLC (MPKIL1, MPKIL2, MPKILLC). This index accounts for the number of misses at a given cache structure for each 1000 instructions committed. It depends on the processor architecture, but also on the data locality of the application and the possible interference caused by co-running processes. The MPKI gives an estimate of the performance penalty due to accessing memory levels further from the processor. Applications with high MPKIs reach lower performance.
* *Hit Ratio* at L1, L2, and the LLC (HRL1, HRL2, HRLLC). HRs are calculated as the ratio between the hits and the accesses to a certain cache level or structure. It is one of the classical indexes used to evaluate the performance of the caches. However, it presents an important problem when trying to relate it with the performance: it does not consider the number of misses. Thus, it does not take into account that a hit ratio by 20% at a cache does not impact on performance if this cache is accessed few times. We can also evaluate the hit ratio of the prefetch requests (e.g., HRL2\_prefetch).
* *Bus Transaction Rate* at L1, L2, LLC, and main memory (BTRL1, BTRL2, BTRLLC, BTRMM). The BTR of a memory structure is calculated as the number of requests to that structure per unit of time (usually presented in microseconds). The main advantage of this index is that it takes into account contention. Hence, an application will see his BTR at a memory structure reduces when other applications frequently access to the same structure.

To evaluate these metrics, we have to measure the number of committed instructions, execution cycles, as well as the number of accesses and misses on each level of the memory hierarchy, for each application being studied. Fortunately, even if the names of these events might vary across existing processors, most processors are able to measure the events considered in this lab. However, when performing other characterization experiments involving more specific events, it is important to make sure that the events to be monitored are available in the experimental system.

1. **LAB SETUP**

In this lab, we monitor the performance of applications running in the system. To avoid any possible interference between the applications to be monitored and the user applications (e.g. the OS user interface, a web browser, or a spreadsheet software), the experiments are launched on a remote server. To this end, a remote server is assigned to each student. This server is accessed through ssh and all the experiments are launched on it[[1]](#footnote-1).

|  |
| --- |
| $ ssh [user@semXXX.upv.es](mailto:user@semXXX.upv.es) |

*Download the scheduling framework and the files required to perform the lab (libpfm library, benchmark binaries and input files). The files can be downloaded from a github repository. Then, compile the libpfm library and the scheduling framework. These actions can be performed issuing the following commands.*

|  |
| --- |
| $ git clone <https://github.com/jofepre/lab_1_files/>  $ cd libpfm-4.8.0/  $ make  $ cd scheduling\_framework/ |

1. **BENCHMARKS**

To develop this lab session and obtain reliable results, we use the SPEC CPU2006 benchmarks with the reference input data set. To reduce the execution time of the experiments, we only consider the first 30 seconds of the standalone execution of each application. To ensure that the same part of the applications is evaluated through any experiment (e.g. running multiple applications at runtime), we measure the number of instructions that each application executes for 30 seconds in standalone execution and set this number as the target number of instructions to be executed in any experiment. Once the application reaches this number of instructions it is considered finished and stopped. All these actions are internally performed by the scheduling framework provided to develop the lab sessions, with the number of instructions set for the target systems where the experiments are to be launched. Thus, students do not need to take care about this issue.

The scheduling framework identifies the benchmarks with an application identifier, ranging from 0 to 24. Table 1 shows the benchmarks and their identifiers. Further information about the benchmarks, including a description of the task they perform, can be checked at <http://spec.org/cpu2006/>. To reduce the length of the lab sessions, we only analyze the performance of the subset of benchmarks marked with asterisk in the table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Índice** | **Integer benchmarks** | **Índice** | **Floating-point benchmarks** |
| 0 | perlbench \* | 12 | bwaves \* |
| 1 | bzip2 \* | 13 | gamess \* |
| 2 | gcc | 14 | milc \* |
| 3 | mcf \* | 15 | zeusmp |
| 4 | gobmk | 16 | gromacs \* |
| 5 | hmmer | 17 | cactusadm \* |
| 6 | sjeng \* | 18 | leslie3d \* |
| 7 | libquantum | 19 | namd |
| 8 | h264ref \* | 20 | dealII |
| 9 | omnetpp \* | 21 | soplex \* |
| 10 | astar \* | 22 | povray \* |
| 11 | xalancbmk \* | 23 | gemsFDTD |
|  |  | 24 | lbm \* |

Table 1. Identifiers for the SPEC CPU2006 benchmarks in the scheduling framework.

1. **The Scheduling Framework**

The scheduling framework is the program used to launch and monitor the performance of the applications. In this lab session, it is used as a *black box*. Experiments are launched with the scheduling framework, setting the input parameters and collecting the experiments results, but its source code is not modified. These activities are developed in future lab sessions.

To launch an experiment, the user should:

1. Select the applications to be run.
2. Select the core on which each application will run.
3. Indicate the events to be monitored. The scheduling framework will set the performance counters to monitor these events for the execution of the selected applications.
4. Report the performance counts for each application per quantum and for the overall execution.

The scheduling framework disposes of four input parameters to set the experiments {*W, C, e, p*}. These parameters are detailed next:

* -W App1[,App2,…,Appn]: it determines the application or workload (set of applications) to be run.

|  |  |
| --- | --- |
| $ ./scheduling\_framework –W 3  $ ./scheduling\_framework –W 3,4,5 | # Runs the *mcf* benchmark  # Runs a workload with benchmarks *mcf*, *gobmk*, and *hmmer* |

* -C C1[,C2,…,Cn]: it determines the core on which each application of the workload will run. More precisely, applications are allocated to a single core according to its index (i.e., App1 will be allocated on core C1, App2 to C2, etc.). The number of applications and cores should thereby coincide. If the set of cores is not specified, the scheduling framework allocates the applications on the first logical CPUs defined by the operating system.

|  |  |
| --- | --- |
| $ ./scheduling\_framework –W 3,4,5 –C 0,1,2 | # Runs a workload where *mcf* is allocated on core 0, *gobmk* is allocated on core 1, and *hmmer* is allocated on core 2. |

* -e ev1[,ev2,…,en]: it indicates the events to be monitored. By default, the scheduling framework only collects the number of execution cycles and committed instructions. The events selected by the user are added to these two events. Notice that the maximum number of events that can be monitored is bounded by the processor architecture (typically between four and six). In addition, some events cannot be monitored simultaneously. This issue will be notified by the scheduling framework if it occurs. Section 5 presents some events that can be monitored and how the complete list of events for the target system can be consulted.

|  |  |
| --- | --- |
| $ ./scheduling\_framework –W 3 –e LLC\_MISSES | # Runs *mcf* and monitors its number of execution cycles, committed instructions and last level cache misses. |

* -p: this flag indicates to the scheduling framework that it should print the events counts per quantum.

Since mots application use the standard output to print their information or results, the scheduling framework prints all its information through the error output. By redirecting the standard output, the users can have a clean output of the information printed by the scheduling framework.

|  |
| --- |
| $ ./scheduler\_basico –W 3 > /dev/null |

**Auxiliary script to launch the experiments**

To reduce the lab session development, the launching\_script.sh is provided. The script uses the scheduling framework to run the applications sequentially, allocating them on the core 0, and monitoring the events specified in the EVENTS variable of the script. This variable should be edited to monitor up to six different events (in addition to the execution cycles and committed instructions, which are always monitored by the scheduling framework).

After each experiment is executed, the script collects and prints the overall value of the monitored events. The collected values are used to prepare charts (using a spreadsheet software or gnuplot) that are required in the following activities aimed at analyzing the memory and related performance metrics.

In addition to printing the overall value of the vents for each application, the script also creates a folder for the experiment (whose name is printed), where it creates a file for each application and saves the dynamic counts of the monitored events for each execution quantum. These results can be used to print a chart that shows the dynamic evolution of the metrics at runtime.

1. **Hardware Events Related with the Memory Hierarchy**

The complete list of the events that can be monitored at the system can be checked launching the following order:

|  |
| --- |
| $ libpfm-4.8.0/examples/showeventinfo |

Nevertheless, since the number of available events is huge, the following list presents the events related with the memory hierarchy that are involved in the development of this lab session.

|  |  |  |
| --- | --- | --- |
| **Event** | **Description** | **Performance metrics** |
| perf\_count\_hw\_cache\_l1d:access | L1 data cache hit accesses | BTRL1, MPKIL1, HRL1 |
| perf\_count\_hw\_cache\_l1d:miss | L1 data cache miss accesses | MPKIL1, HRL1 |
| l2\_rqsts:all\_demand\_data\_rd | L2 cache accesses | BTRL2, MPKIL2, HRL2 |
| l2\_rqsts:all\_demand\_data\_rd\_hit | L2 cache hit accesses | MPKIL2, HRL2 |
| llc\_references | LLC references | BTRLLC, MPKILLC, HRLLC |
| llc\_misses | LLC miss accesses | MPKILLC, HRLLC, BTRMM[[2]](#footnote-2) |
| branch\_instuctions\_retired | Branch instructions completed | BPA |
| mispredicted\_branch\_retired | Mispredicted branch instructions completed | BPA |
| l2\_rqsts:all\_pf | L2 cache prefetch accesses | HRL2\_pref |
| l2\_rqsts:pf\_hit | L2 cache prefetch access hits | HRL2\_pref |

1. **Lab Session Development**

The goal of the lab session is to analyze the described metrics from the experimental data collected in the experiments and identify which metrics present a strong connection with performance. *First, you should launch the experiments required to calculate the memory and performance metrics. To this end, you should edit the launching script to set the events to be accounted and the applications to be monitored. Next, run the script and wait until the end of the experiments. Notice that it should be launched twice with different events because the experimental platform only allows to measure six performance counters at a time.* *Once the data is collected and the metrics calculated, it is important to plot the appropriate charts to reach the correct conclusions. The next subsection will lead you through these steps.*

**Analysis of the memory performance and overall performance for entire executions**

*To perform this study, you should obtain and represent the performance metrics related with the memory system (described in Section 1) and the IPC for the indicated subset of the SPEC CPU2006 benchmarks.* Once the experiments are completed, you will have all the experimental data required to perform the analysis. Using a spreadsheet software, calculate the required metrics. *Then, you should to prepare a chart for each metric, where the different applications are presented on the X-axe and the value of the metric for each application is represented on the Y-axe.* These plots make the analysis and interpretation of the results easier. The remaining of this booklet presents different questions that should answer students to find out the expected conclusions.

* *How do BTRLLCsol and BTRMM affect performance? Study if there exists any connection between a high/low BTRLLC or BTRMM and a high/low IPC.*
* *Which metric related with the L1 cache (HRL1, BTRL1, or MPKIL1) exhibits a stronger connection with performance?*
* *In general, which are the reasons why an application can present a low BTRL1? Benchmarks such omnetppp and milc present a low BTRL1. Study if this value is related with the BTR of other levels of the memory hierarchy.*
* *Observe the hit ratio at the different levels of the cache hierarchy. Which is the cache with higher hit ratio? Why does it happen? And among the L2 and LLC caches? Can you find an explanation for these results?*

**Analysis of the dynamic evolution of the memory and performance metrics**

In this subsection we will study the phase-behavior of the applications and see how the metrics evolve at runtime. *To this end, you should represent charts where the X-axe show the time evolution and the value of the metrics is presented in the Y-axe. The event counts required to represents these charts can be found in the subfolder created when the experiment is run and the file with the application identifier. We will focus on the mcf and bwaves benchmarks. For mcf, represent the dynamic evolution of its IPC, BTRL1, and BTRLLC.For bwaves, represent its IPC, BTRL1, and BTRMM.*

Once the charts are represented, the following questions will guide you through the analysis of the connection that the memory metrics and performance present at runtime.

* *Study the connection among IPC, BTRL1, and BTRLLC of the mcf benchmark. Do the phases with higher IPC correspond with phases with higher or lower bandwidth utilization at the L1 or LLC cache? Explain the reasons of the observed behavior.*
* *Analyze the connection among the IPC, BTRL1, and BTRMM of the bwaves benchmark. Does a higher or lower IPC relate with the bandwidth utilization at the L1 and main memory?*
* *HR, MPKI, and BTR are metrics connected with the cache activity. Explain which metric is better to identify the performance evolution at runtime.*

1. This lab session (including the scheduling framework configuration) has been prepared to be run on an Intel i5 4590. Performance events might differ (in name or implementation) in other architectures. [↑](#footnote-ref-1)
2. To calculate the BTRMM we will assume that the number of accesses to main memory matches the number of LLC misses. [↑](#footnote-ref-2)