**LAB 4. Main memory bandwidth-aware scheduling**

**LEARNING GOALS**

* Realize how improving resource utilization yields to important performance benefits.
* Understand why architecture-aware scheduling can boost the system performance.
* Minimize the interference through a simple scheduling policy and quantify the performance benefits it provides.

1. **Theoretical Concepts**

**Sharing the memory system**

Most microprocessors nowadays are multicore processors, which implement multiple cores on the same chip. Multicore processors provide higher performance than single-core processors while attacking their power, cooling, and package costs issues. The memory system in a multicore processor is shared by all the processor cores. These cores always share the main memory and usually the last level cache (LLC). The L2 cache can be private or shared and the L1 cache is always private to provide a fast access from the core to the stored data. Figure 1 presents the memory hierarchy of the processor used in our experimental platforms. As it can be observer, each core disposes of its private L1 and L2 caches, while the LLC and main memory are shared among all the cores of the processor.



Figure 1. Memory hierarchy of the Intel i5 4590 processor.

Applications running concurrently on a multicore processor can therefore interfere when accessing the shared LLC and main memory. In fact, the main memory is known to be one of the major performance bottlenecks of the system if many applications try to access the main memory frequently. Consequently, a lot of research effort has focused on this problem during the last years.

Several research works have shown that bandwidth-aware schedulers are an effective solution to address the problem of bandwidth contention on commercial multicore processors. The basic idea on these schedulers consists on balancing the bandwidth utilization of the applications along the execution time of a workload. In this way, the scheduler avoids quanta of high bandwidth utilization and contention while there are other quanta where the available bandwidth is wasted. In this lab session we are going to explore to what extent bandwidth-aware scheduler can improve the system performance.

1. **Lab Setup**

In this lab, we are going to 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 will be launched on a remote server. To this end, a remote server will be assigned to each student. This server will be accessed through ssh and all the experiments should be launched on this system[[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\_sched\_framework/tree/master/lab\_sessions/lab\_4/Scheduling\_framework  $ cd libpfm-4.8.0/  $ make  $ cd scheduling\_framework/ |

1. **The Scheduling Framework**

The scheduling framework that we are going to use in the lab session is a simplified version of the scheduling framework designed in the Group of Parallel Architectures (GAP), developed for research purposes and used in many top-conference papers. The scheduling framework is a key component to carry out the lab sessions. Basically, it consists of a user-level scheduler that controls the execution of a workload (a set of application to be executed) on the defined cores while gathering multiple hardware events.

The basic operation of the framework is as follows. When an experiment starts, the framework launches the workload applications and initializes performance counters, setting the events that should be monitored. To this end, it makes use of the functions provided by the *libpfm* library. Then, the framework enters a loop where the following steps are repeated:

1. The process selection policy selects which applications should run during the next quantum.
2. The process allocation policy assigns the applications to the cores using the CPU affinity mask of the Linux processes.
3. Applications run during the quantum length.
4. Once the quantum expires, the framework stops the running applications and reads their performance counters, whose values are printed if required.
5. If the workload execution is completed, the framework prints overall event counts and exits. Otherwise, it goes to step 1.



Figure 2. Framework blocks diagram.

As Figure 2 shows, the framework is composed of three main modules according to the function they carry out: i) performance monitoring, ii) process selection, and iii) process allocation, as depicted in Figure 1[[2]](#footnote-2). This abstraction level simplifies the modifications that can be proposed in advanced lab sessions. In the performance monitoring module, new performance metrics can be calculated from the events counts to guide the scheduling, which is usually performed in two steps: process selection and process allocation. For each step, different policies can be implemented in its corresponding module.

The modular structure of the framework facilitates modifying its source code since it hides the complex internal management of the processes (create, stop, and continue processes) and performance counters (configuration and reading).

To implement the proposed scheduling algorithm, only the modules performance\_monitoring and process\_selection need to be edited.

* 1. **Changes in the Performance Monitoring Module**

The performance monitoring module includes the functions related with the performance counters. Among other functionalities, the functions of this module are used to initialize performance counters, set the events that should be monitored for the processes, and read the value of the performance counters. The module also includes the update\_metrics () function. This function is the best place where metrics related with performance counters can be updated since every time the performance counters are read, the metrics will be updated automatically.

Regarding the performance monitoring module (sf\_performance\_monitoring.h and sf\_performance\_monitoring.c files), you should:

* Configure the memory events to be monitored.

*The events to be monitored can be set in different ways. First, they can just be passed as input parameters of the scheduling framework with the option -e “event1,…,eventN” as explained in “Lab 1: Understanding the Basics on Cache Hierarchy Performance and System Performance”. Second, they can be set in the main function of the scheduling\_framework.c file (line 382). Alternatively, they could also be directly set in the set\_events() function of the sf\_performance\_monitoring module. To properly update the bandwidth utilization later, it is important to be aware of the event number for the events that will be involved in the bandwidth calculation.*

*Since the goal of this lab session is to implement a main memory bandwidth-aware process selection policy, the set of monitored events should include the last level cache misses to calculate the main memory bandwidth.*

* Implement the computation of the main memory bandwidth utilization (in accesses per microsecond).

*As introduced before, the calculation should be done on the update\_metrics () function to be automatically updated each time performance counters are read, and saved on the BW\_MM variable of the nodes. Make sur of using the variables where the event counts of the last quantum are saved (e.g., event\_1 [x]). The value of this variable will then be used by the process selection algorithm to smartly schedule the applications. As discussed in previous lab sessions bandwidth utilization is a better metric than MPKIs since bandwidth is reduced when contention grows.*

* 1. **Changes in the Process Selection Module**

The process selection module includes the functions related with the process selection step of the scheduler. *A new process selection algorithm should be implemented as a different case inside the process\_selection() function. The module also implements auxiliary function that can help performing bandwidth-aware process selection. For instance, calculate\_avg\_MM\_BW\_per\_quantum() calculates the average main memory bandwidth utilization of the applications and the fitness() function can be used to determine which is the process whose bandwidth utilization is closer to the remaining bandwidth per available core.*

*The basic pseudocode of the function is presented next. Checking the random implementation of the policy can also ease the implementation of the new one.*

|  |
| --- |
| bw\_remain = average\_bandwidth\_per\_quantum; // Set BW\_remain to the avg. BW utilization per quantum    cpu\_remain = min (num\_cores, num\_processes); // Set cpu\_remain to the min (cpus, applications)      // The first process of the process\_queue is always selected to avoid starvation  sel = process\_queue.head;  update bandwidth remain // Update bw\_remain subtracting the bw of the selected process  update cpu remain // Update cpu\_remain remain  pull sel from the process queue  insert\_sel into the running queue      // The remaining processes are selected according to their fit to the remaining bandwdith per core  while (available cpus > 0) {    max\_fitness = -1;  max\_node = process\_queue.head; // Avoids max\_node == NULL the first iterations  for (aux = process\_queue.head; aux; aux = aux->sig) { // Check all the available proceses  fit = fitness of aux according the the remaining bandwidth and cpus  if (fit > max\_fitness) { // Save it if it is the highest fit  max\_fitness = fit;  max\_node = aux;  }  }    // Insert the selected node in the running\_queue and update bw\_remain and cpu\_remain  Pull max\_node from the process queue  Update bandwidth remain  Update cpu remain  Insert max\_node into the running queue  } |

Pseudocode 1. Main memory bandwidth-aware scheduling policy

*You should also implement a worst-case scheduler, which will be used study which is the potential performance degradation that a naïve scheduler could cause. Instead of balancing the bandwidth utilization, the worst-case scheduler runs concurrently the processes with highest bandwidth utilization, increasing interference.*

1. **Workload Evaluation**

*Evaluate the workloads presented in Table 1 using the random (-S 1), main memory bandwidth-aware, and worst-case process selection policies. Study the performance benefits that the proposed policy achieves with respect to the other two policies. You can also design your own workloads but to obtain performance benefits (with these small workloads) they should combine applications with high and low main memory bandwidth utilizations. Otherwise, there is no room for the scheduler to enhance the scheduling of a random policy.*

To select the process selection policy, the input parameter -S should be used. Assuming that the main memory bandwidth-aware process selection policy has the identifier 2, workload 1 can be launched with the following command. The command line also assumes that the main memory bandwidth utilization is calculated using the event number 3 (cycles, instructions, and llc\_misses).

|  |
| --- |
| $ ./scheduling\_framework -S 2 –W 8,13,14,24 -C 0,1 –e LLC\_MISSES |

|  |  |  |
| --- | --- | --- |
| Workload | Benchmarks | Identifiers |
| Wk 1 | h264ref, games, milc, lbm | 8, 13, 14, 24 |
| Wk 2 | mcf, sjeng, astar, xalancbmk | 3, 6, 10, 11 |
| Wk 3 | hmmer, astar, leslie3d, lbm | 5, 10, 18, 24 |
| Wk 4 | astar, xalancbmk, bwaves, gamess | 10, 11, 12, 13 |
| Wk 5 | h264ref, bwaves, milc, leslie3d | 8, 12, 14, 18 |

Table 1. Workloads

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. An additional module called sf\_auxiliar includes definitions of structures, global variables and functions to manage the nodes and queues that the scheduling framework internally uses to manage processes. [↑](#footnote-ref-2)