**LAB 5. Process to Core Allocation in SMT Processors**

**LEARNING GOALS**

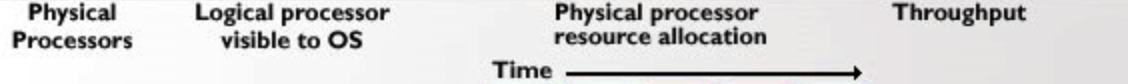
* Identify the connection between L1 bandwidth utilization and performance of application running on an SMT core.
* Study how the L1 bandwidth utilization of an application limits the use of this resource and, consequently, the performance of other applications running simultaneously on the same core.
* Understand how the performance of SMT processors can be boosted by an intelligent thread-to-core allocation policy.
* Design a simple bandwidth-aware thread-to-core allocation policy.

1. **Theoretical Concepts**

**SMT processors**

Multithreaded processors support concurrent execution of multiple threads (or processes) in the same processor. Among them, SMT processors are the only type of multithreaded processors that are able to issue instructions from multiple threads in the same cycle. Most processor manufacturers implement SMT in their high-end products since it is a promising architecture paradigm that offers excellent performance when running a single process and high throughput when running multiple threads or applications.

SMT processors share most of their resources among the threads running simultaneously. Functional units, ROB, instructions queues, and the firsts levels of the memory hierarchy are private in single-threaded cores but become shared among threads in an SMT processor. As Figure 1 shows, this fact potentially increases the utilization of the functional units of the core and consequently its overall throughput.



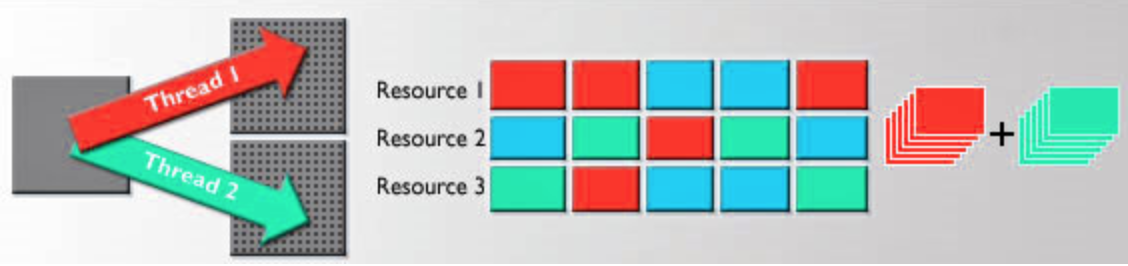


Figure 1. Simultaneous multithreading diagram. By concurrently running tow threads,   
the utilization of the functional units and thus the core throughput grow.

However, scheduling for this architecture is particularly challenging, because SMT performance is very sensitive to the characteristics of the co-running applications. Very important throughput benefits over single-threaded processors can be achieved when running complementary or symbiotic applications on the same core. On the contrary, the throughput of the SMT core can even be worse than that of a single-threaded core when the co-running applications strongly interfere on the shared resources.

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\_5/Scheduling\_framework  $ cd libpfm-4.8.0/  $ make  $ cd scheduling\_framework/ |

1. **Analysis of the Connection Between L1 Bandwidth and Performance**

In this section we are going to analyze the performance degradation due to contention on the L1 bandwidth, one of the most critical resources of SMT cores. *To this end, you should first obtain the evolution of the IPC and L1 bandwidth utilization of two applications when they run alone on the system. Run applications h264ref and bwaves and represent, for each process, both metrics in a chart and analyze whether or not these metrics are connected.*

*After that, you should run both applications concurrently on the same core to obtain how their L1 bandwidth and IPC evolve along their execution.* Launching these experiment is not particularly complicated, but you should put extra attention to the core configuration. Linux organizes the hardware threads of the SMT processors (a quad-core SMT2 processor implements eight hardware threads) as logical CPUs. In general, cpus 0 and 1 should correspond to the core 0, cpus 2 and 3 to the core 1, and so on[[2]](#footnote-2). Taking this issue into account, the experiment can be launched in our experimental platform issuing the following command.

|  |
| --- |
| $ ./scheduling\_framework –W 8,12 –C 0,1 -e perf\_count\_hw\_cache\_l1d:access, perf\_count\_hw\_cache\_l1d:miss |

*Once the experiment finishes, represent the evolution of the L1 bandwidth utilization and IPC of both applications. Analyze the connection between the L1 bandwidth consumption and IPC and both applications. Is there bandwidth contention between both applications?*

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. Please, refer to Section 3 of the “*LAB 4. Main memory bandwidth-aware scheduling”* for further details about the scheduling framework.

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 2[[3]](#footnote-3). 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.



Figure 2. Framework blocks diagram.

To implement the proposed scheduling algorithm, only the modules performance\_monitoring and process\_allocation 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 an L1 bandwidth-aware process allocation policy, the set of monitored events should include the hits and misses to the L1 cache to calculate the L1 bandwidth.*

* Implement the computation of the L1 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\_L1 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 Allocation Module**

The process allocation module includes the functions related with the process allocation step of the scheduler. A new process allocation policy should be implemented as a different case inside the *process\_allocation()* function. The module also implements auxiliary function that can help performing bandwidth-aware process allocation. For instance,the *find\_max\_BW\_L1()* and *find\_min\_BW\_L1()* return the process with highest and lower L1 bandwidth utilization, respectively, among the set of processes selected to run (by the process selection function). Other functions such as *assing\_node\_to\_core ()[[4]](#footnote-4)*, which allocates a process on a core, and *assign\_remaining\_applications()* really facilitate the implementation of the required policies.

*You should implement an L1 bandwidth-aware process allocation policy that minimizes the L1 bandwidth contention. To facilitate the implementation of the new policy, you can assume a simple scenario consisting of two processor cores and four processes to be scheduled. Thus, the L1 bandwidth-aware allocation policy only needs to select the processes with highest and lowest L1 bandwidth utilization and allocate them to the two hardware threads of the same SMT core. Then, the function assign\_remaining\_applications() can be used to assign the remaining two applications to the other core.*

*You should also implement a worst-case process allocation, 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 policy will allocate on the same SMT core the applications with highest L1 bandwidth utilization.*

1. **Workload Evaluation**

*Design a set of 4-application workloads combining applications with high and low L1 bandwidth utilization. Evaluate these workloads using the random (-A 1), L1 bandwidth-aware, and worst-case process allocation policies. Study the performance benefits that the proposed policy achieves with respect to the other two policies.*

To select the process allocation policy, the input parameter -A should be used. Assuming that the L1 bandwidth-aware process allocation policy has the identifier 2, a workload with applications *hmmer*, *h264ref*, *bwaves*, and *gamess* can be launched with the following command. As discussed before, the core selection is also a critical parameter. The set of available cpus should be consistent with the process allocation policy so that if the policy assumes that the first two cpus of the subset correspond to the same SMT core, the configured set of cores should also match that criteria. The command line also assumes that the L1 bandwidth utilization is calculated using the events number 3 and 4 (cycles, instructions, l1 hits, and l1 misses).

|  |
| --- |
| $ ./scheduling\_framework -A 2 –W 5,8,12,13 -C 0,1,2,3 –e perf\_count\_hw\_cache\_l1d:access, perf\_count\_hw\_cache\_l1d:miss |

1. This lab session (including the scheduling framework configuration) has been prepared to be run on an Intel i7. Performance events might differ (in name or implementation) in other architectures. [↑](#footnote-ref-1)
2. The logical organization of the cpus can be different. For instance, the hardware threads of core 0 in a quad-core processor can also be the logical cpus 0 and 4, the ones of the core 1 be the logical cpus 1 and 5, and so on. [↑](#footnote-ref-2)
3. 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-3)
4. The core index of this function refers to the set of available cores configured through the input parameters. Thus, if the scheduler is run with the option -C 1,3,4,5 and *assing\_node\_to\_core ()* is asked to allocate an application on the core 0, it will allocate the application on the cpu 1 (first one in the input string); if it is asked to allocate the application on the core 1 it will allocate it on the cpu 3 (second one in the input string), and so on. [↑](#footnote-ref-4)