**LAB 2. Prefetching and issue stalls**

**LEARNING GOALS**

* Reinforce the issue stage of out-of-order processors. Identify stalls as performance constraints.
* Characterize the issue stalls of a subset of benchmarks running on a real machine.
* Study the connection between issue stalls and processor performance.
* Learn how prefetching can be configured at runtime.
* Study the performance degradation that disabling the prefetcher causes to different applications in a real machine, and how performance losses are reflected as issue stalls.
* Analyze the characteristics of applications with respect to the benefits (or lack of them) that prefetching has on their performance.

1. **Theoretical Concepts**

**Hardware prefetching**

Prefetching is technique widely used in current processors to hide the huge main memory latencies. Prefetching consists on bringing data or instructions closer to the processor before they are actually requested by it. That is, these data/instructions are moved from a distant memory structure (e.g. cache or main memory) to a closer one (e.g. a higher cache level). Prefetching significantly reduce the data access latency in a large amount of applications, and therefore, its execution time. Applications exhibiting predictable memory access patterns are the most benefited by prefetching. This lab shows that the execution time of some applications is reduced in a factor of 3. Thanks to its benefits on performance, current processors use to implement multiple prefetchers.

As Figure 1 shows, as the data is moved from main memory to caches closer to the processor, the total latency (i.e. the elapsed time from the processor to where the data is found) is reduced.



Figure 1. Hardware prefetching schema of the IBM POWER4.

**Superscalar processors and vertical waste of the issue logic**

Superscalar processors implement several issue ports (also known as issue slots) to allow the parallel utilization of the operators and increase the performance of the system. Current high-performance processors usually implement at least six issue ports. For instance, the Intel i5-4590 processors available in our labs implements eight issue ports, as shown in Figure 1. Four ports can issue arithmetic operations, two can perform the address calculation of load and store operations and only through one of them can be transmitted the data to be stored.

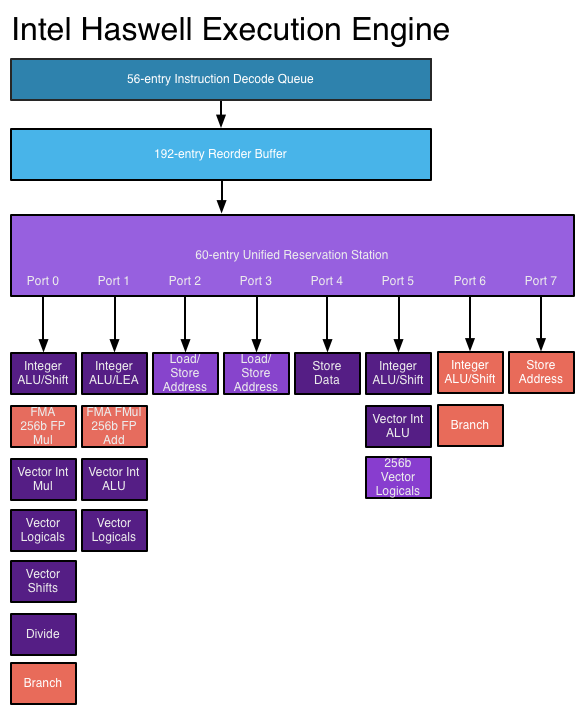


Figure 1. Execution engine on the Intel Haswell microarchitecture processors.

In a given cycle, up to four instructions can be issued provided that they use different issue ports. However, as illustrated in Figure 2, during the execution of the applications there are many cycles where less than four instructions are issued (horizontal waste) and other cycles where no instruction can be issued (vertical waste). In this lab session, we will quantify the vertical waste of the issue logic in the lab computers.

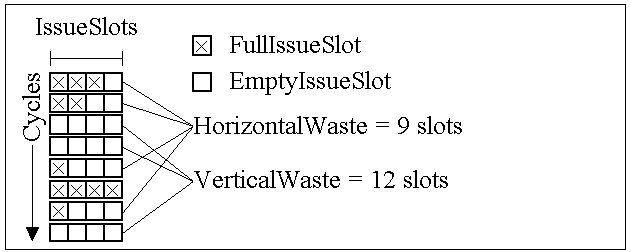


Figure 2. Issue logic waste diagram. Vertical waste happens when no instruction is issued in a cycle.

**Hardware performance counters**

The hardware performance counters are a set of special purpose registers implemented in current processors that allow keeping track of internal processor events such as execution cycles, committed instructions, and requests and misses to the different levels of the memory hierarchy, among many others. For further details, check the “hardware performance counters” section of the “Lab 1: Understanding the Basics on Cache Hierarchy Performance and System Performance”.

**Execution stalls**

Execution stalls are cycles on which the processor cannot proceed on the execution of any instructions, causing a delay in all the instructions that go below. In in-order processors, execution stalls rise when, for example, a data dependence exists between two instructions and the second one needs to wait until its input data is available, blocking the execution of the following instructions.

Accounting execution stalls in out-of-order processors is not a straightforward task, since out-of-order processors are able to forward the execution of younger independent instructions when an older instruction cannot continue its execution. This fact complicates how to define execution stalls unequivocally. For instance, we can consider execution stalls to the cycles where the processor is not able to commit any instruction (execution stalls at the commit stage). The IBM POWER8 processor, for example, accounts the execution stalls at this stage. Nevertheless, we can also consider execution stalls when the processor is not able to issue (stalls at the issue stage) or dispatch (stalls at the dispatch stage) any instruction. Recent Intel processor, for instance, allow accounting both issue and dispatch stalls.

From the foregoing, it can be deduced that the study of the execution stalls depends on the experimental platform and the events that it is able to account for. In this lab session, we will focus on the issue stalls since the processors at our labs allow accounting for the events required.

**Hardware events**

Table 1 presents the hardware events involved in the development of this lab sessions.

|  |  |
| --- | --- |
| Event | Description |
| cycle\_activity:stalls\_l2\_pending | Cycles on which no instruction is issued due to an L2 miss. |
| cycle\_activity:stalls\_ldm\_pending | Cycles on which no instruction is issued due to a load that accesses the memory subsystem. |
| cycle\_activity:cycles\_no\_execute | Cycles on which no instruction is issued. |

Table 1. Hardware events of related with the issue stalls in the Intel i5 processor.

Notice that the event *cycle\_activity:cycles\_no\_execute* includes the event *cycle\_activity:stalls\_ldm\_pending,* which at the same time includes the event *cycle\_activity:stalls\_l2\_pending*. Using the available events, we can classify the issue stalls in three categories:

* Due to loads that miss in the L1 cache and hit in the L2.
* Due to loads that miss in the L2 cache and hit in the LLC or main memory.
* Due to other reasons.

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 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 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_2_files/>  $ cd libpfm-4.8.0/  $ make  $ cd scheduling\_framework/ |

1. **Analyzing the Connection Between IPC and Issue Stalls**

**Connection between IPC and issue stalls with prefetching enabled**

In this section, we measure the IPC and issue stalls of several SPEC CPU 2006 benchmarks to study whether or not there is a connection between them. *In order to perform this analysis, you should first monitor the committed instructions, execution cycles, and the three stall events presented in Table 1 for the standalone execution of the following benchmarks: mcf, sjeng, libquantum, bwaves, games, milc, leslie3d, gemsFDTD and lbm, whose benchmark identifiers in the scheduling framework are 3, 6, 7, 12, 13, 14, 18, 23, and 24, respectively.*

*To ease launching the experiments, you can use the launching\_script.sh, setting the correct events and applications to be monitored. Please, see the Section 4 “The Scheduling Framework” of the Lab 1 booklet for further details about the scheduling framework and launching script.*

*With the data obtained from the previous experiments, represent the following charts:*

* *A bar chart presenting the IPC (y-axis) of the studied benchmarks (x-axis).*
* *A staked bar chart presenting the percentage of stall cycles for the three categories over the total number of execution cycles of the applications.*

*Interpret the results and explain whether or not is there a connection between the issue stalls and performance of the applications.*

**Enabling and disabling the hardware prefetching at runtime**

The model specific registers (MSR) are a set of processor control registers mainly used to set certain processor characteristics and features. Among them, they allow enabling or disabling the different prefetching mechanisms implemented. Prefetching configuration is done using the four less significant bits of the MSR register with address 0x1A4. Table 2 presents the configurable options. A bit set to 0 means that the prefetching mechanism is enabled. To disable it, the corresponding bit should be set to 1.

|  |  |  |
| --- | --- | --- |
| Prefetching mechanism | Bit at MSR 0x1A4 | Description |
| L2 hardware prefetcher | 0 | Fetches additional lines of code or data into the L2 cache. |
| L2 adjacent cache line prefetcher | 1 | Fetches the cache line that comprises a cache line pair (128 bytes). |
| DCU prefetcher | 2 | Fetches the next cache line into L1-D cache. |
| DCU IP prefetcher | 3 | Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines. |

Table 2. Prefetch engine configuration bits at the MSR 0x1A4.

MSRs can be read and write using the *rdmsr* and *wrmsr* Linux commands, respectively. Both instructions take as inputs the cores where the command should run and the address of the target MSR register, plus the value to be wrote in the case of the *wrmsr* command.

|  |  |
| --- | --- |
| $ sudo rdmsr -p 0 0x1A4 | # Read the value of the MSR 0x1A4 for the core 0 |
| $ sudo wrmsr -p 0 0x1a4 0x0 | # Write the value 0x0 to the MSR 0x1A4 of the core 0 |

To prevent any possible error setting the MSRs, we provide the scripts enable\_PF.sh and disable\_PF.sh which enable and disable the hardware prefetching in all the cores, respectively. They must be run using the *sudo* command. After setting the prefetch engine, you should check that the changes have been applied to ensure the correctness of the following experiments.

**Connection between IPC and issue stalls with prefetching disabled**

*Disable the hardware prefetching and repeat the experiments you carried out to obtain the execution cycles, committed instructions and stall events for the evaluated applications. Study IPC and issue stalls depending on whether prefetch is enabled or disabled. Which in the main component of the issue stalls reduced by the prefetch? How does prefetch contributes to reduce this component?*

1. This lab session (including the scheduling framework configuration) has been prepared to be run on an Intel i5 3570. Performance events might differ (in name or implementation) in other architectures. [↑](#footnote-ref-1)