CSC469 Assignment 2
Performance Evaluation

Authors

Jasmeet Brar - 1002342967
Jasmeet Sidhu - 1002469571

October 20, 2019

Contents

1 Introduction

The CPU scheduler is a main component in an operating system that schedules when processes should use the CPU, and it does so in a manner that would increase CPU utilization and increase work throughout. There are many points that the scheduler has to take account for in order to decrease turnaround time and response time, such as having preemptive scheduling, scheduling other process when a process needs I/O, optimally load balance by changing priority of each process, and so on. The task that the scheduler has to put up with is quite complex, and it does leave us wondering on how the CPU scheduler affects the execution of our programs.

Another factor that affects any program execution is the CPU cache, since a program would have significant boost in performance if it is able to access data from the cache than from RAM. Every CPU nowadays is multi-core, and it has multiple caches at different levels. Each core has their own private cache, and some cores may have some caches that are shared amongst themselves. It is quite likely that the largest cache, the one with the greatest latency, which is the last-level cache (LLC), is shared among multiple cores. How big is this cache and how many cores utilize this cache are questions that would come up when thinking about how programs can effectively use CPU cache.

The given assignment aims to resolve our curiousity in these two subject matters. In Part A, we constructed a program that would determine how the CPU scheduler affects the runtime of a given program in a Linux 4.15 system. In Part B, we constructed two programs: one to get the size of the LLC, and the other that determines what cores are sharing the LLC. In both parts, we disclosed the methodologies we had used, any challenges we faced, answers to any followup questions, and the full results of our experiments in this report.

2 Part A: Tracking Process Activity

In this part, we constructed a program that would determine all active and inactive periods of a program's execution, so that one can observe how the scheduler can impact a program's execution.

2.1 Methodology

In order to determine the active and inactive periods of a program's execution, one can read the TSC counter that is available in the CPU in order to measure the current cycle count, and see if it suddenly differs by a large margin. If the cycle count does drastically change, then this would mean that the CPU got preoccupied by another task; it's likely that a timer interrupt had occurred followed by a context switch. This would imply that the process became inactive.

This margin or threshold is going to vary on every machine, due to the differences in architecture. And so this value must be calculated dynamically on the machine the process is running on. This can be done by averaging the cycle counters over the number of iterations we used. Also one would have to jump over any iteration where the difference is far too small or large to remove cache/TLB misses and context switches from the calculation.

One would also have to lock the execution of the process to a core, since the TSC value is going vary on every core. By doing this, the process would have consistent results.

2.2 Results

The program were created, and it was compiled and executed on one of lab computers in Bahen; more specifically it was executed in b2240-09. The specification of the machine is the following [1]:

Table 1: b2240-09 specification

CPU	6 Core Intel(R) Xeon(R) E-2236 CPU @ 3.40GHz
CPU Family and Model	CPU Family: 6, Model: 158
CPU Cache Size	12288 KB per core
Total Memory	32811280 KB (32 GB)

The program was executed with the argument of 20, which is the number of inactive periods being sought. The result was that the program outputted the duration of every active and inactive periods, up until the 20th iteration. The program also produced a csv file containing the starting and finishing timestamps of every active and inactive period (Active is on every odd line and inactive is on every even line).

A Python script was also created to parse the csv file, and generate a script for GNUplot to interpret and produce a graph of the result. Figure 1 is the graph that was produced, and Table 2 shows the starting and finishing timestamps of every active and inactive periods (Shaded rows indicate timestamps for inactive periods).

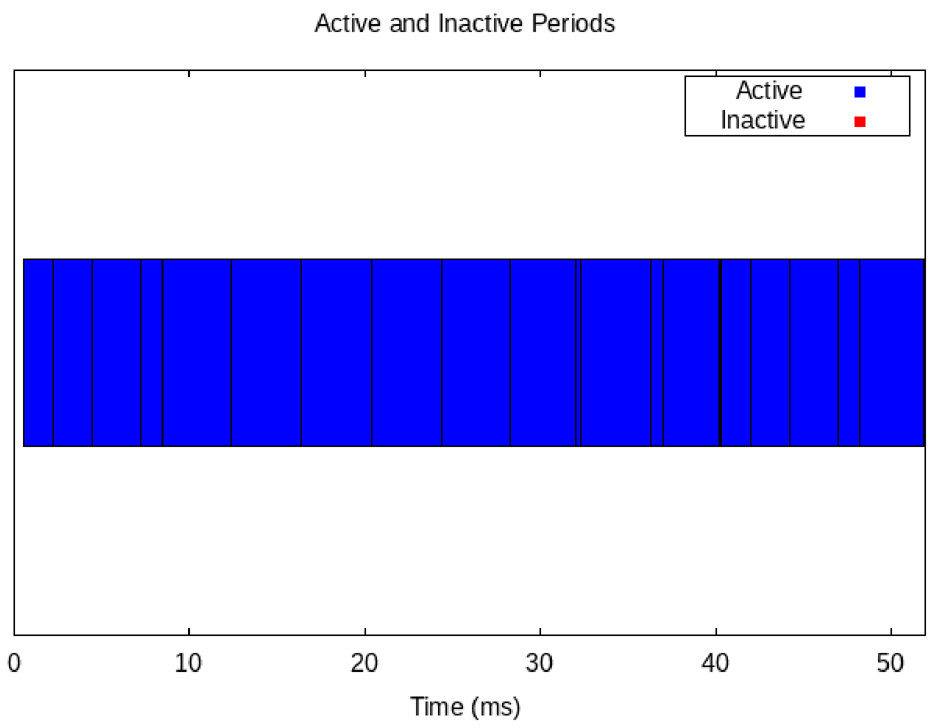

Figure 1: Process that has undergone 20 inactive periods.

Table 2: Active and inactive periods timestamps (all units are in ms).

Start	Finish	Start	Finish	Start	Finish
0.000055	0.463495	16.370051	16.375788	36.257290	36.993818
0.463495	0.492036	16.375788	20.346712	36.993818	37.008159
0.492036	2.179287	20.346712	20.352368	37.008159	40.229915
2.179287	2.193895	20.352368	24.323350	40.229915	40.232969
2.193895	4.440098	24.323350	24.329177	40.232969	41.971807
4.440098	4.446810	24.329177	28.300006	41.971807	41.985405
4.446810	7.177576	28.300006	28.305851	41.985405	44.206537
7.177576	7.192754	28.305851	32.004560	44.206537	44.208983
7.192754	8.416764	32.004560	32.019650	44.208983	46.950254
8.416764	8.422951	32.019650	32.276621	46.950254	46.963378
8.422951	12.393420	32.276621	32.280875	46.963378	48.183190
12.393420	12.399170	32.280875	36.253297	48.183190	48.185289
12.399170	16.370051	36.253297	36.257290	48.185289	51.909972
				51.909972	51.922798

Note: The first active and inactive period timestamps were discarded when plotting the graph, since the first active period actually occurred before the program began measuring.

2.3 Analysis

Based on the results of the experiment, the scheduler is periodically giving off timer interrupts, and causes the program to become inactive due to context switch. However because the experimenter was the only user of that machine, the CPU wasn't preoccupied by another user-level process. Hence the program was able to remain active most of the time.

2.4 Followup Questions and Answers

1. **With what frequency do timer interrupts occur?**

   A timer interrupt occurs roughly every 4ms, after approximately 14000000 cycles, which turns out to be 0.014 GHz.

2. **How long does it take to handle a timer interrupt?**

   A timer interrupt takes about 25000 cycles to be handled, and approximately 0.0075 ms.

3. **If it appears that there are other, non-timer interrupts (that is, other short periods of inactivity that don't fit the pattern of the periodic timer interrupt), explain what these are likely to be, based on what you can determine about other activity on the system you are measuring.**

   There are other very small inactive periods which appear to be interrupts which occur irregularly. Viewing the contents of "/proc/interrupts" before and after running the experiment can help determine which interrupt occurred during the execution of your program; simply see the difference in the interrupt counts of the core the experiment was run on. A majority of the time, "/proc/interrupts" will show the biggest sources of interrupts are hardware devices, in addition to local timer interrupts, TLB shootdowns, function call interrupts, and rescheduling interrupts. Also note that the OS schedules each core to a hardware device, so different cores may have more interrupts coming in if that hardware device is busy.

4. **Over the period of time that the process is running (that is, without lengthy interruptions corresponding to a switch to another process), what percentage of time is lost to servicing interrupts (of any kind)?**

   This can be calculated by taking the differences of the timestamps for the inactive periods (their duration), and dividing it by the the total time for the experiment:

$$\begin{aligned}\%\text{ Time Lost} &= \frac{(0.492036 - 0.463495) + ... + (51.922798 - 51.909972)}{51.922798} \cdot 100\% \\ &= \frac{0.184866}{51.922798} \cdot 100\% \\ &= 0.3560\%\end{aligned}$$

3 Part B: Measuring Cache Behaviour

In this part, we constructed two programs: one to get the size of the LLC, and the other that determines what cores are sharing the LLC.

3.1 Methodology

In order to make a program consistently get accurate results, one must first lock the execution of the program to one core, to prevent having multiple cores from sharing the workload, and disrupting our latency results. Core 0 cannot be used to get reliable results, as it is heavily used by the operating system.

To test the cache, one can test the bandwidth or latency of the cache, and this can be done by accessing an array in memory. However, TLB misses are an issue as it would increase the time for the program to access memory, which would disrupt our results. To ensure that this isn't an issue, every element in an array must be accessed before benchmarking. This would make sure that the TLB has the virtual address of every page containing the array's elements.

Another issue that one would come across is the prefetcher. The prefetcher would prefetch elements into the cache ahead of time, so when the experimenter attempt to measure the time it takes to access an element in higher cache levels, they may end up just measuring the time it takes to access the element from the L1 cache instead. To avoid this issue, one would have to realize that the prefetcher fetches about 128 bytes in advance, and that is equivalent to 16 64-bit integers. So using an access pattern which simply skips a minimum of 16 integers each access in a linear array is enough to surpass the prefetcher, and prevent it from affecting the results; in other words the access pattern looks like: 0, 16, ... array_len - 16, 1, 17, ... array_len - 15, 2, 18, ...

From this, the `perf` tool can be used to collect the data on cache misses and any cache related events. The `run_experiment_B2.sh` script has been setup to use this program, and have the output of the `perf` tool be sent to a csv file, so that it can be used to plot the graph. As for the determining the cores that share the LLC, we would run a few instances of the program, and have them each be pinned to different cores. Then a Python script would be used to gather the results from all the csv files, and then generate some scripts for gnuplot to plot.

3.2 Results and Analysis

For the entire experiment, one of the lab computers at Bahen were used, that have the same specs as the computer used in Part A (refer back to Table 2). These lab computers have 12,256 KB of L3 cache [1]. After running the `run_experiment_B1.sh` script, we generated figure 2, which shows the miss rate of the cache over varying sizes, when the program's accessed blocks of sizes 1MB through 32MB. Even though the LLC is approximately 12MB in size, the miss rate starts to increase around 10MB. This is because the LLC is contains data from other processes and system functions.

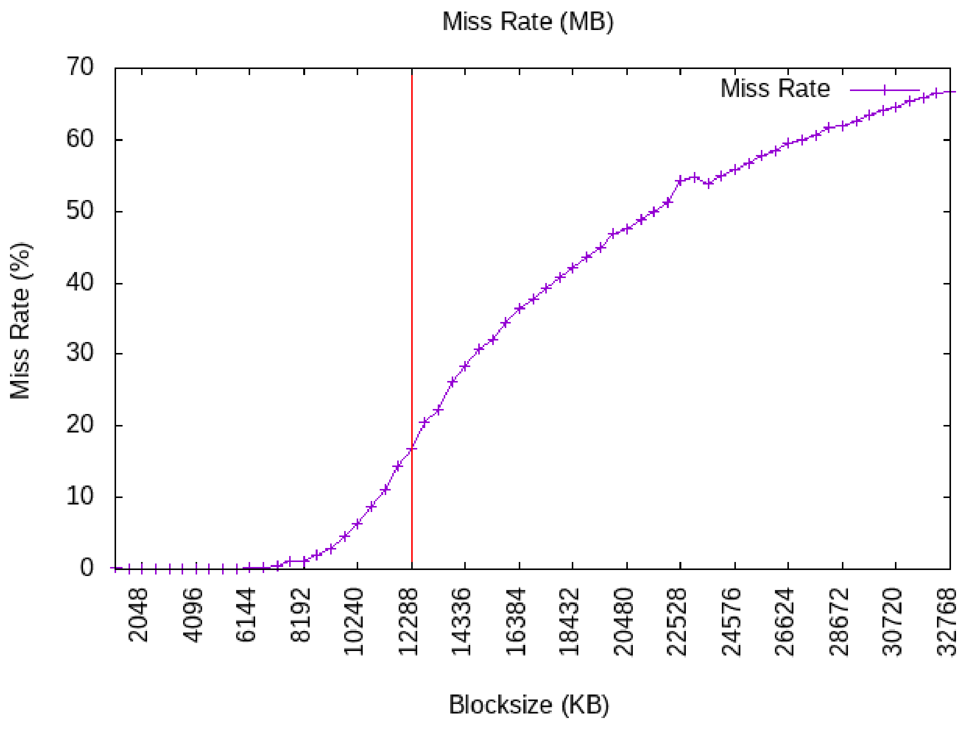

Figure 2: Miss rate vs. Blocksize measured from a single core.

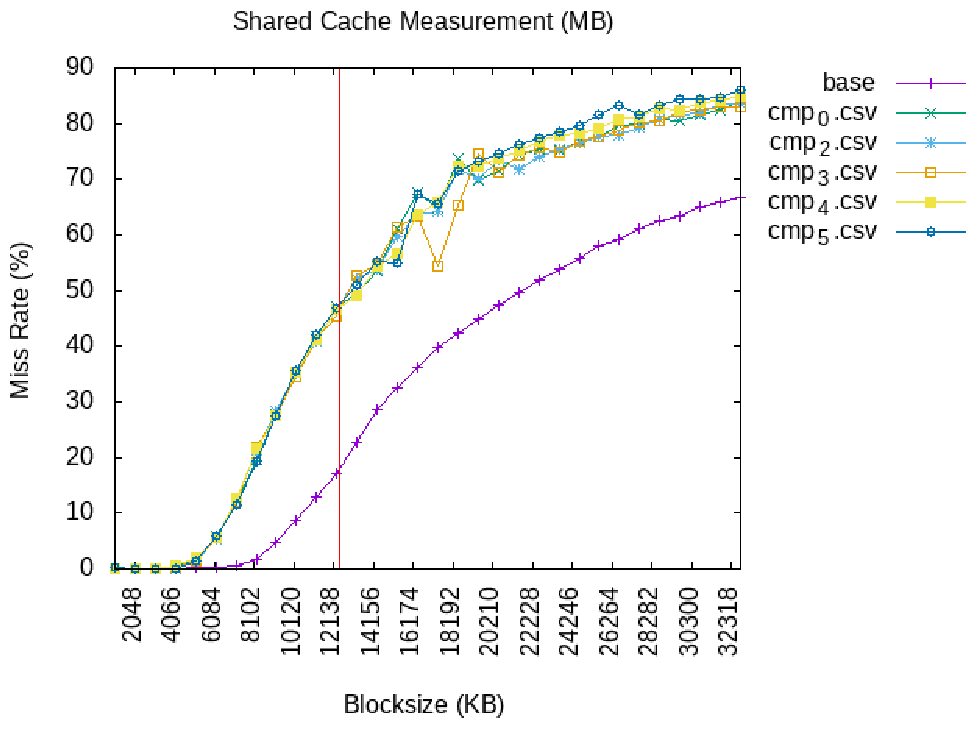

Figure 3: Miss rate vs. Blocksize measured from multiple cores.

The `run_experiment_B2.sh` script determines cores that share the LLC, by first executing the `cache_test` program solely on the target (base) core, and then in parallel with every other core of the CPU ("cmp_x represents base running in parallel with core x"). The results of its executions on the machine are shown in Figure 3. Evidently the LLC is shared among all the cores as the miss rate increases twice as fast with the parallel executions. If the LLC was not shared, the other core would have a similar outcome to the base line.

One point that was noted during the experiment was that when the `test_cache` program was executed with a block size of less than 256 KB, the `perf` tool would end up reporting very large (10% to 60%) and volatile miss rates. Block sizes larger than 256 KB were also quite volatile, but they contained reasonable cache miss rates between 0% and 1%. Due to these issues, this report only discusses block sizes greater than 1MB, but the KB chunk sizes are available and generated in the testing scripts. Small block sizes can be ignored safely because modern systems have LLCs bigger than 1MB. This volatile behaviour likely occurred because the block sizes are so small, that the initial cache misses from warming up the array took a significant portion of the memory references, leading to an exaggerated miss rate.

List of Figures

List of Tables

References

[1] Intel Corporation, "Intel Xeon E-2236 Processor." `https://ark.intel.com/content/www/us/en/ark/products/191040/intel-xeon-e-2236-processor-12m-cache-3-40-ghz.html`. Accessed: 2019-10-18.

4 Appendix

In this section, we list the instructions for running the benchmarking tools for both parts. All the source files for Part A is located in the `process_activity` directory, and all the source code for Part B is located in the `cache_test` directory.

4.1 Part A:

First `cd` into `process_activity` directory, execute the following: `./run_experiment_A.sh <number of inactive periods>`. It will generate two files:

- `results.csv`: file that contains the timestamps of all the active and inactive periods.
- `results.png`: the graph of the timestamps of all the active and inactive periods.

4.2 Part B:

First `cd` into `cache_test` directory.
For part 1 of this section, execute `./run_experiment_B1.sh`. It will generate four files:

- `miss_rate_kb.csv`: file containing cache size in KB followed by miss rate for block size specified with KB as unit of measure.
- `miss_rate_mb.csv`: file containing cache size in KB followed by miss rate for block size specified with MB as unit of measure.
- `cache_bandwidth_kb.png`: graph of results from `miss_rate_kb.csv`.
- `cache_bandwidth_mb.png`: graph of results from `miss_rate_mb.csv`.

For part 2 of this section, execute `./run_experiment_B1.sh <core number>`. It will generate the following files:

- `base.csv`: base results containing cache size followed by miss rate.
- `cmp_<num>.csv`: results containing cache size followed by miss rate for core <num>.
- `data_kb.csv`: merged results from all the `cmp_<num>.csv` files, where it contains cache size in KB, followed by miss rates of all the cores for block size specified with KB as a unit of measure.
- `data_mb.csv`: merged results from all the `cmp_<num>.csv` files, where it contains cache size in KB, followed by miss rates of all the cores for block size specified with MB as a unit of measure.
- `shared_cache_kb.png`: graph of results from `data_kb.csv`.
- `shared_cache_mb.png`: graph of results from `data_mb.csv`.