Introduction

In this report, we look into measuring processor frequency, context switch time, and recording process activity intervals from user space. We also run McCalpin's Stream Benchmark to obtain a few readings quantifying memory access latency and memory bandwidth on a multi-core X86-64bit NUMA system.

System Specs

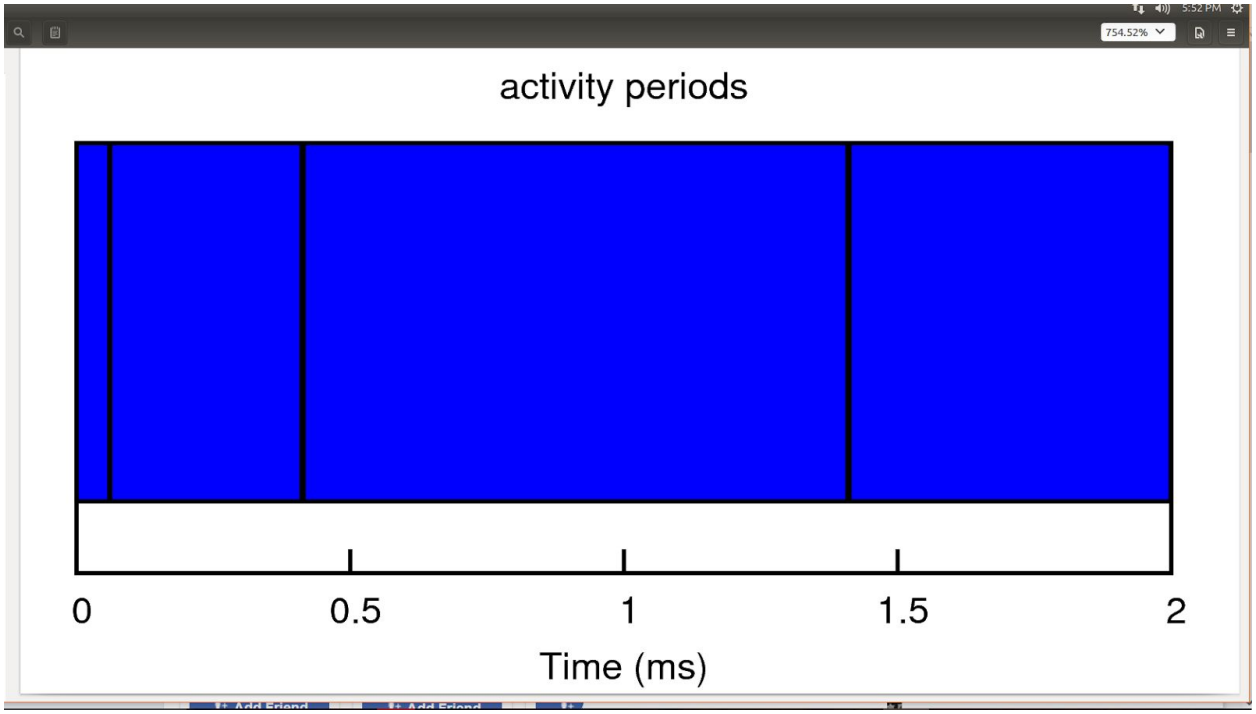
OS Name	Ubuntu 16.04.1 LTS
Processor	AMD A10-7850K Radeon R7, 3700 Mhz, 4 Cores 4MB of shared L3 cache, 1MB of L2 cache
BaseBoard Manufacturer	Gigabyte Technology Co., Ltd.
PCR7 Configuration	Binding Not Possible
Hardware Abstraction Layer	Version = "10.0.14393.206"
Installed Physical Memory (RAM)	8.00 GB
Total Physical Memory	7.95 GB
Available Physical Memory	1.46 GB
Total Virtual Memory	18.9 GB
Available Virtual Memory	11.0 GB
Page File Space	11.0 GB
Hyper-V - VM Monitor Mode Extensions	Yes
Hyper-V - Second Level Address Translation Extensions	Yes
Hyper-V - Virtualization Enabled in Firmware	Yes
Hyper-V - Data Execution Protection	Yes
Storage size:	250GB
Storage type:	SSD

Experiment A - Part 1

For this experiment, we wrote a program tsc.c that measures the approximate frequency in MHz of the CPU in use and measures and records the activity and inactivity periods of the running process. To accomplish this, the program continuously polls on the TSC, the **Time Stamp Counter,** which is a 64-bit register present on all x86 processors since the Pentium. This register counts the number of CPU cycles since reset. The idea is to get to examine the difference in time consecutive reads of the TSC, which is termed threshold in this program. If the difference is large enough to mask out multiple main memory references (100s of ns), then we can infer that the process was interrupted and possibly descheduled. This way we determine when the process becomes active and for how long.
A sample result of running this test:

Active 0: start at 494, duration 48424 cycles, (0.015460 ms)
Inactive 0: start at 48918, duration 9584 cycles, (0.003060 ms)
Active 1: start at 58502, duration 706956 cycles, (0.225705 ms)
Inactive 1: start at 765458, duration 147248 cycles, (0.047011 ms)
Active 2: start at 912706, duration 235576 cycles, (0.075211 ms)
Inactive 2: start at 1148282, duration 3666 cycles, (0.001170 ms)
Active 3: start at 1151948, duration 0 cycles, (0.000000 ms)
Inactive 3: start at 1151948, duration 4404 cycles, (0.001406 ms)
Active 4: start at 1156352, duration 2504572 cycles, (0.799618 ms)
Inactive 4: start at 3660924, duration 9396 cycles, (0.003000 ms)

The plot below shows this result.



This plot represents the activity periods of the parent process. The blue rectangles represent active periods and the red ones (barely visible) represent the inactive periods.

1. There's a regular pattern of every 1ms going by what the plot is showing. So this implies that The CPU timer fires an interrupt every 1ms.

2. In our test, the trace program was bound to a low-activity CPU. This was done to ensure that the process monopolizes the CPU. We did this so we can estimate interrupt handling time using the inactive periods of the process (should be approximately the same in this setup).
Going by the average of these inactive periods, it takes around 3.1us on average to handle an interrupt.

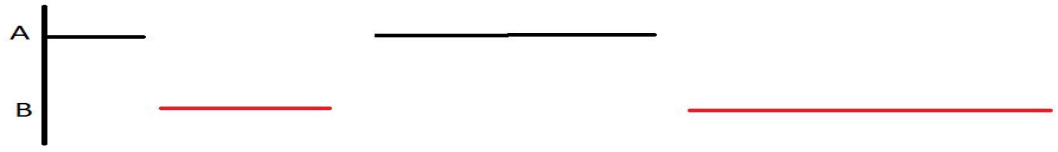
3. Yes, we did trace short periods of inactivity that didn't fit the regular timer interrupt pattern. These are likely triggered by input devices such as mouse and mic.

4. Over the period of process runtime, we computed an average of 0.00262ms per 1ms that the process spends inactive (handling interrupts)

Experiment A - Part 2

.

For this experiment, we wrote a program context_swtich.c that estimates the amount of time a context switch takes. This program spawns a child process and binds both the child and parent processes to a specific lightly-active CPU. This is done using the SCHED_SETAFFINITY() system call. The rationale behind this approach is the following:
If two processes are exclusively running on the same CPU, then we can look at an inactive period of one process to be the active period of the other process + 2 context switches. Thus, if we compute the average active period of say the child process and the average inactive period of the parent process, then the difference between these two averages would be a reasonable estimate of 2 context switch times. The figure below gives you a visualization of of this idea.

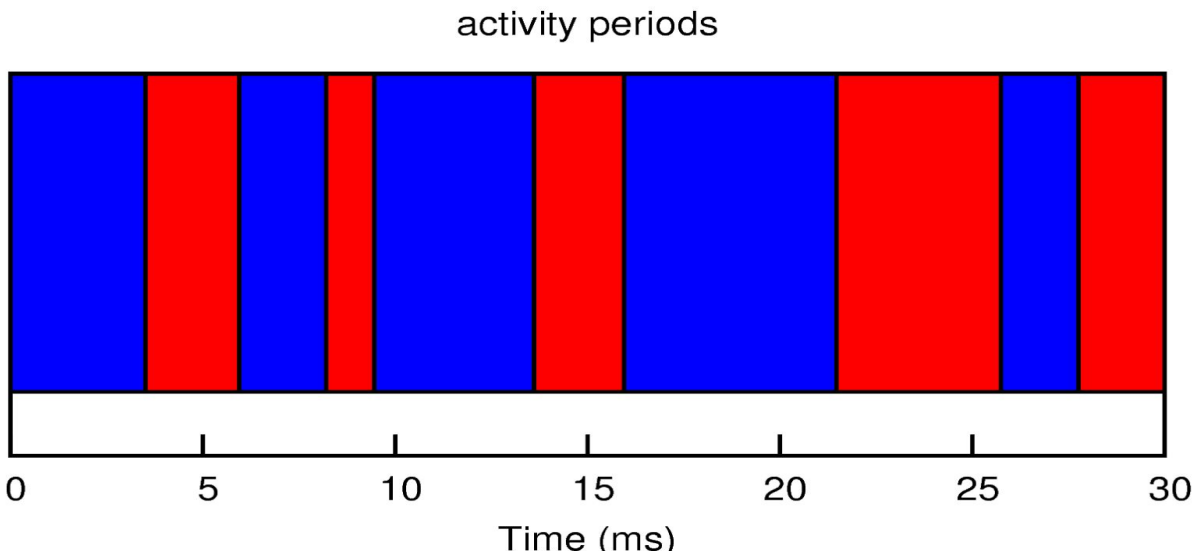


A sample result of running this benchmark:

PID: 42, Active 0: start at 233, duration 2380712 cycles (4.111848 ms)
PID: 42, Inactive 0: start at 2380945, duration 89458 cycles (7.039285 ms)
PID: 42, Active 1: start at 2470403, duration 1193870 cycles (2.011680 ms)
PID: 42, Inactive 1: start at 3664273, duration 87756 cycles (1.04774 ms)
PID: 42, Active 2: start at 3752029, duration 280271 cycles (4.160258 ms)
PID: 42, Inactive 2: start at 4032300, duration 4710 cycles (2.971013 ms)

PID: 42, Active 3: start at 4037010, duration 2342879 cycles (5.993714 ms)
PID: 42, Inactive 3: start at 6379889, duration 86259 cycles (4.178546 ms)
PID: 42, Active 4: start at 6466148, duration 1290712 cycles (2.007500 ms)
PID: 42, Inactive 4: start at 7756860, duration 87901 cycles (3.018899 ms)

Remember this is the activity results of the parent process. The child process is active when the parent is inactive. The child results were not shown for simplicity.
The graph below shows a plot of these activity periods.



This plot shows the activity periods of the parent process. The blue rectangles represent active periods and the red ones represent the inactive periods. This plot was generated by modifying the driver.py script to read input from a .dat file that contained the activity periods of the parent process alone.

In our program, there are only two processes running exclusively on an idle CPU. The amount of time the parent process gets to run before it's descheduled ranges from slightly over a millisecond to around 7ms.

We adjusted our context_swtich.c program, for the sake of this experiment, to spawn 3 children all bound to the same CPU as the parent. The active periods of the parent process were significantly reduced as a result of that.

In CSC369, we learned that the amount of time a process gets to run on the CPU time slice when the algorithm in use is MLFQ is directly affected by the number of jobs in the system and

the nature of the process itself. Our observations in this experiment agree with these technical statements.

Experiment B

For part B we have created several files. Before we get into what was done we give a short description of each relevant file. After which comes the methodology and results.

The Script that creates the data is some-cpus.sh. The data is output onto output.txt. The script has *n* cpus from each node benchmarked against every node. The included text files vary that *n*. The file large-output.txt has that *n* set to 6, medium-output.txt has *n* set to 4, and small-output.txt has *n* set to 2. The script is given where *n* is set to 2.

Our methodology was to run McCalpin's Stream Benchmark against each node but vary the cpu we used. Then we compare the node the cpu was running on and which node's memory we are referencing and compare the cost with the cost given from the --hardware command.

node distances from --hardware command:

node	**0**	**2**	**4**	**6**
0:	10	16	16	16
2:	16	10	16	16
4:	16	16	10	16
6:	16	16	16	10

Best Rate (MB/s) ranges from node (column) to node (row).

node	**0**	**2**	**4**	**6**
0:	4500-6500	3700-4300	2000-3000	1950-2850
2:	2000-3000	4150-6550	3600-4100	2000-3000
4:	2500-4350	2000-3000	4100-6550	2000-3000
6:	2600-4250	2000-3000	2000-3050	4100-6500

The fastest rate range is 4500-6500 and the lowest are 2000-3000. The ranges of numbers occur from the different cpus in each node performing with some variance. If we take the middle value of each range and change it from MB/s to GB/s we can get something easier to compare with.

Best Rate (GB/s) ranges from node (column) to node (row).

node	**0**	**2**	**4**	**6**
0:	5.5	4	2.5	2.4
2:	2.5	5.3	3.85	2.5
4:	2.95	2.5	5.3	2.5

6:	3.4	2.5	2.5	5.3

The problem is these values are speeds not "distances". But if we divide 55 by each number we can get the "distance". (55 was chosen because 55/5.5 =10)

Distance from node (column) to node (row).

node	**0**	**2**	**4**	**6**
0:	10	13.75	22	22.92
2:	22	10.38	14.29	22
4:	18.64	22	10.38	22
6:	16.18	22	22	10.38

The distance from each node to itself is consistent so that was accurate. The approximate distance from a node to another node is 22 not 16 that was given from the --hardware command. Also every node can access another node faster than the other 2. For node 0, it can use node 2 faster. For node 2, it can use node 4 faster. For node 4, it can use node node 0 faster. For node 6, it can use node 0 faster. Additionally, nodes 2 and 0 can access their "closer" nodes faster than nodes 4, 6 can access their "closer" nodes.

In conclusion, the the discrepancies observed where the distance to other nodes should be 22 not 16; each node has 1 node that's closer to it than the others; and the distance between the nodes and their closer nodes is not consistent amongst all the nodes.

Final Notes

Some syntax and implementation parts of tsc.c, context_switch.c, run_experiment_A.py, and run_experiment_A_part2 were found and copied from sources on the web as well as tutorial week 3 material.