CPU Microbenchmark

Yu-Ching Chen   Harris Fok
g3yuch   g5harris

CSC469 - Operating Systems Design and Implementation

Context switching/Threshold value

Introduction

This benchmark will run on a dual core Intel i5-5200U @ 2.2Ghz 64-bit with 2 threads per core. But for this experiment, it will run with hyper-V off, and also have 1 core and 3 threads turned off. The system will run Arch Linux with no other process but everything from systemd (journalctl, networkd, logind, etc..) and other small daemons (full list in list.txt) to ensure low load on the system.

Objective

The objective of this microbenchmark is to understand how the operating system delegates tasks to know when the task is active or inactive. This is done by obtaining approximate cycle counter for context switching. With the cycles for context switching we can bind the program to an approximate threshold cycle. With the threshold cycle we can know when the program is inactive or active.

Methodology

Since we cannot guarantee that the process will only run on one core (unless on single core machine) the code schedules the CPU affinity to 0.

Obtaining the system's CPU frequency is needed in order to determine the active/inactive time periods in ms. This was done by polling the CPU's cycle counter, micro sleeping for a short time, and polling again to get the number of elapsed cycles. We take the average of 10 trials to ensure accuracy. This method gives very consistent and accurate results on the CDF servers and the local machine specified above.

To get the number of cycles for context switching, it has to be done on a low load system to ensure accuracy. With a low load system, the program forks a process and each of the processes outputs the current cycle counter. Both the parent and child will force context switch using sch_yield() to ensure context switching happens, so that even if a process outputs twice, it still guarantees that a context switch happens in between. To ensure no extra load we save the output after the program terminates and find the smallest difference to get the context switches.

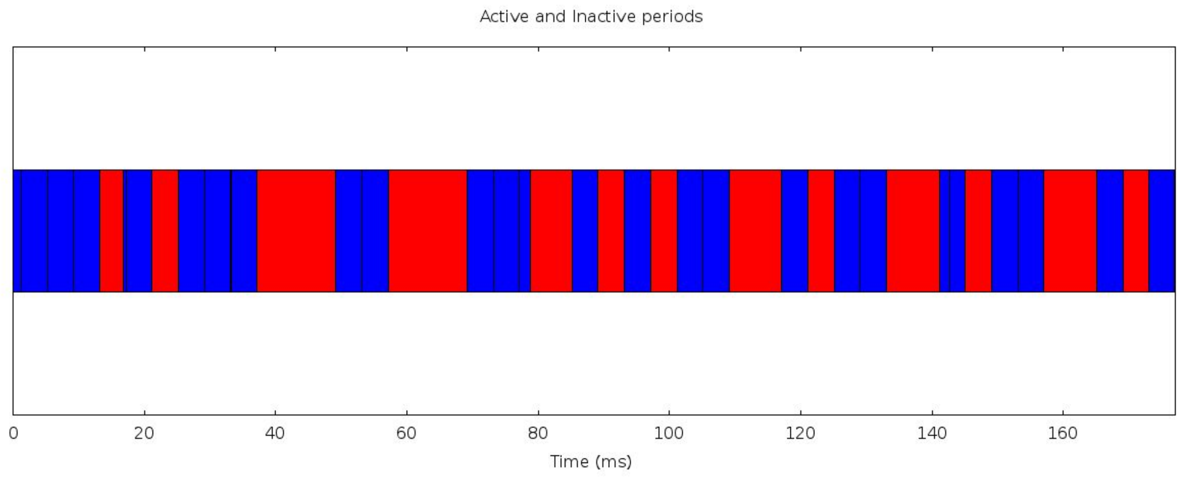
Afterwards, to get the inactive periods, the benchmark enters a loop and constantly polls the cpu cycle counter. If a certain number of cycles above a threshold has elapsed since the previous polled result, it's the inactive period. To get an approximate threshold, we polled the cpu cycle many times and took the difference between 2 cycles and knowing the context switch cycles, the threshold is just a combination of the number of cycles to run the program (loop part) plus context switching. Using the CPU frequency, the program calculates the elapsed time, and outputs these periods.

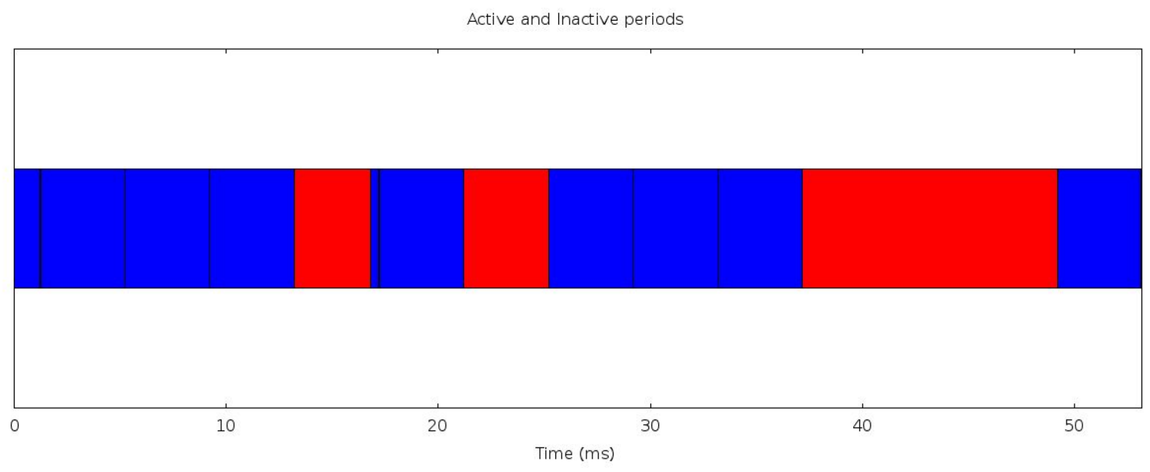
To process the data, a combination of shell, Python, and Gnuplot scripts were used. The shell script simply runs the other two scripts, while the Python script processes and generates the Gnuplot script, and the Gnuplot script plots the data. To process the data, the Python script reads the benchmark output from a file, gets the millisecond durations for each period, and converts it into rect objects for Gnuplot to plot.

Results

For our graphs, blue represents active periods, while red represents inactive periods, similar to those given on the assignment instructions page.

Here are the active/inactive periods measured on CDF.
The first graph plots 30 sets of active/inactive periods, while the second zooms in on the first 10.





And a table of the first 10 sets of active/inactive periods

```
Status    |  Start time  |  Duration  |  Elapsed time
Active    |          95 |    3442306 |   1.228668 ms
Inactive  |     3519430 |      32143 |   0.011473 ms
Active    |     3528065 |   11115351 |   3.967422 ms
Inactive  |    14673453 |      27250 |   0.009726 ms
Active    |    14676870 |   11166540 |   3.985693 ms
Inactive  |    25884906 |      37949 |   0.013545 ms
Active    |    25887776 |   11156007 |   3.981933 ms
Inactive  |    47138378 |   10090778 |   3.601719 ms
Active    |    47140554 |    1103802 |   0.393982 ms
Inactive  |    48304174 |      56099 |   0.020024 ms
Active    |    48307119 |   11137537 |   3.975341 ms
Inactive  |    70690020 |   11240539 |   4.012106 ms
Active    |    70693400 |   11152139 |   3.980553 ms
Inactive  |    81874900 |      26562 |   0.009481 ms
Active    |    81877839 |   11168102 |   3.986250 ms
Inactive  |    93075198 |      26460 |   0.009444 ms
Active    |    93078195 |   11168194 |   3.986283 ms
Inactive  |   137892184 |   33641648 |  12.007773 ms
Active    |   137907516 |   11140519 |   3.976405 ms
Inactive  |   149078708 |      27580 |   0.009844 ms
```

Comments

With what frequency do timer interrupts occur?
As observed in the graphs, on the CDF servers, timer interrupts appear to occur every 4 milliseconds.

How long does it take to handle a timer interrupt?
The interrupts are usually handled in about 26-27k cycles, or just under 10 microseconds.

If it appears that there are other non-timer interrupts, explain what these are likely to be.
For the most part, we ran our tests on a lightly loaded system, and during times that CDF wasn't heavily loaded, so the interrupts in our results were regular timer interrupts.

Over the period of time that the process is running, what percentage of time is lost to servicing interrupts?
Since the majority of interrupts are regular timer interrupts, the time of 10 usec lost every 4 ms comes out to 0.25%.

How long is a time slice before a context switch occurs?
In a different version of the context switch test, we observed that on the CDF servers, a time slice typically lasts for about 12 milliseconds, or 3 ticks of the timer interrupt. However, it was decided that the sch_yield() method produces better results for this test. The graph can instead be found here.

Other comments
Outputs for the local machine is saved in output.txt and output2.txt. Where output is the output for running ./parta 10 and output2.txt is the output of ./parta --contextswitch 20. Also list.txt is the output of all the daemons running on my system, as the program was run on init.
2300 cycles was chosen for a threshold value as running the process on low load gives around an active cycle of 2000 where around 1000 is context switch and rest is to run the program and overhead of context switch from TLB misses etc.
After a bunch of tests, microsleep 10000 gave a good approximation for cpu frequency, as any lower than that varies too much (too much inaccuracy).

Numa access benchmark

Introduction

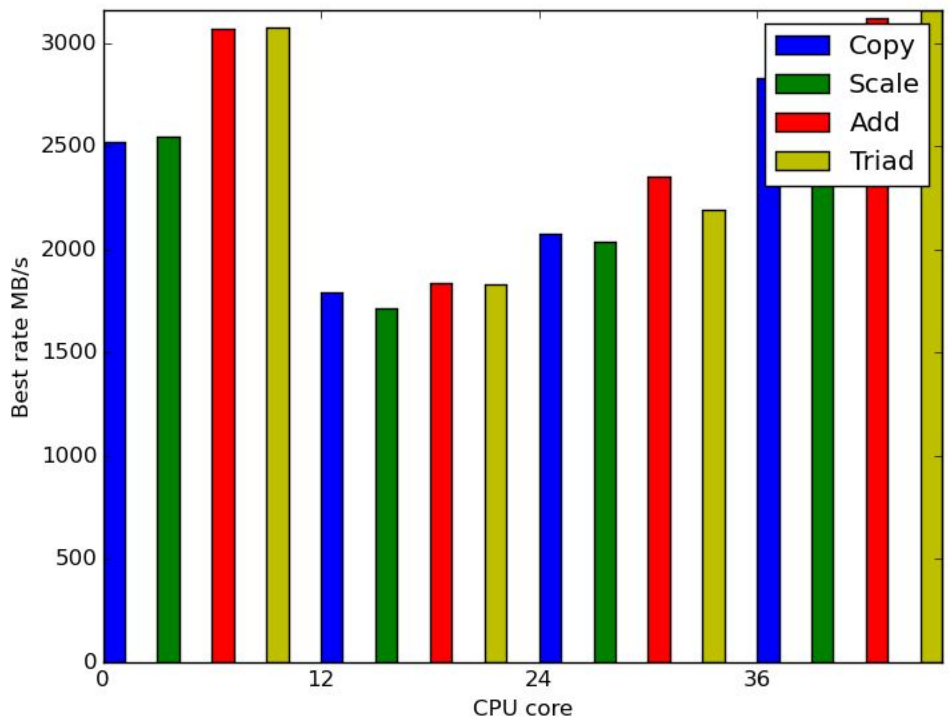
The CDF server has 48 cores and 4 numa nodes. Cores 0-11 share the bus and memory in numa node 0, 12-23 in numa node 2, 24-35 in numa node 4 and 36-47 in numa node 6. The cpus in the same numa nodes are supposed to have the fastest latency from sharing memory between each other as the distance is the shortest, while different numa nodes have an higher amount of latency. For the CDF server specifically, a cpu accessing memory in its own numa node has the distance of 10 while accessing memory from a different node has the distance of 16. Therefore this paper is about benchmarking the numa nodes of the CDF paper.

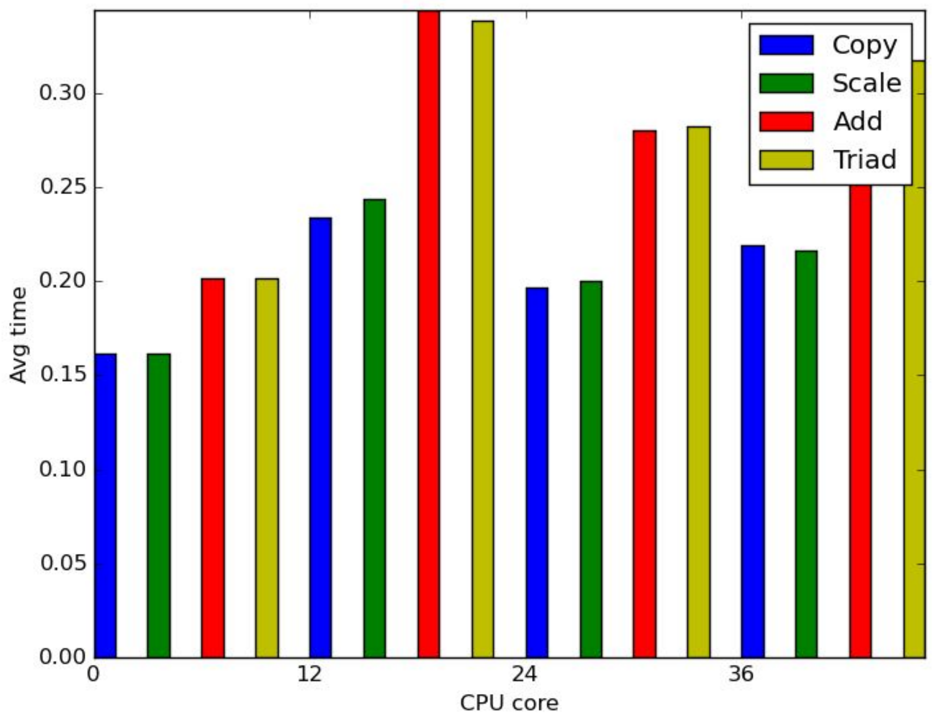
Methodology

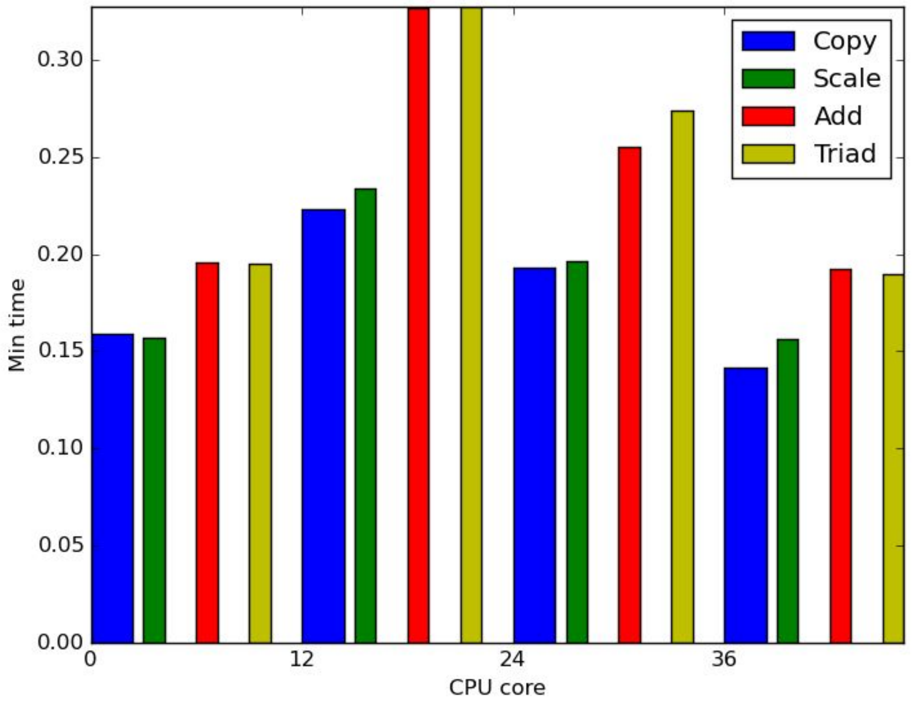
Since this paper focuses on the performance of the numa nodes, the experiment tests the latency between a cpu on each numa node and numa node 0. Given a program that measures transfer rates in MB/s for simple computational kernels, and numactl to bind the cpu and numa node the program outputs a list of the time it takes all numa nodes communicating with numa node 0.

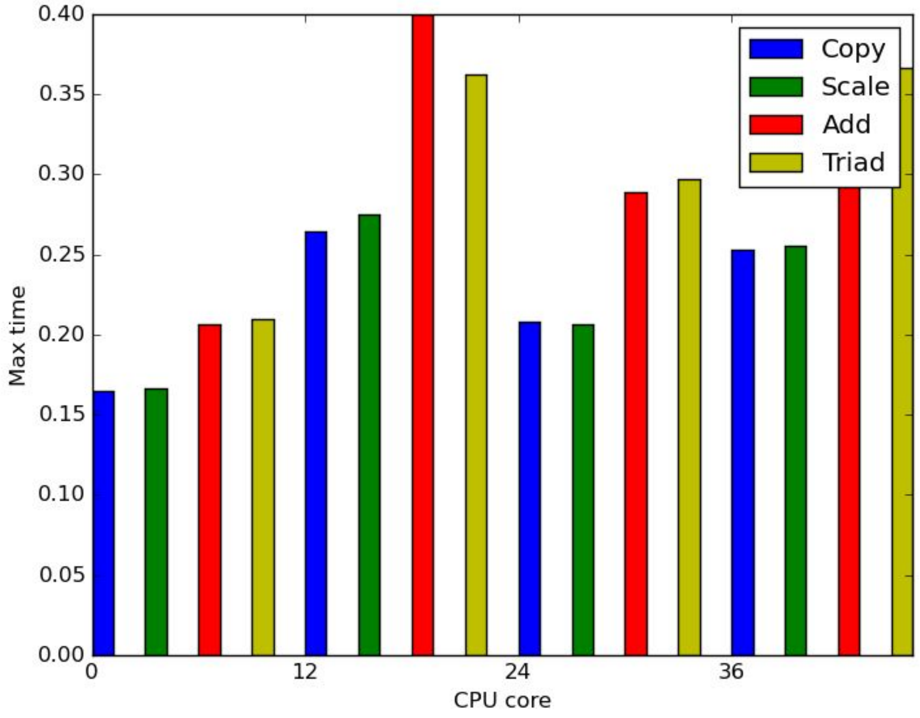
Results

The next two pages has the results of the numa benchmark. Most of the benchmark seems to correlate with the distances with the numa nodes, where less distance results in faster transfers leading to faster times. But what was interesting was that even though some nodes that had longer distances, they had faster communication than cpus in the same numa node.


Copy
Scale
Add
Triad
Best rate MB/s
CPU core
0
12
24
36
0
500
1000
1500
2000
2500
3000


Copy
Scale
Add
Triad
Avg time
CPU core
0
12
24
36
0.00
0.05
0.10
0.15
0.20
0.25
0.30


Copy
Scale
Add
Triad
Min time
CPU core
0
12
24
36
0.00
0.05
0.10
0.15
0.20
0.25
0.30


Copy
Scale
Add
Triad
Max time
CPU core
0
12
24
36
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40