CSC469 A1: Benchmarking

Cheung, Eugene Yue-Hin (cheun550) and Snyder, Eric (snyderer)

2016-10-14

1 Introduction

The performance of computer processes can be affected by many greatly varying factors. In this assignment, we performed experiments to explore the sources of interference when a process of interest is running, along with the effects of non-uniform memory access (NUMA) on memory performance. We performed our tests on the Computer Science Teaching Labs's wolf server (`wolf.teach.cs.toronto.edu`), which is currently running Linux 4.4.0-36-generic x86_64. The following table details some relevant specifications of the server as gathered from `/proc/cpuinfo`, `/proc/meminfo`, `lscpu`, and `GETCONF`:

CPUs	4 × AMD Opteron 6348 2.8 GHz 12-Core G34 Processor
CPU cache size	2048 KB
CPU TLB size	1536 4K pages
L1d cache	16K
L1i cache	64K
L2 cache	2048K
L3 cache	6144K
Cache line size	64 bytes
NUMA nodes	4
NUMA node0 CPUs	0-11
NUMA node2 CPUs	12-23
NUMA node4 CPUs	24-35
NUMA node6 CPUs	36-47
Total memory	65951744 kB
PAGE_SIZE	4096 kB
VmPTE	28 kB

Table 1: wolf specifications.

2 Part A: Process Interference

To explore the timer interrupt and context switch performance of a Linux system, we developed a tool to measure the performance of several basic operations on x86 systems.

2.1 Tracking progress activity

2.1.1 Hardware

The experiments for this part were carried out on both local CS Teaching Lab machines (2.8GHz) and the wolf server. Before the activity program is executed the CPU affinity is set to CPU 1 to get consistent cache performance. The code for this can be found in `common.c`.

2.1.2 Methodology

For determining CPU frequency, we used the recommended `nanosleep` command as recommended by the professor and TA. Our research[1] found that `nanosleep` has a slight amount of overhead to each call, but because the overhead is only a relatively small number of nanoseconds, we increased the sleep time to about 0.1 seconds (10^8) compared to our initial tests with sleeping for 1 nanosecond (10^6). Our results immediately beacme consistent 100% of the time, always getting exactly 2.8GHz on both the local machine and wolf (which both have 2.8GHz CPUs). Initially, we performed more trials and found an average from them, then later moved on to using k-best while still sleeping for one nanosecond. After switching to 0.1 second sleeps, this was no longer necessary as our results would always be accurate after taking a simple average and rounding, so we no longer take the k-best results because it was unnecessary.

To determine the optimal threshold, we first had to consider what exactly the threshold needed to be above to be considered "accurate" and whether it should be a constant number or vary. The first step was to do a series of `get_timer()` calls and take a look at how long it actually took to do two subsequent calls. Generally, it was found that the first cold access would take about roughly 110 CPU cycles (`start_counter -> a = get_counter -> b = get_counter`.) It would quickly settle down to be around 56 cycles for each `get_counter`. Considering this, and the fact that the first assignment took around 100, we knew that the number had to be above 100.

Next, we had to consider what kind of misses could occur, since we didn't want to mistake a TLB or cache miss for an inactive period. Thankfully, Linux makes system information readily available and found the cache sizes (refer to Table 1). These are the caches available to the CPU, additionally the size of one line in the CPU cache is 64 bytes (more on this later). Theoretically this indicates that accessing beyond 64 bytes would cause a cache miss. The PAGE_SIZE is the size of one page miss and the amount of page space our program was using during execution.

It seemed safe to say we couldn't cause a page during regular execution, so the only concern would be a cache miss, but how long does a cache miss take? We devised an experiment based on the above cache line size to make a matrix that had `CACHE_LINE_SIZE / sizeof(int)` elements per row, and some number of rows. The theory was that we would start the timer, access element 0 and then access element 1. The first counter should show the time it took to do a miss while

[1] `http://www.martani.net/2011/07/nanosleep-usleep-and-sleep-precision.html`

the second will be after the row had been cached (since C caches arrays in row-order). For this experiment, we would additionally randomize the cell accessed instead of only 0 and 1. Our results from this experiment were a little surprising, there was a reasonable difference for the first few accesses to the array but quickly the times started to normalise. We believe this illustrated how the L caches work, move the entire array into the cache with each row being in a higher cache. While the second access was certainly quicker eventually the whole array was cached in some lower level cache and could be accessed pretty quickly no matter what.

Moving on, we made the rows and columns of the array much bigger (800 by 800 elements) and continued to access random positions in the row. Still doing one access per row, we quickly had a set of 800 points of data to use to determine a threshold. Observing the data there would be some heavy outliers (where the time delta between the two accesses were incredibly small, indicating the random value was positioned very close to the previous access and already cached) we decided to clear out elements that were very small (anything below 100, a number we established earlier) and anything over 100,000. Sometimes because of the aforementioned caching and/or interrupts occurring the first access may be faster than the second, leading to an overflow when subtracting the two unsigned numbers which would skew the data if left in. Legitimate numbers tended to be somewhere in the range of 10,000 - 20,000 cycles. After sorting, we were still armed with a set of around 600-700 data points which we could then take an average of. Generally this number would be around 14,000 cycles.

The actual program used this CPU frequency and threshold to run the `inactive_periods` function and print out the results to the terminal as specified. We would always run this with affinity set to CPU 1 to avoid inconsistent results.

2.1.3 Conclusions

1. With what frequency do timer interrupts occur?

   Running the program multiple times, you begin to commonly see an interrupt after 3.9 ms of activity. If the period of activity is less than that, usually there will be two smaller periods of activity close to 3.9 ms in between another, unrelated interrupt:

   ```
   Active 8:  start at 76036242, duration 11167030 cycles (3.983506 ms)
   Inactive 8:  start at 87203272, duration 46203 cycles (0.016482 ms)
   Active 9:  start at 87249475, duration 11154045 cycles (3.978874 ms)
   Inactive 9:  start at 98403520, duration 39611 cycles (0.014130 ms)
   ```

   The above snippet of data shows this occurring twice in a row.

   ```
   Active 1:  start at 8838952, duration 3204304 cycles (1.143040 ms)
   Inactive 1:  start at 12043256, duration 26448 cycles (0.009435 ms)
   Active 2:  start at 12069704, duration 7931276 cycles (2.829247 ms)
   Inactive 2:  start at 20000980, duration 29586 cycles (0.010554 ms)
   ```

   Here you can see a shorter active period followed by a short inactive period and another smaller active period. 1.1 + 2.8 adds up to the familiar 3.9 figure.

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

   Referring again to the above data we see that during the inactive cycles between two active periods tends to be around 0.01 ms (0.016482, 0.014130, 0.009435, 0.010554.)

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.

   When running the `inactive_periods` function, the program will not make any intentional system calls, such as printing or writing to a file, so the interrupts were probably unrelated to these. The tests were performed with Mozilla Firefox and a file manager running in the background so function call interrupts could have occur ed. A look at `/proc/interrupts/` indicates that this did occur, along with TLB Shootdowns, Rescheduling Interrupts, IRQ work interrupts, Performance monitoring interrupts, etc. in addition to the expected timer interrupts. The TLB interrupts probably happen earlier in the program when determining the threshold (since it tries to force misses by accessing a massive array randomly.) The other interrupts in `/proc/interrupts/` could also occur during the runtime, such as the rescheduling and performance monitoring interrupts.

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)?

   Based on the provided trace from activity you see 34.968011 ms spent running the program and 0.14855 ms spent inactive for a total of 35.116561 ms running time. About 0.004% of the running time was spent handling interrupts during this trace.

2.2 Context switches

We then attempt to measure the context switch time.

2.2.1 Methodology

In the file `ContextSwitch.c` is where the majority of the code for this experiment lies (besides the common functions from Part A in `common.c`.) The process starts the CPU timer and then forks (so that both processes are using the same CPU timer) and splits into two paths for the parent and child.

The parent will run through until *num* inactive periods over *threshold* (same as part A) and then wait for the child. The child will run through *num* inactive periods over *threshold*, print then return. The parent will stop waiting, print, and return. The parent and child both append whether they are the parent or child to their respective print statements so that graphs can be generated based on the data. Because they share a similar CPU timer when one process yields so another can run, it will be apparent because there will be a matching inactive period in one process where another was running. The script supports running with any number of children but we primarily ran it with 2 as it made it easier to illustrate the differences in graphs.

The system tests on was the same as part A.

2.2.2 Conclusion

Lets take a look at the provided trace:

```
Active parent 0:  start at 687054, duration 319812 cycles (0.114138 ms)
Active child 0:  start at 46024493, duration 10984493 cycles (3.920262 ms)
Inactive parent 9:  start at 103467888, duration 21215 cycles (0.007571 ms)
Inactive child 9:  start at 180213581, duration 33661 cycles (0.012013 ms)
```

Immediately, we see that parent started way before child started, but child also started before parent finished, meaning that two two must have run at the same time.

```
Inactive parent 5:  start at 45808788, duration 33673332 cycles (12.017696 ms)
```

During this period parent was inactive for a very long time, looking above we can match the CPU cycle from child's 0 to that inactive period for the parent. The number of cycles between when parent 5 starts it's inactive period and when child's active period 0 starts is 215705, considering the 2.8GHz processor this is tested on that will take around 0.07ms on the context switch.

Generally speaking, looking at the provided trace and graph, you can see a context switch happened after roughly 10 ms of runtime when running two processes (parent and child) which is more than the timer period found in part A. Consider that the OS tested on implements the Completely Fair Scheduler as their default process scheduler. The CFS supports more granular time slices depending on the number of processes running. When running our part B script with 4+ processes we begin to see the time slices more closer resemble the above number. In the case with two processes the 10 ms number reflects the constant `HZ` defined in `asm/param.h` and in `sched/rt.h`[2], which is the internal kernel timer frequency. After further research[3], we discovered that the time slices in the Linux kernal are flexible depending on the work load as theorised before. The data we've collected also indicates this, with two processes showing a quanta similar to the default while adding more processes starts to shrink to accommodate more processes.

Compared to previous OS courses where we talked about somewhat naive schedulers that had more strict time quanta, the Linux scheduler appears to be more adaptable to the load of the system which makes sense since the examples in CSC369 were supposed to be illustrative of the process, while the Linux scheduler tries to be more optimal and fair.

3 Part B: Measuring NUMA Effects

To explore the effects of the hardware memory organisation on application performance, we specifically measured the effects of NUMA on the performance of standard memory benchmarks. It was expected that memory access from within a node would perform the fastest, while accessing remote memory (i.e. memory in a different node) would result in higher latencies.

3.1 Methodology

To test the effects of NUMA time, we ran our experiments on the wolf server as well which uses four interconnected AMD processors with the Magny-Cours architecture. Within a Python 2 script (`partb_data`), a subprocess was spanwed where `numactl` was executed with the provided precompiled `mccalpin-stream` binary as the command. `numactl` was called with the `membind` argument always set to 0 while the `physcpubind` argument varied. The test was run sequentially

[2] `http://stackoverflow.com/questions/16401294/how-to-know-linux-scheduler-time-slice`
[3] `http://permalink.gmane.org/gmane.linux.kernel/1346063`

four times, choosing a random core from each of the four nodes. Each test was also preceded by a call to `numactl --hardware` to record the free memory available at the time, although this data was left unused. The results then parsed and outputted to a CSV (comma-separated values) file. Table 2 shows a sample of the collected data – the results from the first run, to be specific. To save space, the columns for nodes 2/4/6's free space and the results for the scale, add, and triad tests are not omitted from these results.

CPU	Time	N0 Free	Copy: Best	Copy: Avg	Copy: Min	Copy: Max
8	1476115806.0167	3843	4683.1	0.090395	0.085414	0.097353
13	1476115813.39232	3852	2928.7	0.136969	0.136581	0.137167
24	1476115818.81704	3852	4122.7	0.097143	0.097024	0.097264
45	1476115824.95741	3852	3504.6	0.114516	0.114136	0.115149

Table 2: Sample of collected data.

The resulting CSV file was later parsed with a second Python 2 script (`partb_plot`) to generate a bar graph of core occurrences to ensure roughly equal distribution (see Figure 1) and a boxplot graph of the four nodes' bandwidth timings (see Figure 2). The four best times from the results of the `mccalpin-stream` program (for the copy, scale, add, and triad tests) were averaged and used as the data points in the boxplot to provide a rough sense of what bandwidths were achieved in the tests. Using a boxplot, we are able to easily see the quartiles and outliers. The graphs were generated using `gnuplot`. Boxplots were generated with `gnuplot`'s default settings (i.e. its default statistical calculations).

In this experiment, the test script was run 256 times over a period of approximately 24 hours. Aside from the first couple of test that were run at sporadic intervals (while setting up the cron job to repeat the test), tests were run every 5 minutes. This resulted in a total of 1,024 rows of data. We can see in Figure 1 that all 48 of the available cores on the CS Teaching Lab server (cores 0 through 47) were sampled from multiple times at a relatively even distribution. It should also be noted that while the first script to gather data was run on the Teaching Lab server (with Python version 2.7.5), the script to generate the graphs were run on a local machine with macOS 10.12, Python version 2.7.10, and gnuplot version 5.0 patchlevel 4.

3.2 Results

The aggregated results from the 256 tests can be seen in Figure 2. The raw data was not filtered in order to achieve as accurate of a graph as possible. As such, outliers can be seen in the graph for nodes 0, 4, and 6, which can likely be explained by higher server load during those tests resulting in lower bandwidths achieved. As expected, memory accesses from node 0 were the fastest (i.e. accessing local memory has lower latency than accessing remote memory). Although the other three nodes were slower, we can see that nodes 4 and 6 performed similarly, while node 2 performed the slowest.

Figure 3 offers more insight into the average timings for the various tests run by `mccalpin-stream` on each node. This corresponds with what we see in Figure 2, where Node 0 performed the fastest (with times hovering around the 0.1μs mark) and Node 2 performing the worst (with times ranging from around 0.15μs to 0.2μs). Like in Figure 2, Nodes 4 and 6 have similar results, with times ranging from around 0.1μs to 0.15μs.

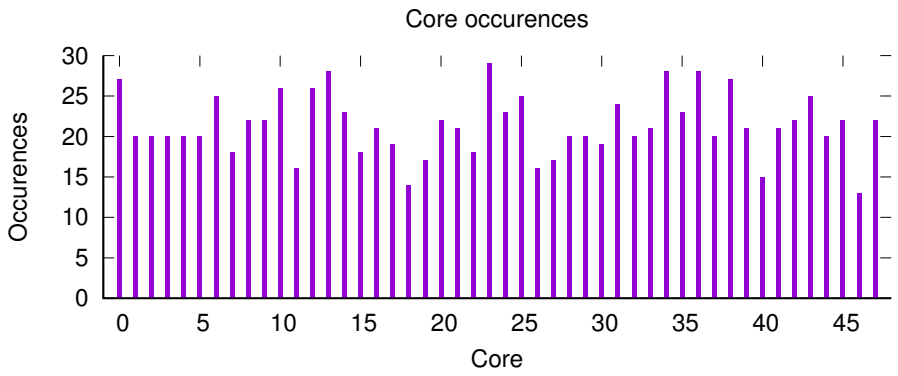

Figure 1: Occurrences of the randomly selected cores in the tests.

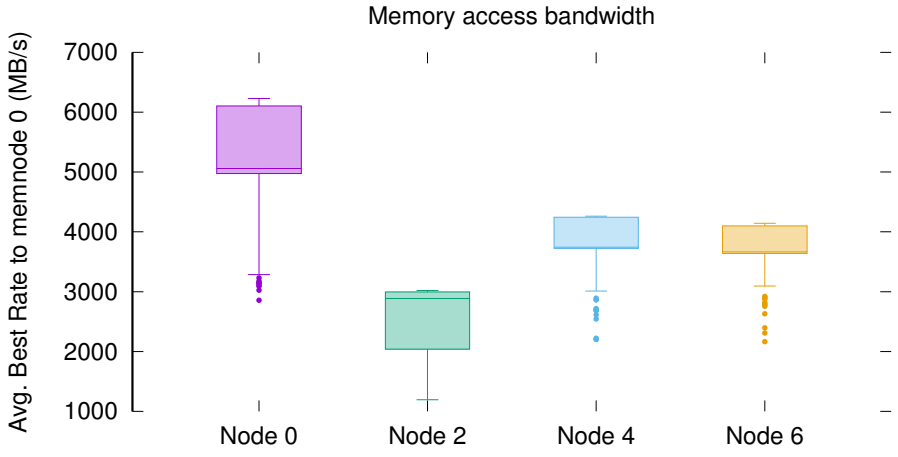

Figure 2: Bandwidth of memory access from the different nodes.

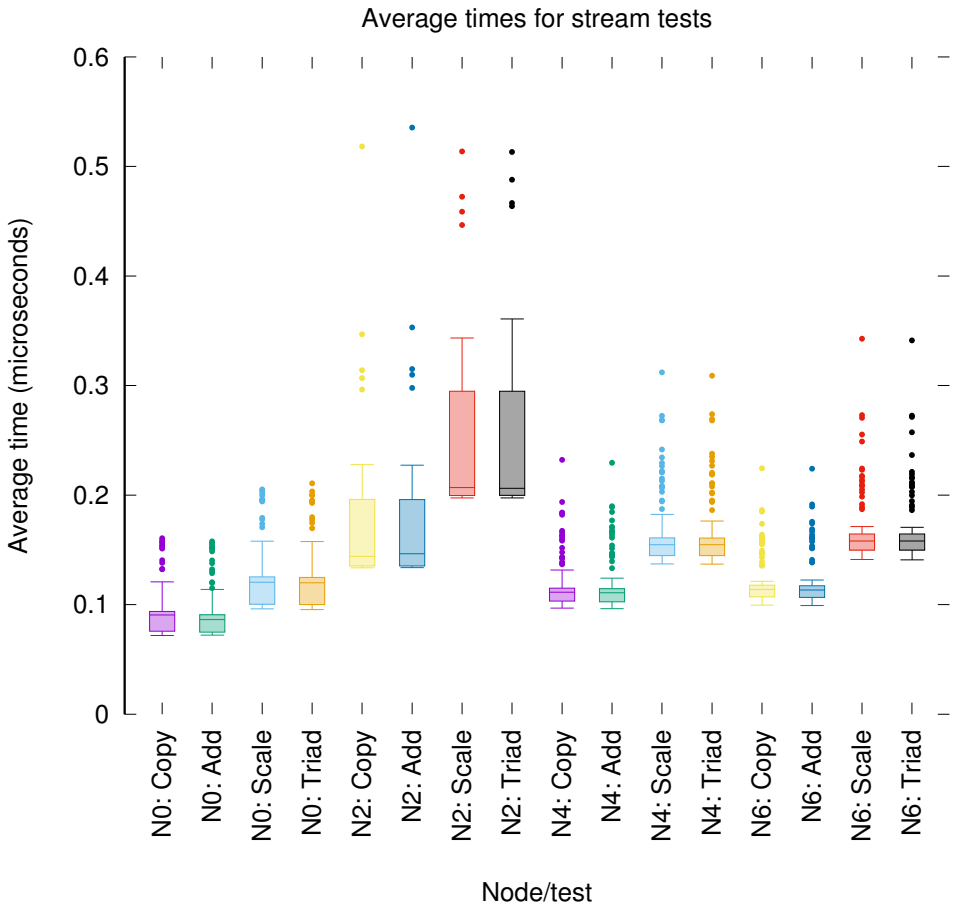

Figure 3: Average test times.

3.3 Discussion

Interestingly, the results do not seem consistent with the matrix of node distances in the hardware configuration reported by `numactl --hardware`. The reported hardware configuration suggests that all external nodes would perform similarly. On the other hand, the architectural diagram of the Magny-Cours (see Figure 4) suggests that one of the nodes (node 3 in the figure) could potentially perform slower due to a higher propagation delay as a result of the physically larger distance. This could be further investigated by running additional tests against different memory nodes to see if a similar pattern appears. From the relatively large amount of outliers seen between Figures 2 and 3, it's likely that clearer results could be achieved during lower/more consistent server load.

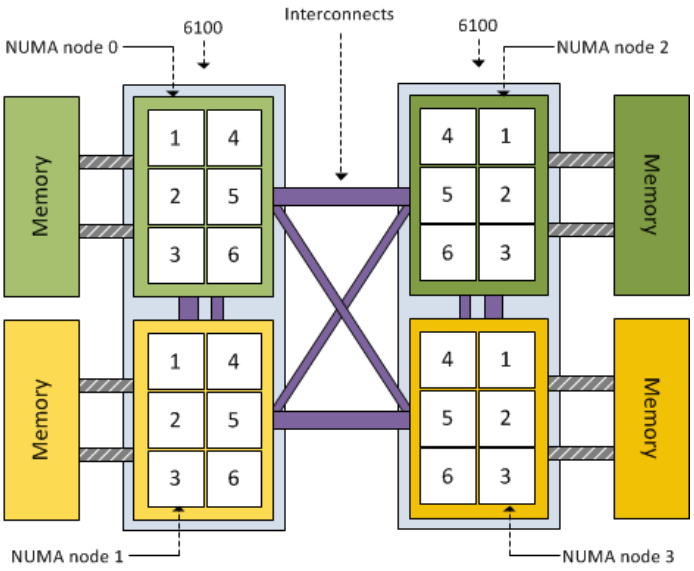

Figure 4: Architectural diagram of Magny-Cours.

4 Appendix

Information on how to run the programs/scripts written can be found in `README.md`.

The following are graphs generated by the programs written for Part A, although they do not correspond directly to the data discussed above:

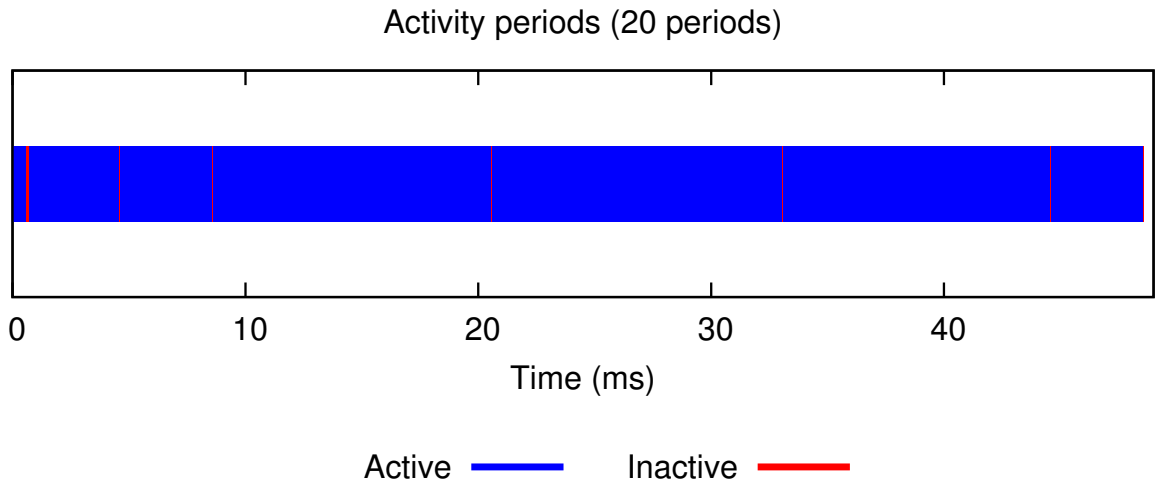

Figure 5: Active/inactive periods.

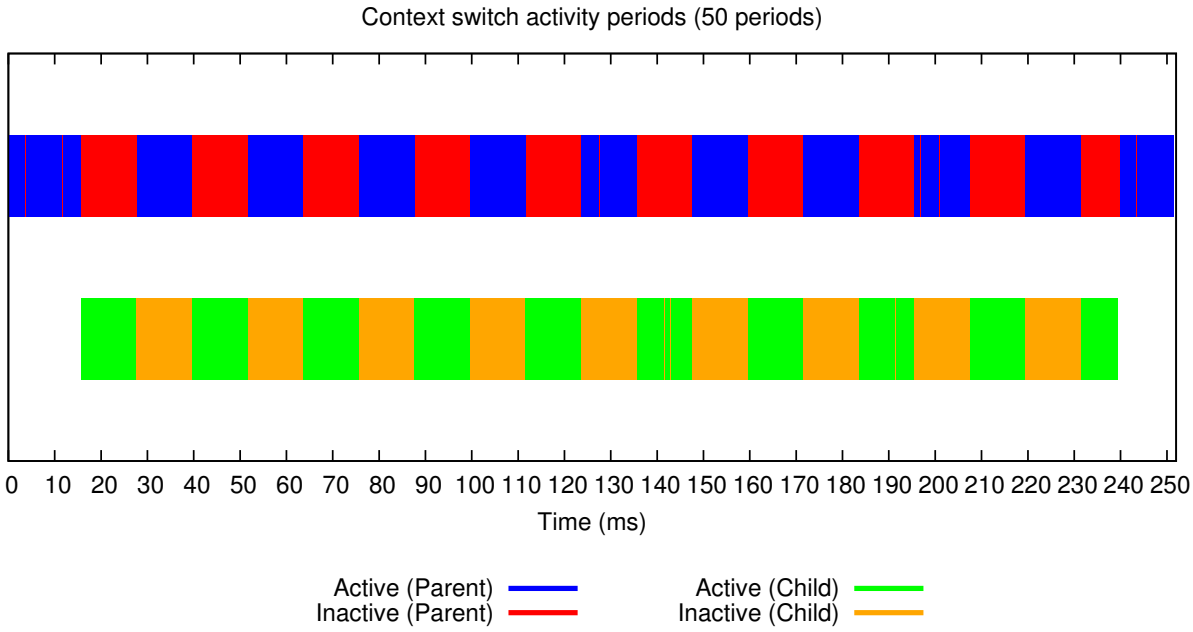

Figure 6: Active/inactive periods for parent/child processes.