Calvin Deutschbein

Faulting Logging Module

Methodology:

The fault logging operation was implemented as a kernel module. When called, function pointers in virtual memory structures we were replaced with pointers to a local functions wrapper that logged data to the kernel log while also calling the pre-existing fault management function. For each fault, the virtual memory address structure's address, the virtual page number, the virtual page offset, the physical page number, and elapsed time for fault management where all recorded.

This module was implemented in log_faults.{c/h} and are provided. The module was loaded using the reset.sh script which calls the included makefile.

Faults were generated using the included caller.c file, compiled with gcc with no options. It adapts material from the provided maprand code fragment as well as existing code from previous module tests. It was run directly from command line with no other user programs running concurrently.

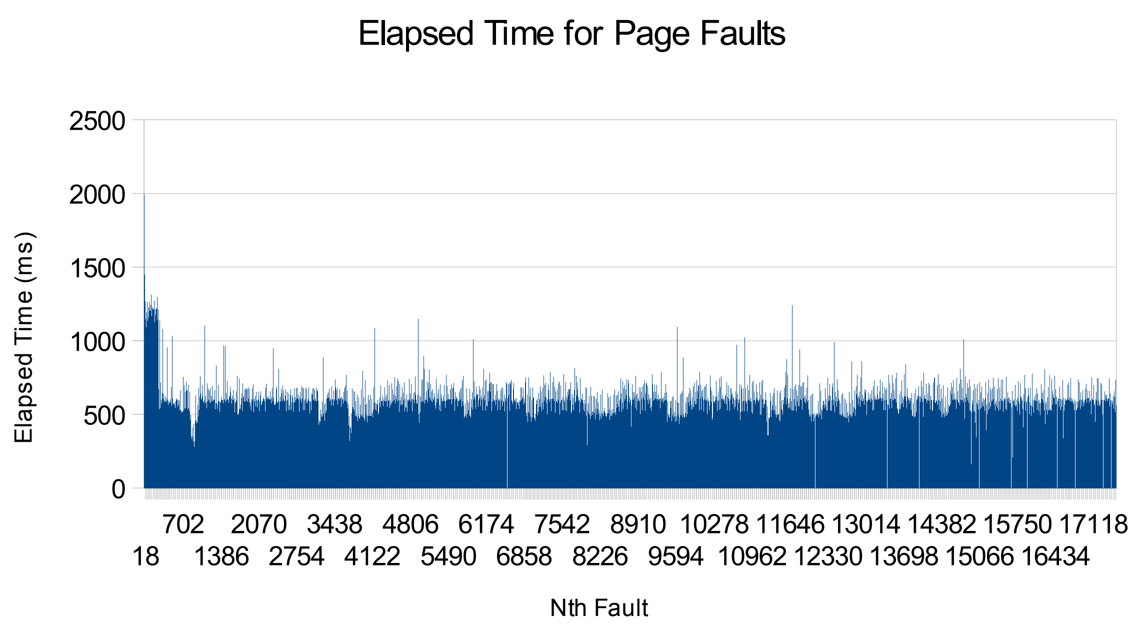
All runs where performed immediately after startup on the virtual machine provided by instructional staff. Faults were logged to /var/log/kern.log using printk with KERN_DEBUG priority. Immediately after the test function finished running, the kern.log file was copied and all analysis was performed on this log file. log_faults.c includes other printk statements that were used to demarcate the beginning of the fault logging from other kernel logging.

The output exclusively associated with the fault logging from caller is included in test.log. It was converted into a faults.csv for use in spreadsheet programs for data visualization.

Data:

The output exclusively associated with the fault logging from caller is included in test.log. It was converted into a faults.csv for use in spreadsheet programs to create visualizations.

The first trend is that the average and the variance in page fault response times decreases over time (do note an outlier is exluded, the 15th fault had a response time of 12666 ms):

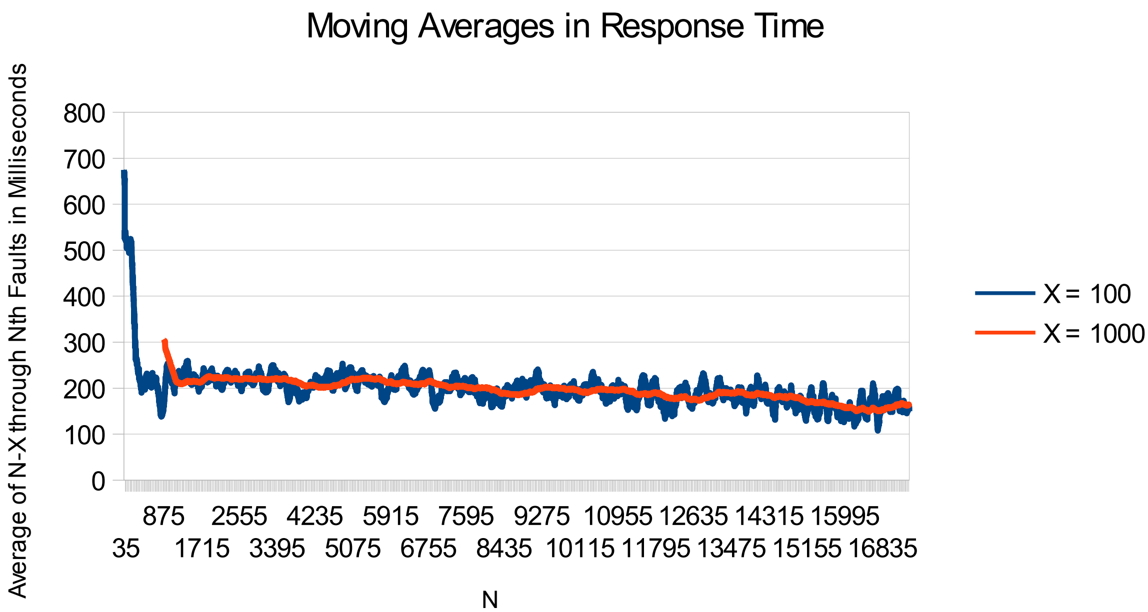


Note also that high elapsed times do not occur on every page fault. In fact, consider the first few rows of the .csv:

f3788960	751061	69	523931	0
f3788e40	750811	11120	0	1996
f3788e40	750811	11120	194114	0
f3788e40	743070	3379	0	1847
f3788e40	743070	3379	193031	1
f3788e40	743479	3788	0	1210
f3788e40	743479	3788	196091	1
f3788e40	740223	532	0	1273
f3788e40	740223	532	195816	1
f3788e40	743079	3388	193660	0
f3788e40	741569	1878	0	1236

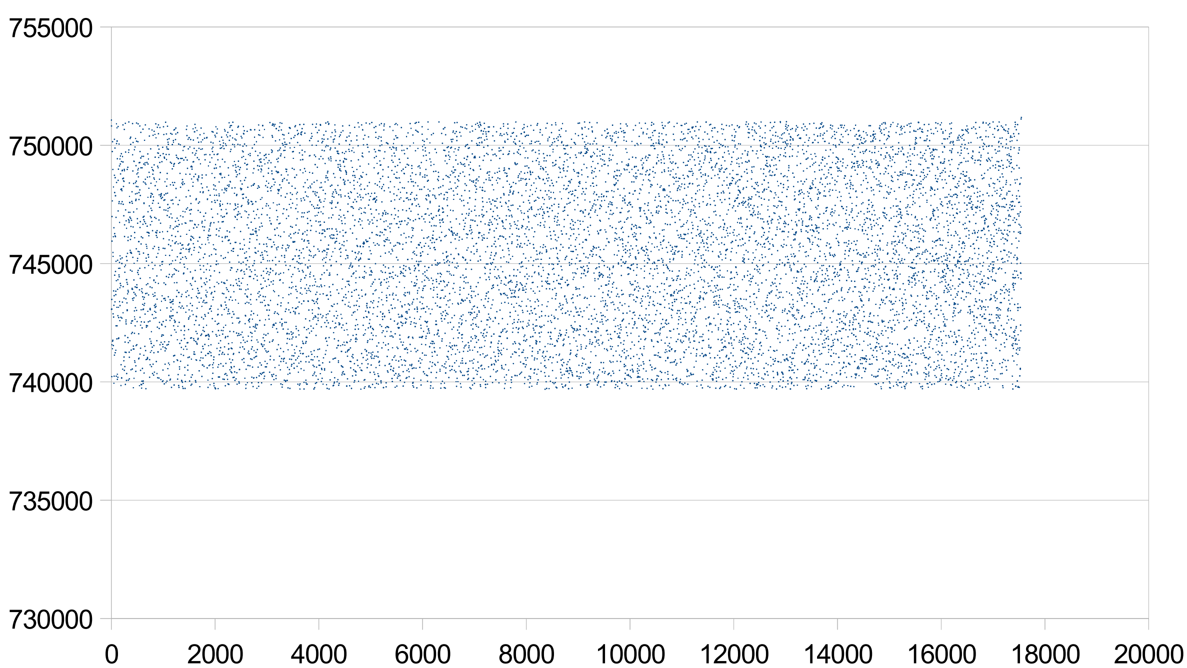
Long response times often occur once for a memory location and then successive accesses are much lower cost. This also shows the relation that early accesses may need to access disk which does not necessarily populate the cache and another fault must be triggered to populate caches from main memory.

Also, this shows that the data captured in the graph is not necessarily representative so that only shows the higher cost operations. To combat this effect, I considered moving averages of cohorts of 100 and 1000 contigous page faults:



This shows more clearly that time spent on faults decreases over time, as well as that time spent is heavily clustered (given the fluctuates in the X = 100 line). Again, refering to the table, these costs are largely incurred the first time some data is pulled from disk.

The randomized nature of accesses can also be verified by considering pages accessed over time:



Conclusion:

The cost of memory access is dramatically reduced on successive calls as data is moved from disk, to main memory, into the cache hierarchy. This can be exploitated to develop optimized programs aware of spatial and temporal localities.

It also highlights difficulties in timing analysis. While the fault response times do have trends, within this broader context they are largely unpredictable. Especially considering the prevalence of cache evictions later on (once the cache has been population) and the difficulty of modeling what will be in the cache and when, predicting the time cost of memory operations must be incredibly difficult.