**Optimal Refresh Value**

To find my optimal refresh value, I used graphical analysis by covering all frame options from 2 to 100. I wrote a shell script that would run my program 99 times (using frame values 2 to 100) and write the raw page fault and disk write values to a text file in a format that I could import directly into excel. I then graphed refresh rate vs page faults and refresh rate vs disk writes on the same graph so I would be able to find a refresh that minimized both values.

Below is a graph of the number of page faults produced for every refresh period from 2 to 100, for 4, 8, 16, 32, and 64 frames on gcc.trace:

The number of faults for 4 frames seems to stay about the same. While 8, 16, and 32 frames have a much higher fault rate at a lower (less than 10) refresh rate. It appears visually that the ideal refresh in terms of page faults is around 25, as we can see below.

Below is a graph of the number of disk writes produced for every refresh period from 2 to 100, for 4, 8, 16, 32, and 64 frames on gcc.trace:

The number of faults for 4 frames seems to level out around a refresh of 25. The optimal refresh for 8 frames is around 10, while 16, 32, and 64 frames levels out around 25 as well. Comparing these results to the page fault data, it seems that a good refresh period for any number of frames is around 25.

For these reasons I have chosen **25** as my **optimal refresh value**.

**Best Algorithm for a Real OS**

In determining the best algorithm for a real OS, I am only examining FIFO and aging, and comparing their results to the opt algorithm. Since the optimum algorithm uses perfect knowledge of the pages that will be referenced in the future, it is impossible to implement in a real OS.

First, I will look at the page fault rates for all three files. Below are the page faults for all three algorithms for gcc.trace:

It is clear from the graph above that the aging algorithm gives desirable performance (less page faults) for 4 and 8 frames. However, both aging and FIFO give similar results for 16 frames, and FIFO gives desirable performance (less page faults) for 32 and 64 frames.

Below are the page faults for all three algorithms for swim.trace:

Much like gcc.trace, the aging algorithm gives desirable performance (less page faults) for 4 and 8 frames, while both FIFO and aging give similar results for 16 frames. FIFO gives slightly better performance for 32 and 64 frames compared to aging.

Below are the page faults for all three algorithms for gzip.trace:

It is clear from the graph that unlike the other two files, gzip.trace has more variability of frames. The results for all numbers of frames for aging are almost identical to those for the optimal algorithm, while FIFO produces less than optimal results as the number of frames decreases.

Below are the disk writes for all three algorithms for gcc.trace:

Again, aging seems to have better performance for most page numbers, and a similar trend can be seen in swim.trace:

For gzip.trace:

The disk writes are closer together as with page faults, but it looks like aging coming out on top.

In conclusion, it appears that the **aging algorithm** is the best for real world applications when compared to the FIFO algorithm, regarding a real OS. It provides the biggest improvement in number of page faults and disk writes. Although FIFO gives less writes and faults for some occasions when the frame numbers are very high, this doesn’t really help because we’re more likely to have memory constraints in a real-world OS.

**Belady’s Anomaly**

For all frame variations from 2 to 100 for each of the three trace files, I found only one instance of Belady’s Anomaly. Do find this easily I wrote a shell scrip to write all the permutations to an excel spreadsheet, then used functions in excel to check if each number of faults was greater than the preceding fault number. There was only one instance of this: **92 frames for gcc.trace.**