Joshua Spisak
CS 1550
Project 3
April 6, 2020

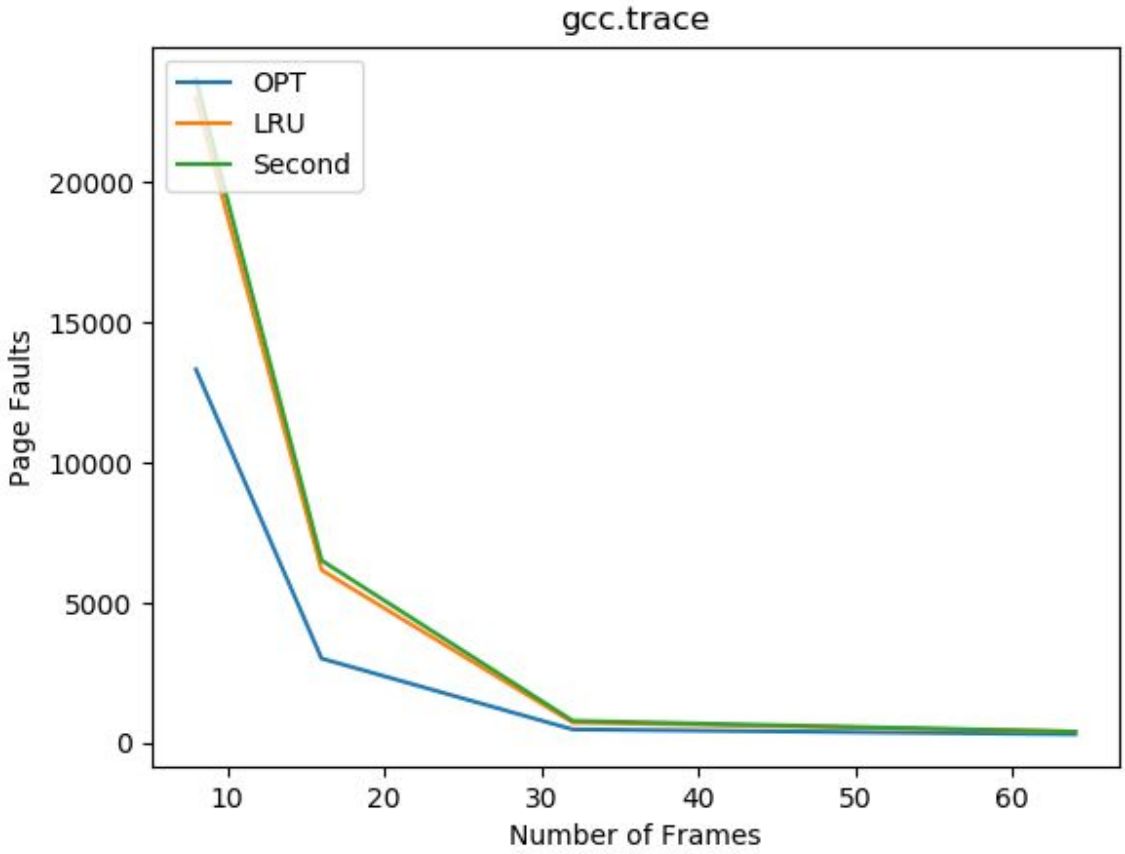
Report

Part 1: Algorithm Comparison

Note: I included tables to make it easier to see when the values are similar

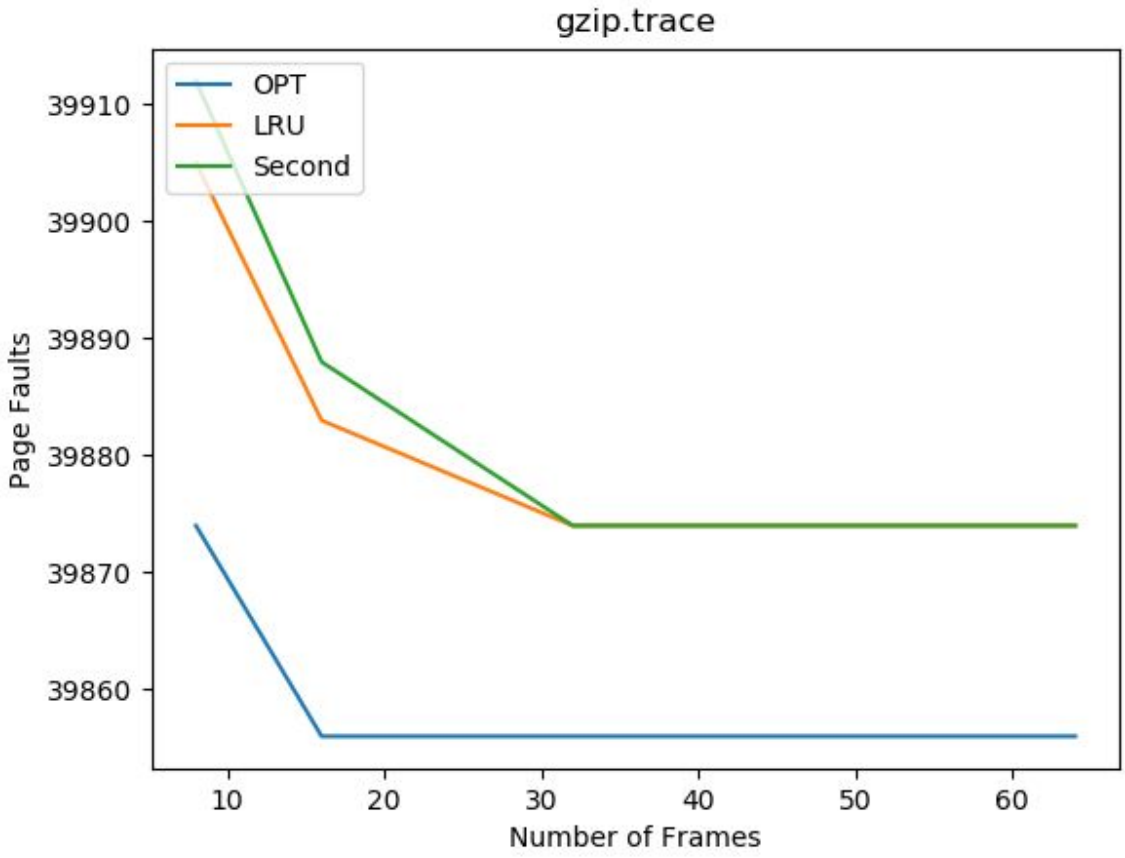
Results for gcc.trace

	8	16	32	64
OPT	13328	3020	491	318
LRU	23027	6172	736	409
Second	23680	6517	797	412



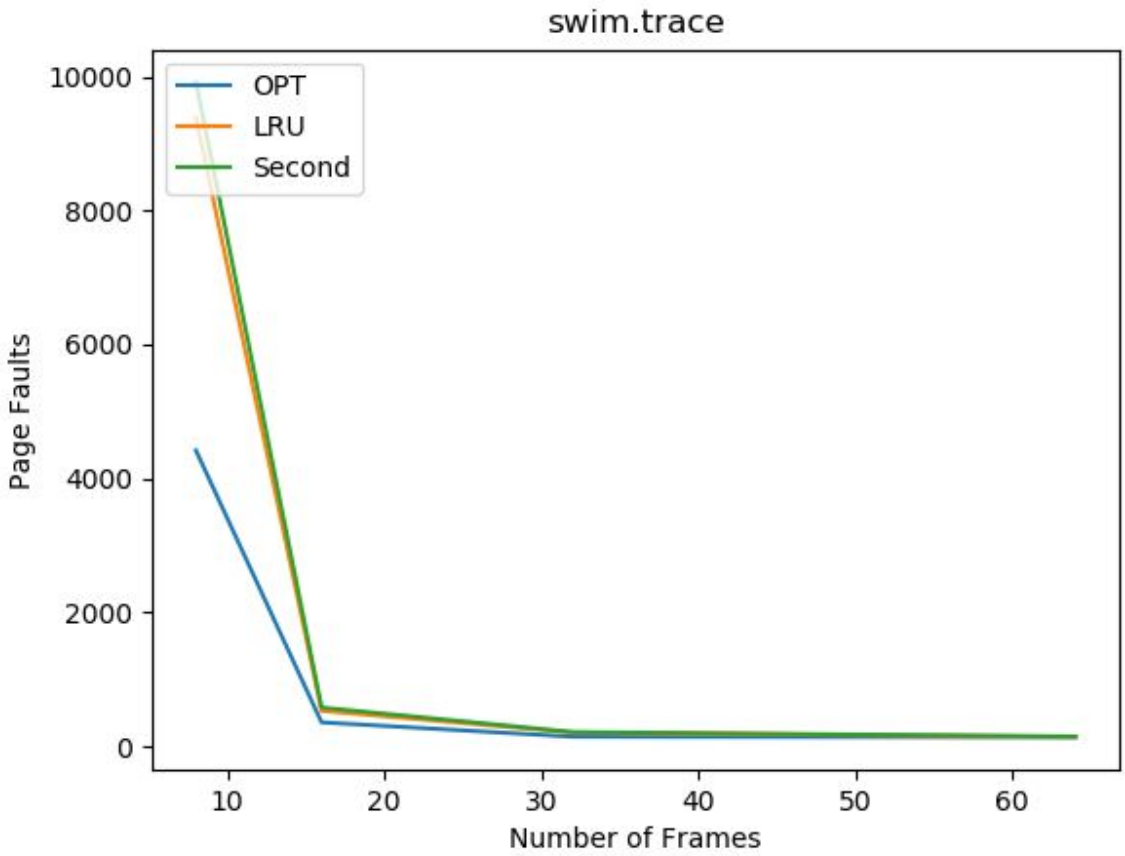
Results for gzip.trace

	8	16	32	64
OPT	39874	39856	39856	39856
LRU	39905	39883	39874	39874
Second	39912	39888	39874	39874



Results for swim.trace

	8	16	32	64
OPT	4417	358	144	135
LRU	9387	530	205	140
Second	9916	579	213	147

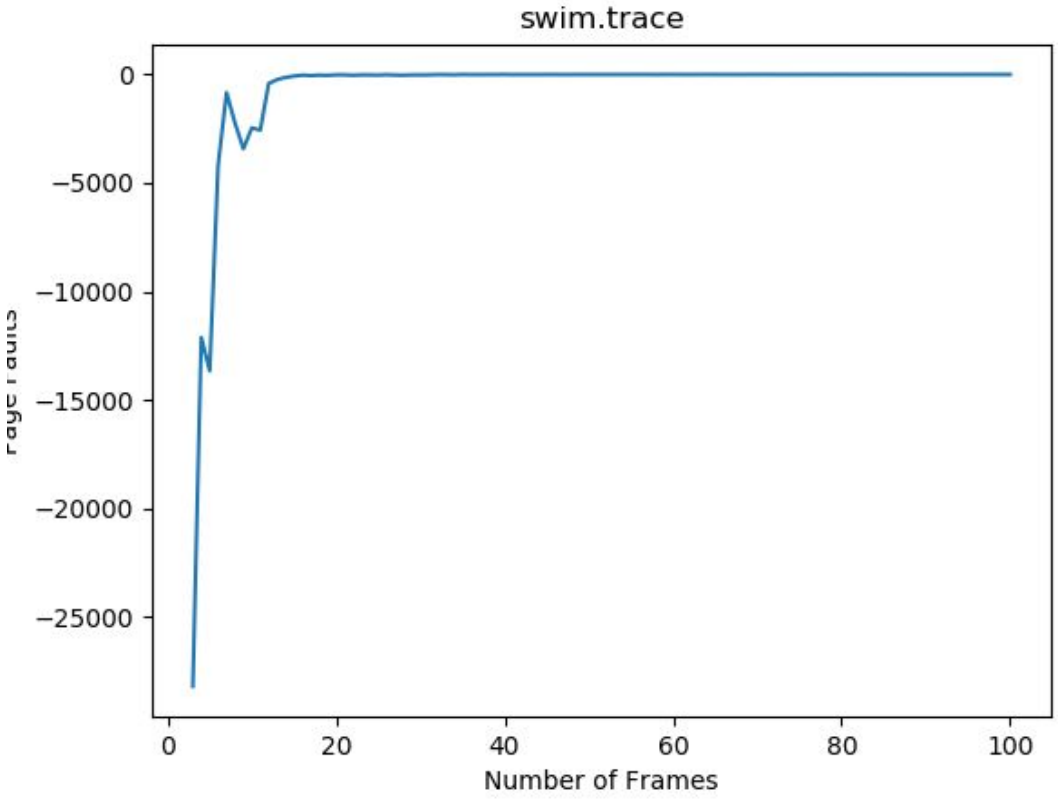


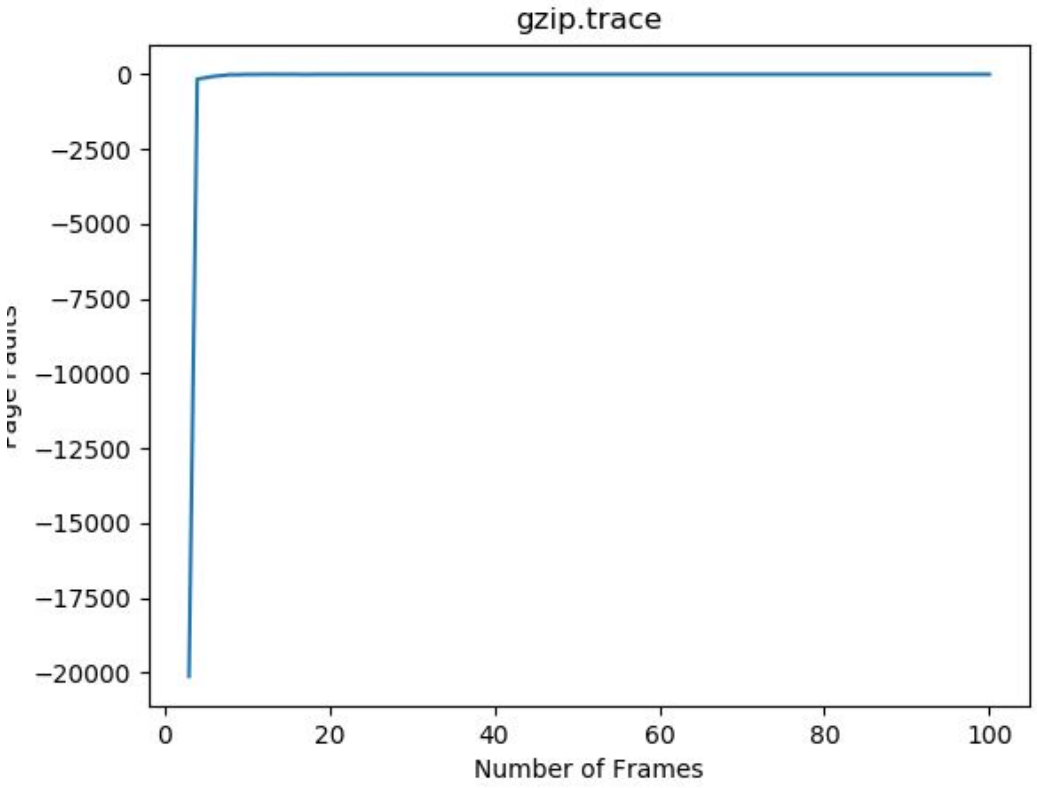
Conclusions: I don't see strong evidence either Second Chance or LRU is exceptionally superior. They both ballparked the same as frames increased. That said LRU was slightly better or tied for all of the tests and isn't really more complex in implementation than Second so that's probably better for OS use. Obviously OPT isn't practical for real OS use, but perhaps something clever that worked with the process scheduler to see which regions would be running soon could do better.

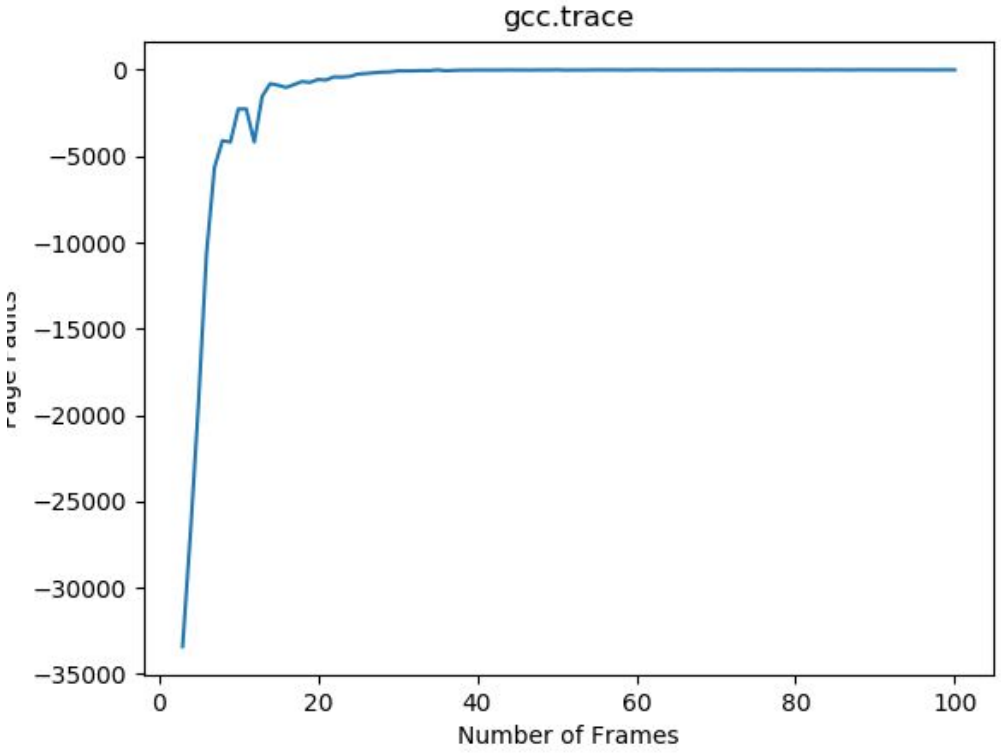
Another random thought (beyond the scope of this assignment) would be to compare how often it needed to write to the disk… avoiding that could probably save some resources and be more efficient in that respect.

Part 2: Second Analysis

For this part I took the derivative of the page faults over frame count and plotted them. I didn't seem to see any instances where the derivative became positive (faults increased with frame count). So it does not seem I saw Belady's anomaly. My guess would be perhaps that would occur at even larger frame counts than 100 or that these cases are just well-behaved.




gzip.trace
Page Faults
0
−2500
−5000
−7500
−10000
−12500
−15000
−17500
−20000
0
20
40
60
80
100
Number of Frames


gcc.trace
Page Faults
0
−5000
−10000
−15000
−20000
−25000
−30000
−35000
0
20
40
60
80
100
Number of Frames

Part 3: OPT Implementation

My implementation is pretty simple, whenever I got to an entry in the trace file I would run down the rest of the trace file to see when it next occured. Then whenever I got a page fault I would look up these values in a map for all of the frames currently in the table to see which one was furthest and evict that frame.

So my big-O runtime is **n^2 + f * m * n** where *n* is the number of entries in the trace file, *f* is the frame count, and *m* is the number of page faults. Since for each value of n I would have to check at most the whole array (all n) of trace entries, then each time there was a page fault I would have to perform a lookup for each of the frames loaded into the table (f of them).. And since I used the STL unordered_map that's an additional n for the lookups. (NOTE: there will be no more than n collisions and you can treat f as a constant so really the asymptotic runtime as a function of entries in the trace file is n^2). On average the performance is much better than this. My memory efficiency is **f + n** because I keep at most a single entry in a vector and a map for each entry of the trace file (n) then add in however many frames there are since I keep an entry for each of those as well.

If I wanted to make it better in terms of runtime efficiency I could use memoization and keep a giant list of the indexes of the frame ids in the trace file to see when they reoccur. I don't really feel like working the math on this out because I didn't implement this, but you would get closer to linear. The memory usage would suck unless you developed some killer heuristics. I almost did this at the getgo but didn't feel like dealing with the complexity.