Arianna Ali & Jhun-Thomas Calahatian

Operating Systems

Spring 2022

Project 2

Simulating Page Replacement Algorithms

Introduction

The main reason for page replacement algorithms is that memory is limited and it is not possible to have access to every single program in your system's memory at the same time. Hence, the program that we wish to run, or access is not always loaded in memory. We must either load the program into memory if there is enough space or replace a program that is already in memory that is not being used.

The first page replacement algorithm implemented in this project is the first-in, first-out algorithm, also known as FIFO. This algorithm behaves quite similarly to a queue, that is a first-in, first-out data structure. Due to these similarities, a double ended queue was used to implement this algorithm.

The second page replacement algorithm implemented is the least recently used algorithm, also known as LRU. This algorithm also behaves similarly to a queue, except that it keeps track of the last time that a particular page has been used. Due to the similarities with the queue data structure, a queue was also used to implement this algorithm.

The third page replacement algorithm implemented is the segmented first-in first-out algorithm, also known as SFIFO or segmented FIFO. Essentially, this algorithm divided the memory between the FIFO algorithm and the LRU algorithm. Since both algorithms were implemented using a double ended queue, two double ended queues were used to implement this algorithm.

Method

For each paging algorithm, a set range of page frames were used, that is 2^0 to 2^{10} page frames. This was done to observe how the algorithms respond to a wide variety of values, and how the hit rate responds to having less or more page frames. However, in the case of the segmented first-in-first-out algorithm, testing was done for percentages 10 to 90, with increments of 10, as well as a range of page frames, that is 2^3 to 2^{11} page frames. This was done to obtain more accurate segments of the primary and secondary caches, as the distribution of the caches using smaller values would not match very closely with the percentage given. Most users have between 8 to 16GB of memory, hence it would not be ideal to test past this threshold.

RESULTS

Table #1 showing the frame rate and respective hit rate for the first-in first-out (FIFO) algorithm in bzip.trace and sixpack.trace

Frame Size	Hits in bzip.trace	Hits in sixpack.trace
1	370263	207621
2	771162	470462
4	871399	648190
8	952172	769832
16	996180	859917
32	997503	914717
64	998533	951699
128	999109	972222
256	999489	984560
512	999683	991991
1024	999683	994508

From the table above, we can hypothesize that bzip.trace has more consecutive entries with the same page number than sixpack.trace since there are significantly more hits occurring in the former when the frame size is one. This would explain why the growth of the hit rate for sixpack.trace is slower until the frame size becomes large enough to hold most of the unique pages. However, the hit rate for bzip.trace plateaus because at 512 and 1024 frames, algorithm reports 317 reads, which means that there are 317 unique page numbers in this trace file, which is likely less than the unique page files in sixpack.trace as the numbers are still slowly increasing.

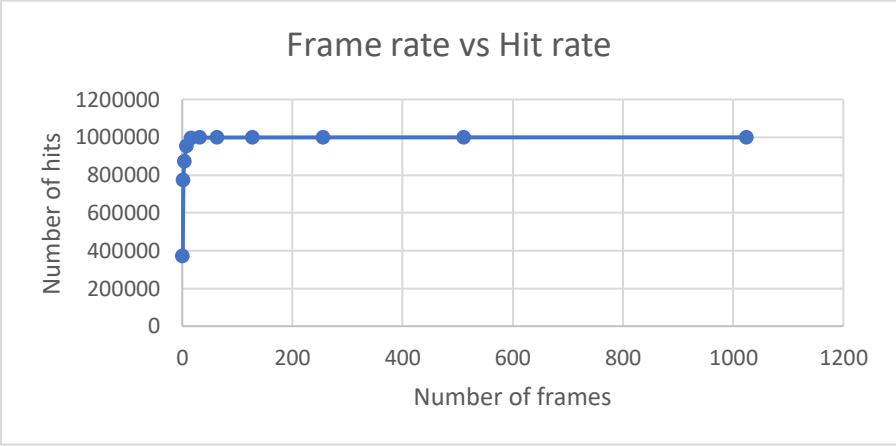

Figure 1 showing the number of frames versus hit rate for the FIFO algorithm for bzip.trace

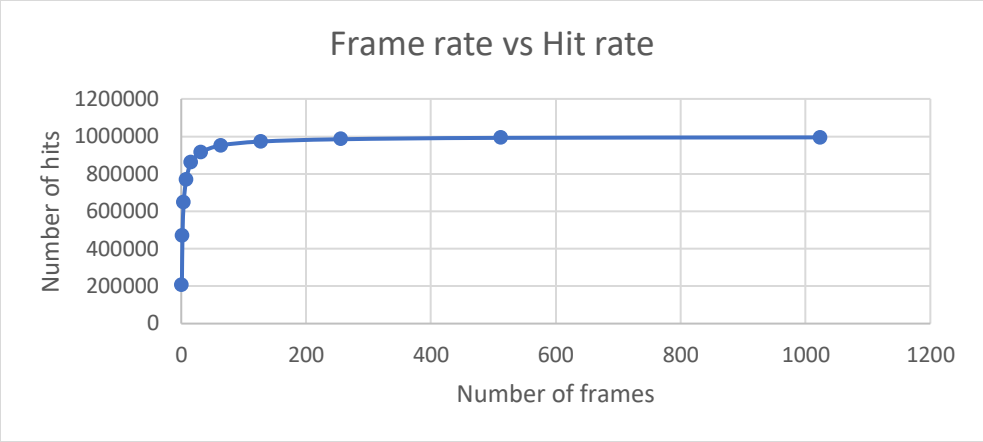

Figure 2 showing the number of frames versus hit rate for the FIFO algorithm for sixpack.trace

Table #2 showing the frame rate and respective hit rate for the least-recently-used (LRU) algorithm in bzip.trace and sixpack.trace

Frame Size	Hits in bzip.trace	Hits in sixpack.trace
1	370264	207622
2	845572	516840
4	907231	717381
8	969310	823505
16	996657	891319
32	997868	932254
64	998737	958815
128	999203	978911
256	999604	988761
512	999684	994178
1024	999684	995533

Similar to the previous table, bzip.trace has a more rapidly growing hit rate than sixpack.trace. Since, the LRU algorithm keeps track of the last time a page was used, it may be safe to assume that identical pages are further apart in sixpack.trace than in bzip.trace. However, it is important to note that there is an improvement between the hit rates from the FIFO algorithm that was previously tested, not to mention that the growth rate of the hit rates is steeper as well. The growth rates plateaus for bzip.trace because it only contain 317 unique pages.

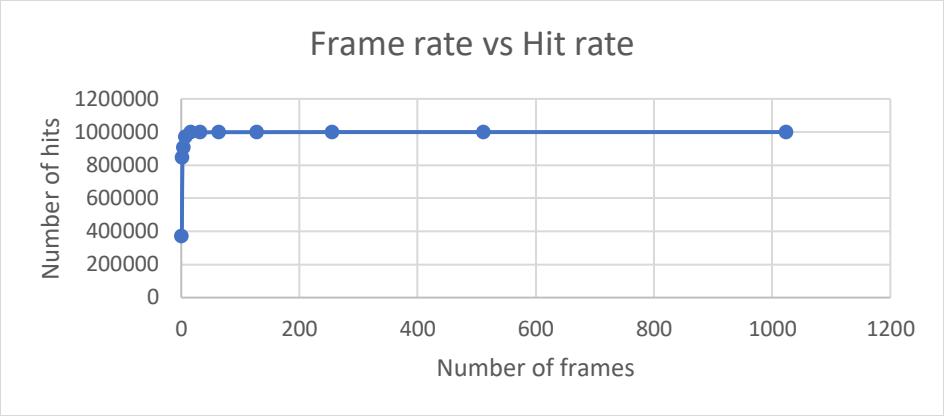

Figure 3 showing the number of frames versus hit rate for the LRU algorithm for bzip.trace

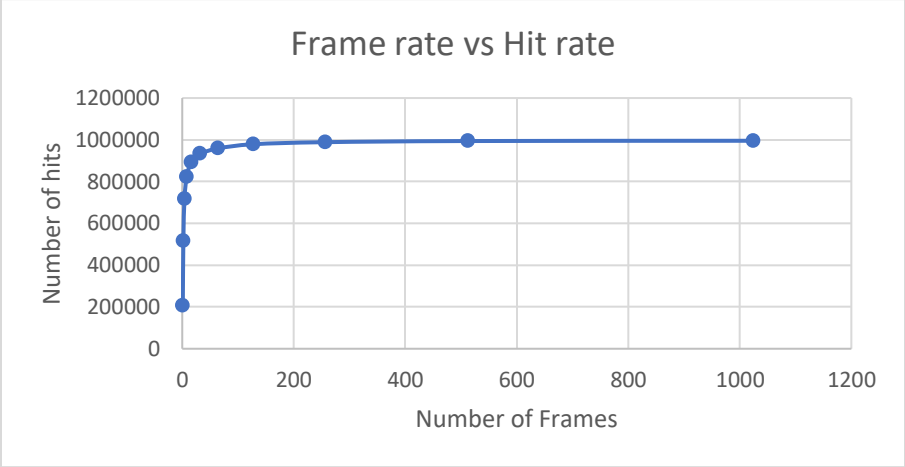

Figure 4 showing the number of frames versus hit rate for the LRU algorithm for sixpack.trace

Table #3 showing the frame rate and respective hit rate for the segmented first-in-first -out (SFIFO) algorithm for bzip.trace at 10%-90%

Number of Frames	10	20	30	40	50	60	70	80	90
16	996473	996557	996570	996577	997759	996631	996651	996652	996658
32	997631	997661	997681	997719	998712	997795	997844	997843	997858
64	998599	998613	998647	998696	999217	998732	998738	998735	998735
128	999155	999187	999220	999227	999596	999227	999225	999230	999229
256	999522	999556	999577	999593	999685	999601	999608	999608	999606
512	999685	999685	999685	999685	999685	999685	999685	999685	999685
1024	999685	999685	999685	999685	999685	999685	999685	999685	999685
2048	999685	999685	999685	999685	999685	999685	999685	999685	999685

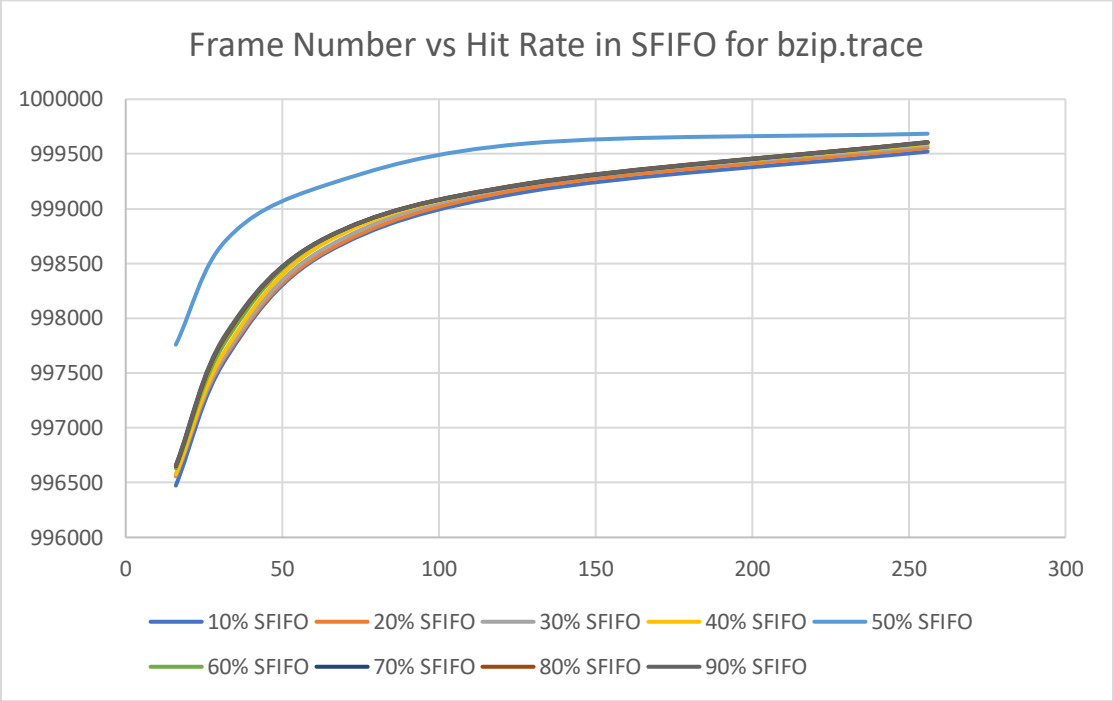

Figure 5 showing the comparison between the hit rates in SFIFO with different percentages in bzip.trace

Since we needed to segment the cache, larger frame values were used for the segmented FIFO tests. As the number of frames surpass 128, the hit rate begins to plateau for all the page replacement algorithms including SFIFO because we are approaching the point where the number of frames is equal to the number of unique page numbers in the trace file. Hence, having many frames can be useful, but if it is too large, some of the memory will remain unused.

Furthermore, SFIFO at 50% is has the fastest growing hit rate. As seen in Figure 5, there is a noticeable difference between the hit rate between SFIFO 50% and the rest of the percentages when using bzip.trace. Although the rest are very close together, we can still see that SFIFO at 10% has the slowest hit rate of all the test cases. This is because the secondary cache is only 10% of the total cache size and the LRU algorithm would not be able to make a noticeable difference at this value.

Table #4 showing the frame rate and respective hit rate for the segmented first-in-first -out (SFIFO) algorithm for sixpack.trace at 10%-90%

Number of Frames	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
16	878863	885064	886802	889341	890137	890820	891186	891283	823506
32	926009	928532	929727	930658	931386	931832	932015	932145	891320
64	956175	957235	958006	958140	958294	958400	958517	958755	932245
128	975269	976299	976886	977319	977786	978163	978571	978804	958797
256	986704	987693	988204	988430	988552	988632	998702	988730	978896
512	993234	993630	993910	993998	994128	994148	994164	994180	988728
1024	995211	995351	995462	995488	995514	995530	995535	995533	994180
2048	995946	995999	996021	996029	995686	996043	996045	996053	995532

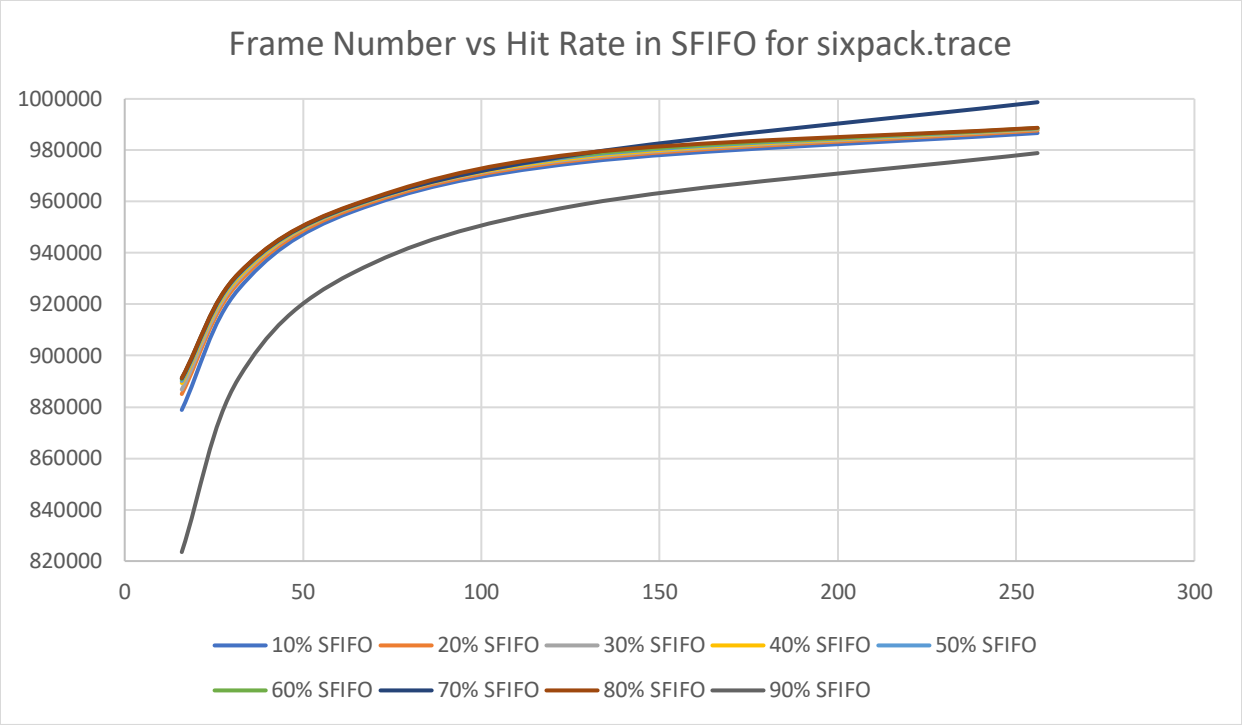

Figure 6 showing the frame number vs hit rate for SFIFO in sixpack.trace with p ranging from 10-90%

Surprisingly, the 90% SFIFO has the slowest hit rate out of all the test cases for sixpack.trace. After that, we can see that the 10% SFIFO has the slowest hit rate after that. The rest have very similar hit rates until reaching a frame number of 256 where the hit rate of 50% SFIFO surpasses the rest of test cases.

Conclusion

Ultimately, memory traces provide an insight into the computer architecture and how the hardware works with the operating system software. It provides an insight into computer architecture because memory traces provide a pointer to the addresses of instructions to be fetched by the CPU. The operating system handles these as the virtualization of memory implies that the memory is not contiguous physically and thus needs to be translated to provide the illusion of many memory frames that can be allocated into. It becomes apparent when utilizing the paging algorithms. These algorithms make it clear how the operating system can take advantage of the hardware provided, such as the cache. In other words, the paging algorithm will look into the cache to see if the instruction has been accessed and is in memory for efficiency. Otherwise, the paging algorithm will load the instruction from the disk and replace another entry in the cache if the cache is full. Furthermore, the operating system software virtualization process is limited by its hardware. As mentioned previously, the cache size places a constraint on the paging algorithms. In other words, the paging algorithms will have lower hit rates if the cache size is too small as there is less “room” for the number of pages that can be loaded. This is apparent during the experiment for frame sizes 32 and under, there were many more page faults, as the paging algorithm had to continuously load and replace pages in the cache. Conversely, if there is an excess of memory - that is the cache size is much greater than what is required to handle the number of pages – the Belady’s Anomaly is observed within the page replacement algorithm. This happens because the paging algorithms do not need to use all the frames in the cache, which would lead to no performance increases after a certain point. Memory is expensive, and we want our systems to utilize the memory given to it. Not to mention that the paging algorithm will still iterate throughout the entire cache, which will lead to slower execution times, as observed in the plateauing within the relationship between the number of frames and the number of hits where the graph portrays a logarithmic curve.