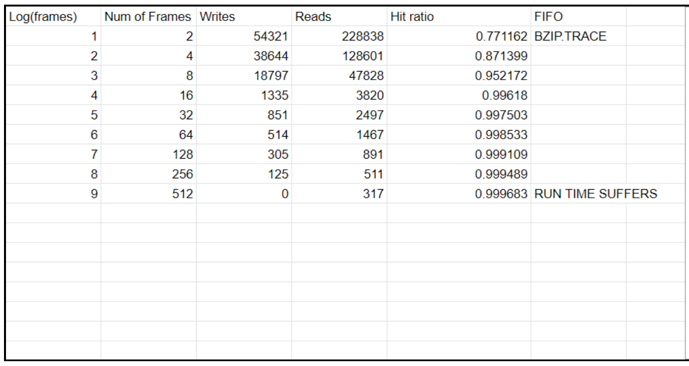
Introduction

Page replacement is a system that decides how to deal with limited sizes and number of pages, and there are a myriad of different algorithms with different use cases; the ones being implemented in this project are FIFO (First In First Out), LRU (Least Recently Used), and a VMS, a Segmented-FIFO algorithm which combines both. Paging is a memory management scheme which uses pages to store and retrieve data as a secondary storage, which have uniform sizes, and are stored in a page table. The different algorithms determine how the operating system will handle what happens when the page table is full or needs to be filled. Our simulator emultates virtual memory addressing, and implements paging. Our "page tables" are implemented using C++ data structures, and we created structs that operate as our page entries. The user, when executing the program, specifies which type of algorithm they would like to run, and uses parameters to specify the number of frames, the type of output they would like to receive, and other variables. We then have different functions in our program that are implementations of FIFO, LRU, and VMS. They have similar foundations, but vary when it comes to how the page table treats replacement. After running the specified page replacement algorithm on a specified input file, the program then outputs the number of disk reads and writes.

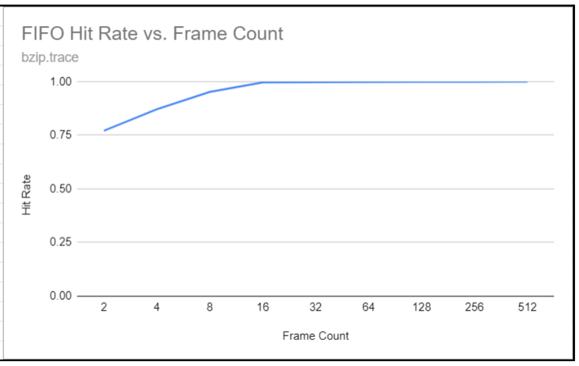
Methods

To accurately observe system performance of the three page replacement algorithms, the experiment was done with multiple runs on varying numbers of frames. This was done in order to reflect how different system's memory allocation can alter how it performs during page replacement and the effectiveness of the algorithms it uses. A sequence of 2*nframes where nframes: [1,10] was used to test the varying results of the aforementioned. Tests were run using these sizes on two memory traces that are snippets of a real trace taken from SPEC benchmarks. This ensures validity on the traces being tested. For VMS in particular, a p value of 25, 50, 75 were selected to simulate the varying ranges of how VMS can be tested. Furthermore, after each algorithm execution the number of reads (page faults), writes(ejections of memory address that needs to be overwritten), and number of memory accesses were recorded and outputted. By calculating the hit rate using 1-(read/trace), it is then compared to the number of frames used to achieve results. By comparing these values as the y and x values respectively, this allows a proper performance metric to then evaluate and draw conclusions from.

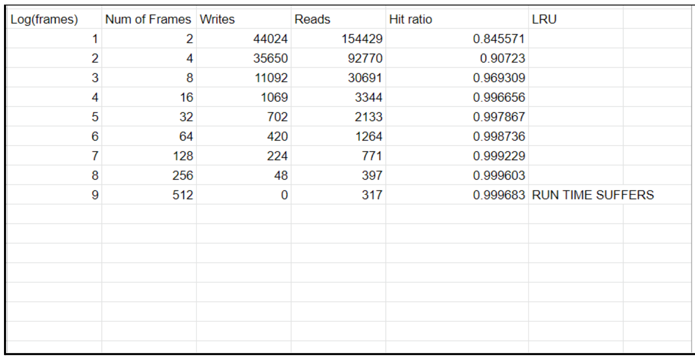
Results



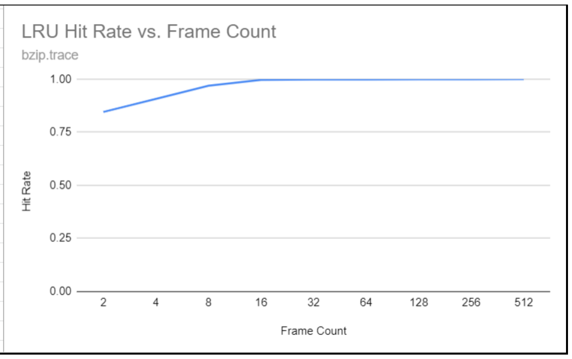
Log(frames)	Num of Frames	Writes	Reads	Hit ratio	FIFO
1	2	54321	228838	0.771162	BZIP.TRACE
2	4	38644	128601	0.871399	
3	8	18797	47828	0.952172	
4	16	1335	3820	0.99618	
5	32	851	2497	0.997503	
6	64	514	1467	0.998533	
7	128	305	891	0.999109	
8	256	125	511	0.999489	
9	512	0	317	0.999683	RUN TIME SUFFERS



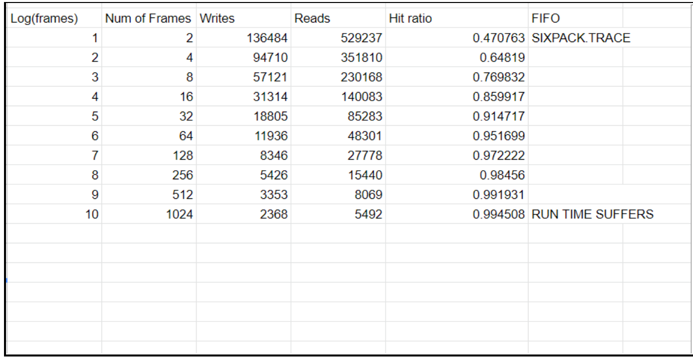
To begin, we'll be analyzing the FIFO algorithm run using various amounts of memory size upon the bzip.trace file. Around 16 frames, diminishing returns starts heavily setting in, and with a hit ratio of 99.62%, each further added memory provides less and less benefit, while coming at the heavily increased cost of runtime. Notably, around 512 frames, runtime begins to increase exponentially, and the number of writes that are performed decreases all the way to 0. This might be a result of the program not being able to properly function when given that much memory to work with.



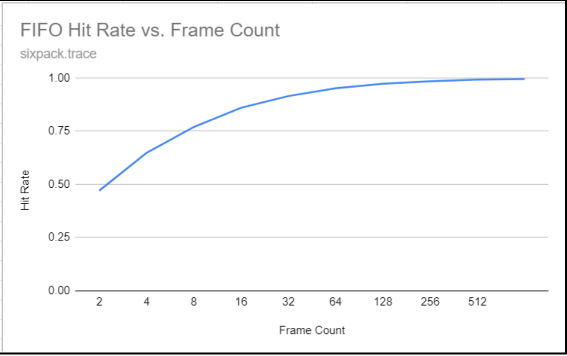
Log(frames)	Num of Frames	Writes	Reads	Hit ratio	LRU
1	2	44024	154429	0.845571	
2	4	35650	92770	0.90723	
3	8	11092	30691	0.969309	
4	16	1069	3344	0.996656	
5	32	702	2133	0.997867	
6	64	420	1264	0.998736	
7	128	224	771	0.999229	
8	256	48	397	0.999603	
9	512	0	317	0.999683	RUN TIME SUFFERS



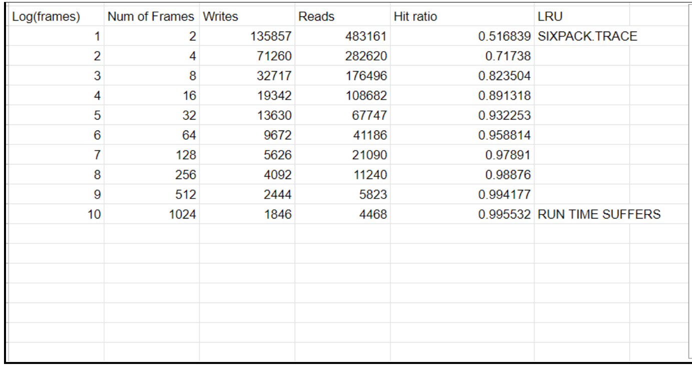
Our implementation of the LRU algorithm, also performed on the bzip.trace file, has similar results to our FIFO algorithm. They both get steady increases until 16 frames, at which there are heavily diminishing returns for each additional frame. It also has a marginally higher hit rate across the board. They also both experience heavy run time detriment at 512 frames, and experience 0 writes at that point. However, the LRU algorithm seems to have fewer numbers of writes overall than the FIFO algorithm, which can be attributed to the fact that LRU tends to "eject" pages less often by keeping page tables with commonly recurring addresses in frame.



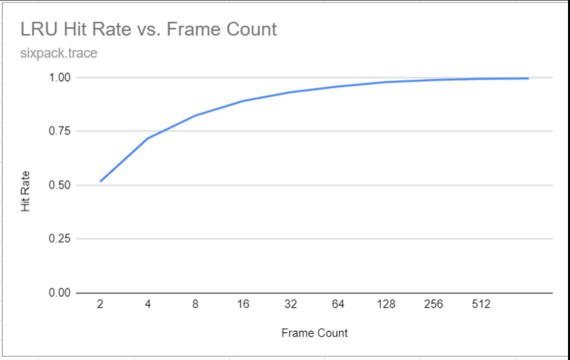
Log(frames)	Num of Frames	Writes	Reads	Hit ratio	FIFO
1	2	136484	529237	0.470763	SIXPACK.TRACE
2	4	94710	351810	0.64819	
3	8	57121	230168	0.769832	
4	16	31314	140083	0.859917	
5	32	18805	85283	0.914717	
6	64	11936	48301	0.951699	
7	128	8346	27778	0.972222	
8	256	5426	15440	0.98456	
9	512	3353	8069	0.991931	
10	1024	2368	5492	0.994508	RUN TIME SUFFERS



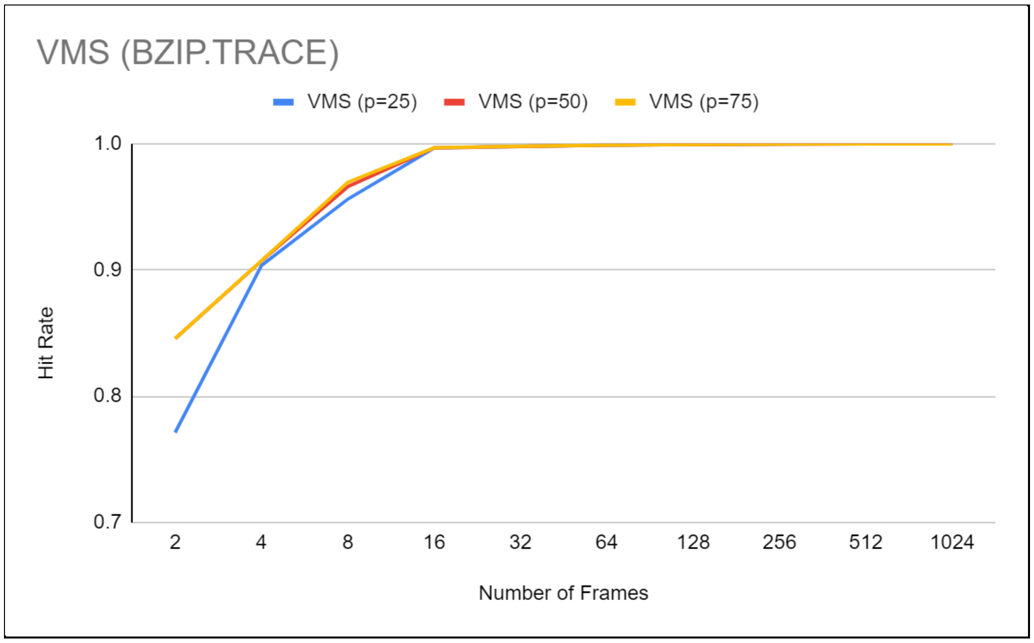
With the sixpack.trace file for the FIFO algorithm, it notably has a much less sharp increase in hit rate proportionally compared to the frame count, and also seems to have lower overall hit rates, starting at 50% whereas bzip.trace usually started at around 80%. For this file, it only starts to have heavily diminishing returns at around 256 frames, much higher than the 16 of the other file. Interestingly, the runtime only begins to suffer at 1024 frames, and at no point do the writes end up being reduced to 0. This could indicate that our implementation of our paging algorithms are better lent to the data contained within the sixpack file.



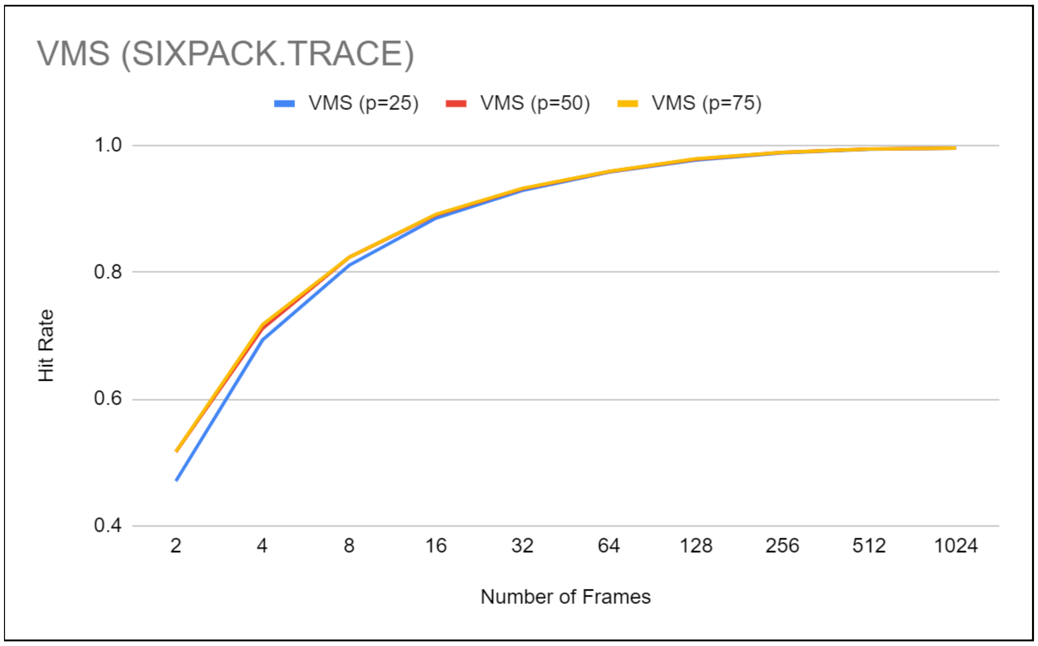
Log(frames)	Num of Frames	Writes	Reads	Hit ratio	LRU
1	2	135857	483161	0.516839	SIXPACK.TRACE
2	4	71260	282620	0.71738	
3	8	32717	176496	0.823504	
4	16	19342	108682	0.891318	
5	32	13630	67747	0.932253	
6	64	9672	41186	0.958814	
7	128	5626	21090	0.97891	
8	256	4092	11240	0.98876	
9	512	2444	5823	0.994177	
10	1024	1846	4468	0.995532	RUN TIME SUFFERS



The LRU algorithm follows the same trends on the sixpack.trace file as it does on the bzip.trace file. It has a similar diminishing returns point and point at which the runtime suffers as the FIFO algorithm, with similarly fewer writes. The only difference is that it seems like the hitrate for LRU is very marginally higher at all frame counts than the FIFO algorithm. This may indicate that LRU is a slightly better algorithm to use with paging for our virtual memory simulation, at least for the two files provided.



When the VMS is performed on the bzip.trace file, the first thing to notice is that it is very similar to the other algorithms when performed on the same file. It experiences a quick rise and then heavy diminishing returns at 16 frames, and starts out around 80% hit rate, when given 2 frames. However, the 25% P version of the VMS seems to lag heavily behind in terms of hit rate, until they all equalize at around 16 frames.



When performed upon the sixpack.trace file, the three breakpoints of percentage LRU for the VMS share a much more uniform trend when it comes to hit rate improvement over number of frames. Facing diminishing returns at 512 frames, this is also an aspect shared with the other algorithms when performed on the sixpack.trace file. The overall hitrates, however, are an amount higher than the FIFO algorithm. This signifies that our implementation of VMS is more effective on the data contained within the bzip.trace file compared to FIFO when at percentages 50-75 but more so mirrors LRU's performance.

Conclusions

Looking at the results, it can be safely stated that the size of available memory (in terms of number of frames given to the page table) affects memory performance. Specifically, by increasing available frames for the page table it increased the hit rate of memory accesses. As seen in the experimental process, memory traces are evidence of the complex management that a computer system has to control in order to do things trivial to the human brain. Even with a trace smaller than the one given, such organization of virtual memory is a highly complicated process for a human. Luckily, paging algorithms exist to solve this solution. The algorithms presented are innovative solutions to the issue of making faster and more compact memory systems. By simulating these algorithms with a sample trace, the experiment yielded distinct yet similar results in all tests.

Firstly, the FIFO algorithm followed the aforementioned trend yet had a different point at which it hit a 1.0 hit rate. It diminished to only 16 frames in bzip.trace but at 256 frames in sixpack.trace. This highlights the occurrence of where diminishing returns begins and how it differs based on the algorithm and the trace file. Secondly, the LRU algorithm also followed the trend and was similar to FIFO. LRU diminished at 16 frames for bzip.trace and 256 frames for sixpack.trace. The difference was how LRU always started with a higher hit rate than FIFO showing that it offers a higher hit rate at low frames but also improves with an increase in frames faster than FIFO. Lastly, VMS followed the same trend. Peaking at 16 frames for diminishing returns for bzip.trace and 256 frames for sixpack.trace like FIFO and LRU, it shows that it follows the trend of increasing performance when given more frames. However, it showed improvement over FIFO with having as little as 50 percent reserved for its secondary buffer to increase its performance over FIFO. When increasing the percentage for VMS it lended to results similar to LRU. This lends to the idea that VMS is a solution between the two algorithms FIFO and LRU and its modular design allows it to be flexible in situations.

These results not only showed that increasing frame size increases hit rate but also that paging algorithms vary in effectiveness from each other and on different memory traces in general. More difficult memory traces result in a lower hit rate and dealing with such is key to designing a page replacement algorithm that can handle such situations. In particular in reference to the 1981 study, VMS truly ended up offering a solution that mimicked LRU's performance without having to deal with the costs of maintaining the usage information of each memory access in the page table, making it comparable to FIFO in terms of resources needed. FIFO cements itself as the easiest solution in terms of implementation with the worst performance and LRU as the best algorithm but with drawbacks. Altogether, simulating these page replacement algorithms yielded results that allowed understanding of not only memory traces but how such algorithms can affect system performance.

Contribution Breakdown:
Nathaniel (8 hours) - Main code, FIFO algo, VMS algo, Report
Anthony (8 hours) - LRU algo, VMS algo, Report