Matthew Luker and Daniel Rocchi
Operating Systems
March 4st, 2019

Project 2: Measuring the Performance of
Page Replacement Algorithms on Real Traces

I. **Introduction**:

The goal of this project was to gain a better understanding of page tables and how they interact with memory. By implementing three different types of page replacement algorithms, we should be able to better understand the decision making process when maximizing memory efficiency. The focus of this project is on the number of reads and writes to the memory for each of the three methods. By evaluating how these values change depending on the number of frames as well as the algorithm chosen, we can effectively measure the performance for each trial. Due to the fact that our focus is on the interactions with the memory, we chose not to fully implement a page table, nor did we implement the reads and writes from the disk. These operations can simply be accounted for through other methods. Implementing those interactions simply would not offer any additional useful information in the context of the performance of a certain page replacement algorithm. By choosing to not include a true page table, we are essentially putting all of the strain onto the page replacement algorithms. This means our tests will be quite strenuous.

When planning out the specific structure and implementation of our simulation, we recognized that implementing the simulated memory as a queue data structure would be the most appropriate. This is because in FIFO, we use a standard queue, in LRU, we use a priority queue, and finally in VMS, we use a combination of two FIFO queues. Keeping these data structures in mind we built the majority of our interaction with a focus on which elements to dequeue and enqueue to the virtual memory at the any given time. In particular, the queues will be storing the page numbers which are being referenced by each reference string from the trace files. Then if we encounter a reference string for which we have already stored the page table, we do not need to do a new lookup. Our memory size would be determined by the number of frames specified

in the command line input. The number of frames indicates the max number of pages that can be held in memory at any time.

II. **Methods:**

One thing that is noteworthy is that the memory addresses we receive from the trace files contain both the page number as well as the offset bits. For our implementation we only need to track which pages are loaded in memory. Due to this, we can simply ignore the offset bits of the requests from the trace. In order to convert the memory addresses in the trace to page numbers, we divide the input by our given page size of 4096. This is very important to recognize, because if we were not provided with the page/frame size, we would be unable to determine how many unique pages we encounter.

We also recognized that the most important thing to understand first was the tracking the number of reads, and tracking the dirty bit of a page that is being stored in memory. By understanding these we would be able to implement the most important part of our simulation. Without tracking this information we would have no way of understanding the performance of a certain trial. First, to track the number of reads, we simply increment a counter every time a new page is added to memory. If the page of current reference already exists in memory, that means we are able to save time by not having to read that particular page into memory. Being that we are not fully implementing a page table, we can expect this number to be quite high due to the fact that only pages currently stored in memory are able to be referenced. In order to track the number of writes, we need to track the dirty bit of a page stored in memory. If at any time a reference string contains a write instruction, that must be tracked in the memory. If the page of the reference string is already present in memory, we make sure to check if the dirty bit needs to be changed. However, if the page being referenced by the reference string is a new page not yet found in memory, we must enqueue that page along with the dirty bit of 1. These dirty bits are then tracked whenever something is dequeued from memory to make space for a new page. If a page is removed from memory with a dirty bit of 1, we know that the information in that page has been modified, therefore we need to make not that a write to the disk is necessary. In the case of a page being dequeued

with a dirty bit of 0, this means that no writes occurred during it's time in the memory, so no modification of the disk is needed.

With a strategy chosen for the tracking of reads and writes, it was then time to determine the behavior for each particular page replacement algorithm. For FIFO, we were able to use a standard queue to track which pages should be removed first. By following a last-in-first-out order of pages, this gave us an algorithm that was extremely easy to implement. However, this also meant that the performance was quite poor due to the fact that the pages used most frequently are not prioritized over any other page. This means that with a limited number of frames to work within the memory, we find ourselves often dequeuing pages that are used often, leading to a higher number of disk reads and writes.

An effective strategy to account for this shortfall present in FIFO is to use the Least-Recently-Used page replacement strategy. To implement this algorithm, we simply had to make a couple modifications to our current FIFO strategy. By converting our queue system into a priority queue, we can reorder elements by their age relative to when they were last used. We are able to enqueue and dequeue elements like normal, however, every time we receive a new reference string, all the ages of elements in the memory are incremented by 1 so. New elements arrive with age 1 for easy tracking. The most important part to note here that makes LRU a better algorithm than FIFO, is that when the page of the reference string is already present in memory, the age of that page entry is reset. This allows that page to stay in the memory longer and to avoid being replaced due to the fact that it was used recently. It is important to note that though LRU outperforms FIFO, it is nearly optimal in fact, it is also much more costly to implement.

In order to avoid a costly implementation like LRU while still improving on the poor performance of FIFO, we can implement a strategy called the VMS second-chance algorithm. In this strategy we are able to utilize our FIFO implementation from the FIFO algorithm, but in a slightly modified way. This algorithm works best when two or more processes are trying to utilize memory at the same time. First we split these processes into two queues, each which represent half of the available memory. These queues are treated in almost exactly the same way that we treated them in the FIFO implementation, however, when pages are dequeued we move them into a separate queue of either Clean or Dirty. If we have space in memory, we keep these dequeued pages in the the Clean/Dirty queue in hopes that they will be called upon by the reference string in the

near future. This is why we call it the second chance algorithm because we do not immediately discard those pages which are dequeued from the FIFO queue of a particular process. Another important thing to note about VMS is that it expects FIFOA and FIFOB to both be filled with NULL values initially. In our implementation, we simply filled the queues with page numbers of -1 (ffffffff in hex) because this should not be a value we ever encounter in the trace.

III. **Results**:

We chose to run our simulations using gcc.trace because the results across all three algorithms were the closest match to the expected behavior. For FIFO, we found that bzip was the main outlier in terms of performance. The same number of frames resulted in much better performance on bzip, meaning that this particular trace is simply better suited for a FIFO implementation. Looking at LRU, we found that all the traces performed fairly similarly, with swim and bzip outperforming sixpack and gcc by a decent margin. For this reason we chose to test LRU using gcc in order to be more consistent when comparing to FIFO. Finally, for VMS gcc.trace is the only file which is applicable. This works out nicely because gcc seems to be most appropriate for testing across the board. This means when we compare results between the algorithms, we are comparing the results from the same trace.

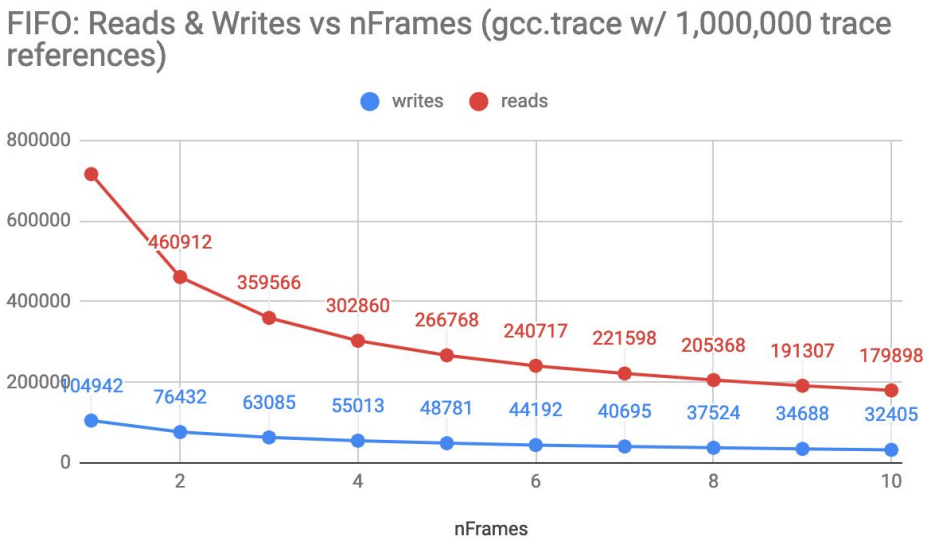
We found that LRU compiled much much slower than FIFO and VMS even though it returned more efficient results. Once we started testing frame numbers of over 128, compilation time took at least 1 minute. Our test for lru with a frame size of 256 took over 5 minutes. This is interesting to see because LRU expected to be inefficient in terms of hardware strain and compilation time, however the end result has a much better performance in terms of hit rate.

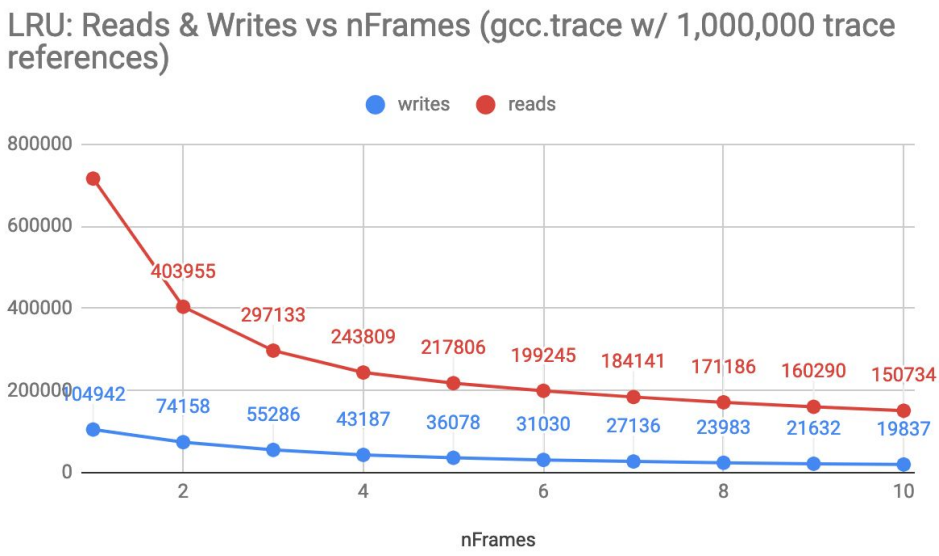
As expected, FIFO performed worse than LRU due to the FIFO algorithm only holding addresses in the order they were entered and not on how frequently each address was interacted with. LRU holding addresses for longer amounts of time based on how often they are changed increases the likelihood of not having to read or write the address for a second time in a nearby duplicate address. In the end, our results for VMS had more reads and writes than both LRU and FIFO. This must have been due to small errors with our code, most likely within our queue implementation, as we struggled at

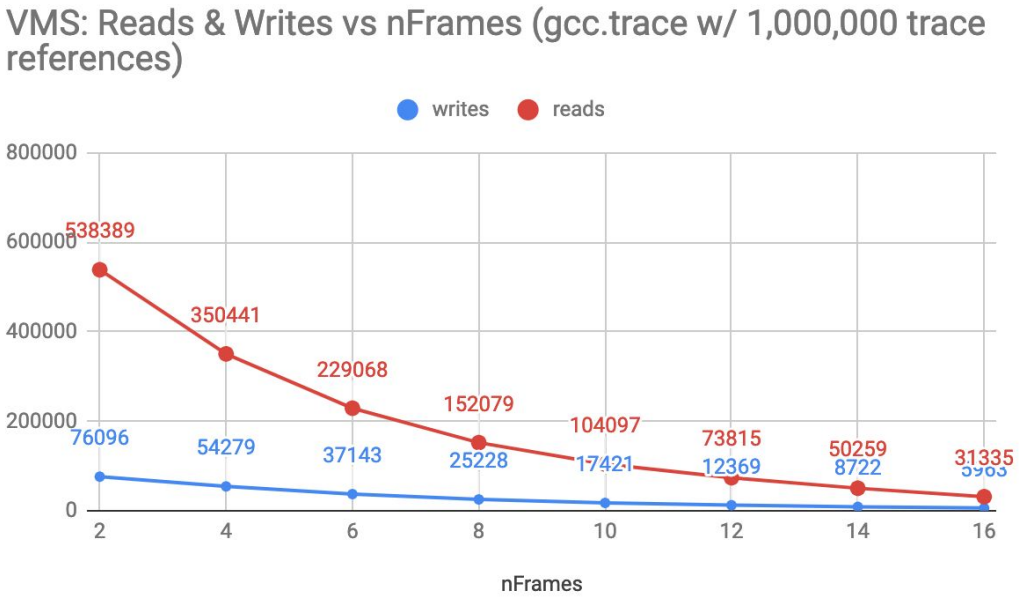
the start of the project to properly implement the queue. VMS is supposed to have an amount of reads and writes closer to LRU when given a higher number of frames and a count of reads and writes closer to FIFO when given a lower number of frames. VMS should have an amount of reads and writes in a range between LRU and FIFO since VMS only operates differently from FIFO until its spots in FIFOA and FIFOB fill completely. Once the two smaller FIFO queues fill inside of VMS then it holds no more recently visited spots in its Clean or Dirty lists. It is at this point when the two queues in VMS are completely full and its Clean and Dirty queues are empty that it gets the same efficiency as a normal FIFO queue. This is why giving VMS more frames of memory to work with gives it better efficiency, since it prolongs the amount of time until it becomes full and therefore the same as FIFO.

Plots of hit rate(reads) and writes vs. cache size for each algorithm

When testing the outputs for each algorithm, we first used a range of 1 to 10 frames for FIFO and LRU, as well as a range of even values between 2 to 16 for VMS. These results seemed to reflect what we expected, with the exception of VMS that we described above. However, this test set represents an unrealistically small amount of memory for a real world application. The following charts depict these observations.

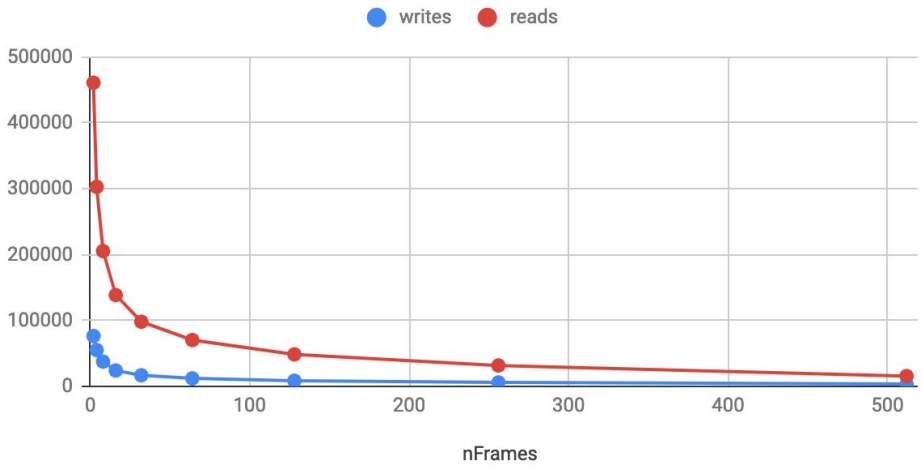




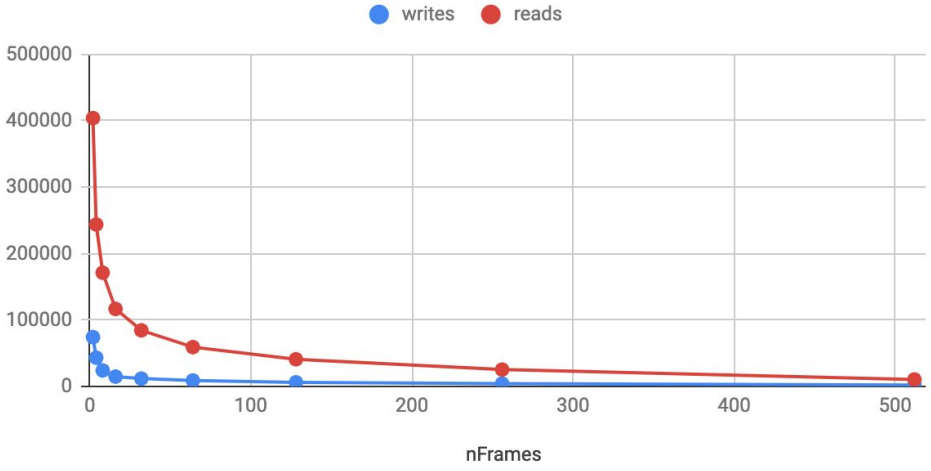


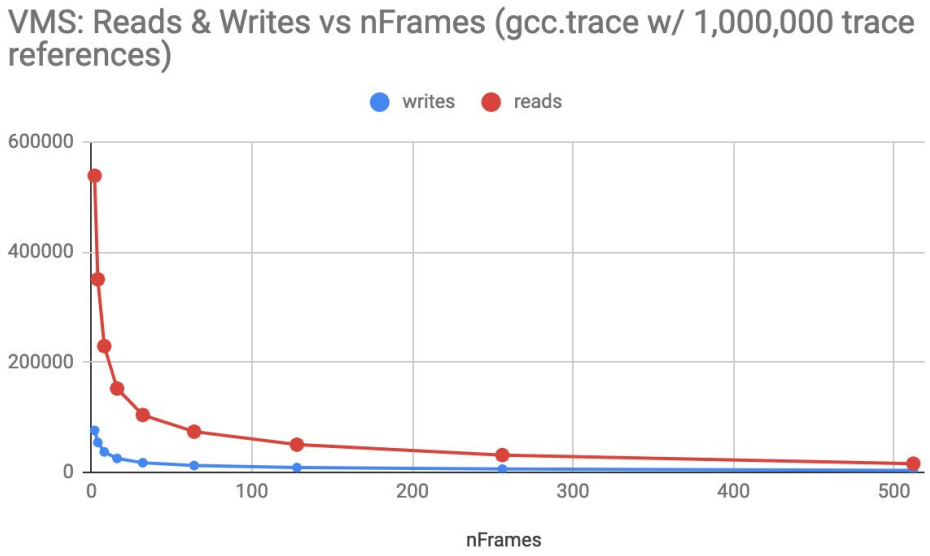
To test our algorithms on more realistic memory sizes, we modified our test ranges to be from the range of 2^1 to 2 ^9 for all algorithms. This gave us a much more interesting set of data. We can see in the following plots that there is an area centered around 2^4 number of frames in which the dataset experiences the most variation.

FIFO: Reads & Writes vs nFrames (gcc.trace w/ 1,000,000 trace references)

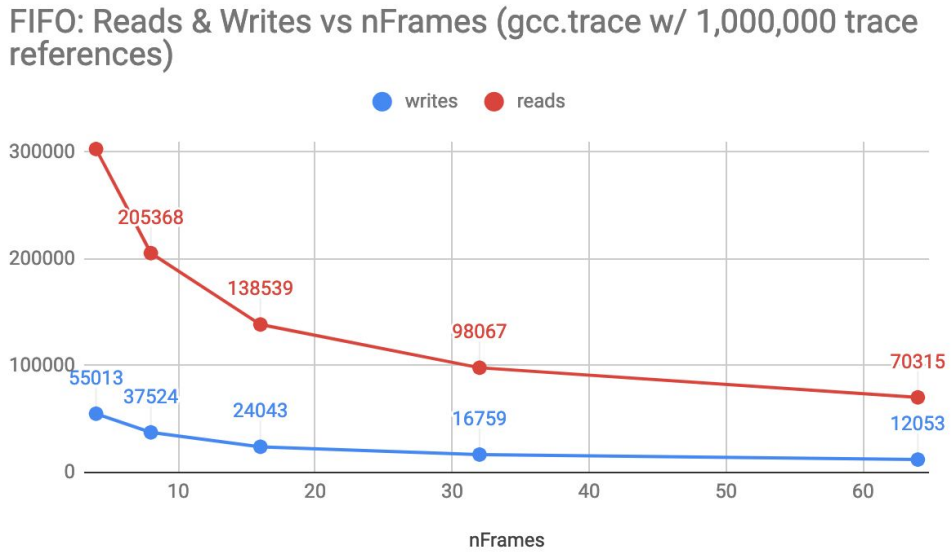


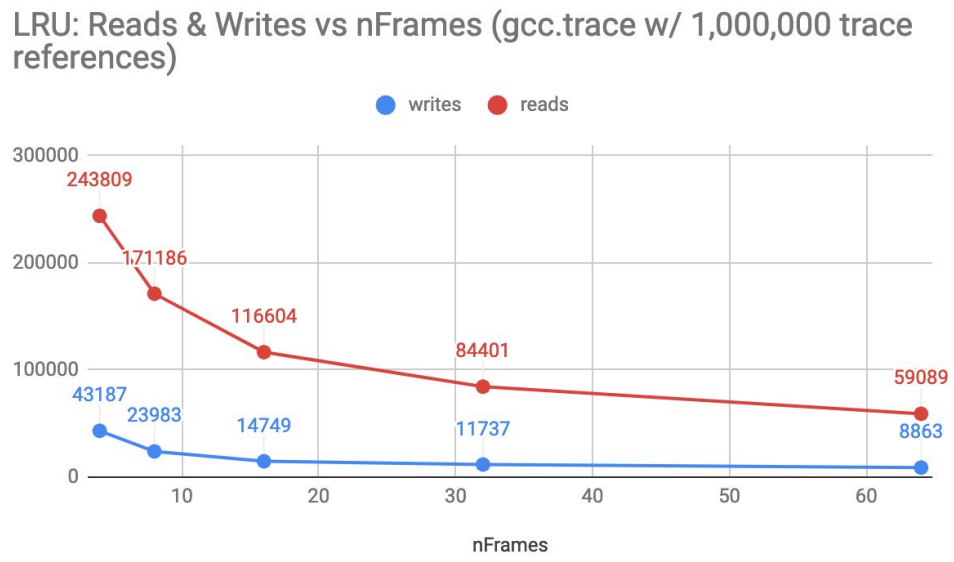
LRU: Reads & Writes vs nFrames (gcc.trace w/ 1,000,000 trace references)

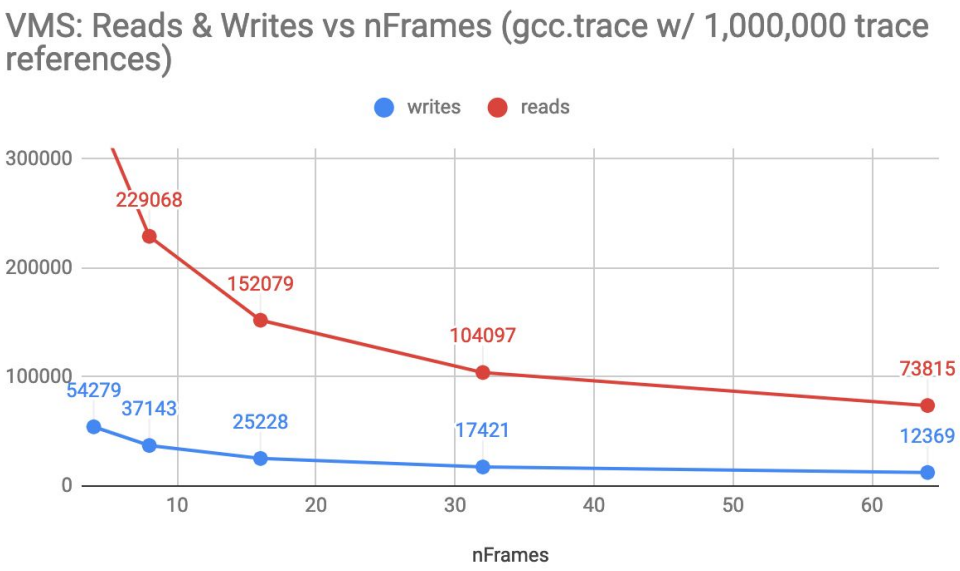




In order to take a closer look at this subsection of data, we restricted the range of the data to be from a range of 2^2 to 2^6. Looking at this subset we can more easily see the curvature and the difference in data points across varying frame sizes.







Here we can clearly see that FIFO is outperformed by LRU by a fair margin. However as stated above, we unfortunately see that VMS does not perform as expected, as its performance was expected to be in between the ranges of FIFO and LRU.

IV. **Conclusion**:

During our time working on this project, we found that the most important part of reaching a successful simulation was proper planning. Initially we started working immediately on the code without a true plan in place, this meant that we ran into a lot of issues early on relating to the data structures we should be using to track the memory for each algorithm. We took a step back and put together a more solid plan. Taking a look at each algorithm, we created examples which we worked out on a whiteboard. We developed cases for each scenario in the algorithm and made note of the expected behavior. By translating this expected behavior into an algorithm, we had a much better understanding of how our queue data structure would be utilized. This was especially important when working with the VMS algorithm. By making note of each possible path of execution, we were able to clearly understand how FIFOA, FIFOB, Clean, Dirty, and Memory all interacted with one another. By combining this new knowledge with the pseudocode provided in the project description, we were able to fairly quickly implement a fully working VMS simulation. We found that most issues during this project were a result of our poor early planning. There were a couple times when we had to restructure our queue implementation to be able to properly support the behavior we expected from our algorithms. If we had a better plan at the start, these issues would have been mitigated to a large degree.

It was very interesting to see how our algorithms performed on memory traces from real world program executions. It is extremely interesting to us that you can track exactly how the memory was modified from a program. We could certainly see how this could be helpful for situations where you already know how a program will affect memory, and you want to find out what page replacement strategies will be most effective to implement.

Taking a look at our results for each page replacement algorithm, and comparing them, we actually found that they performed fairly close to the expected behavior. To summarize, LRU outperformed FIFO by a large degree, and the performance of VMS fell somewhere in between the two. As discussed previously, these results make sense in terms of both the expected behavior, as well as the implementation for each. Because LRU is so costly to implement, we would expect the return for this increased cost would be increased performance. FIFO, being the easiest to implement, performs decently as it

was expected to. Finally looking at VMS, because it is a slightly modified FIFO with only a slight increase in cost, we see that it returns a slight increase in performance when comparing to the simple FIFO.

Due to the fact that we were provided with such a large trace file, this meant that the larger our memory was, the better the performance was. With a large memory size, the main trade off is that when initially filling the memory, each page stored results in a page fault. However, once we get through the initial page faults, we save lots of time on any future page lookups. This is true across the board for all three algorithms, however especially in the case of LRU, this creates an extremely effective page replacement algorithm. In LRU, when we have a lot of memory space, the pages used most often are almost always present in memory. This leads to a very low number of both reads and writes. In the case of FIFO and VMS, we still see a vast improvement, but the rate of improvement is not quite as good as in LRU.