CSE 34341: Operating Systems
Programming Project 5: Virtual Memory

Hoai Trinh

I. Abstract

The project aims to demonstrate an understanding of virtual memory concepts and develop proficiency in implementing faults handlers. By constructing a user-level, demand-paged virtual memory system, I am able to investigate the performance of different page replacement algorithms without the complexity of kernel-level code.

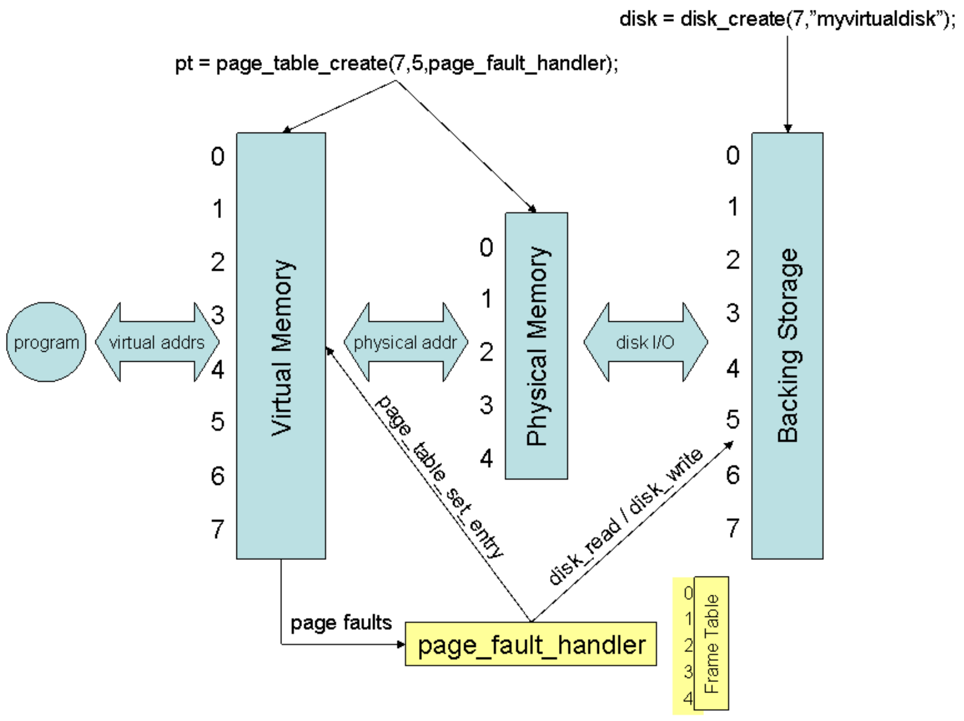

Fig.1. Virtual Memory System

In this project, I utilize a provided "virtual" page table and a "virtual" disk to establish a small virtual and physical memory space. The main objective is to implement a page fault handler that efficiently manages page faults by trapping them, updating the page table, and handling data transfers between the disk and physical memory.

Once the virtual memory system is functioning correctly, I will evaluate the performance of several page replacement algorithms, including Least Recently Used (LRU), First-In-First-Out

(FIFO), and a custom-developed algorithm. The evaluation process will involve running a selection of simple programs across various memory sizes to understand the behavior and efficiency of each algorithm. This analysis will help draw conclusions about the effectiveness of the different algorithms under different conditions, ultimately leading to a better understanding of virtual memory management.

II. Purpose of the Experiments

The primary objective of these experiments is to gain a thorough understanding of virtual memory management in the context of operating systems. By implementing a user-level, demand-paged virtual memory system and evaluating different page replacement algorithms, I aim to:

- Demonstrate mastery of the virtual memory concept by simulating the process of mapping virtual addresses to physical memory locations.
- Develop skills in handling page faults, a critical component of virtual memory management, by implementing custom fault handlers.
- Investigate the performance of various page replacement algorithms, such as Random, First-In, First-Out (FIFO), and a custom algorithm, to identify their strengths and weaknesses under different conditions.
- Enhance our ability to perform quantitative system evaluations by analyzing the number of page faults, disk reads, and disk writes generated by each algorithm and program.

III. Experimental Setup

To conduct these experiments, I use a provided "virtual" page table and a "virtual" disk to create a small virtual and physical memory space. The experimental setup consists of the following components:

- Student machine: student5.cse.nd.edu
- Command line arguments:
    - ./virtmem <npages> <nframes> <rand | fifo | custom> <alpha | beta | gamma | delta>
- Page replacement algorithms: I implement and test three replacement algorithms - random, FIFO, and a custom algorithm. The custom algorithm is described in detail in the lab report.
- Programs: A selection of simple programs (alpha, beta, gamma, delta) is used to evaluate the performance of page replacement algorithms across varying memory sizes
- Metrics: The performance of each algorithm and program is assessed by measuring the number of page faults, disk reads, and disk writes. These metrics are graphed for a visual comparison of algorithm performance.

IV. Page Replacement Algorithms

1. Random Page Replacement

The random page replacement algorithm randomly selects a page (frame) from the page table and replaces it. This policy is easy to implement and has low overhead, as it does not require any additional data structures or counters to maintain usage information. However, this is not a very sophisticated algorithm since the performance of the random algorithm can be highly unpredictable, as it entirely depends on the random frame selection.

2. First-In-First-Out (FIFO) Page Replacement

This is the most straightforward algorithm, which assumes that the oldest page is likely to be the least recently used. When a page fault occurs and there is no free frame available in the physical memory, the FIFO algorithm selects the oldest page (i.e., the one that was loaded first) for replacement. The main idea behind this algorithm is to remove the least recently added page to make room for the new page.

In the program, the FIFO algorithm is implemented using a linked list. The “fifo_head” and “fifo_tail” pointers are used to manage the list. When a new page is added to the physical memory, a node representing the frame is added to the tail of the list. When a page needs to be replaced, the node at the head of the list is removed, and the corresponding frame in the physical memory is replaced with the new page.

3. Custom Page Replacement

The custom algorithm implemented is a variation of the Least Recently Used (LRU) page replacement algorithm. When a page fault occurs and there is no free frame available, the page that has been used least recently is replaced.

In the program, the LRU-based custom algorithm is implemented using a linked list (“lru_head” and “lru_tail”) is used to keep track of the usage of frames. Each node in the list represents a page frame and contains an LRU counter. Whenever the page frame in physical memory is accessed, the LRU counter of that frame is updated to 1 (i.e., the most recently used), while the LRU counters of all other frames in the list are incremented by 1. When a page fault occurs and there is no free frame available, the program will iterate through the list to find the page frame with the highest LRU counter value (i.e., the least recently used frame). The page associated with the least recently used frame is then replaced, and the LRU counters for other frames are updated accordingly.

This custom algorithm attempts to balance the time complexity of accessing the least recently used page frame and the memory overhead of maintaining the LRU counters. In general, it is a simple and efficient approach for managing page replacement in virtual memory systems.

V. Results

I measure and graph the number of page faults, disk reads, and disk writes for each program and each page replacement algorithm using 100 pages for the application versus the number of page frames (NPF) for NPF from 3 to 99.

In most cases, the difference in metrics between FIFO and custom algorithms is really small compared to the actual values, so the lines for FIFO and custom in the below graphs are overlapping each other.

1. alpha Program

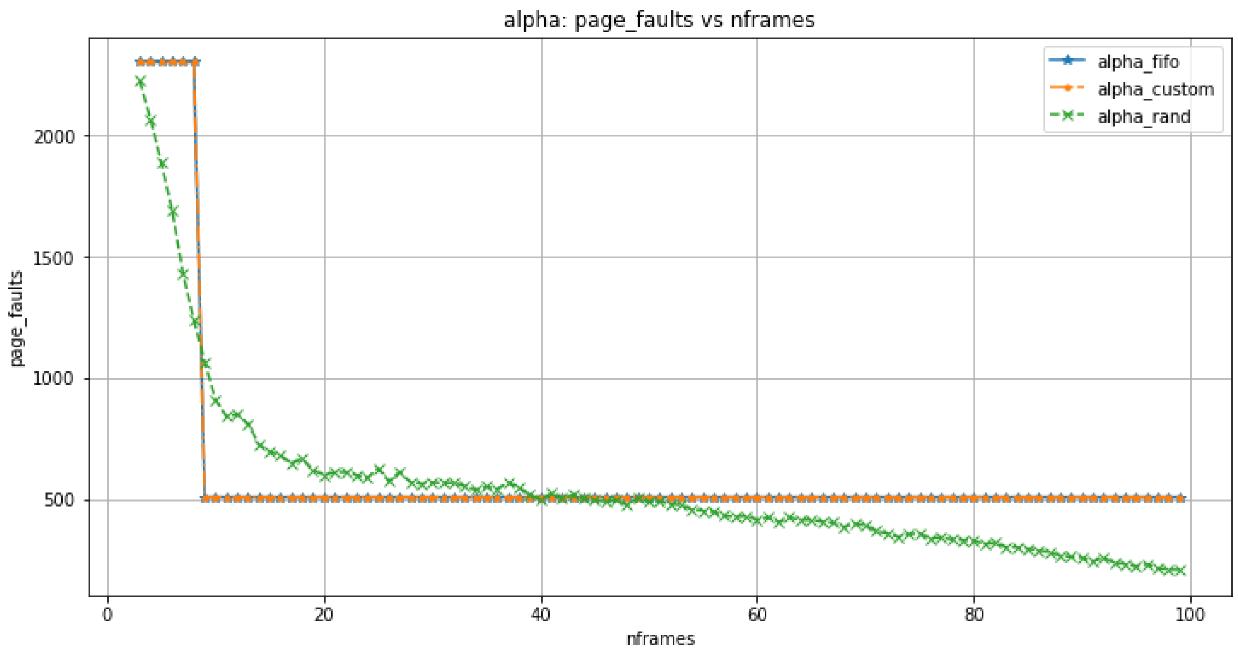

Fig. 2. alpha Program: Page Faults vs NPF

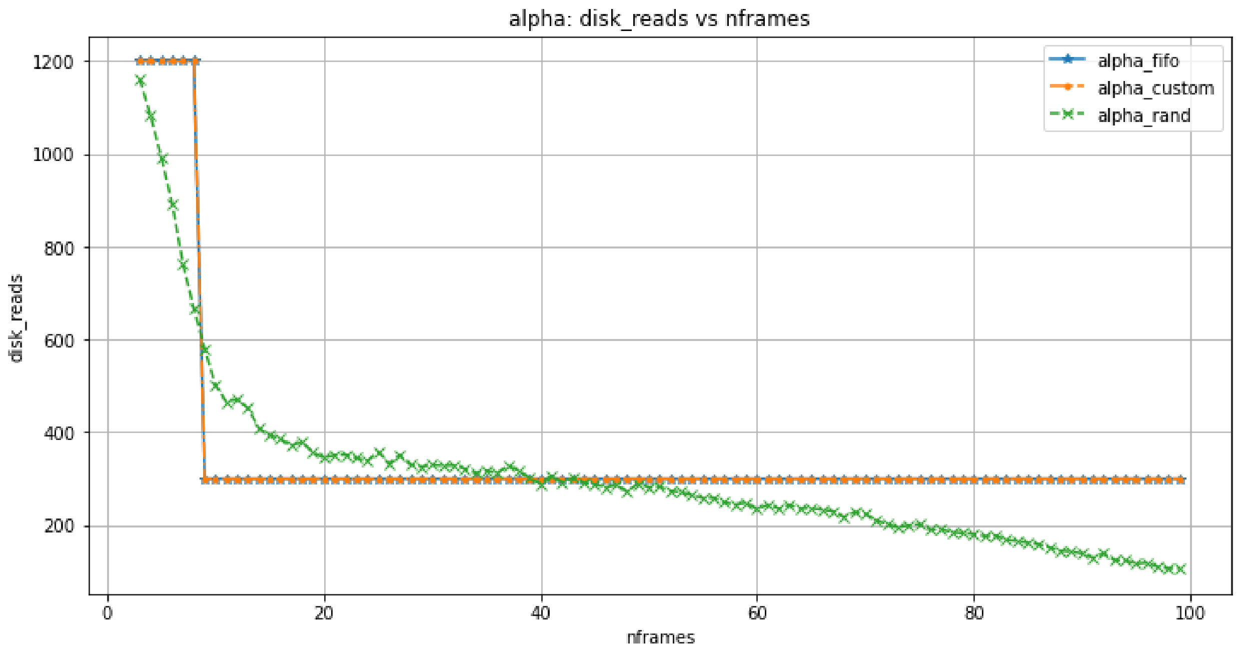

Fig. 3. alpha Program: Disk Reads vs NPF

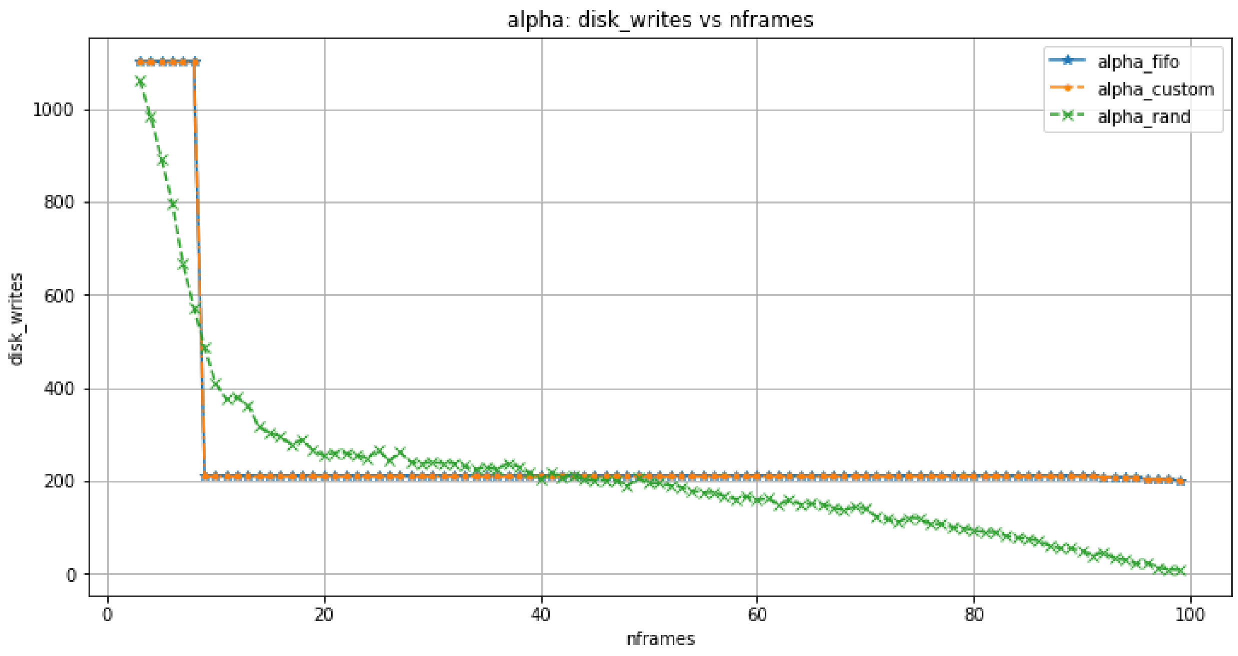

Fig. 4. alpha Program: Disk Writes vs NPF

The results from the 3 graphs for the alpha program which is a modifying memory program indicate that:

- For the alpha program, the graphs for page faults, disk reads, and disk writes (Fig. 2, Fig. 3, and Fig. 4) are almost identical. This suggests that the program's behavior affects all three metrics similarly. In the range of 10-50 NPF, both the custom and FIFO algorithms perform better than the random algorithm.

- This indicates that these algorithms are more efficient in managing memory for the alpha program within this specific range of page frames.
- The random algorithm performs better in other ranges of page frames outside the 10-50 range.
- The performance difference between the custom and FIFO algorithms is very small, with the lines in the graphs overlapping. This suggests that the two algorithms have similar efficiency in managing memory for the alpha program.

2. <u>beta Program</u>

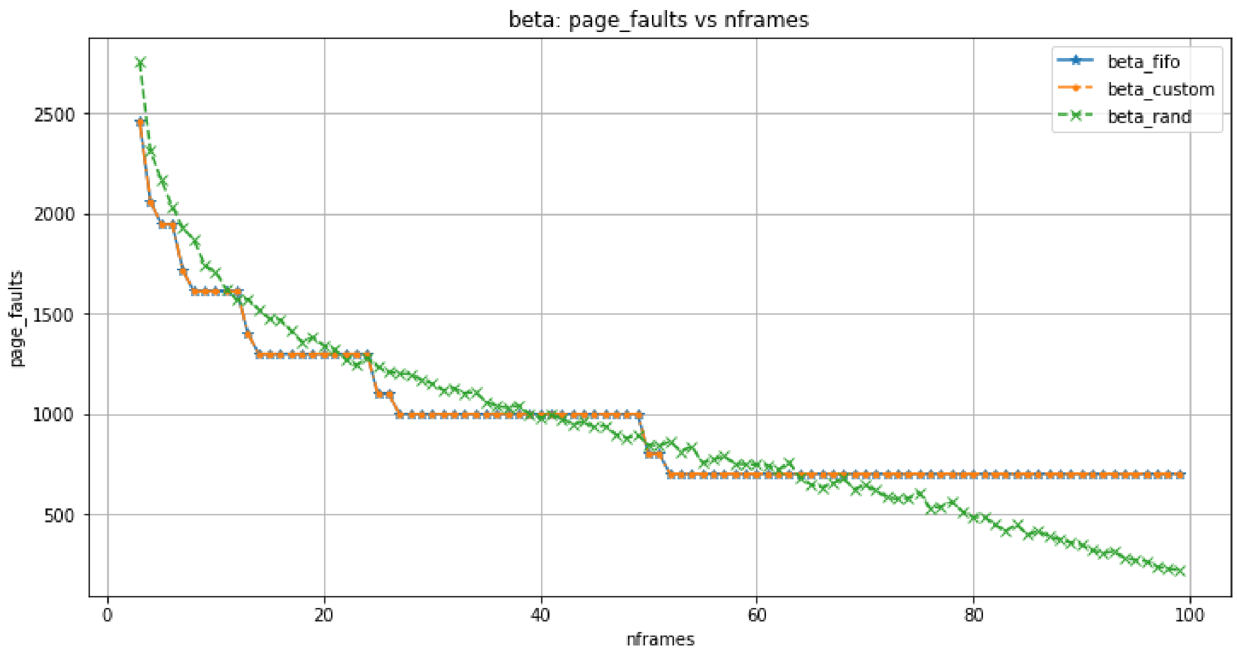

Fig. 5. beta Program: Page Faults vs NPF

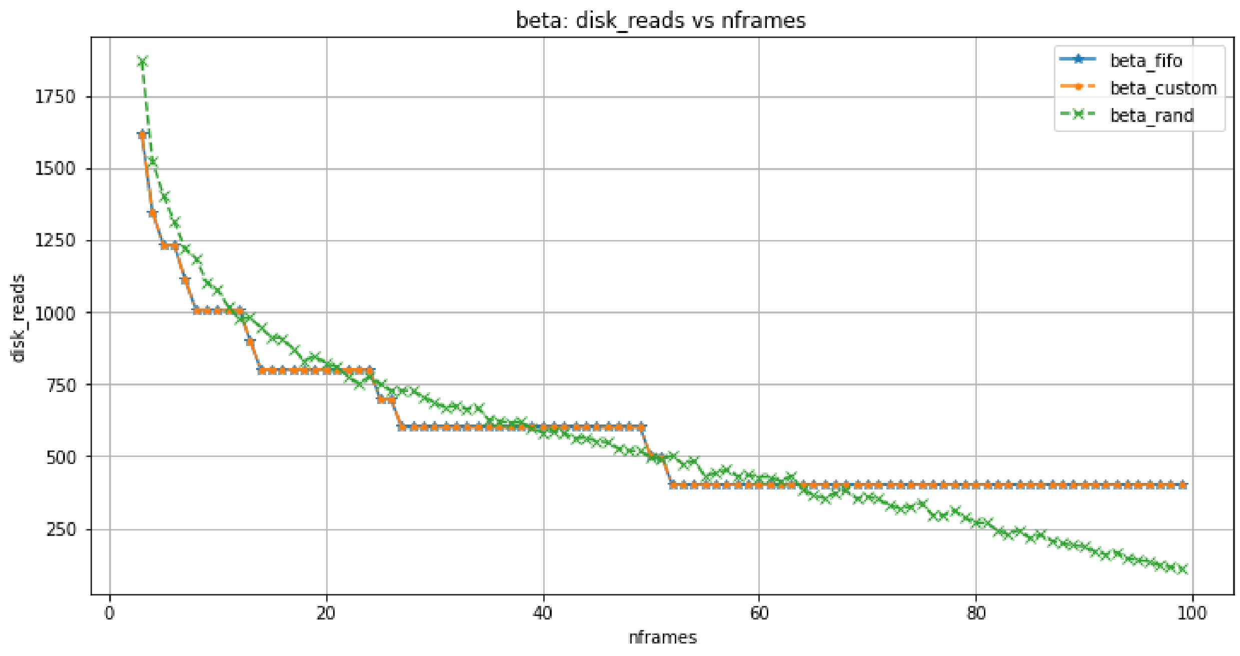

Fig. 6. beta Program: Disk Reads vs NPF

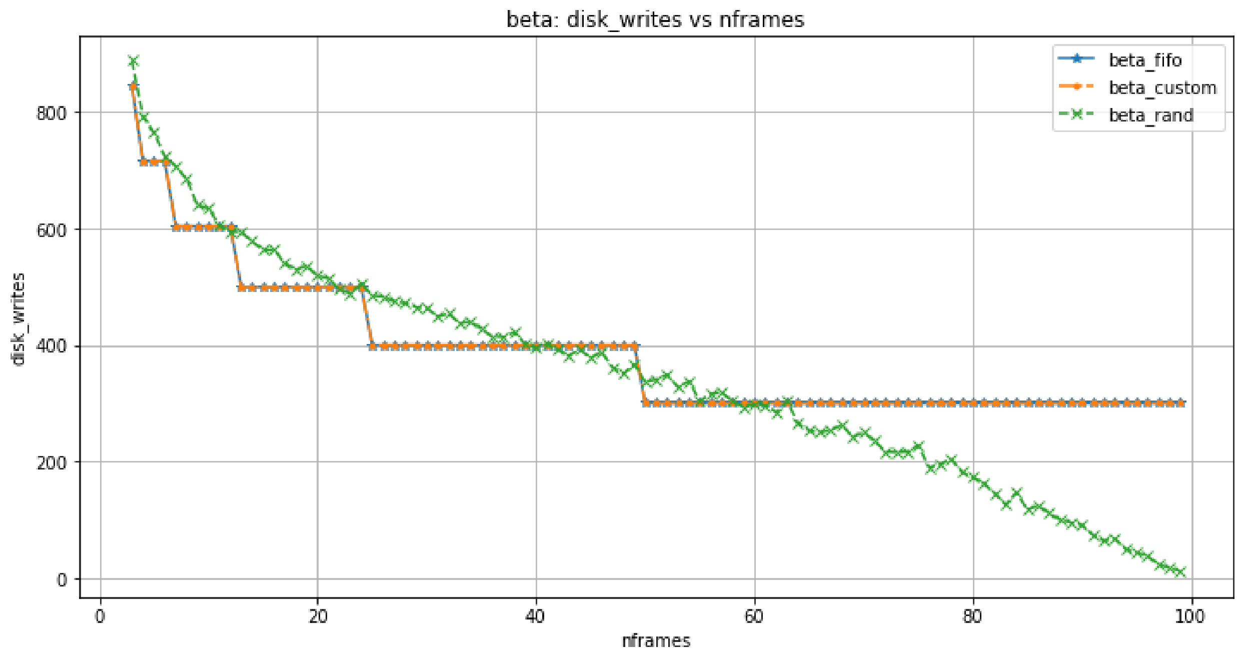

Fig. 7. beta Program: Disk Writes vs NPF

The results from the 3 graphs for the beta program which is a sorting memory program indicate that:

- The graphs for page faults, disk reads, and disk writes (Fig. 5, Fig. 6, and Fig. 7) show the lines for FIFO and custom algorithms overlapping, indicating similar performance between the two.
- The performance pattern for FIFO and custom algorithms resembles a staircase with increasing segments, while the random algorithm's line consistently decreases as the number of page frames (NPF) increases.
- In the range of 3-40 NPF, the custom algorithm performs better than the other two.
- Between 40-60 NPF, it is unclear which algorithm performs better, as the custom and FIFO algorithms might be slightly better in some areas, while the random algorithm may excel in others.
- From 60-99 NPF, the random algorithm is significantly better, with the performance difference becoming more pronounced as NPF increases.
- The disk writes graph (Fig. 7) shows an even larger difference between the random algorithm and the other two algorithms in the 40-60 NPF range, compared to the page faults and the disk reads graphs. This suggests that the random algorithm is particularly more efficient in managing disk writes for the beta program within this specific range of page frames.

3. <u>gamma Program</u>

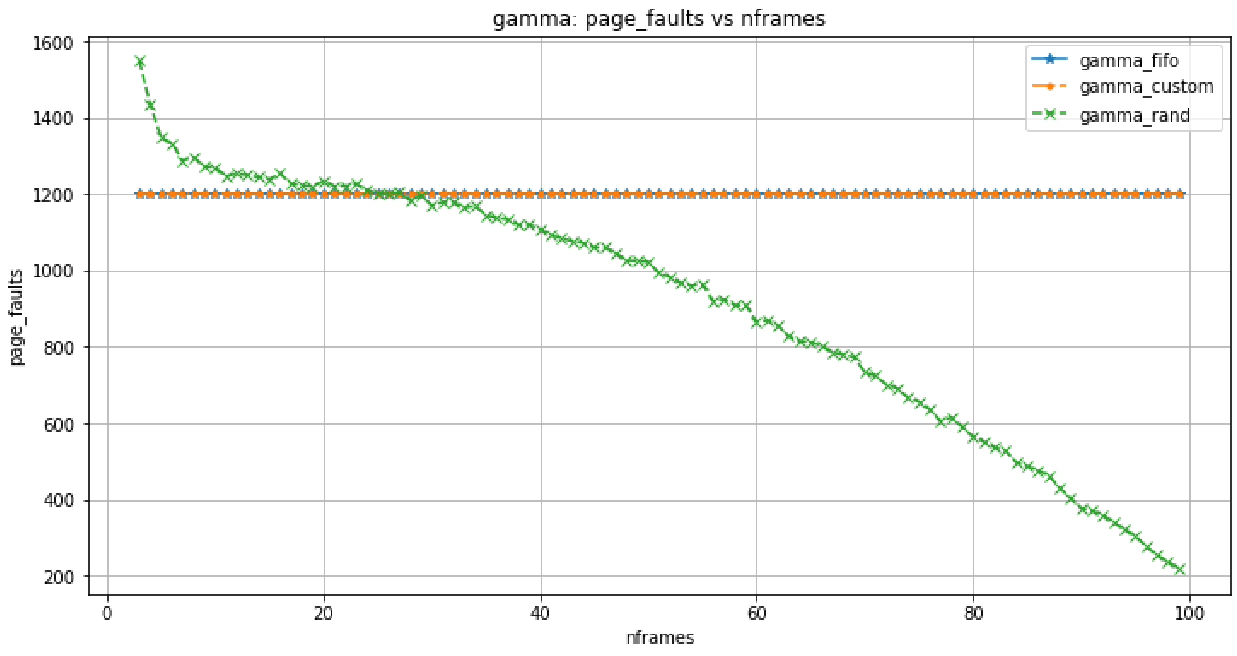

Fig. 8. gamma Program: Page Faults vs NPF

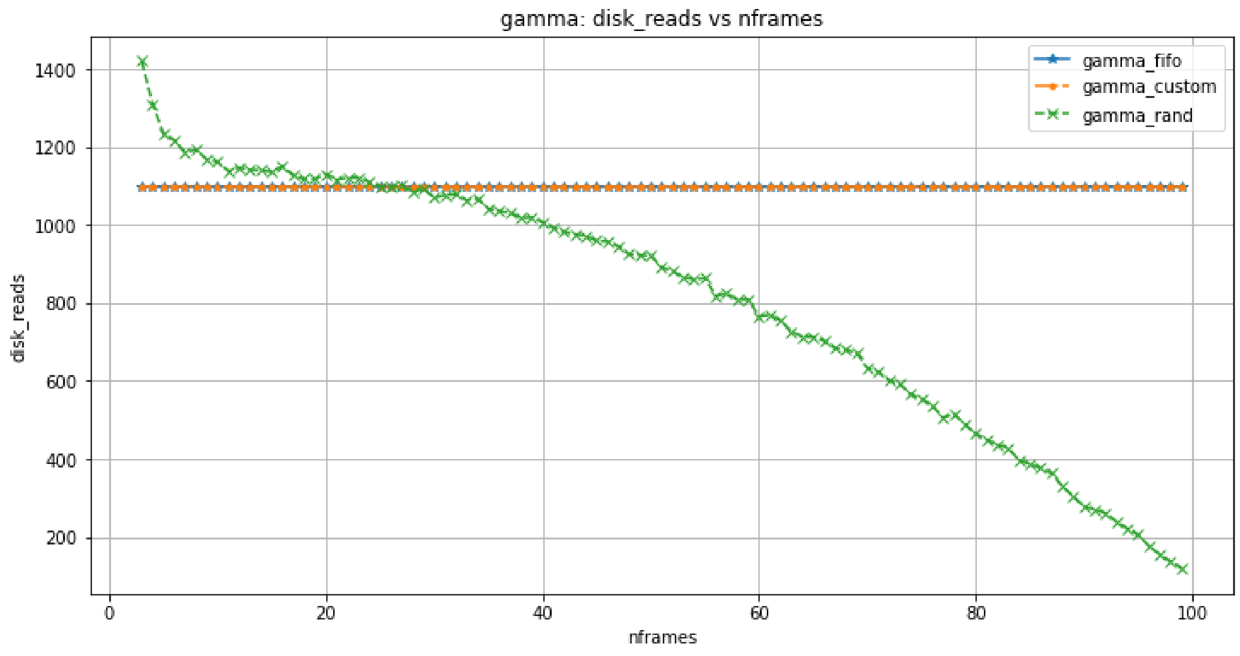

Fig. 9. gamma Program: Disk Reads vs NPF

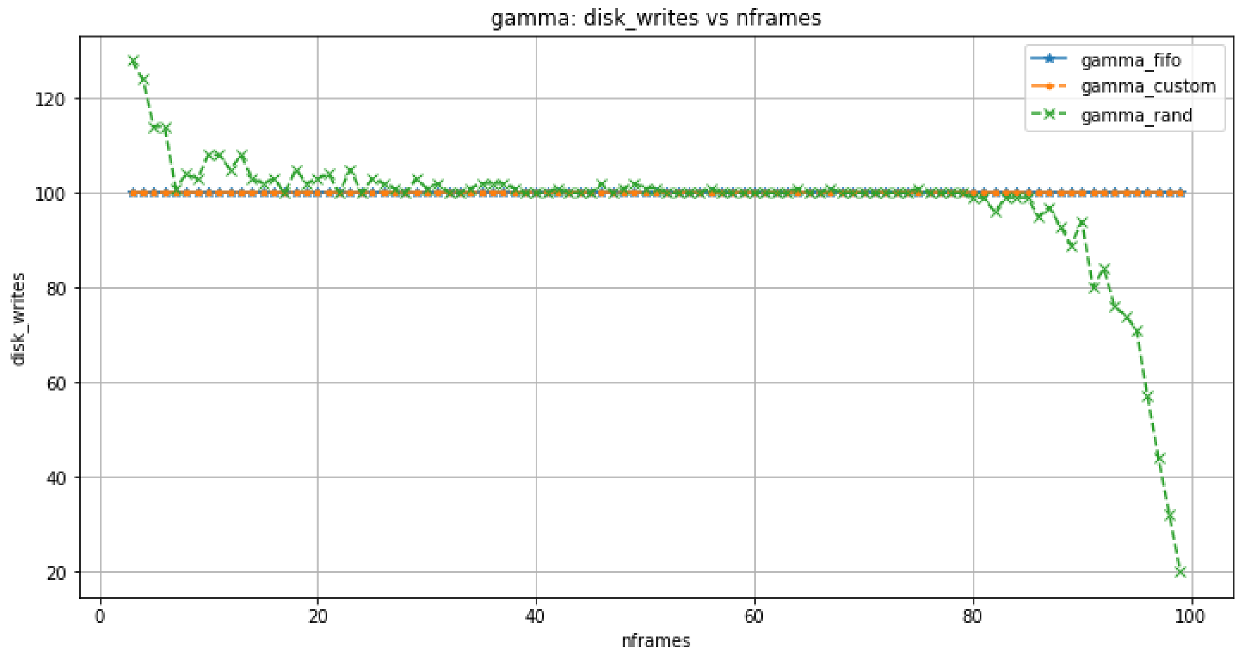

Fig. 10. gamma Program: Disk Writes vs NPF

The results from the 3 graphs for the gamma program which is a computing dot product program indicate that:

- The page faults and the disk reads graphs (Fig. 8 and Fig. 9) have similar trends. In disk writes (Fig. 10) graph, all three algorithms results in approximately the same number of disk writes (100) within the range of 30-80 NPF.
- The custom and FIFO algorithms result in 1200 page faults, 1100 disk reads, and 100 disk writes consistently, indicating similar performance between the two.
- The random algorithm shows a continuous improvement as NPF increases, potentially making it a more suitable choice for larger NPF values for the gamma program.
- This also suggests that, in comparison with FIFO and the random algorithm, the random algorithm becomes more efficient at managing memory for the gamma program as the number of available page frames increases.

4. <u>delta Program</u>

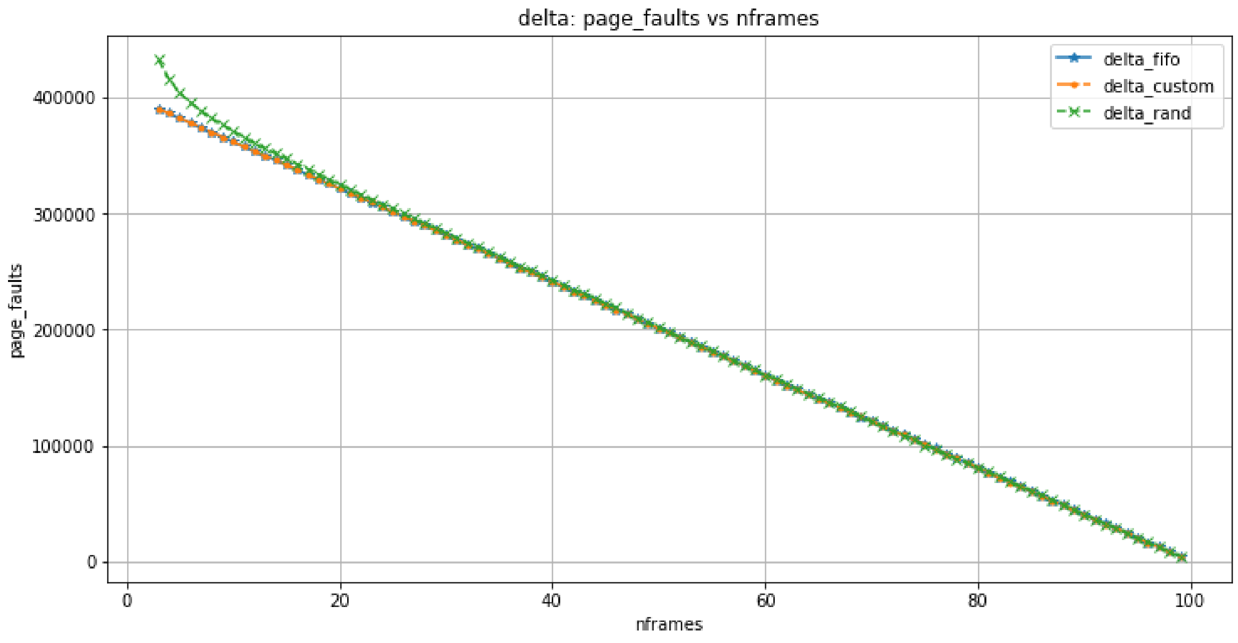

Fig. 11. delta Program: Page Faults vs NPF

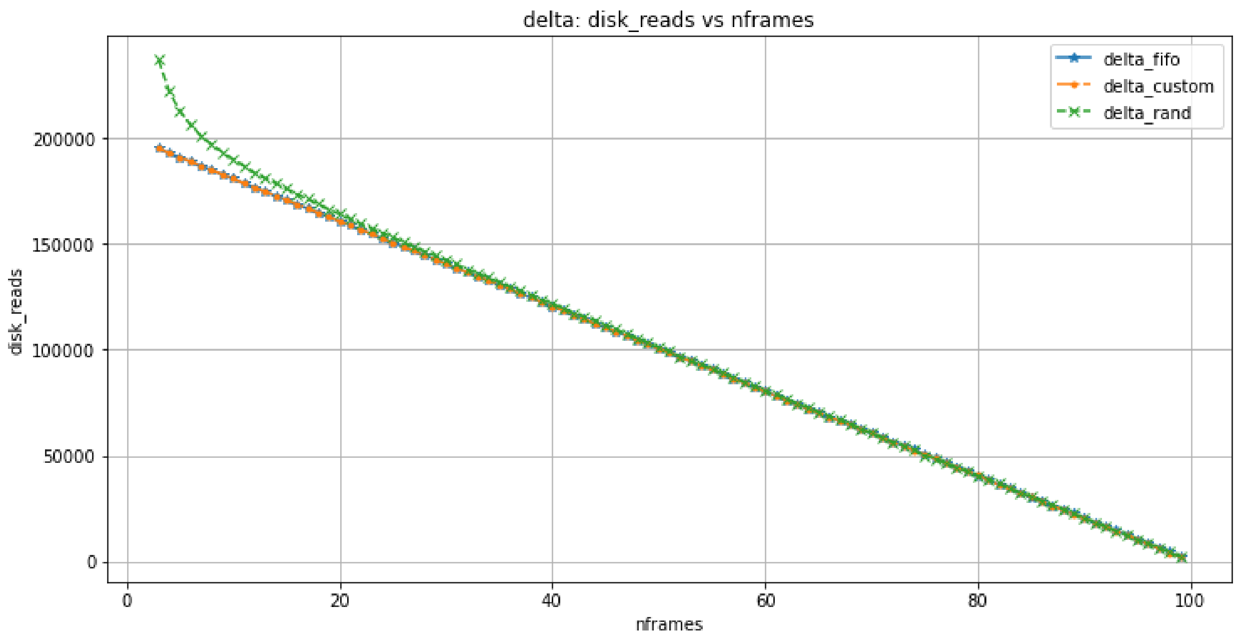

Fig. 12. delta Program: Disk Reads vs NPF

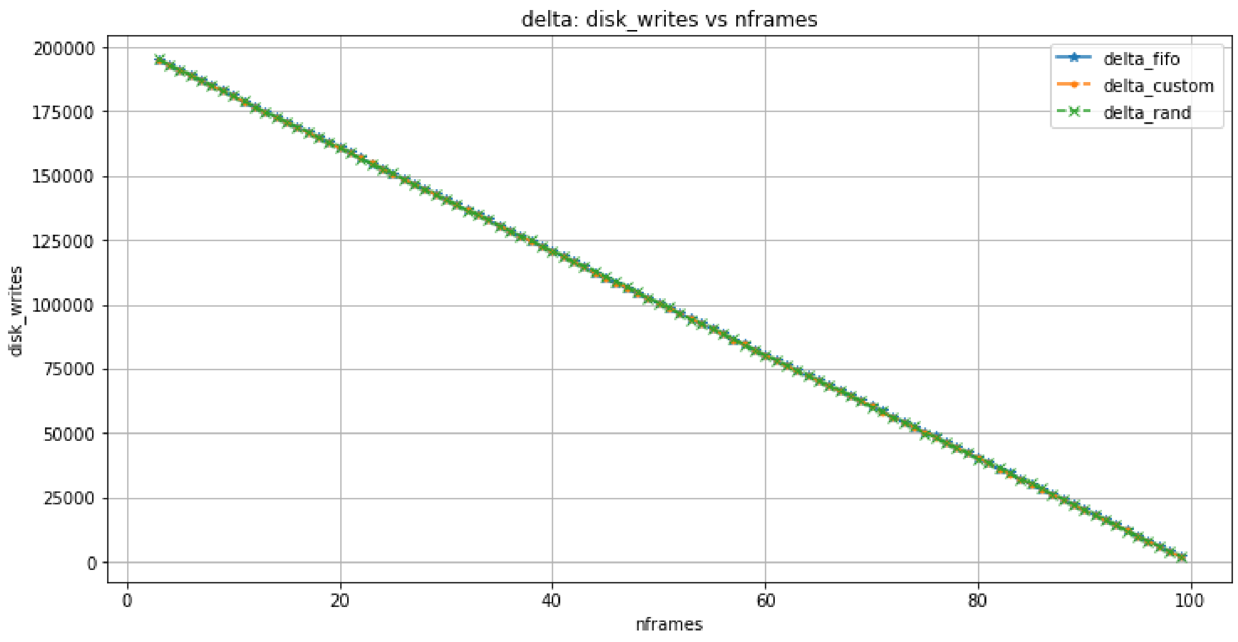

Fig. 13. delta Program: Disk Writes vs NPF

The results from the 3 graphs for the delta program which is a computing dot product program indicate that:

- All three algorithms overlap in the disk writes graph (Fig. 13) and have similar steady downward trend as the NPF increases.

- In the page faults and disk reads graphs (Fig. 11 and Fig. 12), the random algorithm's line is slightly curved down from 3-10 while custom and FIFO form a linearly declining trend as the NPF increases. This means that, for this program, the custom and FIFO algorithms perform slightly better than the random algorithm.
- In fact, the custom algorithms has fewer page faults, disk reads, and disk writes compared to random and FIFO algorithms. However, since the difference are so small compared to the actual values, we cannot see them from the graphs above.
- Below are comparison graph between custom algorithms and and FIFO algorithms for all three metrics page faults, disk reads, and disk writes:

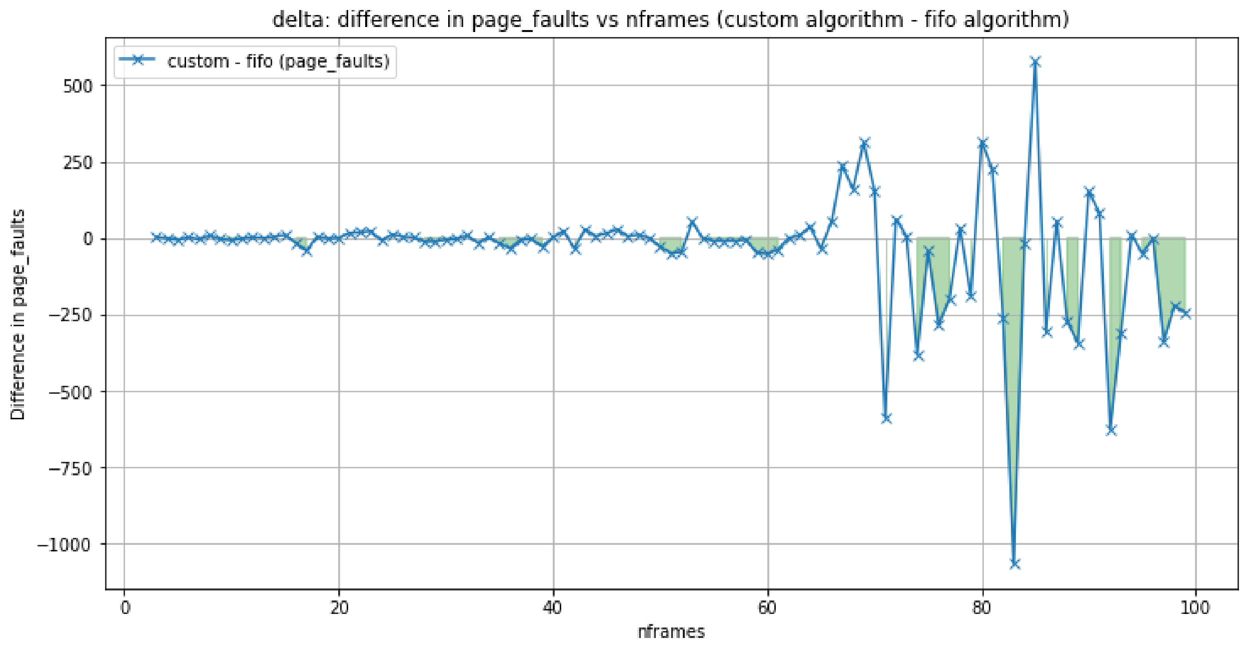

Fig. 14. The difference in page faults between custom and FIFO algorithms over the NPF (3-99)

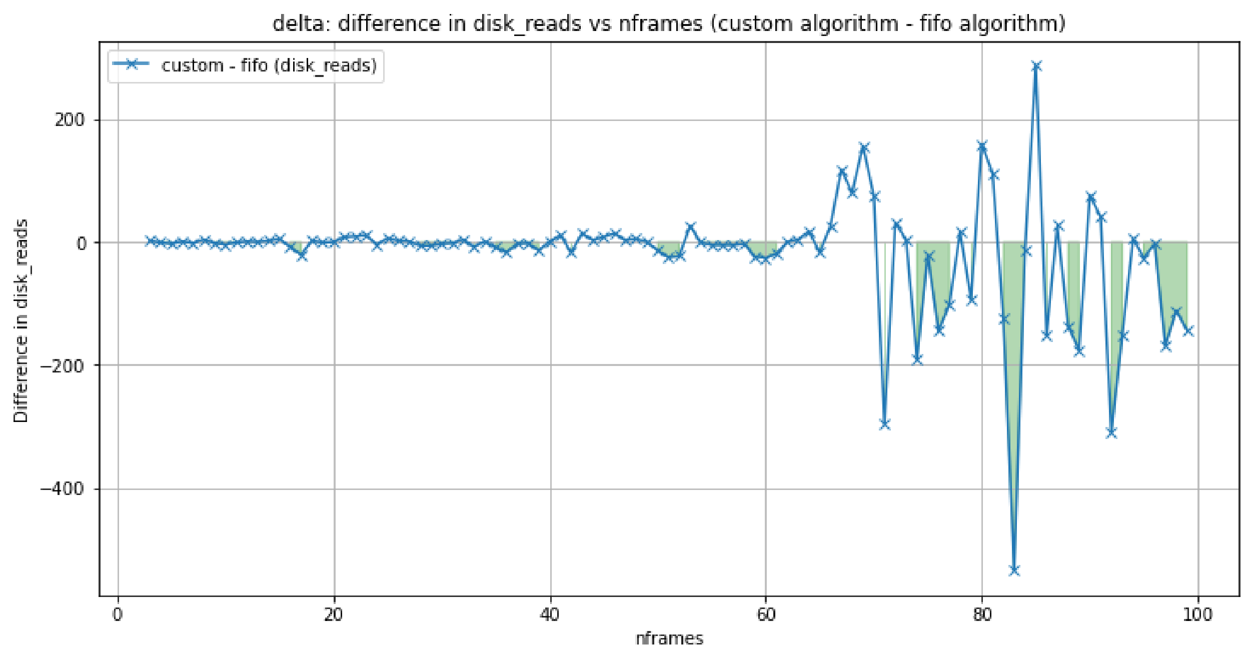

Fig. 15. The difference in disk reads between custom and FIFO algorithms over the NPF (3-99)

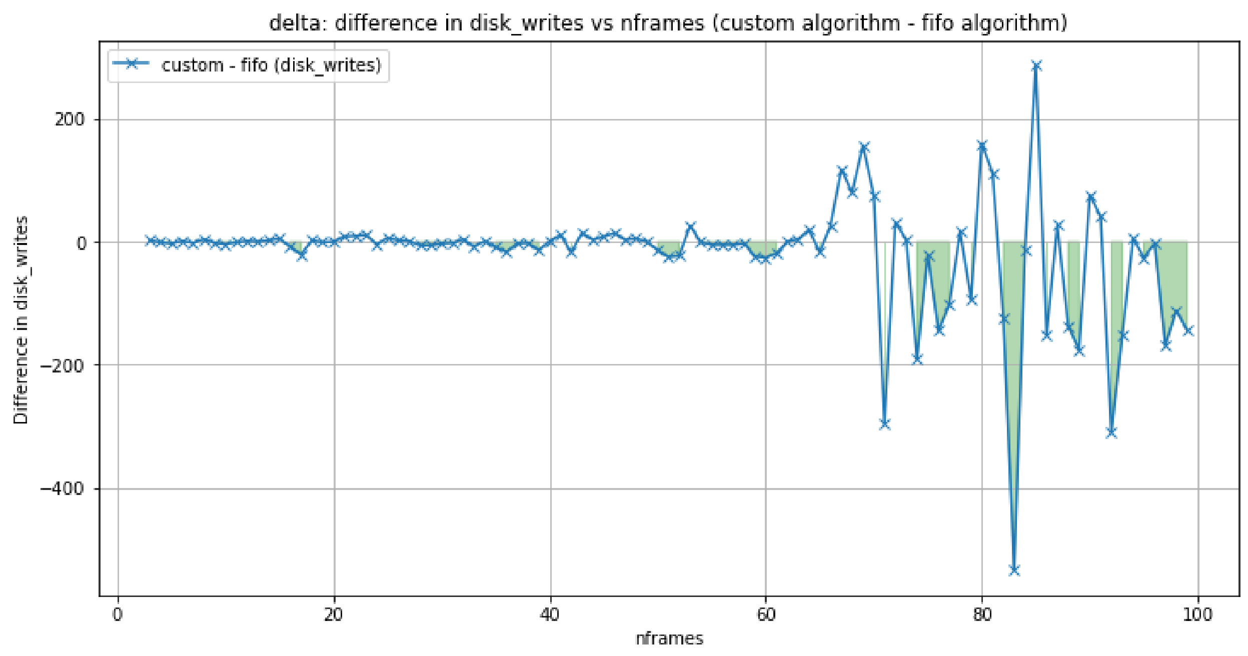

Fig. 16. The difference in disk writes between custom and FIFO algorithms over the NPF (3-99)

- From the three graphs represent the difference between custom and FIFO algorithms, the presence of more negative (green) ranges indicates that the custom algorithm results in less page faults, disk reads, and disk writes. This means that the LRU-based custom algorithm seems to perform better than both random and FIFO algorithms for the delta program.

VI. Analysis

1. Program Characteristics and Algorithm Performance

The performance of different page replacement algorithms varies based on the characteristics of the program used in the experiments. This emphasizes the importance of understanding a program's memory access patterns and help developers in choosing the appropriate algorithm to optimize virtual memory management.

a. alpha Program

In the alpha program, which modifies memory, custom and FIFO algorithms perform better than the random algorithm in the 10-50 NPF range. This is because both custom and FIFO algorithms take into account the order of memory accesses, allowing them to make more informed decisions when replacing pages. In the given NPF range, this characteristic allows them to manage memory more efficiently than the random algorithm, which selects a page to replace without considering memory access patterns.

b. beta Progam

For the beta program, which sorts memory, the custom algorithm performs better in the 3-40 NPF range. This can be attributed to the custom algorithm's LRU (Least Recently Used) mechanism, which favors retaining more frequently accessed pages in memory. Sorting operations often access data sequentially or in a pattern, which the LRU-based custom algorithm can exploit to minimize page faults.

The random algorithm, however, performs significantly better in the 60-99 NPF range. This is likely due to the larger number of available page frames reducing the impact of random page replacement decisions. In this case, the random algorithm can take advantage of the increased memory resources without suffering from its lack of memory access pattern awareness.

c. gamma Program

In the gamma program, which computes dot products, the random algorithm shows continuous improvement as NPF increases. This could be attributed to the program's memory access pattern, which might involve a more uniform distribution of memory accesses across pages. As the number of available page frames increases, the random algorithm's performance benefits from the greater probability of selecting a less frequently accessed page for replacement.

d. delta Program

For the delta program, which swaps random locations, the custom algorithm performs slightly better than both FIFO and random algorithms. The LRU-based custom algorithm is better suited to this program because it takes into account the recency of memory access, allowing it to make more informed decisions when replacing pages. In a program that swaps random locations, memory access patterns may be unpredictable but can still benefit from an LRU-based approach, as it can retain the most recently accessed pages in memory and minimize page faults.

2. Scalability

The experiments also reveal that the performance of different algorithms may change as the number of page frames increases. For example, for the gamma program, which computes dot products, the performance of the random algorithm improves as the number of available page frames increases. This improvement can be attributed to the fact that, with a larger number of page frames, the random algorithm has a higher probability of choosing a less frequently accessed page for replacement. As a result, the impact of its lack of awareness of memory access patterns diminishes, allowing it to manage memory more efficiently.

This observation highlights the importance of considering the scalability of an algorithm in the context of the virtual memory system's size. An algorithm that performs well in a smaller virtual memory system may not necessarily be the most efficient choice for larger systems. Conversely, an algorithm that struggles with smaller systems could potentially be more effective when more page frames are available.

VII. Conclusion

In conclusion, the experiments conducted in this project have provided valuable insights into the behavior and efficiency of various page replacement algorithms under different conditions. The custom LRU-based algorithm generally shows better performance across different programs, indicating its effectiveness in managing virtual memory. However, it is essential to consider the program's memory access patterns, the size of the virtual memory system, and other factors when selecting the most suitable page replacement algorithm for a specific scenario.