Virtual Memory Simulation

Assignment 2 report - Group Prac2 UGG3
Shaoxuan Tang a1838275
Daniel Terrison a1744072

Introduction

In the dynamic world of computer science, the efficiency and efficacy of memory management form the bedrock of system performance. At the heart of this issue lies the concept of virtual memory, a system component that gives an application the impression it is using large contiguous blocks of memory, even when it isn't. Such a setup allows for the efficient execution of large applications and multitasking operations. However, the management of virtual memory brings forth its own unique challenges, particularly in the realm of page replacement.

Page replacement algorithms determine how a system decides which pages to evict from the memory when a page fault occurs. The choice of which algorithm to use isn't straightforward; it requires an understanding of application behaviors and memory access patterns. Not all applications access memory uniformly or sequentially. Some might access memory pages in bursts, while others might have a more scattered access pattern. The intricacies of these patterns mean that the 'best' page replacement algorithm may vary depending on the application in question.

This study delves deep into this problem, attempting to discern the performance of various page replacement strategies against real-world memory traces. Using a virtual memory simulator, we evaluate the effectiveness of different algorithms against memory access patterns extracted from authentic applications. Our aim is not just to find the most efficient algorithm but to also understand the nature of memory access in real applications and how different algorithms cater to these patterns. Through this exploration, we seek to bridge the gap between theoretical algorithm performance and real-world applicability.

Methods

To determine the efficiency of different page replacement algorithms in a memory management unit (MMU) simulator, a set of experiments were performed using different traces and configurations. The MMU simulator was implemented in the C language. The simulator is designed to support various page replacement algorithms including:

1. Random Replacement: Selects a random frame for eviction.

2. First-In-First-Out (FIFO): Evicts the oldest page in memory (not implemented in our code).

3. Least Recently Used (LRU): Evicts the page that hasn't been accessed for the longest duration.

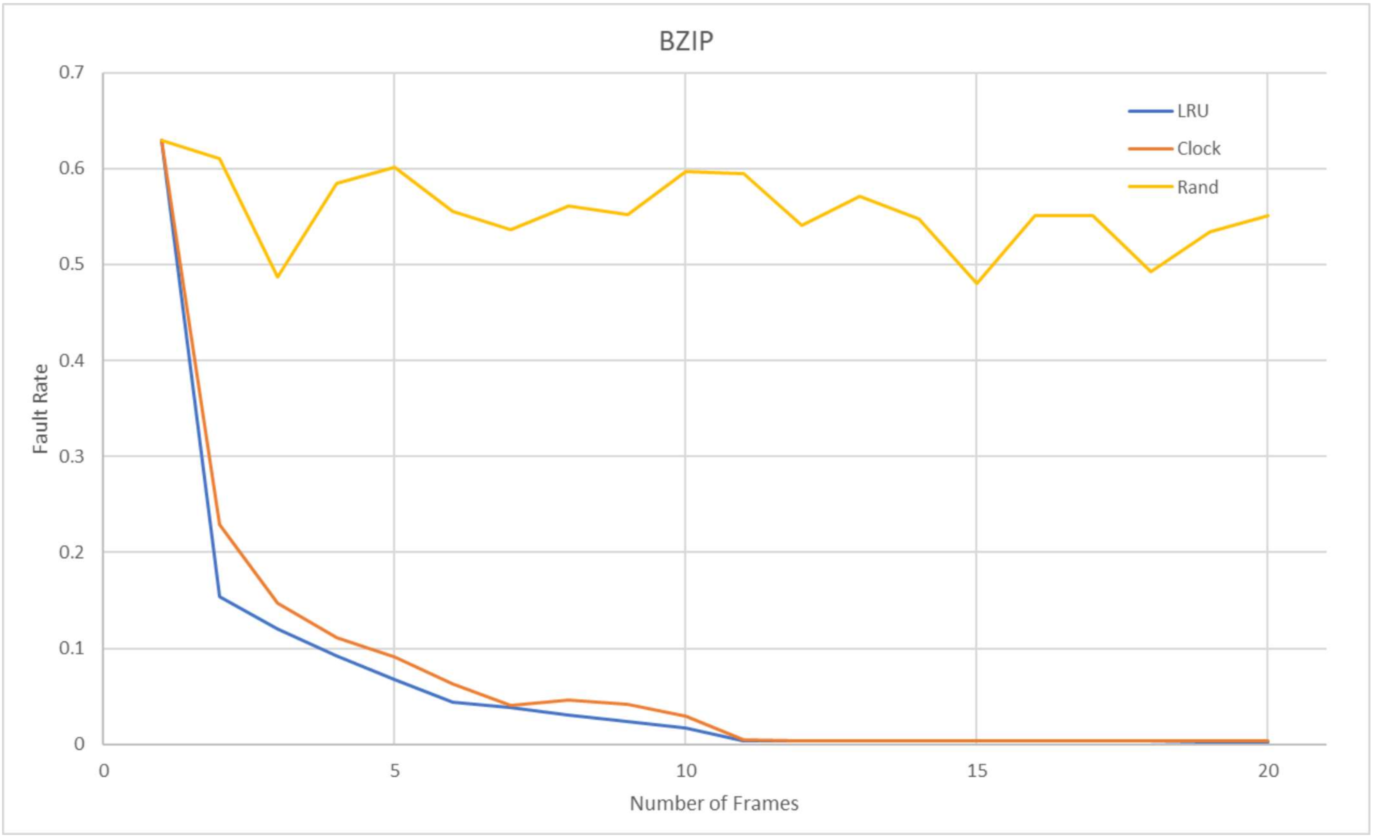
4. Clock Algorithm: Operates like a circular buffer with a second chance use bit, such that recently used pages get a second chance before eviction.

Each simulation involved reading memory access patterns from a trace file. Each line of the trace file contained a memory address and an operation (either Read or Write). For each access, the simulator checks if the page is already in memory. If not, a page fault occurs, leading to a disk read. If a page is evicted and has been modified (write operation), a disk write is necessary to save the content. We executed the memory management simulator on a set of memory traces using various memory sizes and replacement algorithms. The memory traces used were the four given example traces, taken from the four real programs: bzip, gcc, swim and sixpack. Each of these were one million operations, more than enough for a clear insight into the various replacement algorithms.

To simulate the effects of varying memory availability, the MMU simulator was run with a number of frames ranging from 1-1000 and the page fault rate was recorded. Due to the time needed to run these tests and the reduction in variance with larger numbers, the increment between tested values increased with size. For values 1-20, each number was tested. For 20-30, the tested values were in increments of 2. For 30-100, tested values were in increments of 5 and finally, 100-1000 was in increments of 100. This provided good detail in data at the lower end of values, while saving time and simplifying the data set in the larger range where results had little difference between tests. Additionally, since the random algorithm produces different results each time it is run, for each value it was run 3 times and an average of the three was taken. These tests for the random algorithm were only performed on the first two traces. This was done in the interest of time, and since its performance is significantly worse than the other algorithms the same level of scrutiny wasn't warranted.

Results

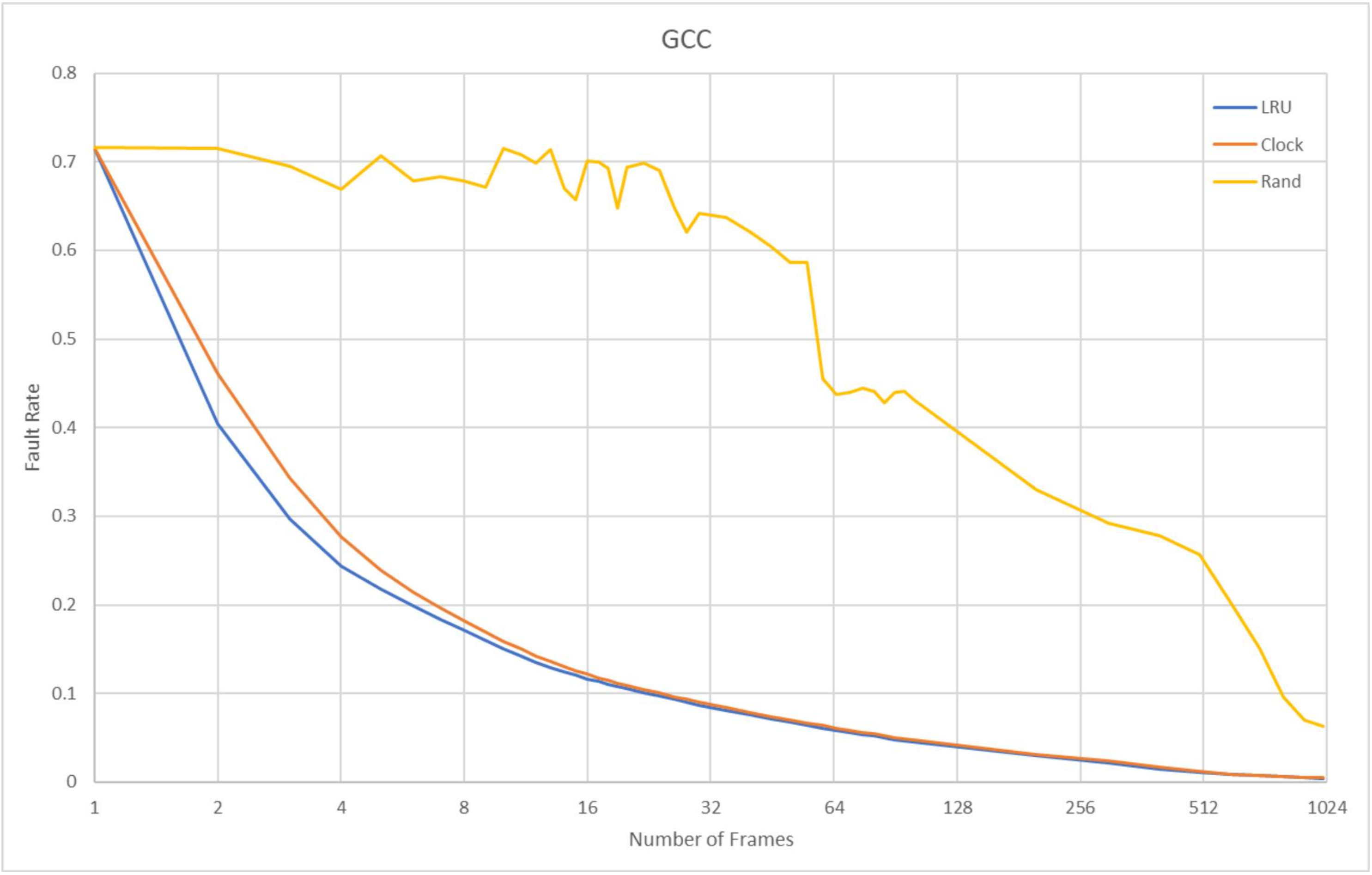
The goal was to understand the efficiency of these algorithms in terms of page faults, and see how this varies with the amount of memory provided to them. Thus, the plots below show the page fault rate for each trace with varying number of frames.



The first trace tested was the bzip trace. This program has much simpler memory usage and requirements than the other traces and thus the full range of test values were not necessary. Within the 20 values tested for LRU and clock, the page fault rate flatlined at a fixed value for both algorithms, so we believed there would be no further improvement with increasing frame count.

The most obvious point of data in the above plot is the massive difference in efficiency between LRU and clock, versus random replacement. LRU and clock both get significantly more efficient as the frame count increases, while random seems largely unaffected. The results are also very inconsistent. A larger dataset to pull the average from would likely improve this, but we felt this wasn’t warranted due to the lack of efficiency.

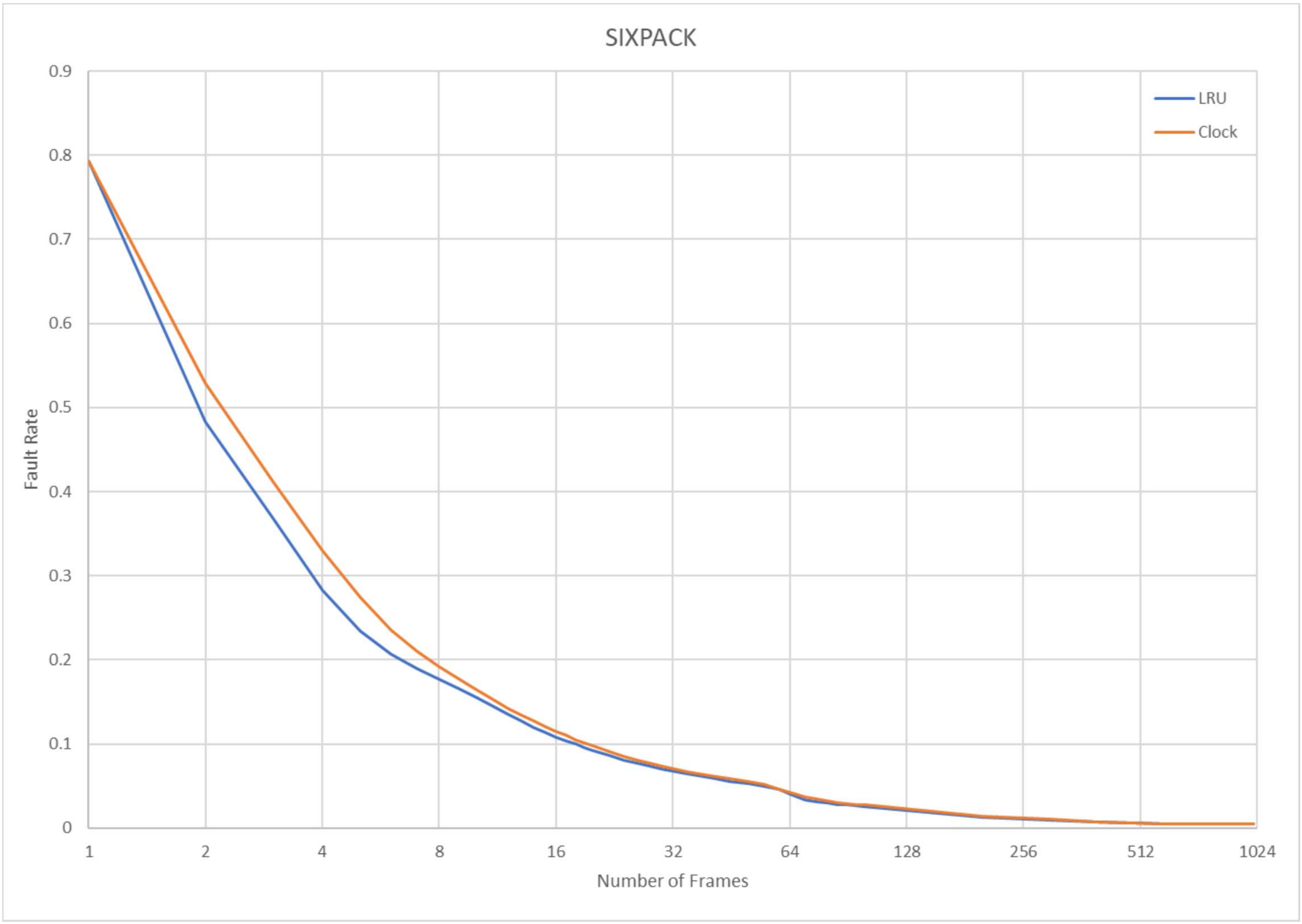
LRU and Clock algorithms both show a similar trend for this trace. Starting at a high page fault rate for a single frame, the fault rate decreases considerably as the frame count increases. This decrease is more pronounced for LRU as compared to Clock, especially when transitioning from 1 to 2 frames. Beyond 11 frames, there's a minimal difference in page fault rates for both algorithms, suggesting that for bzip trace, anything beyond 11 frames wouldn't offer significant performance gains. For this trace, 1-5 frames would be considered a shortage of memory, with the lack of space resulting in a very high fault rate. For 5-10 frames, the fault rate is acceptable, however 11 frames appears to be the optimal amount. Increasing beyond 11, there is an excess of memory and so the fault rate barely changes.



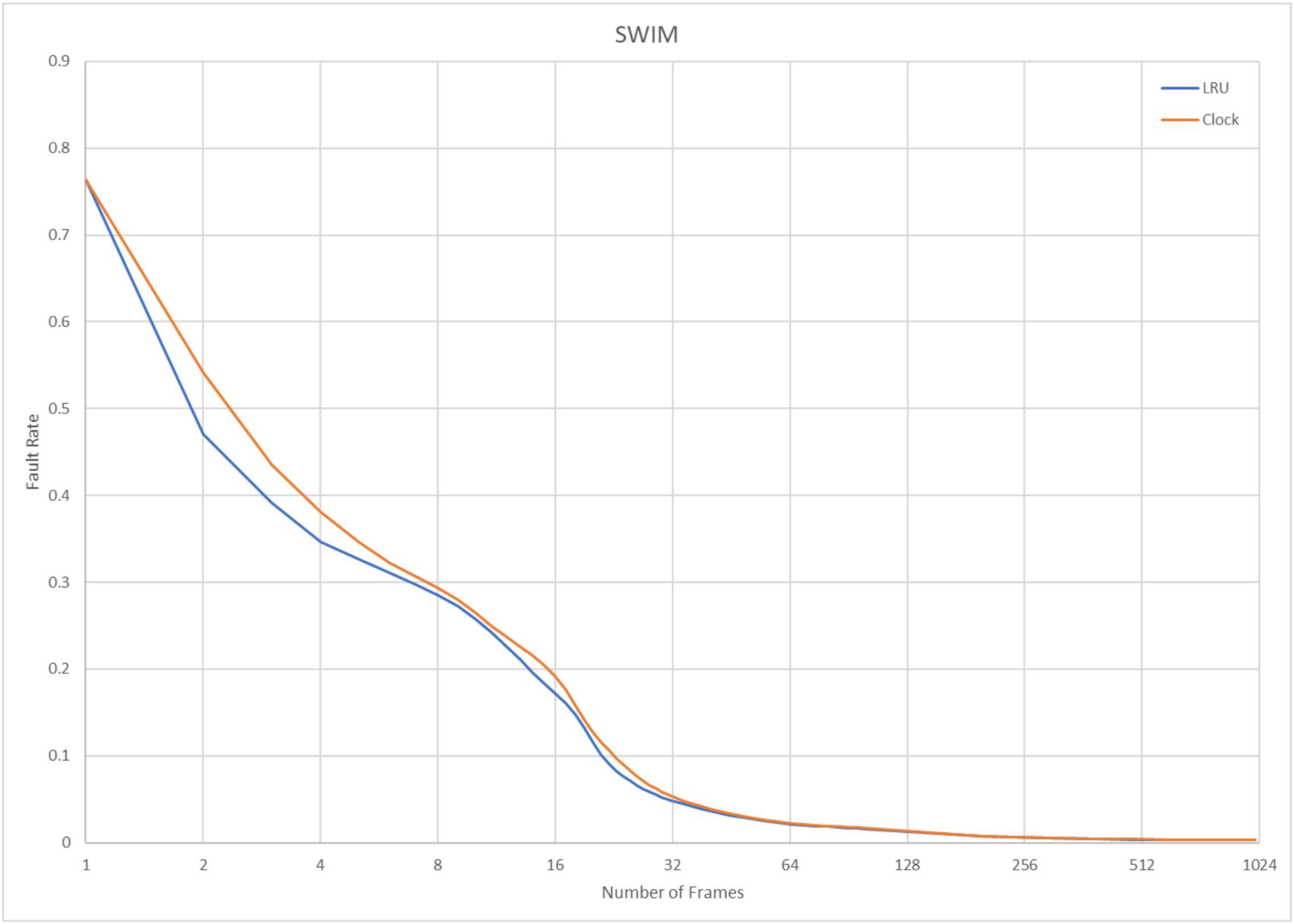
The above plot, along with the following two, use a base 2 logarithmic scale for the x axis. This was done to provide better readability at the low end where there are larger differences in results.

Again in the above plot we can see a massive difference in efficiency between LRU and clock, versus random replacement. However, with the larger data set this time we can see some additional characteristics of random replacement. Mainly that it does improve with increasing frame count. As the frame count gets quite large, random replacement does get more efficient. However, it improves at a much slower rate and never gets close to the efficiency of the other two algorithms. The massive difference in the above plots shows the lack of efficiency in the random replacement algorithm and highlights the fact that it should never be used when alternative algorithms are available.

This trace is much more demanding than bzip, reflected in the higher fault rates across all frame counts. Both LRU and Clock demonstrate a clear benefit as the frame count increases, however LRU tends to perform slightly better than Clock throughout the range of frames. There's a considerable difference in page fault rates between LRU and Clock at lower frame counts, suggesting that LRU might be more suited for this workload. The difference between LRU and Clock narrows down as the frame count increases. By 1000 frames, both algorithms have nearly identical page fault rates, indicating diminishing returns on performance benefits beyond this point. Thus, we can conclude that roughly 1000 frames, or roughly 4 MB, is the optimal amount of memory for this program.



The sixpack trace has a trend similar to the gcc trace. LRU tends to perform slightly better than Clock throughout the range of frames, with both LRU and Clock algorithms benefiting from increased frame counts. The difference between LRU and Clock narrows down as the frame count increases. By 1000 frames, both algorithms have nearly identical page fault rates, indicating diminishing returns on performance benefits beyond this point. Again, roughly 1000 frames is ideal for this program.



The swim trace presents a more varied pattern. For lower frame counts, the difference between LRU and Clock is more pronounced, but as the frame count increases, the two algorithms converge in performance. Beyond 500 frames, the gains in performance from adding more frames are minimal for both algorithms. Thus, roughly 500 frames, or roughly 2MB, is ideal for this program.

Conclusions:

- Across all traces, the LRU algorithm tends to perform at par or slightly better than the Clock algorithm, particularly at lower frame counts.

- Both LRU and Clock algorithms display diminishing returns on performance as the frame count increases. For most workloads, beyond a certain point, adding more frames won't substantially decrease the page fault rate.

- The data does not provide full information on the performance of the random replacement strategy for all traces, but based on the context provided in the report, it's evident that random replacement is considerably less efficient than both LRU and Clock. This reaffirms the notion that deterministic algorithms like LRU and Clock, which make decisions based on historical data, tend to be more efficient than a stochastic strategy like random replacement.

- The choice between LRU and Clock would likely come down to the specific requirements of the system and the nature of the workload. While LRU typically performs slightly better, the Clock algorithm might be chosen for its simplicity and lower overhead in certain scenarios. LRU cannot be implemented on all hardware, while the clock algorithm can. Thus, for systems unable to use LRU, clock provides an excellent alternative.