Analysis of Page Replacement Algorithms across Four Memory Traces

Author: Zhuang Tang

Introduction:

Page replacement algorithms are fundamental to the efficiency of virtual memory systems. When a page must be retrieved and memory is full, the system must decide which existing page to replace. This research aims to discern the performance of several page replacement algorithms across four distinct memory traces: swim, bzip, gcc, and sixpack. The objective is to understand how these algorithms behave in different memory conditions, helping in selecting the most effective approach.

Methods:

To probe the efficiency and performance of various page replacement algorithms with changing memory sizes across our four chosen memory traces (bzip, GCC, sixpack, and swim), we pursued a structured methodology. This section delineates our experimental approach, aiming to give an exhaustive rundown of our simulation procedures.

1. Algorithms Utilized: We selected the following six page replacement algorithms due to their widespread use and relevance in memory management:

- **First-In, First-Out (FIFO)**: This algorithm ousts the most elderly page in memory.
- **Least Recently Used (LRU)**: Ousts the page that hasn't been accessed for the most prolonged period.
- **CLOCK**: A representation of LRU employing a circular queue along with a reference bit.
- **Least Frequently Used (LFU)**: Evicts the page with the fewest accesses.
- **Random (RAND)**: Arbitrarily picks a page for ousting.
- **Adaptive Replacement Cache (ARC)**: Melds the advantages of LRU and LFU, flexibly adapting to workload dynamics.

2. Memory Traces Used: Our simulations incorporated the subsequent memory traces:

- **bzip**: A mechanism for data compression.
- **GCC**: GNU Compiler Compilation.
- **sixpack**: A form of lossless data compression.
- **swim**: A module for scientific computational tasks.

3. Frames Allocation Strategy: For each trace, we modulated the quantity of memory frames to replicate scenarios of:

- **Excess Memory**: Here, we endowed more frames than a trace necessitated. This provided insights into algorithm behaviors when memory availability isn't stringent.
- **Perfect Memory**: Designated the precise number of frames a trace fundamentally demands, offering a vantage point to gauge algorithm performance in ideal settings.
- **Shortage of Memory**: We consciously offered fewer frames, setting the stage where algorithms regularly confronted page faults, thereby gauging the resilience and adeptness of each in demanding

conditions.

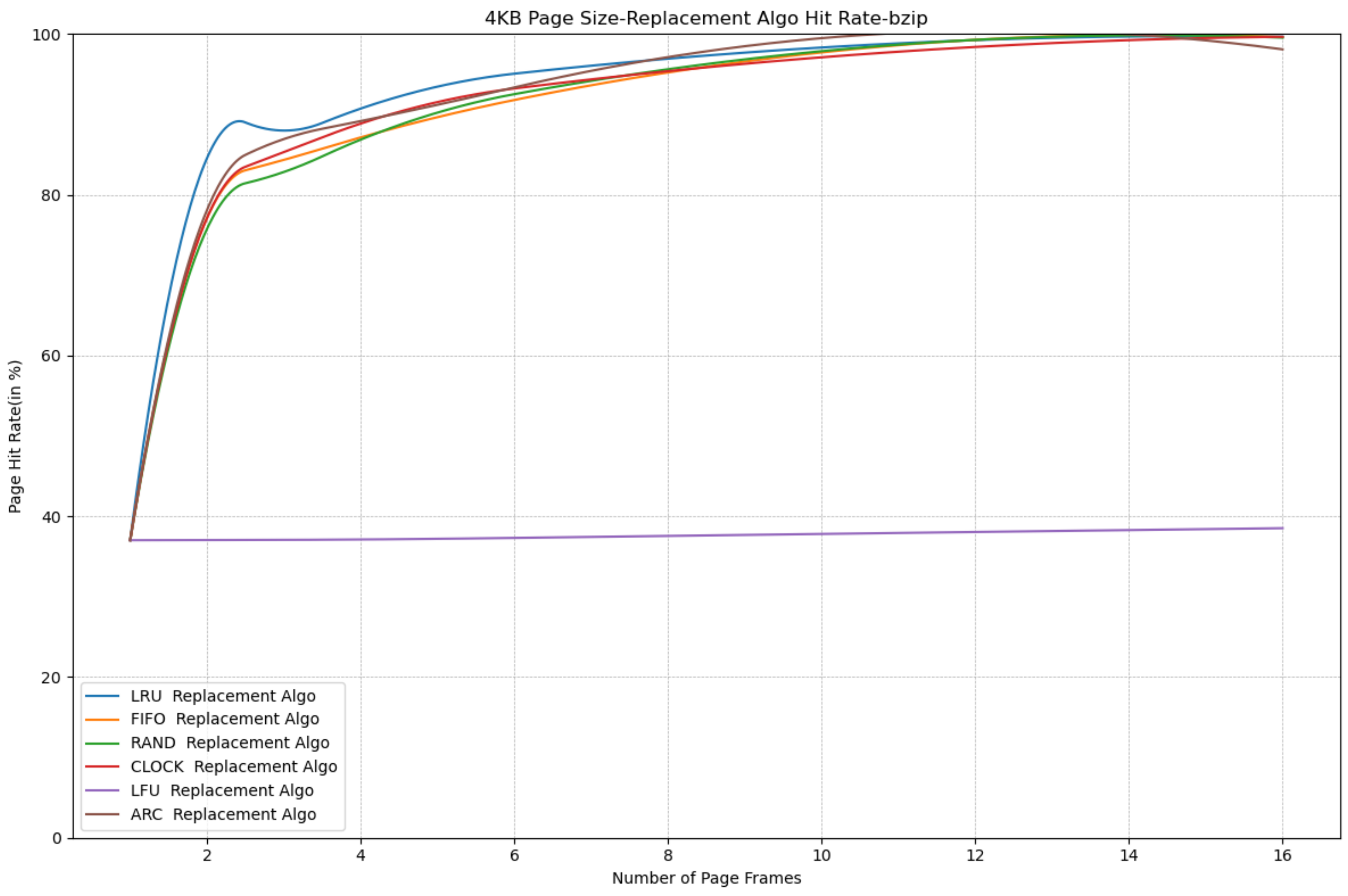
4. Metrics Used: The pivotal metric for performance gauging was the 'Page Hit Rate'. This metric denotes how frequently an algorithm successfully locates a page in memory without necessitating a fetch from the secondary storage. Contrarily, a lower rate signifies subpar performance, considering the lag when retrieving from secondary storage entities like hard drives compared to primary memory access.

Evaluating the page hit rate across diverse algorithms and fluctuating memory frame allocations, we aspired to extract conclusions about which algorithm proved most agile and proficient for disparate memory configurations and distinct memory traces.

Our methodical procedure ensures a panoramic view of each algorithm's relative pros and cons under varied circumstances, which substantiates a thorough assessment of their utility.

Results:

Our experiment leveraged four distinct traces, namely: swim, bzip, gcc, and sixpack. The actual memory usage of one trace is dependant on the setting of page size. Here we are setting the page size to 4KB with the page_offset = 12. Hence, when we increase the number of frames, we increase the page_table size. When the size is big enough to accomodate all the unique addresses from the trace, there will be no page fault and no disk write operation. In the context of page_size of 4KB, the unique addresses of the four traces are bzip = 317, gcc = 2852, sixpack = 3890, swim = 2543. So the actual memory usage of the four traces in this context are bzip = 317 X 4 KB = 1.238MB, gcc = 2852 X 4KB = 11.125MB, sixpack = 3890 X 4KB = 15.195MB, swim = 2543 X 4KB = 9.932MB.



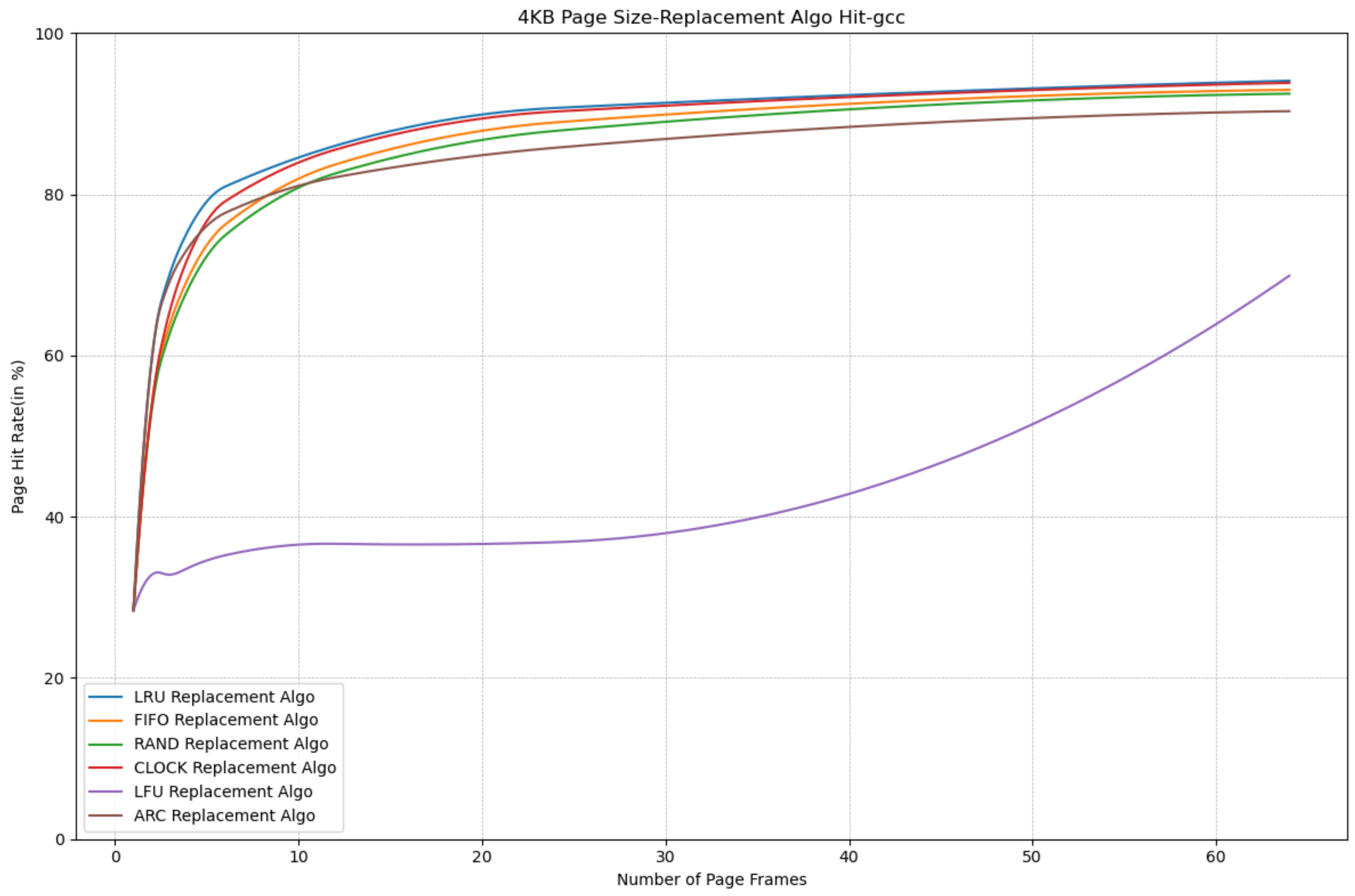
The above graph is generated in the context of page_size of 4KB, we are measuring the performance of six paging replacement algorithms. The metrics here is the hit rate. The analysis is based on **bzip** trace.

From the graph, one interesting observation is that the LFU is almost the worst replacement algorithm for the bzip trace. Even if the number of frame is constantly increasing, the hit rate remains almost unchanged(A bit increase).

The rest of the replacement algorithms have a similar trend, they all have a "sharp" turning point. Before this turning point, with the increase of number of frames, the hit rate is increasing dramatically. This simply simulate the **shortage of memory** case. In this context, the shortage of memory is the dominant factor, so with the increase of page frames, the page table size is increasing, the hit rate is increasing in a high rate. However, after the turning point, with the increase of page frames, the hit rate is although increasing but at a much slower rate than before. This means, at this stage, due to the marginal benefit effect, the size of the page table is not the single dominant factor. This simulates the **Excess Memory** case. The **Perfect Memory** case is extremely hard to simulate. Because in real time, we can't make sure to allocate the exact size of memory needed for each program. But in this experiment, around the turning point would be a better condidate of perfect memory.

When having a low number of page frames, LRU is performing best regarding the hit rate. Following LRU is ARC. Although ARC's hit rate is lower than LRU with lower frame number, but its hit rate is increasing at a very high rate. At the point of page frame number 7 and 8, the ARC has identical hit rate as LRU and ARC is the first to achive a almost 100% hit rate in the graph at the frame number around 11.

Only with the frame number reaching around 16, almost all the replacement algorithms has reached a 100% hit rate, which means at this stage, there are enough memory to accomodate almost all the unique addresses in the bzip trace.



The above graph is also generated in the context of page_size of 4KB, we are measuring the performance of six paging replacement algorithms. The metrics here is the hit rate. The analysis is based on the **gcc** trace.

A similar observation is that LFU's performance is the worst, but an interesting thing is that with the increase of number of page frames, the hit rate increased from 25% to almost 70% unlike the bzip trace. This means in the gcc trace, there may be multiple accesses for some programs because for LFU, we simply count the number of occurrence of unique addresses, and solely based on the occurrence number to evit page when the memory is full.

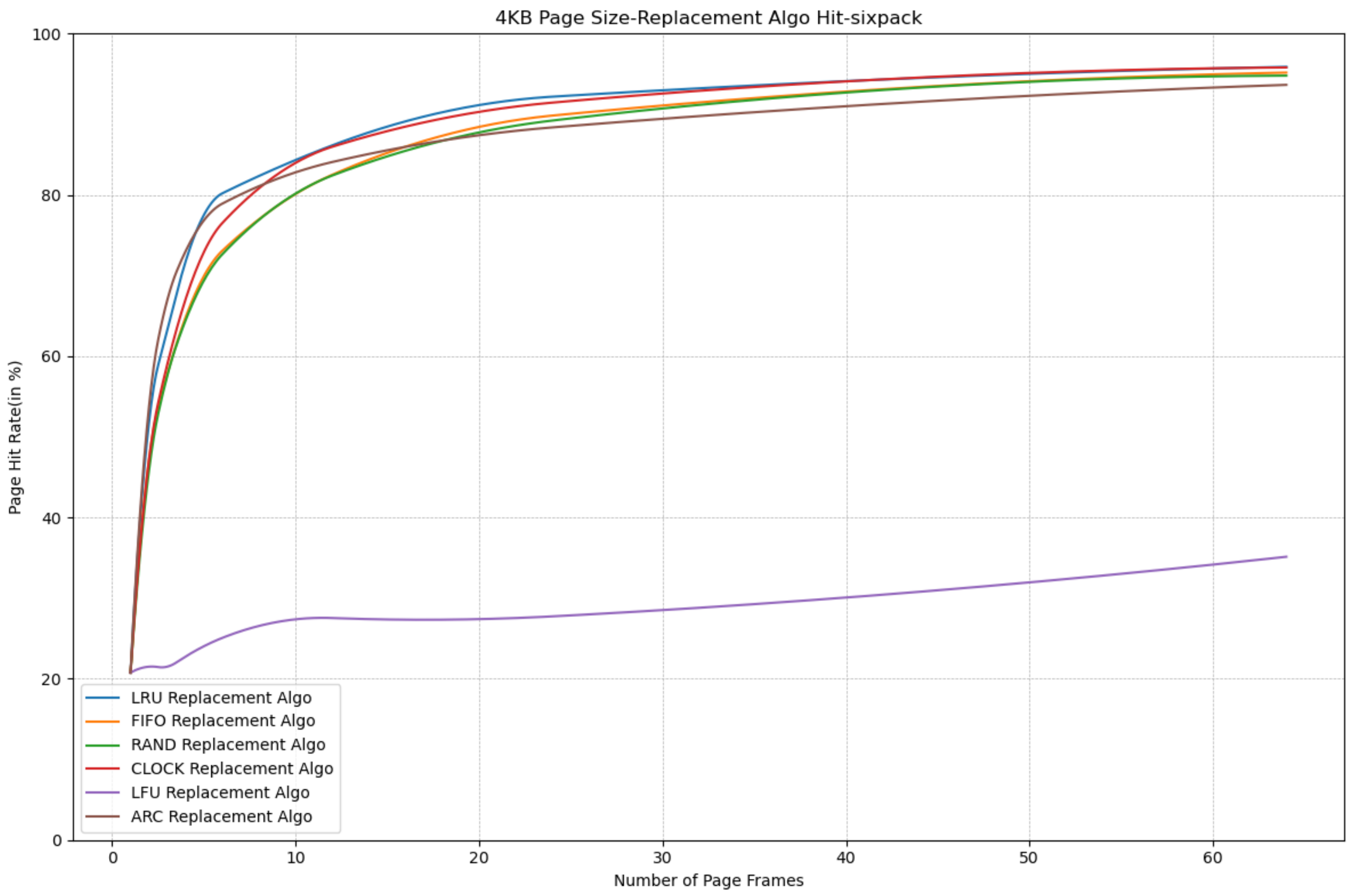
The rest of the algorithms are following a very similar trend. Like the bzip trace, all the hit rate trend have a turning point at around number of frame 5. Before the turning point, there is a **Shortage of memory** ,hence, the overall hit rate is very low starting from around 27%. But at this stage, the memory size is the dominant factor so with every page frame increased, the hit rate is increasing accordingly in a high rate. But after the turning point, it's obvious that the hit rate is increasing at a much slower rate with the increase of page frames. Most importantly, after around 23 page frames point, the hit rate almost remain stable although the number of page frames is still increasing at the same rate.

Another interesting observation of the gcc trace is that although we are offering larger number of page frames, at this stage, the memory still cannot accomodate all the unique addresses in the gcc trace because the hit rate is at around 95%. This means the memory is not the sole dominant factor in this senerio. Even if we are increasing the number of frames dramatically, we are not receiving a good increase in the hit rate.

The graph is not simulating the **Excess Memory** case. We use our simulator to test the threshold of the gcc trace. In the context of 4KB page size, when we increase our number of frames to 2852, we got a page hit at around 100% and we got no disk writes. And from this point forward, there is no point to increase the number of frames any more since there is already excess memory to accomodate all the unique addresses in the gcc trace.

Regarding the performance, LRU is still the best but the difference between LRU and the rest algorithms are negligible. From number of page frames 1 to 5, ARC performs almost identical as LRU, After around 10 frames, CLOCK is performing almost identical to LRU with less than 1% difference. So in this case, the CLOCK replacement algorithm would be more preferable since the overhead to implement the CLOCK algorithm is much more accepatable than LRU and ARC.

Apart from the above discussion, the FIFO and RAND are working in a really good manner. Intuitively, if we only take into account the access order(FIFO) or we randomly evit a page(RAND) when the memory is full are not reliable policies, but the result turn out to be very good.

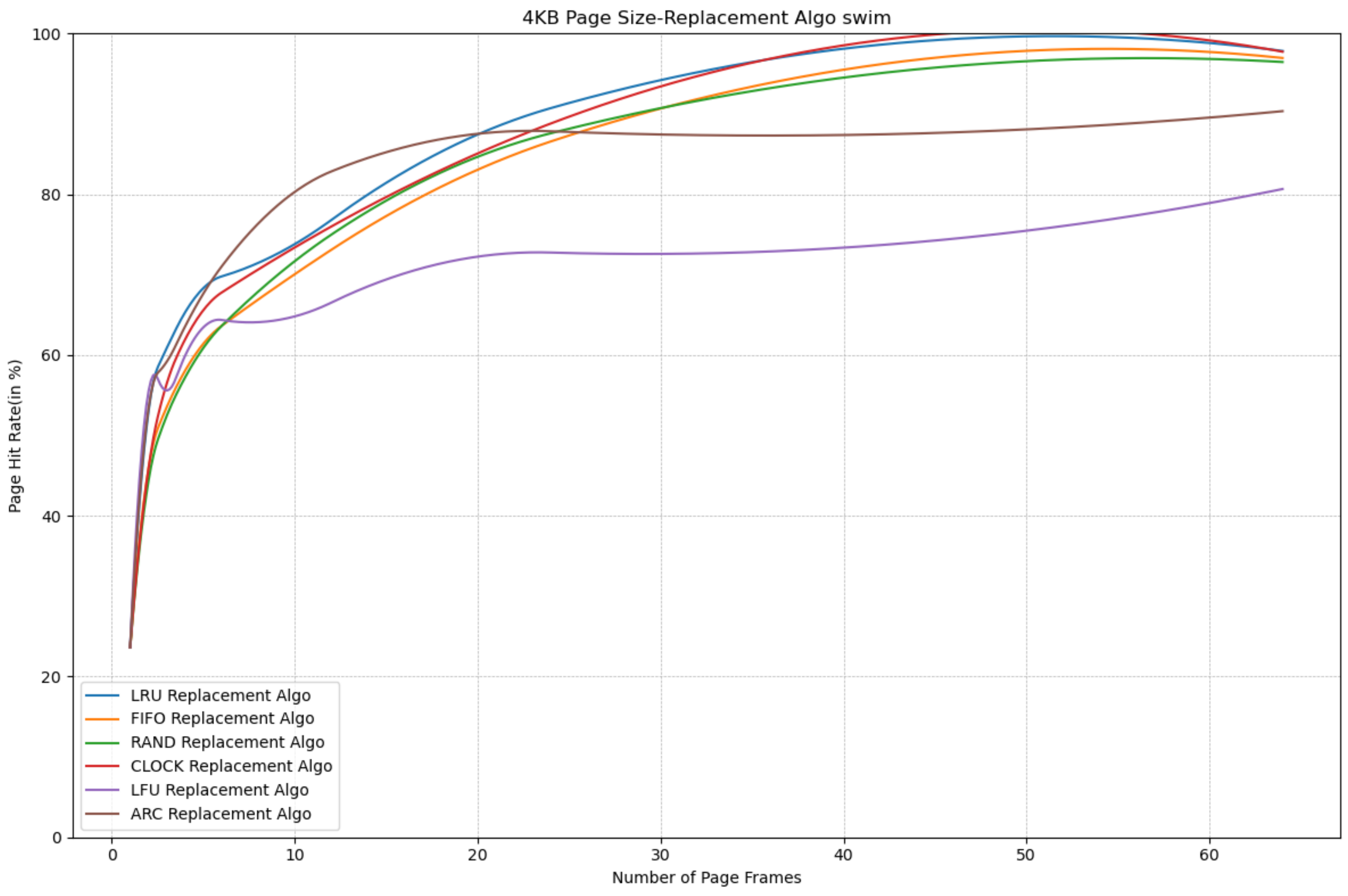


The above graph is generated in the context of page_size of 4KB, we are measuring the performance of six paging replacement algorithms. The metrics here is the hit rate. The analysis is based on the **sixpack** trace.

The first interesting observation is that LFU is doing so poorly with the hit rate around 20%. Even with the increase of number of page frames, the hit rate almost remain steady with a tiny increase.

The rest replacement algorithms are all following similar trend. They all have a turning point. Before this point, the number of frames is the dominant factor. It's a simulation of **Shortage of Memory** case. At this stage, the curve is very steep meaning a slight increase in the number of frames can lead to huge increase in the hit rate. After the turning point, the curve is getting flatter, meaning the number of the frames is not the solely dominant factor for the hit rate.

When the number of frames is low (specifically below 5), ARC has the highest hit rate. With the continuing increasing of the number of frames, LRU has a better hit rate, and following that is CLOCK. The performance of CLOCK and LRU is almost identical.



The above graph is generated in the context of page_size of 4KB, we are measuring the performance of six paging replacement algorithms. The metrics here is the hit rate. The analysis is based on the **swim** trace.

The LFU is behaving extremely well in the swim trace. Especially with the number of frames below 4 or 5, it's the best one among all the replacement algorithms. Even afterwards, with the continuous increase of the number of frames, the hit rate using LFU is increasing obviously. With more frames, the hit rate increase to around 81% which is by far the highest among all the four traces using LFU.

When in the context of low frame number, ARC works almost the best among all the algorithms. Most noticebly, in the range of 5 to 20 number of frames, The hit rate of ARC is exceptionally better than the rest of the algorithms. Although after 20, with the continuous increase of frames, the hit rate almost remain the same, but it stopped at around 90% which is totally acceptable. There are two sharp turning points for ARC at around 10 and 20 frames. Before 10 frames, the number of page frams is the dominant factor, which simulate the **Shortage of Memory** case. After 20, the hit rate almost remain the same even with the increase of number of frames, this simulates the **Excess Memory** case.

Besides LFU and ARC, the rest of the replacement algorithms are following the similar trend. However, for the swim trace, all the algorithms don't have a very obvious turning point like the previous three traces. Although the increase rate of hit rate is decreasing, but the decrease rate is very smooth.

After 20 number of frames, LRU has the best hit rate performance, and CLOCK is very close to LRU. After around 35 number of frames, CLOCK has the best hit rate but still very close to the preformance of LRU.

The most interesting ovservation is that the swim trace may be highly patterned. Because the ARC adopts a hybrid way which integrate both LRU and LFU. In this specific trace, there should be a lot of repeating accesses in a short period of time, hence the LFU is working surprisingly well, which lead to the boosting increase in the hit rate of ARC.

Conclusions:

Our study illuminates the intricate dynamics of various page replacement algorithms in the context of specific memory traces.

The primary observations are:

Performance Variances: An algorithm that excels in one trace may not necessarily shine in another. For instance, while LFU struggles with bzip and gcc, its efficiency markedly improves in the swim trace. This underscores the necessity of contextualizing the efficiency of a page replacement algorithm within the specifics of a given application or memory trace.

LRU's Consistency: LRU consistently delivered robust performance across all traces. However, its complexity might not always be justified, especially when simpler algorithms like CLOCK achieve comparable results.

Adaptive Nature of ARC: ARC exhibited an adaptive behavior, outperforming other algorithms in specific regions, indicating its capacity to adjust to different workload patterns.

Inefficacy of Blindly Increasing Memory: After a certain threshold, merely increasing the memory doesn't significantly enhance the page hit rate. This demarcates the boundary between optimal memory allocation and excessiveness.

In essence, there's no one-size-fits-all page replacement algorithm. The right choice hinges on the specific requirements of the memory trace in question, computational overheads one is willing to tolerate, and the desired balance between efficiency and memory utilization.