**Operating Systems Practical 2 Report**

**Virtual Memory Simulation**

**Introduction**

In an operating system (OS), the base of all modern computers, memory is an integral part to it functioning as we know it by allowing data to be accessed and stored for later use. However, there is no feasible way to obtain unlimited memory space so it must be managed correctly so no memory is lost or interacted with unintentionally. One such scheme of memory management is paging, which involves splitting up the physical and logical spaces in memory into identical fixed sized blocks, called page frames and pages, respectively.

When any computer process requests access to any memory, this is then handled by the page table in the memory management unit (MMU), where 1 or more page frames are allocated to the process in the page table and then links these frames to the same number of logical pages. This allows for each process to access repeated memory easily via the page tables connections from logical pages to physical page frames in memory. However, when the number of unique entries being made into the page table exceeds the number of frames allocated to the table, new entries cannot be easily added, and other entries need to be replaced to make room for the new entries.[1] (GeeksForGeeks, 2023)

The policies which govern what entries should be replaced are known as replacement policies. In this report, 3 different replacement policies will be investigated: rand, lru and clock. Rand replaces a random page in the page table with the new entry. Lru replaces the oldest accessed page with the new entry. Clock is an approximate version of lru which aims to have the same outcome but with less overhead. These 3 replacement policies will be tested on 4 different trace files based on 4 processes, gcc, swim, bzip and sixpack. Each trace file is a set of 1 million instructions which maps a set of executions that could be done by each of the 4 processes from when they start running.

The main metric that will be used to see the performance of the 3 policies is the rate of page faults that occur compared to the total executions ran on a different number of frames available for the page table. A page fault occurs when the page that is trying to be accessed is not in the page table, so it must be added as a new entry. This causes minor slowdown to the OS and can cause serious performance issues if a large amount of page faults occurs due to the constant swapping of data. So, a good replacement policy should minimise the amount of page faults that occur to maintain high performance for the OS. However, when the number of frames is reduced, more page faults will occur due to the limit pages available, which may show that certain replacement policies are better than others when the number of frames is lowered, thus making it a good metric to compare and test the performance of the 3 replacement policies.

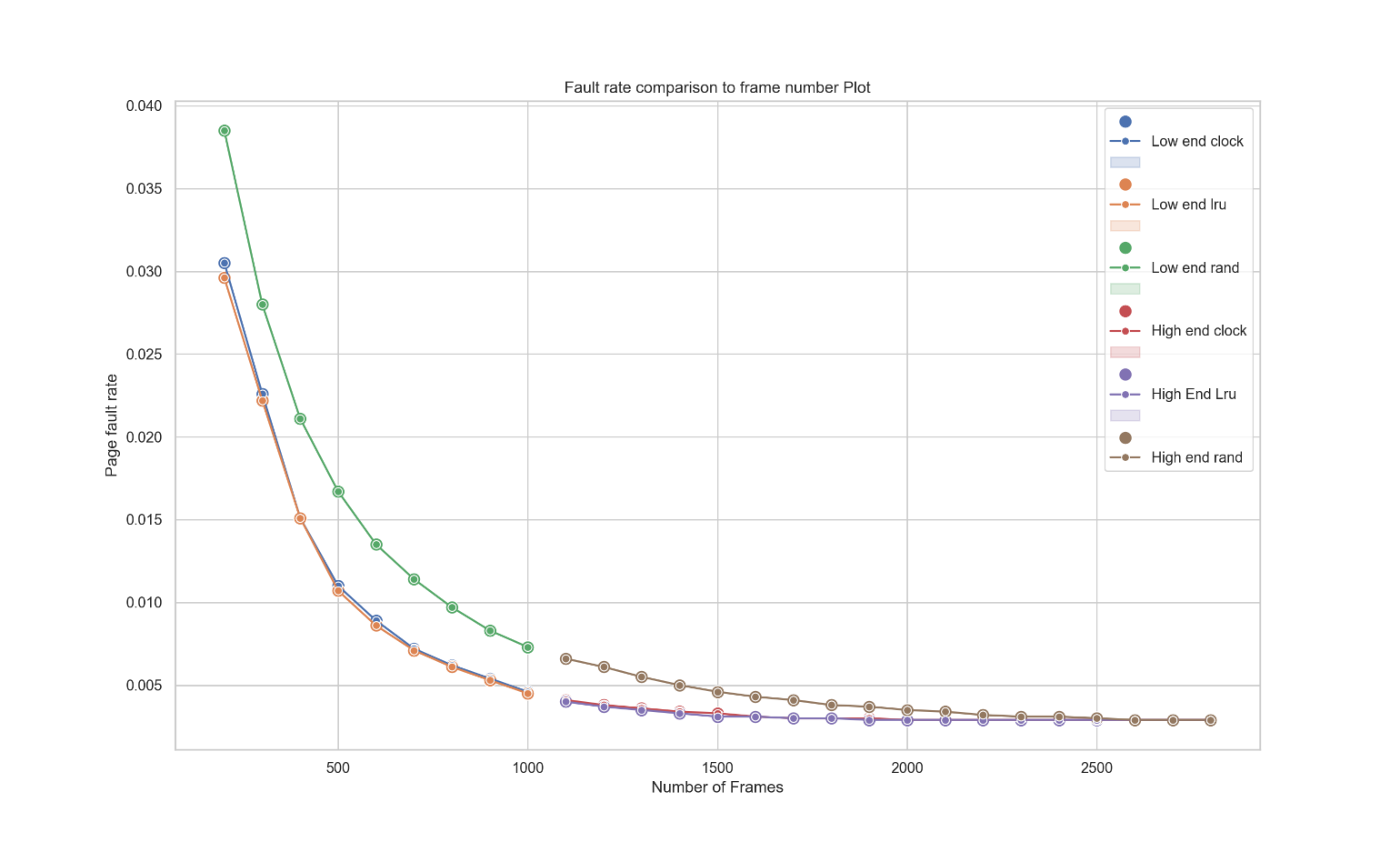
The goal of this investigation is to learn more about each of these trace files and about each of the replacement policies, including in which situations each policy performs best and the major difference and use cases between each policy.

**Methods**

Our approach for running the simulations was to first run the sim file with each page replacement policy on frame values ranging from 1000 to 5000 with steps of 1000 or 500 depending on how concentrated we need to go (i.e. steps of a 1000 means running on the values 1000, 2000, 3000, etc.). We considered this a “high end” run, in that it covers a range of high frame numbers for the purpose of determining how much memory each traced program actually needs for each replacement policy, and how the policies compare to each other with a high number of frames. We assume that where the page fault rates plateau is the memory need for that replacement policy, as the only cause of page faults at this point are from compulsory cold-start page faults. Following this, we ran the policies again on frame values ranging from 100 to 1000 with steps of 100, dubbing this the “low end” run, with for the purpose of determining which page replacement algorithm works best when the number of frames is low. The results from both the “high end” and “low end” runs are displayed side by side on the same scatterplot, resulting in one graph for each of the trace programs, except for the case of bzip a second more-concentrated graph was needed with a smaller step gap to get a better view.

**Results**

GCC

Frame Number on Page Fault Rate (GCC Trace)

The graph above shows the relationship between available frames and the fault rate for replacement policies of lru, clock and rand when run on the gcc trace program. For all three replacement polices, as the number of available frames increased, the page fault rate decreased. With a lower number of frames this decrease was much more significant in size due to page faults being much more frequent. In terms of how the policies compare to each other, rand was always worse than the others in terms of page faults, while clock and lru performed similarly but with clock ultimately always being outperformed by lru, even if just by a little. The amount of memory they need can be seen by where the curve flattens for each of them, it was around 1900 frames for lru, 2000 for clock, and 2600 for rand.

SIXPACK

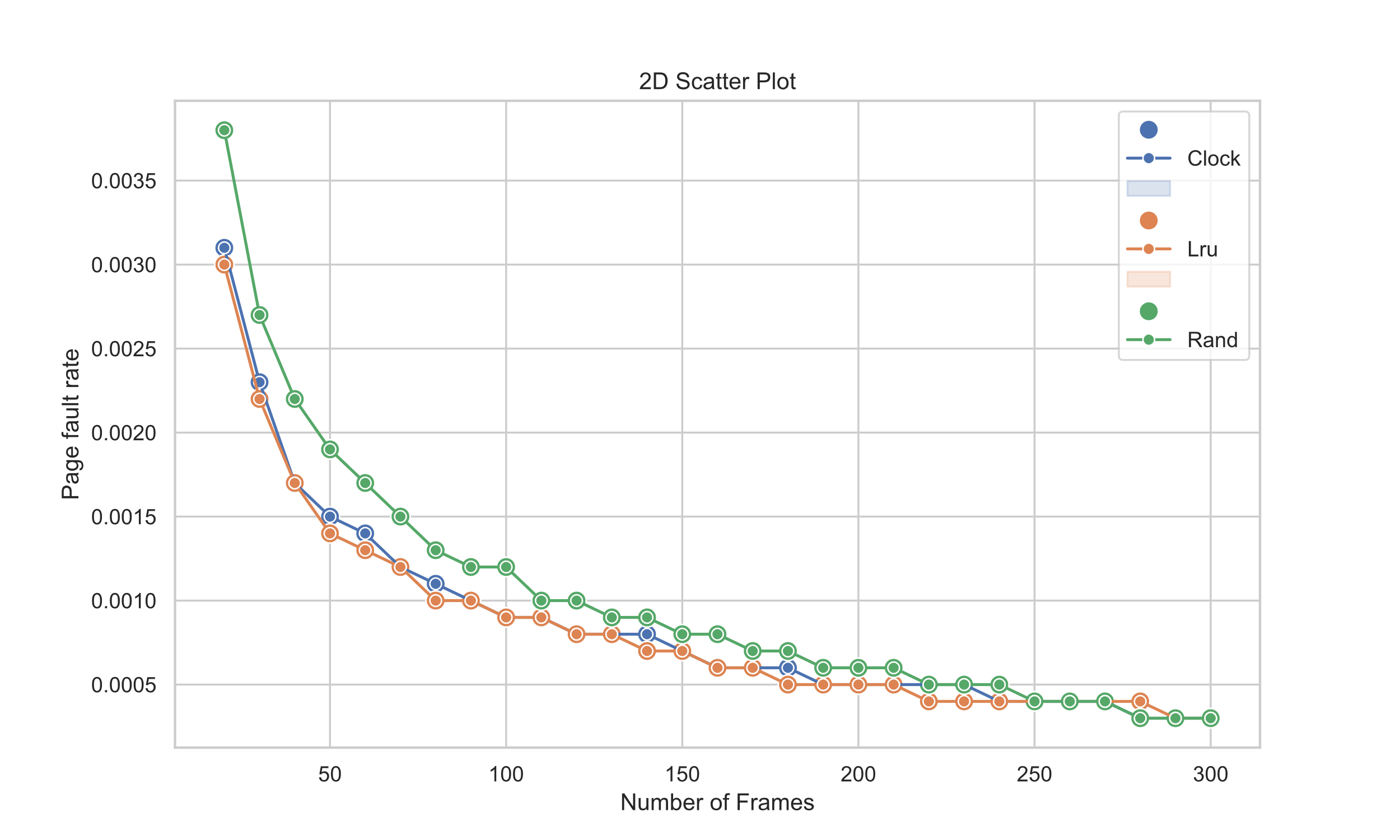
A graph with different colored lines

Description automatically generatedFrame Number on Page Fault Rate (SIXPACK Trace)

At the low end, our graph displays a steep decrease for each policy, and it seems clear that with low frames lru performs marginally better than clock and significantly better than rand. This is evident in the fact that lru's fault rate curve is lower than the two other policies. The page fault rate curve flattens between 1000 to 2000. From this flattening of the curve, we can deduce that the memory need for sixpack is between 1000 and 2000 for lru and clock, and around 4000 for rand.

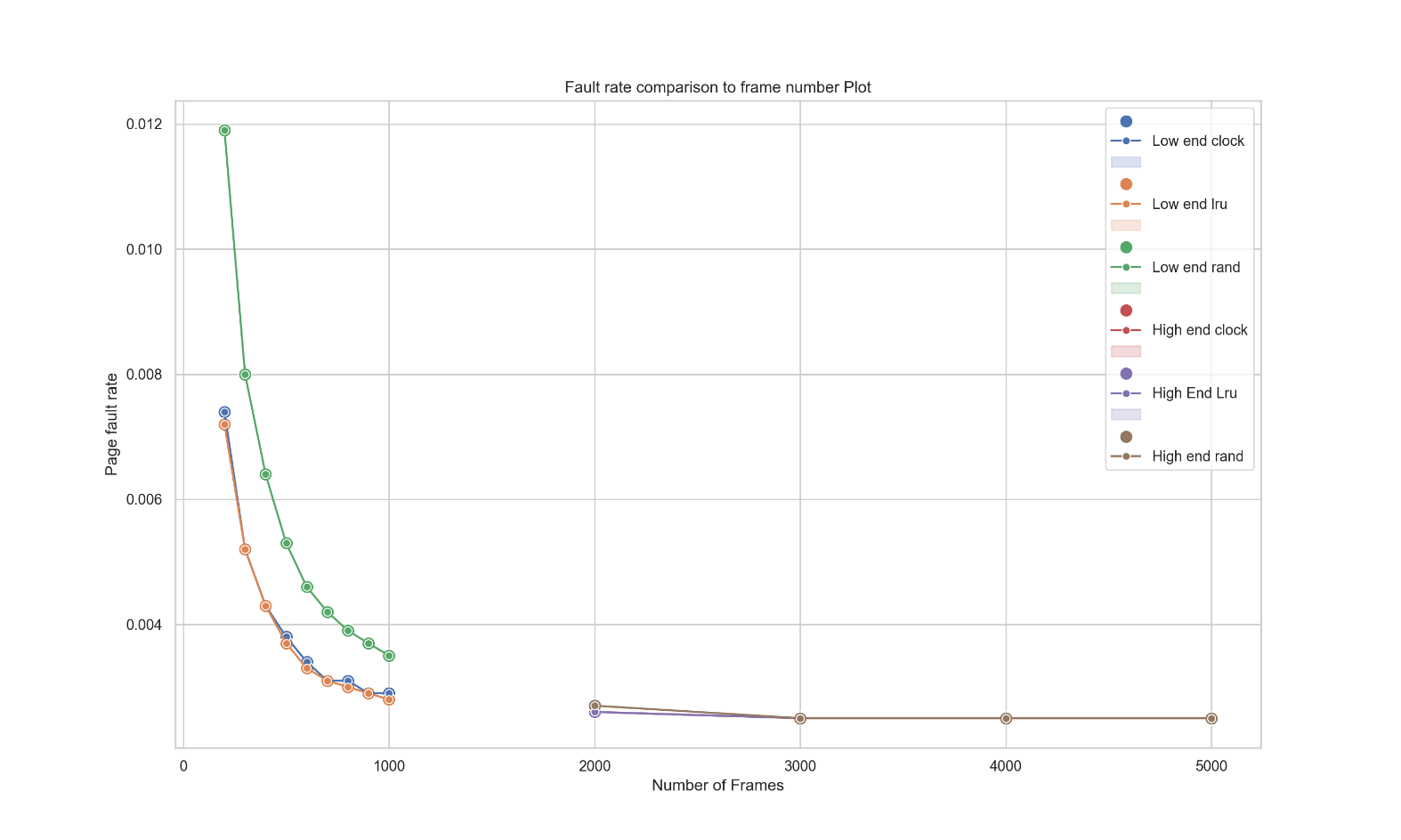
A graph with numbers and lines

Description automatically generated with medium confidenceBZIP  
Frame Number on Page Fault Rate (BZIP Trace)

 Frame Number on Page Fault Rate (BZIP Trace, Concentrated Graph)

The page fault rate curve for this application is very steep in the initial graph. The graph seems to flatten somewhere between 200 and 300 frames, so we can deduce that the memory need for bzip falls between this range. This is quite low compared to the other applications simulated in this report. The lru and clock replacement policies seem to have a similar efficiency with bzip at low frame rates, as indicated by their very similar page fault rate curve. The rand policy seems to perform worse, but the extent is unclear due to the lack of data points, so it was decided that an additional figure would be created for bzip to determine the performance of the different policies more-clearly. The additional simulation data produced shows that the original hypothesis holds true, the figure shows clearly that both lru and clock outperform rand. From this second graph we can also now see that lru seems to perform marginally better on certain frame points with clock never exceeding lru in performance at any point in the curve. It’s worth noting that there was actually a single point where rand performed better than lru at 280 frames, this shows that on rare occasion, due to its random nature, rand may get lucky and actually outperform even lru. It makes sense that it happened when rand had almost reached its full memory needs because this reduces the amount of times it needs to get lucky with the page it chooses to replace.

SWIM

Frame Number on Page Fault Rate (SWIM Trace)

In the above graph for the swim trace, the page fault rate steeply declines early, plateauing out at close to 0% for all policies past 3000 frames. Out of the 3 policies, rand is by far the worst in both the high and low end sections tested, with it being significantly higher at all frames on the low end and is the only policy that isn’t plateauing by 2000 frames on the high end section. Both clock and lru have similar results in both sections, with both being very close at all times that were tested, however, lru is slightly better in most sections of the low end section, with clock only being equal to lru at 3 points and never being better than lru. So, for the swim trace, both clock and lru are fine policies to choose, with lru only being slightly better if its viable to run, with the only policy that shouldn’t be used under normal circumstances is rand due to its poor performance.

**Conclusion**

To conclude, we set out to compare which page replacement policy was the best out of lru, clock, and rand, using page fault rate as our metric which we recorded for varying numbers of frames and with four different trace files, each file containing 1,000,000 events. Ranges of both low and high frame numbers were considered for each trace program in order to determine which page replacement policy work best with limited physical memory and how much memory each trace actually needs, respectively, with the use of different trace files also helping to determine if one policy works best in all situations.

It was found that across all trace files, rand was almost always the worst in terms of page fault rate with both higher and lower frames. Meanwhile, clock and lru were fairly close-together in terms of page fault rate, even with low frame numbers. However, in the data collected, clock was not seen to have a lower fault rate than lru at any point, and its fault rate was often above lru, even if by only a little. This almost makes lru the clear winner in terms of fault rate for the situations that were looked at in this investigation, except for the one case at 280 frames for bzip where rand was better. At first it may seem that the frame numbers didn’t play a role here and that rand just chose the correct page each time, but on closer inspection, since this was so close to the memory needs for bzip, there wasn’t as much need to replace pages, meaning that rand didn’t need to get absurdly lucky as many times. This also reveals a limitation in our investigation in that we didn’t run the simulator many times with frame values that were really close to the memory needs of the trace but without actually meeting them, thus giving rand more of a chance to get lucky.

While fault rate isn’t the only factor in what makes a good page replacement policy, it is generally a good indicator of runtime as each page fault significantly delays the time a program completes in, and so lru is generally faster than the other policies examined. However, whether there is one replacement policy that unanimously works best in all situations is actually left unclear, as it is possible that rand would actually beat out clock or lru in repeated runs where the frame number is only, say, 1 or 2 away from the program’s memory needs.

In terms of how much memory each trace program needs, taking the flattening of the graph curve as an indicator, for sixpack and swim the only replacement policy that needed more than 2000 frames was rand, needing around 3000 frames for swim and 4000 for sixpack. Interestingly, though lru and clock needed around 2000 frames for gcc as well, rand only needed 2600. Finally, gzip had a very small memory need of around 300 frames for all replacement policies, supporting the idea that rand in general isn’t as bad when working with programs with smaller memory needs.

**References**

1. GeeksforGeeks. (2023). *Paging in Operating System*. [Online]. GeeksforGeeks. Last Updated: 13 July 2023. Available at: https://www.geeksforgeeks.org/paging-in-operating-system/ [Accessed 15 September 2023].