**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.

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.

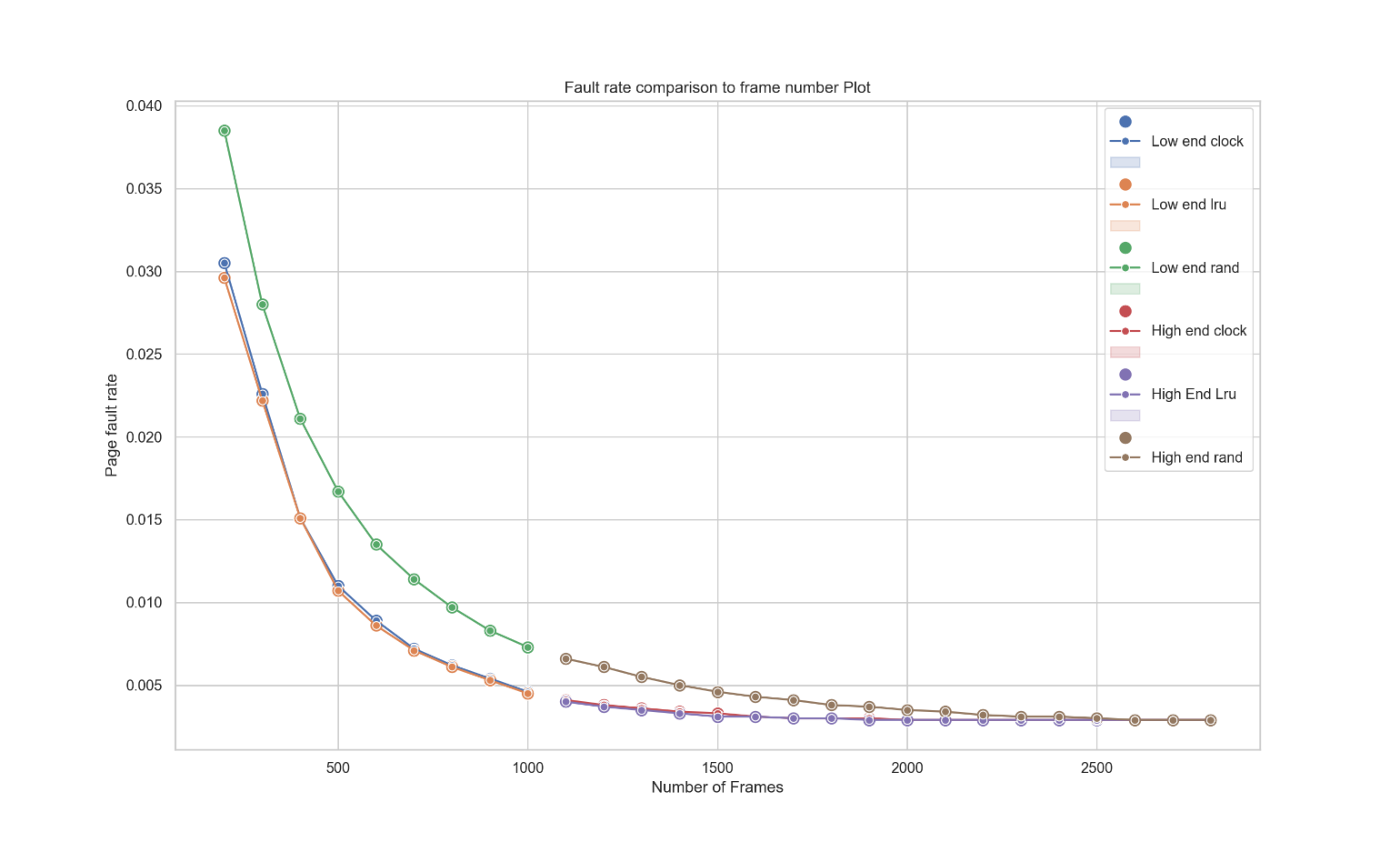
The goal of this investigation is to learn more about each of these trace files and about each of the replacement policies, including in which situations each policy performs best and the major difference and use cases between each policy.

**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

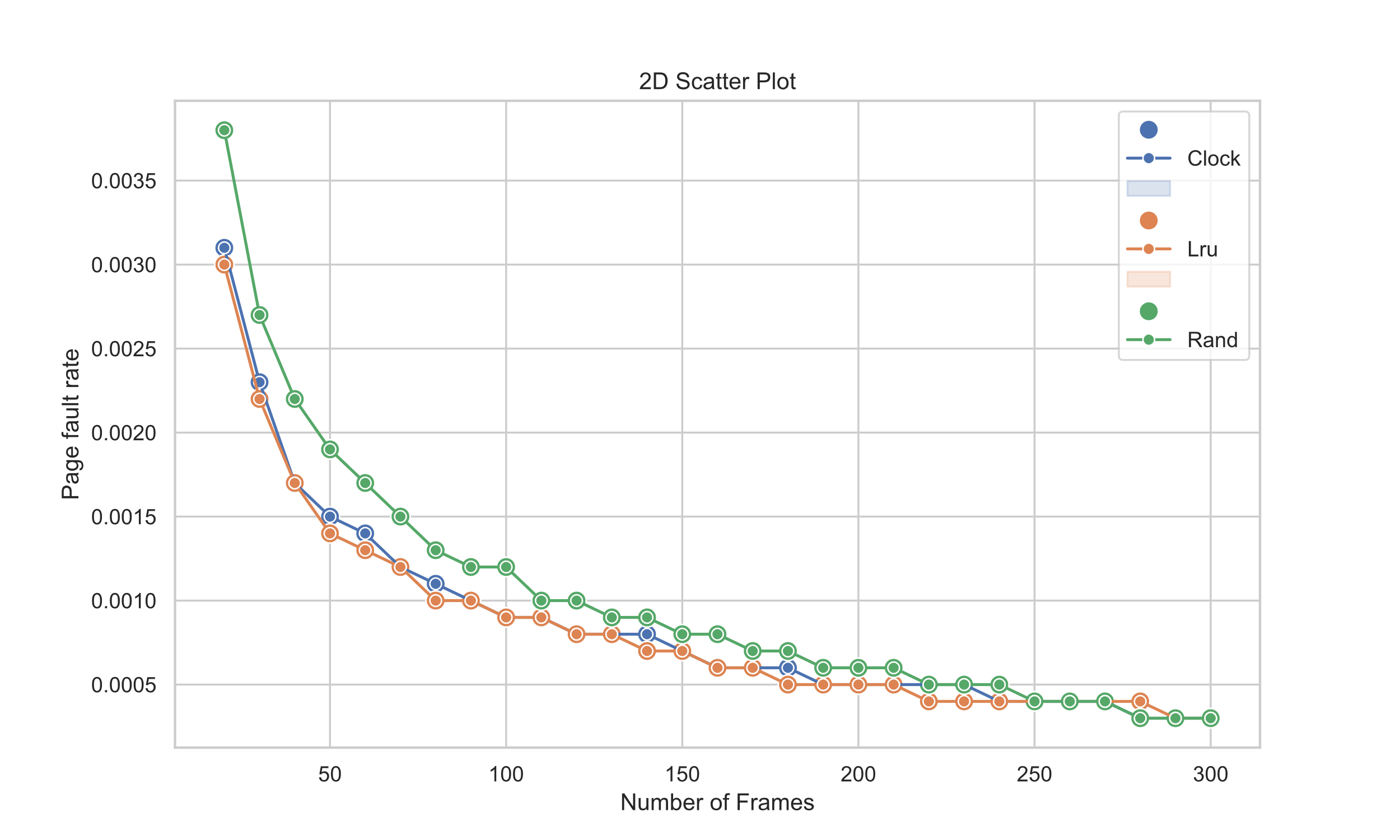
A graph with different colored lines

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.

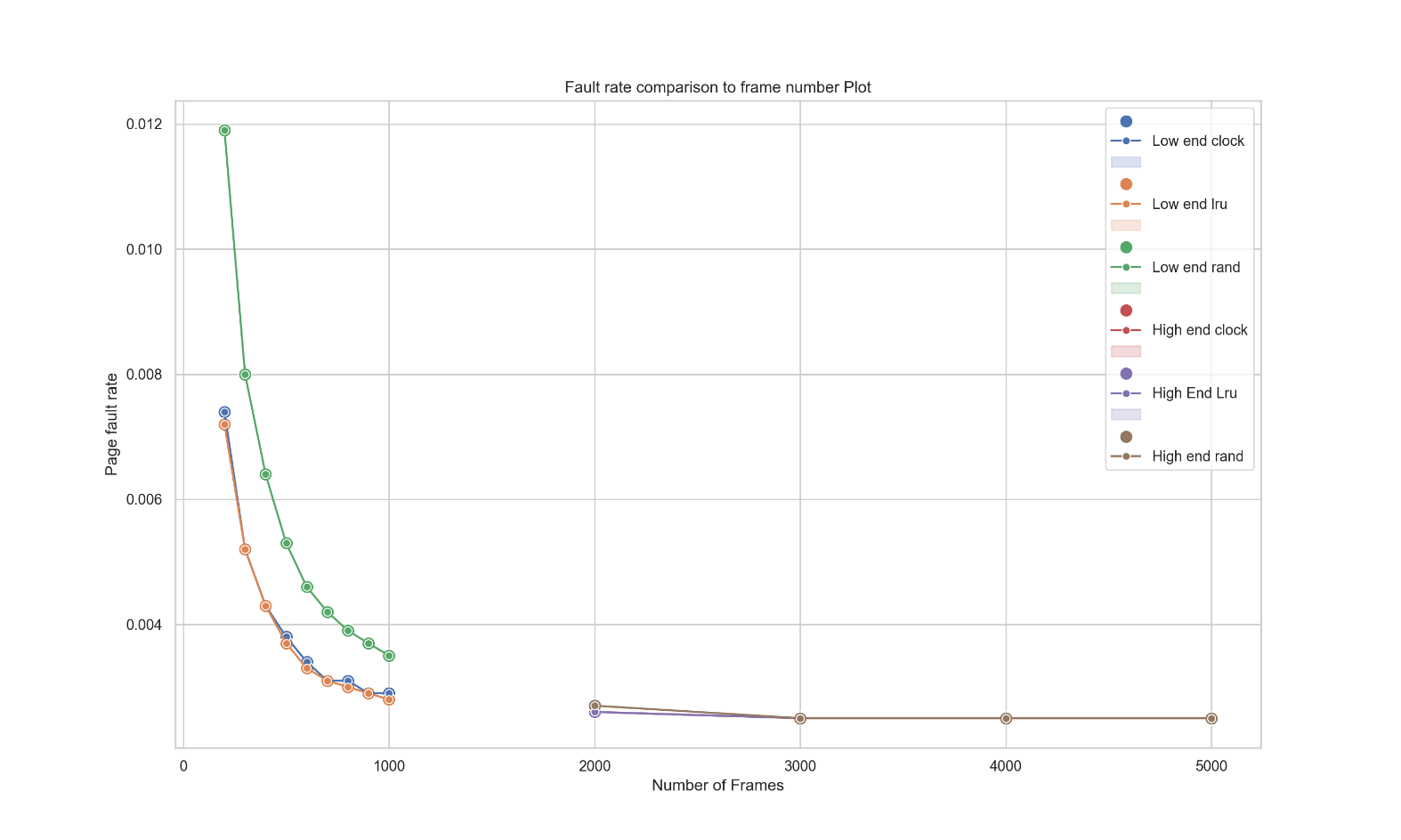
A graph with numbers and lines

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.

SWIM

Frame Number on Page Fault Rate (SWIM Trace)

**Conclusion**

[Brief conclusion summarizing observations across all trace files, including things they all shared, differences, etc.]