Report
Project 05
Operating Systems
netIDs: amceache, cbury, kingram1

- In your own words, briefly explain the purpose of the experiments and the experimental setup. Be sure to clearly state on which machine you ran the experiments, and exactly what your command line arguments were, so that we can reproduce your work in case of any confusion.
- Very carefully describe the custom page replacement algorithm that you have invented. Make sure to give enough detail that someone else could reproduce your algorithm, even without your code.
- Measure and graph the number of page faults, disk reads, and disk writes for each program and each page replacement algorithm using 100 pages and a varying number of frames between **3 and 100**. Spend some time to make sure that your graphs are nicely laid out, correctly labelled, and easy to read.
- Explain the nature of the results. If one algorithm performs better than another under certain conditions, then point that out, explain the conditions, and explain *why* it performs better.

Purpose of Experiments & Experimental Setup

The overall purpose of this project was essentially to get our first look at coding underlying components of an operating system. In this project we gained a better understanding of mapping between virtual addresses and physical addresses and what factors to consider when choosing a replacement/eviction policy for page tables. In designing our own replacement algorithm, we first had to research other existing algorithms to see if we could modify any to fit our project. As we have no way to tell when a page is accessed or at what time a page was accessed, strategies employed in policies like LRU (which looks at how recently pages were accessed) or Second-chance (which looks at whether a page has been accessed since it was last considered for eviction) were not applicable to this project.

We ran our experiments for this project on student00.

To run our experiments, we used command lines of the format:

```
./virtmem [PAGES] [FRAMES] [rand|fifo|custom] [sort|scan|focus]
```

Page Replacement Algorithms

- **Random (Rand)**

The first algorithm we implemented was a random eviction policy. When we determined that we needed to evict a page from the page table, we took a random number with rand() from 0 to the number of frames and simply evicted that random number entry in the page table.

- **First-In, First-Out (FIFO)**

The next algorithm was First-In First-Out. To implement this, we simply kept a deque in our main program - we pushed the page number into the back of the deque when adding or

replacing a page in the page table, and popped page numbers from the front to determine which to evict when needed.

- **Custom**

Finally, we implemented a custom eviction policy that's sort of like a modified version of FIFO, except we evaluate first based on how frequently pages have been evicted. We chose this strategy primarily because we are unable to see how and when a page is accessed in the page table - meaning that we couldn't implement policies like LRU. In our custom policy, we perform FIFO as usual by pushing page numbers to the back of the deque. We additionally store the number of times a page has been evicted in a vector of integers. Instead of simply popping the first pushed page number when evicting, however, we iterate through the eviction count vector and select the page with the lowest number of evictions. In the case of a tie, the front-most, lowest-eviction-count page is selected.

The inspiration behind this algorithm was primarily FIFO but also Second-Chance. We have no way to actually tell whether a page has been accessed with the provided code, which is integral in implementing Second-Chance, so we "fudge" this by examining how frequently a page has been evicted instead. In theory, if a useful page is being evicted too often, this approach will balance how much time it's given in the page table.

Analysis

To perform analysis on the performance of different replacement strategies, we measured the number of disk reads, disk writes, and page faults for each combination of programs to test (sort, scan, or focus) and replacement policies (FIFO, Rand, or custom). In general, we found that FIFO performs slightly better than Rand for all metrics and for every program run. Custom performs either worse than FIFO or equivalently for all metrics.

Page faults occurred in our program both when a page was not available in the page table and when a page was missing required permissions. As such, in the case where every page can fit on the page table, we have 2*(the total number of pages) page faults. Disk reads occurred to load data into the table initially and also to read in data when a page was evicted, whereas data was only written to the disk after an eviction, so the disk read metric is always higher than the disk write metric for any given set of parameters.

For all the programs, we need to perform a disk read when there is a page that is not present. We need to read the page fro mthe disk in order to add it and that happens the first time the page is read or if we are reading it from disk from an eviction. When there is a page fault we needed to determine whether there is room in the table or if the is not. If there is room, then we can just read from the disk and then place the page at the next open location in the table. If there is no room left, then we need to perform an eviction policy. When there is an eviction, we use pop off of the deque for FIFO because that would be the first item in the list. For random, we find a random spot in the table to evict and for the custom algorithm, we check the array of how many times a page in the front has been accessed and evict the lowest valued page.

We now consider the performances for each of the three programs to test (sort, scan, or focus). In the case of sort, FIFO performs best in the worst case scenarios. However, at around

75 frames to 100 pages, Rand works the best of the three algorithms for all three metrics; this is because at a large enough number of frames, there will be a significant number of less-frequently-accessed pages sitting in the page table, waiting for pages before them to be popped off. Rand ensures that these unused pages have a greater chance of being evicted - this leads to fewer page faults when a page is not found, and then fewer evictions from that point. The last entry (the rightmost one in every graph) has only page faults and disk reads because the page faults occur only because the table is initially empty, and the reads occur only to fill up the table. There are no disk writes because the page table is large enough to fit every possible page.

In the case of scan, our graphs (Figures 2, 4, and 6) are identical because this program gives the same result for the same number of pages passed as an argument. The results do not change based on the number of frames, but they do change if the As in our results for sort and scan, in the rightmost set of data points, there are no disk writes because the page table is large enough to fit every possible page.

For focus, the custom policy performed essentially the same as FIFO, but in this case they both performed ever so slightly worse than rand. Rand does especially well compared to the other two in the case where there are 75 frames, for the same reason as it performs better in that specific case in the focus program.

For our custom algorithm, we were able to find that it performs better in the range of 30 to 70 frames. With our implementation the speed of the program for custom is faster than FIFO or random.

Results and Figures

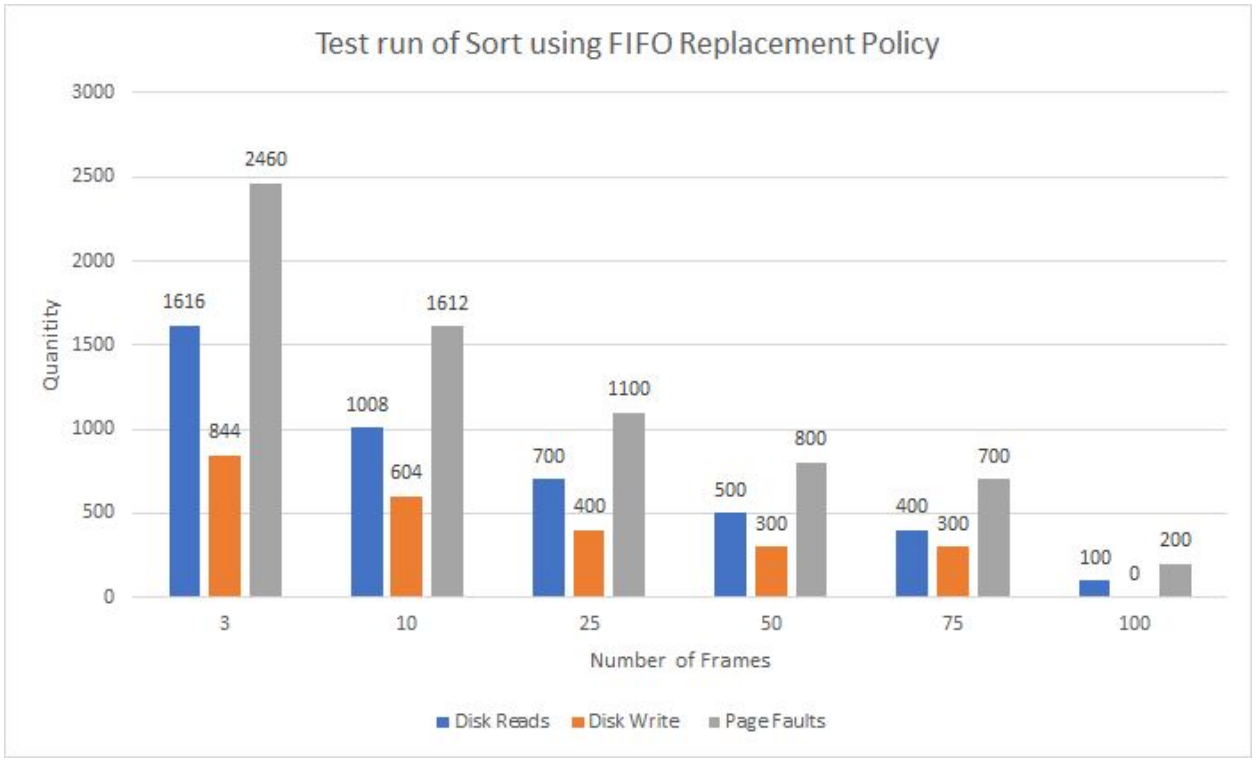

Figure 1: Test runs of the program “Sort” using the FIFO replacement policy, varying the number of frames.

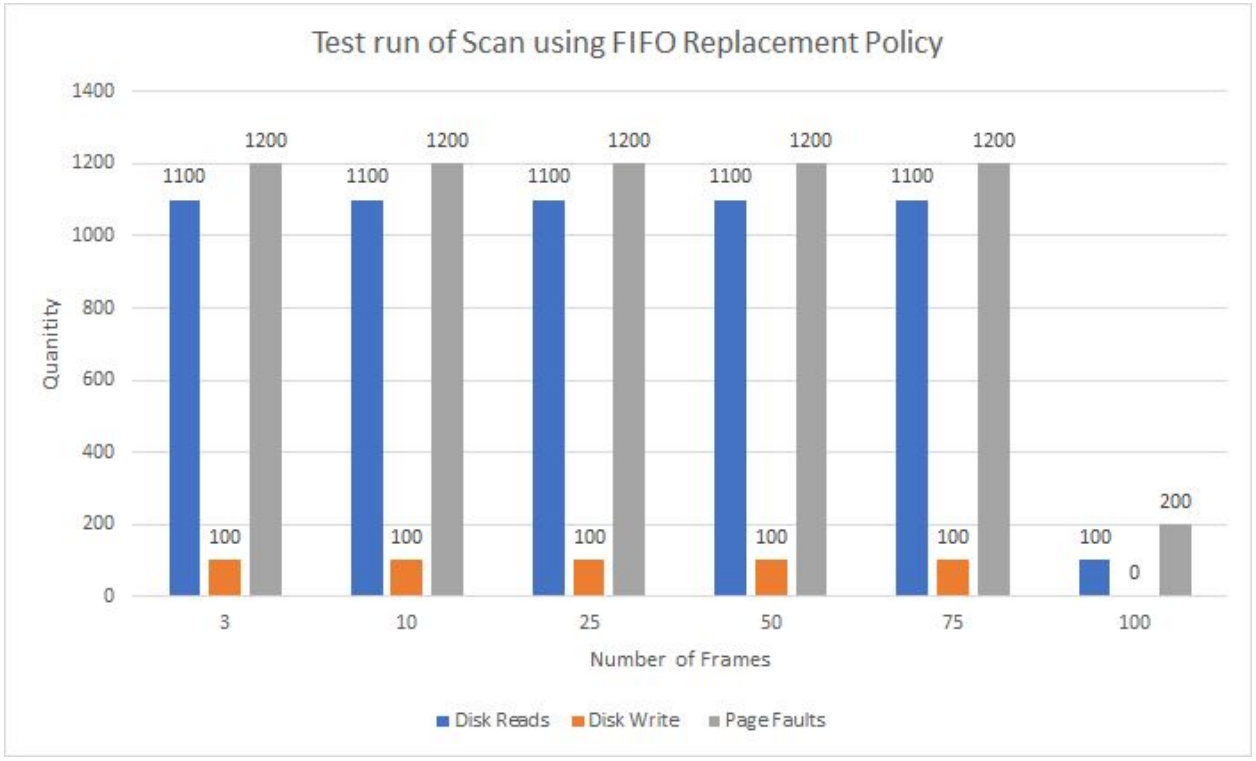

Figure 2: Test runs of the program “Scan” using the FIFO replacement policy, varying the number of frames.

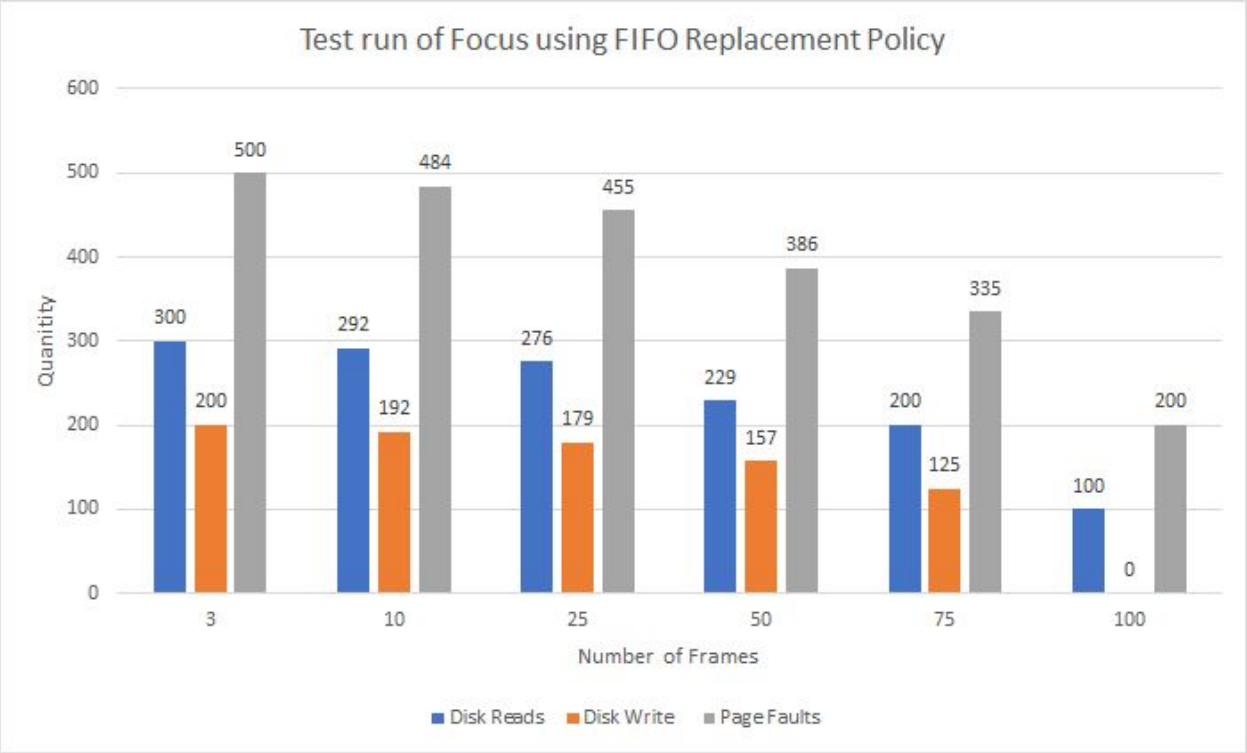

Figure 3: Test runs of the program “Focus” using the FIFO replacement policy, varying the number of frames.

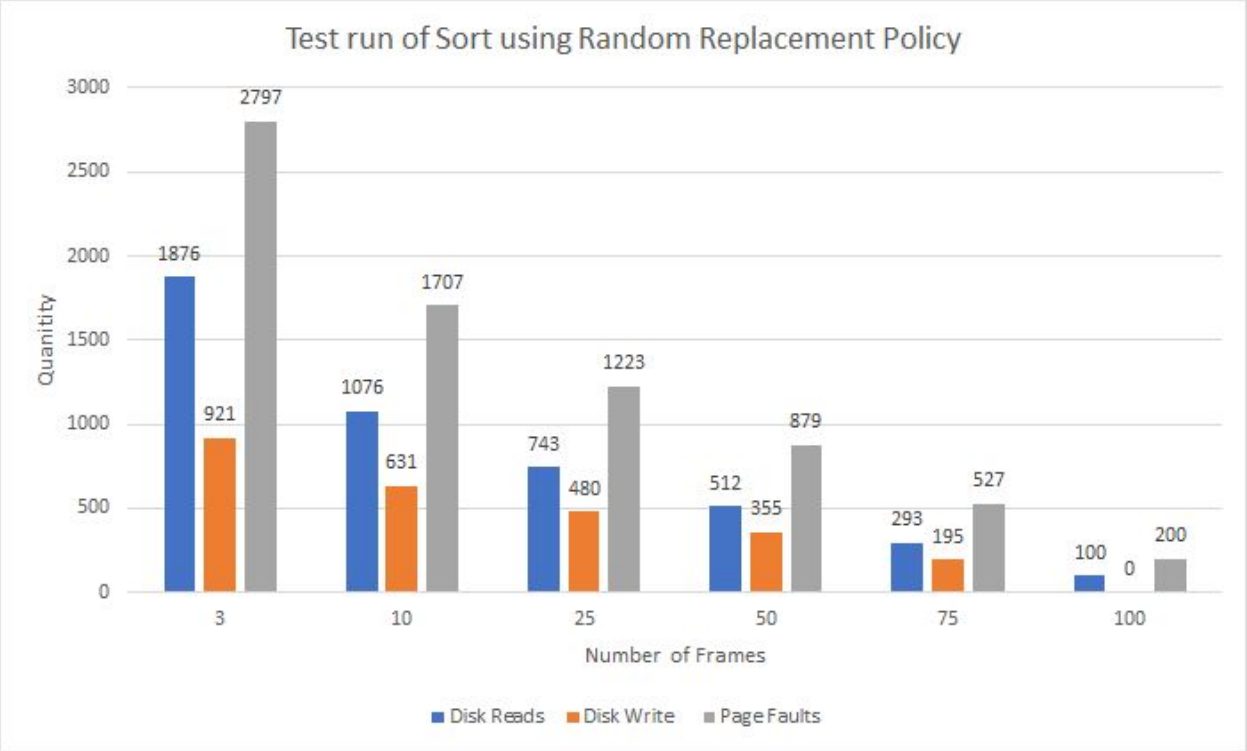

Figure 4: Test runs of the program “Sort” using the Random replacement policy, varying the number of frames.

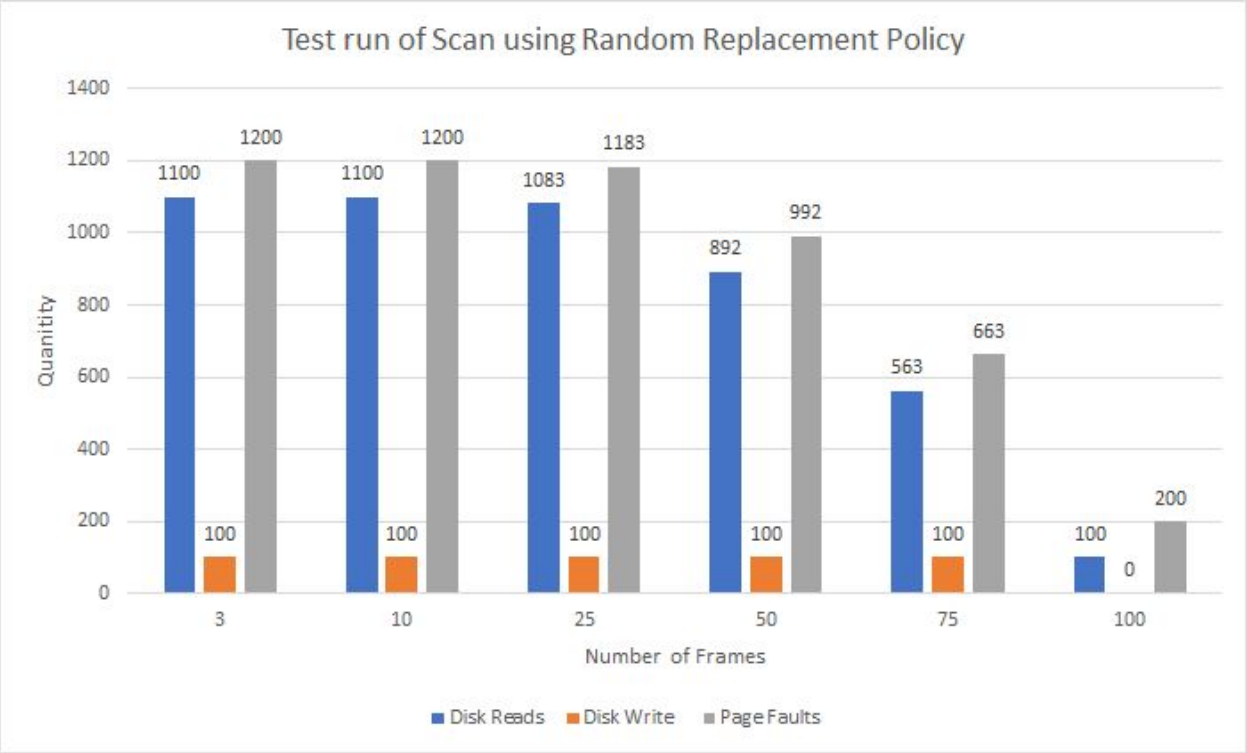

Figure 5: Test runs of the program “Scan” using the Random replacement policy, varying the number of frames.

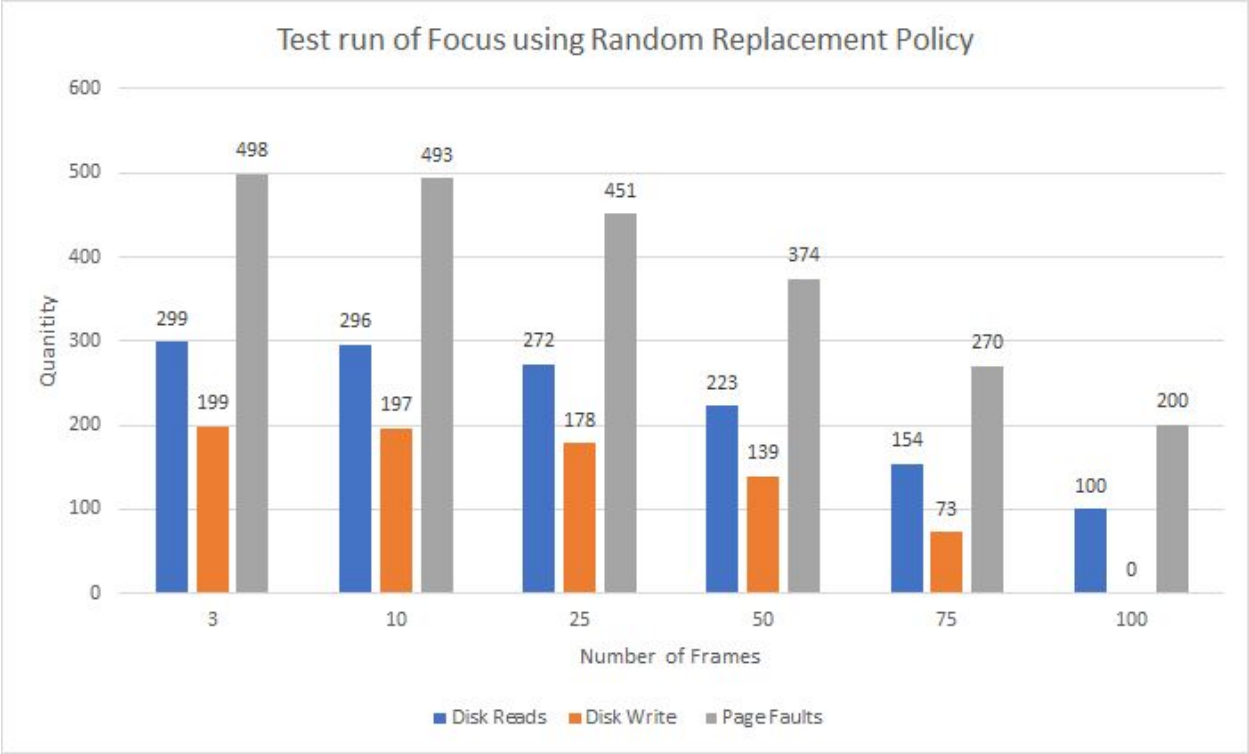

Figure 6: Test runs of the program “Focus” using the Random replacement policy, varying the number of frames.

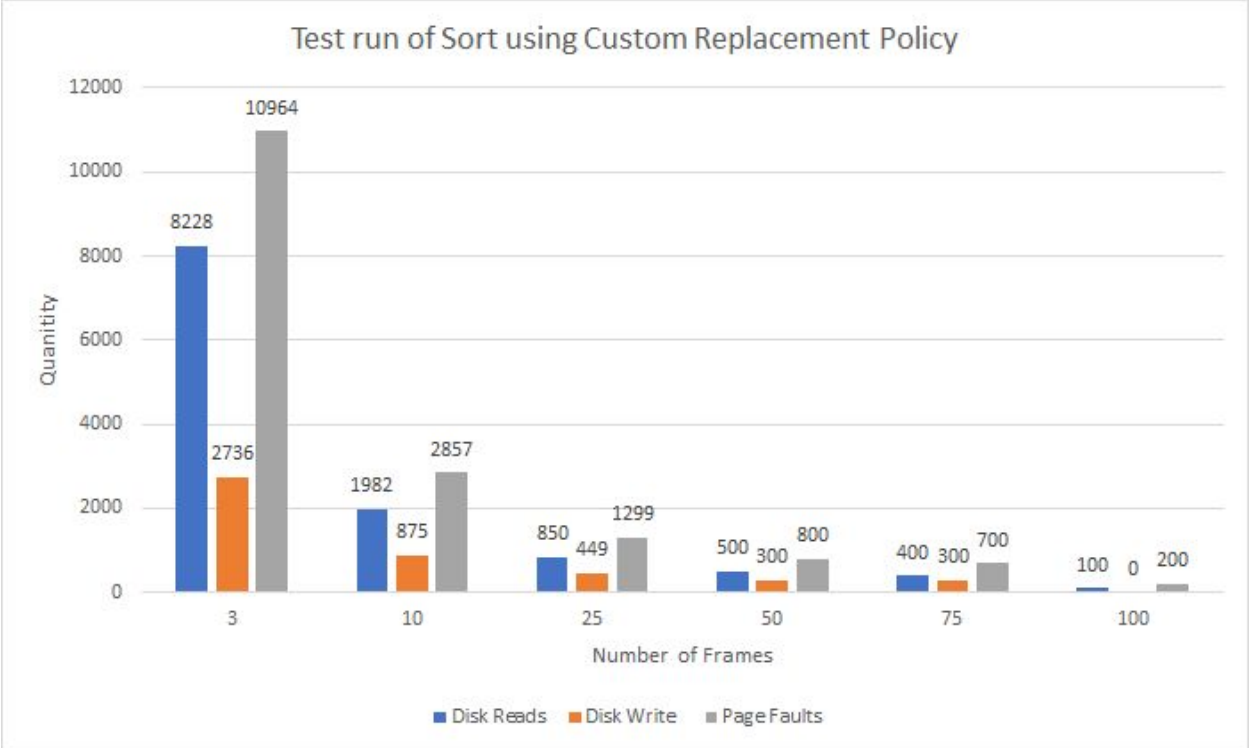

Figure 7: Test runs of the program “Sort” using the Custom replacement policy, varying the number of frames.

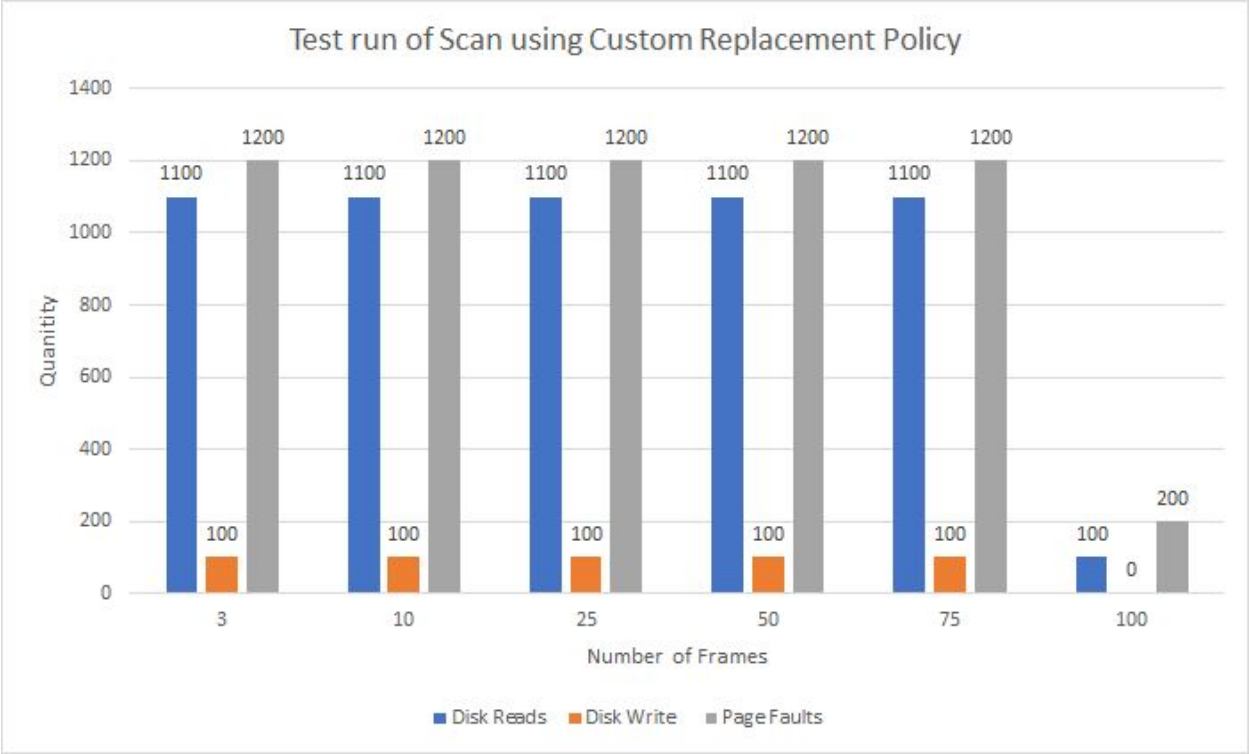

Figure 8: Test runs of the program “Scan” using the Custom replacement policy, varying the number of frames.

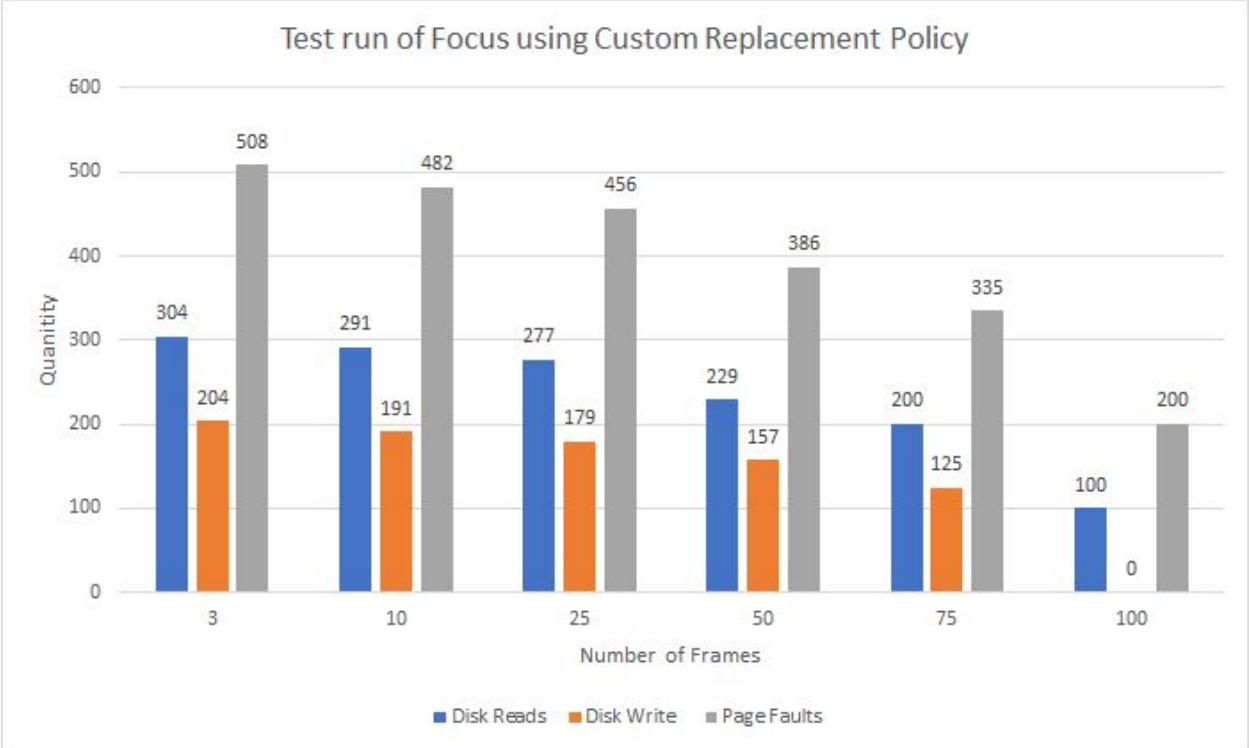

Figure 9: Test runs of the program “Focus” using the Custom replacement policy, varying the number of frames.