Group Members

Anthony Dierssen-Morice - diers040

Tushar Gowda - gowda019

Purpose

The purpose of this experiment is to compare and understand different page replacement algorithms with respect to the number of page faults and disk writes that occur while executing programs with different memory access patterns. By running the same page replacement algorithm on different memory access patterns we are able to assess its general strengths and weaknesses. Additionally, by comparing the results of different page replacement algorithms used on similar memory access patterns, we are able to gain insights into which one is better suited to a particular use case. All results herein discussed were generated using the VOLE-2D virtual machine. To generate the graphs presented below, the following commands were run from the project root directory:

> ./generate_graphs.sh

> python3 compare_policies_generate_graph.py

The gererate_graphs.sh script runs ./virtmem with 100 pages and a varying number of frames (1-100) for each of the 9 possible combinations of page replacement algorithms and test programs. All output from generate_graphs.sh is stored in the directory csv-files/. The compare_policies_generate_graph.py program uses the results of generate_graphs.sh to create comparison graphs which are combined into a single figure called Compare_policies.png stored in the project root directory.

Custom Page Replacement Algorithm

Our custom page replacement algorithm takes inspiration from the classic clock algorithm. We maintain a use bit for each frame that is stored in the core map entry. This bit is set whenever a new page is allocated to a frame. It is also set when we trap back to the OS due to a permission issue.

There is an alarm that goes off every 1ms. This alarm helps to trap back to main.cpp. Here we scan through all the frames in memory and check the use bit that is stored against each frame. For any frame, if the use bit is set to false, we know that the frame was not used between the previous alarm and now and thus we log this in a data structure called not_used by incrementing its value by one. If the use bit is set to true, we set it to false and change the page permissions to PROT_NONE. We also reset the not_used data structure for this frame back to zero. If the

program tries to access this frame between two consecutive alarms, it will trap back to the page handler where we reset the use bit and reset the not_used data structure (As mentioned above).

Not_used is a data structure that tracks the number of consecutive times that the frame was marked as not_used and the alarm went off. For example, a value of 3 signifies that the frame was not used between 3 consecutive alarms. This gives us a measure to determine the least recently used page. Whenever there is a page fault, we can use the not_used data structure to select the appropriate frame to evict from memory.

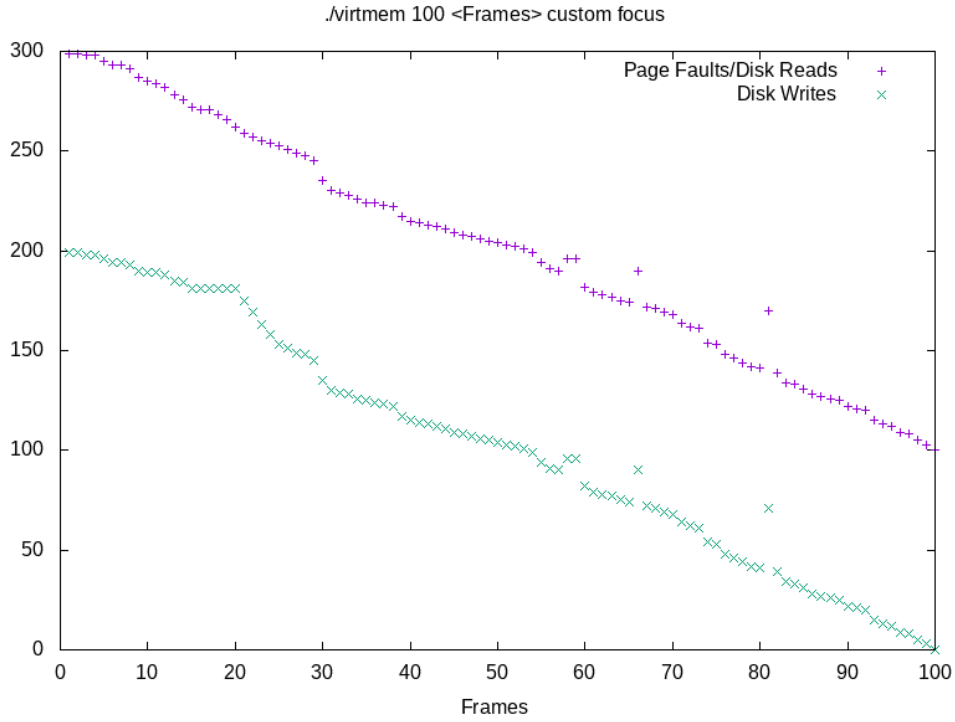
Whenever there are multiple candidate frames for eviction, we filter down the list by trying to optimize in terms of disk writes. We select the pages that are not dirty and hence will not require a write back. If we still have more than one candidate frame, we use FIFO.

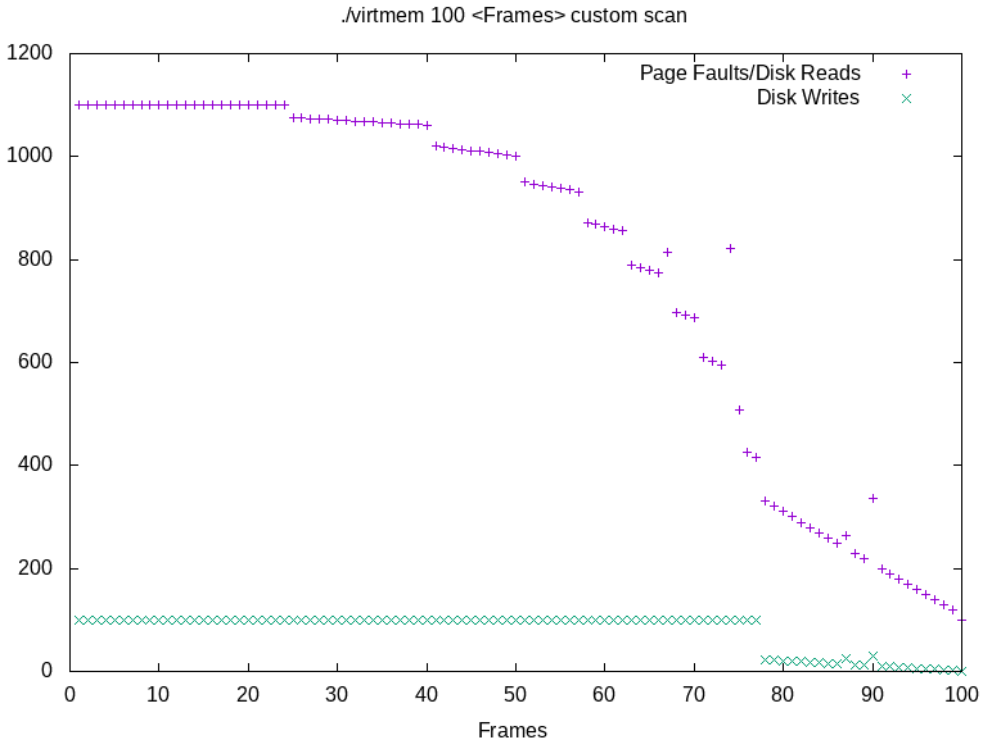
We are using a millisecond alarm. This resolution is not ideal and can sometimes fail to give us eviction candidate frames. In this case, the other 2 filters are triggered. In order to prevent dirty pages from unfairly staying resident in memory for too long under this filtering scheme, we have also incorporated a heuristic that depends on the total number of frames and increases the likelihood of eviction if a dirty frame has survived eviction for too long.

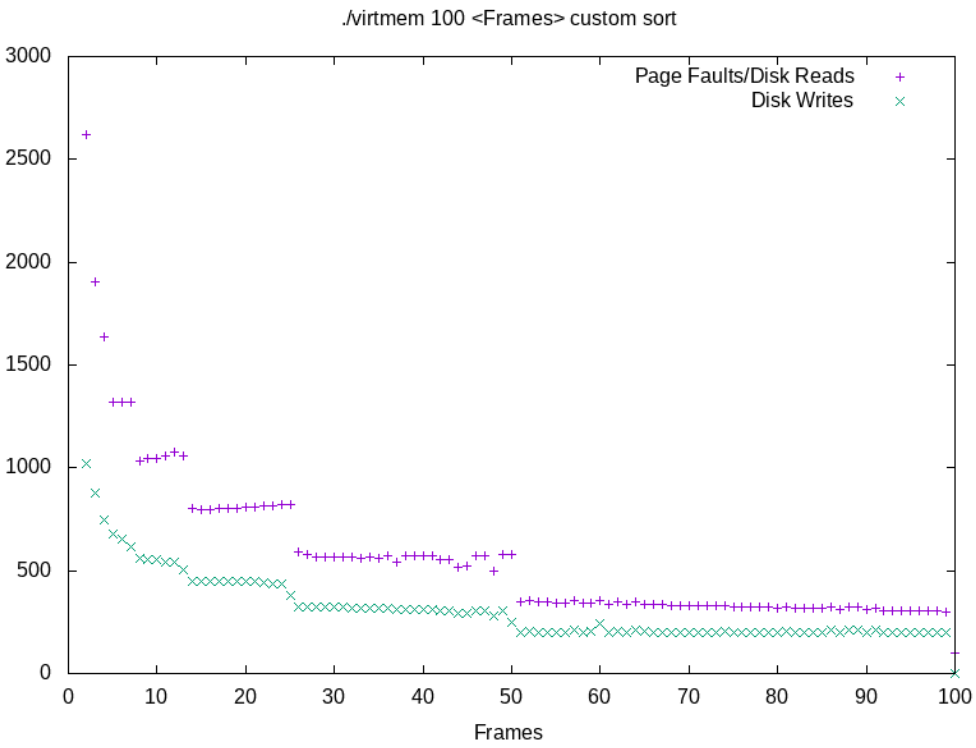
Results

Individual graphs for each of the page replacement algorithms and programs are presented on the pages that follow. The underlying data for each of the graphs can be found in csv-files/ in the project root directory. Additionally, comparative graphs have been included which overlay the results of the three different page replacement algorithms.

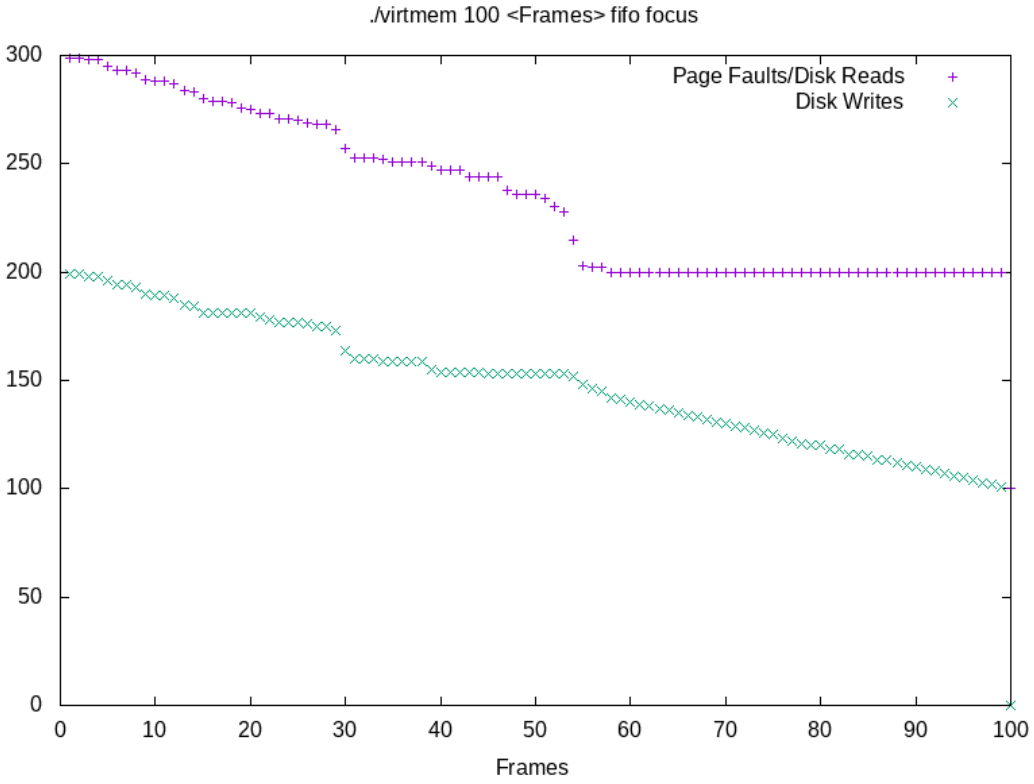
Custom Page Replacement Algorithm:

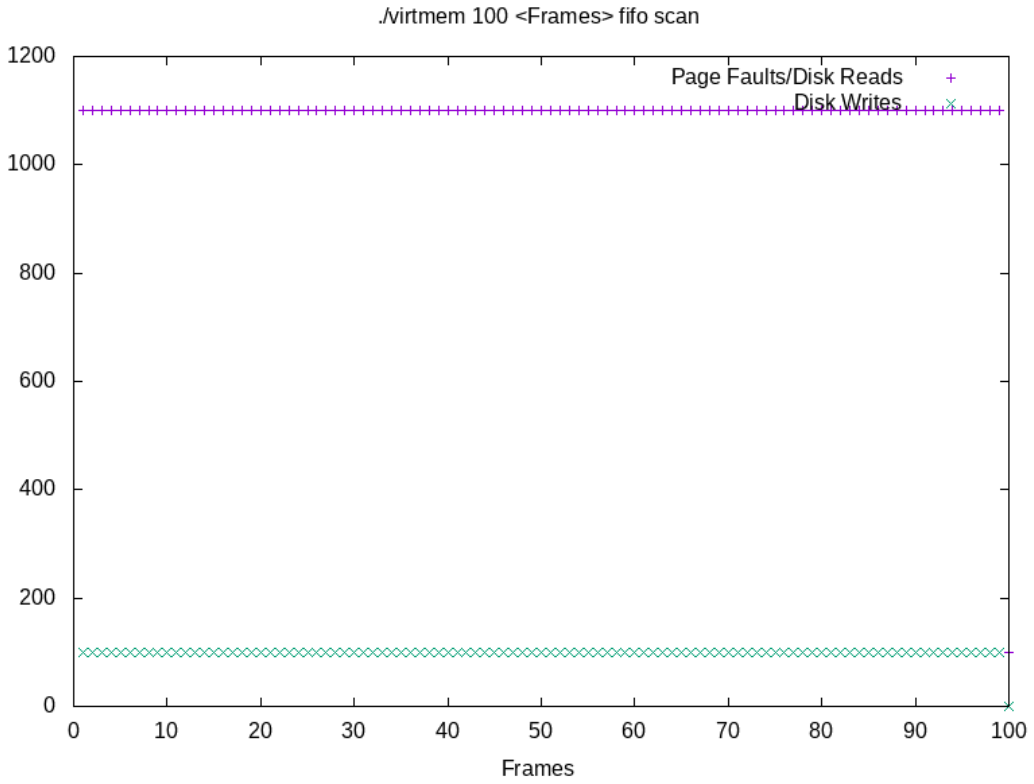


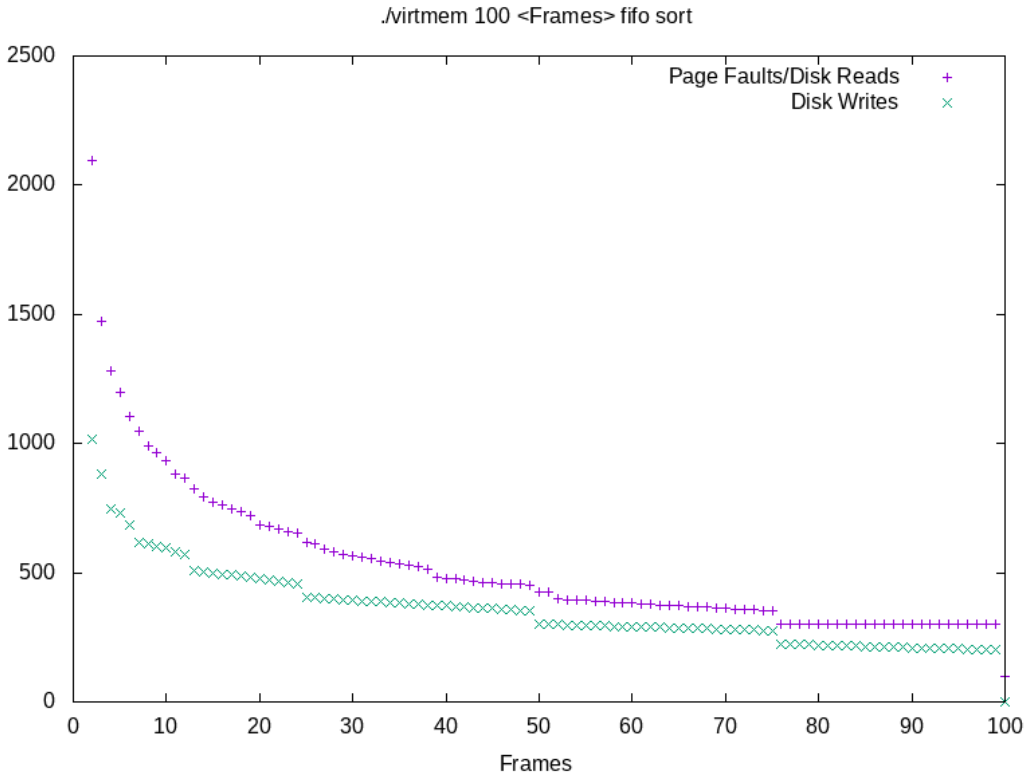



./virtmem 100 <Frames> custom sort
Page Faults/Disk Reads
Disk Writes
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100
Frames

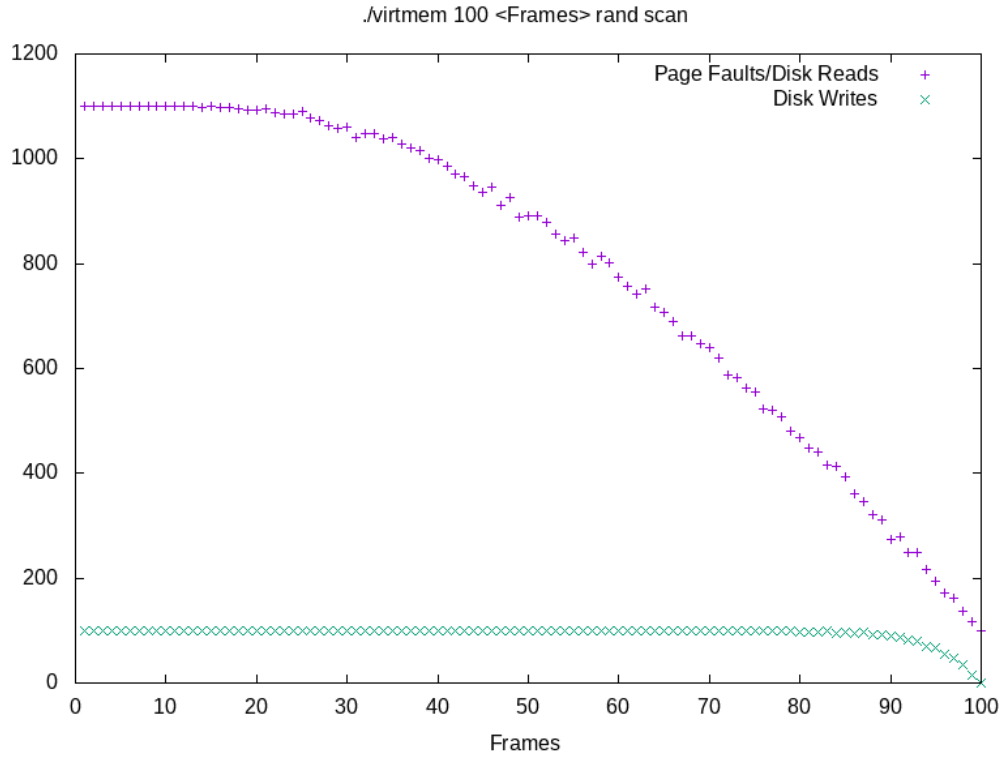
FIFO Replacement Algorithm

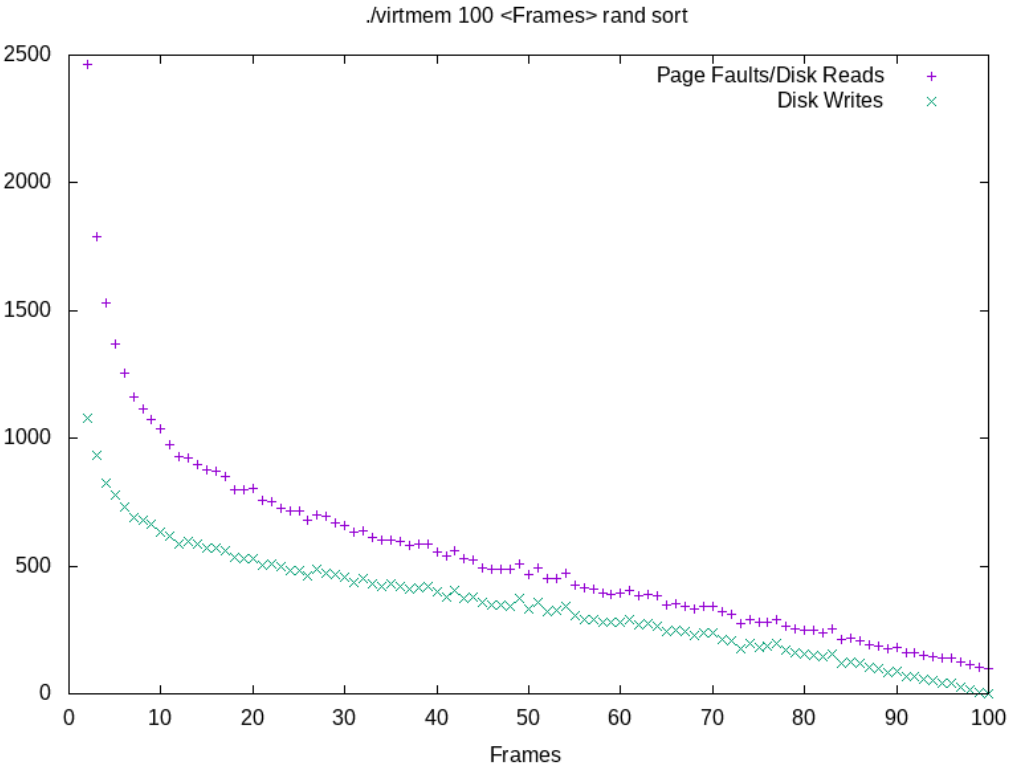


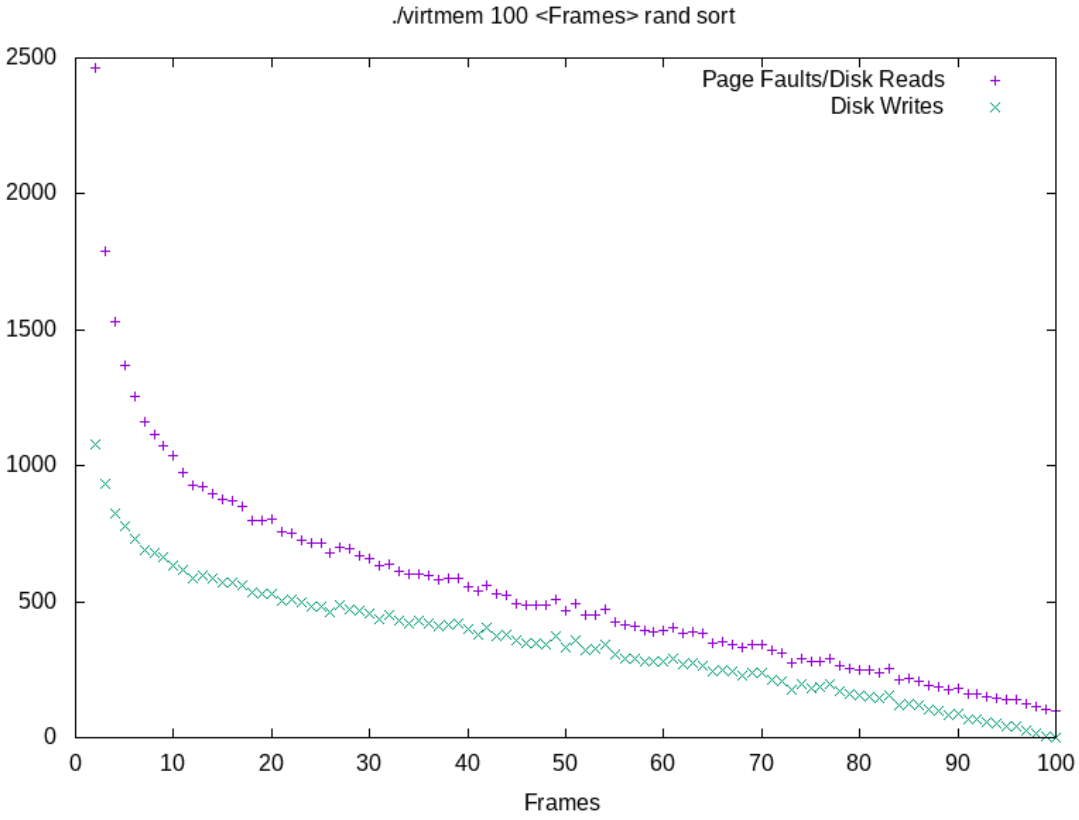



./virtmem 100 <Frames> fifo sort
Page Faults/Disk Reads
Disk Writes
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100
Frames

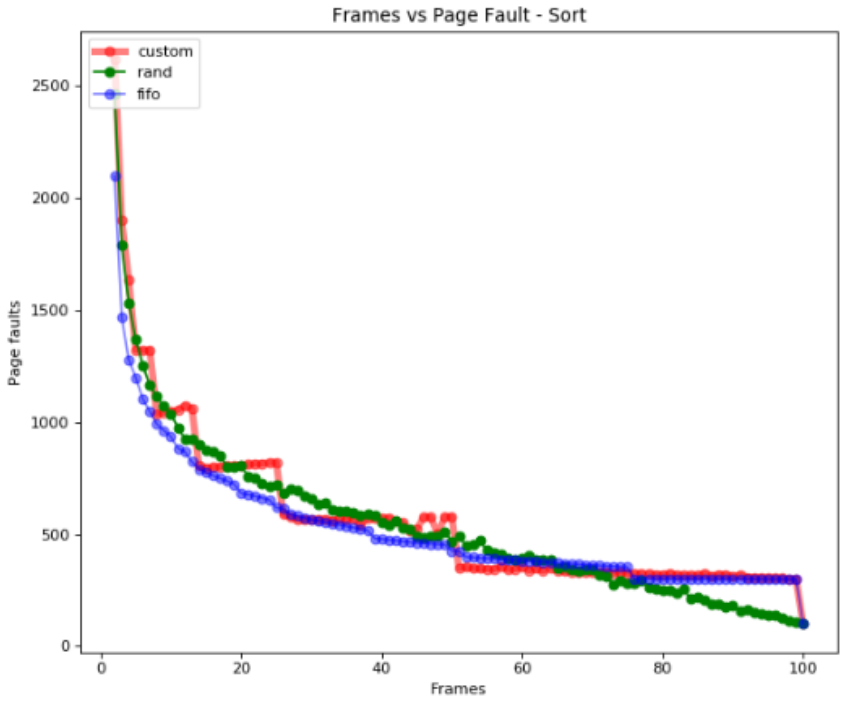
Random Replacement Algorithm

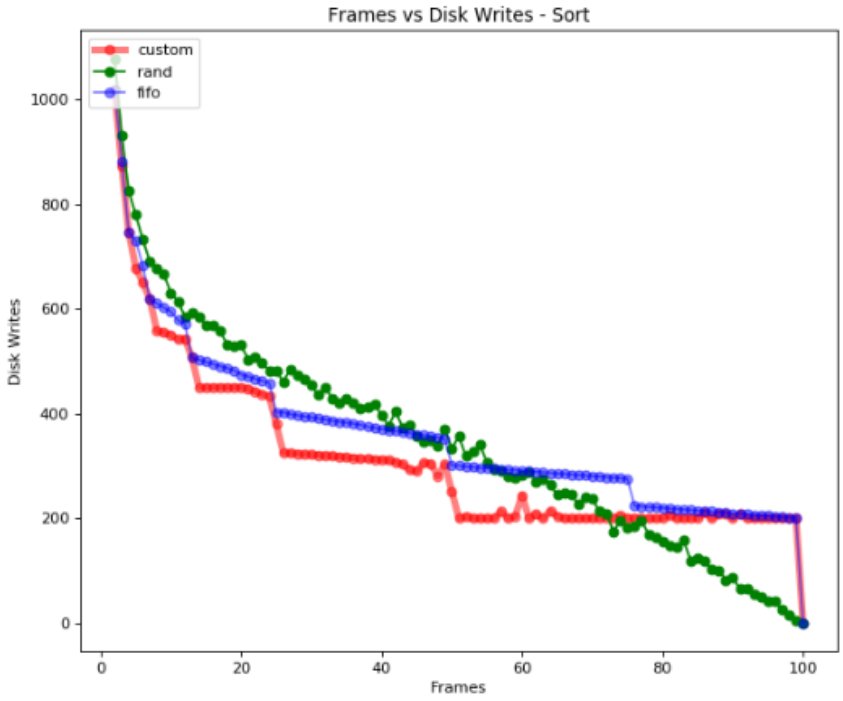


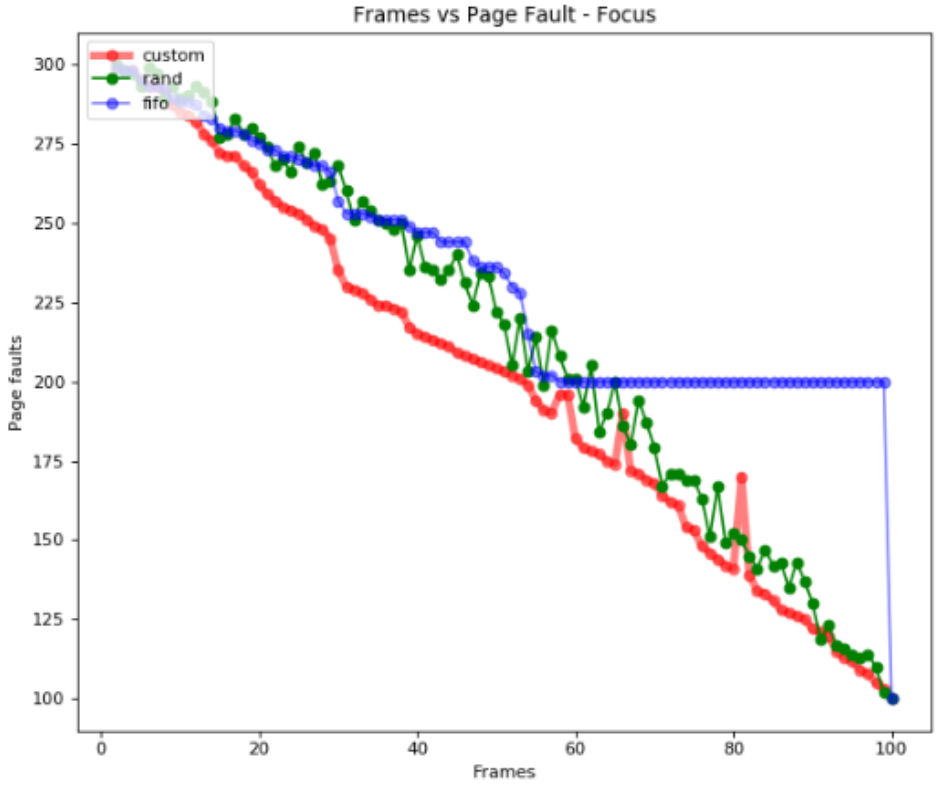


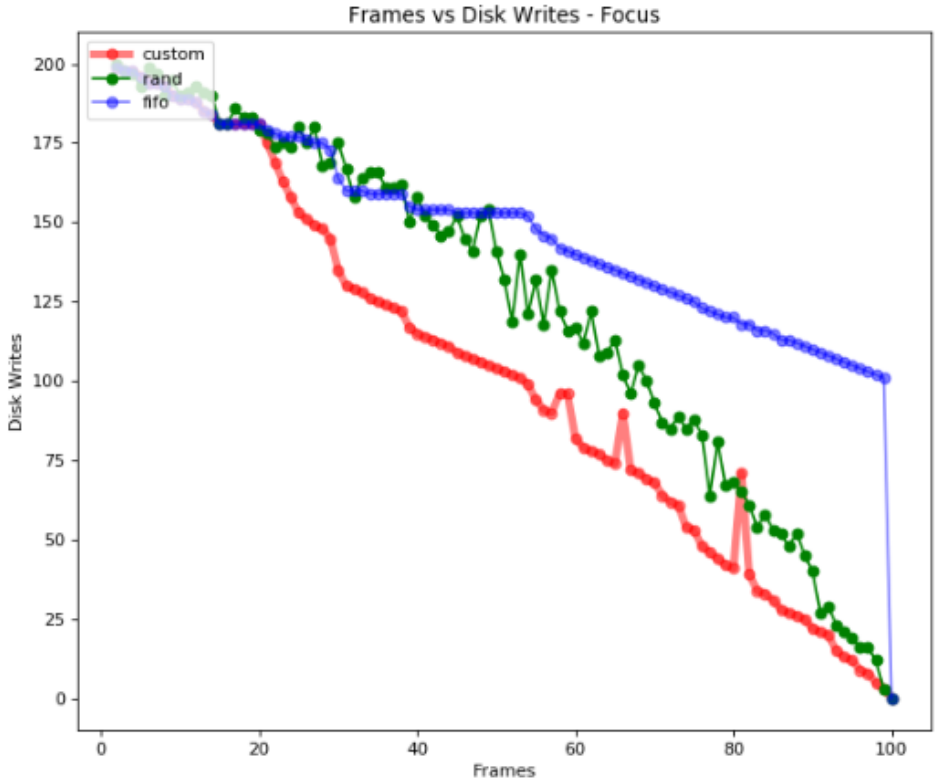

./virtmem 100 <Frames> rand sort
Page Faults/Disk Reads
Disk Writes
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100
Frames

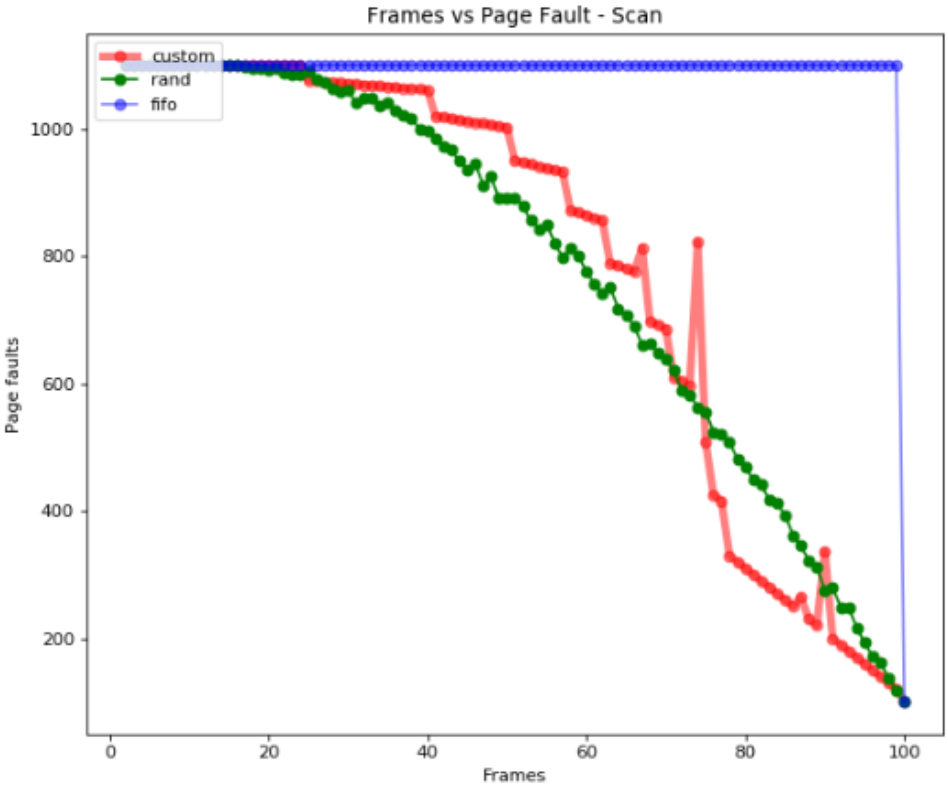
Comparative Graphs

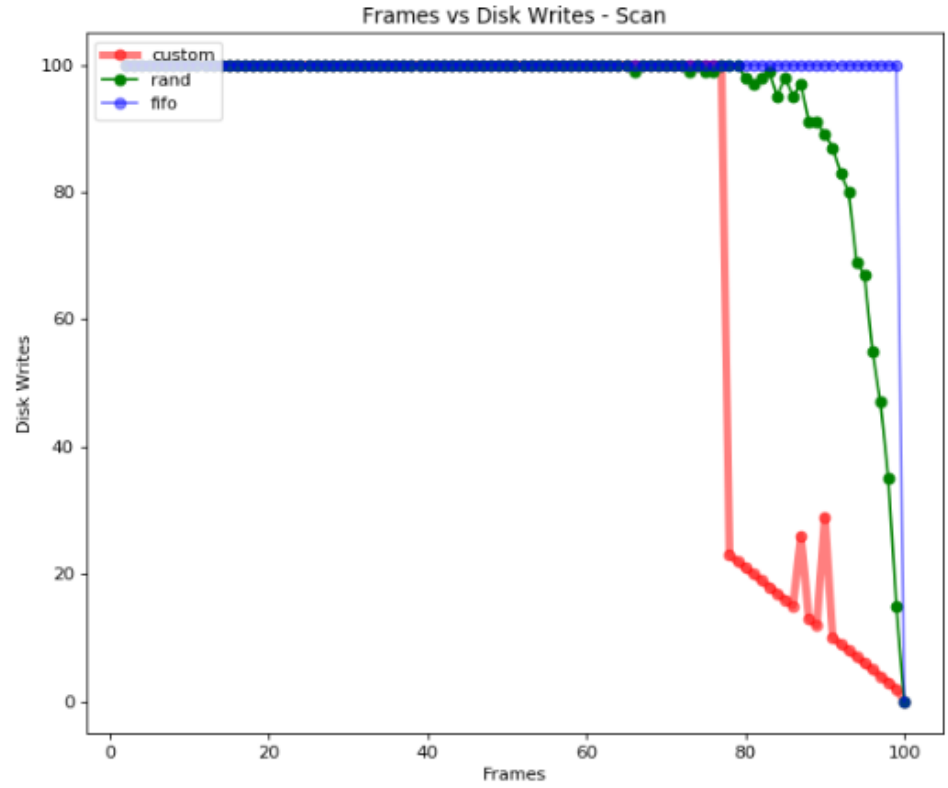





Frames vs Page Fault - Focus
custom
rand
fifo
Page faults
Frames
300
275
250
225
200
175
150
125
100
0
20
40
60
80
100


Frames vs Disk Writes - Focus
custom
rand
fifo
Disk Writes
Frames
200
175
150
125
100
75
50
25
0
0
20
40
60
80
100


Frames vs Page Fault - Scan
custom
rand
fifo
Page faults
Frames
1000
800
600
400
200
0
20
40
60
80
100


Frames vs Disk Writes - Scan
custom
rand
fifo
Disk Writes
Frames
100
80
60
40
20
0
0
20
40
60
80
100

Analysis of Results

First note that in the event that the number of pages is equal to the number of frames (i.e. nPages = nFrames = 100), once the frames have been brought into memory, the working set of the program remains resident in memory for the entire duration of the program. This means that no further page faults will occur: a fact which results in identical performance for all page replacement algorithms regardless of the memory access pattern when nPages = nFrames since each program, at some point, checks each element of the data array.

Next, consider the scan program. In the average case, it was observed that the random page replacement algorithm performed best for the scan program in terms of page faults, while the custom clock replacement algorithm performed best in terms of disk writes. Looking at the memory access pattern of the scan program, we can see that there is a sequential traversal of each page 11 times. This is the worst possible access pattern for the FIFO replacement algorithm as the longest resident frame is the most likely to be accessed next when looping through the pages in this fashion. As such, it is no surprise to see that FIFO performed the worst of the three page replacement algorithms for the scan program. Our custom clock replacement algorithm, meanwhile, performed better than FIFO in terms of page faults, resembling instead the performance of the random replacement algorithm. This is of note given that the clock algorithm is attempting to approximate LRU which should perform exactly the same as FIFO for this particular memory access pattern. The reason for which our clock algorithm differs from FIFO, in this case, is due to the limitations described in the implementation notes above. Given that the granularity of the timer is not fine enough, our algorithm uses the dirty page filter and keeps a set of dirty pages in memory. This reduces the number of writebacks that are required as we increase the number of frames.

For the focus program, it was found that our custom clock replacement algorithm performed best both in terms of page faults and disk writes. When looking at the focus program’s memory access pattern, we can see that after an initial traversal of the entire data array, the program isolates, at random, small segments of the array and writes elements at random positions in the segment. Given that the segment of the array that is being looped changes at random, and that the size of each of these segments is smaller than the page size, we would expect that all page replacement algorithms perform roughly the same. This largely matches what we see in the graphs, with the slopes of all the series being approximately equivalent. Notice, however, that the FIFO replacement algorithm plateaus at 200 page faults, regardless of the number of frames, except in the case that nFrames = nPages. This is because the focus program does both an initial loop through all elements in the data array (which necessarily leads to nPages = 100 page faults), and again, in the end, iterates through all elements in the data array after the focus loops have finished (this along with the page faults in the focus algorithm causes 100 page faults (if nFrames < nPages) due to the FIFO property). In the case of disk writes, random and fifo are similar up to a point as we would expect but since fifo removes the wrong frames from the memory during the

final loop, it has more disk writes than random. Custom, on the other hand, performs brilliantly as it is optimized to remove pages that are not dirty.

For the sort program, in the average case, it was observed that the FIFO page replacement algorithm performed slightly better in terms of page faults than the other two, while the custom clock replacement algorithm performed best in terms of disk writes. While we observe some differences, in a broad sense we can see that all the replacement algorithms have similar trends for both page faults and disk writes. As we increase the number of frames both the page faults and writes decrease similarly for all three algorithms. Fifo is performing better in terms of page faults when the number of frames are less because of the way quick sort is designed but it tends to be worse than random at higher number of frames. In terms of disk writes the custom algorithm works best as it is optimized for writes and gives priority to clean pages over dirty pages when deciding which page to evict.