Project 2: Measuring the performance of Page Replacement Algorithms on Real Traces

Ghulam Jilani Quadri
U74821298
ghulamjilani@mail.usf.edu

Mehul Suresh
U52982115
mehul1@mail.usf.edu

Aim: To program in C to implement LRU and VMS page replacement algorithm on real traces.

Introduction:

LRU:

Known as Least Recently Used which uses the history and evicts a page which is not recently been used. If a program has accessed a page in the near past, it is likely to access it again in the future. One type of historical information a page- replacement policy could use is frequency; if a page has been accessed many times, perhaps it should not replace as it clearly has some value.

VMS:

This policy is also known as second chance algorithm. This algorithm is based on the FIFO page replacement algorithm but with little improvement. In the Second Chance page replacement policy, the candidate pages for removal are consider in a round robin matter, and a page that has been accessed between consecutive considerations will not be replaced.

Implementation:
The final submission file consists of the following contents:

- memsim.c
- makefile
- report.pdf

LRU: Least recently used, we have simulated this page replacement algorithm with the help of array of structures and array. We have two functions to implement LRU – lru() and update_table(); lru() functions actually performs the methodology of cold start and finding the victim in case the table is full. The update_table() method is used to update the table based on the hit, miss or eviction. We have created a frame using the array and each page entry to the frame is a structure which consists of –address, read/write bit, dirty bit(in case of modification the page content and reference and instance. To implement the concept of LRU, we have a variable *gcount*- which stores the clock timing of referenced page. The page which was not recently referenced will have the lowest clock count and caused to be the victim for the replacement.

The following table shows the performance of LRU replacement algorithm with frame size of 12 for all given four traces:

Name of the Trace	Number of events	Hit count	Miss count
Gcc.trace	1000000	864579	134521
Bzip.trace	1000000	996093	3907
Sixpack.trace	1000000	864356	135644
Swim.trace	1000000	774181	225819

Table1: Reading of LRU for all the four trace input.

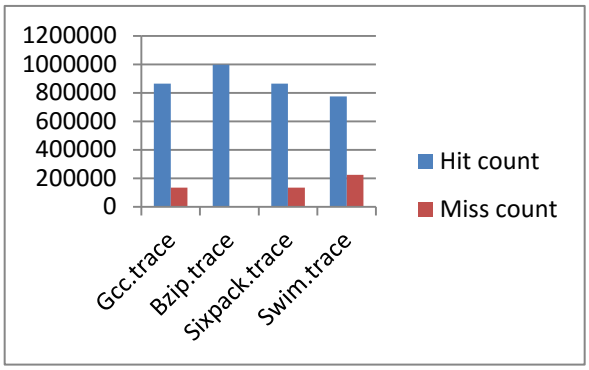

Fig1: bar-graph showing Hit and count ratio.

Seeing the above table readings, we can say that Bzip.trace performed well in this scenario. But considering the term ***Frequency,*** the table reading values does not varies much except bzip.trace. LRU entirely depend on the Frequency and bit used to represents the number of time a particular page is referenced. and here comes "The Principle Of locality", which says-programs tends to access Certain codes sequence and data structures quite frequently and thus we should try to use history to figure out which pages are important . and this concepts benefits the LRU most.

Below Table summarizes the hit and miss counts for the bzip.trace with different cache size
Algorithm: **LRU** Trace: **bzip.trace**

Frame size	Hit count	Hit rate	Miss count
4	907230	90%	92770
8	969309	96%	30691
16	996656	99%	3344
32	997867	99.78%	2133
256	9999603	99.99%	397
1024	9999683	99.99%	317

Table 2: Table readings summarizes hit and miss count on bzip trace for various frame size

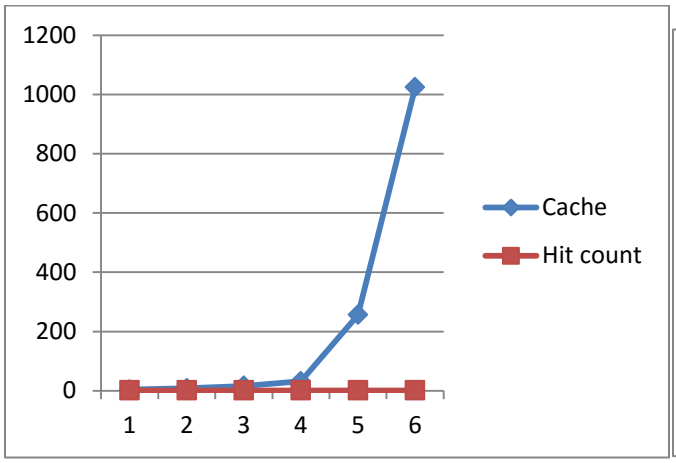

Fig3: Hit count Vs Cache size

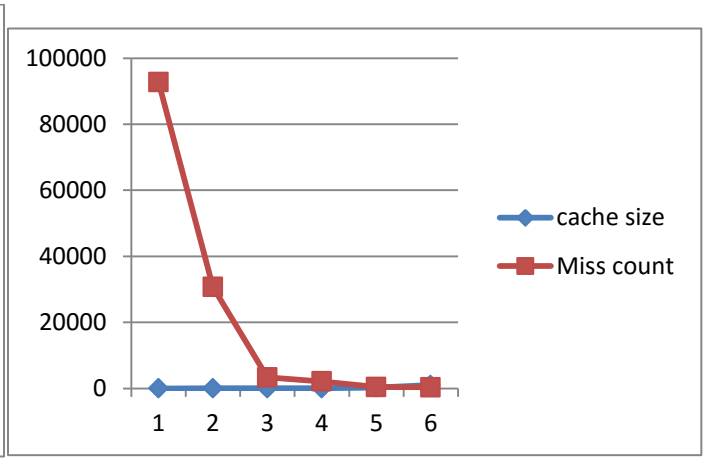

Fig4: Miss count Vs Cache size

VMS: In our Implementation of VMS which is also known as second chance algorithm on FIFO. We have the structure for this simulation as follows- we have three frame lists, one for primary FIFO cache, one for global clean list and one for global dirty list. The size of the primary FIFO is defined by frame cache size by the passed argument while for the global clean list and dirty list we have hard coded. As per our Implementation, simulation works as follows. The primary FIFO will first gets populated by cold start.

- When a new page is requested, it will be first searched in the Primary FIFO and if found - hit. If not present and primary is not full, it is loaded from disk into primary. If primary is full,
  - Based on the FIFO, a victim is chosen for eviction. If it is a dirty entry, it will be stored in dirty list. If is not a modified entry, it will go to clean list.
  - Now, if either or both of global dirty and clean list are full, a victim is replaced from evicted entry of primary FIFO.
- When searching for page entry in FIFO first, if not found we will search in both global list. If found it is a hit

Performance:
The following table shows the performance of VMS algorithm for the frame size 12 on all the four trace.

Name of the Trace	Number of events	Hit count	Miss count
Gcc.trace	1000000	903617	96383
Bzip.trace	1000000	997467	2533
Sixpack.trace	1000000	915369	84631
Swim.trace	1000000	909048	90952

Table2: Reading of LRU for all the four trace input.

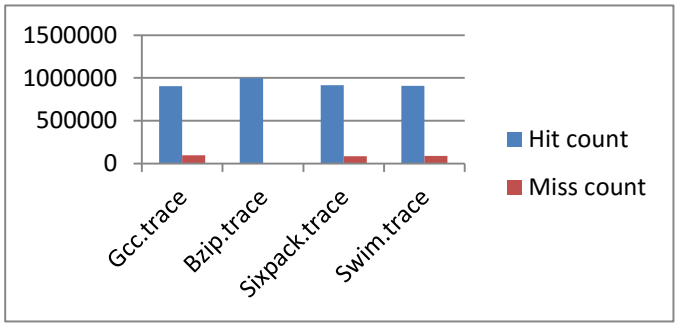

Fig5: bar-graph showing Hit and count ratio.

Below Table summarizes the hit and miss counts for the bzip.trace with different cache size
Algorithm: **VMS** Trace: **bzip.trace**

Frame size	Hit count	Hit rate	Miss count
4	970929	97%	29072
8	996658	99.66%	3342
16	997907	99.79%	2097
32	998762	99.87%	1238
256	999683	99.96%	317
1024	999683	99.96%	317

Table 3: Table readings summarize hit and miss count on bzip trace for various frame size

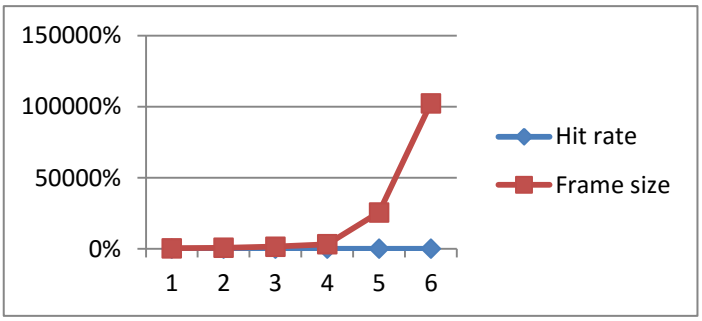

Fig 5: cache size vs hit rate ratio.

Observations:
In both algorithm it is observed that, after a certain limit of number of frames, there comes a stable time where there is negligible observation in hit. Since, cache memories are costly and they should be utilized efficiently, so we have to allocate a close frame size.

The bigger these global second-chance lists are the performance of is closer to LRU algorithm.

Answers of the following questions:
Q1) How much memory does each traced program actually need?
Answer:
Amount of memory consumed: no.of misses*4KB
Below are memories required by traces if they will work frame size of 12.

Trace name	LRU	VMS
Gcc.trace	525.4MB	376.24MB
Bzip.trace	15.26MB	9.89MB
Sixpack.trace	529.8MB	330.5MB
Swim.trace	882.1MB	355.2MB

Another solution is :
Consider, if I will have the maximum number of frames to be 1024 and each size is 4096.
Then memory each trace will need is : 2^10 X 2^12= 2^22= 4194304= 4MB

Q2) which page replacement algorithm works best?
Answer: In our case , VMS page replacement algorithm worked best, because as per our consideration, VMS is having three time more cache memory as compared to LRU(if we are comparing on the same frame size input). Since VMS is having two extra global page list which gives an extra efficiency by giving victim a second chance. But again VMS needs extra memory and computation speed as compared to LRU.

Q3) Does one algorithm work best in all situations? If not, what situations favor what algorithm?
Answer: We can't say any one algorithm works best. LRU works best in the case where frequency is considered and in case of maintaining history. This is clearer by the concept of Principle of Locality. The bigger these global second-chance lists are the performance of is closer to LRU algorithm.
VMS, there is no hardware bit to hold the reference bit hence it works on the concept of FIFO.For better performance Of VMS, the global page list size should be good enough to perform well closer to LRU.
VMS needs more computation time and memory space than LRU. Implementation of LRU is much simpler than the VMS.