Anthony Poerio (adp59@pitt.edu)
CS1550 – Summer 2016
University of Pittsburgh
Project #3 - VMSimulator
For: Prof Jon Misurda

Overview

For project 3 we are tasked with implementing 4 different Page Replacement algorithms, then comparing the results using two memory traces named "swim.trace" and "gcc.trace". All algorithms are run within page table implemented for a 32-bit address space; all pages in this page table are 4kb in size.

I chose to implement these algorithms and the page table in Python. The final source code can be found in the same folder as this document. The full data set that I collected can be found at the end of this document. Illustrative graphs are interspersed throughout, wherever they are necessary for explaining and documenting decisions.

In this essay itself, I will first outline the algorithms implemented, noting design decisions I made during my implementations. I will then compare and contrast the results of each algorithm when run on the two provided memory traces. Finally, I will conclude with my decision about which algorithm would be best to run in a real Operating System.

Please note, to run the algorithms themselves, you must run:

- `python vmsim.py –n <numframes> -a <opt|clock|aging|lru> [-r <refresh>] <tracefile>`

The Algorithms

The algorithms we have been asked to implement are:

- **Opt** – Simulate what the optimal page replacement algorithm would choose if it had perfect knowledge
  - **EXAMPLE RUN**: python vmsim.py –n 8 –a opt gcc.trace
- **Clock** – Use the better implementation of the second-chance algorithm
  - **EXAMPLE RUN**: python vmsim.py –n 16 –a clock swim.trace
- **Aging** – Implement the aging algorithm that approximates LRU with an 8-bit counter
  - **EXAMPLE RUN**: python vmsim.py –n 32 –a aging –r 1 gcc.trace
- **LRU** – Do exact LRU.
  - **EXAMPLE RUN**: python vmsim.py –n 64 –a lru swim.trace

Implementation Notes

The main entry point for the program is the file named **vmsim.py**. This is the file which must be invoked from the command line to run the algorithms. Because this is a python file, it must be called with "python vmsim.py … etc.", instead of "./vmsim", as a C program would. Please make sure the selected trace file is in the same directory as vmsim.py.

All algorithms run within a page table, the implementation of which can be found in the file named **pageTable.py**.

Upon program start, the trace file is parsed by the class in the file **parseInput.py**, and each memory lookup is stored in a list of tuples, in this format: [(MEM_ADDRESS_00, R/W), [(MEM_ADDRESS_01, R/W), … [(MEM_ADDRESS_N, R/W)]. This list is then passed to whichever algorithm is invoked, so it can run the chosen algorithm on the items in the list, element by element.

The **OPT algorithm** can be found in the file **opt.py**. My implementation of OPT works by first preprocessing all of the memory addresses in our Trace, and creating a **HashTable** where the *key is our VPN*, and the *value is a python list* containing each of the address numbers at which those VPNs are loaded. Each time a VPN is loaded, the element at index 0 in that list is discarded. This way, we only have to iterate through the full trace once, and from there on out we just need to hash into a list and take the next element, whenever we want to know how far into the future that VPN is next used.

The **Clock Algorithm** is implemented in the files **clock.py** and **circularQueue.py**. My implementation uses the second chance algorithm with a Circular Queue. Of importance: whenever we need to make an eviction, but fail to find ANY pages that are clean, we then run a 'swap daemon' which writes out ALL dirty pages to disk at that time. This helps me get fewer page faults, at the expense of more disk writes. That's a calculated decision for this particular algorithm, since the project description says we should use page faults as our judgment criterion for each algorithm's effectiveness.

The **LRU Algorithm** can be found in the file **lru.py**. For LRU, each time a page is Read, I mark the memory address number at which this happens in the frame itself. Then, in the future—whenever I need to make an eviction—I have easy access to see which frame was used the longest time in the past, and no difficult calculations are needed. This was the simplest algorithm to implement.

The **Aging Algorithm** is likely the most complex, and its source code can be found in the file named **aging.py**. Aging works by keeping an 8 bit counter and marking whether each page in the page table was used during the last 'tick' a time period of evaluation which must be passed in by the user as a 'refresh rate', whenever the Aging Algorithm is selected. All refresh rates are in milliseconds on my system, but

this relies on the implementation of Python's "time" module, so it's possible that this could vary on other systems. **For aging, I suggest a refresh rate of 0.01 milliseconds**, passed in on the command line as "-r 0.01", in the 2nd to last position in the arg list. This minimizes the values in my testing, and going lower does not positively affect anything. In the next section, I will show my rationale for selecting 0.01 as my refresh rate.

Aging Algorithm Refresh Rate

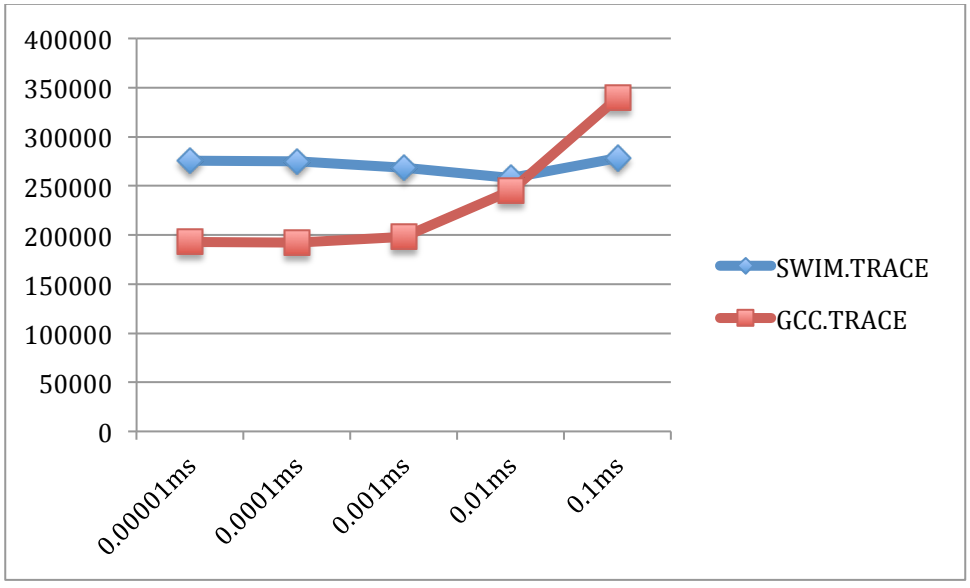
In order to find a refresh rate that would work well, I decided to start at 1ms and move 5 orders of magnitude in each direction, from 0.00001ms to 100000ms.

To ensure that the results were not biased toward being optimized for a single trace, I tried both to confirm that the refresh rate would work will for all inputs.

The graphs below show the Page Faults and Disk Writes I found during each test.

For all tests, I chose a frame size of 8, since this small frame size is most sensitive to the algorithm used. At higher frame sizes, all of the algorithms tend to perform better, across the board. So I wanted to focus on testing at the smallest possible size, preparing for a 'worst case' scenario.

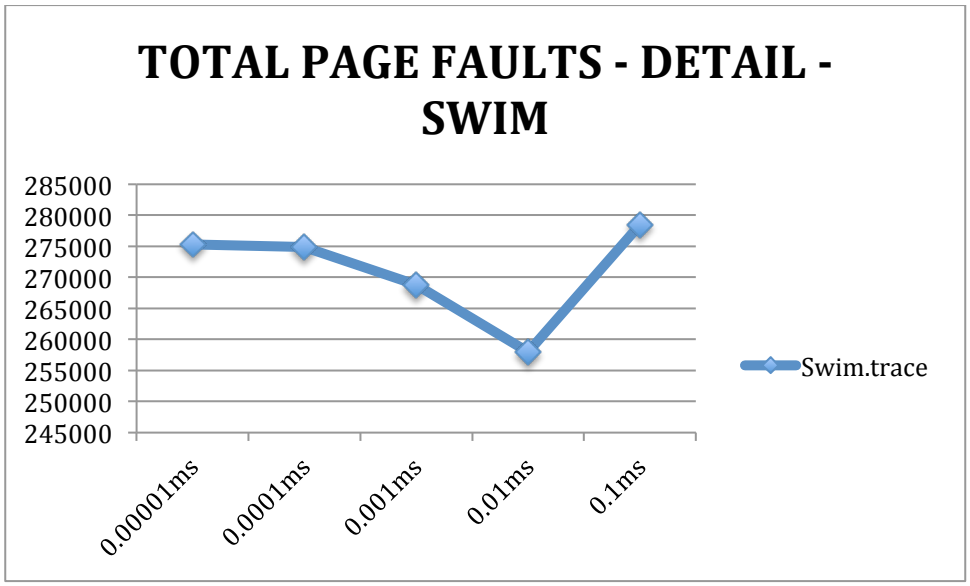
Aging Algorithm – Page Faults Over Refresh Rate



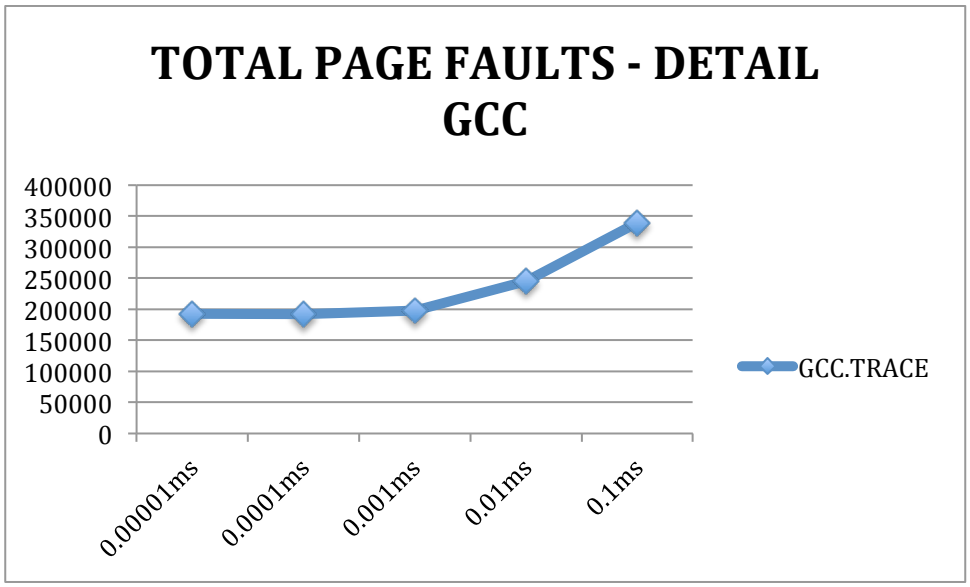
X-axis: Refresh rate in milliseconds
Y-axis: Total Page faults

GCC. TRACE reaches its minimum for page faults at 0.0001ms and SWIM.TRACE reaches its own minimum at 0.01ms. The lines cross at around 0.01ms.

Total Page Faults For SWIM.TRACE – Detail



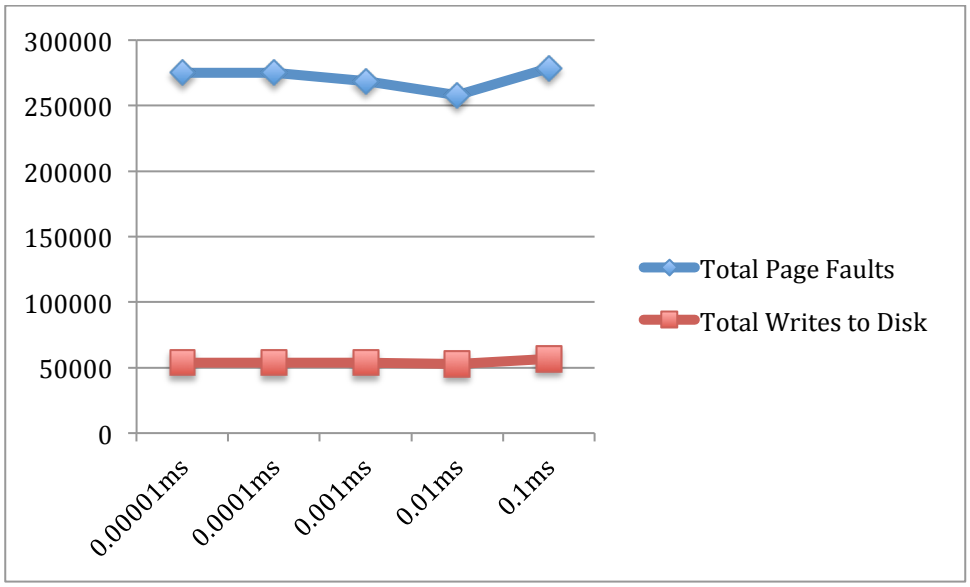
X-axis: Page Faults
Y-Axis: Refresh Rates



X-axis: Page Faults
Y-Axis: Refresh Rates

Additionally, 0.01ms seems to achieve the best balance, in my opinion, if we need to select a single time for BOTH algorithms.

Aging Algorithm – Disk Writes Over Refresh Rate



X-axis: Refresh rate in milliseconds
Y-axis: Total Disk Writes

The number of disk writes also bottoms out at 0.01ms from SWIM.TRACE. It is relatively constant for GCC.TRACE across all of the different timing options.

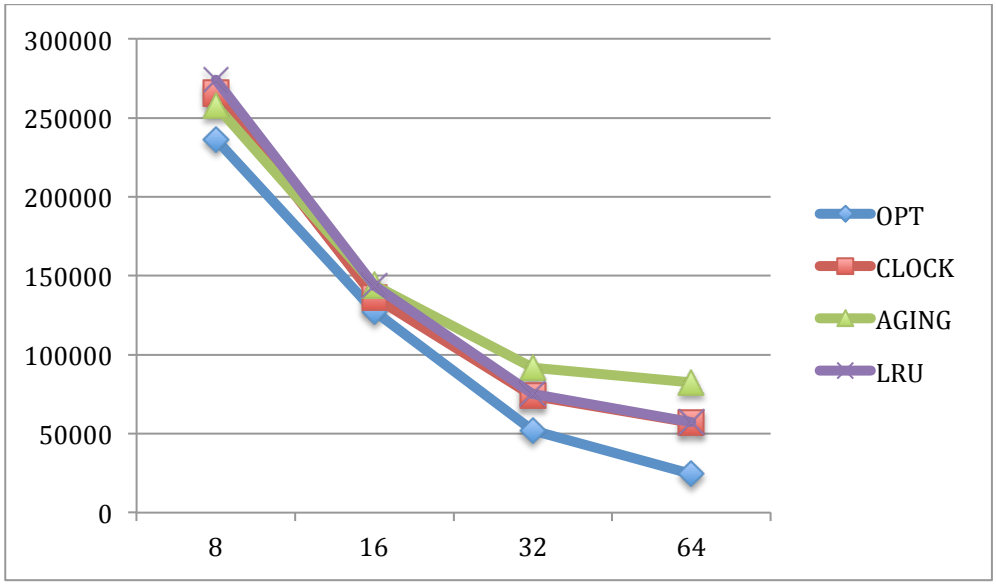
Because of this, I suggest 0.01ms as the ideal refresh rate. This is because it is optimal for SWIM.TRACE. For GCC.TRACE, it is not the absolute best option, but it is still acceptable, and so I think this selection will achieve a good balance.

Results & Decisions

With the algorithms all implemented, my next step was to collect data for each algorithm at all frame sizes, 8, 16, 32, and 64. OPT always performed best, and thus it was used as our baseline.

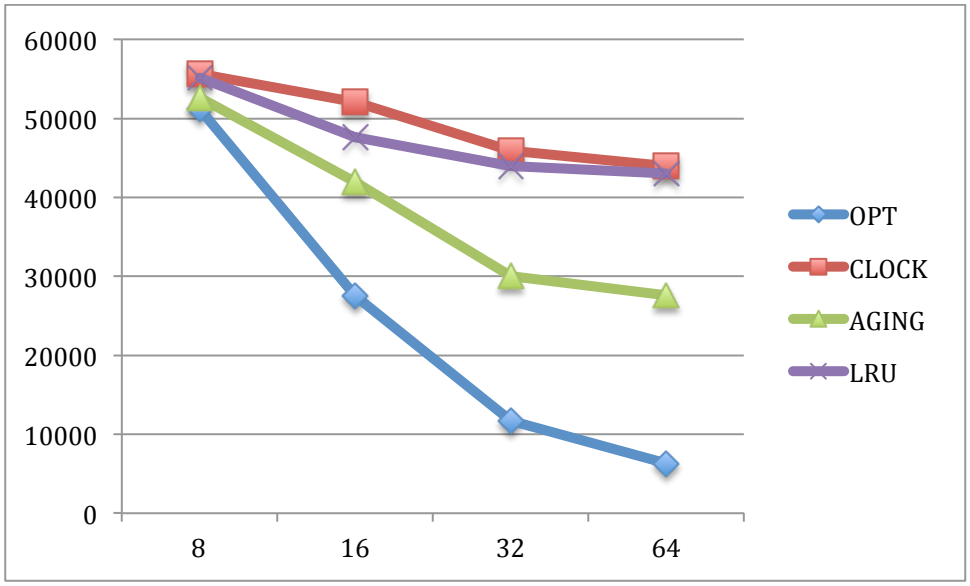
In the graphs below, I show how each algorithm performed, both in terms of total page faults and total disk writes.

SWIM.TRACE – Page Faults Over Frame Size



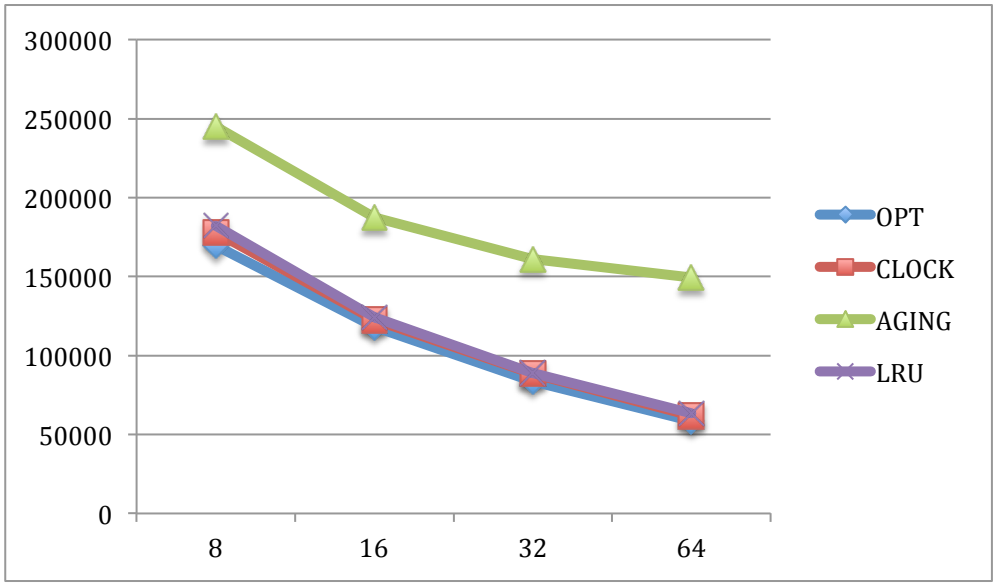
X-axis: Frame Size
Y-axis: Page Faults
Data for all algorithms processing swim.trace

SWIM.TRACE – Disk Writes Over Frame Size



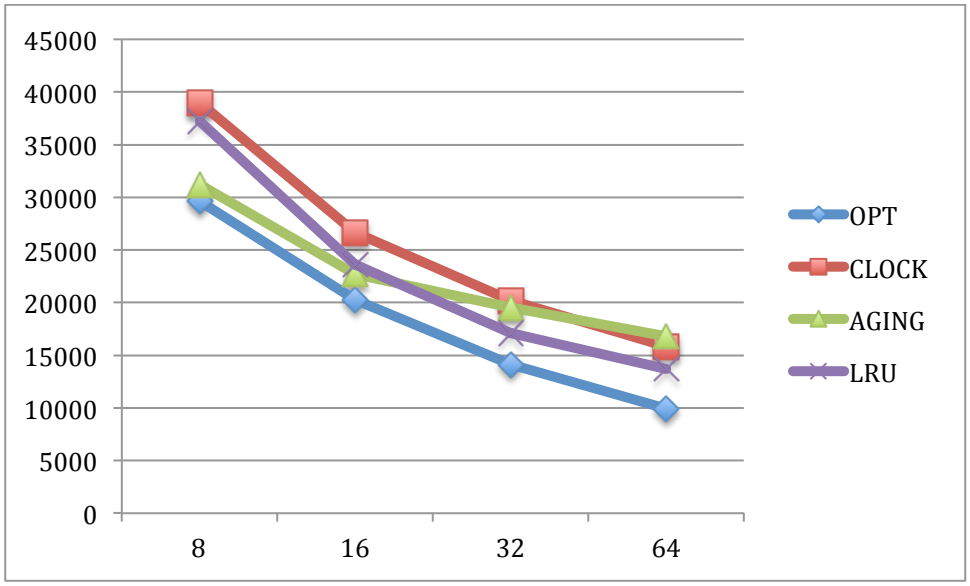
X-axis: Frame Size
Y-axis: Disk Writes
Data for all algorithms processing swim.trace

GCC.TRACE – Page Faults Over Frame Size



X-axis: Frame Size
Y-axis: Page Faults
Data for all algorithms processing gcc.trace

GCC.TRACE – Disk Writes Over Frame Size



X-axis: Frame Size
Y-axis: Disk Writes
Data for all algorithms processing gcc.trace

--

Given this data, I was next tasked with choosing which algorithm is most appropriate for an actual operating system.

In order to determine which algorithm would be best, I decided to use an algorithm. I'll call it the 'Decision Matrix', and here are the steps.

DECISION MATRIX:

1. **RANK ALGORITHMS FOR EACH CATEGORY FROM 1=BEST to 4=WORST,**
2. **SELECT ALGORITHM WITH LOWEST OVERALL SCORE**

ALGORITHM	SWIM - Page Faults	SWIM - Disk Writes	GCC - Page Faults	GCC - Disk Writes	TOTAL (Lowest is Best)
OPT	1	1	1	1	4
CLOCK	2	4	2	4	12
AGING	4	2	4	2	12
LRU	3	3	3	3	12

Ranking: **1=Best;**
4=Worst

NOTE:

- *Wherever lines cross each other in our graphs, the algorithm ranked as "better" is the one with MORE total low data points. Ties were broken by my own personal judgment.*

Decision:

- Of course, using this set of judgment criteria, OPT is by far the winner.
- However OPT isn't an option in a real OS, since it requires perfect knowledge of the future, which is impossible in practice.
- *So we **want to pick between the 3-way tie** for CLOCK and LRU and AGING.*
    - Of these three:
        - AGING does well on disk writes, but generally has the most page faults.
        - Conversely, CLOCK does well on page faults, but often has the most disk writes.
        - LRU achieves a balance.
    - Therefore, My pick goes to LRU, because where clock does beat LRU, on page faults it does so only by a narrow margin.
    - But where LRU beats clock—on disk writes—it does so by a large amount. This means LRU is better overall, if both page faults and disk writes are equally weighted for judgment purposes.

Therefore, I would select LRU for my own operating system.

Data Set

Figure 1.1 – Full Data Set

ALGORITHM	NUMBER OF FRAMES	TOTAL MEMORY ACCESSES	TOTAL PAGE FAULTS	TOTAL WRITES TO DISK	TRACE	REFRESH RATE
OPT	8	1000000	236350	51162	swim.trace	N/A
OPT	16	1000000	127252	27503	swim.trace	N/A
OPT	32	1000000	52176	11706	swim.trace	N/A
OPT	64	1000000	24344	6316	swim.trace	N/A
OPT	8	1000000	169669	29609	gcc.trace	N/A
OPT	16	1000000	118226	20257	gcc.trace	N/A
OPT	32	1000000	83827	14159	gcc.trace	N/A
OPT	64	1000000	58468	9916	gcc.trace	N/A
CLOCK	8	1000000	265691	55664	swim.trace	N/A
CLOCK	16	1000000	136154	52104	swim.trace	N/A
CLOCK	32	1000000	73924	45872	swim.trace	N/A
CLOCK	64	1000000	56974	43965	swim.trace	N/A
CLOCK	8	1000000	178111	38992	gcc.trace	N/A
CLOCK	16	1000000	122579	26633	gcc.trace	N/A
CLOCK	32	1000000	88457	20193	gcc.trace	N/A
CLOCK	64	1000000	61832	15840	gcc.trace	N/A
AGING	8	1000000	257952	52664	swim.trace	0.01ms
AGING	16	1000000	143989	41902	swim.trace	0.01ms
AGING	32	1000000	91852	29993	swim.trace	0.01ms
AGING	64	1000000	82288	27601	swim.trace	0.01ms
AGING	8	1000000	244951	31227	gcc.trace	0.01ms
AGING	16	1000000	187385	22721	gcc.trace	0.01ms
AGING	32	1000000	161117	19519	gcc.trace	0.01ms
AGING	64	1000000	149414	16800	gcc.trace	0.01ms
LRU	8	1000000	274323	55138	swim.trace	N/A
LRU	16	1000000	143477	47598	swim.trace	N/A
LRU	32	1000000	75235	43950	swim.trace	N/A
LRU	64	1000000	57180	43026	swim.trace	N/A
LRU	8	1000000	181950	37239	gcc.trace	N/A
LRU	16	1000000	124267	23639	gcc.trace	N/A
LRU	32	1000000	88992	17107	gcc.trace	N/A
LRU	64	1000000	63443	13702	gcc.trace	N/A

Figure 1.2 – GCC.TRACE - Refresh Rate Testing – 8 Frames

ALGORITHM	NUMBER OF FRAMES	TOTAL MEMORY ACCESSES	TOTAL PAGE FAULTS	TOTAL WRITES TO DISK	TRACE	REFRESH RATE
AGING	8	1000000	192916	33295	gcc.trace	0.00001ms
AGING	8	1000000	192238	33176	gcc.trace	0.0001ms
AGING	8	1000000	197848	31375	gcc.trace	0.001ms
AGING	8	1000000	244951	31227	gcc.trace	0.01ms
AGING	8	1000000	339636	140763	gcc.trace	0.1ms

Figure 1.3 –SWIM.TRACE - Refresh Rate Testing – 8 Frames

ALGORITHM	NUMBER OF FRAMES	TOTAL MEMORY ACCESSES	TOTAL PAGE FAULTS	TOTAL WRITES TO DISK	TRACE	REFRESH RATE
AGING	8	1000000	275329	53883	swim.trace	0.00001ms
AGING	8	1000000	274915	53882	swim.trace	0.0001ms
AGING	8	1000000	268775	53540	swim.trace	0.001ms
AGING	8	1000000	257952	52664	swim.trace	0.01ms
AGING	8	1000000	278471	56527	swim.trace	0.1ms