Operating Systems
COP4600-001

Measuring the Performance of Replacement Algorithms on Real Traces

Date:	March 2, 2017
Name:	Tyler Simoni Andrew Rodiek
UNumber:	U25068858 U84332681
Approx Hours Spent:	86 hours
Difficulty:	Moderately difficult

1. Project Objectives

The objective of `memsim` is to correctly simulate the actions of a single level virtual page table using different page replacement policies. The policies used in `memsim` are *Optimal*, *Least Recently Used*, and *Clock*.

The *Optimal* page replacement algorithm looks ahead in what is called a reference string – a list of every memory access that occurs – and selects pages that are used the farthest away to evict.

The *Least Recently Used* page replacement algorithm uses a policy fitting of its name; it evicts the page that was accessed furthest from the current time.

The *Clock* page replacement algorithm iterates through the pages that are currently loaded in frames. If the current page is not already loaded, the policy will begin marking a use bit for replacement. If the "clock" hand reaches a frame where the page is marked as not being recently used, it is evicted.

2. Methods Used

2.1. Optimal Page Replacement

The first algorithm implemented was the *Optimal* page replacement algorithm. The first approach the group took was quite complex.

Initially, a structure that contained a Boolean value, an unsigned integer, and an array of integers. The Boolean value marked if the trace was designated as a read or right instruction. The unsigned integer represented the virtual page number being loaded into a frame. The array of integers stored the index of every occurrence of the unique page number. The program would then create a linked list between each unique virtual page structure and iterate through them as per the policy description in the textbook. After hours of testing and debugging, the group decided to scrap this idea and try something new altogether.

The new idea was to create a structure that holds the virtual page number, stored as an unsigned integer, and the read/write flag, stored as a single character. An array of structures containing all the traces and their values was instantiated, as well as an array of structures representing the frames in physical memory. The optimal algorithm then used these to look ahead in the reference array and mark which of the currently placed frames were farthest and evicted them.

Tests of hit ratio vs cache size experiments were performed and explained in more detail below. This also goes for the *Least Recently Used* and *Clock* page replacement policies.

2.2. Least Recently Used Page Replacement

The second algorithm implemented was the *Least Recently Used* page replacement algorithm. This algorithm only used one approach; however, it took a significant amount of time to get the logic correct.

The structure of traces, previously created for *Optimal*, was recycled for this algorithm implementation, as well as used in *Clock* as it proved to be highly useful in storing the information needed for the virtual pages. The difference in this implementation comes from the way the algorithm needed to move and iterate through the array of structures.

Coincidentally, the *Least Recently Used* algorithm shares a few attributes with the way a queue data structure is implemented. That is, if the array of frames is not full, it functions as a *first in – first out* structure, in that the new values enter at the "back" of the array, moving with the first one in being the "front". This continues until the array is full. Once full, it needs to check each frame to see if the current page is already loaded into it. This is where the group chose to use a common implementation of queues in coding and iterate through it circularly, rather than linearly. This is done by incrementing by the index, plus one, modulus the size of the array (for example, (i + 1) % size). This means that when the index is at the last index, the next index will be instead the first index of the array. This proved to be very efficient as well as prevent buffer overflows. The `front` and `back` of the array were help as indices that were constantly being updated as the algorithm worked through the logic.

2.3. Clock Page Replacement

The final algorithm implemented was the *Clock* page replacement algorithm. For this algorithm, the group also used one approach. It shared many attributes with the previously programmed *Least Recently Used* policy, as well as the structure, and, thus, the group was able to use a lot of code implementation from the last algorithm. This cut down significantly on programming time.

The difference in code from the *Least Recently Used* came in the form of the clock hand and the 'use' bit. The `clock` and `back` pointer functioned similarly to the `front` and `back` of before, with slight tweaks to both of their movements. It also used the idea of a circular queue to move around, as if it were on a "clock". The use bit was added into the structure definition in order to make it simpler to hold its value. It needed to be checked once the algorithm had circled around and not found the page is was looking for in the frames.

3. Results

The experiments on the three implemented algorithms provided close to expected results. The best overall was the *Optimal* page replacement policy. The *Clock* page replacement policy performed very closely to *Optimal*. The group believes the advantage to come from the "look-ahead" mechanic that *Optimal* uses. *Least Recently Used* performed pretty poorly compared to the other two, as can be seen from the graph and data below. At its core *Least Recently Used* is a flawed policy due to the fact that just because a program was used recently does not mean it will need to be loaded in memory in the future.

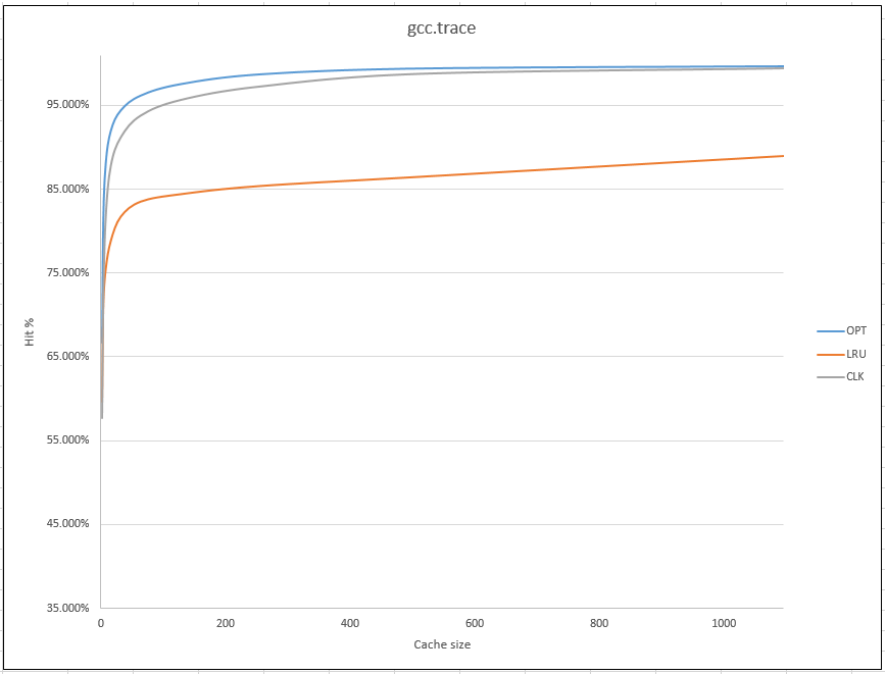

Figure 3-1. Hit Ratio vs Cache Size for gcc.trace

Optimal Policy				
Hits	Hit %	Cache Size	Reads	Writes
666479	66.648%	2	333521	52587
814240	81.424%	4	185760	16862
881515	88.152%	8	118485	10277
919691	91.969%	16	80309	7675
944196	94.420%	32	55804	5431
961946	96.195%	64	38054	3734
975601	97.560%	128	24399	2692
987321	98.732%	256	12679	1633
994315	99.432%	512	49600	4813
996996	99.700%	1096	3004	596

CLK				
Hits	Hit %	Cache Size	Reads	Writes
577129	57.713%	2	203312	26956
723348	72.335%	4	222876	28023
818348	81.835%	8	176257	17659
878308	87.831%	16	121522	10682
912300	91.230%	32	87695	8221
938347	93.835%	64	61652	6027
956919	95.692%	128	43080	4269
973072	97.307%	256	26927	2995
988169	98.817%	512	11830	1709
994774	99.477%	1096	5225	1385

LRU				
Hits	Hit %	Cache Size	Reads	Writes
596045	59.605%	2	403955	39123
711718	71.172%	4	288282	29409
755144	75.514%	8	244856	29497
788234	78.823%	16	211766	31038
817161	81.716%	32	182839	31943
835151	83.515%	64	164849	31739
844026	84.403%	128	155974	31496
853836	85.384%	256	146164	31256
864816	86.482%	512	135184	29113
889670	88.967%	1096	110330	22332

Figure 3-2. Hit Ratio vs Cache Size data for gcc.trace

When *Optimal* was run with a shortage of memory, it completes significantly faster than with an average amount of memory. This is, in part, due to the nature of the algorithm. Because there are less frames to check for each "look into the future", the algorithm can quickly move through the entire array of memory. Even with more memory, it still slowly speeds up as it gets closer and closer to the trace file. This is because there are less and less trace files to check for as it moves through the array. The loop has a near dynamic quality to it, as the loop gets shorter with longer runtime.

When *Optimal* was run with an excess of memory, its runtime is comparable when it has a shortage of memory. This is because the algorithm doesn't have to make any checks for hits, as it is just filling up an array that is bigger than the amount of values being put in. Also with an excess of memory, there are no writes to disk due to a page never having to replace another page that is writing to disk.

The biggest thing to take from *Optimal* is that once it surpasses 16 frames, it stays above the 90th percentile in hit rating. This, in turn, leads to less reads and writes to the disk, which means it is less time consuming.

When *Least Recently Used* was ran with a shortage of memory, it was have a very high excess of reads and writes compared to other algorithms, as well as other memory sizes. When given an excess of memory, *Least Recently Used* had an equal number of reads to traces. This is because as the algorithm starts it first has to fill the array, or queue, of frames up with pages. If there's more frames than pages, then it just quickly fills them up and doesn't have to do any comparisons other than checking to see if has filled up. Overall *Least Recently Used* performed averagely, only barely getting into the 80th percentile of hit ratio vs cache size.

When *Clock* was used with a shortage of memory, it performed similarly to *Least Recently Used*. However, once the caches size starts increasing, it quickly starts to speed up and almost catches up to the performance of *Optimal* with the same cache sizes. With an excess of memory, it performs identically to *Least Recently Used,* as they are functionally identical in terms of the way the pages are stored in the array frames.

The real difference between *Clock* and *Optimal* comes in the form of run time. *Clock* significantly out performs *Optimal* in this aspect, wholly in part due to, again, the nature of the *Optimal* algorithm. Because it has to look-ahead so many times, over and over, it loses speed in the beginning, whereas *Clock* keeps a near constant pace at the same cache sizes.

From these results, the group would say that the *Clock* page replacement policy would be the best algorithm to implement in a system, to it outperforming *Optimal* in runtime speed and *Least Recently Used* in the sheer number of reads and writes occurring.

4. Conclusions

The three algorithms all proved to be quite interesting and challenging to implement successfully. Many functions, parameters, and a whole lot of time was put into making it both functional and as efficient as possible, within the scope of the group's knowledge. The algorithms are quite powerful considering the service they provide for computers, yet they are quite simple in both nature and implementation. It would be interesting to try to implement other page replacement policies, a bigger trace file or try and devise even a more efficient algorithm.