Operating Systems

Project 2: Paging Simulation

Osniel Quintana
Felipe Gutierrez

INTRODUCTION:

Page replacement is an important component of an operating system. When a page containing a desired piece of information or instruction is searched in translation buffers or page tables and found missing from main memory, a page fault is said to occur. A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. As the size of main memory is limited and is much smaller than the size of physical memory, the role of page replacement is to identify the best page to remove from main memory as a result of a page fault and replace it by the new page from disk that contains the requested value or instruction. The page replacement algorithm used in the project are First-In-First-Out, Least Recently Used , and the VMS Second Chance Algorithm.

This project uses a simulator to conduct FIFO, LRU or VMS page replacement policies when accessing memory specified by the traces given for this project. The simulator maintains a Page Table, as well as it's insertions and removals. When the Page Table is full, and a new address needs to be loaded, one of the replacements policies are used to decide which page to remove. When pages are loaded into memory (into the Page Table), the queue pertaining to FIFO, LRU or VMS is updated accordingly. Similarly, when a page hit occurs (an address is already in memory), the page is referenced for LRU or VMS.

The user has control over the number of frames used in the page table and the trace file to be used for the simulation. These parameters can be passed as arguments on terminal when the simulator is to be executed.

The purpose of this project is to determine the overall efficiency of these three different replacement policies when applied to large trace files (containing 1 million entries). Multiple

frames sizes will also be used in the calculation of this efficiency. The measures used to calculate the overall performance of these policies will be the total number of reads from disk and writes to disk and the hit rate, given some frames that the user inputs.

Time Spent: 48 hours

METHODS:

The algorithms we used in this project as the replacement policies are FIFO, LRU and VMS Second Chance Algorithm. FIFO keeps a list of all pages currently in memory, the page at the beginning of the list being the oldest one and the page at the end being the most recent one. When the Page Table is full, and some space needs to be made for another page, FIFO replacement policy evicts the oldest page in memory (first one to arrive). Least Recently Used policy replaces the least-frequently used page when an eviction must take place. Pages are added to the queue as they are loaded into memory, but when a page is referenced and it is in memory, the page is brought to the front of the queue. LRU evicts pages the same way FIFO does but maintains its queue differently, as previously shown. The last method that we used is the VMS second chance lists where pages are placed in two different process queues when loaded into memory and instead of being evicted from memory they are loaded into a global clean-page free list and dirty-page list, from which they might be evicted later on. When a process queue is out of space, it removes one of its pages in FIFO order and adds it to the global clean or dirty queue depending on whether the page was dirty or not (if it was ever written to). If another process Q needs a free page, it takes the first free page off of the global clean list. However, if a page is referenced from memory, and it is present in the global queues, the appropriate process reclaims it (enqueues) in its queue. The bigger these global second-chance lists are, the closer the VMS algorithm performs to LRU.

For the experiments we ran all three replacement policies in all four traces using frames sizes from 4 to 4096 (or 512 for bzip.trace) which corresponds to cache size of 16 KB to 16384 KB (16 MB). We increment the frame size in powers of 2 for two reasons: to make it easier to calculate the cache size needed and because increments of 1 or some other random number seemed inefficient. As the results will show, choosing these increments clearly shows when the maximum efficiency is achieved. To calculate the cache size, we multiply the number of frames by 4 since each frame (page) is 4 KB in size per the assignment instructions. The results were recorded and the annexed graphs show the hit rate vs cache size and the amount of reads and writes of each replacement policy per trace file. The hit rate is calculated by the formula:

$$hit\ rate = \frac{1000000 - (reads + writes)}{1000000} * 100$$

Where *reads* is the number of reads from disk and *writes* is the number of writes to disk. The hit rate is displayed as a percentage, and the size of each trace file was shown to be of 1 million entries. This hit rate is hence the number of page hits, a page hit is anything that is not a read or a write to disk. Every time a page is loaded into memory the *reads* increase, and when a page is evicted from memory and it was ever written the *writes* to disk increase.

To improve upon the efficiency of our LRU replacement algorithm, we implemented a HashMap that maintains a mapping from page number of the table to a pointer of the place in the LRU queue where this page is located. The LRU queue is implemented on a double linked list, and this in combination with the hash map made it trivial to move a node on the queue to the front while updating the rest of the pointers. The other two replacement policies are implemented in a standard way also using the double linked list based queue.

RESULTS:

After testing the three algorithms of choice we notice that as the frame size increases the disk access (reads or writes) decreases drastically. This is because having fewer frames will increase the number of page faults because of the lower freedom in replacement choice, in other words, the algorithms don't have enough pages to choose from. On the other hand, having to many pages becomes expensive in terms of memory.

Analyzing the Hit rate vs Cache Size, we can see that except for the bzip.trace, all other traces need at least 2 MB of memory to maximize the hit rate. Most contemporary memories are in the size of 8 GB (8192 MB) but taking into consideration how many other processes use memory and the fact that a single process that handles this trace needs 2 MB of memory, it becomes evident that this size of memory is a little excessive. All algorithms in all traces perform poorly with low cache size (< 64 KB), but as the cache size increases the algorithms need to replace less pages, hence more hits are achieved. The bzip.trace appeared to be an exception. This trace achieves it maximum hit rate at 512 KB under all replacement policies. We believe this is partly due to the nature of the traces it contains: probably the virtual page number of those addresses are actually a limited number, meaning that although there is one million traces, most of the traces reference the same page multiple times for a read or write. Due to this nature, the maximum hit rate is easier to achieve as the replacement policies won't be called as often to evict a page.

Overall the LRU algorithm seemed to perform the best. It achieves a higher Hit Rate than all other policies with lower cache size. However, in some traces as in bzip.trace and gcc.trace, VMS was slightly better than LRU. VMS is supposed to come really close to the performance of LRU as the cache size goes up, but the fact that it outperforms LRU in some traces indicates that

these traces contains addresses that are referenced really often. Since VMS maintains two additional global queues (clean and dirty) to keep pages that should be evicted, it reduces the number of evictions as long as these pages are referenced multiple times. LRU will evict a page before VMS even if the page is referenced too often, as LRU does not give a "second chance" to the page and waits for it to be referenced again. For these reasons, LRU performs really well on swim.trace and sixpack.trace. Our intuition is that these files have a larger number of unique VPN, which decreases the chances of a hit and increases the number of evictions for a new page.

FIFO has the overall poorest performance (as expected) since it evicts the first page to arrive when a replacement is needed but does not account for a page that was recently referenced. With FIFO, pages in the middle of the queue might be referenced but will be treated as "old" eventually and be evicted, when in reality the chances of the page being referenced again increased.

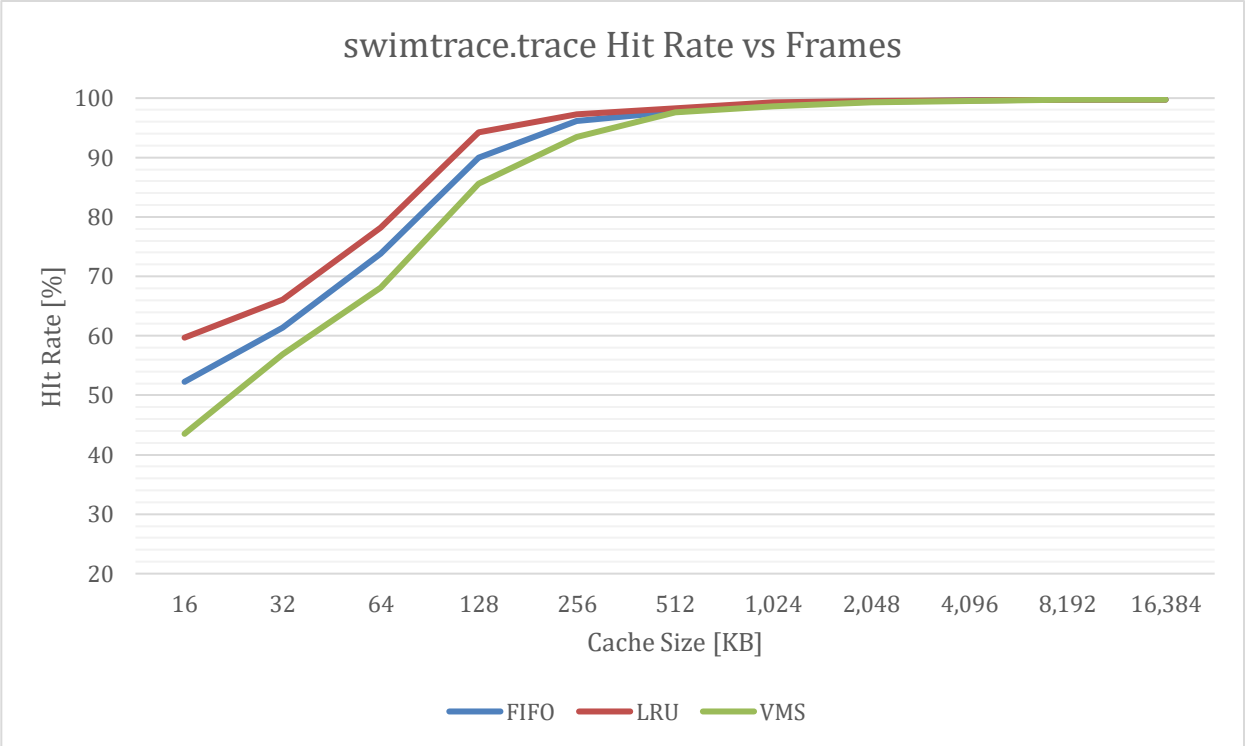
CONCLUSION:

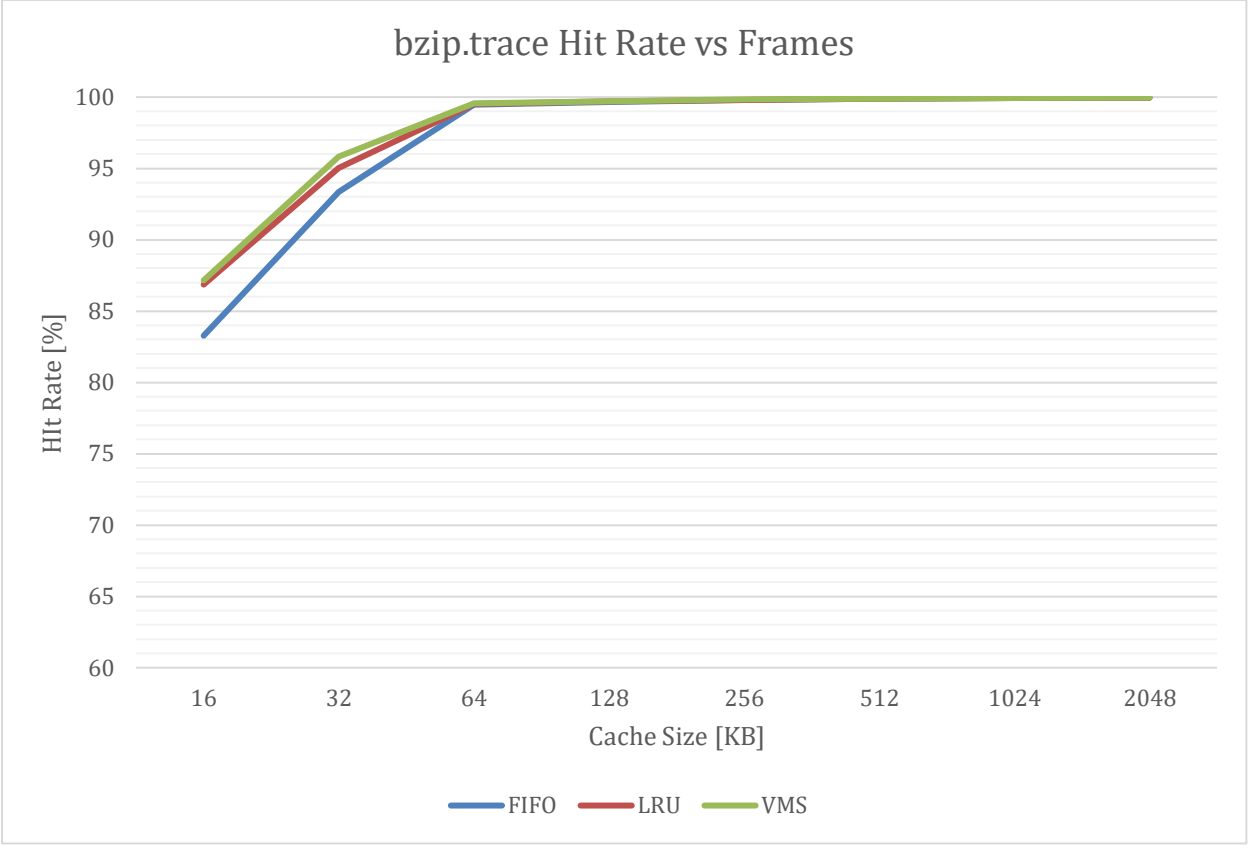
Throughout this project we have used various page replacement algorithms like FIFO, LRU, and VMS Second Chance then simulated and compared them to evaluate their efficiency. Upon analyzing the results, it becomes evident that LRU performs best in most situations. The situations that favor LRU are those were not a lot of page hits will happen even if the memory was big enough to fit the memory space. VMS favors from the opposite, having traces that reference the same page number multiple times. FIFO will achieve a performance similar to VMS and LRU if the pages being referenced maintain the same FIFO order as in the FIFO queue, in other words, if pages that are referenced after being loaded are currently at the end fo the queue (the newest in the queue).

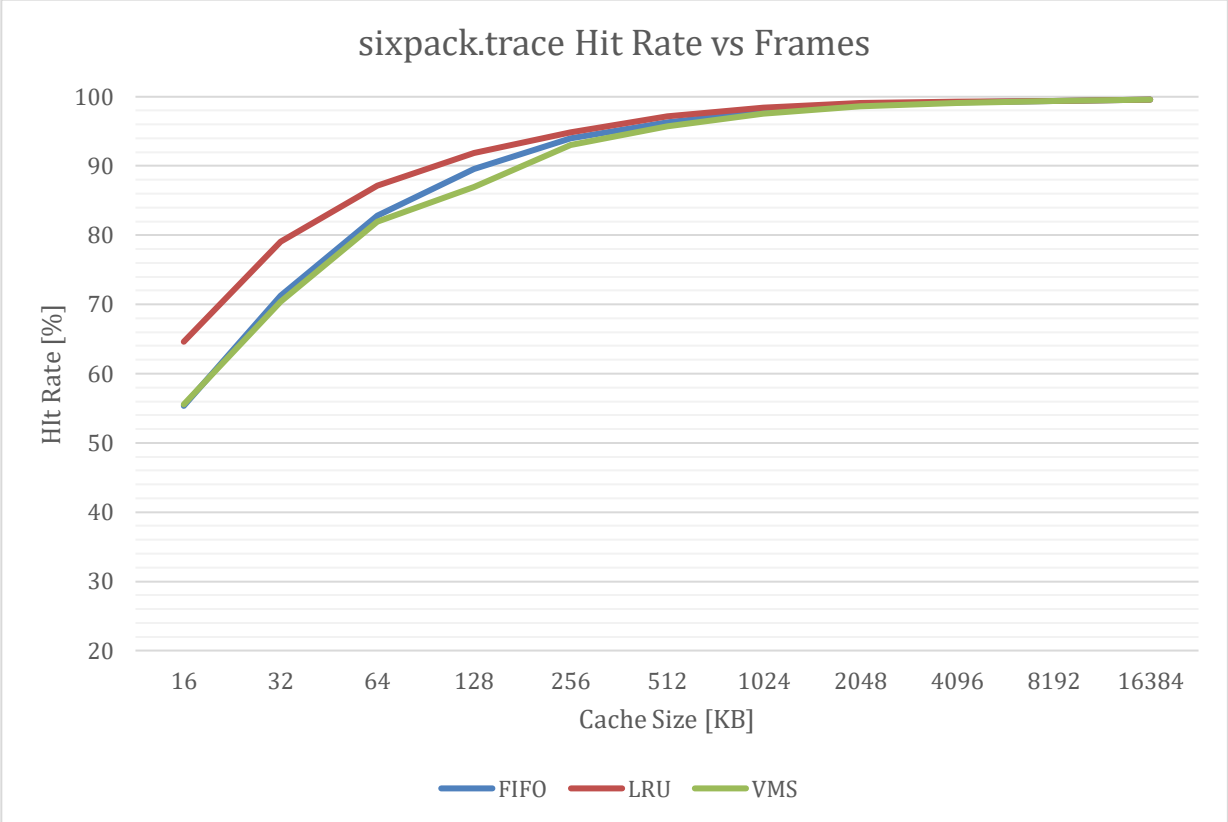
Based on the number of reads and writes for each replacement policy, the bzip.trace seems to have the higher chances of referencing the same page, whereas the swim.trace and sixpack.trace contain a lot of unique virtual page numbers that are not referenced as often. Therefore, LRU performed really well on swim.trace and sixpack.trace, but VMS outperformed LRU on bzip.trace. The higher the memory space (cache size), the higher the hit rate all algorithms achieved. For all traces, 2 MB was enough to achieve near 100% hit rate, and 512 KB was enough to achieve 100% in the case of bzip.trace.

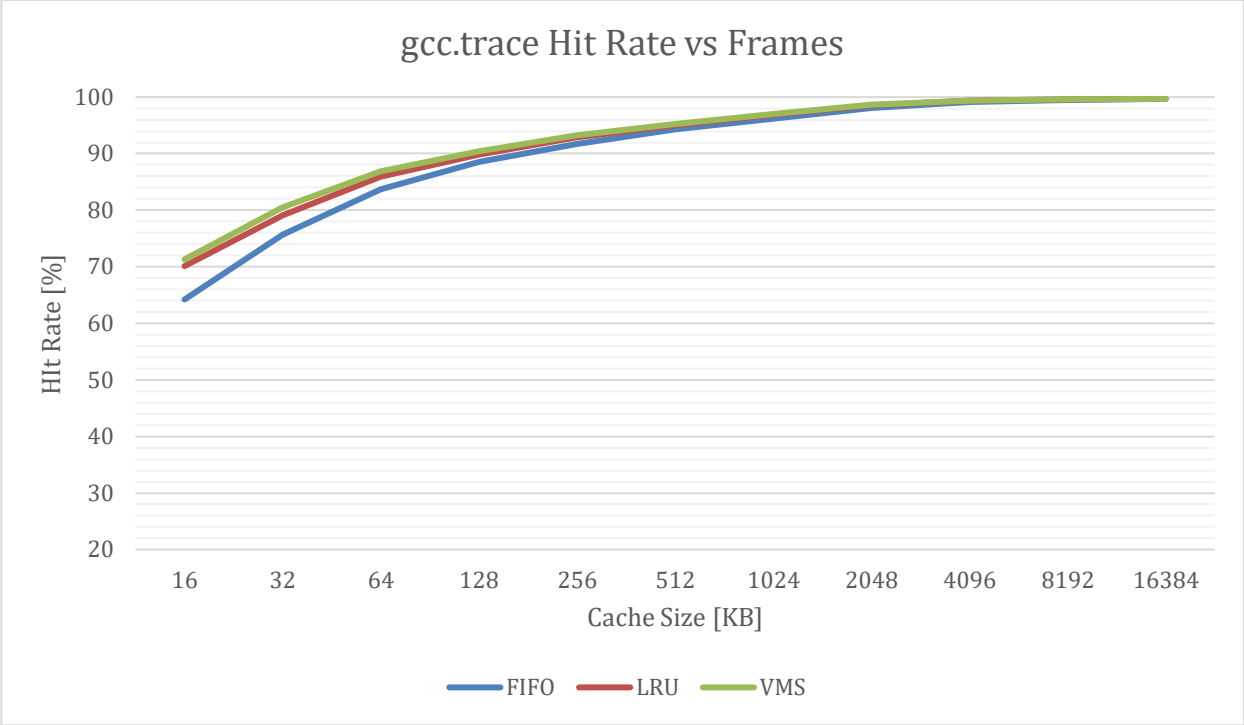
We have seen that out of all three replacement policies, FIFO performs the worst. VMS, depending on the situation, can outperform or come close to LRU, as indicated in the hit rate vs cache size for all traces. Since we can't know the nature of the traces before execution, we concluded that the safest bet is to use LRU as it achieved a high hit rate on average.

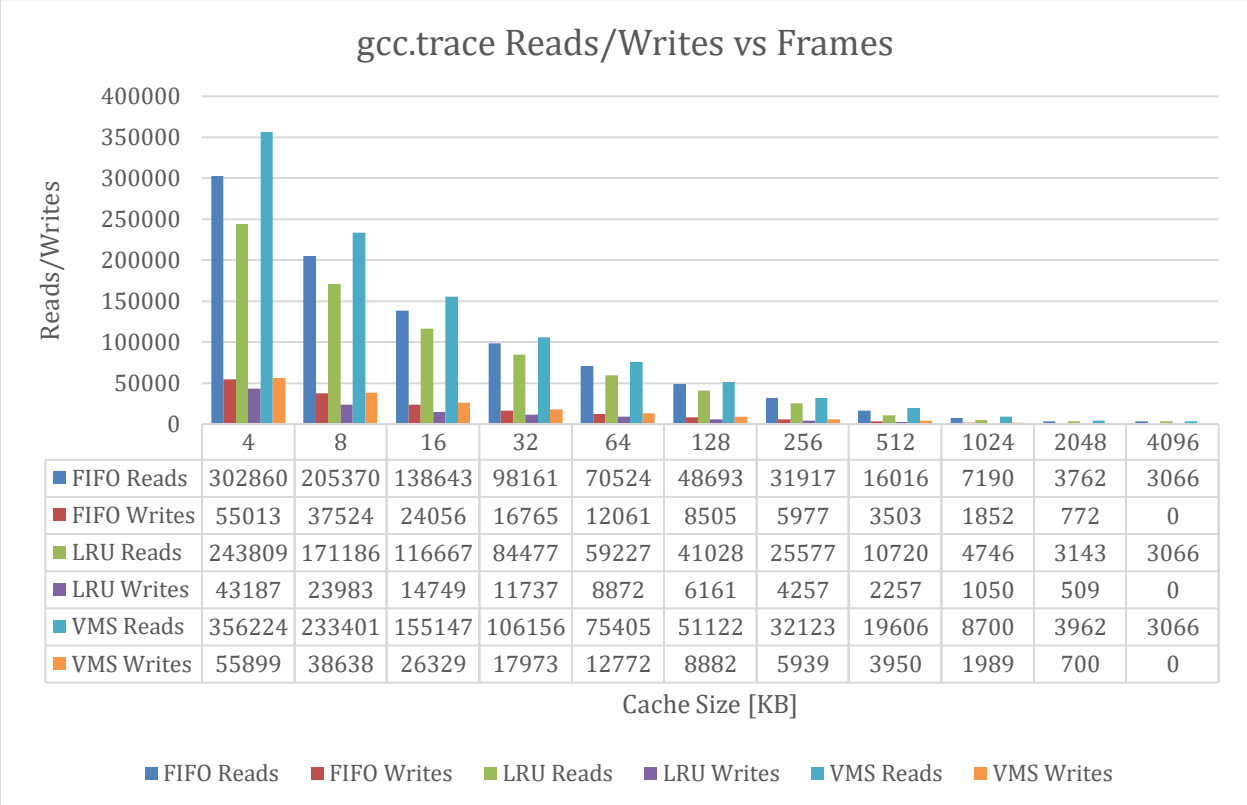
ANNEX:



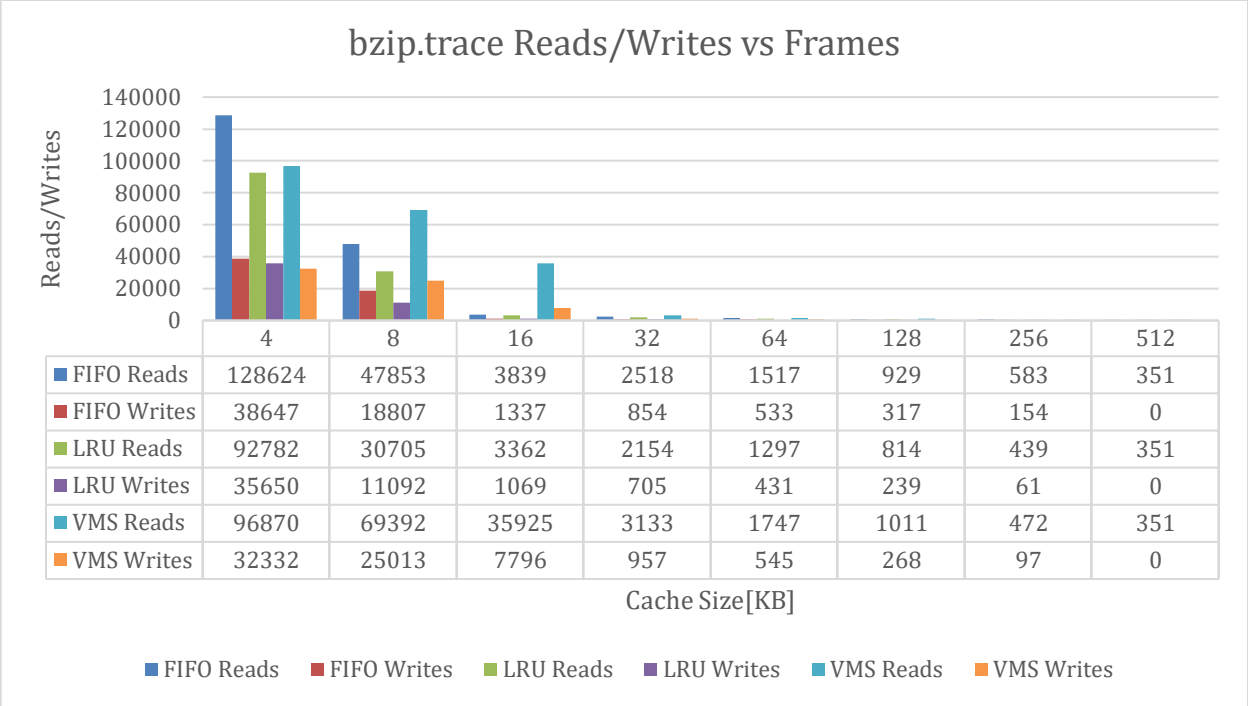

bzip.trace Hit Rate vs Frames
HIt Rate [%]
100
95
90
85
80
75
70
65
60
16
32
64
128
256
512
1024
2048
Cache Size [KB]
FIFO
LRU
VMS


sixpack.trace Hit Rate vs Frames
HIt Rate [%]
100
90
80
70
60
50
40
30
20
16
32
64
128
256
512
1024
2048
4096
8192
16384
Cache Size [KB]
FIFO
LRU
VMS

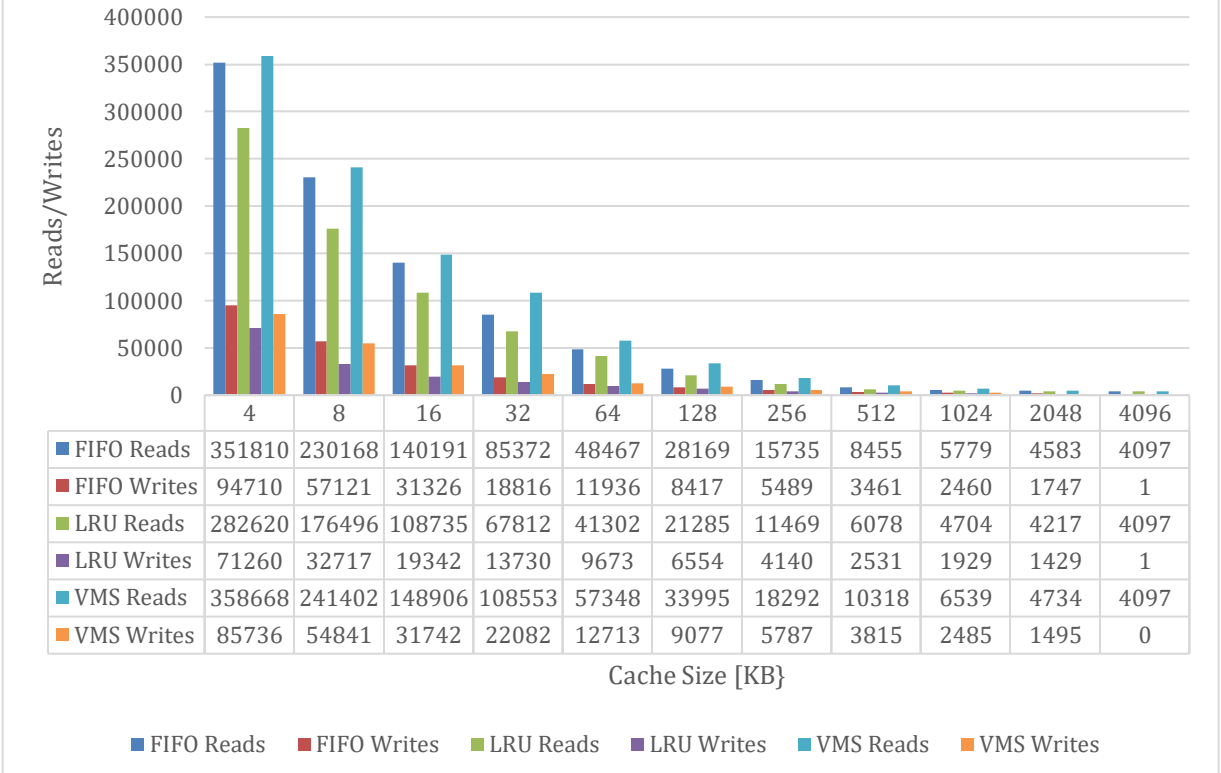




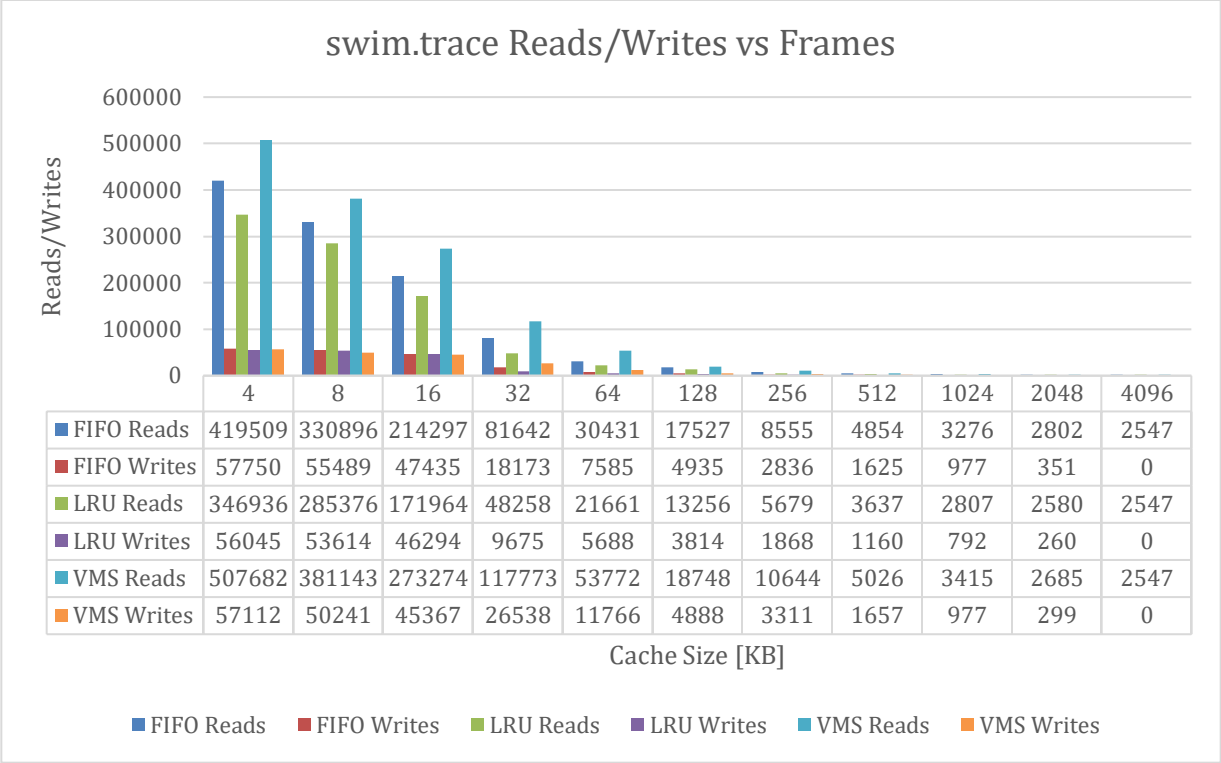
	4	8	16	32	64	128	256	512	1024	2048	4096
FIFO Reads	302860	205370	138643	98161	70524	48693	31917	16016	7190	3762	3066
FIFO Writes	55013	37524	24056	16765	12061	8505	5977	3503	1852	772	0
LRU Reads	243809	171186	116667	84477	59227	41028	25577	10720	4746	3143	3066
LRU Writes	43187	23983	14749	11737	8872	6161	4257	2257	1050	509	0
VMS Reads	356224	233401	155147	106156	75405	51122	32123	19606	8700	3962	3066
VMS Writes	55899	38638	26329	17973	12772	8882	5939	3950	1989	700	0



	4	8	16	32	64	128	256	512
FIFO Reads	128624	47853	3839	2518	1517	929	583	351
FIFO Writes	38647	18807	1337	854	533	317	154	0
LRU Reads	92782	30705	3362	2154	1297	814	439	351
LRU Writes	35650	11092	1069	705	431	239	61	0
VMS Reads	96870	69392	35925	3133	1747	1011	472	351
VMS Writes	32332	25013	7796	957	545	268	97	0



	4	8	16	32	64	128	256	512	1024	2048	4096
FIFO Reads	351810	230168	140191	85372	48467	28169	15735	8455	5779	4583	4097
FIFO Writes	94710	57121	31326	18816	11936	8417	5489	3461	2460	1747	1
LRU Reads	282620	176496	108735	67812	41302	21285	11469	6078	4704	4217	4097
LRU Writes	71260	32717	19342	13730	9673	6554	4140	2531	1929	1429	1
VMS Reads	358668	241402	148906	108553	57348	33995	18292	10318	6539	4734	4097
VMS Writes	85736	54841	31742	22082	12713	9077	5787	3815	2485	1495	0



	4	8	16	32	64	128	256	512	1024	2048	4096
FIFO Reads	419509	330896	214297	81642	30431	17527	8555	4854	3276	2802	2547
FIFO Writes	57750	55489	47435	18173	7585	4935	2836	1625	977	351	0
LRU Reads	346936	285376	171964	48258	21661	13256	5679	3637	2807	2580	2547
LRU Writes	56045	53614	46294	9675	5688	3814	1868	1160	792	260	0
VMS Reads	507682	381143	273274	117773	53772	18748	10644	5026	3415	2685	2547
VMS Writes	57112	50241	45367	26538	11766	4888	3311	1657	977	299	0