# T1 (Changing Frame Count)

## *Description:*

The purpose of this test is to measure the effect of increasing the number of global frames on various test results such as cpu utilization, context switches, total page faults, and average time to completion. The programs used in this test sweep through a large set of data in-order doing reads then sweep back through doing writes. While there are many page faults on the initial sweep, with more global frames the faults can prevented the second time through.

In the worst case, all pages will fault again on the second sweep causing a doubling in the lowest possible number of faults. As you can see in the results, with very few frames the fault rate is quite high. As the frame count increases the total faults at first drops very rapidly and then down to its lowest possible with 80 frames. With 80 frames, every frame for every process can be held in memory. All the "large_dataset.trace" programs sweep over memory on 15 pages and the "good_locality.trace" program accesses data on 5 pages with good locality so given enough frames it should only fault 5 times.

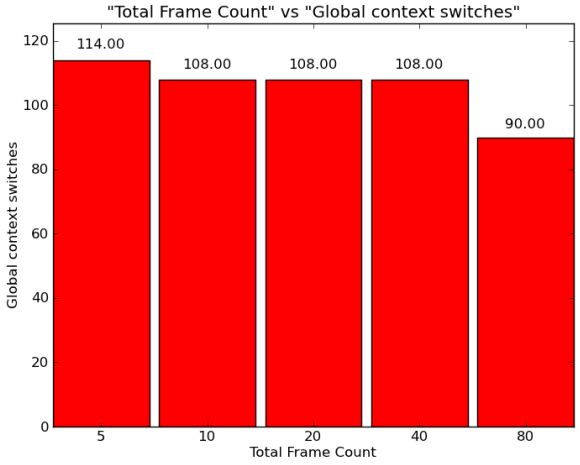
Therefore, given enough frames the fault count is 15 * 3 + 5 = 50 and that is what we see in the result graph. This is the lowest possible without some mechanism for sharing pages.

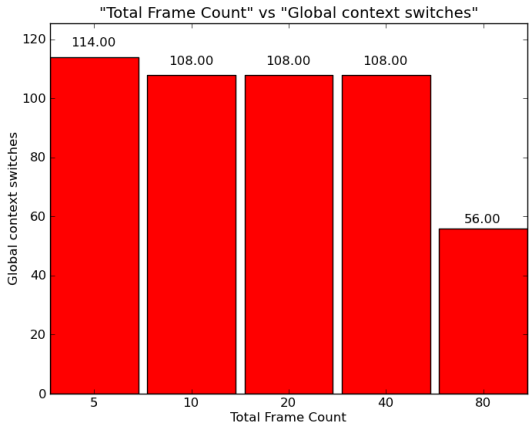
Other things to note:
Since our time keeping (virtual counter) is not the finest grain, the cpu utilization measurements aren't the best but you can still see that with a decreasing number of global page faults the cpu utilization was increased.

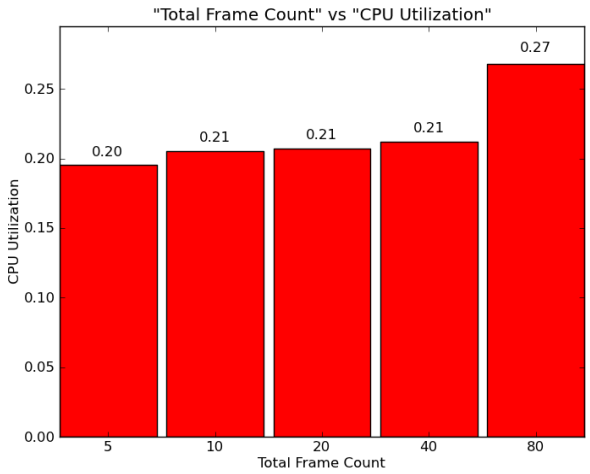
For context switches, with lower frame counts the number of context switches was controlled more directly by the number page faults since processes never finished their full quanta. When the frame count got large enough (80) the number of context switches was actually increased by the small quanta size. If you look at the same graph but with the quanta size modified to be 20 you can see that up until the frame count is substantially large it is still dependent on the fault rate. Once we get to 80 with a quanta of 20 the context switches drop dramatically. Also, with a quanta of 20 we also see a large cpu utilization increase at frame count 80.

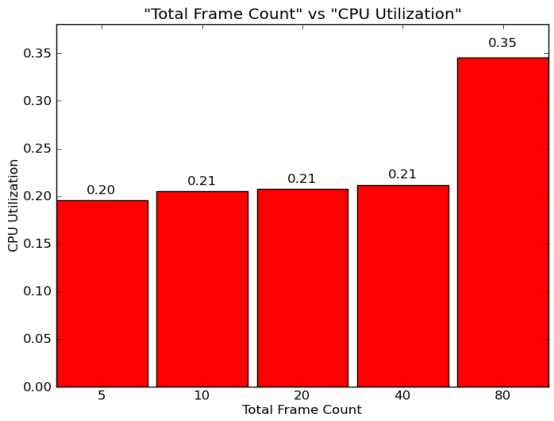
## *Graphs:*

These two graphs show the global context switch count with respect to the global frame count. The left is for the original test and the left is with an increased quanta as described above.



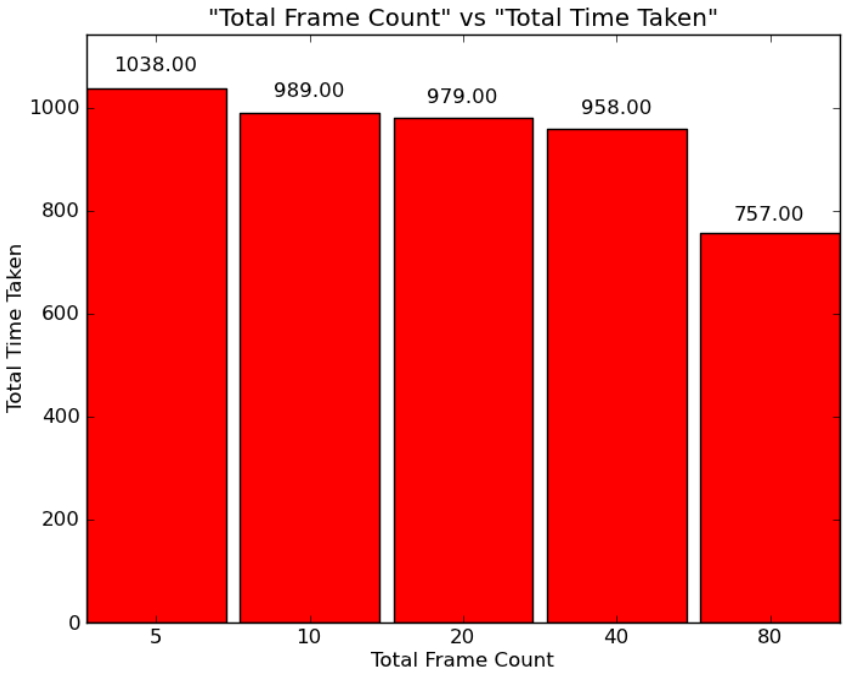



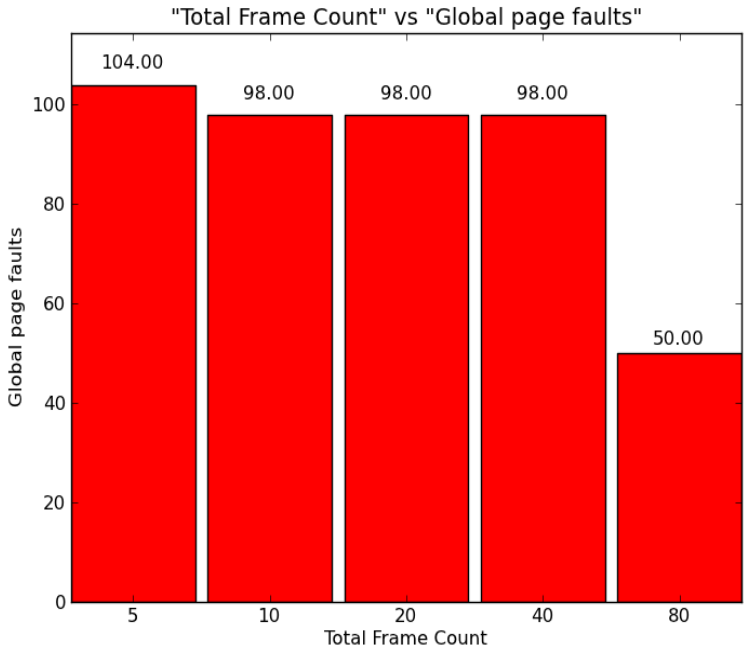


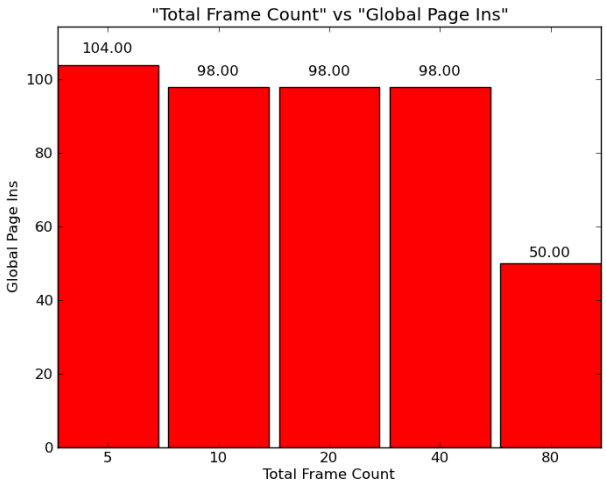



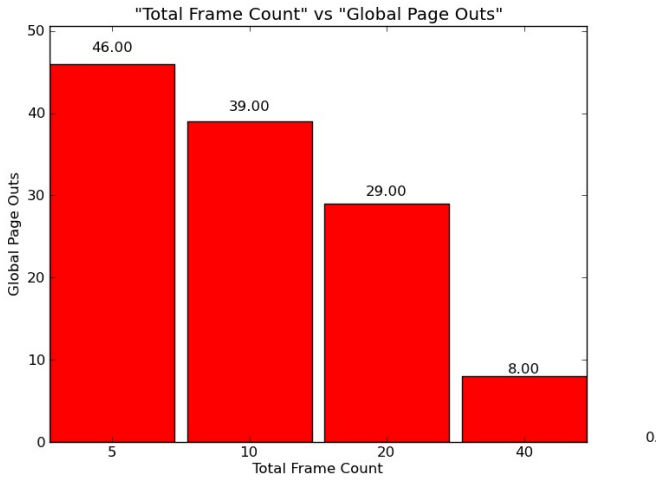

(Above) On the left is the CPU utilization for the original test and on the left is with an increased quanta. As described above you can see that at larger frame counts the efficiency of the execution is limited more by the small quanta than page fault rates.

The following are graphs for the original test that we did not create for the increased quanta. These graphs are analyzed above.






# T2 (Increasing Page Size)

## *Description:*

This test is intended to show the results of increasing the page size. As the size of the pages increases, the number of frames goes down to keep the same amount of total memory. The traces used for this test "large_dataset.trace" have good spatial locality but not the best temporal locality because they re-sweep through the large dataset and their original pages may have been removed.

No matter what each process faults the first time and then the larger the page that is brought in the more instructions that can be executed on the next quanta. When the page size gets large enough then context switch count is more affected by the quanta.

If we were to scale page-in/page-out I/O time based on page size then we would see that a larger page size would benefit these processes because of their spatial locality but it would severely penalize processes with random accesses. We should not that we realize that increasing page size is not beneficial in all scenarios but it is for this set of traces. To show that increasing page size has not affect or in fact is worse check the test T2Rand.

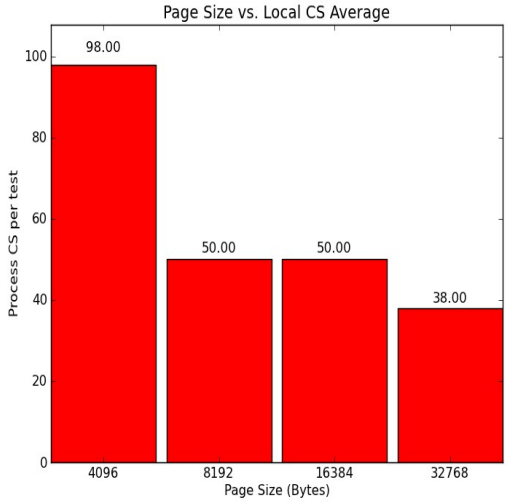
## *Results:*

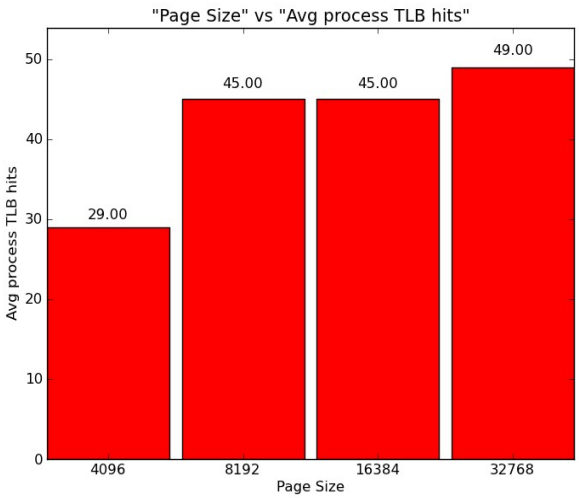
All the results are as expected. As the page size increases we see a decrease in the number of average context switches. With a small page size the processes were faulting every 2 instructions but as the page size increased the small execution quanta played a larger role in the number of page faults.

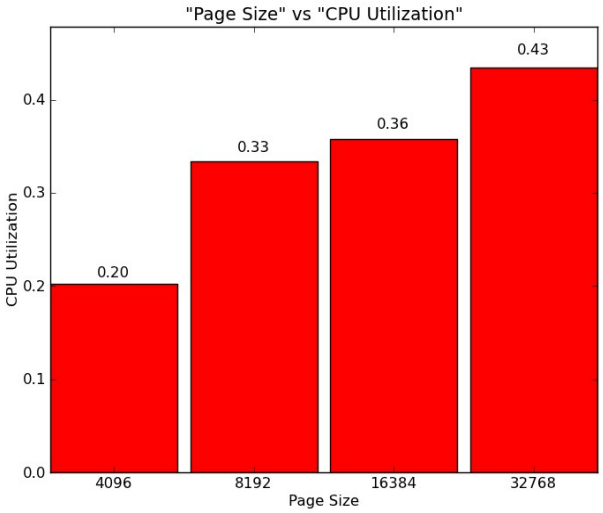
As expected the number of page faults decreased with an increased page size. Given the spatial locality of the processes, bringing a larger page in is more beneficial than having many frames in memory (with respect to page fault rate). Many smaller pages may allow the process to keep more pages in memory but it will have more faults to get them in.

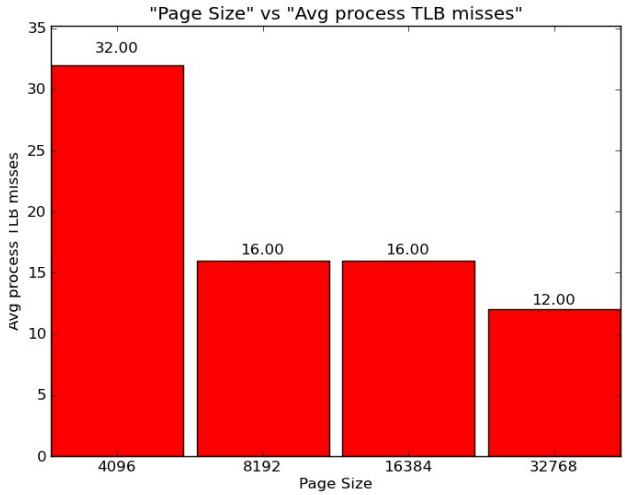
Most of the other results follow intuitively from these such as the average tlb hits increasing with the page size and the tlb misses decreasing. Also, as a direct result of less context switching and less page faulting the cpu utilization increased with page size.

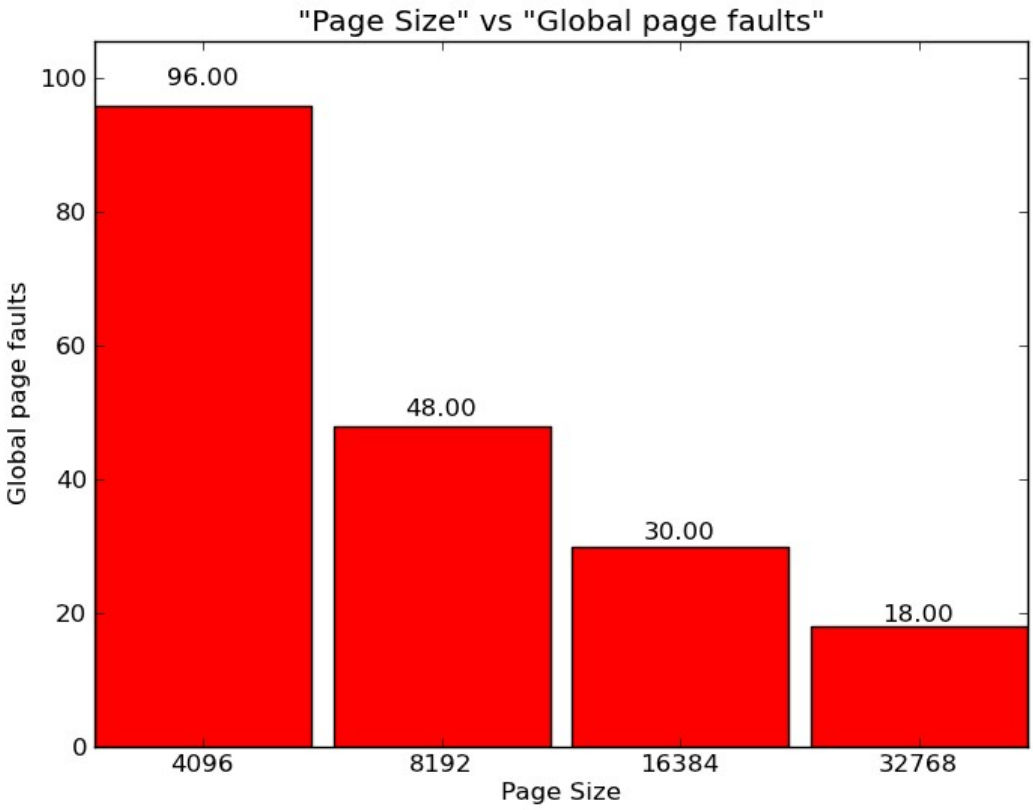
## *Graphs:*






# T2Rand (Increase PS with random trace file)

## *Description:*

This is meant to show why the choice of processes in T2 was important. Given processes with very random accesses, increasing the page size has no effect because the next access is likely to not be

anywhere close.

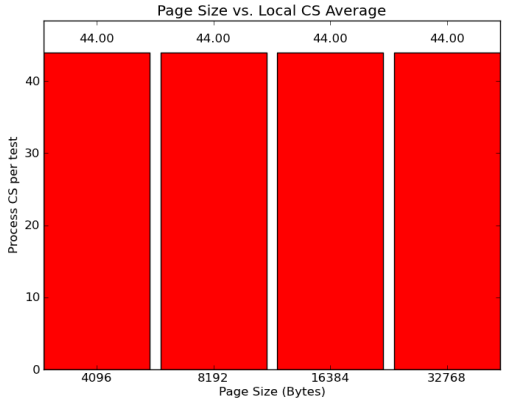
All graphs for context switches, fault rate, tlb hits/misses all were the same across page sizes. The only metric that slightly changed is cpu utilization which dropped with a larger page size. Since the fault and cs rates stayed the same this must have come from an extra I/O operation. If we were measuring I/O request and service times relative to the size of the page we would see even more drastic changes in the utilization and overall process execution time. If we spend a lot of time to bring in a 32k page only to access it once then we wasted resources.

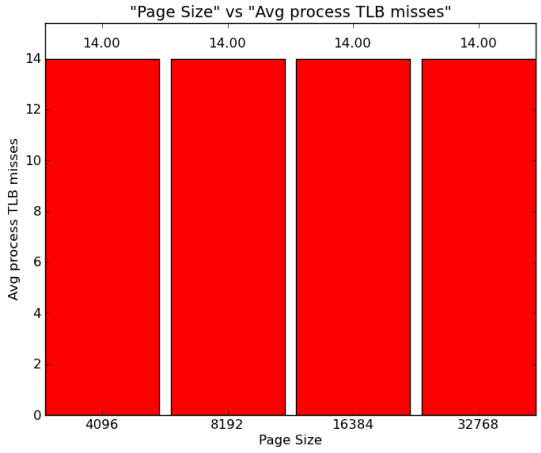
## *Results:*

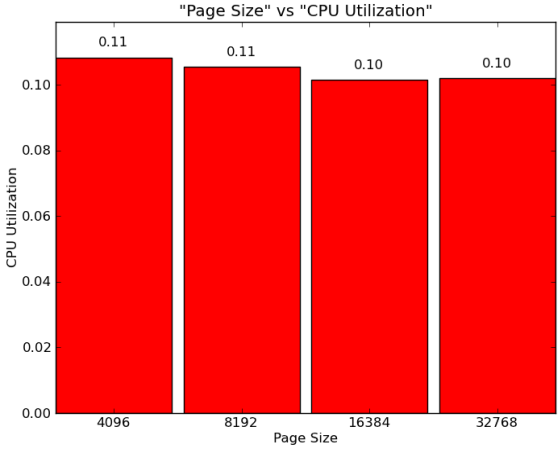
As expected, with a **very** random trace the effect of increased page size does not help the system and as mentioned above it should be worse in a real system because of the added overhead in bringing in a page that is substantially larger.

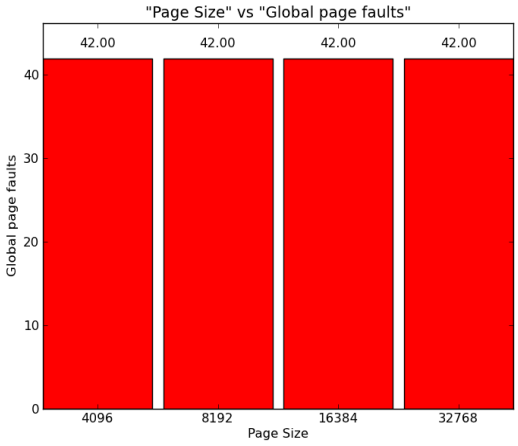
## *Graphs:*

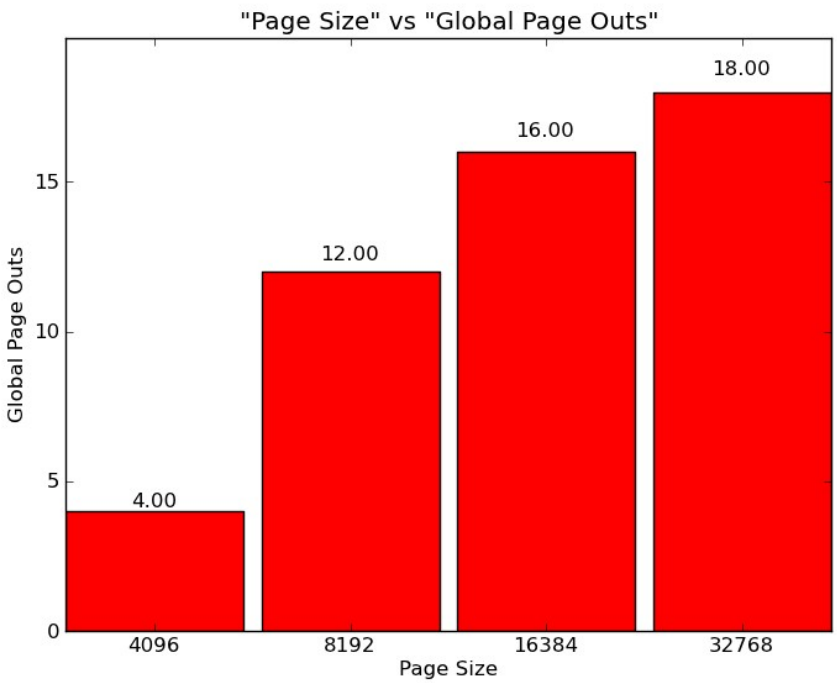
There really isn't much to look at in these and the reasons for these results have already been noted.






In the test for T2 rand we saw that the cpu utilization decreased slightly with page size for this random trace. We originally speculated that this was due to an extra I/O because all of the context switches and page faults remained the same. Here we see that the number of page outs increased dramatically. Although the project description says we have unlimited controller bandwidth on the disk we implemented it so that page-ins had to wait for the previous page out to complete. Despite the increased page-outs the cpu utilization did not increase nearly as much because the overhead in all the other processes page faulting generally masked out the extra I/O time.

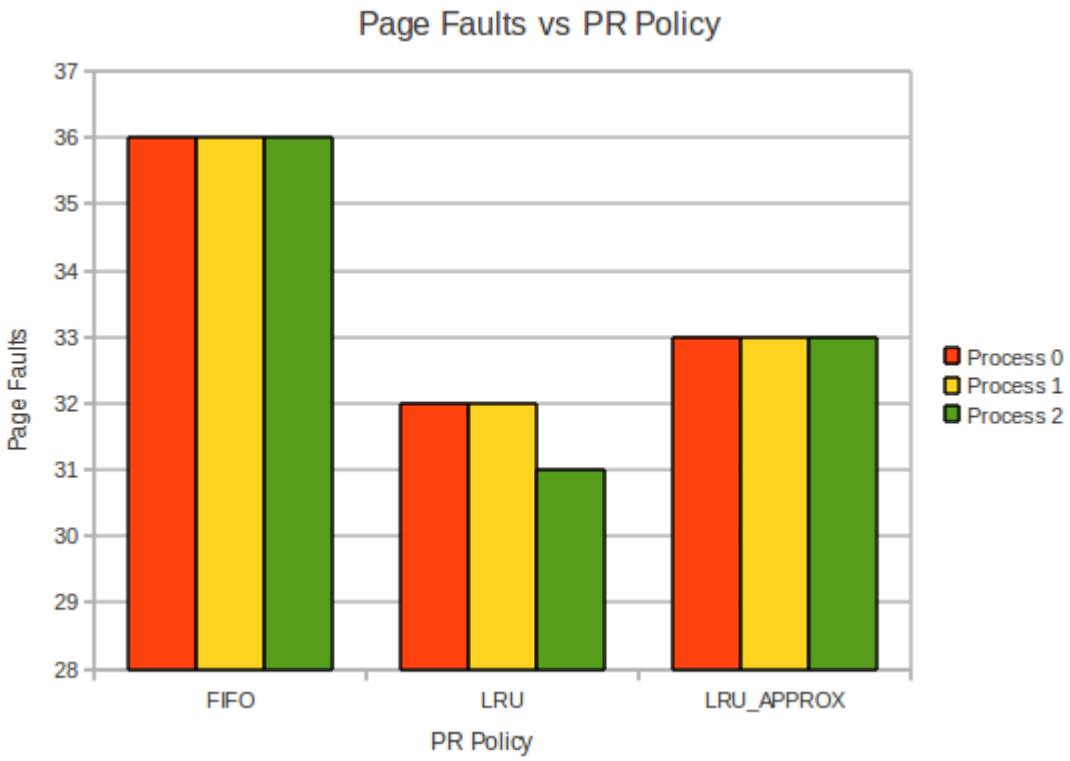
# T3 (Comparison of PR Policies)

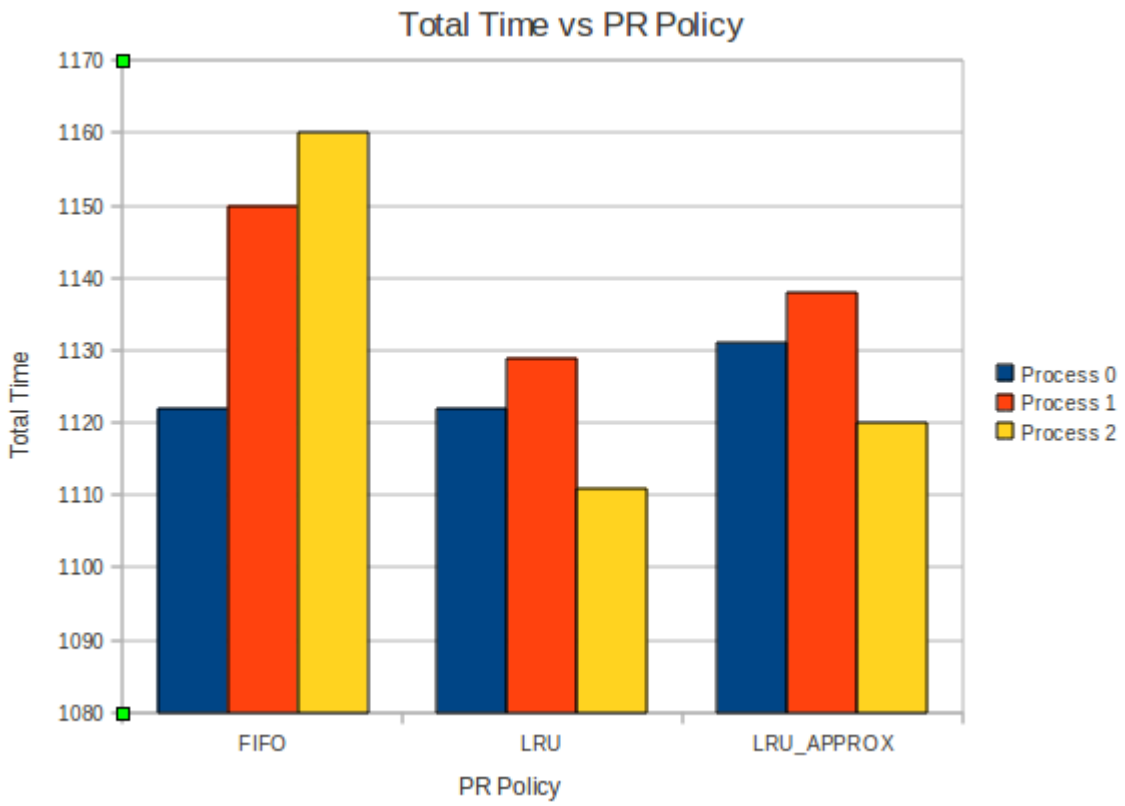
## *Description:*

This test runs each of the Page Replacement Policies against each other on the same settings. A trace file, large_dataset_badfifo.trace, was specifically chosen so that FIFO would perform badly. Each of the three processes runs this trace. The trace has a global variable that is read from frequently. FIFO will dispose of the page containing this global variable since the page is loaded at the beginning whereas LRU and LRU_APPROX will keep it around since it is used often.

The total time for a process is the least with LRU, then LRU_APPROX, and then finally FIFO. It follows then, that using LRU or LRU_APPROX will lead to higher CPU utilization. In addition the number of page faults for each process is lowest with LRU, then LRU_APPROX, and then followed by FIFO. As we expected, LRU approximate fell between the highly optimal pure LRU and the naïve FIFO.

***Graphs:***


Page Faults vs PR Policy
37
36
35
34
33
32
31
30
29
28
Page Faults
FIFO
LRU
LRU_APPROX
PR Policy
Process 0
Process 1
Process 2

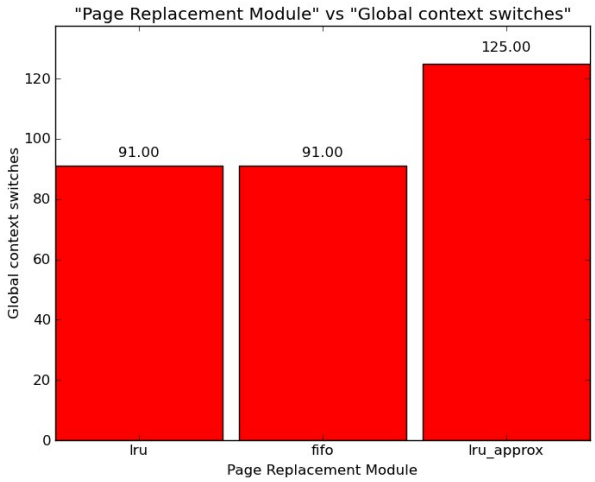


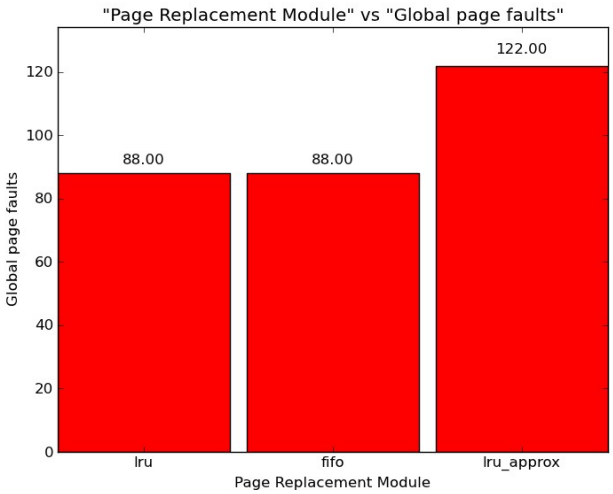

# BadLRU2 (LRU Approx worst scenario)

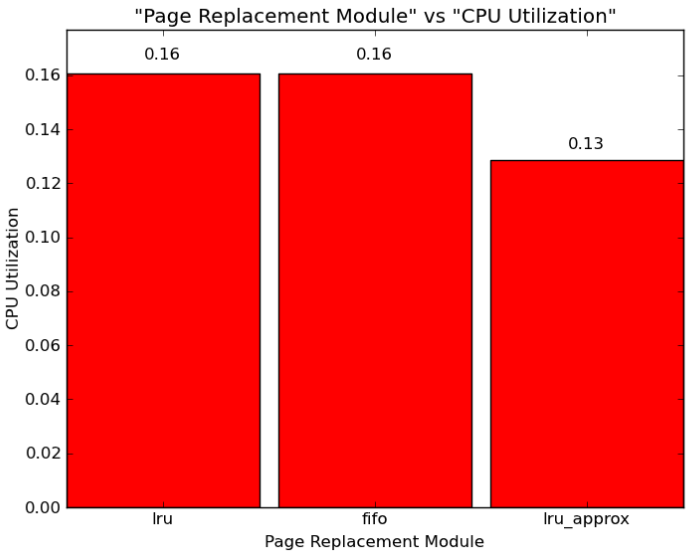
## *Description:*

The purpose of this test is to show when LRU approximate can be worse than both pure LRU and FIFO. The large_dataset.trace program goes through 15 pages in total twice over so keeping its frames in memory is very important. Comparatively, badlru.trace only has 4 pages. We staggered the execution of large_dataset and badlru.trace so that if the page replacement policy made a bad decision and replaced pages in large_dataset it would incur a larger penalty. The trace badlru.trace is made so that after its initial faults, the order it comes back to reference its pages is just after they will usually be replaced given the number of global frames, 8, which is small for the number of pages that need to be dealt with. With lru approximate this results in a back and forth faulting scenario between the badlru traces and the large_dataset traces.

### *Graphs:*




# BadLRU2 (LRU worse than FIFO)

### *Description:*

In this test case we are showing how LRU, both pure and approx, can be worse than FIFO.

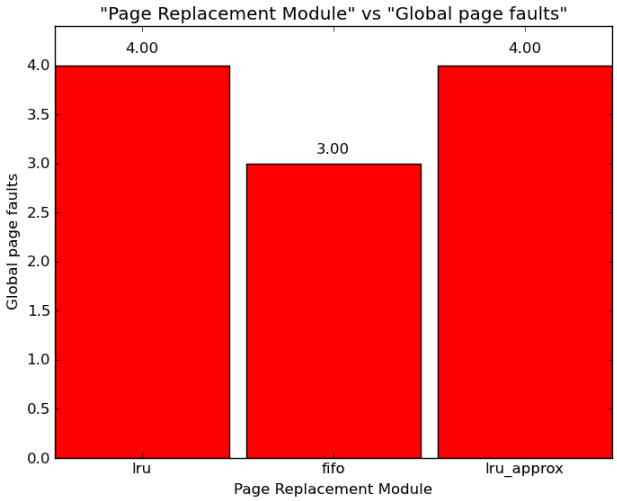
Initial access pattern: 0x0, 0x1000, 0x0, 0x2000

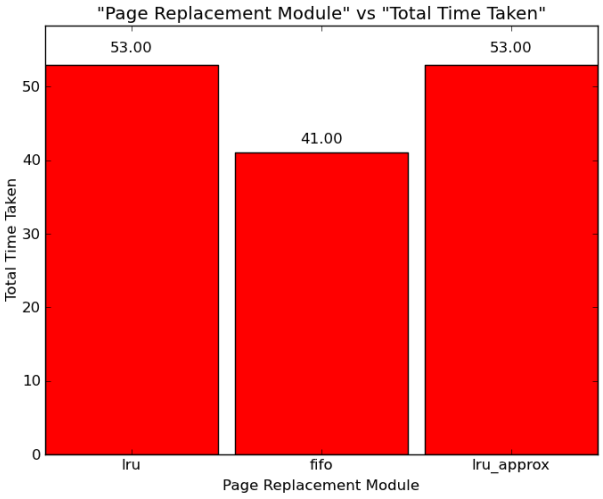
Let's look at what the LRU and FIFO caches look like before we make another access.

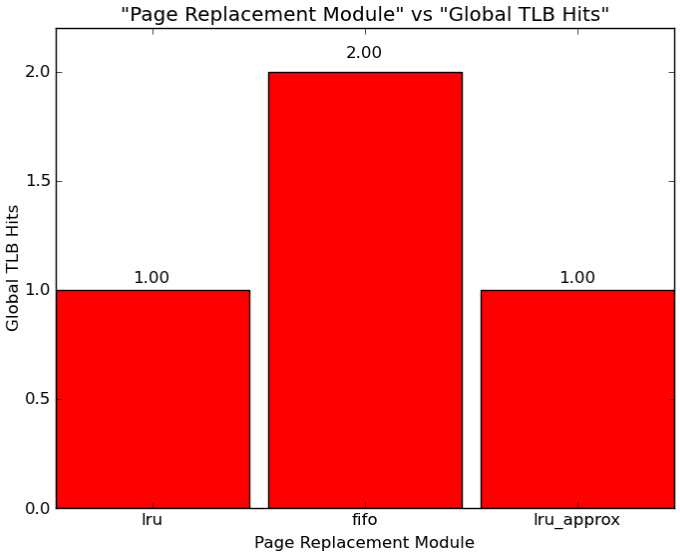
LRU contains "0x0, 0x2000" and FIFO contains "0x1000, 0x2000". If the next page access is a "0x1000", LRU will page fault and therefore perform worse than FIFO. If the next page

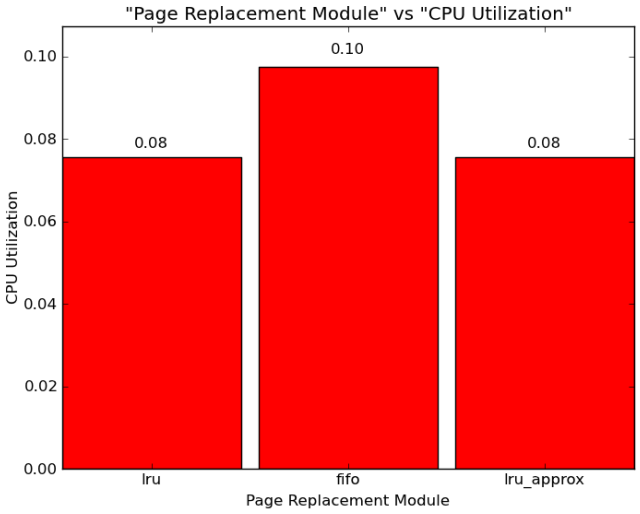
access is an "0x0", then LRU will perform better. If the next page access is a "0x2000", they will perform equally as well. In our test the next access is "0x1000", showing that LRU can be worse than FIFO is some cases. Since LRU doesn't look ahead it ends up hurting itself in this pattern.

## *Graphs:*



## LowMem (Low memory behavior test)

### *Description:*

These tests are intended to fail if run. The VMM should detect a circular (cascading) page fault scenario and warn the user. If the setting ignore_circular_fault_warn is set to false this will exit the program. This is the default action.

The python script executing the test should pause at this point so the user can see the message.

## LowMem2

### *Description:*

Test that runs each Page Replacement Policy with only 2 global frames. In this scenario because there are more than 2 processes but only 2 frames the 3rd page fault will be prevented from getting a frame because they are both pinned. In this case we do nothing with the process and allow it to page fault on its next quanta and hope that there are open frames. Worst case the first two processes continue to fault on every instruction and all processes after them must wait for them to complete before getting any cpu time.

## TCleanD

### *Description:*

Tests the cleaning daemon of each Page Replacement Policy.  Look at the trace files to see that the daemon is running correctly.

## TDecAddr

### *Description:*

This test is just to make sure the VMM can read in addresses in base 10. In order for this to work it must be specified in the settings file.