Page Replacement Policy Comparison: GCA vs FIFO

1. Description

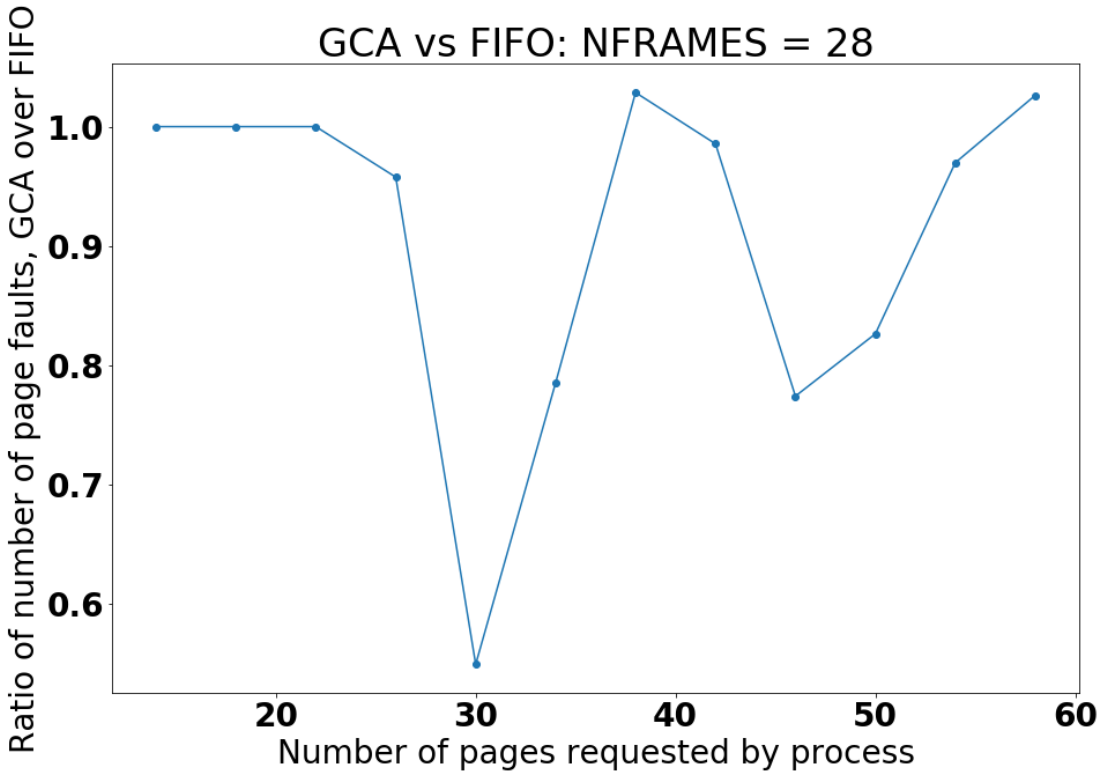
For Lab 3, two page replacement algorithms were implemented: global clock algorithm (GCA) and first in, first out (FIFO). To quantify the performance differences between the two algorithms, two metrics were used: time and number of page faults. Test code was developed to spawn various memory intensive processes that request, read, and write to a given number of pages of virtual memory.

Each of these processes requested virtual memory for four different regions of memory that are used with varying frequency. Each time the memory is used, the process alternates between reading from or writing to the memory.

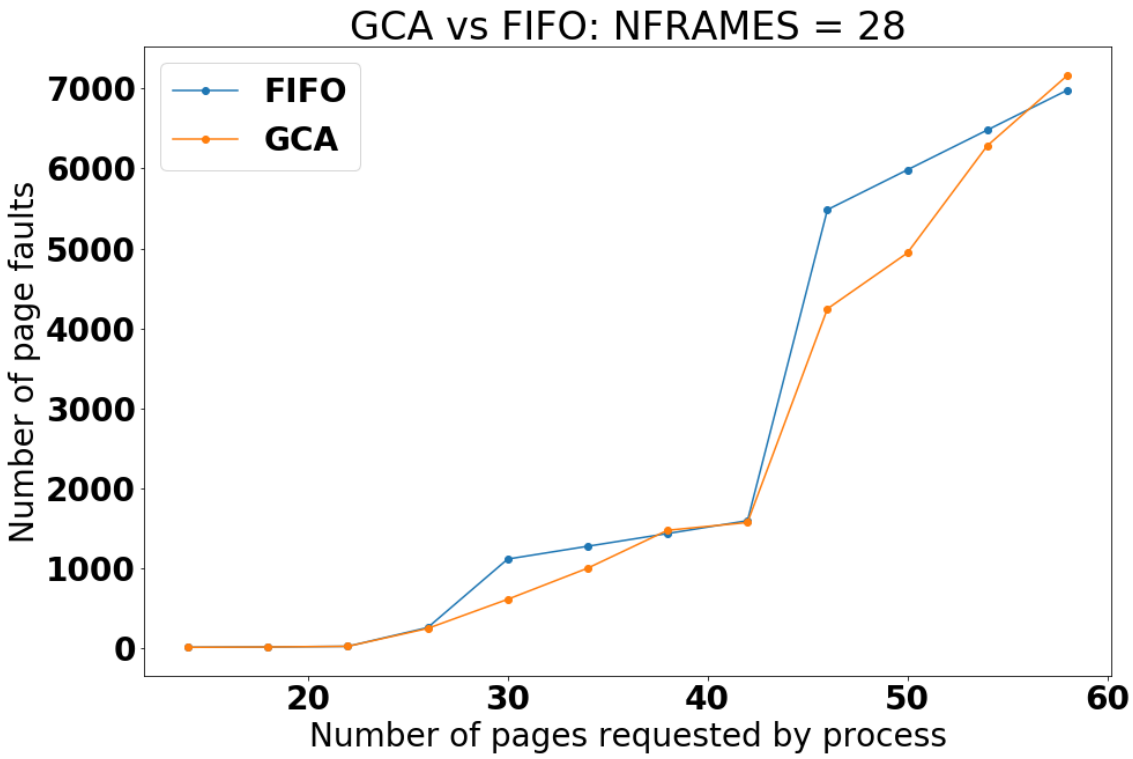
This test code was run under GCA and FIFO for various request sizes of virtual memory (*numPages* in the test code shown in Section 3). The number of page faults and time to completion was plotted as a function of the request size. All results were taken with a small number of frames (NFRAMES = 28) to speed up testing.

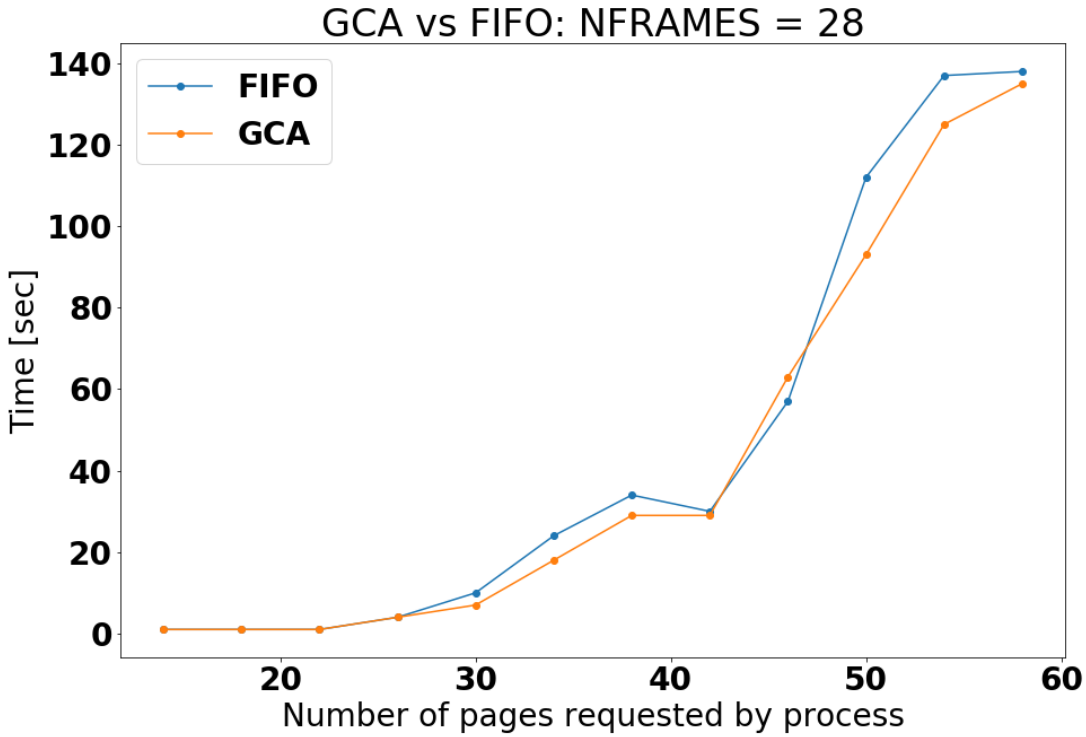
2. Results

The first plot below shows that the improvement that GCA offers over FIFO is dependent on the number of pages that the process requests. For low numbers of pages, the two algorithms perform similarly because the system rarely needs to evict frames. For requests of similar magnitude to the number of frames in the system, GCA gives significant improvement over FIFO. This



improvement tapers off at higher requests. Performance is reduced because each of the four regions of memory represent a larger portion of available RAM, so fewer and fewer pages can be kept that are expected to be used in the future. As a result, the page replacement algorithm has less influence on the performance of the system.




GCA vs FIFO: NFRAMES = 28
FIFO
GCA
Time [sec]
0
20
40
60
80
100
120
140
20
30
40
50
60
Number of pages requested by process

3. *Test Code*

```
void looper(uint32 numPages, uint32 loops, sid32 s, int mulPrFlg){
    int i;
    int req = numPages/numRegions;
    uint32* ptr[numRegions];
    for( i = 0; i < numRegions; i++){
        ptr[i] = (uint32*)vgetmem(req*NBPG);
    }
    //Alternate between reading and writing memory
    char flags[numRegions] = {1,1,1,1};
    for( i = 0; i < loops; i++){
        //Frequently used memory
        if(flags[0])
            writemem(ptr[0], req, i);
        else
            readmem(ptr[0], req, i-1);
        flags[0] = !flags[0];

        //Less frequently used memory
        if(i%5 == 0){
            if(flags[1])
                writemem(ptr[1], req, i);
            else
                readmem(ptr[1], req, i-5);
            flags[1] = !flags[1];
        }

        // Rarely used memory
        if(i%23 == 0){
            if(flags[2])
                writemem(ptr[2], req, i);
            else
                readmem(ptr[2], req, i-23);
            flags[2] = !flags[2];
        }

        // Almost never used memory
        if(i%107 == 0){
            if(flags[3])
                writemem(ptr[3], req, i);
            else
                readmem(ptr[3], req, i-107);
            flags[3] = !flags[3];
        }
    }
```

```
void readmem(uint32* p, uint32 numPages, uint32 count){
    int i;
    uint32 val;
    uint32 mem;
    for(i = 0; i < numPages*NBPG; i+=4){
        val = get_test_value3(p, count);
        mem = *p;
        if(val != mem){
            panic("FAIL: Checking memory failed\n");
        }
        p++;
    }
}

void writemem(uint32* p, uint32 numPages, uint32 count){
    int i;
    for(i = 0; i < numPages*NBPG; i+=4){
        *p= get_test_value3(p, count);
        p++;
    }
}

uint32 get_test_value3(uint32 *addr, uint32 seed) {
  return (uint32)((uint32)addr & 0x0000FFFF) + (seed << 20) + (currpid << 16);
}
```

```
void test7(uint32 numPages){
    kprintf("\n================= TEST 7 === ============\n");
    char* policy[5] = {"blank","blank", "blank", "FIFO", "GCA "};
    uint32 starttime = clktime;
    uint32 pfcstart = pfc;
    kprintf("START: POLICY = %s, NFRAMES = %d, NUMPAGES, = %d, Page Fault Count = %d, Time = %d\n",
         policy[currpolicy], NFRAMES, numPages, pfc-pfcstart, clktime-starttime);
    uint32 numLoops = 1000;
    sid32 s = semcreate(0);
    if(s == SYSERR) kprintf("FAIL: semcreate returned syserr\n");
    pid32 p1 = vcreate(looper, INITSTK, numPages, INITPRIO, "proc1", 4, numPages, numLoops, s, 0);
    resume(p1);

    if(SYSERR == wait(s))
        kprintf("FAIL: wait failed on sid %d\n", s);
    kprintf("END  : POLICY = %s, NFRAMES = %d, NUMPAGES, = %d, Page Fault Count = %d, Time = %d\n",
        policy[currpolicy], NFRAMES, numPages, pfc-pfcstart, clktime-starttime);
    kprintf("================ END OF TEST 7 ============\n");
}
```