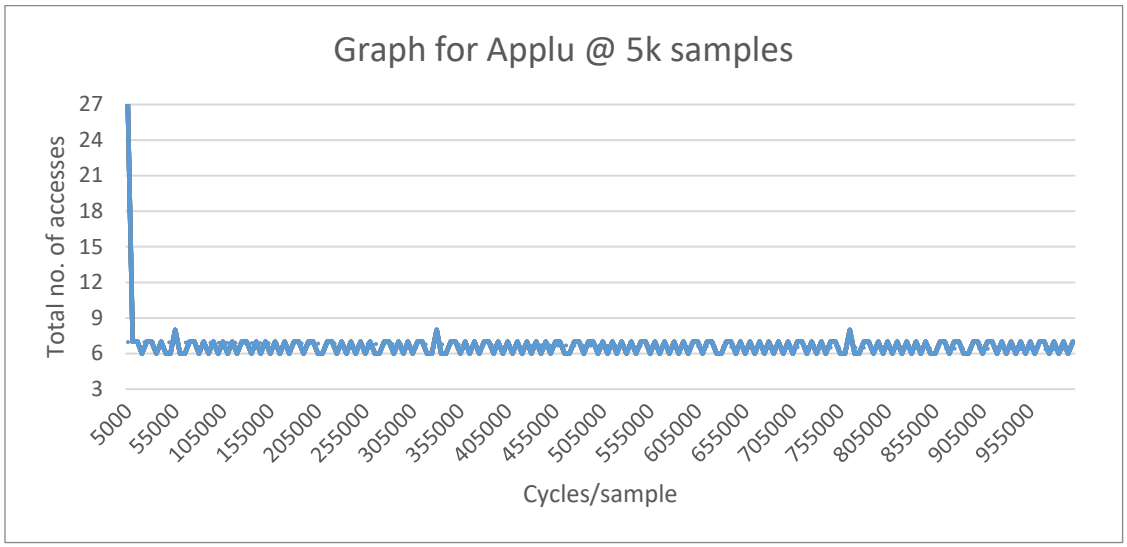
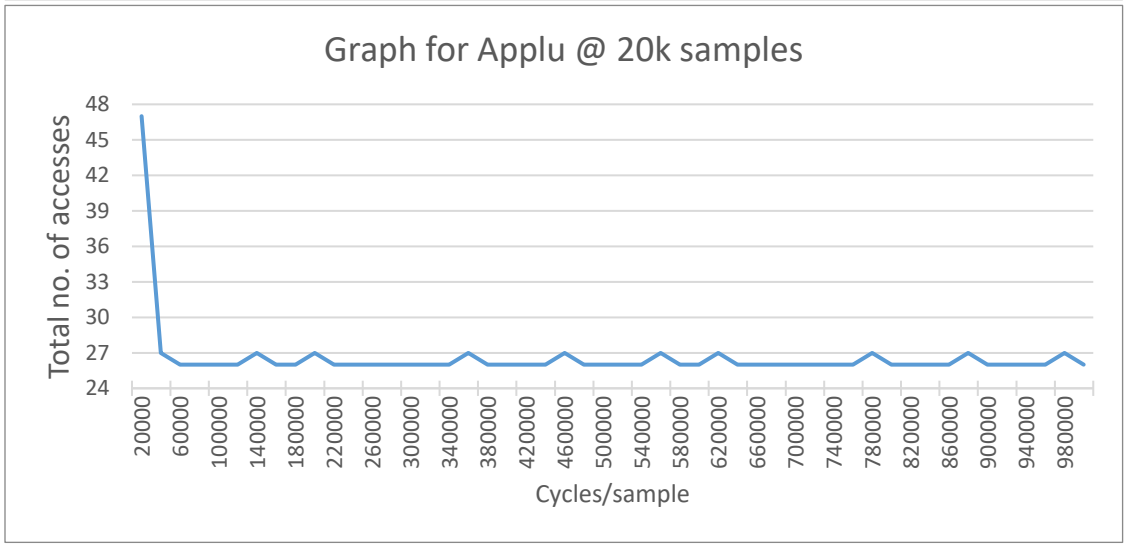
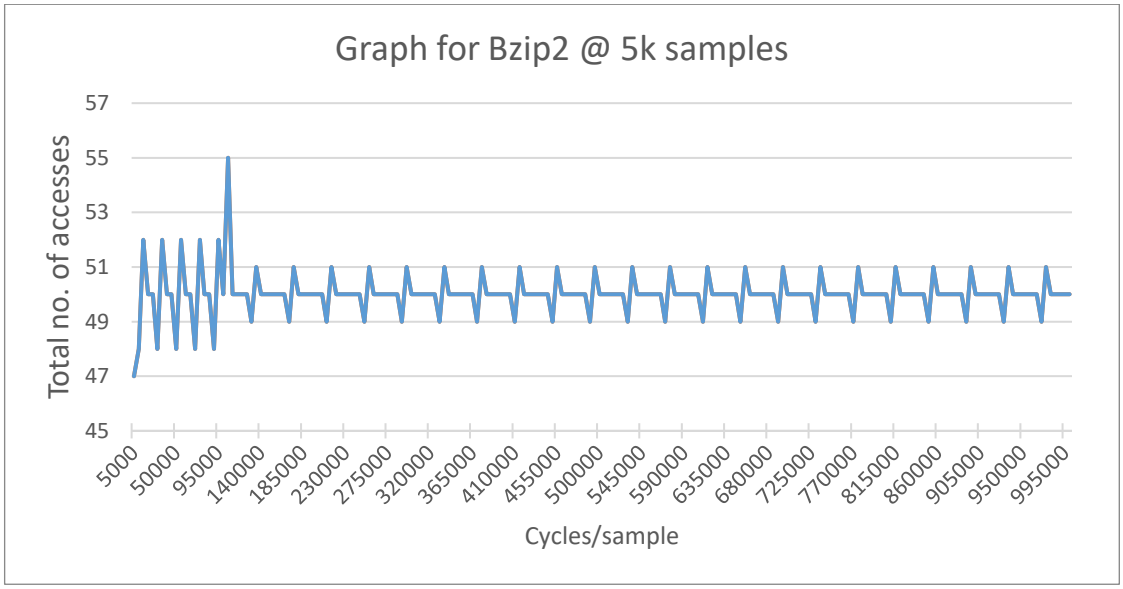
**1)** Graphs

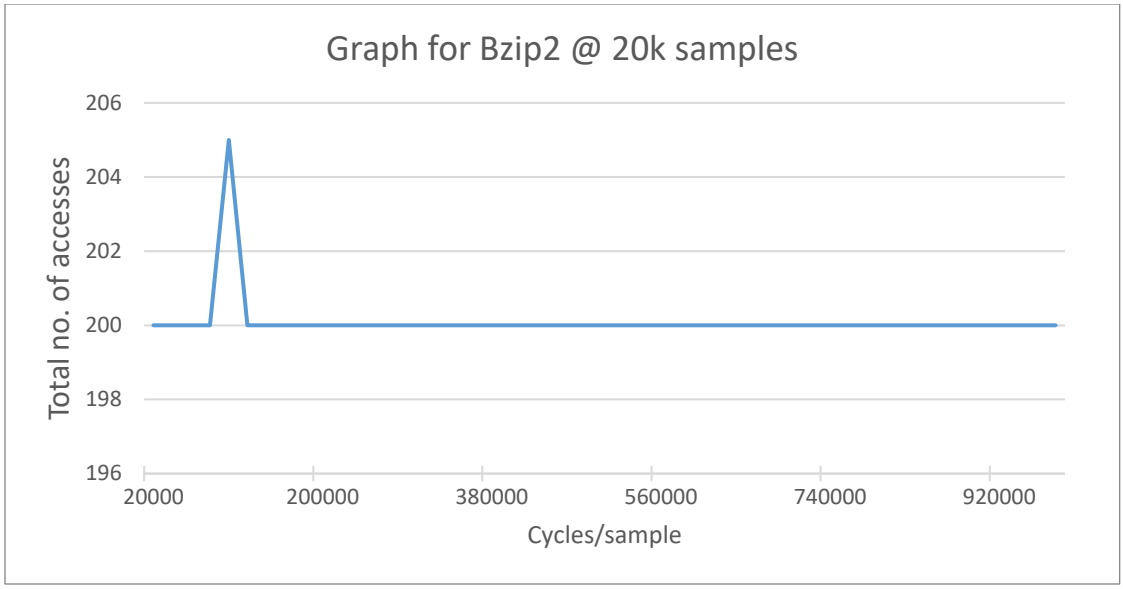
a. Graphs for Applu Benchmark (one core)



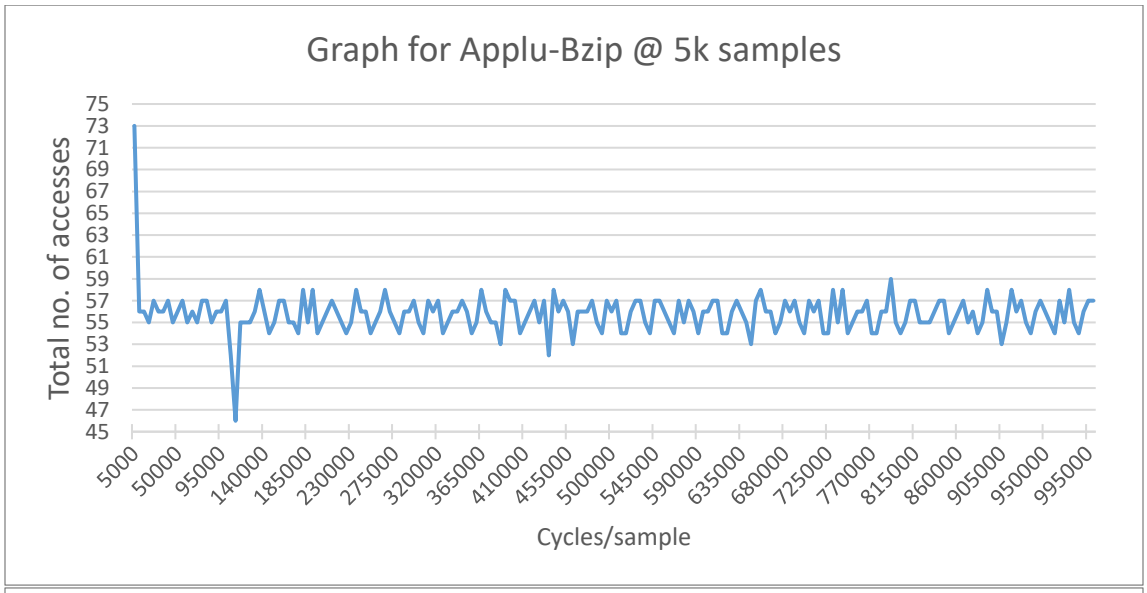



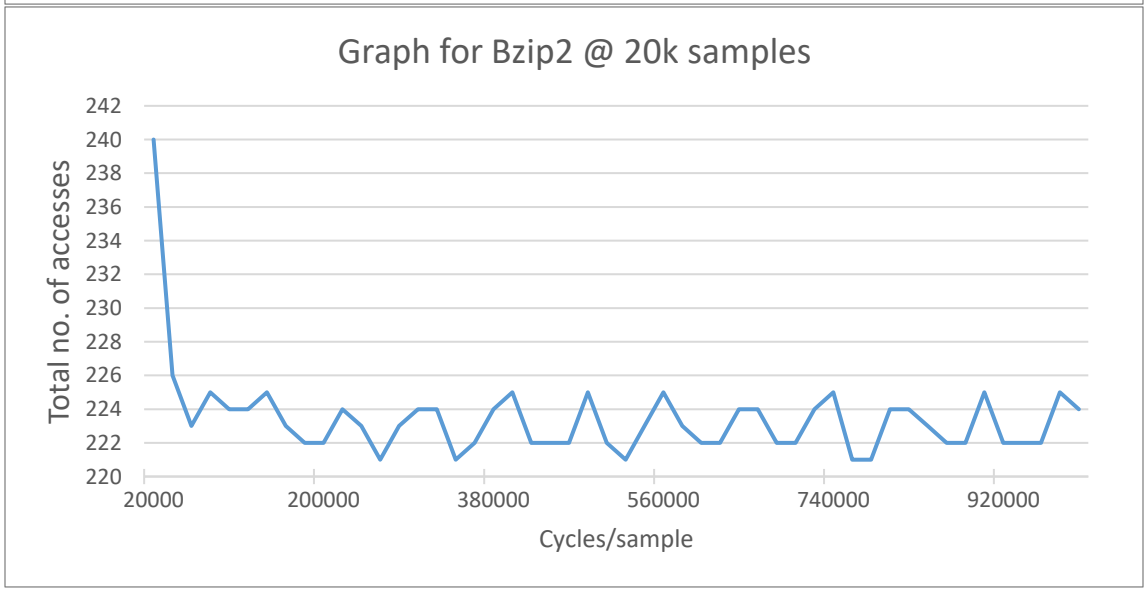

b. Graphs for Bzip2 Benchmark (one core)






c. Graphs for Applu-Bzip2 Benchmark (two core)



**2)** Ratio of number of reads and number of writes to the total number of accesses

a. For Applu

$$\frac{\text{Total number of reads}}{\text{Total number of access}} = \frac{1331}{1331} = 1$$

$$\frac{\text{Total number of writes}}{\text{Total number of access}} = \frac{0}{1331} = 0$$

b. For Bzip2

$$\frac{\text{Total number of reads}}{\text{Total number of access}} = \frac{5518}{10002} = 0.5517$$

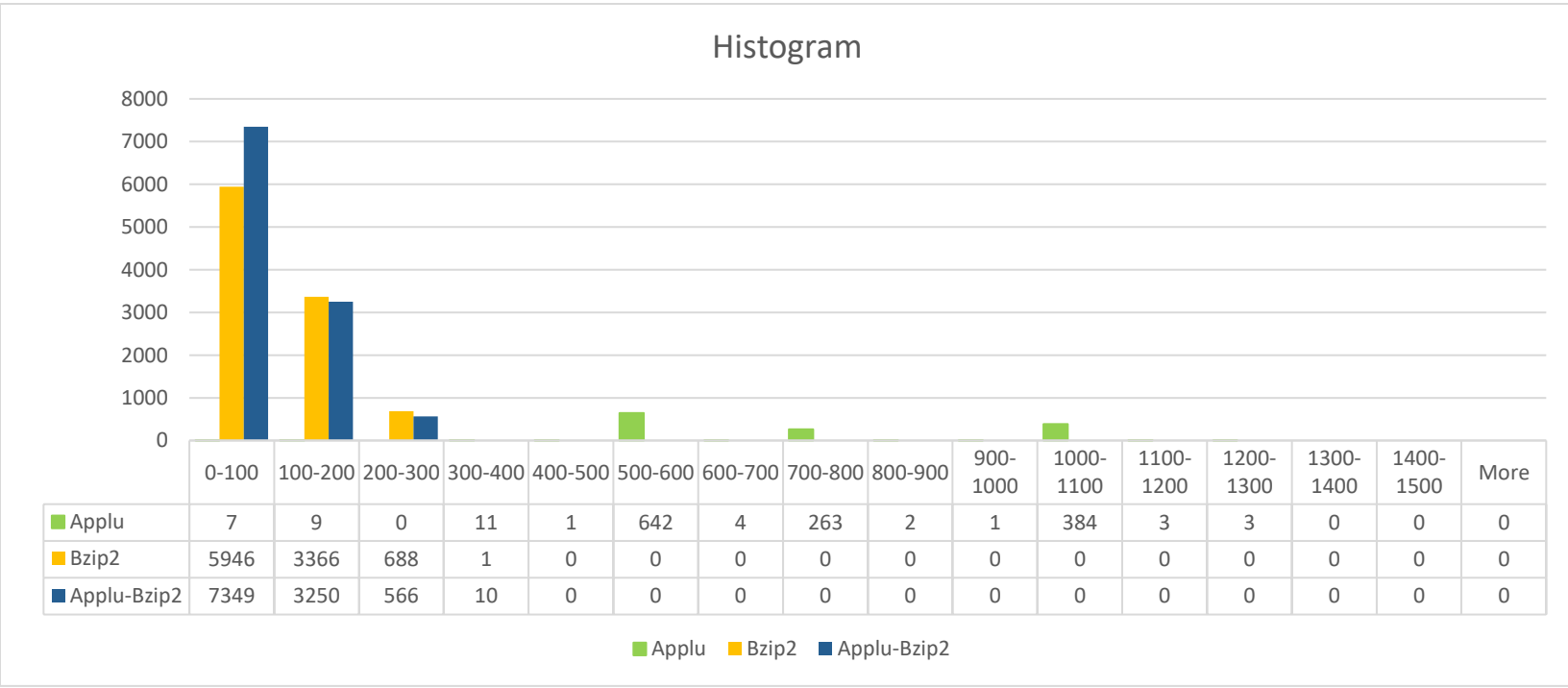
$$\frac{\text{Total number of writes}}{\text{Total number of access}} = \frac{4484}{10002} = 0.4483$$

c. For Applu-Bzip2

$$\frac{\text{Total number of reads}}{\text{Total number of access}} = \frac{6725}{11156} = 0.628$$

$$\frac{\text{Total number of writes}}{\text{Total number of access}} = \frac{4431}{11156} = 0.3971$$

**3)** Histogram for time gap




| | 0-100 | 100-200 | 200-300 | 300-400 | 400-500 | 500-600 | 600-700 | 700-800 | 800-900 | 900-1000 | 1000-1100 | 1100-1200 | 1200-1300 | 1300-1400 | 1400-1500 | More |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Applu | 7 | 9 | 0 | 11 | 1 | 642 | 4 | 263 | 2 | 1 | 384 | 3 | 3 | 0 | 0 | 0 |
| Bzip2 | 5946 | 3366 | 688 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Applu-Bzip2 | 7349 | 3250 | 566 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**4) Analysis and discussion**

From the graphs the following can be seen

(a) For Applu, the memory access starts off with a very high memory access and then eventually settles down to get an almost flat line with occasional small spikes. This can be seen for when the sampling is at 5k and at 20k.

(b) In the case of bzip2_source, the memory accesses are repeatedly sharp spikes above and below a fixed line when sampling at 5k. However, in the case of 20k samples there is just one spike at the start of the memory access and is constantly flat from there on.

(c) When looking at the two core benchmarks, the number of accesses is constantly changing around a fixed threshold line with a couple of spikes that show up for both 5k and 20k sampling.

**5)** Modification in the code

process_memaccess & process_memwb are added to the code

ff_dist is changed to 10^6 and when cycles are greater than 1M it will stop printing

All memory accesses are printed out in sequence.

Code is below

```
static void
process_memaccess(CacheRequest *creq)
{
    MemUnit *mu = SharedMemUnit;
    creq->request_time = memunit_access(mu, creq->base_addr, cyc,
                                        MemUnit_Read);
    CacheAccessType access_type = Cache_Read; // Access type defined
    if( cyc < 1000000 )
    printf("%s %s %s \n",CacheAccessType_names[access_type],fmt_now(),
fmt_laddr(creq->base_addr));



    creq->service_level = SERVICED_MEM;
    if (GlobalParams.mem.use_l3cache) {
        creq->action = L3FILL;
    } else {
        creq->action = (GlobalParams.mem.private_l2caches) ?
            BUS_REPLY : L2FILL;
    }
    {
        // Writebacks are not billed per-application, though they're still
        // counted in the MemUnit stats.
        AppState * restrict as = first_request_app(creq);
        if (as)
            as->extra->mem_accesses++;
    }
```

```
    place_in_cache_queue(creq);
}


static void
process_memwb(CacheRequest *creq)
{

    MemUnit *mu = SharedMemUnit;
    CacheAccessType access_type = Cache_Write; // Access type defined
    if( cyc < 1000000 )
    printf("%s %s %s \n",CacheAccessType_names[access_type],fmt_now(),
fmt_laddr(creq->base_addr));

    if (GlobalParams.mem.use_l3cache) {
        cache_wb_accepted(SharedL3Cache, creq->base_addr);
    } else if (!GlobalParams.mem.private_l2caches) {
        cache_wb_accepted(SharedL2Cache, creq->base_addr);
    }



    creq->request_time = memunit_access(mu, creq->base_addr, cyc,
                                        MemUnit_Write);
    sim_assert(!creq->dependent_coher); // Not used (and not handled)
    free_cache_request(creq);

}
```