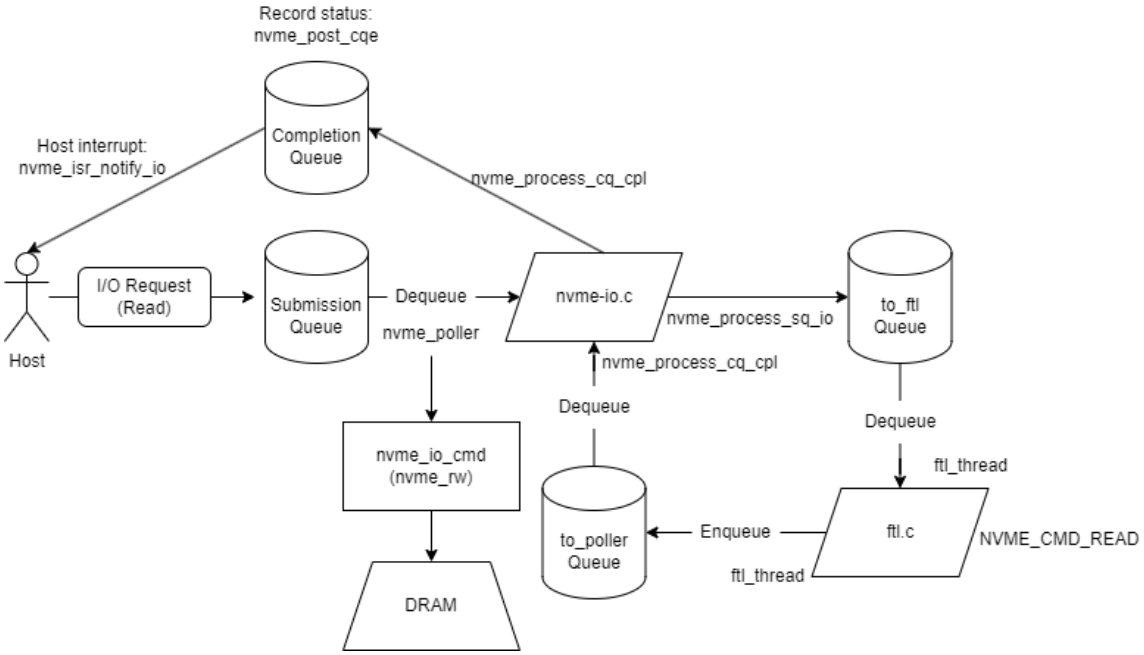
Project 1

Analyzing code and mechanisms of Femu

과목명	시스템프로그래밍	학과	컴퓨터학부
교수명	최원일	학번	2020052551
제출일	2024.10.30.	이름	성주원

(A) Explanation of code analysis for (i)read, (ii)write, and (iii)Garbage Collection Mechanisms

(i)Read code analysis(flow)



1. Host(OS) sends a I/O request(which is read in this case) and I/O is Enqueued in submission queue.
2. The function(nvme_poller) gets request from submission queue. Then, nvme_io_cmd and nvme_rw function uses DRAM to process our requested read I/O.
3. After updating request, nvme-io.c enqueue the request in to_ftl queue using nvme_process_sq_io.
4. ftl.c recognize there is request in to_ftl queue using ft_thread and operate function based on the request(which is NVME_CMD_READ in this case)
5. After operation, ftl.c enqueue the update in to_poller queue using ftl_thread.
6. nvme-io.c recognize there is update in to_poller queue using nvme_poller function.
7. Check if the request has been operated properly and completed in time using nvme_process_cq_cpl. If request is all fine, nvme-io.c enqueue completed request in completion queue with request status(nvme_post_cqe).
8. nvme-io.c send interrupt to host using nvme_isr_notify_io and host brings completed request from completion queue.

(i)Read code analysis

1. When nvme-io.c brings request from submission queue

```
    default:
        while (1) {
            if ((!n->dataplane_started)) {
                usleep(1000);
                continue;
            }

            for (i = 1; i <= n->nr_io_queues; i++) {
                NvmeSQueue *sq = n->sq[i];
                NvmeCQueue *cq = n->cq[i];
                if (sq && sq->is_active && cq && cq->is_active) {
                    nvme_process_sq_io(sq, index);
                }
            }
            nvme_process_cq_cpl(n, index);
        }
        break;
    }

    return NULL;
}
```

-This is part of nvme_poller function. This case is when multipoller is disabled(default case). Keep checking if there is request in submission queue or to_poller queue. In this case, If submission queue and completion queue can accept I/O, it process to nvme_process_sq_io to dequeue from submission queue.

```
nvme_update_sq_tail(sq);
while (!(nvme_sq_empty(sq))) {
    if (sq->phys_contig) {
        addr = sq->dma_addr + sq->head * n->sqe_size;
        nvme_copy_cmd(&cmd, (void *)&(((NvmeCmd *)sq->dma_addr_hva)[sq->head]));
    } else {
        addr = nvme_discontig(sq->prp_list, sq->head, n->page_size,
                              n->sqe_size);
        nvme_addr_read(n, addr, (void *)&cmd, sizeof(cmd));
    }
    nvme_inc_sq_head(sq);

    req = QTAILQ_FIRST(&sq->req_list);
    QTAILQ_REMOVE(&sq->req_list, req, entry);
    memset(&req->cqe, 0, sizeof(req->cqe));
    /* Coperd: record req->stime at earliest convenience */
    req->expire_time = req->stime = qemu_clock_get_ns(QEMU_CLOCK_REALTIME);
    req->cqe.cid = cmd.cid;
    req->cmd_opcode = cmd.opcode;
    memcpy(&req->cmd, &cmd, sizeof(NvmeCmd));
```

-This is part of nvme_process_sq_io function. First, update the submission queue tail part and dequeue from submission queue until submission queue is empty. Next operation brings the request and reads the command using nvme_copy_cmd or nvme_addr_read. To do so, it gets the address of submission queue head. But the way it reads the head part differs depending whether queues are contiguous or not. After, it updates the submission queue and store the queue into request structure in req and removes the queue we have stored. After, it initialize the

queue part. With the request, we store expiration time, id, and opcode(operation code and at last, we store in cmd structure.

2. nvme-io.c enqueue the processed request in to_ftl queue

```
        status = nvme_io_cmd(n, &cmd, req);
        if (1 && status == NVME_SUCCESS)
        {
            req->status = status;

            int rc = femu_ring_enqueue(n->to_ftl[index_poller], (void *)&req, 1);
            if (rc != 1)
            {
                femu_err("enqueue failed, ret=%d\n", rc);
            }
        }
        else if (status == NVME_SUCCESS)
        {
            /* Normal I/Os that don't need delay emulation */
            req->status = status;
        }
        else
        {
            femu_err("Error IO processed!\n");
        }

        processed++;
    }

    nvme_update_sq_eventidx(sq);
    sq->completed += processed;
}
```

-Using the cmd variable and request variable, nvme_io_cmd operates the I/O by calling nvme_rw function and returns I/O request status. We then check whether I/O has been successfully processed before continue to enqueue. If it returns success, we store status in request structure and enqueue in to_ftl queue. After all the procedure, we update processed variable and update the submission queue event index and add processed variable in submission queue's completed variable to update.

```c
static uint16_t nvme_io_cmd(FemuCtrl *n, NvmeCmd *cmd, NvmeRequest *req)
{

    NvmeNamespace *ns;
    uint32_t nsid = le32_to_cpu(cmd->nsid);

    if (nsid == 0 || nsid > n->num_namespaces)
    {
        femu_err("%s, NVME_INVALID_NSID %" PRIu32 "\n", __func__, nsid);
        return NVME_INVALID_NSID | NVME_DNR;
    }

    req->ns = ns = &n->namespaces[nsid - 1];
```

-Now, we will lokk into the nvme_io_cmd function but will look in partly. First, we will bring nsid from cmd. We check integrity of our received nsid. If we checked the integrity, we allocate namespace for our nsid.

```c
    default:
        if (n->ext_ops.io_cmd)
        {
            return n->ext_ops.io_cmd(n, ns, cmd, req);
        }

        femu_err("%s, NVME_INVALID_OPCODE\n", __func__);
        return NVME_INVALID_OPCODE | NVME_DNR;
```

-There were a lot of if statement in the nvme_io_cmd function but those if statement is basically for the exceptional operations so we will look in default part. We check corresponding io_cmd for our opcode and we internally invoke the nvme_rw function(at the third parameter of io_cmd(the [cmd] part)) to execute our I/O. If not, we return 'invalid opcode' line to user.

```
uint16_t nvme_rw(FemuCtrl *n, NvmeNamespace *ns, NvmeCmd *cmd, NvmeRequest *req)
{
    NvmeRwCmd *rw = (NvmeRwCmd *)cmd;
    uint16_t ctrl = le16_to_cpu(rw->control);
    uint32_t nlb = le16_to_cpu(rw->nlb) + 1;
    uint64_t slba = le64_to_cpu(rw->slba);
    uint64_t prp1 = le64_to_cpu(rw->prp1);
    uint64_t prp2 = le64_to_cpu(rw->prp2);
    const uint8_t lba_index = NVME_ID_NS_FLBAS_INDEX(ns->id_ns.flbas);
    const uint16_t ms = le16_to_cpu(ns->id_ns.lbaf[lba_index].ms);
    const uint8_t data_shift = ns->id_ns.lbaf[lba_index].lbads;
    uint64_t data_size = (uint64_t)nlb << data_shift;
    uint64_t data_offset = slba << data_shift;
    uint64_t meta_size = nlb * ms;
    uint64_t elba = slba + nlb;
    uint16_t err;
    int ret;

    req->is_write = (rw->opcode == NVME_CMD_WRITE) ? 1 : 0;

    err = femu_nvme_rw_check_req(n, ns, cmd, req, slba, elba, nlb, ctrl,
                                 data_size, meta_size);
    if (err)
        return err;

    if (nvme_map_prp(&req->qsg, &req->iov, prp1, prp2, data_size, n))
    {
        nvme_set_error_page(n, req->sq->sqid, cmd->cid, NVME_INVALID_FIELD,
                            offsetof(NvmeRwCmd, prp1), 0, ns->id);
        return NVME_INVALID_FIELD | NVME_DNR;
    }

    assert((nlb << data_shift) == req->qsg.size);

    req->slba = slba;
    req->status = NVME_SUCCESS;
    req->nlb = nlb;
```

When the cmd parameter is allocated, the nvme_rw function is invoked internally. After we initalizes the values we needed to operate(too many so explain as we go), first, we check is_write by checking the op_code. We check the integrity of our requested I/O operation(cmd format) by using femu_nvme_rw_check_req and if the function finds there is something wrong with our request, we return error. We again try to map our request to the prp(physical region page) and if it failes, we return error using nvme_set_error_page function. After we check integrity again using size of our request and updates status.

```
    ret = backend_rw(n->mbe, &req->qsg, &data_offset, req->is_write);
    if (!ret)
    {
        return NVME_SUCCESS;
    }

    return NVME_DNR;
}
```

-We invoke backend_rw function where it process our I/O finally in the DRAM.

3. ftl.c dequeue request from to_ftl queue

```
while (1) {
    for (i = 1; i <= n->nr_pollers; i++) {
        if (!ssd->to_ftl[i] || !femu_ring_count(ssd->to_ftl[i]))
            continue;

        rc = femu_ring_dequeue(ssd->to_ftl[i], (void *)&req, 1);
        if (rc != 1) {
            printf("FEMU: FTL to_ftl dequeue failed\n");
        }

        ftl_assert(req);
        switch (req->cmd.opcode) {
        case NVME_CMD_WRITE:
            lat = ssd_write(ssd, req);
            break;
        case NVME_CMD_READ:
            lat = ssd_read(ssd, req);
            break;
        case NVME_CMD_DSM:
            lat = 0;
            break;
        default:
            //ftl_err("FTL received unkown request type, ERROR\n");
            ;
        }
```

-This is part of ftl_thread function code in the ftl.c file. It keep check to_ftl queue if there is any queue in it. If request is found in to_ftl queue, it dequeue from to_ftl queue using femu_ring_dequeue function. Before bringing the opcode(nvme command)(in this case, NVME_CMD_READ), it checks validation and then brings op code.

```
static uint64_t ssd_read(struct ssd *ssd, NvmeRequest *req)
{
    struct ssdparams *spp = &ssd->sp;
    uint64_t lba = req->slba;
    int nsecs = req->nlb;
    struct ppa ppa;
    uint64_t start_lpn = lba / spp->secs_per_pg;
    uint64_t end_lpn = (lba + nsecs - 1) / spp->secs_per_pg;
    uint64_t lpn;
    uint64_t sublat, maxlat = 0;
```

-Since we are looking into read I/O request, our I/O will go to the ssd_read function. ssd_read operation gets the starting point of logical block address and number of logical blocks of I/O request. With the variables, we calculate the start of lpn and the end of lpn. We will use rest of variable as we go throughout the function.

```
/* normal IO read path */
for (lpn = start_lpn; lpn <= end_lpn; lpn++)
{
    ppa = get_maptbl_ent(ssd, lpn);
    if (!mapped_ppa(&ppa) || !valid_ppa(ssd, &ppa))
    {
        // printf("%s,lpn(%" PRId64 ") not mapped to valid ppa\n", ssd->ssdname, lpn);
        // printf("Invalid ppa,ch:%d,lun:%d,blk:%d,pl:%d,pg:%d,sec:%d\n",
        // ppa.g.ch, ppa.g.lun, ppa.g.blk, ppa.g.pl, ppa.g.pg, ppa.g.sec);
        continue;
    }
```

-It goes over the lpn from start lpn to end lpn. By using get_maptbl_ent function, we get physical page address. If the ppa(physical page address) is not valid, we don't proceed further.

```
        struct nand_cmd srd;
        srd.type = USER_IO;
        srd.cmd = NAND_READ;
        srd.stime = req->stime;
        sublat = ssd_advance_status(ssd, &ppa, &srd);
        maxlat = (sublat > maxlat) ? sublat : maxlat;
    }

    return maxlat;
}
```

-But if the ppa is valid, we store informations in srd variable and calculate the latency of our I/O using ssd_advance_status function and the srd. After calculating, it returns maximum latency.

```
case NAND_READ:
    /* read: perform NAND cmd first */
    nand_stime = (lun->next_lun_avail_time < cmd_stime)
                    ? cmd_stime
                    : lun->next_lun_avail_time;
    lun->next_lun_avail_time = nand_stime + spp->pg_rd_lat;
    lat = lun->next_lun_avail_time - cmd_stime;
```

-This is part of ssd_advance_status function. Since we are looking into read I/O, it calculates the read I/O latency with nand command start time(nand_stime) and pg_rd_lat(read latency in nano second). It finally calculates latency by subtracting the command start time.

4. ftl.c enqueue request in to_poller queue

```
            req->reqlat = lat;
            req->expire_time += lat;

            rc = femu_ring_enqueue(ssd->to_poller[i], (void *)&req, 1);
            if (rc != 1) {
                ftl_err("FTL to_poller enqueue failed\n");
            }

            /* clean one line if needed (in the background) */
            if (should_gc(ssd)) {
                do_gc(ssd, false);
            }
        }
    }

    return NULL;
}
```

-Back to ftl_thread function, it updates the latency we've just calculated and update the expiration time using latency. After, we enqueue in to_poller queue.(Check if garbage collection should be done(we will look into it when we analyze the GC))

5. nvme-io.c dequeue from to_poller using nvme_poller

```
                }
            }
            nvme_process_cq_cpl(n, index);
        }
        break;
    }

    return NULL;
}
```

-Back to nvme_poller function, Call nvme_process_cq_cpl function

```
while (femu_ring_count(rp)) {
    req = NULL;
    rc = femu_ring_dequeue(rp, (void *)&req, 1);
    if (rc != 1) {
        femu_err("dequeue from to_poller request failed\n");
    }
    assert(req);

    pqueue_insert(pq, req);
}
```

-This is part of nvme_process_cq_cpl function. It dequeue from to_poller queue using femu_ring_dequeue and store in vaariable req temporarily. After we check integrity of req, we enqueue in pq temporarily to align queues in fast order.

6. Enqueue request to completion queue using nvme_proccess_cq_cql

```
while ((req = pqueue_peek(pq)))
{
    now = qemu_clock_get_ns(QEMU_CLOCK_REALTIME);
    if (now < req->expire_time)
    {
        break;
    }

    cq = n->cq[req->sq->sqid];
    if (!cq->is_active)
        continue;
    nvme_post_cqe(cq, req);
    QTAILQ_INSERT_TAIL(&req->sq->req_list, req, entry);
    pqueue_pop(pq);
    processed++;
    n->nr_tt_ios++;
```

-We bring queues until pq is empty. We check the time now and check if whether request time is expired or not. Using submission queue id, the function enqueue to completion queue using nvme_post_cqe. After it has been enqueued in completion queue, it update pq queue. It updates the how many time it processed and total number of I/O request they've done.

```
static void nvme_post_cqe(NvmeCQueue *cq, NvmeRequest *req)
{
    FemuCtrl *n = cq->ctrl;
    NvmeSQueue *sq = req->sq;
    NvmeCqe *cqe = &req->cqe;
    uint8_t phase = cq->phase;
    hwaddr addr;

    if (n->print_log)
    {
        femu_debug("%s,req,lba:%lu,lat:%lu\n", n->devname, req->slba, req->reqlat);
    }
    cqe->status = cpu_to_le16((req->status << 1) | phase);
    cqe->sq_id = cpu_to_le16(sq->sqid);
    cqe->sq_head = cpu_to_le16(sq->head);

    if (cq->phys_contig)
    {
        addr = cq->dma_addr + cq->tail * n->cqe_size;
        ((NvmeCqe *)cq->dma_addr_hva)[cq->tail] = *cqe;
    }
    else
    {
        addr = nvme_discontig(cq->prp_list, cq->tail, n->page_size, n->cqe_size);
        nvme_addr_write(n, addr, (void *)cqe, sizeof(*cqe));
    }

    nvme_inc_cq_tail(cq);
}
```

-Now we will look into nvme_post_cqe function. The purpose of this function is to enqueue the request into the completion queue. We will translate the request format into cqe format by updating status, sq_id, and sq_head. Whether the completion queue is physically contiguous or not(if it is not contigous, it uses nvme_discontig and nvme_addr_write function as a helper function), they both calculate the physical address in DMA memory(the memory used without cpu interference) and writes our request(completion queue format) into DMA memory.

```
    switch (n->multipoller_enabled) {
    case 1:
        nvme_isr_notify_io(n->cq[index_poller]);
        break;
    default:
        for (i = 1; i <= n->nr_io_queues; i++) {
            if (n->should_isr[i]) {
                nvme_isr_notify_io(n->cq[i]);
                n->should_isr[i] = false;
            }
        }
        break;
    }
}
```

-If the expiration time has exceeded 20000 nano seconds, we update the number of late I/O operation and print the log. After everything is done, we notify the completion by updating should_isr into true. If there is no process done, we return.

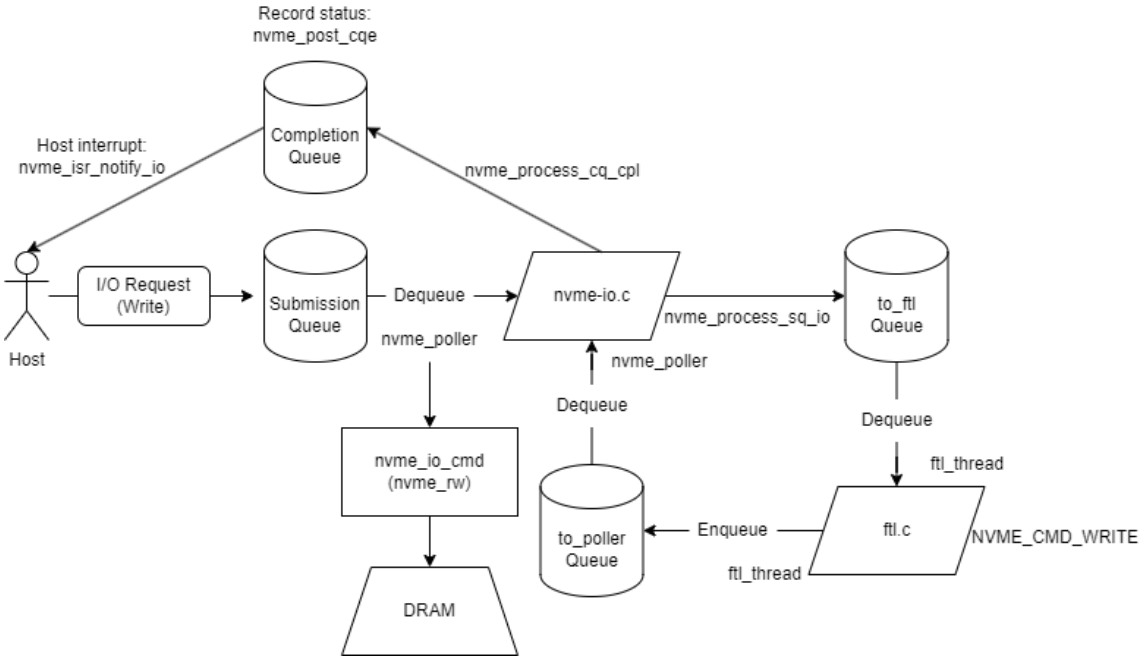
7. Send interrupt to host using nvme_isr_notify_io to notify host requested has been processed

-Depends on multipoller enabled or not(disabled is default), the operation is divided but both of them notify(send interrupt) host using nvme_isr_notify_io to let host dequeue from completion queue.

```
        if (now - req->expire_time >= 20000)
        {
            n->nr_tt_late_ios++;
            if (n->print_log)
            {
                femu_debug("%s,diff,pq.count=%lu,%" PRId64 ", %lu/%lu\n",
                           n->devname, pqueue_size(pq), now - req->expire_time,
                           n->nr_tt_late_ios, n->nr_tt_ios);
            }
        }
        n->should_isr[req->sq->sqid] = true;
    }

    if (processed == 0)
        return;
```

(ii)Write code analysis(flow)



1. Host(OS) sends a I/O request(which is read in this case) and I/O is Enqueued in submission queue.
2. The function(nvme_poller) gets request from submission queue. Then the nvme_poller function invokes nvme_io_cmd and nvme_rw function to operate our reqeusted write I/O in DRAM.
3. After updating request, nvme-io.c enqueue in to_ftl queue using nvme_process_sq_io.
4. ftl.c recognize there is request in to_ftl queue using ft_thread and operate function based on the request(which is NVME_CMD_WRITE in this case)
5. After operation, ftl.c enqueue the update in to_poller queue using ftl_thread.
6. nvme-io.c recognize there is update in to_poller queue using nvme_poller function.
7. Check if the request has been operated properly and completed in time using nvme_process_cq_cpl. If request is all fine, nvme-io.c enqueue completed request in completion queue with request status(nvme_post_cqe).
8. nvme-io.c send interrupt to host using nvme_isr_notify_io and host brings completed request from completion queue.

(ii)Write code analysis

-1, 2 step is same with read code analysis

3. ftl.c dequeue request from to_ftl queue

```
while (1) {
    for (i = 1; i <= n->nr_pollers; i++) {
        if (!ssd->to_ftl[i] || !femu_ring_count(ssd->to_ftl[i]))
            continue;

        rc = femu_ring_dequeue(ssd->to_ftl[i], (void *)&req, 1);
        if (rc != 1) {
            printf("FEMU: FTL to_ftl dequeue failed\n");
        }

        ftl_assert(req);
        switch (req->cmd.opcode) {
        case NVME_CMD_WRITE:
            lat = ssd_write(ssd, req);
            break;
        case NVME_CMD_READ:
            lat = ssd_read(ssd, req);
            break;
        case NVME_CMD_DSM:
            lat = 0;
            break;
        default:
            //ftl_err("FTL received unkown request type, ERROR\n");
            ;
        }
```

-This is part of ftl_thread function code in the ftl.c file. It keep check to_ftl queue if there is any processed queue. If request is found in to_ftl queue, it dequeue from to_ftl queue and depend on opcode(in this case, NVME_CMD_WRITE), it brings from ssd.Then we go to the ssd_write function

```c
static uint64_t ssd_write(struct ssd *ssd, NvmeRequest *req)
{
    uint64_t lba = req->slba;
    struct ssdparams *spp = &ssd->sp;
    int len = req->nlb;
    uint64_t start_lpn = lba / spp->secs_per_pg;
    uint64_t end_lpn = (lba + len - 1) / spp->secs_per_pg;
    struct ppa ppa;
    uint64_t lpn;
    uint64_t curlat = 0, maxlat = 0;
    int r;

    if (end_lpn >= spp->tt_pgs) {
        ftl_err("start_lpn=%"PRIu64",tt_pgs=%d\n", start_lpn, ssd->sp.tt_pgs);
    }

    while (should_gc_high(ssd)) {
        /* perform GC here until !should_gc(ssd) */
        r = do_gc(ssd, true);
        if (r == -1)
            break;
    }

    for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
        ppa = get_maptbl_ent(ssd, lpn);
        if (mapped_ppa(&ppa)) {
            /* update old page information first */
            mark_page_invalid(ssd, &ppa);
            set_rmap_ent(ssd, INVALID_LPN, &ppa);
        }
```

-Same as the ssd_read operation, ssd_write calculates the start lpn and the end lpn using lba and number of sectors. After calculating lpn, we check if garbage collection is needed(high threshold in this case(discussed in detail at GC part)). Until garbage collection is not needed(when number of free line goes under the threshold), it iterate over the logical page number from the start to the end. We check mapping table using get_maptbl_ent to get ppa(physical page address). If the ppa is mapped, we mark the old page as invalid first using mark_page_invalid function. After, reverse update mapping table(PPA to LPN mapping table) using set_rmap_ent function.

```c
/* update SSD status about one page from PG_VALID -> PG_VALID */
static void mark_page_invalid(struct ssd *ssd, struct ppa *ppa)
{
    struct line_mgmt *lm = &ssd->lm;
    struct ssdparams *spp = &ssd->sp;
    struct nand_block *blk = NULL;
    struct nand_page *pg = NULL;
    bool was_full_line = false;
    struct line *line;

    /* update corresponding page status */
    pg = get_pg(ssd, ppa);
    ftl_assert(pg->status == PG_VALID);
    pg->status = PG_INVALID;

    /* update corresponding block status */
    blk = get_blk(ssd, ppa);
    ftl_assert(blk->ipc >= 0 && blk->ipc < spp->pgs_per_blk);
    blk->ipc++;
    ftl_assert(blk->vpc > 0 && blk->vpc <= spp->pgs_per_blk);
    blk->vpc--;

    /* update corresponding line status */
    line = get_line(ssd, ppa);
    ftl_assert(line->ipc >= 0 && line->ipc < spp->pgs_per_line);
```

-Before proceeding further lets look into the mark_page_invalid function. First, we will get the page using get_pg function, check the integrity, and updates the status into invalid. Again we get the block of our page and check the invalid page, valid page integrity and updates both count respectively. After we get the corresponding line and check the integrity as well.

```
    if (line->vpc == spp->pgs_per_line)
    {
        ftl_assert(line->ipc == 0);
        was_full_line = true;
    }
    line->ipc++;
    ftl_assert(line->vpc > 0 && line->vpc <= spp->pgs_per_line);
    /* Adjust the position of the victime line in the pq under over-writes */
    if (line->pos)
    {
        /* Note that line->vpc will be updated by this call */
        pqueue_change_priority(lm->victim_line_pq, line->vpc - 1, line);
    }
    else
    {
        line->vpc--;
    }

    if (was_full_line)
    {
        /* move line: "full" -> "victim" */
        QTAILQ_REMOVE(&lm->full_line_list, line, entry);
        lm->full_line_cnt--;
        pqueue_insert(lm->victim_line_pq, line);
        lm->victim_line_cnt++;
    }
}
```

-If the number of valid page in the line is same with the number of pages per line, we check if the number of invalid page and updates the was_full_line flag. After, we increment the number of invalid page count and check the integrity of number of the valid pages. Before we over write, we adjust the position of priority queue of victim line. If the line is in the victim line priority queue, we update the queue by using pqueue_change_priority function. If it is not, we will update valid page count simply by subtracting. If the was_full_line is true(false by default, we've discussed when this flag is set to true.) we remove our line from full_line_list and updates the count. After, we insert into victim_line_pq(priority queue) and updates the victim_line_cnt.

```
    /* new write */
    ppa = get_new_page(ssd);
    /* update maptbl */
    set_maptbl_ent(ssd, lpn, &ppa);
    /* update rmap */
    set_rmap_ent(ssd, lpn, &ppa);

    mark_page_valid(ssd, &ppa);

    /* need to advance the write pointer here */
    ssd_advance_write_pointer(ssd);

    struct nand_cmd swr;
    swr.type = USER_IO;
    swr.cmd = NAND_WRITE;
    swr.stime = req->stime;
    /* get latency statistics */
    curlat = ssd_advance_status(ssd, &ppa, &swr);
    maxlat = (curlat > maxlat) ? curlat : maxlat;
}

return maxlat;
```

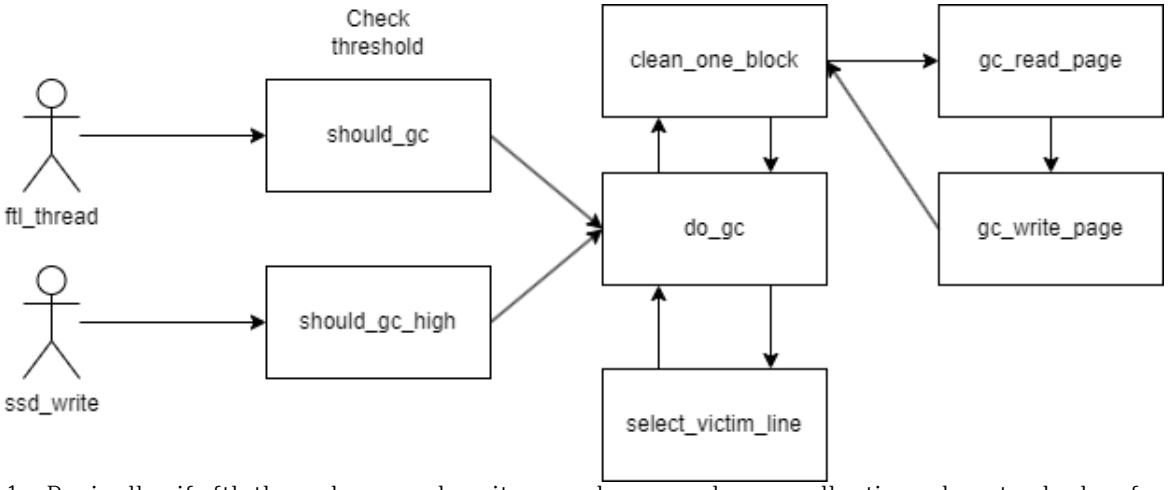
-Back to our ssd_write function we get the new page to write and update both mapping table and reverse mapping table and mark page as valid. After advancing the write pointer, store write operation result in swr and calculate the latency. After checking the max latency, we return that time.

```
    case NAND_WRITE:
        /* write: transfer data through channel first */
        nand_stime = (lun->next_lun_avail_time < cmd_stime)
                   ? cmd_stime
                   : lun->next_lun_avail_time;
        if (ncmd->type == USER_IO)
        {
            lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat;
        }
        else
        {
            lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat;
        }
        lat = lun->next_lun_avail_time - cmd_stime;
```

-Now we have to look into the ssd_advance_status function. Before we've looked into NAND_READ. Since we are looking into write I/O, we will look into NAND_WRITE case. There is if and else statement but no matter what, they operate in the same way which uses the nand start time(nand_stime) and write latency time(pg_wr_lat) to calculate the write I/O latency. At last, it substracts cmd start time(cmd_stime) to calculate final latency value.

-Rest of 4~7 step is same as Read operation

(iii)Garbage Colletion code analysis(flow)



1. Basically if ftl_thread or ssd_write needs a garbage collection due to lack of space(passing threshold value), it calls do_gc function.
2. At do_gc function, it selects victim line first using select_victim_line function
3. do_gc function calls clean_one_block to operate erase
4. Before clean_one_block actually starts to delete, it calls gc_read_page to check if there is any valid page in the block that function is tries to delete.
5. If there is any valid page, gc_write_page moves valid page into free page and updates the table and mark
6. After deleting, do_gc now update by using mark_block_free function and updates line either using mark_line_free function.

(iii)Garbage Collection code analysis

Prerequisite

```
# GC Threshold (1-100)
gc_thres_pcent=75
gc_thres_pcent_high=95
```

```
spp->gc_thres_pcent = n->bb_params.gc_thres_pcent / 100.0;
spp->gc_thres_lines = (int)((1 - spp->gc_thres_pcent) * spp->tt_lines);
spp->gc_thres_pcent_high = n->bb_params.gc_thres_pcent_high / 100.0;
spp->gc_thres_lines_high = (int)((1 - spp->gc_thres_pcent_high) * spp->tt_lines);
```

```
static inline bool should_gc(struct ssd *ssd)
{
    return (ssd->lm.free_line_cnt <= ssd->sp.gc_thres_lines);
}

static inline bool should_gc_high(struct ssd *ssd)
{
    return (ssd->lm.free_line_cnt <= ssd->sp.gc_thres_lines_high);
}
```

-should_gc will be invoked(returns true) when number of free lines are less or equal to 25% of total lines.
should_gc_high will be invoked(returns true) when number of free lines are less or equal to 5% of total lines.

1. GC can be invoked in ftl_thread and ssd_write function

```
            if (should_gc(ssd))
            {
                do_gc(ssd, false);
            }
        }
    }
```

-This is from ftl_thread function and if should_gc condition is satisfied(in other word, of the number of free lines are less than or equal to 25% of total lines), the do_gc function, which operate gc function, will be invoked until enough free lines are made(making free lines over 25% of total lines).

```
while (should_gc_high(ssd))
{
    r = do_gc(ssd, true);
    if (r == -1)
        break;
}
```

-This is from ssd_write function, which will invoke more gc due to low threshold(5%). do_gc function will be invoked until there is free lines less than or equal to 5% of total lines.

2. GC is operated at do_gc function

```
static int do_gc(struct ssd *ssd, bool force)
{
    struct line *victim_line = NULL;
    struct ssdparams *spp = &ssd->sp;
    struct nand_lun *lunp;
    struct ppa ppa;
    int ch, lun;

    victim_line = select_victim_line(ssd, force);
    if (!victim_line) {
        return -1;
    }
```

-Now when the do_gc is invoked from above conditions, this function calls select_victim_line to check if there is any available victim line. If none, the function ends.

```
static struct line *select_victim_line(struct ssd *ssd, bool force)
{
    struct line_mgmt *lm = &ssd->lm;
    struct line *victim_line = NULL;

    victim_line = pqueue_peek(lm->victim_line_pq);
    if (!victim_line) {
        return NULL;
    }

    if (!force && victim_line->ipc < ssd->sp.pgs_per_line / 8) {
        return NULL;
    }

    pqueue_pop(lm->victim_line_pq);
    victim_line->pos = 0;
    lm->victim_line_cnt--;

    /* victim_line is a danggling node now */
    return victim_line;
}
```

-This function selects victim line using pqueue_peek function. The pqueue_peek function returns lowest valid page from priority queue. If the victim_line is null or below threshold and the force flag is false, it returns null. But except for the

mentioned conditions, it removes from queue using pqueue_pop and updates initialize victim line and reduce number of victim_line_cnt as well. It returns found victim_line at last.

```
ppa.g.blk = victim_line->id;
ftl_debug("GC-ing line:%d,ipc=%d,victim=%d,full=%d,free=%d\n", ppa.g.blk,
          victim_line->ipc, ssd->lm.victim_line_cnt, ssd->lm.full_line_cnt,
          ssd->lm.free_line_cnt);
```

-Back to do_gc function, it gets victim line and shows log for debug.

```
/* copy back valid data */
for (ch = 0; ch < spp->nchs; ch++)
{
    for (lun = 0; lun < spp->luns_per_ch; lun++)
    {
        ppa.g.ch = ch;
        ppa.g.lun = lun;
        ppa.g.pl = 0;
        lunp = get_lun(ssd, &ppa);
        clean_one_block(ssd, &ppa);
        mark_block_free(ssd, &ppa);
```

-After, it iterate each channel and LUN(logical unit number) to operate erase for specified block as it iterates. But before doing so, it gets LUN of the specified physical page address. After, it erases and marks the block as free using clean_one_block and mark_block_free function respectively.

```c
/* here ppa identifies the block we want to clean */
static void clean_one_block(struct ssd *ssd, struct ppa *ppa)
{
    struct ssdparams *spp = &ssd->sp;
    struct nand_page *pg_iter = NULL;
    int cnt = 0;

    for (int pg = 0; pg < spp->pgs_per_blk; pg++) {
        ppa->g.pg = pg;
        pg_iter = get_pg(ssd, ppa);
        /* there shouldn't be any free page in victim blocks */
        ftl_assert(pg_iter->status != PG_FREE);
        if (pg_iter->status == PG_VALID) {
            gc_read_page(ssd, ppa);
            /* delay the maptbl update until "write" happens */
            gc_write_page(ssd, ppa);
            cnt++;
        }
    }

    ftl_assert(get_blk(ssd, ppa)->vpc == cnt);
}
```

-To clean block, it checks all the pages in the block. Using the get_pg function, it gets the pointer to the current pointing page using PPA. It checks the integrity using ftl_assert function. If the page is valid, we want to move valid page into other block so we read the valid page using gc_read_page function and write it into new page using gc_write_page function. We increment cnt variable by 1 to accumulate the numbers of valid pages moved for integrity by using get_blk function and ftl_assert function.

```c
static void gc_read_page(struct ssd *ssd, struct ppa *ppa)
{
    /* advance ssd status, we don't care about how long it takes */
    if (ssd->sp.enable_gc_delay) {
        struct nand_cmd gcr;
        gcr.type = GC_IO;
        gcr.cmd = NAND_READ;
        gcr.stime = 0;
        ssd_advance_status(ssd, ppa, &gcr);
    }
}
```

-The function was given with PPA to read the valid page. So this function reads the valid block and save in gcr. First, it checks whether delay is enabled(set true by default). Because of the delay it makes, it updates delay time using ssd_advance_status. Because cmd is set to NAND_READ. as I've explained at the read I/O operation, when the ssd_advance_status is given with the NAND_READ case, it calculates the latency and returns it.

```c
/* move valid page data (already in DRAM) from victim line to a new page */
static uint64_t gc_write_page(struct ssd *ssd, struct ppa *old_ppa)
{
    struct ppa new_ppa;
    struct nand_lun *new_lun;
    uint64_t lpn = get_rmap_ent(ssd, old_ppa);

    ftl_assert(valid_lpn(ssd, lpn));
    new_ppa = get_new_page(ssd);
    /* update maptbl */
    set_maptbl_ent(ssd, lpn, &new_ppa);
    /* update rmap */
    set_rmap_ent(ssd, lpn, &new_ppa);

    mark_page_valid(ssd, &new_ppa);

    /* need to advance the write pointer here */
    ssd_advance_write_pointer(ssd);

    if (ssd->sp.enable_gc_delay) {
        struct nand_cmd gcw;
        gcw.type = GC_IO;
        gcw.cmd = NAND_WRITE;
        gcw.stime = 0;
        ssd_advance_status(ssd, &new_ppa, &gcw);
    }

    /* advance per-ch gc_endtime as well */
#if 0
    new_ch = get_ch(ssd, &new_ppa);
    new_ch->gc_endtime = new_ch->next_ch_avail_time;
#endif

    new_lun = get_lun(ssd, &new_ppa);
    new_lun->gc_endtime = new_lun->next_lun_avail_time;

    return 0;
}
```

-After gc_read_function, we've seen there was gc_write_page. So after reading the found valid page, the function stores LPN in lpn variable using ppa and get_rmap_ent(reverse map(PPA->LPN)). After, it checks integrity of lpn using ftl_assert and valid_lpn function. Then, it updates both map and reverse map. Since we've updated a map, we have to mark the written page as valid using mark_page_valid function. In order to make the next write at the next consecutive page, it advance the write pointer using ssd_advance_write_pointer function. Because this also takes time, if the GC delay is enabled(which is set to true by default), it applies delay time using ssd_advance_status(we've seen how NAND_WRITE latency is calculated at the write I/O analysis). After all the operation, since we wan to update next LUN available time, we translate out updateed ppa into new_lun variable using get_lun function and updates the gc_endtime using next_lun_avail_time variable.

```c
static void mark_page_valid(struct ssd *ssd, struct ppa *ppa)
{
    struct nand_block *blk = NULL;
    struct nand_page *pg = NULL;
    struct line *line;

    /* update page status */
    pg = get_pg(ssd, ppa);
    ftl_assert(pg->status == PG_FREE);
    pg->status = PG_VALID;

    /* update corresponding block status */
    blk = get_blk(ssd, ppa);
    ftl_assert(blk->vpc >= 0 && blk->vpc < ssd->sp.pgs_per_blk);
    blk->vpc++;

    /* update corresponding line status */
    line = get_line(ssd, ppa);
    ftl_assert(line->vpc >= 0 && line->vpc < ssd->sp.pgs_per_line);
    line->vpc++;
}
```

-In the mark_page_valid function, we will get our page using get_pg function and check the integrity and updates its status. We will also get the block for our page using get_blk function and check the integrity and increments the number of valid page in our block. We will also update our line by getting line using get_line function and check the integrity using ftl_assert function and updates the number of valid page for our line.

```
static void ssd_advance_write_pointer(struct ssd *ssd)
{
    struct ssdparams *spp = &ssd->sp;
    struct write_pointer *wpp = &ssd->wp;
    struct line_mgmt *lm = &ssd->lm;

    check_addr(wpp->ch, spp->nchs);
    wpp->ch++;
    if (wpp->ch == spp->nchs) {
        wpp->ch = 0;
        check_addr(wpp->lun, spp->luns_per_ch);
        wpp->lun++;
        /* in this case, we should go to next lun */
        if (wpp->lun == spp->luns_per_ch) {
            wpp->lun = 0;
            /* go to next page in the block */
            check_addr(wpp->pg, spp->pgs_per_blk);
            wpp->pg++;
            if (wpp->pg == spp->pgs_per_blk) {
                wpp->pg = 0;
```

-This ssd_advance_write_pointer function advances the write pointer to make the pointer points the next consecutive page. First it check the channel index is in valid range(uses the ftl_assert function). Then it increases the channel index and if the increased channel index is same with the number of channels, it initializes the number of channel index. Before we increases the LUN, we will check if the LUN is in the valid range. After this, we will increase the LUN and if the increased LUN is same with the number of LUN index, we will initalize the number of LUN index. Than, we will check the page index integrity same with the others and add afterward.

```
/* move current line to {victim,full} line list */
if (wpp->curline->vpc == spp->pgs_per_line) {
    /* all pgs are still valid, move to full line list */
    ftl_assert(wpp->curline->ipc == 0);
    QTAILQ_INSERT_TAIL(&lm->full_line_list, wpp->curline, entry);
    lm->full_line_cnt++;
} else {
    ftl_assert(wpp->curline->vpc >= 0 && wpp->curline->vpc < spp->pgs_per_line);
    /* there must be some invalid pages in this line */
    ftl_assert(wpp->curline->ipc > 0);
    pqueue_insert(lm->victim_line_pq, wpp->curline);
    lm->victim_line_cnt++;
}
```

-If the number of valid page in the line is same with the number of lines per page, after we check number of invalid page is 0 using ftl_assert, we will add our line into the tail of the full_line_list and increases the full_line_cnt as well. If the condition is unsatisfied, it checks the if there is 1 or more valid pages for our line and checks the number of valid pages are smaller than the number of pages per line. After, it checks if there is one or more invalid pages. It inserts our line into victim priority queue and increments the number of victim_line_cnt.

```
/* current line is used up, pick another empty line */
check_addr(wpp->blk, spp->blks_per_pl);
wpp->curline = NULL;
wpp->curline = get_next_free_line(ssd);
if (!wpp->curline) {
    /* TODO */
    abort();
}
wpp->blk = wpp->curline->id;
check_addr(wpp->blk, spp->blks_per_pl);
/* make sure we are starting from page 0 in the super block */
ftl_assert(wpp->pg == 0);
ftl_assert(wpp->lun == 0);
ftl_assert(wpp->ch == 0);
/* TODO: assume # of pl_per_lun is 1, fix later */
ftl_assert(wpp->pl == 0);
```

-After the updates, we check if the block index is within the valid range and initializes the next to write pointer. Than we find other empty line using get_next_free_line function and updates the next to write pointer(curline). After updating id, we check the block integrity again to check with the update we've made. Before finishing, we check if our updated block starts from 0 in page, LUN, channel and plane.

```c
static struct line *get_next_free_line(struct ssd *ssd)
{
    struct line_mgmt *lm = &ssd->lm;
    struct line *curline = NULL;

    curline = QTAILQ_FIRST(&lm->free_line_list);
    if (!curline)
    {
        ftl_err("No free lines left in [%s] !!!!\n", ssd->ssdname);
        return NULL;
    }

    QTAILQ_REMOVE(&lm->free_line_list, curline, entry);
    lm->free_line_cnt--;
    return curline;
}
```

-We will see the get_next_free_line function before going back. We will get the free line from the tail of the free_line_list. If we can't get any, this function will return NULL. But if it is not, which means we got free line, we will update the free_line_list and updates the free_line_cnt as well and return the found free line.

```c
static void mark_block_free(struct ssd *ssd, struct ppa *ppa)
{
    struct ssdparams *spp = &ssd->sp;
    struct nand_block *blk = get_blk(ssd, ppa);
    struct nand_page *pg = NULL;

    for (int i = 0; i < spp->pgs_per_blk; i++)
    {
        /* reset page status */
        pg = &blk->pg[i];
        ftl_assert(pg->nsecs == spp->secs_per_pg);
        pg->status = PG_FREE;
    }

    /* reset block status */
    ftl_assert(blk->npgs == spp->pgs_per_blk);
    blk->ipc = 0;
    blk->vpc = 0;
    blk->erase_cnt++;
}
```

-Back to the do_gc function, after clean_one_block function, we've seen there was mark_block_free func tion. At the clean_one_block function, we've moved all the valid pages from our victim block. So this function will mark our victim block as free. for each pages in our victim block, we will get the pages into the pg variable for each iteration and after checking integrity using ftl_assert, it will mark page as PG_FREE, which means free page. After marking all the pages free in our victim block, We will check the integrity of our block and initailize the numbers of invalid page and valid pages number in our victim block and increment the number of times the block was erased.

```
            if (spp->enable_gc_delay)
            {
                struct nand_cmd gce;
                gce.type = GC_IO;
                gce.cmd = NAND_ERASE;
                gce.stime = 0;
                ssd_advance_status(ssd, &ppa, &gce);
            }

            lunp->gc_endtime = lunp->next_lun_avail_time;
        }
    }
```

-After all the operations. back to do_gc function, if gc delay is allowed(which is set to true by default), It also updates delay time using ssd_advance_status function(In this case we will look into NAND_ERASE case). After, it updates the end time including the delay if there is any.

```
case NAND_ERASE:
    /* erase: only need to advance NAND status */
    nand_stime = (lun->next_lun_avail_time < cmd_stime)
                    ? cmd_stime
                    : lun->next_lun_avail_time;
    lun->next_lun_avail_time = nand_stime + spp->blk_er_lat;
    lat = lun->next_lun_avail_time - cmd_stime;
    break;
```

-Before we've looked into NAND_READ and NAND_WRITE. But in this case, since we want to calculate the latency of erase, we've used NAND_ERASE. Basically it is similar with other cases. It records the nand command start time in the nand_stime variable and add it with blk_er_lat(block erase latency in nano seconds) and store it into next_lun_avail_time. After, it subtracts the cmd start time(cmd_stime) and stores result in the lat variable and return it.

```
    /* update line status */
    mark_line_free(ssd, &ppa);

    return 0;
}
```

-Now finally, we use mark_line_free function to update the line status and end the do_gc function.

```
static void mark_line_free(struct ssd *ssd, struct ppa *ppa)
{
    struct line_mgmt *lm = &ssd->lm;
    struct line *line = get_line(ssd, ppa);
    line->ipc = 0;
    line->vpc = 0;
    /* move this line to free line list */
    QTAILQ_INSERT_TAIL(&lm->free_line_list, line, entry);
    lm->free_line_cnt++;
}
```

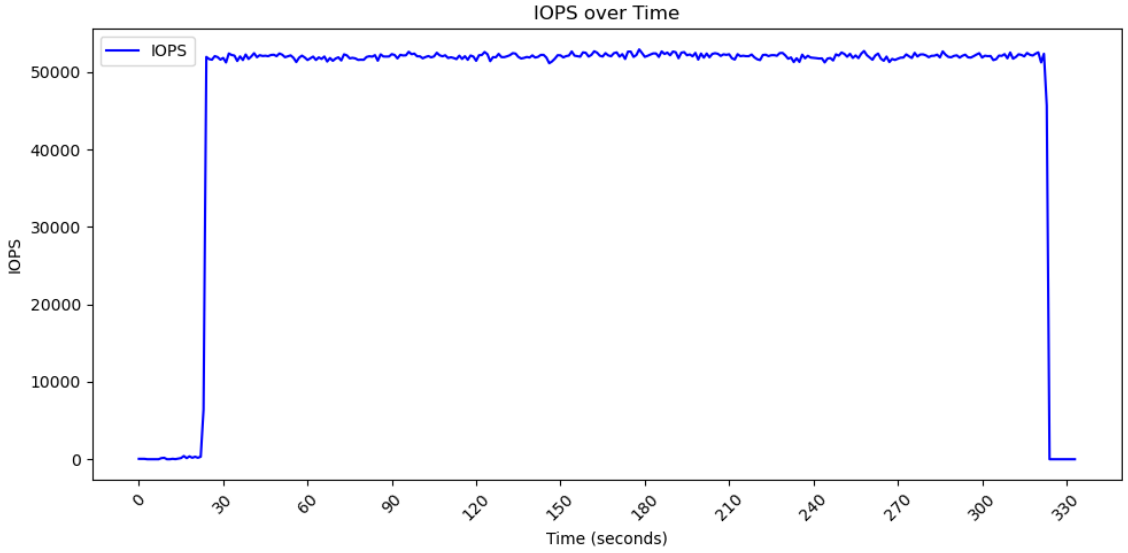
-We've initialized the number of invalid pages and valid pages in the block so we need to update the number of invalid pages and valid pages in each line as well. We get the line using get_line function and initialize the number of invalid pages and valid pages. After doing so, we move the line into the free line list tail and update the number of free_line_cnt.

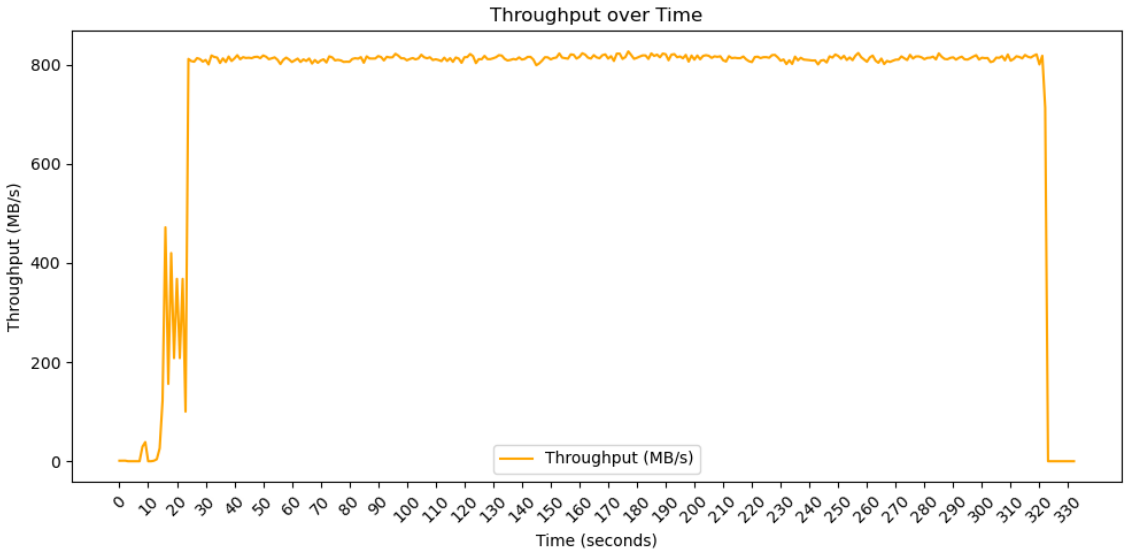
(B) Performance analysis results(IOPS and GC) using benchmarks

Environment: 4GB SSD Setting

Fio

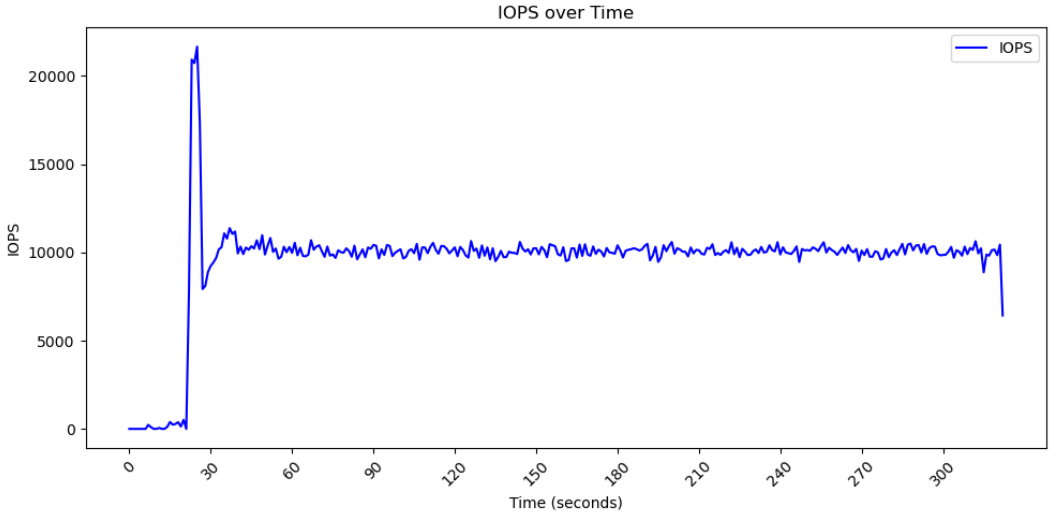
(i) Read 100%

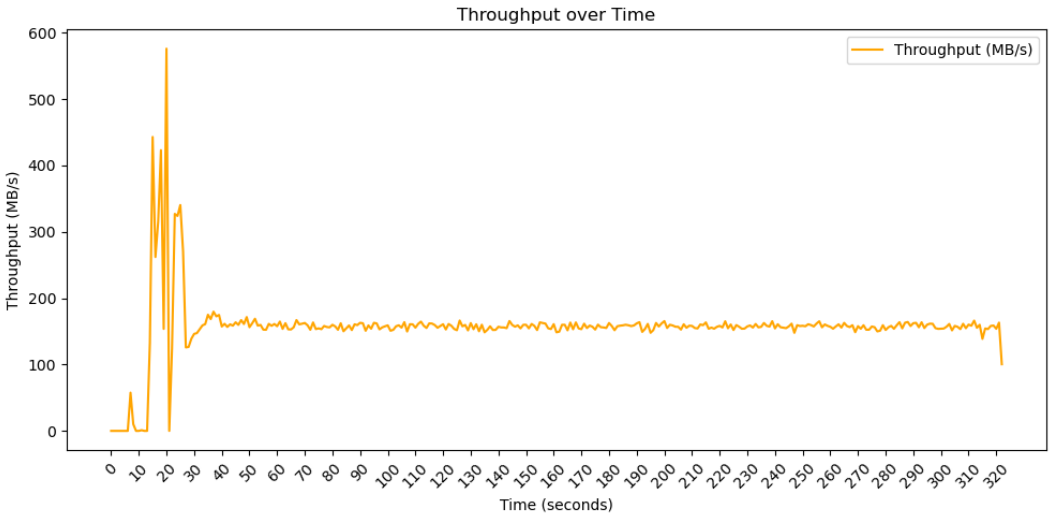




-At the initial reading operation part, the throughput part it lower because of warm up phase but after warm-up phase, both IOPS and throughtput shows high performance because this test is only includes reading part.

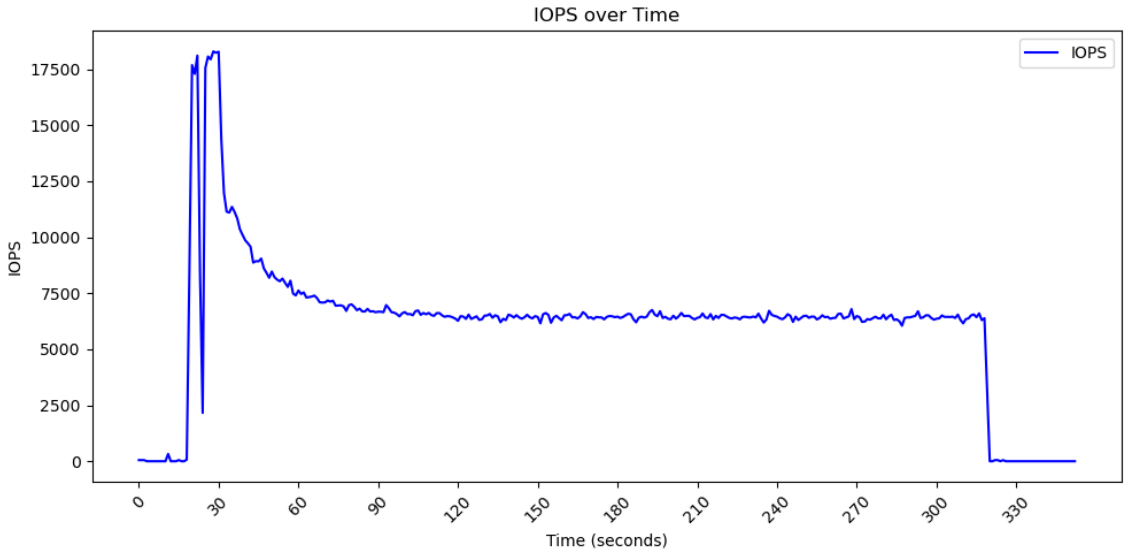
(ii) Read 50%/Write 50%

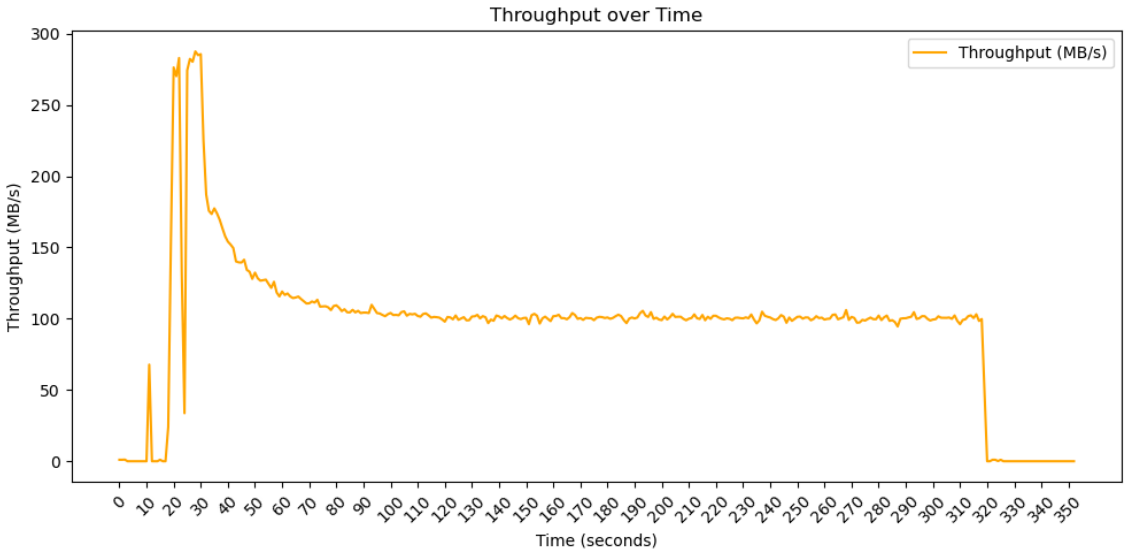




-At first, since it has enough space, it shows significantly high performance both on IOPS and throughput. But because this test includes write, it triggers GC and as a result, it dramatically drops due to continuous GC. The GC is invoked throughout the test because of the test we operate. GC is invoked according to the threshold which are 25% and 5%. The total page numbers is blocks*planes*channels, which is 256*256*2=131072 in the given 4GB environment. GC threshold and high GC threshold will be 131072*0.25=32768 lines, 131072*0.05=6554 lines respectively. Since each line is approximately 4KB so the thresholds are 128MB and 25MB approximately. The current fio test we are looking at is 50% write so the average IOP would be approximately 5000, and the test we've ran as block size of 16KB so it would be 80MB. Since write has high gc threshold, it should be taken as a granted that every second the GC will be invoke.(80MB>25MB) And the GC is more likely to occur because this test is random write and read. So after the initial part, both of the graph shows low performance. Also stabilized because GC and write operation got into stable state.

(iii) Write 100%

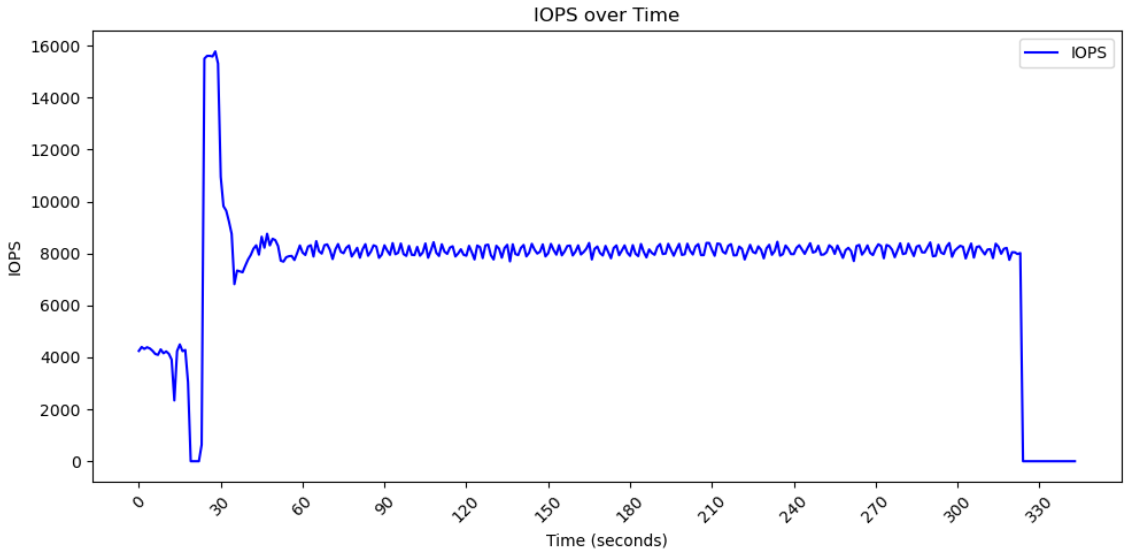


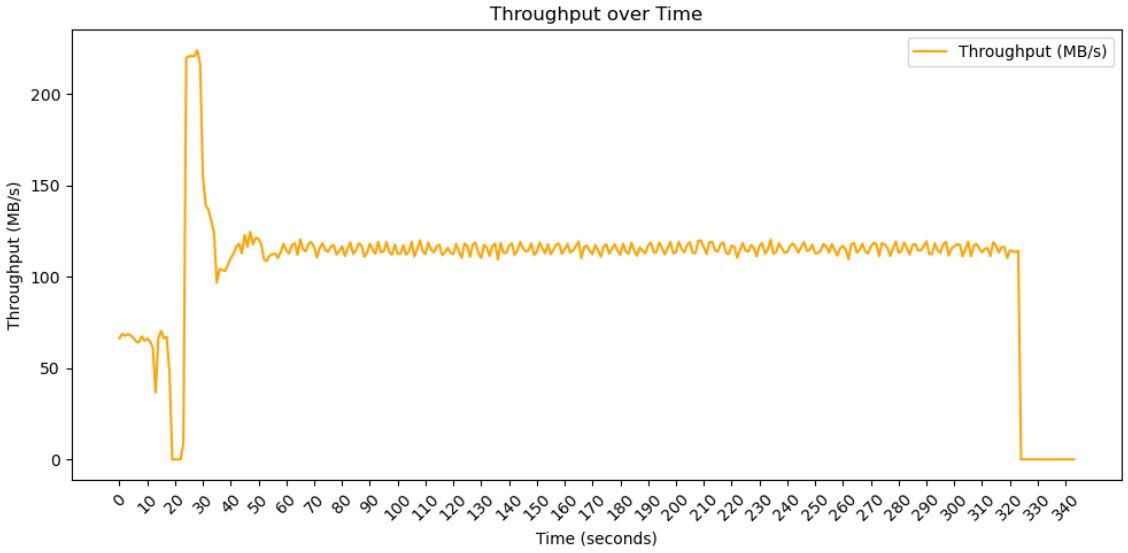


-Since this is only write operation, same as the other tests, the initial part shows good performance. But As you can see after the initial part, the performance drops significantly due to GC and keeps stable state but shows low performance both on IOPS and throughput.

Sysbench

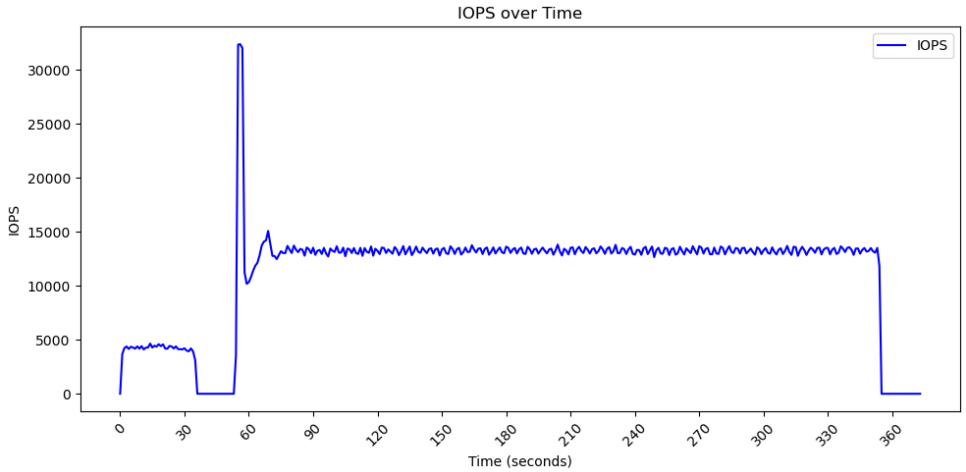
(iv) 4 Threads

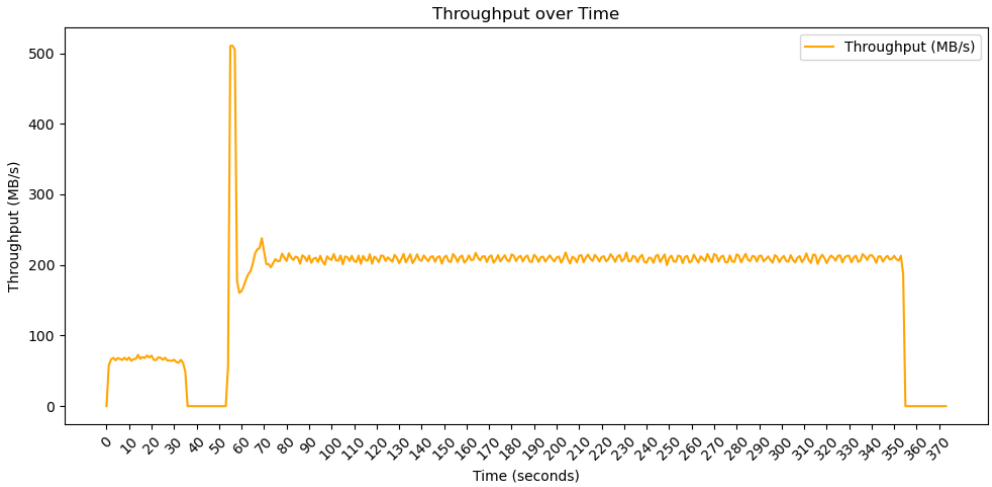




-Actually this test is almost the same with the fio read/write test. Because at the fio test, it uses 4 threads also. Anyway, is shows high performance at the initial part. But after GC invoked, the GC is continuously invoked. At the certain point, which is approximately 40 seconds in this case, the write operation and GC got into stable state and shows low performance.

(v) 32 Threads



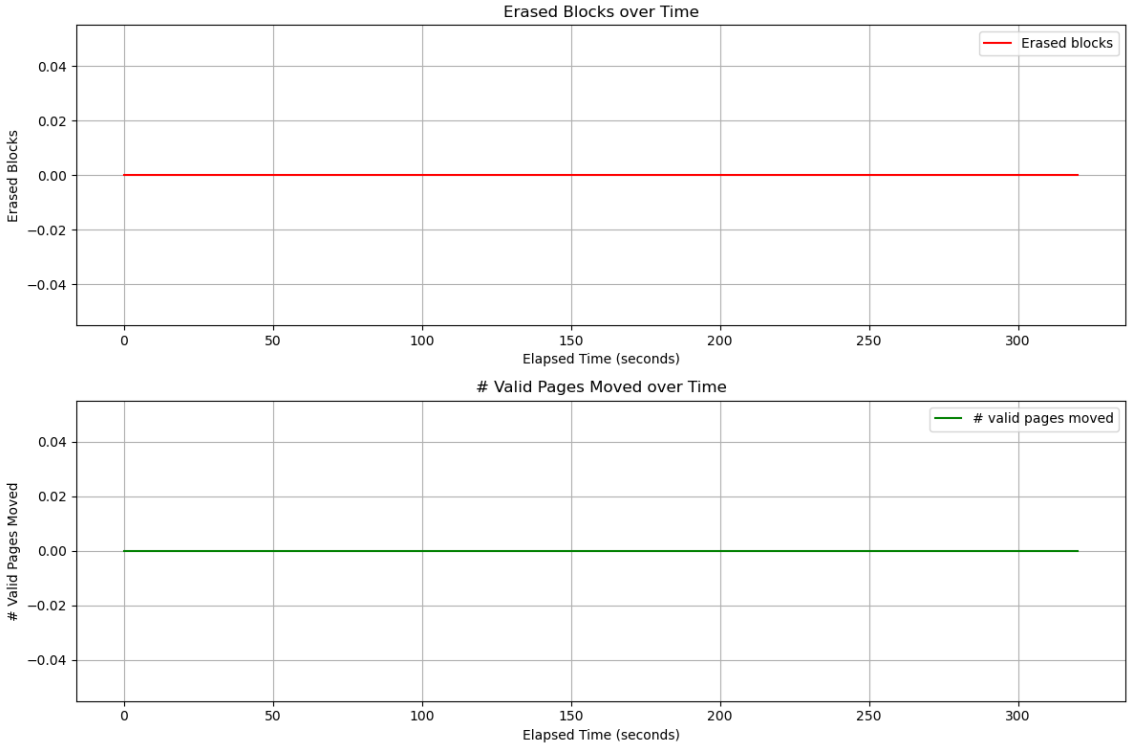


-(The inital 0~40 seconds is setting up the environment. The test actually started from 50 seconds) Since this test uses more threads, we should much higher performance. As you can see, After initial part, same as the others. shows great performance but after GC started get invoked, the performance drops but still shows descent performance even if it is stabilized.

-GC

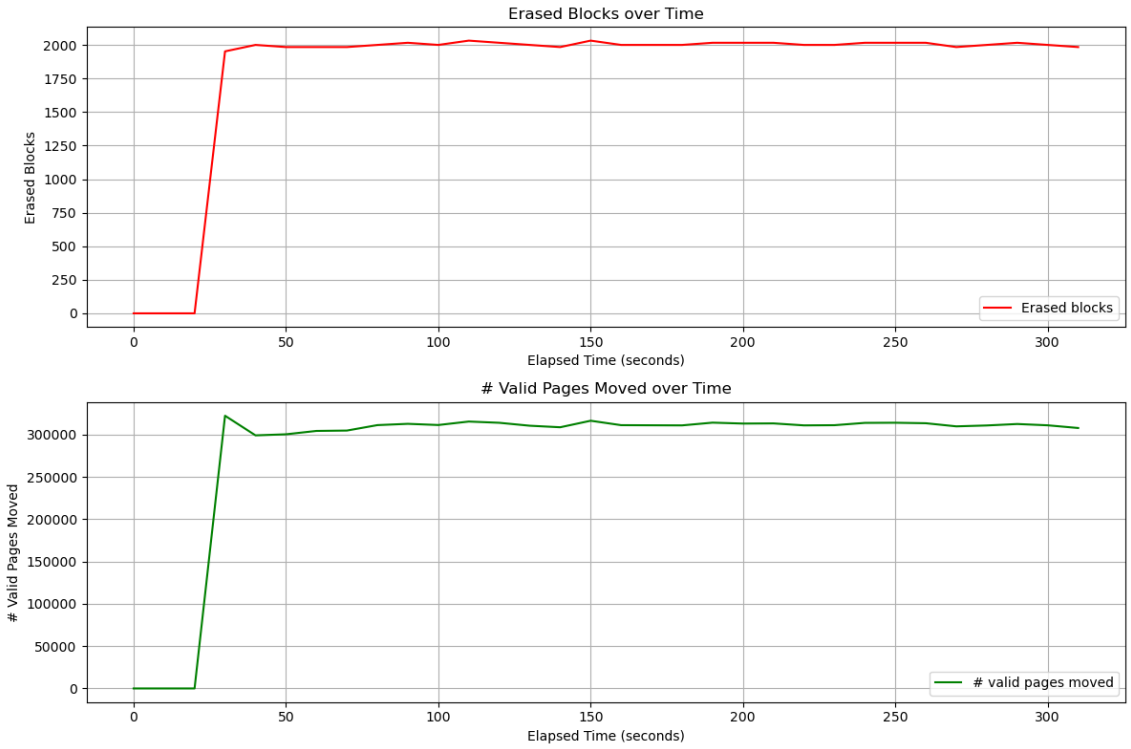
Fio

-Read 100%



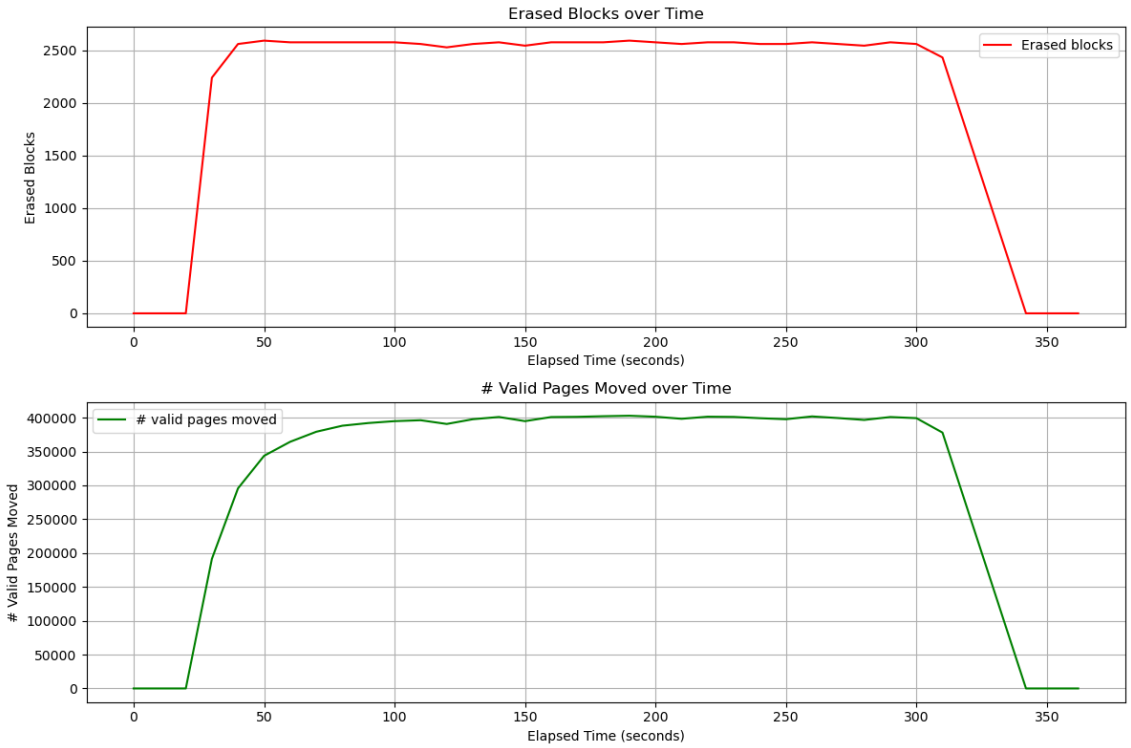
-Since this is read 100%, as you can see there was no GC throughout the test. As you can see, since GC is not invoked, there is no need to move valid pages.

-Read 50% / Write 50%



-At the earlier throughput and IOPS graph, we've discussed that GC has been invoked throughout the test. So the graph shows high number of erased blocks throughout the graph. For GC to erase blocks properly, the valid page should be moved as well.

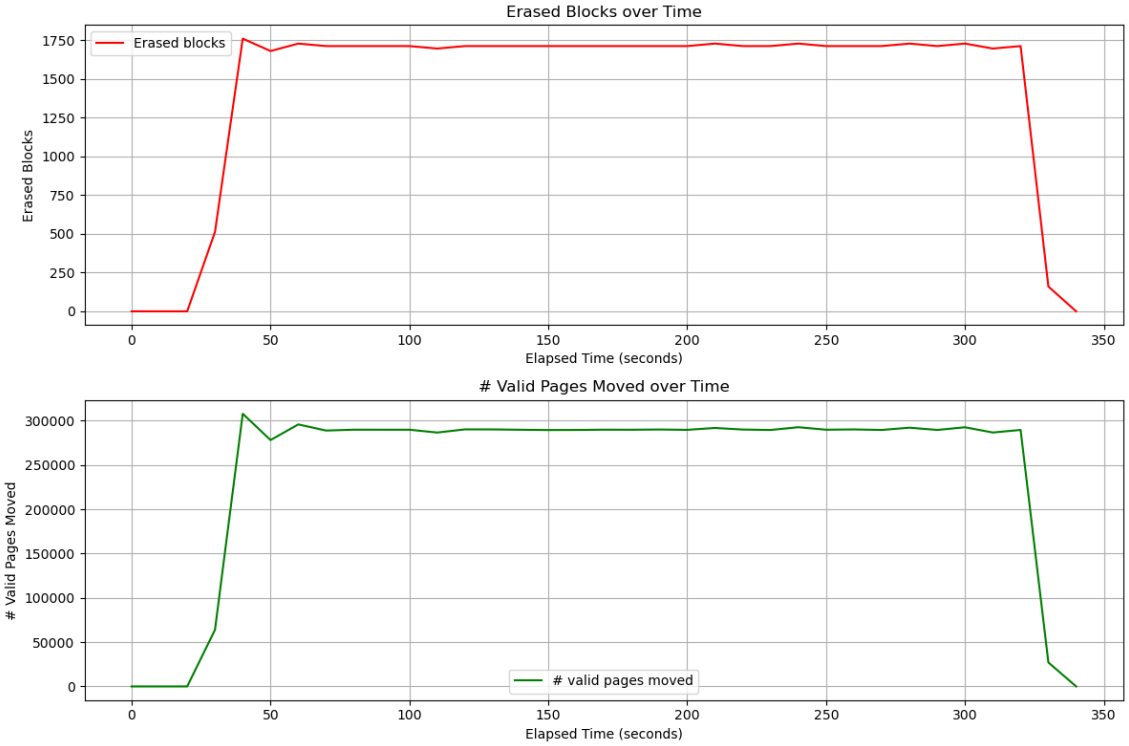
-Write 100%



-This test only operates write so it shows much more erased block than the read/write 50% test throughout the test compared to read/write test. Since GC created more erased block, the amount of number of valid pages moved should be increased as well and it did.

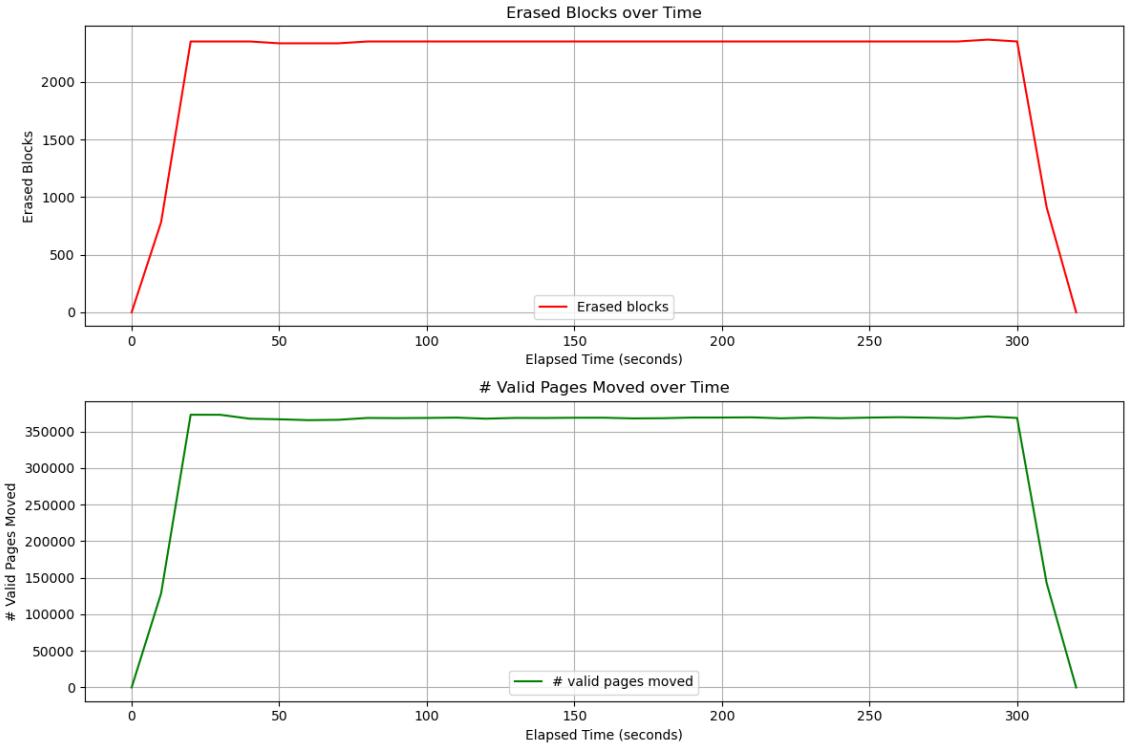
Sysbench

-4 Threads



-As we've seen in the IOPS and throughput graph earlier, the 4 threads sysbench test graph was similar with the read/write fio test graph. So I expected similar GC performance as well and it is actually quite similar. GC was invoked throughout the test and valid pages are moved as GC invoked.

-32 Threads



-We've seen throughput and IOPS graph that when we increased threads, it showed higher performance. But to show higher performance without increasing size of the SSD, SSD should erase much more blocks to obtain free space. So as you can see in the graph, the number of erased block throughout the test are higher than the 4 threads test. Since erased blocks increased, the number of valid pages moved should be increased as well.

(C) Explanation of your implementation for your I/O statistics rountine

```
static uint64_t io_count = 0;
static uint64_t throughput = 0;
uint64_t pages_moved = 0;
uint64_t erased_blocks = 0;

static void statistics_record(int sig);
static void timer(void);
```

-I've created 2 functions and 4 variables at nvme-io.c file to calculate the I/O and we will look into it. I will be using pages_moved(the number of pages moved) and erased_blocks(number of erased block) in ftl.c file. Since I will use io_count(the number of I/O count) and throughput(the I/O data size) in same class file, I've declared in static.

```
static void timer(void)
{
    struct sigaction sig;
    struct itimerval timer;

    sig.sa_handler = &statistics_record;
    sigaction(SIGALRM, &sig, NULL);

    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = 1;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_usec = 0;

    setitimer(ITIMER_REAL, &timer, NULL);
}
```

-To record our statistics by seconds, we create timer function. We start our timer as soon as we start when we call the function timer from other function which will be nvme_poller function(will show code below). We set start time using timer.it_value(usec is micro second). We start timer using setitimer function giving parameters as real time and our timer set(start time and interval). We will be sending signal in 1 second interval using timer.it_interval(sec is second) to statistic_record. We decide type of signal at sigaction function(this case sending alarm). Then We bind signal handler with statistics_record function which will be

function that actually calculate all the I/O statistics(statistics_record function).

```
void *nvme_poller(void *arg)
{
    FemuCtrl *n = ((NvmePollerThreadArgument *)arg)->n;
    int index = ((NvmePollerThreadArgument *)arg)->index;
    int i;
    timer();
```

-We know that nvme_poller function invokes when there is queue in the to_poller queue where is stacks operated I/O, so I've placed timer function at nvme_poller function.

```
static void statistics_record(int sig)
{
    static int counter = 0;

    FILE *iops_statistic = fopen("/home/sungtv2613/iops_statistics.txt", "a");
    if (iops_statistic == NULL) {
        printf("Failed to open file");
        return;
    }

    FILE *gc_statistic = fopen("/home/sungtv2613/gc_statistics.txt", "a");
    if (gc_statistic == NULL) {
        printf("Failed to open file");
        return;
    }
```

-This is initial part of the statistics_record function receiving which is opening file part

I've divided file into 2 part the iops_statistics text file will include IOPS and throughput.

In the gc_statistic, the text file will include erased blocks and # of valid pages moved in 10 seconds interval(the counter variable is for 10 seconds interval).

```c
time_t now;
struct tm *timeinfo;
char time_str[20];

time(&now);
timeinfo = localtime(&now);
strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", timeinfo);
```

-This will be the time structures to record time.

Now variable will be recorded with current time in seconds.

So we will translate it into local time zone using localtime function and allocate in timeinfo variable

we will change into string type store in variable time_str in a readable value.

```c
    double throughput_mb = (double)throughput / (1024 * 1024);

    fprintf(iops_statistic,"[%s] IOPS: %lu, Throughput: %.2f MB/s\n", time_str, io_count, throughput_mb);

    if (++counter >= 10)
    {
        fprintf(gc_statistic,"[%s] Erased blocks: %lu, # valid pages moved: %lu\n", time_str, erased_blocks, pages_moved);
        erased_blocks = 0;
        counter = 0;
        pages_moved = 0;
    }

    io_count = 0;
    throughput = 0;
    fclose(iops_statistic);
    fclose(gc_statistic);
}
```

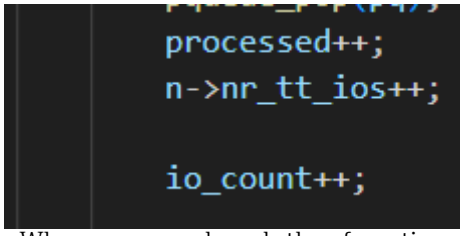
-This is essential part, the calculating part.

The throughput(we will discuss later where is calculated) is byte unit so we will be converting byte unit into MB so we will divide by 1024*1024 and store the throughput value in throughput_mb.

We will be storing IOPS and throughput in iops_statistic file each time we call function which is 1 second interval by the timer function.(We will discuss io_count later)

Since we call function by 1 second interval, when the counter becomes 10 or larger, it means time has gone more than equal to or more than 10 seconds so we will be recording erased block and pages moved in gc_statistics file. We don't want values to be accumulated so we initializes the values.(The erased_blocks and pages_moved will be explained later)

Each time we go through the function, we initializes the values and close the file.



-When we analyzed the functions for read I/O operation and write I/O operation, we've noticed that when I/O operation was successfully made, we've updated that total number of I/O in the variable nr_tt_ios at the nvme_proccess_cq_cpl function. So we can get the number of I/O simply adding a io_count++ line under that line.

```
uint16_t nvme_rw(FemuCtrl *n, NvmeNamespace *ns, NvmeCmd *cmd, NvmeRequest *req)
{
    NvmeRwCmd *rw = (NvmeRwCmd *)cmd;
    uint16_t ctrl = le16_to_cpu(rw->control);
    uint32_t nlb = le16_to_cpu(rw->nlb) + 1;
    uint64_t slba = le64_to_cpu(rw->slba);
    uint64_t prp1 = le64_to_cpu(rw->prp1);
    uint64_t prp2 = le64_to_cpu(rw->prp2);
    const uint8_t lba_index = NVME_ID_NS_FLBAS_INDEX(ns->id_ns.flbas);
    const uint16_t ms = le16_to_cpu(ns->id_ns.lbaf[lba_index].ms);
    const uint8_t data_shift = ns->id_ns.lbaf[lba_index].lbads;
    uint64_t data_size = (uint64_t)nlb << data_shift;
    uint64_t data_offset = slba << data_shift;
    uint64_t meta_size = nlb * ms;
    uint64_t elba = slba + nlb;
    uint16_t err;
    int ret;

    throughput += data_size;
```

-At the nvme_rw, this function receives read and write operation and calculates data size and does physical addressing(get logical address and maps into physiccal address). Since this function gives read and write I/O data size, I've inserted in throughput variable to calculate throughput.

FTL.C

```
extern uint64_t erased_blocks;
extern uint64_t pages_moved;
```

-Since we can get this values in ftl.c file, and we also want to use the variable in the nvme-io.c file, I've declared in external variable in the ftl.c file.

```
static void clean_one_block(struct ssd *ssd, struct ppa *ppa)
{
    struct ssdparams *spp = &ssd->sp;
    struct nand_page *pg_iter = NULL;
    int cnt = 0;

    erased_blocks++;
```

-At the clean_one_block function, we know this function is invoked to clean block(We don't need to worry about failure because this function also moves valid pages into other block)

```
    for (int pg = 0; pg < spp->pgs_per_blk; pg++)
    {
        ppa->g.pg = pg;
        pg_iter = get_pg(ssd, ppa);
        ftl_assert(pg_iter->status != PG_FREE);
        if (pg_iter->status == PG_VALID)
        {
            gc_read_page(ssd, ppa);
            gc_write_page(ssd, ppa);
            cnt++;
            pages_moved++;
        }
    }
```

-This is located within the clean_one_block function, which works as moving valid page into the other empty page. So I've incremented pages_moved variable in this iteration.