FEMU Analysis

Computer Science

Chaeheon Lee

Contents

I would appreciate it if you could check both the parts written in small letters and the parts written in large letters.

(Preview) How is the SSD structured?

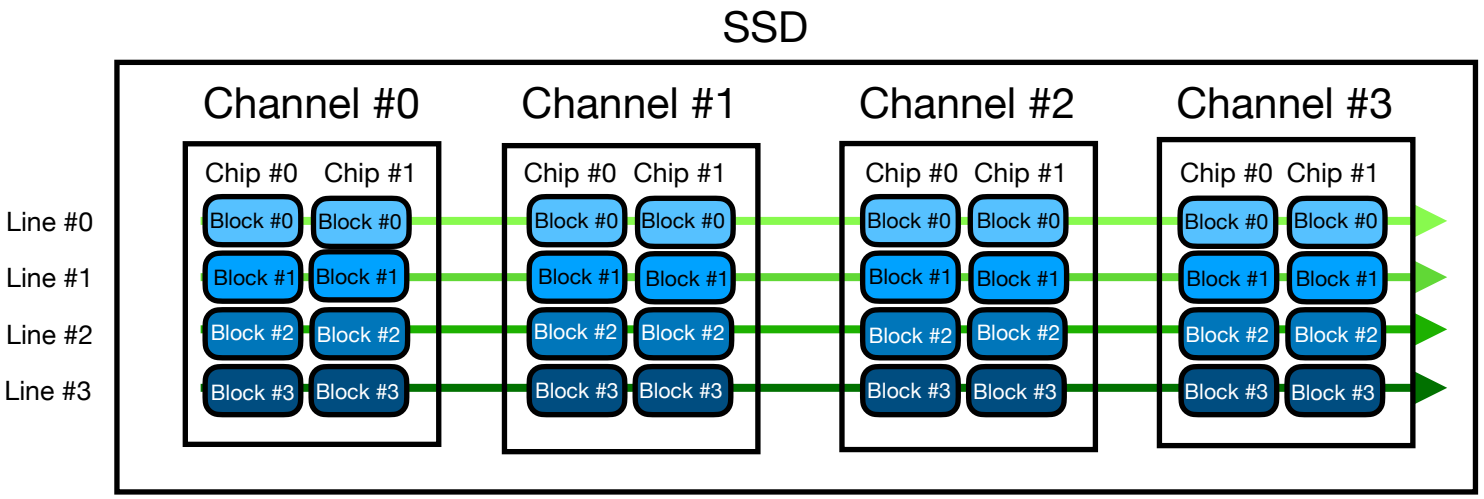
Before analyzing how to read and write in FEMU,
First, let's explain about the **SSD architecture**.

An SSD is made up of multiple chips.
(**Each chip can handle I/O,**
So, parallel processing is possible using multiple chips,
which can improve performance.)

Also, these chips contain multiple planes.
Multiple planes again contain multiple blocks,
the block is again made up of several pages.

Channel is the bus used to send commands to each chip.

(Preview) About the line

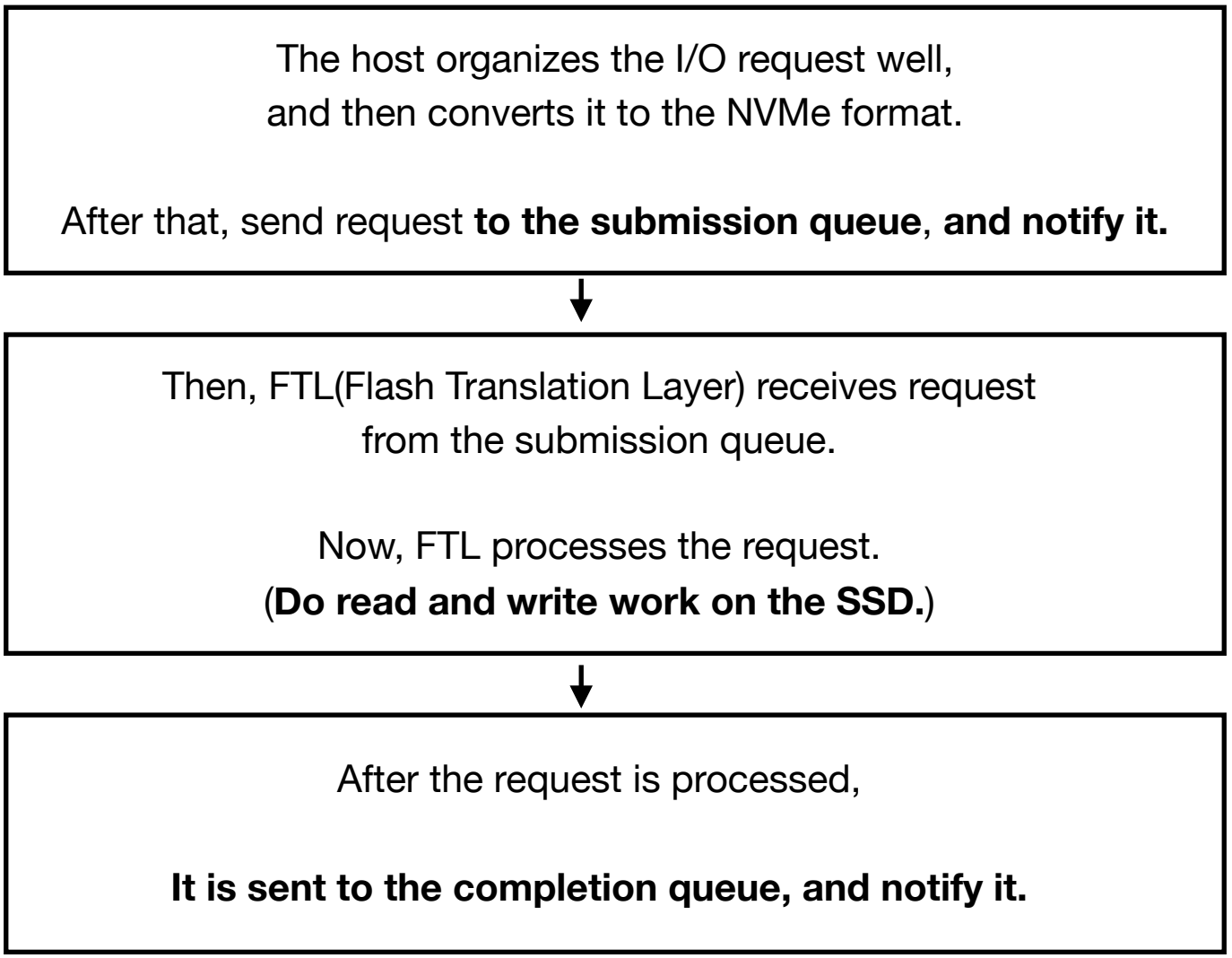


There is something in FEMU that uses multiple chips at once. It's **line**.

In the picture above, if you look at line #0,
You can see that **eight block #0 are tied to line #0.**

If **all block** in line #0 is used, then line #1 handles I/O.
Next, line #2, and then line #3 plays the role.

(Preview) How is I/O handled on an SSD? (NVMe)



(Preview) Explanation of various terms

LBA	Logical Block Address, which means one sector.	OS
LPN	Logical Page Number, grouping multiple sectors into SSD page sizes.	SSD
PPA	Physical Page Address, the address where the actual data is stored.	SSD
Mapping table	Serves as a link to the PPA, for the physical location of the LPN.	

(Preview) A description of the role of each file

nvme-io.c	Check for I/O requests originating on the host, It serves to pass request in the submission queue to FTL.
ftl.c	Process the request received with nvme-io.c, and then send it back to nvme-io.c. It serves as the FTL of the SSD. (I/O execution, mapping table management, GC)

(A) Explanation of code analysis for **(i) read, (ii) write**

nvme-io.c

***nvme_poller** is running infinite loop in **default:** because the multipoller is turned off by default.

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

    switch (n->multipoller_enabled) {
    case 1:
        while (1) {
            if ((!n->dataplane_started)) {
                usleep(1000);
                continue;
            }

            NvmeSQueue *sq = n->sq[index];
            NvmeCQueue *cq = n->cq[index];
            if (sq && sq->is_active && cq && cq->is_active) {
                nvme_process_sq_io(sq, index);
            }
            nvme_process_cq_cpl(n, index);
        }
        break;
    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;
}
```

1

Multipoller is disabled.

Data plane check

Check for all I/O queues

Make sure both queues are active

2

A description of the next process can be found **on the upcoming page.**

nvme-io.c

nvme_process_sq_io identifies the I/O request in the submission queue, and then puts it in the to_ftl.

```
static void nvme_process_sq_io(void *opaque, int index_poller)   2
{
    NvmeSQueue *sq = opaque;
    FemuCtrl *n = sq->ctrl;

    uint16_t status;
    hwaddr addr;
    NvmeCmd cmd;
    NvmeRequest *req;
    int processed = 0;

    nvme_update_sq_tail(sq);   Submission queue update
    while (!(nvme_sq_empty(sq))) {   Repeat until the submission queue is empty
        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);   The next request preparation process

        req = QTAILQ_FIRST(&sq->req_list);   Getting the first request
        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));   Copy command

        if (n->print_log) {
            femu_debug("%s,cid:%d\n", __func__, cmd.cid);
        }

        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;
}
```

Number 3 is the process of executing the command.

3

nvme-io.c

nvme_io_cmd handles each command.

```
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];

    switch (cmd->opcode) {
    case NVME_CMD_FLUSH:
        if (!n->id_ctrl.vwc || !n->features.volatile_wc) {
            return NVME_SUCCESS;
        }
        return nvme_flush(n, ns, cmd, req);
    case NVME_CMD_DSM:
        if (NVME_ONCS_DSM & n->oncs) {
            return nvme_dsm(n, ns, cmd, req);
        }
        return NVME_INVALID_OPCODE | NVME_DNR;
    case NVME_CMD_COMPARE:
        if (NVME_ONCS_COMPARE & n->oncs) {
            return nvme_compare(n, ns, cmd, req);
        }
        return NVME_INVALID_OPCODE | NVME_DNR;
    case NVME_CMD_WRITE_ZEROES:
        if (NVME_ONCS_WRITE_ZEROS & n->oncs) {
            return nvme_write_zeros(n, ns, cmd, req);
        }
        return NVME_INVALID_OPCODE | NVME_DNR;
    case NVME_CMD_WRITE_UNCOR:
        if (NVME_ONCS_WRITE_UNCORR & n->oncs) {
            return nvme_write_uncor(n, ns, cmd, req);
        }
        return NVME_INVALID_OPCODE | NVME_DNR;
    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;
    }
}
```

3

Error handling

Handling each command

It is designed to be handled separately according to each command.

If a read or write operation is executed, it will execute the part in the **default:**.

Error handling

nvme-io.c

In FEMU, Store the actual data separately.
(This is the difference from the actual SSD.)

```
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;

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

    return NVME_DNR;
}
```

4

Read/write classification

It is the process of storing the actual data.

5

backend_rw is defined in dram.c.

dram.c

backend_rw is a function that stores **actual data**.
This function is defined in **dram.c**.

```
int backend_rw(SsdDramBackend *b, QEMUSGList *qsg, uint64_t *lbal, bool is_write)
{
    int sg_cur_index = 0;
    dma_addr_t sg_cur_byte = 0;
    dma_addr_t cur_addr, cur_len;
    uint64_t mb_oft = lbal[0];
    void *mb = b->logical_space;

    DMADirection dir = DMA_DIRECTION_FROM_DEVICE;

    if (is_write) {
        dir = DMA_DIRECTION_TO_DEVICE;
    }

    while (sg_cur_index < qsg->nsg) {
        cur_addr = qsg->sg[sg_cur_index].base + sg_cur_byte;
        cur_len = qsg->sg[sg_cur_index].len - sg_cur_byte;
        if (dma_memory_rw(qsg->as, cur_addr, mb + mb_oft, cur_len, dir, MEMTXATTRS_UNSPECIFIED)) {
            femu_err("dma_memory_rw error\n");
        }

        sg_cur_byte += cur_len;
        if (sg_cur_byte == qsg->sg[sg_cur_index].len) {
            sg_cur_byte = 0;
            ++sg_cur_index;
        }

        if (b->femu_mode == FEMU_OCSSD_MODE) {
            mb_oft = lbal[sg_cur_index];
        } else if (b->femu_mode == FEMU_BBSSD_MODE ||
                   b->femu_mode == FEMU_NOSSD_MODE ||
                   b->femu_mode == FEMU_ZNSSD_MODE) {
            mb_oft += cur_len;
        } else {
            assert(0);
        }
    }

    qemu_sglist_destroy(qsg);

    return 0;
}
```

5

Setting the direction according to reading and writing

nvme-io.c

nvme_process_sq_io identifies the I/O request in the submission queue, and then puts it in the to_ftl.

```c
static void nvme_process_sq_io(void *opaque, int index_poller)
{
    NvmeSQueue *sq = opaque;
    FemuCtrl *n = sq->ctrl;

    uint16_t status;
    hwaddr addr;
    NvmeCmd cmd;
    NvmeRequest *req;
    int processed = 0;

    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));

        if (n->print_log) {
            femu_debug("%s,cid:%d\n", __func__, cmd.cid);
        }

        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;
}
```

2

Submission queue update

Repeat until the submission queue is empty

The process for processing the next request

Getting the first request

Copy command

3

Error handling

to ftl

6

Error handling

Processed reflect

I have explained all the processes from number 1 to 5.

Now it goes through the process of forwarding I/O requests to FTL.

ftl.c

***ftl_thread** is responsible for verifying the command type and then performing the task **when it receives I/O command.**

```
static void *ftl_thread(void *arg)
{
    FemuCtrl *n = (FemuCtrl *)arg;
    struct ssd *ssd = n->ssd;
    NvmeRequest *req = NULL;
    uint64_t lat = 0;
    int rc;
    int i;

    while (!*(ssd->dataplane_started_ptr)) {
        usleep(100000);
    }

    /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    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");
                ;
            }

            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;
}
```

6

This is the task of ejecting request from the queue.

Pull out request

Performs write and read operations.

7-write

Performing a writing task

7-read

Performing a reading task

Reflection of latency time

to poller

GC will be described in (iii) the GC mechanism!

The descriptions on this page are all descriptions of write.

ftl.c

ssd_write handles write requests.

```
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);
        }

        /* 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;
}
```

7.0-write

LBA to LPN

When judged, if a GC is needed, it is defined to do GC.

Perform GC if necessary

Perform writing for each page

For each LPN, PPA(Physical Page Address) is fetched.

Invalid mark

PPA validation

New PPA

Update mapping table

Update reverse mapping table

Valid mark

7.1-write

This is the part that calculates the latency.

(Each PPA write latency)

8-write

When calculating latency, a greater value is maxlat.

Returns maximum latency

ftl.c

ssd_advance_write_pointer serves to set the pointer. It is involved in a lot of the configuration of channel, LUN, page, and line.

```
157 static void ssd_advance_write_pointer(struct ssd *ssd)
158 {
159     struct ssdparams *spp = &ssd->sp;
160     struct write_pointer *wpp = &ssd->wp;
161     struct line_mgmt *lm = &ssd->lm;
162
163     check_addr(a: wpp->ch, max: spp->nchs);
164     wpp->ch++;
165     if (wpp->ch == spp->nchs) {
166         wpp->ch = 0;
167         check_addr(a: wpp->lun, max: spp->luns_per_ch);
168         wpp->lun++;
169         /* in this case, we should go to next lun */
170         if (wpp->lun == spp->luns_per_ch) {
171             wpp->lun = 0;
172             /* go to next page in the block */
173             check_addr(a: wpp->pg, max: spp->pgs_per_blk);
174             wpp->pg++;
175             if (wpp->pg == spp->pgs_per_blk) {
176                 wpp->pg = 0;
177                 /* move current line to {victim,full} line list */
178                 if (wpp->curline->vpc == spp->pgs_per_line) {
179                     /* all pgs are still valid, move to full line list */
180                     ftl_assert(wpp->curline->ipc == 0);
181                     QTAILQ_INSERT_TAIL(&lm->full_line_list, wpp->curline, entry);
182                     lm->full_line_cnt++;
183                 } else {
184                     ftl_assert(wpp->curline->vpc >= 0 && wpp->curline->vpc < spp->pgs_per_line);
185                     /* there must be some invalid pages in this line */
186                     ftl_assert(wpp->curline->ipc > 0);
187                     pqueue_insert(q: lm->victim_line_pq, d: wpp->curline);
188                     lm->victim_line_cnt++;
189                 }
190                 /* current line is used up, pick another empty line */
191                 check_addr(a: wpp->blk, max: spp->blks_per_pl);
192                 wpp->curline = NULL;
193                 wpp->curline = get_next_free_line(ssd);
194                 if (!wpp->curline) {
195                     /* TODO */
196                     abort();
197                 }
198                 wpp->blk = wpp->curline->id;
199                 check_addr(a: wpp->blk, max: spp->blks_per_pl);
200                 /* make sure we are starting from page 0 in the super block */
201                 ftl_assert(wpp->pg == 0);
202                 ftl_assert(wpp->lun == 0);
203                 ftl_assert(wpp->ch == 0);
204                 /* TODO: assume # of pl_per_lun is 1, fix later */
205                 ftl_assert(wpp->pl == 0);
206             }
207         }
208     }
209 }
210
```

7.1-write

Channel check

Reset to 0 when it needs to **change** channel

LUN check

Reset to 0 when it needs to **change** LUN

Page check

Reset to 0 when it needs to **change** page

If all pages are valid

Increasing full_line_cnt.

Line

Put in the victim line priority queue

The process of selecting the next free line

Settings for managing the writing process

The number of invalid pages **must exceed 0** to be placed in the victim line priority queue.

The descriptions on this page are all descriptions of read.

ftl.c

ssd_read handles read requests.

```
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;

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

    /* 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;
        }

        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;
}
```

7-read

LBA to LPN

Perform reading for each page

For each LPN, PPA(Physical Page Address) is fetched.

(Each PPA read latency)

8-read

When calculating latency, a greater value is maxlat.

Returns maximum latency

In **ssd_read**, you can see that **there is no part about GC**!

ftl.c

As mentioned earlier,
ssd_advance_status calculates latency.

```
static uint64_t ssd_advance_status(struct ssd *ssd, struct ppa *ppa, struct
        nand_cmd *ncmd)
{
    int c = ncmd->cmd;
    uint64_t cmd_stime = (ncmd->stime == 0) ? \
        qemu_clock_get_ns(QEMU_CLOCK_REALTIME) : ncmd->stime;
    uint64_t nand_stime;
    struct ssdparams *spp = &ssd->sp;
    struct nand_lun *lun = get_lun(ssd, ppa);
    uint64_t lat = 0;

    switch (c) {
    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;
#if 0
        lun->next_lun_avail_time = nand_stime + spp->pg_rd_lat;

        /* read: then data transfer through channel */
        chnl_stime = (ch->next_ch_avail_time < lun->next_lun_avail_time) ? \
            lun->next_lun_avail_time : ch->next_ch_avail_time;
        ch->next_ch_avail_time = chnl_stime + spp->ch_xfer_lat;

        lat = ch->next_ch_avail_time - cmd_stime;
#endif
        break;

    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;
#if 0
        chnl_stime = (ch->next_ch_avail_time < cmd_stime) ? cmd_stime : \
                     ch->next_ch_avail_time;
        ch->next_ch_avail_time = chnl_stime + spp->ch_xfer_lat;

        /* write: then do NAND program */
        nand_stime = (lun->next_lun_avail_time < ch->next_ch_avail_time) ? \
            ch->next_ch_avail_time : lun->next_lun_avail_time;
        lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat;

        lat = lun->next_lun_avail_time - cmd_stime;
#endif
        break;

    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;

    default:
        ftl_err("Unsupported NAND command: 0x%x\n", c);
    }

    return lat;
}
```

8

8-read : Latency calculation

This part calculates the latency for read.

The process of calculating when the LUN is available next

Latency calculation

8-write : Latency calculation

This part calculates the latency for write.

The process of calculating when the LUN is available next

Latency calculation

The parts marked in black are ignored.

ftl.c

***ftl_thread** is responsible for verifying the command type and then performing the task when it receives I/O command.

```c
static void *ftl_thread(void *arg)
{
    FemuCtrl *n = (FemuCtrl *)arg;
    struct ssd *ssd = n->ssd;
    NvmeRequest *req = NULL;
    uint64_t lat = 0;
    int rc;
    int i;

    while (!*(ssd->dataplane_started_ptr)) {
        usleep(100000);
    }

    /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    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");
                ;
            }

            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;
}
```

6

Now the process from 1 to 8 has been done.

Pull out request

7-write

Performing a writing task

7-read

Performing a reading task

to poller

9

Now go back to ***nvme_poller**.

nvme-io.c

***nvme_poller** can be seen as consisting of nvme_process_sq_io and nvme_process_cq_cpl.

```
void *nvme_poller(void *arg) 1
{
    FemuCtrl *n = ((NvmePollerThreadArgument *)arg)->n;
    int index = ((NvmePollerThreadArgument *)arg)->index;
    int i;

    switch (n->multipoller_enabled) {
    case 1:
        while (1) {
            if ((!n->dataplane_started)) {
                usleep(1000);
                continue;
            }

            NvmeSQueue *sq = n->sq[index];
            NvmeCQueue *cq = n->cq[index];
            if (sq && sq->is_active && cq && cq->is_active) {
                nvme_process_sq_io(sq, index);
            }
            nvme_process_cq_cpl(n, index);
        }
        break;
    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); 2
                }
            }
            nvme_process_cq_cpl(n, index);
        }
        break;
    }

    return NULL;
}
```

Multipoller is disabled.

Data plane check

Check for all I/O queues

Make sure both queues are active

10

nvme-io.c

nvme_process_cq_cpl takes the request from to_poller and forwards it to the completion queue **after I/O processing time.**

```
static void nvme_process_cq_cpl(void *arg, int index_poller)
{
    FemuCtrl *n = (FemuCtrl *)arg;
    NvmeCQueue *cq = NULL;
    NvmeRequest *req = NULL;
    struct rte_ring *rp = n->to_ftl[index_poller];
    pqueue_t *pq = n->pq[index_poller];
    uint64_t now;
    int processed = 0;
    int rc;
    int i;

    if (BBSSD(n) || ZNSSD(n)) {
        rp = n->to_poller[index_poller];
    }

    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);
    }

    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++;
        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;

    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;
    }
}
```

10

Pull out the request, and put it in the priority queue.

Pull out request

Error handling

Priority queue

(The order is the fastest I/O processing time)

11

Adding completed request to the completion queue

Multipoller is disabled.

The role of notifying that the request has been processed

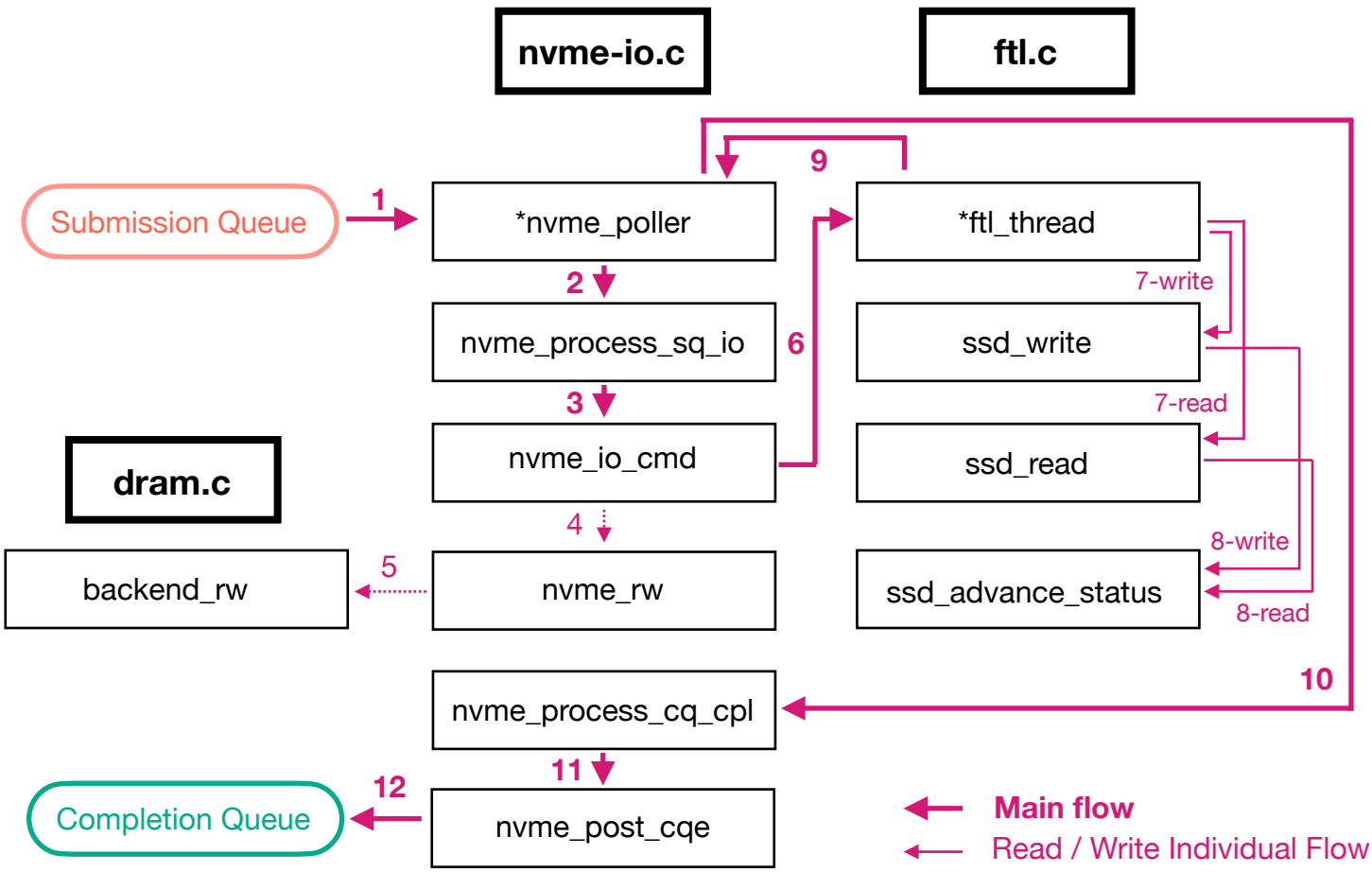
nvme-io.c

nvme_post_cqe adds the completed request to the completion queue.

```
static void nvme_post_cqe(NvmeCQueue *cq, NvmeRequest *req)                    11
{
    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);  12 Completion queue update
}
```

I explained the whole process of read, write!

Structure for (i) read, (ii) write



Some processes have been omitted for readability.

Read and write main part analysis

In ***nvme_poller**, by default, multipoller is disabled,
so goes to the while loop of **default:**.
Here, check the data plane, all I/O queues verify that
the submission queue and completion queue are active.

Then, it goes to the **nvme_process_sq_io** function.
Check the I/O request in submission queue, runs in **nvme_io_cmd**.
Actual data is stored via **nvme_rw** and **backend_rw**.
Also, **nvme_process_sq_io** passes the I/O request to the FTL.

In the while loop of ***ftl_thread**,
handle write and read tasks according to each case.

(Write)
Do the LBA to LPN in the write process,
and go through the process of checking if it needs to do GC.
Do LPN to PPA, and check if the LPN is already valid.
If it is valid, make invalid mark on the existing PPA.
After that, create a new page, updating the mapping table
and the reverse mapping table, make valid mark on the page.
Then, update the write pointer, then calculate the delay and reflect it.

(Read)
In the read process, it also converts LBA to LPN,
It then goes through the mapping table check process.
Finally, calculate and reflect latency.

and then, reflects the latency time, back to ***nvme_poller**.
In **nvme_process_cq_cpl**, put the request
in the completion queue after I/O processing time.
And **nvme_isr_notify_io** notifies the completion of the request.

Some processes have been omitted for readability.

(A) Explanation of code analysis for **(iii) GC mechanisms**

ftl.c

Earlier, in the read and write description, ***ftl_thread** explained that it is responsible for checking the type of command and then performing the task when receiving an I/O command.

Now let's focus on the GC mechanism!

```
static void *ftl_thread(void *arg)                                  1
{
    FemuCtrl *n = (FemuCtrl *)arg;
    struct ssd *ssd = n->ssd;
    NvmeRequest *req = NULL;
    uint64_t lat = 0;
    int rc;
    int i;

    while (!*(ssd->dataplane_started_ptr)) {
        usleep(100000);
    }

    /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    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);        Pull out request
            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;                                                      Performing a writing task
            case NVME_CMD_READ:
                lat = ssd_read(ssd, req);
                break;                                                      Performing a reading task
            case NVME_CMD_DSM:
                lat = 0;
                break;
            default:
                //ftl_err("FTL received unkown request type, ERROR\n");
                ;
            }

            req->reqlat = lat;
            req->expire_time += lat;                                        Reflection of latency time

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

            /* clean one line if needed (in the background) */
            if (should_gc(ssd))                                             2
                do_gc(ssd, false);                                          3
            }                                                               Condition judgment
        }                                                                   Running the Garbage Collection
    }

    return NULL;
}
```

***ftl_thread checks the should_gc at the end of all I/O operations.**

The conditions for executing the GC, and the actual execution.

ftl.c

should_gc checks to see if it exceeds the threshold.

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

2

free_line_cnt <= if gc_thres_lines, call do_gc

```
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;
    }

    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);

    /* 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);

            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;
        }
    }

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

    return 0;
}
```

do_gc actually performs GC operations.
Number 4 is the process of choosing the victim line.

4

Victim line selection process

Repeat for each channel

(Logical Unit Number)

The process progresses for the LUNs in each channel.

5

It's the process of going to clean the blocks.

A description of the next process can be found **on the upcoming page.**

ftl.c

```
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;
}
```

4

This is the process of selecting a victim line using a priority queue.

Choosing the highest priority

Pop the selected victim line in the priority queue.

```
/* 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);
}
```

5

6

7

The process of reading pages, and moving valid pages.

When a GC occurs, an important task is performed to read the page, and move the valid page. This process is 6 and 7.

ftl.c

As we have seen in other parts earlier, latency is calculated in **ssd_advance_status.**

gc_write_page function performs the process for moving valid pages.

```
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);
    }
}
```

6

enable_gc_delay is enabled by default, so the if statement is executed.

```
/* 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;
}
```

7

The process of verifying that LPN is valid

New PPA

The process of updating LPN and new PPA to the mapping table

Update reverse mapping table

Valid marking on the newly created page

A function previously used in ssd_write, also in gc_write_page.

enable_gc_delay is enabled by default, so the if statement is executed.

latency calculation

The parts marked in black are ignored.

gc_endtime reflects the next available time for the LUN.

Now, I've explained all the processes from 1 to 7.

Number 8, which makes the block free, and then let's look at the other processes.

ftl.c

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

2

if free_line_cnt <= gc_thres_lines, call do_gc

```
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;
    }

    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);

    /* 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);

            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;
        }
    }

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

    return 0;
}
```

3

4

Victim line selection process

Repeat for each channel

5

8

enable_gc_delay is enabled by default, so the if statement is executed.

9

gc_endtime reflects the next available time for the LUN.

10

There is also a process of setting the **line to free**!

The unit of GC is **line**.

ftl.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++;
}
```

8

For every page in a block

Reset number of valid/invalid pages **in block** to 0

The important part here is that it **resets** for every page **in the block**!

ftl.c

The latency it takes to erase will also be calculated.

9

```c
static uint64_t ssd_advance_status(struct ssd *ssd, struct ppa *ppa, struct
        nand_cmd *ncmd)
{
    int c = ncmd->cmd;
    uint64_t cmd_stime = (ncmd->stime == 0) ? \
        qemu_clock_get_ns(QEMU_CLOCK_REALTIME) : ncmd->stime;
    uint64_t nand_stime;
    struct ssdparams *spp = &ssd->sp;
    struct nand_lun *lun = get_lun(ssd, ppa);
    uint64_t lat = 0;

    switch (c) {
    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;
#if 0
        lun->next_lun_avail_time = nand_stime + spp->pg_rd_lat;

        /* read: then data transfer through channel */
        chnl_stime = (ch->next_ch_avail_time < lun->next_lun_avail_time) ? \
            lun->next_lun_avail_time : ch->next_ch_avail_time;
        ch->next_ch_avail_time = chnl_stime + spp->ch_xfer_lat;

        lat = ch->next_ch_avail_time - cmd_stime;
#endif
        break;

    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;

#if 0
        chnl_stime = (ch->next_ch_avail_time < cmd_stime) ? cmd_stime : \
                     ch->next_ch_avail_time;
        ch->next_ch_avail_time = chnl_stime + spp->ch_xfer_lat;

        /* write: then do NAND program */
        nand_stime = (lun->next_lun_avail_time < ch->next_ch_avail_time) ? \
            ch->next_ch_avail_time : lun->next_lun_avail_time;
        lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat;

        lat = lun->next_lun_avail_time - cmd_stime;
#endif
        break;

    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;

    default:
        ftl_err("Unsupported NAND command: 0x%x\n", c);
    }

    return lat;
}
```

The parts marked in black are ignored.

The process of calculating when the LUN is available next

Latency calculation

Latency return

ftl.c

mark_line_free resets all pages **on the line**.

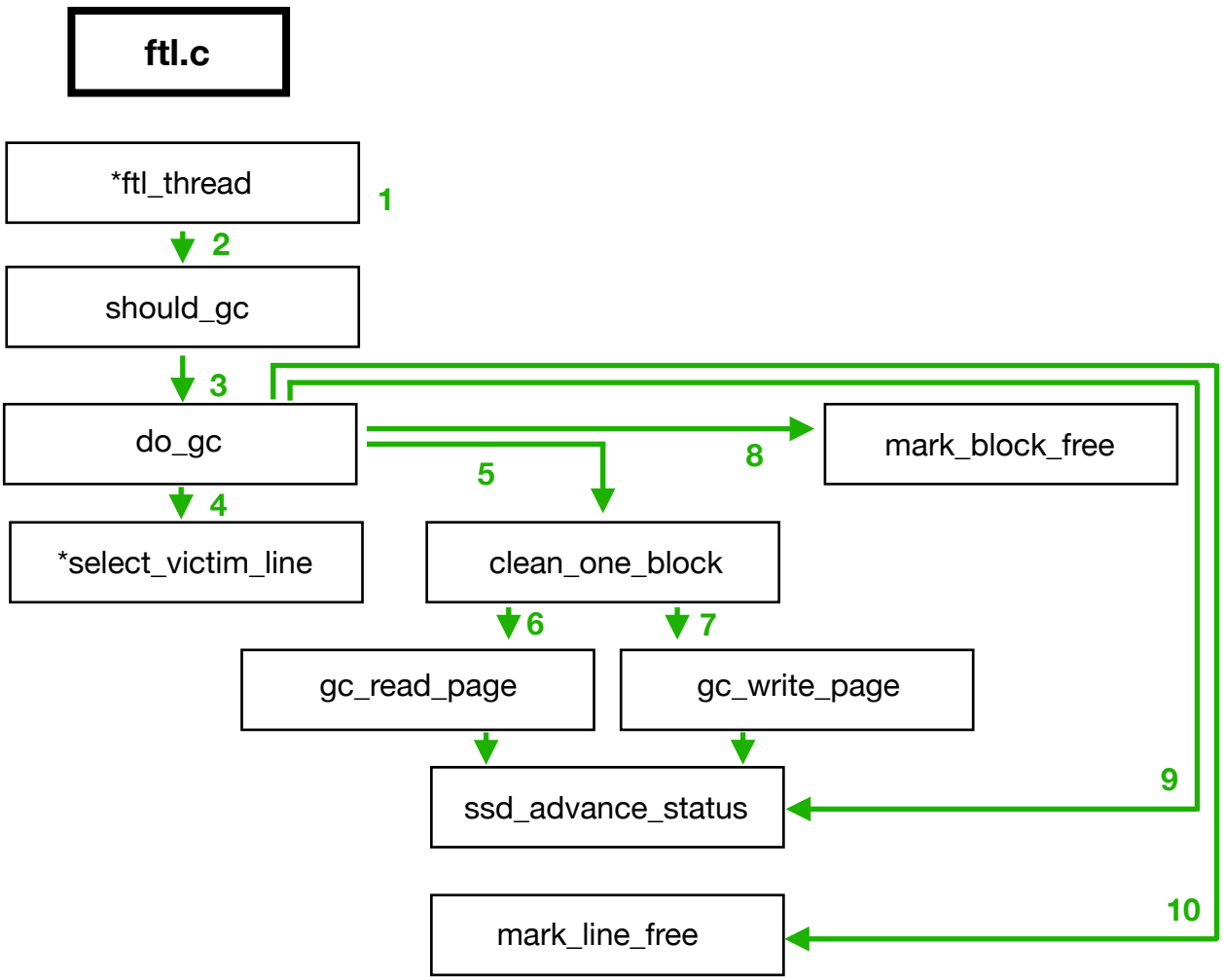
```
static void mark_line_free(struct ssd *ssd, struct ppa *ppa)
{
    struct line_mgmt *lm = &ssd->lm;
    struct line *line = get_line(ssd, ppa);
    line->ipc = 0;
    line->vpc = 0;
    /* move this line to free line list */
    QTAILQ_INSERT_TAIL(&lm->free_line_list, line, entry);
    lm->free_line_cnt++;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

10

Reset number of valid/invalid pages in line to 0

Now I've finished all the explanations for the GC mechanism!

Structure for (iii) GC mechanisms



Some processes have been omitted for readability.

Analysis of key **GC mechanisms**

Earlier, I checked how read and write were handled.
Now let's analyze the main parts of the GC mechanism.

After every I/O request in the while loop of ***ftl_thread**, if it needs GC,
It goes through the process of checking through the conditions.
should_gc works under the condition that
free_line_cnt <= gc_thres_lines, It will perform GC in **do_gc**.

In **do_gc**, the process of selecting the victim line takes place,
use the priority queue to select the highest priority as the victim line.

It then goes through the process of cleaning the block
for each LUN in each channel.
From here, read the page.

And move valid page. Do LPN validation in **gc_write_page**.
Then, create a new page,
and update the mapping table and reverse mapping table.
And, after valid marking the newly created page,
updates the pointer in the **ssd_advance_write_pointer** function.
Then, calculate latency, and **gc_endtime**
reflects the next available time for the LUN.

After that, mark that the block is free,
After going through the latency calculation process,
gc_endtime reflects the next available time for the LUN.

And proceed with the process of making the line free outside the for loop.

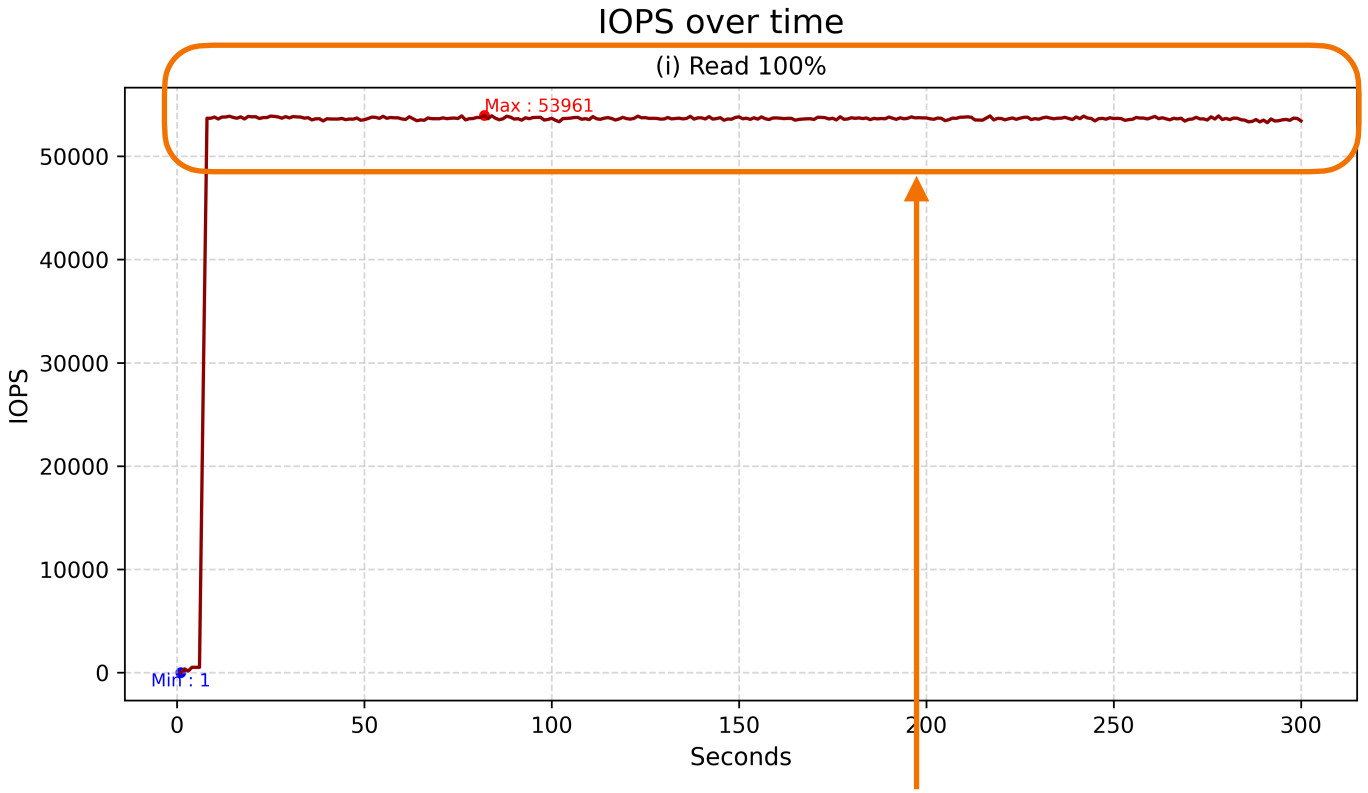
(The unit when GC is performed is line,
the valid page of the victim line is moved to another line.)

Some processes have been omitted for readability.

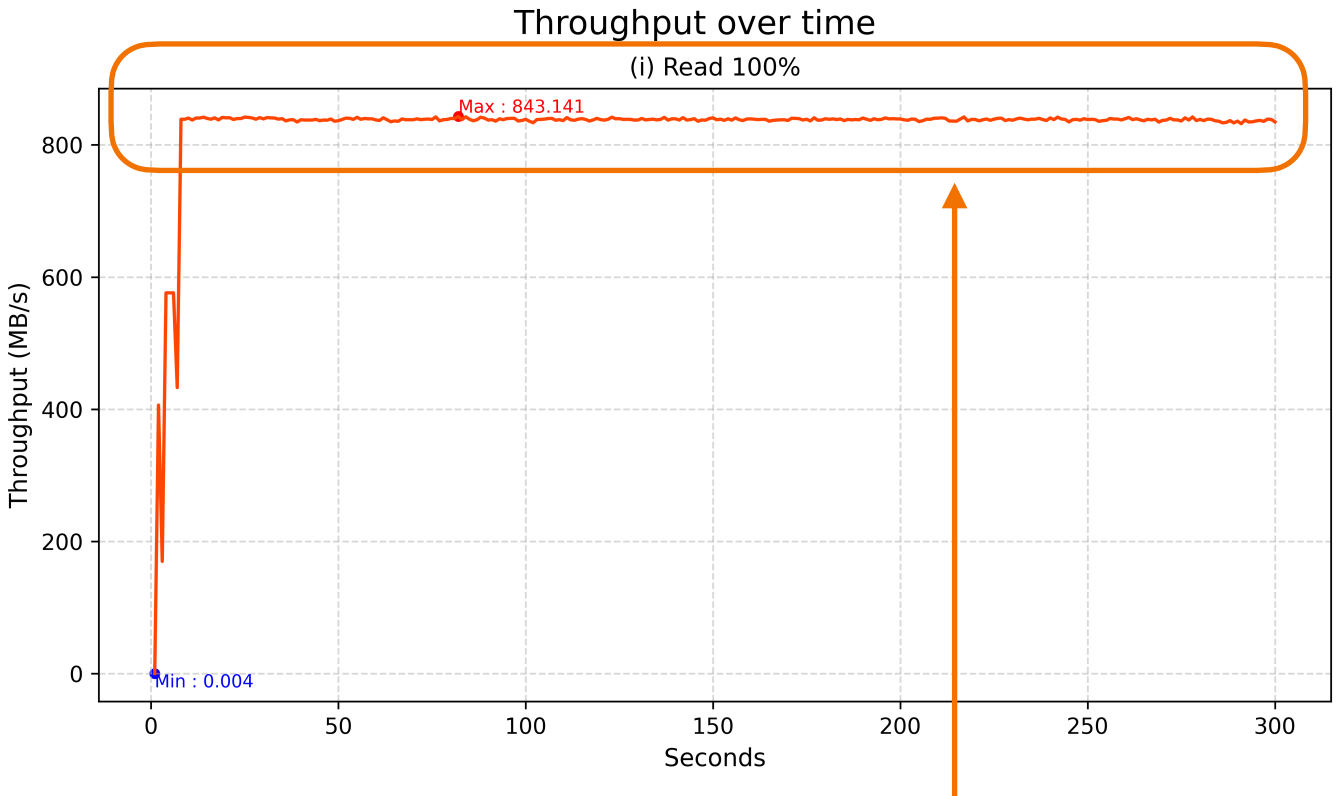
(B) Performance analysis using **benchmarks**

(i) Read 100%

```
fio --directory=/mnt/nvme0n1 --name=fio_test --direct=1\
    --rw=randread --bs=16k --size=576M --numjobs=4 \
    --time_based --runtime=300 --group_reporting --norandommap
```

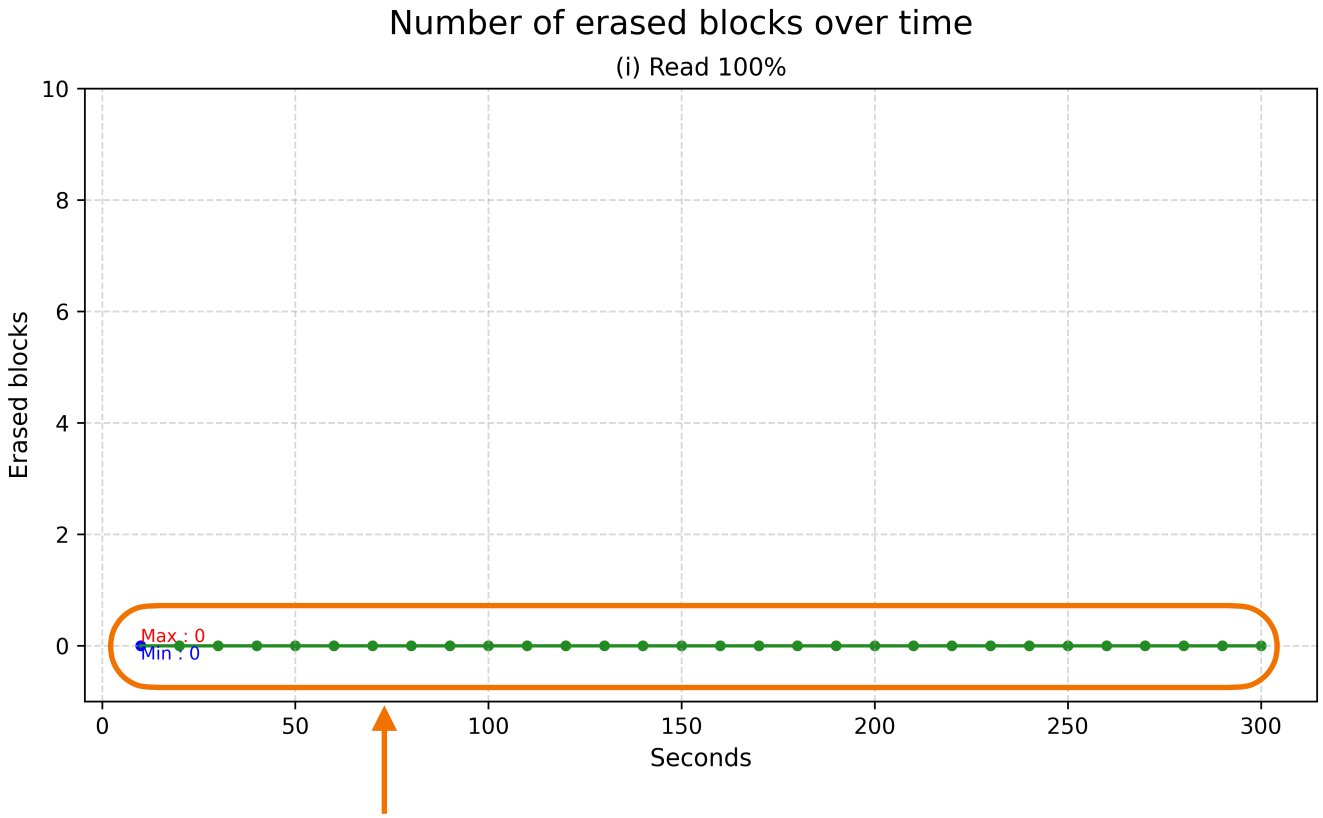


Since it only performs read operations,
it is not possible to observe write cliff.



For the same reason,
it is not possible to observe write cliff.

(i) Read 100%

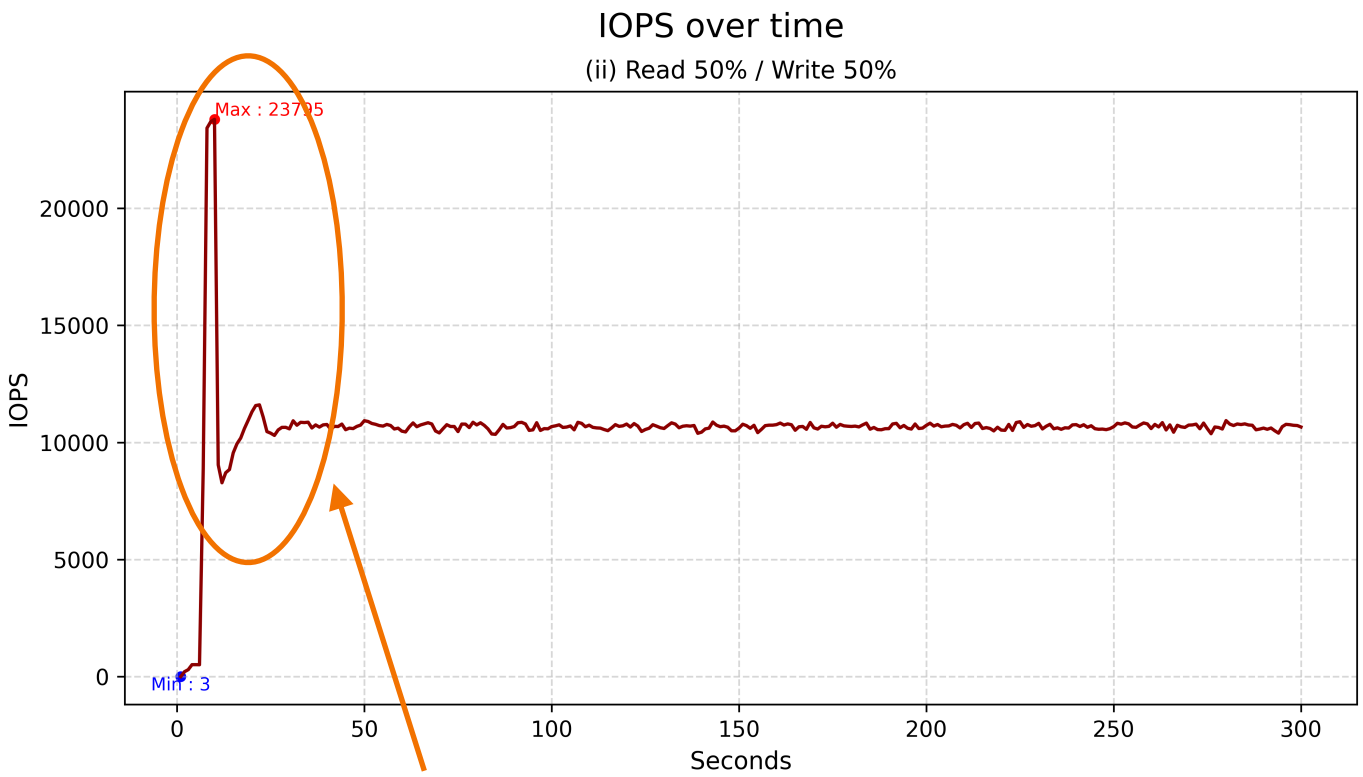


Since GC does not occur, the measured values are all zero.

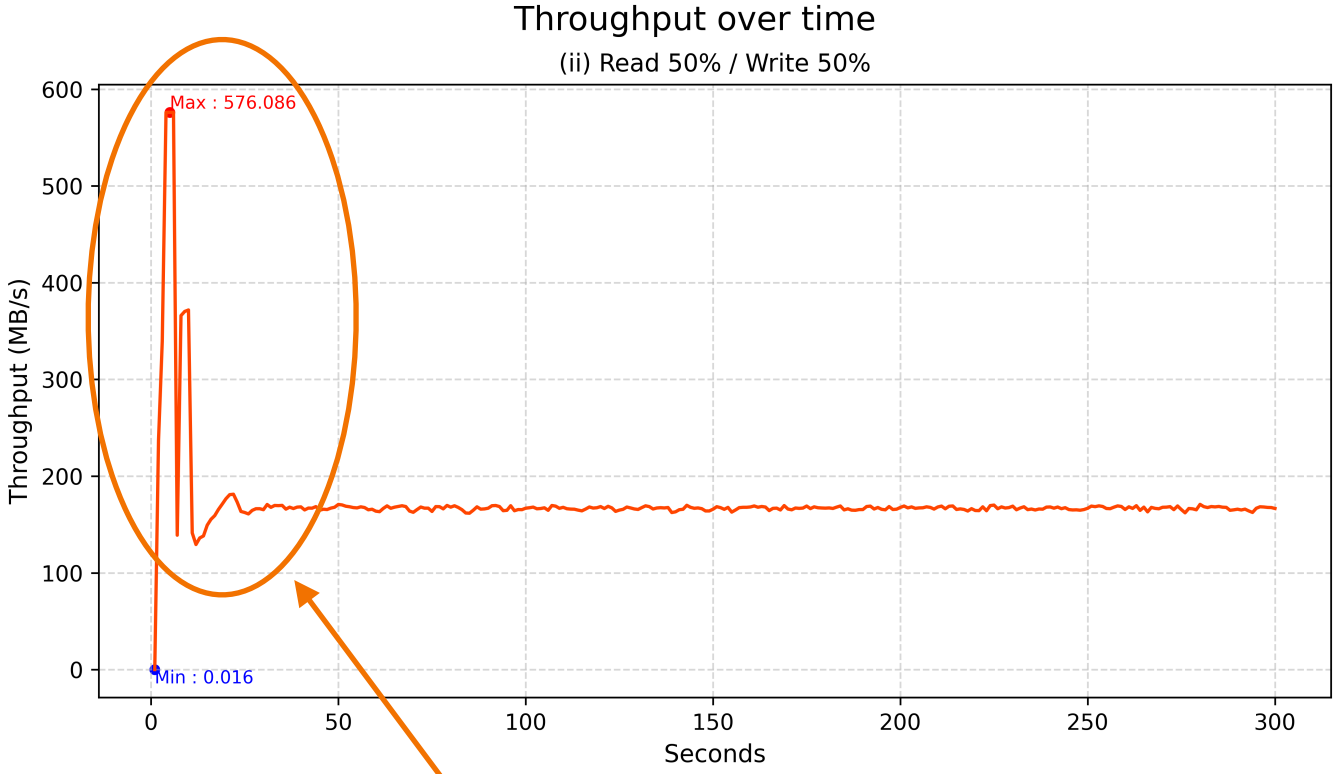
Also, for the same reason, the moved valid pages are zero.

(ii) Read 50% / Write 50%

```
fio --directory=/mnt/nvme0n1 --name=fio_test --direct=1\
    --rw=randrw --bs=16k --size=576M --numjobs=4 \
    --time_based --runtime=300 --group_reporting --norandommap
```

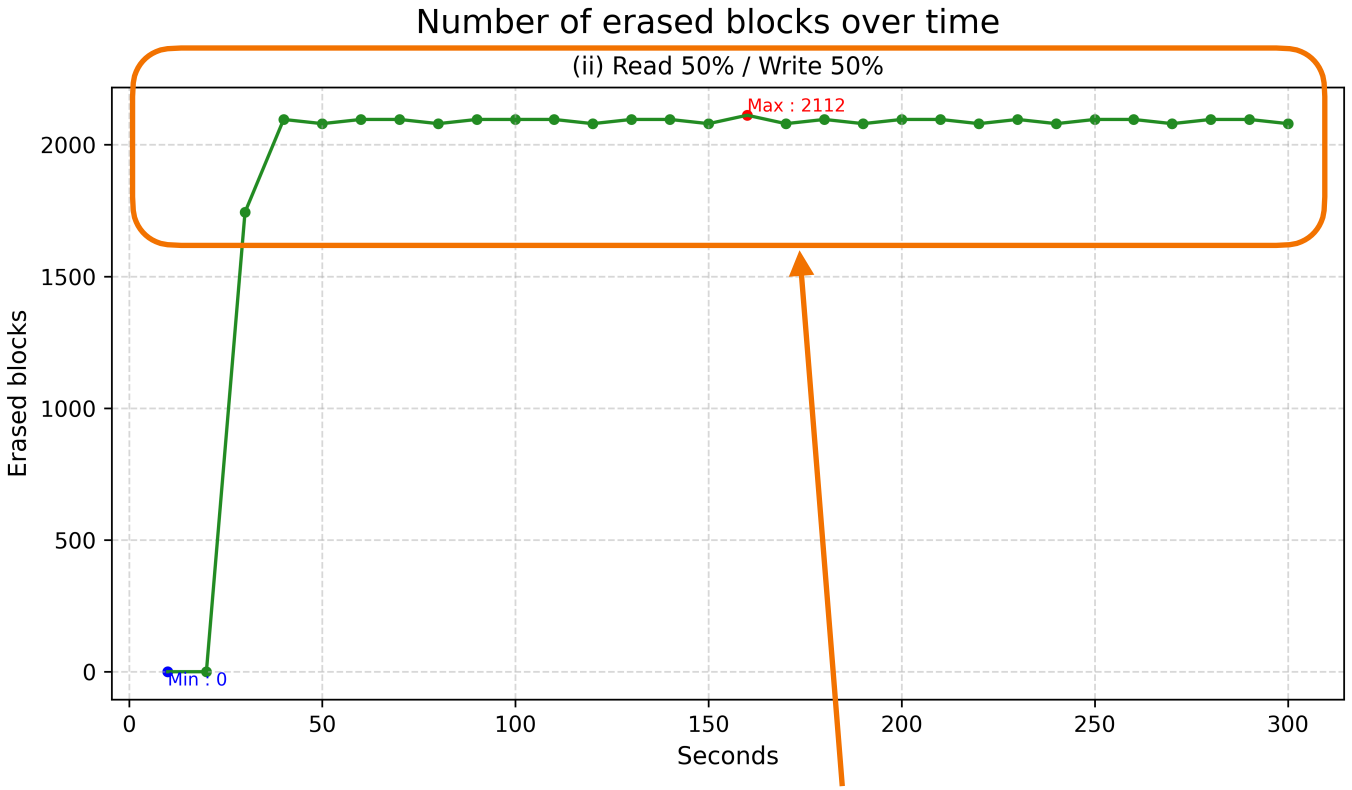


Write cliff occurs, which was not seen in (i) Read 100%.
Because it performs Read 50% / **Write 50%**.



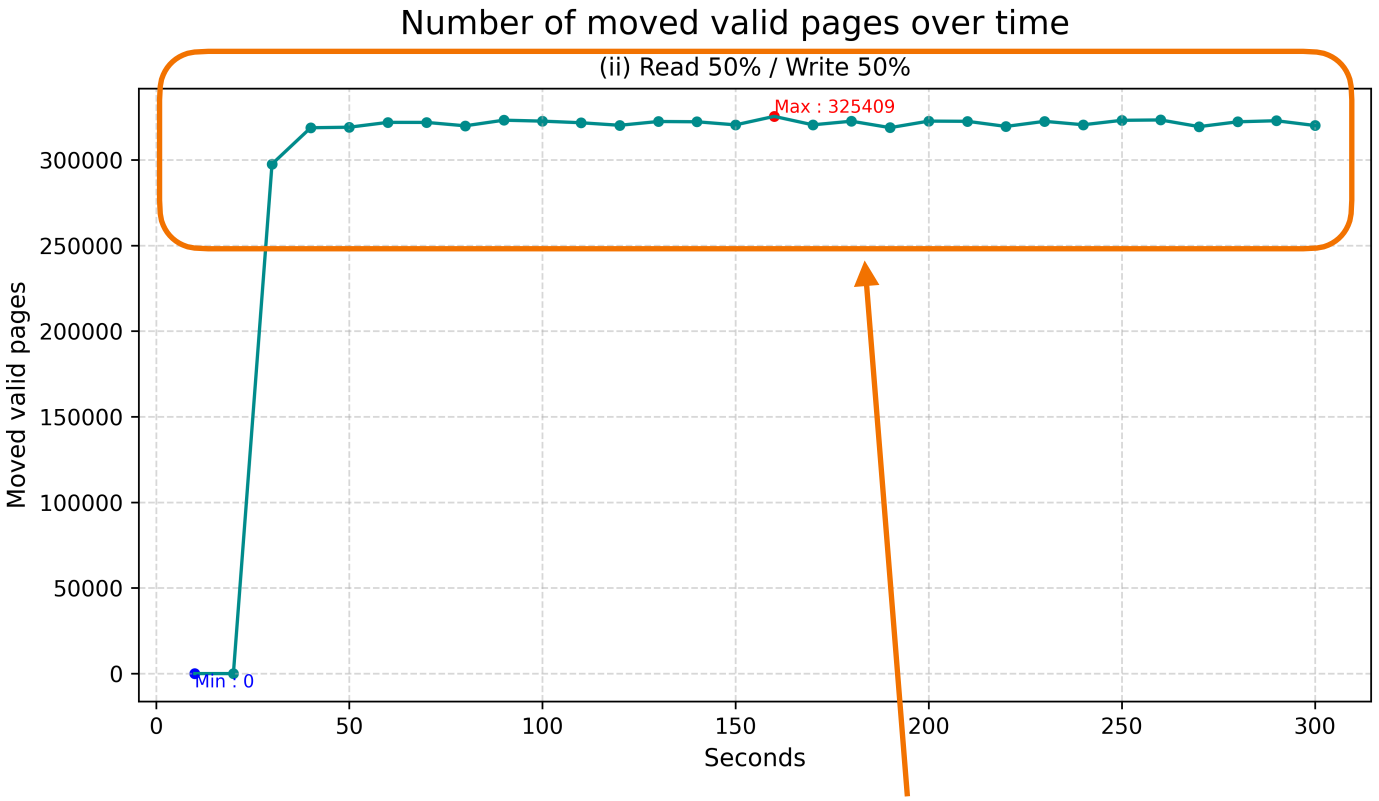
For the same reason,
write cliff is observed.

(ii) Read 50% / Write 50%



Since GC is performed,

the value of erased blocks is found in the graph above.

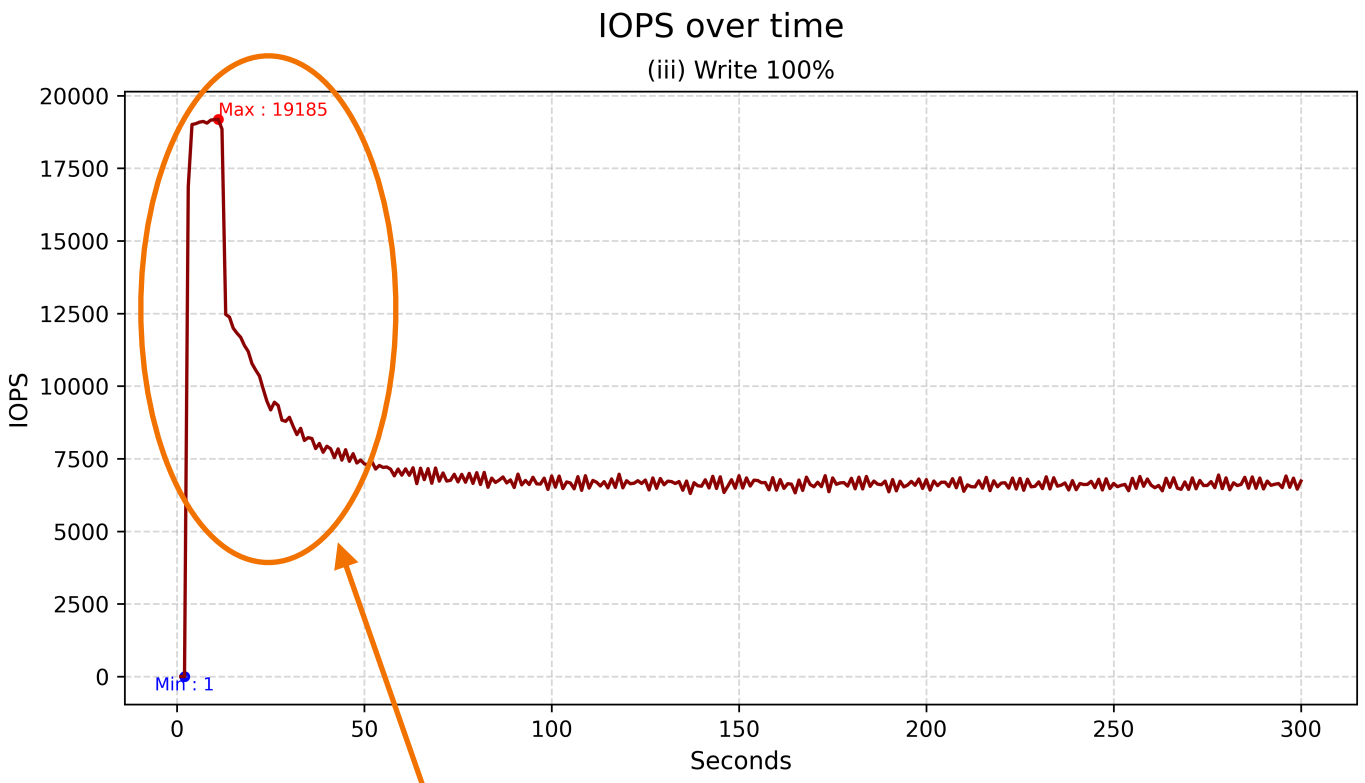


During the GC, valid pages are moved.

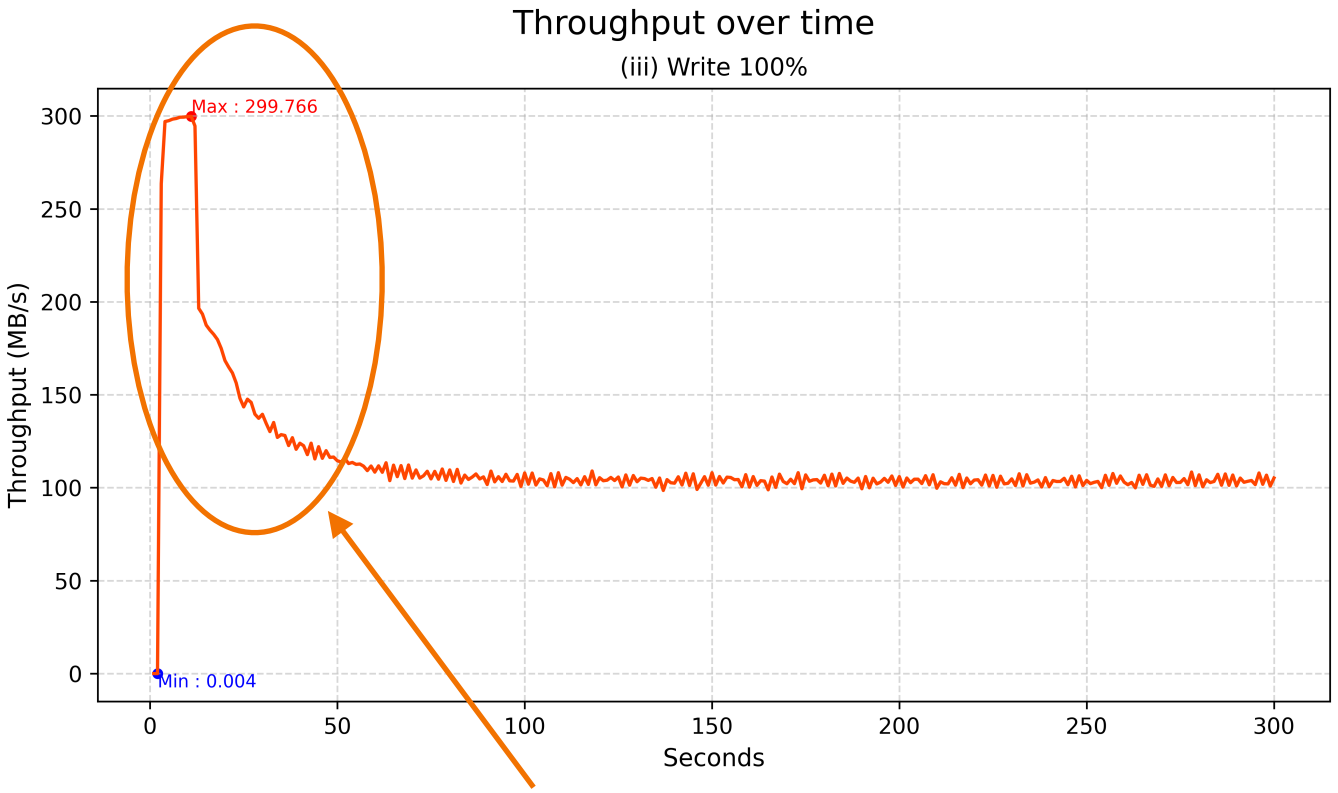
Measurements are confirmed in the graph above.

(iii) Write 100%

```
fio --directory=/mnt/nvme0n1 --name=fio_test --direct=1\
    --rw=randwrite --bs=16k --size=576M --numjobs=4 \
    --time_based --runtime=300 --group_reporting --norandommap
```



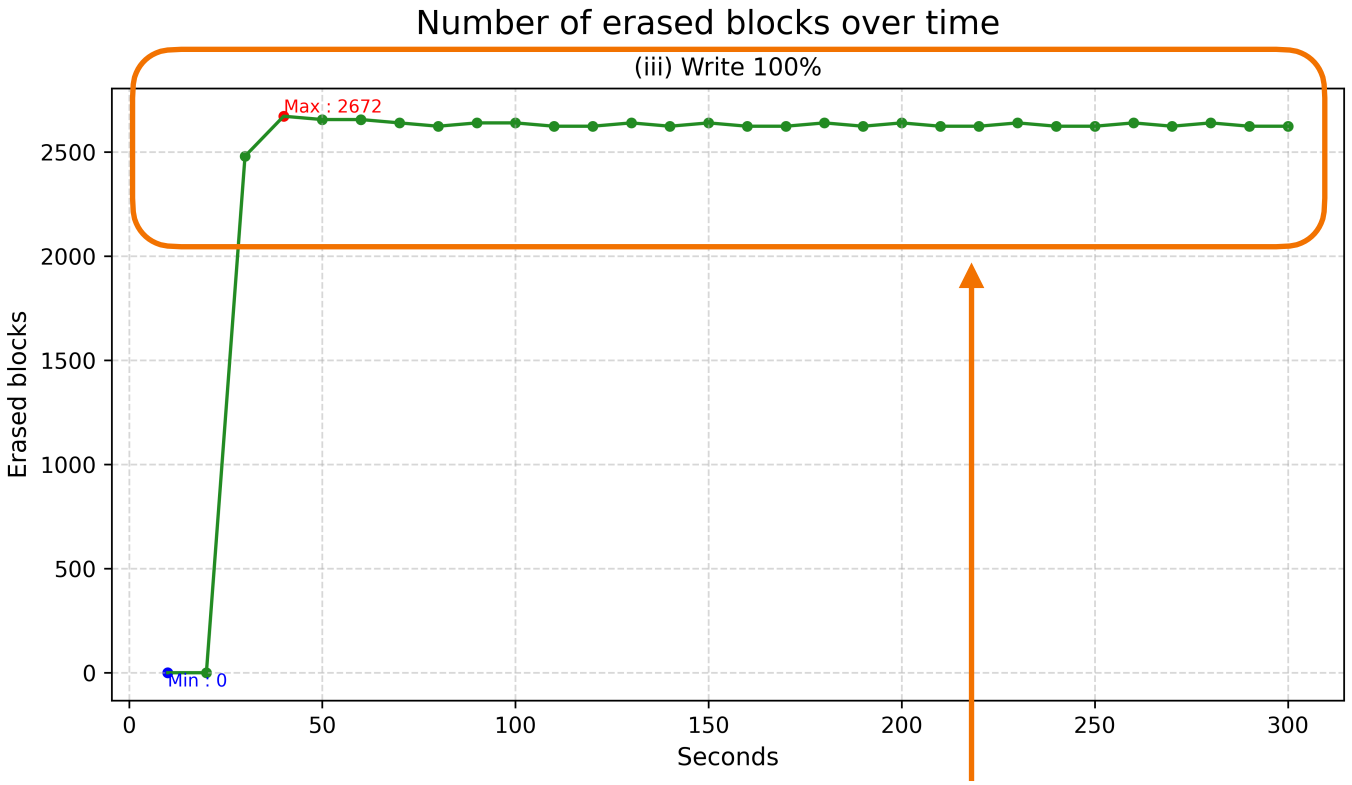
Unlike (i) Read 100%,
write cliff is observed at (iii) Write 100%.



Since it is Write 100%,
write cliff occurs.

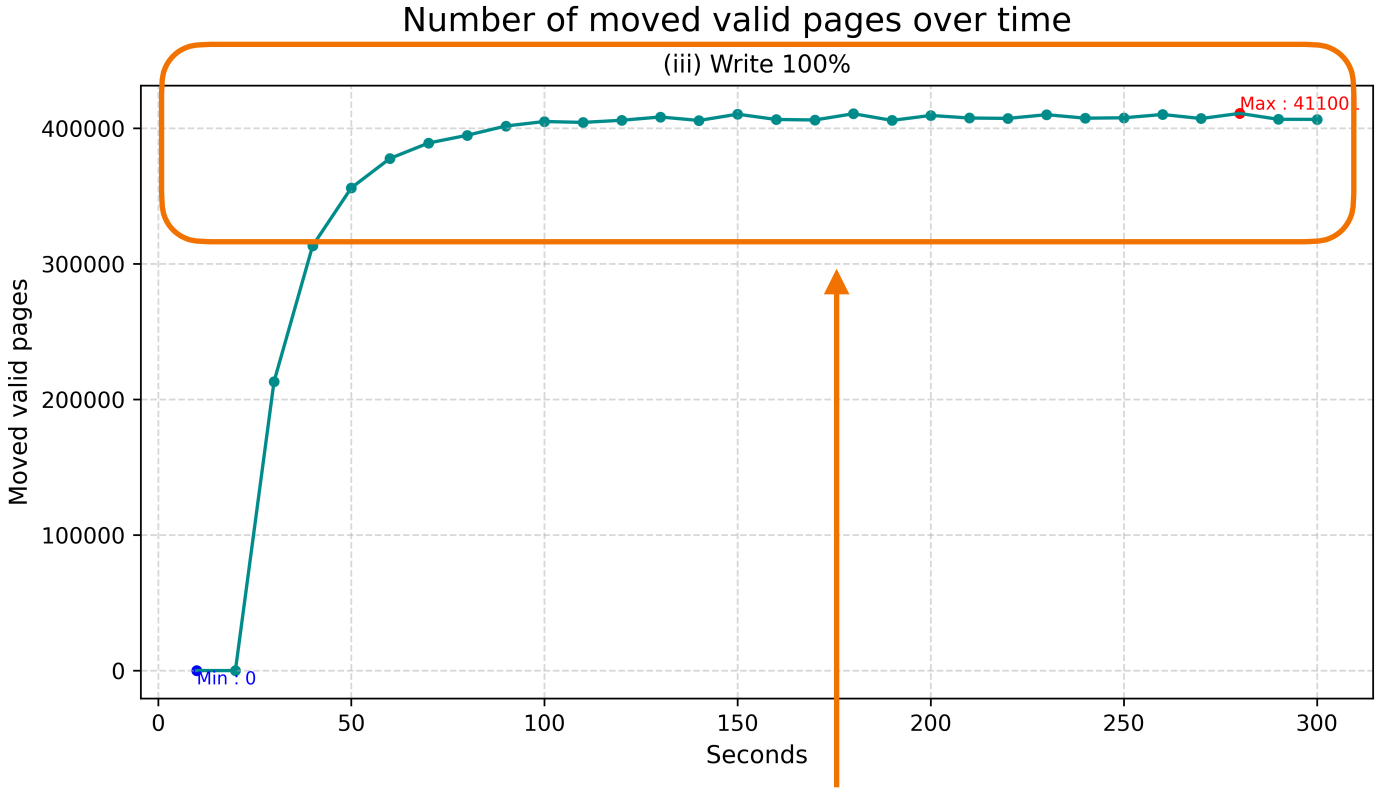
(iii) Write 100%

There is 100% write here, so the process of performing GC and moving valid pages is executed.



Since it is (iii) Write 100%, GC occurs.

So, the value of the erased blocks exists.



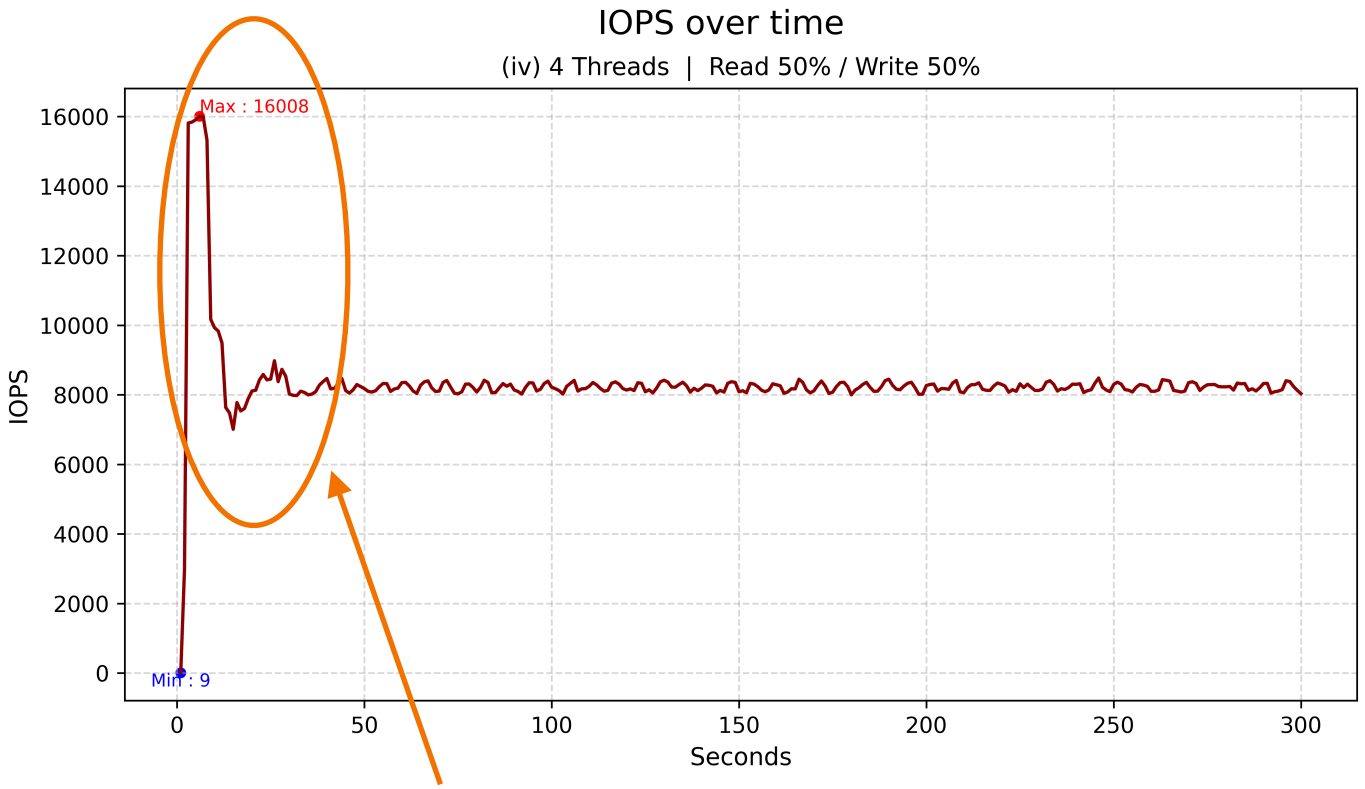
During the GC, valid pages are moved.

The measured values appear in the graph.

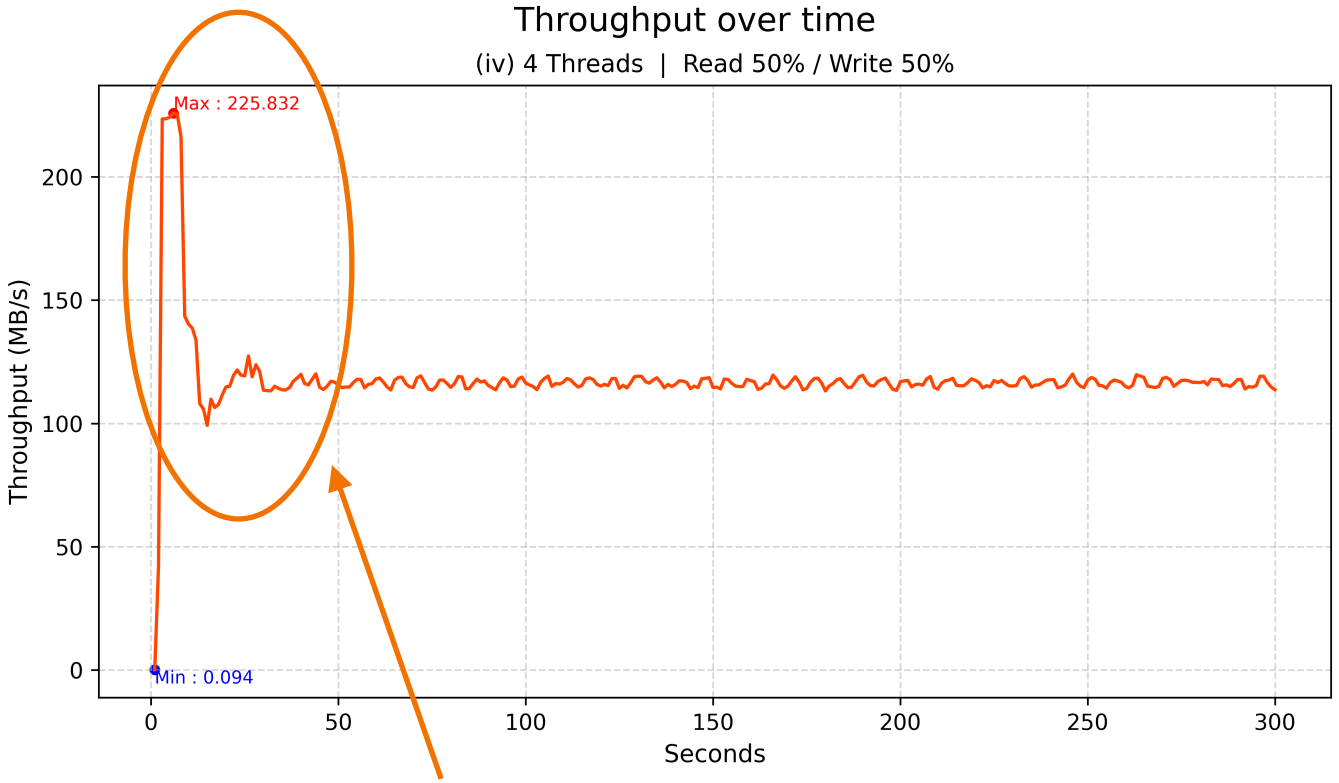
(iv) 4 Threads | Read 50% / Write 50%

sysbench fileio **--threads=4** --file-total-size=2304M **--file-test-mode=rndrw** \
--file-extra-flags=direct --file-fsync-all=on prepare

sysbench fileio **--threads=4** --file-total-size=2304M **--file-test-mode=rndrw** \
--file-extra-flags=direct --file-fsync-all=on --time=300 run

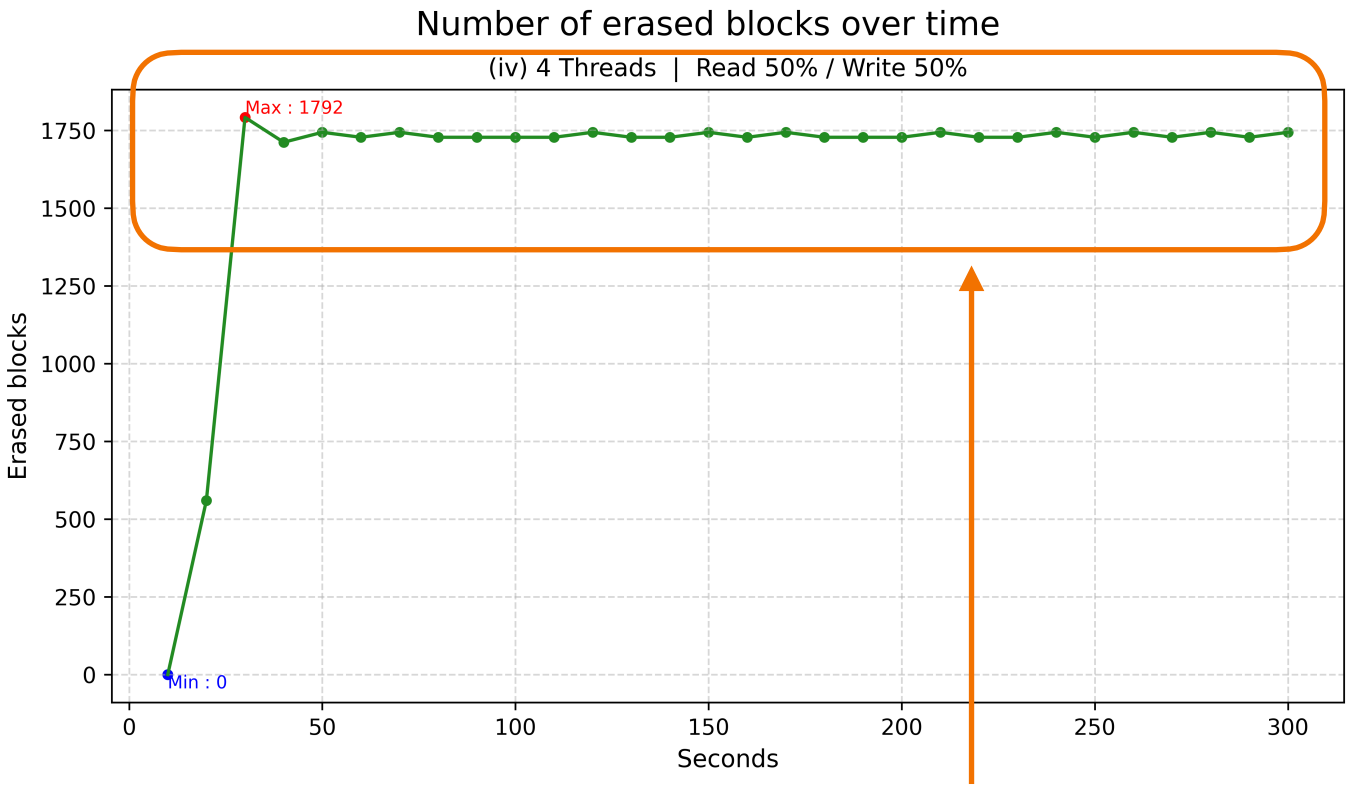


The average value of IOPS is about 8337.
Write cliff is observed in the graph.



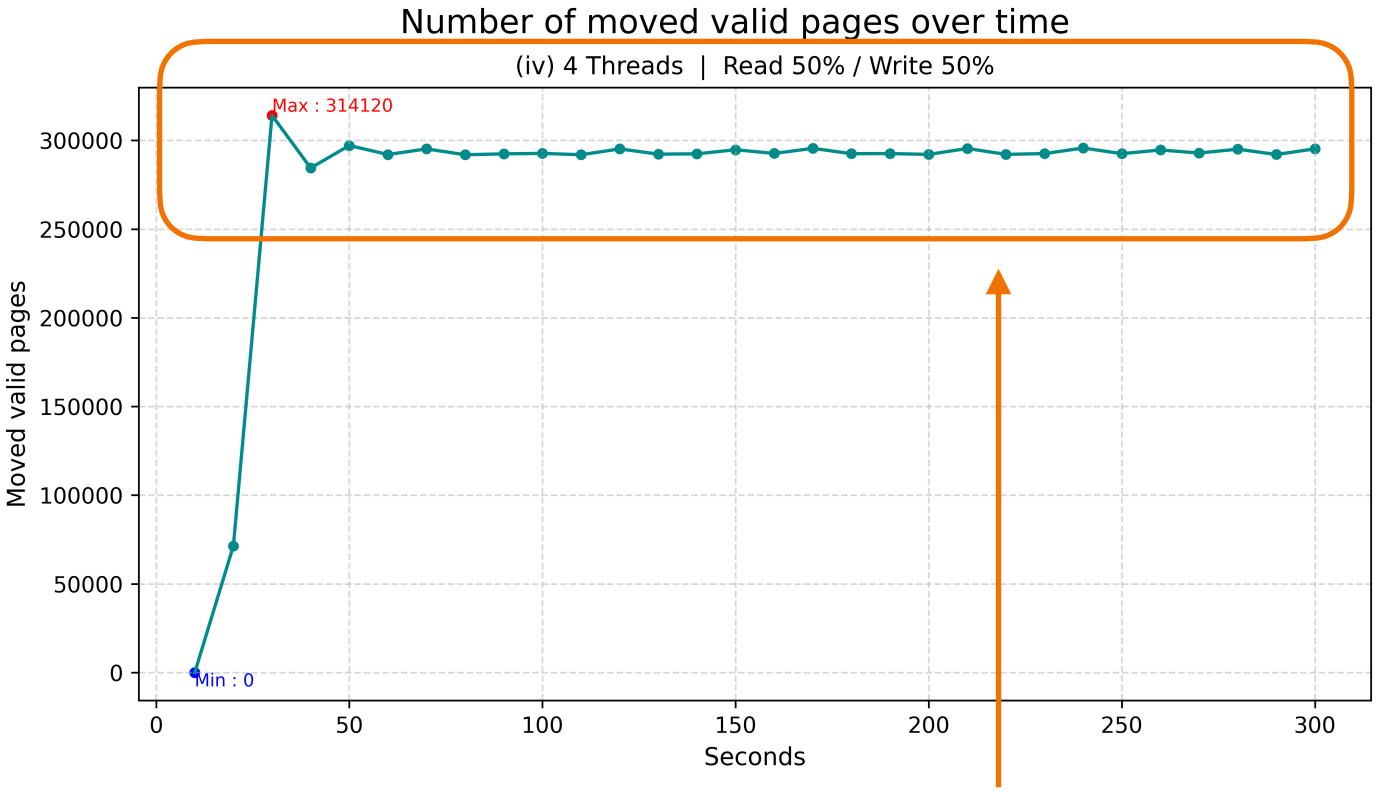
The average value of throughout is about 118 MB/s.
Write cliff is observed in the graph.

(iv) 4 Threads | Read 50% / Write 50%



Since GC is performed, the value of the erased blocks exists.

(Because it's Read 50% / **Write 50%**)



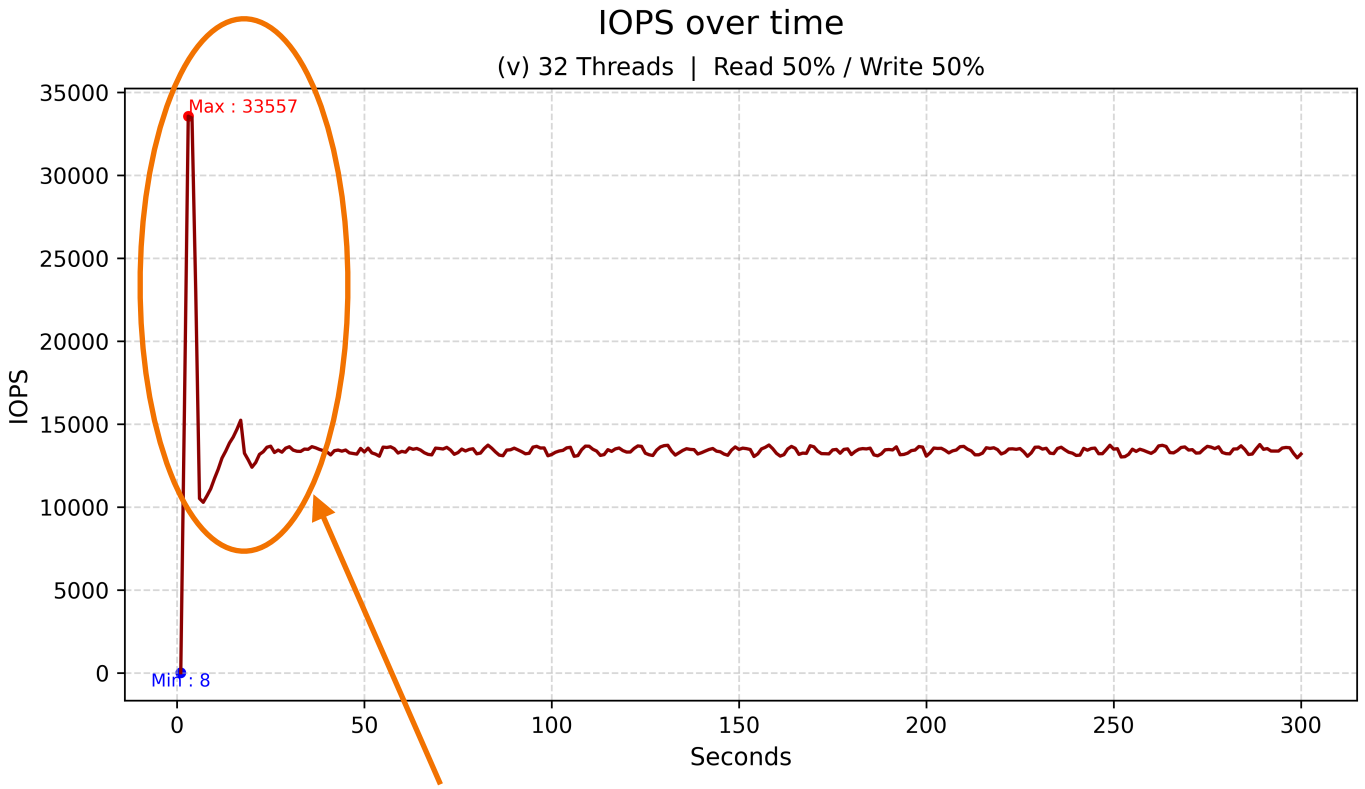
Read 50% / **Write 50%**, so GC occurs.

For this reason, **valid pages are moved.**

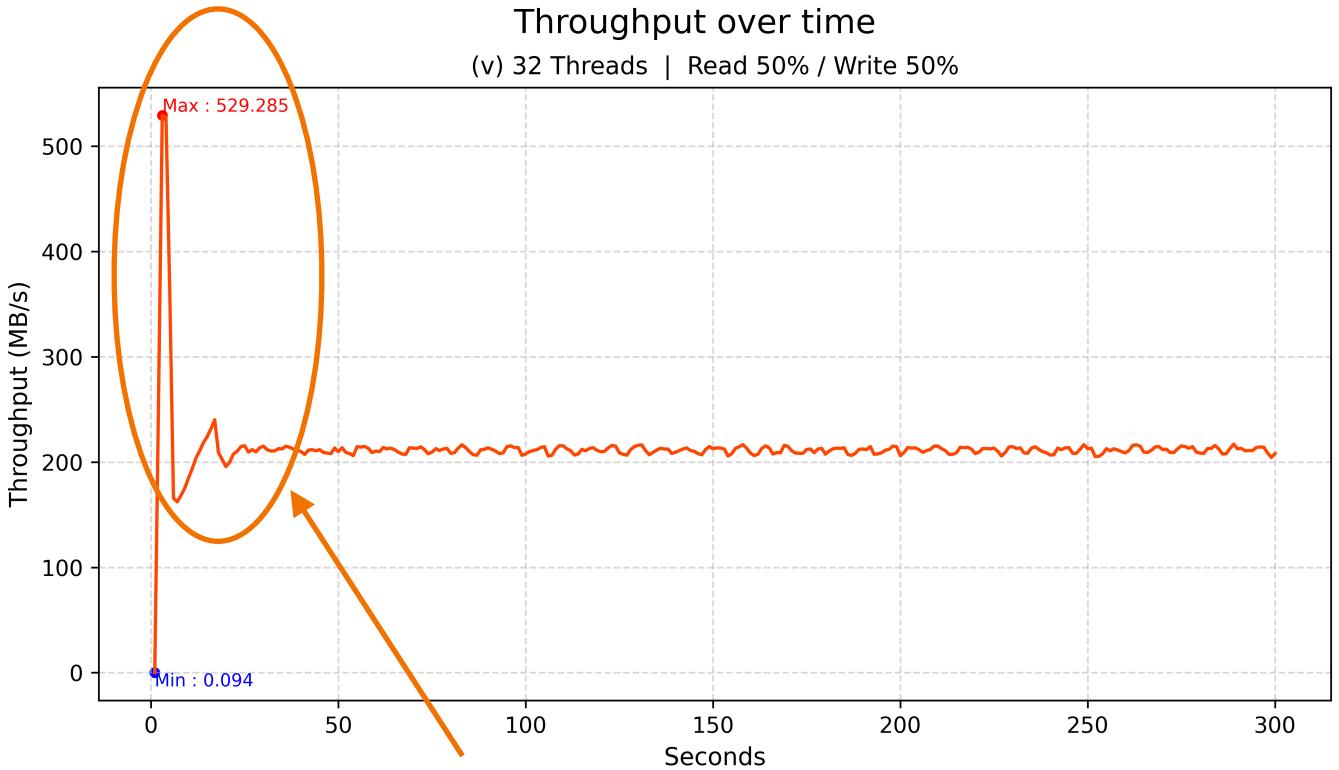
(v) 32 Threads | Read 50% / Write 50%

sysbench fileio **--threads=32** --file-total-size=2304M **--file-test-mode=rndrw** \
--file-extra-flags=direct --file-fsync-all=on prepare

sysbench fileio **--threads=32** --file-total-size=2304M **--file-test-mode=rndrw** \
--file-extra-flags=direct --file-fsync-all=on --time=300 run

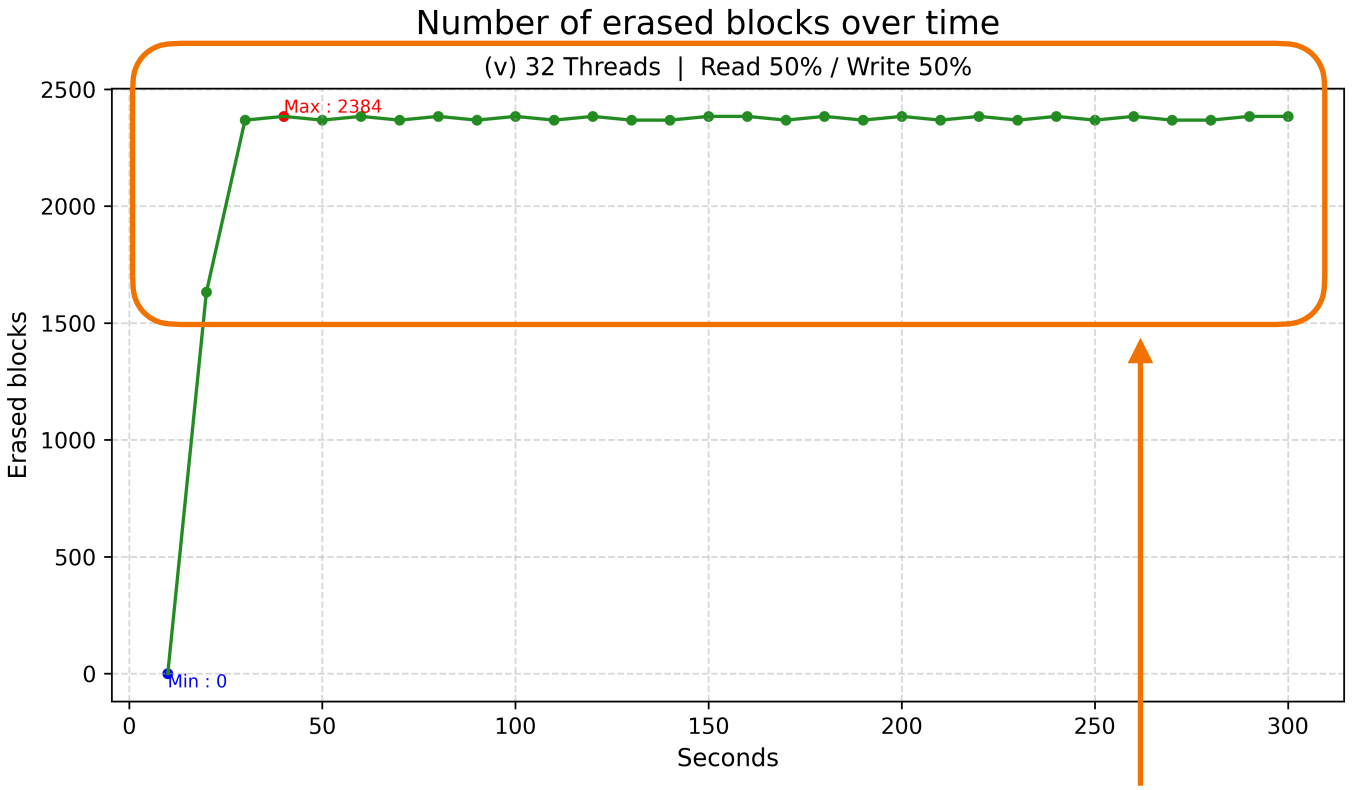


Write cliff is observed in the graph.
The average value of IOPS is about 13490.



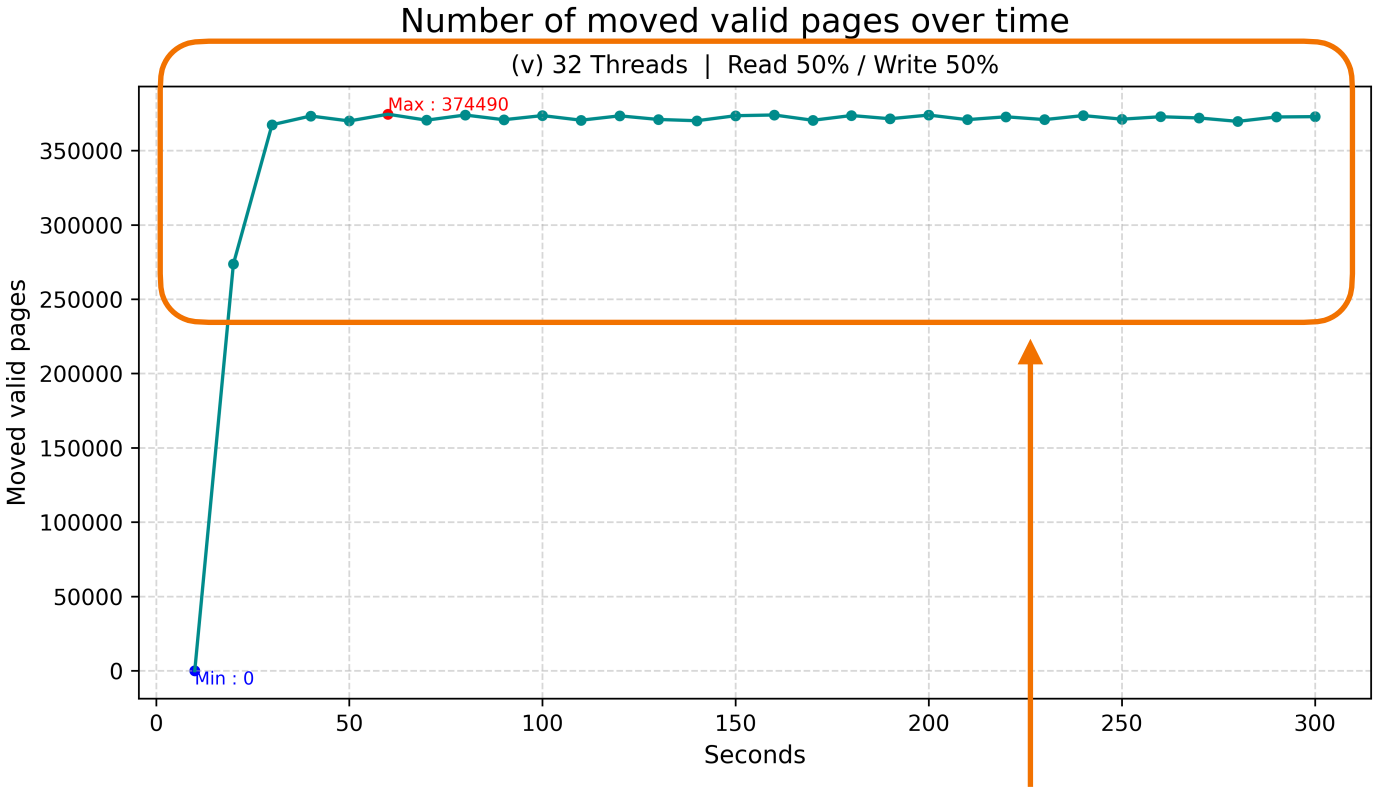
Write cliff is observed in the graph.
The average value of throughout is about 213 MB/s.

(v) 32 Threads | Read 50% / Write 50%



Since GC is performed, the value of the erased blocks exists.

(Because it's Read 50% / **Write 50%**)

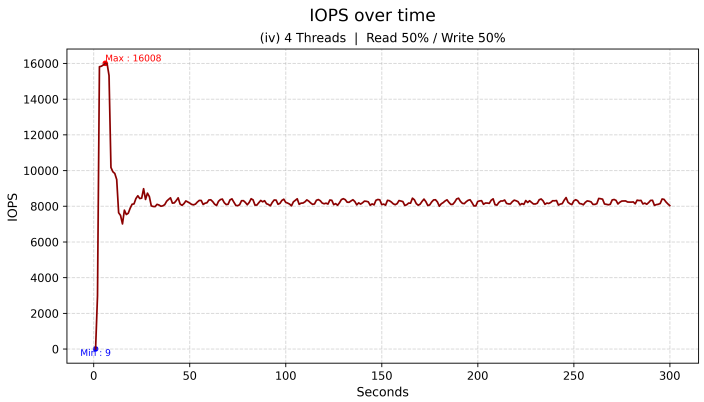


During the GC, valid pages are moved.

Thus, the presence of measured values can be confirmed in the graph.

IOPS and throughput comparison of 4 Threads and 32 Threads

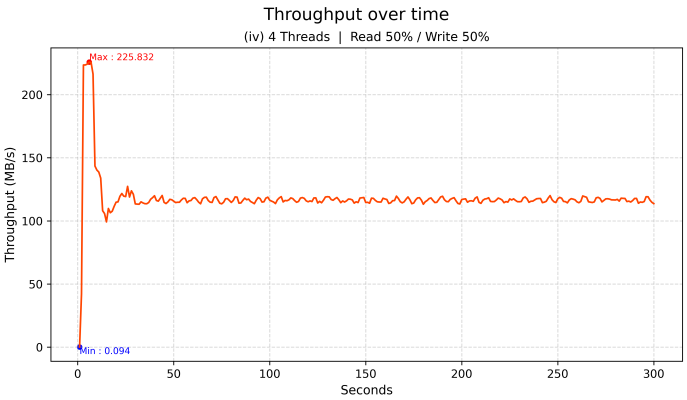
(iv) 4 Threads | Read 50% / Write 50%



When comparing the average IOPS, **32 threads** have higher IOPS.

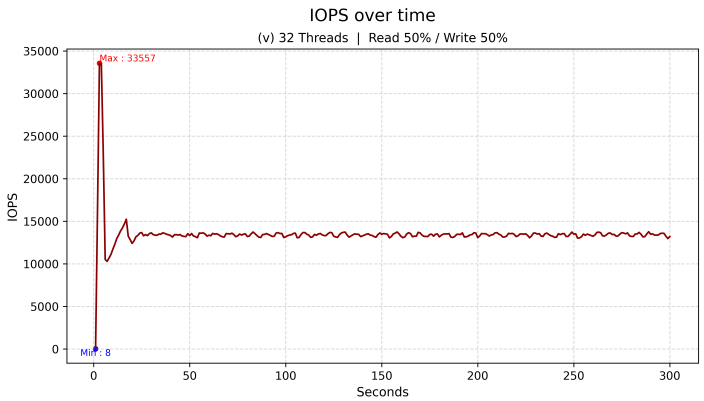
The average value of IOPS is about 8337.

The average throughput likewise has a higher value for **32 threads**.



The average value of throughout is about 118 MB/s.

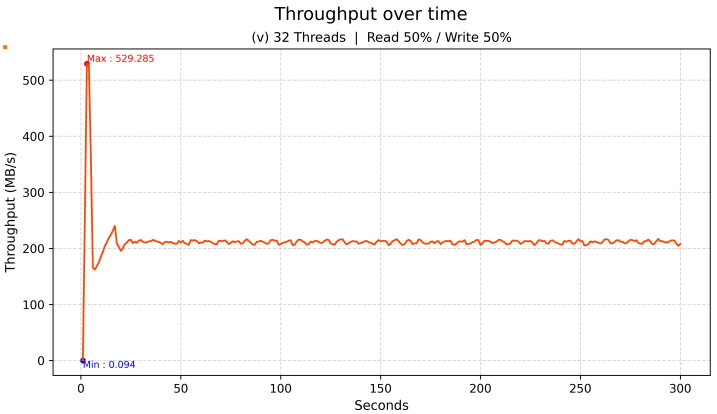
(v) 32 Threads | Read 50% / Write 50%



However, as much as the percentage increased from 4 threads to 32 threads, the average IOPS and throughput did not increase.

The average value of IOPS is about 13490.

Here, I was able to confirm that the performance improvement due to the increase in the number of threads requires that **there must be hardware to support it.**



The average value of throughout is about 213 MB/s.

(C) Explanation for **I/O statistical routines**

ftl.c

```c
#include "ftl.h"

/* ----- IOPS ----- */
static unsigned int inputOp = 0;
static unsigned int outputOp = 0;
static FILE *IOPSData;

/* ----- THROUGHPUT ----- */
static unsigned int totalRead = 0;
static unsigned int totalWrite = 0;
static FILE *throughputData;

/* ----- GC ----- */
static unsigned int reclaimedBlocks = 0;
static FILE *GCData;

/* ----- WAF ----- */
static unsigned int GCWrite = 0;
static FILE *WAFData;
```

Declaration of global variables

IOPS and throughput, GC and WAF related variables have been declared.

In addition, it is designed to go through the process of storing each related data as csv file.

```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++;

    /* ----- GC ----- */
    reclaimedBlocks++;
}
```

Measure the number of reclaimed blocks

This part is used to measure reclaimed blocks in GC. When the mark_block_free function is executed, **reclaimedBlock** is incremented.

ftl.c

```
712   /* here ppa identifies the block we want to clean */
713   static void clean_one_block(struct ssd *ssd, struct ppa *ppa)
714   {
715       struct ssdparams *spp = &ssd->sp;
716       struct nand_page *pg_iter = NULL;
717       int cnt = 0;
718
719       for (int pg = 0; pg < spp->pgs_per_blk; pg++) {
720           ppa->g.pg = pg;
721           pg_iter = get_pg(ssd, ppa);
722           /* there shouldn't be any free page in victim blocks */
723           ftl_assert(pg_iter->status != PG_FREE);
724           if (pg_iter->status == PG_VALID) {
725               gc_read_page(ssd, ppa);
726               /* delay the maptbl update until "write" happens */
727               gc_write_page(ssd, ppa);
728               cnt++;
729
730               /* ----- WAF ----- */
731               GCWrite++;
732
733           }
734       }
735
736       ftl_assert(get_blk(ssd, ppa)->vpc == cnt);
737   }
```

The number of pages written
while performing the Garbage Collection

This part is related to the WAF. It is difficult to say that the writing that occurred during the GC process was caused **by the user's request**.

GCWrite measures how many pages were used during the GC process within the clean_one_block function.

ftl.c

```
796  static uint64_t ssd_read(struct ssd *ssd, NvmeRequest *req)
797  {
798      struct ssdparams *spp = &ssd->sp;
799      uint64_t lba = req->slba;
800      int nsecs = req->nlb;
801      struct ppa ppa;
802      uint64_t start_lpn = lba / spp->secs_per_pg;
803      uint64_t end_lpn = (lba + nsecs - 1) / spp->secs_per_pg;
804      uint64_t lpn;
805      uint64_t sublat, maxlat = 0;
806
807      if (end_lpn >= spp->tt_pgs) {
808          ftl_err("start_lpn=%"PRIu64",tt_pgs=%d\n", start_lpn, ssd->sp.tt_pgs);
809      }
810
811      /* normal IO read path */
812      for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
813          ppa = get_maptbl_ent(ssd, lpn);
814          if (!mapped_ppa(&ppa) || !valid_ppa(ssd, &ppa)) {
815              //printf("%s,lpn(%" PRId64 ") not mapped to valid ppa\n", ssd->ssdname, lpn);
816              //printf("Invalid ppa,ch:%d,lun:%d,blk:%d,pl:%d,pg:%d,sec:%d\n",
817              //ppa.g.ch, ppa.g.lun, ppa.g.blk, ppa.g.pl, ppa.g.pg, ppa.g.sec);
818              continue;
819          }
820
821          struct nand_cmd srd;
822          srd.type = USER_IO;
823          srd.cmd = NAND_READ;
824          srd.stime = req->stime;
825          sublat = ssd_advance_status(ssd, &ppa, &srd);
826          maxlat = (sublat > maxlat) ? sublat : maxlat;
827
828          /* ----- THROUGHPUT ----- */
829          totalRead += spp->secs_per_pg * 512;
830      }
831
832      return maxlat;
833  }
```

Number of sectors per page * sector size (512 bytes)

When measuring throughput, I measured read throughput and write throughput separately.

Then, I implemented throughput by adding read and write throughput.

I calculated the read throughput by multiplying the sector size per page by the sector size of 512 bytes.

ftl.c

```
835  static uint64_t ssd_write(struct ssd *ssd, NvmeRequest *req)
836  {
837      uint64_t lba = req->slba;
838      struct ssdparams *spp = &ssd->sp;
839      int len = req->nlb;
840      uint64_t start_lpn = lba / spp->secs_per_pg;
841      uint64_t end_lpn = (lba + len - 1) / spp->secs_per_pg;
842      struct ppa ppa;
843      uint64_t lpn;
844      uint64_t curlat = 0, maxlat = 0;
845      int r;
846
847      if (end_lpn >= spp->tt_pgs) {
848          ftl_err("start_lpn=%"PRIu64",tt_pgs=%d\n", start_lpn, ssd->sp.tt_pgs);
849      }
850
851      while (should_gc_high(ssd)) {
852          /* perform GC here until !should_gc(ssd) */
853          r = do_gc(ssd, true);
854          if (r == -1)
855              break;
856      }
857
858      for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
859          ppa = get_maptbl_ent(ssd, lpn);
860          if (mapped_ppa(&ppa)) {
861              /* update old page information first */
862              mark_page_invalid(ssd, &ppa);
863              set_rmap_ent(ssd, INVALID_LPN, &ppa);
864          }
865
866          /* new write */
867          ppa = get_new_page(ssd);
868          /* update maptbl */
869          set_maptbl_ent(ssd, lpn, &ppa);
870          /* update rmap */
871          set_rmap_ent(ssd, lpn, &ppa);
872
873          mark_page_valid(ssd, &ppa);
874
875          /* need to advance the write pointer here */
876          ssd_advance_write_pointer(ssd);
877
878          struct nand_cmd swr;
879          swr.type = USER_IO;
880          swr.cmd = NAND_WRITE;
881          swr.stime = req->stime;
882          /* get latency statistics */
883          curlat = ssd_advance_status(ssd, &ppa, &swr);
884          maxlat = (curlat > maxlat) ? curlat : maxlat;
885
886          /* ----- THROUGHPUT ----- */
887          totalWrite += spp->secs_per_pg * 512;
888      }
889
890      return maxlat;
891  }
```

Similarly, I calculated the write throughput by multiplying the sector size per page by the sector size of 512 bytes.

Number of sectors per page * sector size (512 bytes)

ftl.c

```
893  static void *ftl_thread(void *arg)
894  {
895      /* ----- IOPS, THROUGHPUT ----- */
896      struct timespec startTime;
897      clock_gettime(CLOCK_MONOTONIC, &startTime);
898
899      /* ----- IOPS ----- */
900      IOPSData = fopen("IOPSData.csv", "w");
901      fprintf(IOPSData, "time,read,write\n");
902
903      /* ----- THROUGHPUT ----- */
904      throughputData = fopen("throughputData.csv", "w");
905      fprintf(throughputData, "time,readThroughput,writeThroughput\n");
906
907      /* ----- GC, WAF ----- */
908      struct timespec startTimeGCWAF;
909      clock_gettime(CLOCK_MONOTONIC, &startTimeGCWAF);
910
911      /* ----- GC ----- */
912      GCData = fopen("GCData.csv", "w");
913      fprintf(GCData, "time,reclaimedBlocks\n");
914
915      /* ----- WAF ----- */
916      WAFData = fopen("WAFData.csv", "w");
917      fprintf(WAFData, "time,GCWrite\n");
918
919      FemuCtrl *n = (FemuCtrl *)arg;
920      struct ssd *ssd = n->ssd;
921      NvmeRequest *req = NULL;
922      uint64_t lat = 0;
923      int rc;
924      int i;
```

Preparation for measurement every second

Preparation for storing logs (IOPS, throughput)

Prepare for measurement every 10 seconds

Preparation for storing logs (GC, moved valid pages)

I designed a separate routine that measures IOPS and throughput every 1 second and a routine that measures GC and WAF related data every 10 seconds.

ftl.c

```
925
926     while (!*(ssd->dataplane_started_ptr)) {
927         usleep(100000);
928     }
929
930     /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
931     ssd->to_ftl = n->to_ftl;
932     ssd->to_poller = n->to_poller;
933
934     while (1) {
935         for (i = 1; i <= n->nr_pollers; i++) {
936             if (!ssd->to_ftl[i] || !femu_ring_count(ssd->to_ftl[i]))
937                 continue;
938
939             rc = femu_ring_dequeue(ssd->to_ftl[i], (void *)&req, 1);
940             if (rc != 1) {
941                 printf("FEMU: FTL to_ftl dequeue failed\n");
942             }
943
944             ftl_assert(req);
945             switch (req->cmd.opcode) {
946             case NVME_CMD_WRITE:
947                 lat = ssd_write(ssd, req);
948                 /* ----- IOPS ----- */
949                 outputOp++;
950                 break;
951             case NVME_CMD_READ:
952                 lat = ssd_read(ssd, req);
953                 /* ----- IOPS ----- */
954                 inputOp++;
955                 break;
956             case NVME_CMD_DSM:
957                 lat = 0;
958                 break;
959             default:
960                 //ftl_err("FTL received unkown request type, ERROR\n");
961                 ;
962             }
```

Write measurement process

Read measurement process

Continue to *ftl_thread.

I saved the values for each run of ssd_write and ssd_read.

Later, when drawing graphs in Python, I added outputOp and inputOp to get IOPS.

ftl.c

```
963
964             req->reqlat = lat;
965             req->expire_time += lat;
966
967             rc = femu_ring_enqueue(ssd->to_poller[i], (void *)&req, 1);
968             if (rc != 1) {
969                 ftl_err("FTL to_poller enqueue failed\n");
970             }
971
972             /* ----- IOPS, THROUGHPUT ----- */
973             struct timespec currentTime;
974             clock_gettime(CLOCK_MONOTONIC, &currentTime);
975             char *timeStamp = ctime(&(time_t){time(NULL)});
976
977             /* ----- GC, WAF ----- */
978             struct timespec currentTimeGCWAF;
979             clock_gettime(CLOCK_MONOTONIC, &currentTimeGCWAF);
980             char *timeStampGCWAF = ctime(&(time_t){time(NULL)});
981
982             if (currentTime.tv_sec - startTime.tv_sec >= 1) {
983
984                 /* ----- IOPS ----- */
985                 fprintf(IOPSData, "%.15s,%u,%u\n", timeStamp + 4, inputOp, outputOp);
986                 fflush(IOPSData);
987                 inputOp = 0;
988                 outputOp = 0;
989
990                 /* ----- THROUGHPUT ----- */
991                 double readThroughput = totalRead / (1024.0 * 1024.0);
992                 double writeThroughput = totalWrite / (1024.0 * 1024.0);
993                 fprintf(throughputData, "%.15s,%.5f,%.5f\n", timeStamp + 4, readThroughput, writeThroughput);
994                 fflush(throughputData);
995                 totalRead = 0;
996                 totalWrite = 0;
997
998                 startTime = currentTime;
999             }
1000
1001            if (currentTimeGCWAF.tv_sec - startTimeGCWAF.tv_sec >= 10) {
1002
1003                /* ----- GC ----- */
1004                fprintf(GCData, "%.15s,%u\n", timeStampGCWAF + 4, reclaimedBlocks);
1005                fflush(GCData);
1006                reclaimedBlocks = 0;
1007
1008                /* ----- WAF ----- */
1009                fprintf(WAFData, "%.15s, %u\n", timeStampGCWAF + 4, GCWrite);
1010                fflush(WAFData);
1011                GCWrite = 0;
1012
1013                startTimeGCWAF = currentTimeGCWAF;
1014            }
1015
1016            /* clean one line if needed (in the background) */
1017            if (should_gc(ssd)) {
1018                do_gc(ssd, false);
1019            }
1020        }
1021    }
1022
1023    return NULL;
1024 }
1025
```

Different timers are designed depending on the recording cycle

Save IOPS, throughput logs (Recorded every second)

Convert to MB/s

Time update(IOPS, throughput)

Time update(GC, moved valid pages)

Save GC, moved valid pages logs (Recorded every 10 seconds)

When saving throughput, read throughput and write throughput were stored separately to ensure that they were measured normally.

You can also check the process of initializing the variables for the next measurement.

(D) Data preprocessing and **visualization**

1

```
[1]: import pandas as pd
     import matplotlib.pyplot as plt
     import matplotlib.image as mpimg
```

```
[2]: n = pd.read_csv('IOPSData.csv').iloc[8:308, 1:]
     n.insert(0, 'time', range(1, 301))
     n.to_csv('newIOPSData.csv', index=False)
```

```
[3]: n = pd.read_csv('throughputData.csv').iloc[8:308, 1:]
     n.insert(0, 'time', range(1, 301))
     n.to_csv('newThroughputData.csv', index=False)
```

```
[4]: n = pd.read_csv('GCData.csv').iloc[3:33, 1:]
     n.insert(0, 'time', range(1, 31))
     n['time'] = 10 * n['time']
     n.to_csv('newGCData.csv', index=False)
```

```
[5]: n = pd.read_csv('WAFData.csv').iloc[3:33, 1:]
     n.insert(0, 'time', range(1, 31))
     n['time'] = 10 * n['time']
     n.to_csv('newWAFData.csv', index=False)
```

I considered that I needed to ignore the data before the measurement, such as in the **mount /dev/nvme0n1 /mnt/nvme0n1** process.

Ignore values measured **before the benchmark**

IOPS

2

```
[6]: dfIOPS = pd.read_csv('newIOPSData.csv')
     dfIOPS['IOPS'] = dfIOPS['read'] + dfIOPS['write']

     plt.figure(figsize=(10, 5))
     plt.tight_layout()

     plt.suptitle('IOPS over time', fontsize=15)
     plt.title('(i) Read 100%', fontsize=11)

     plt.xlabel('Seconds', fontsize=11)
     plt.ylabel('IOPS', fontsize=11)
     plt.grid(True, linestyle='--', alpha=0.5)

     plt.plot(dfIOPS['time'], dfIOPS['IOPS'], label='IOPS', color='darkred', linewidth=1.5)


     maxIOPS = dfIOPS['IOPS'].max()
     minIOPS = dfIOPS['IOPS'].min()
     maxSec = dfIOPS['time'][dfIOPS['IOPS'].idxmax()]
     minSec = dfIOPS['time'][dfIOPS['IOPS'].idxmin()]

     plt.scatter(maxSec, maxIOPS, color='red', s=15, label='Max')
     plt.scatter(minSec, minIOPS, color='blue', s=15, label='Min')
     plt.text(maxSec, maxIOPS, f'Max : {maxIOPS}', ha='left', va='bottom', color='red', fontsize=8)
     plt.text(minSec, minIOPS, f'Min : {minIOPS}', ha='center', va='top', color='blue', fontsize=8)

     plt.savefig('IOPSOverTime.png', dpi=600)
     plt.legend()
     plt.show()
```

It is the process of getting IOPS by adding read and write.

This is the visualization code. Configured to show max and min values in the graph.

1 An example of a process in which data is ignored before the benchmark measurement

throughputData.csv

iv > throughputData.csv

This part is ignored.

```
30  Oct 22 11:07:57,0.00000,75.19141
31  Oct 22 11:07:58,0.00000,75.52344
32  Oct 22 11:07:59,0.00000,75.59766
33  Oct 22 11:08:00,0.00000,75.67188
34  Oct 22 11:08:01,0.00000,75.64453
35  Oct 22 11:08:02,0.00391,75.56641
36  Oct 22 11:08:03,0.00000,75.78125
37  Oct 22 11:08:04,0.00000,75.21484
38  Oct 22 11:08:05,0.00000,75.62500
39  Oct 22 11:08:06,0.00000,75.59766
40  Oct 22 11:08:07,0.00000,75.64453
41  Oct 22 11:08:08,0.00000,75.50000
42  Oct 22 11:08:25,0.00000,1.32812
43  Oct 22 11:08:37,0.01562,0.07812
44  Oct 22 11:08:38,20.79688,21.35938
45  Oct 22 11:08:39,110.50000,113.01562
46  Oct 22 11:08:40,110.39062,113.17578
47  Oct 22 11:08:41,110.89062,113.35938
48  Oct 22 11:08:42,111.71875,114.11328
49  Oct 22 11:08:43,111.54688,113.87891
50  Oct 22 11:08:44,106.87500,109.44922
51  Oct 22 11:08:45,70.75000,72.72266
52  Oct 22 11:08:46,69.29688,70.99219
53  Oct 22 11:08:47,68.65625,70.03516
54  Oct 22 11:08:48,66.14062,67.75781
55  Oct 22 11:08:49,53.50000,54.56641
56  Oct 22 11:08:50,52.37500,53.43359
57  Oct 22 11:08:51,49.18750,50.02734
58  Oct 22 11:08:52,54.26562,55.55078
59  Oct 22 11:08:53,52.75000,53.71875
60  Oct 22 11:08:54,53.29688,54.38672
61  Oct 22 11:08:55,55.21875,56.30859
62  Oct 22 11:08:56,56.90625,57.90234
63  Oct 22 11:08:57,57.06250,58.00391
64  Oct 22 11:08:58,59.23438,60.22656
65  Oct 22 11:08:59,60.18750,61.50391
66  Oct 22 11:09:00,59.25000,60.36328
67  Oct 22 11:09:01,59.04688,60.23438
68  Oct 22 11:09:02,63.04688,64.29688
69  Oct 22 11:09:03,58.87500,60.04297
70  Oct 22 11:09:04,61.39062,62.42969
```

newThroughputData.csv

iv > newThroughputData.csv

```
1   time,readThroughput,writeThroughput
2   1,0.01562,0.07812
3   2,20.79688,21.35938
4   3,110.5,113.01562
5   4,110.39062,113.17578
6   5,110.89062,113.35938
7   6,111.71875,114.11328
8   7,111.54688,113.87891
9   8,106.875,109.44922
10  9,70.75,72.72266
11  10,69.29688,70.99219
12  11,68.65625,70.03516
13  12,66.14062,67.75781
14  13,53.5,54.56641
15  14,52.375,53.43359
16  15,49.1875,50.02734
17  16,54.26562,55.55078
18  17,52.75,53.71875
19  18,53.29688,54.38672
20  19,55.21875,56.30859
21  20,56.90625,57.90234
22  21,57.0625,58.00391
23  22,59.23438,60.22656
24  23,60.1875,61.50391
25  24,59.25,60.36328
26  25,59.04688,60.23438
27  26,63.04688,64.29688
28  27,58.875,60.04297
29  28,61.39062,62.42969
30  29,60.1875,60.87109
31  30,56.07812,57.39062
32  31,56.14062,57.15234
33  32,56.09375,57.11719
34  33,57.03125,58.05859
35  34,56.70312,57.52734
36  35,56.32812,57.30859
37  36,56.26562,57.33594
38  37,56.73438,57.84375
39  38,57.89062,59.25781
40  39,58.5625,59.82812
41  40,59.4375,60.48438
```

This is the process of preprocessing data from (iv) workloads.

Because I need to measure the data when **sysbench fileio --threads=4 --file-total-size=2304M --file-test-mode=rndrw --file-extra-flags=direct --file-fsync-all=on --time=300 run**.

3 Throughput

[7]:
```
dfThroughput = pd.read_csv('newThroughputData.csv')
dfThroughput['Throughput'] = dfThroughput['readThroughput'] + dfThroughput['writeThroughput']

plt.figure(figsize=(10, 5))
plt.tight_layout()

plt.suptitle('Throughput over time', fontsize=15)
plt.title('(i) Read 100%', fontsize=11)

plt.xlabel('Seconds', fontsize=11)
plt.ylabel('Throughput (MB/s)', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.5)

plt.plot(dfThroughput['time'], dfThroughput['Throughput'], label='Throughput (MB/s)', color='orangered', linewidth=1.5)


maxThroughput = dfThroughput['Throughput'].max()
minThroughput = dfThroughput['Throughput'].min()
maxSec = dfThroughput['time'][dfThroughput['Throughput'].idxmax()]
minSec = dfThroughput['time'][dfThroughput['Throughput'].idxmin()]

plt.scatter(maxSec, maxThroughput, color='red', s=15, label='Max')
plt.scatter(minSec, minThroughput, color='blue', s=15, label='Min')
plt.text(maxSec, maxThroughput, f'Max : {maxThroughput:.3f}', ha='left', va='bottom', color='red', fontsize=8)
plt.text(minSec, minThroughput, f'Min : {minThroughput:.3f}', ha='left', va='top', color='blue', fontsize=8)

plt.savefig('ThroughputOverTime.png', dpi=600)
plt.legend()
plt.show()
```

Read throughput and write throughput are added here.

Each graph is applied with different colors to make it easy to distinguish.

4 GC

[8]:
```
dfGC = pd.read_csv('newGCData.csv')

plt.figure(figsize=(10, 5))
plt.savefig('GCOverTime.png', dpi=300)
plt.tight_layout()

plt.suptitle('Number of erased blocks over time', fontsize=15)
plt.title('(i) Read 100%', fontsize=11)

plt.xlabel('Seconds', fontsize=11)
plt.ylabel('Erased blocks', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.5)

plt.plot(dfGC['time'], dfGC['reclaimedBlocks'], label='Erased blocks', color='forestgreen', linewidth=1.5)
plt.scatter(dfGC['time'], dfGC['reclaimedBlocks'], color='forestgreen', s=15)
plt.ylim(-1, 10)

maxGC = dfGC['reclaimedBlocks'].max()
minGC = dfGC['reclaimedBlocks'].min()
maxSec = dfGC['time'][dfGC['reclaimedBlocks'].idxmax()]
minSec = dfGC['time'][dfGC['reclaimedBlocks'].idxmin()]

plt.scatter(maxSec, maxGC, color='red', s=15, label='Max')
plt.scatter(minSec, minGC, color='blue', s=15, label='Min')
plt.text(maxSec, maxGC, f'Max : {maxGC}', ha='left', va='bottom', color='red', fontsize=8)
plt.text(minSec, minGC, f'Min : {minGC}', ha='left', va='top', color='blue', fontsize=8)

plt.savefig('GCOverTime.png', dpi=600)
plt.legend()
plt.show()
```

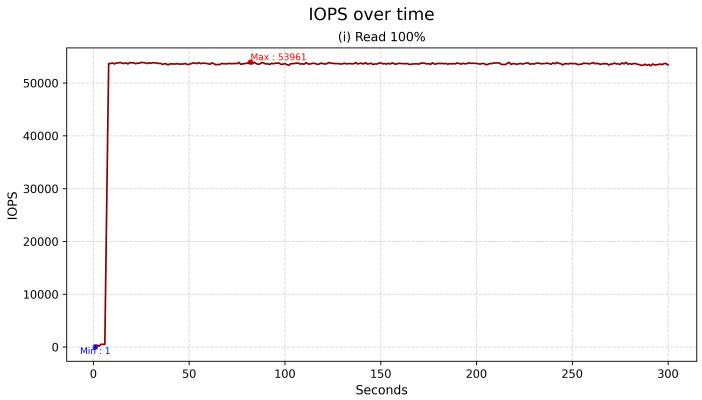
In a (i) read 100% workload, I added this part because the number of reclaimed blocks is 0.

Most visualization code is similar, but some code is added or missing depending on the data.

5 Moved valid pages

```
[9]: dfWAF = pd.read_csv('newWAFData.csv')

plt.figure(figsize=(10, 5))
plt.tight_layout()

plt.suptitle('Number of moved valid pages over time', fontsize=15)
plt.title('(i) Read 100%', fontsize=11)

plt.xlabel('Seconds', fontsize=11)
plt.ylabel('Moved valid pages', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.5)

plt.plot(dfWAF['time'], dfWAF['GCWrite'], label='Moved valid pages', color='darkcyan', linewidth=1.5)
plt.scatter(dfWAF['time'], dfWAF['GCWrite'], color='darkcyan', s=15)
plt.ylim(-1, 10)

maxWAF = dfWAF['GCWrite'].max()
minWAF = dfWAF['GCWrite'].min()
maxSec = dfWAF['time'][dfWAF['GCWrite'].idxmax()]
minSec = dfWAF['time'][dfWAF['GCWrite'].idxmin()]

plt.scatter(maxSec, maxWAF, color='red', s=15, label='Max')
plt.scatter(minSec, minWAF, color='blue', s=15, label='Min')
plt.text(maxSec, maxWAF, f'Max : {maxWAF}', ha='left', va='bottom', color='red', fontsize=8)
plt.text(minSec, minWAF, f'Min : {minWAF}', ha='left', va='top', color='blue', fontsize=8)

plt.savefig('MovedValidPagesOverTime', dpi=600)
plt.legend()
plt.show()
```

Each graph is stored as a png file.



This is the result of the execution of **2**.
Other graphs can be found in the (B) part.

(E) Device Configuration

```
# Configurable SSD Controller layout parameters (must be power of 2)
secsz=512 # sector size in bytes
secs_per_pg=8 # number of sectors in a flash page
pgs_per_blk=256 # number of pages per flash block
blks_per_pl=256 # number of blocks per plane
pls_per_lun=1 # keep it at one, no multiplanes support
luns_per_ch=8 # number of chips per channel
nchs=2 # number of channels
ssd_size=3072 # in megabytes, if you change the above layout parameters,
```

I used a **4GB (3GB, 25% OP)** SSD Setting.

This is why I can observe the **write cliff** relatively **early**!