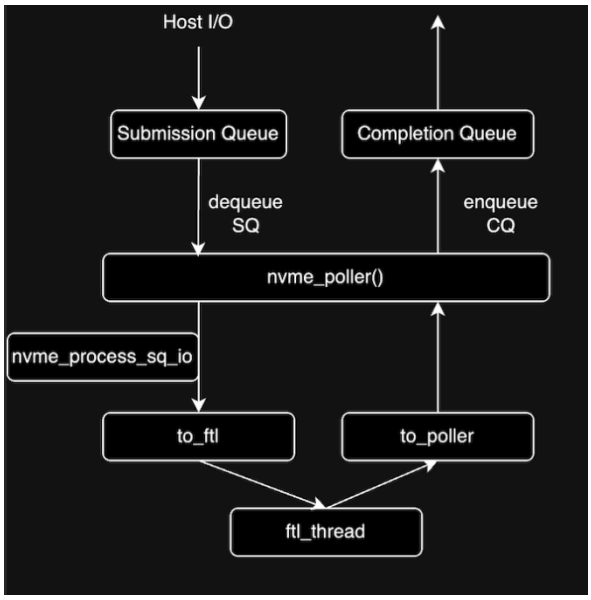
System Programming
Project1 Report

2019040519
Taehyung Kim

1. Structure Diagram



Host I/O: I/O commands from the host are sent to the Submission Queue (SQ).

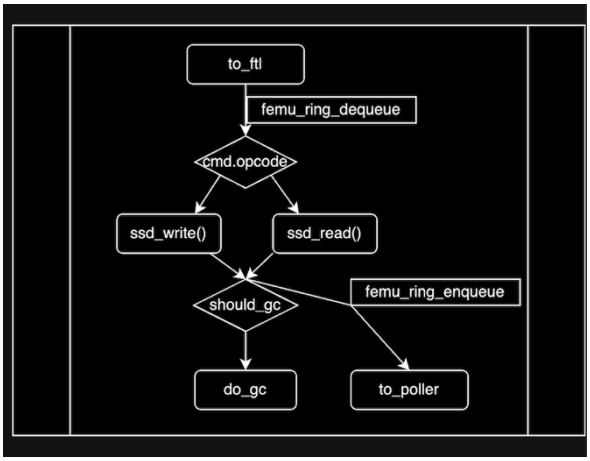
nvme_poller(): This function fetches commands from the SQ and processes them, forwarding them to nvme_process_sq_io().

to_ftl: The processed commands are stored in this queue to be sent to the FTL.

ftl_thread(): The FTL thread retrieves commands from the queue for further processing.

Completion Queue (CQ): Completed I/O commands are transferred to the CQ.

nvme_poller(): also manages the CQ, notifying the host once the commands are completed.



femu_ring_dequeue(): This function retrieves commands from the to_ftl queue.

Depending on the cmd.opcode, either **ssd_write()** or **ssd_read()** is called.

should_gc(): This function checks if garbage collection (GC) is required.

do_gc(): If GC is necessary, it is performed.

Once the command is processed, the result is placed in the to_poller queue using **femu_ring_enqueue().**

2. Code Analysis

2-1. I/O Request Mechanism Analysis

- **nvme_poller()**

```c
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;
}
```

```c
typedef struct NvmeSQueue {
    struct FemuCtrl *ctrl;
    uint8_t     phys_contig;
    uint8_t     arb_burst;
    uint16_t    sqid;
    uint16_t    cqid;
    uint32_t    head;
    uint32_t    tail;
    uint32_t    size;
    uint64_t    dma_addr;
    uint64_t    dma_addr_hva;
    uint64_t    completed;
    uint64_t    *prp_list;
    NvmeRequest *io_req;
    QTAILQ_HEAD(sq_req_list, NvmeRequest) req_list;
    QTAILQ_HEAD(out_req_list, NvmeRequest) out_req_list;
    QTAILQ_ENTRY(NvmeSQueue) entry;

    uint64_t    db_addr;
    uint64_t    db_addr_hva;
    uint64_t    eventidx_addr;
    uint64_t    eventidx_addr_hva;
    bool        is_active;
} NvmeSQueue;
```

```c
typedef struct NvmeCQueue {
    struct FemuCtrl *ctrl;
    uint8_t     phys_contig;
    uint8_t     phase;
    uint16_t    cqid;
    uint16_t    irq_enabled;
    uint32_t    head;
    uint32_t    tail;
    uint32_t    vector;
    uint32_t    size;
    uint64_t    dma_addr;
    uint64_t    dma_addr_hva;
    uint64_t    *prp_list;
    EventNotifier guest_notifier;
    QEMUTimer   *timer;
    QTAILQ_HEAD(sq_list, NvmeSQueue) sq_list;
    QTAILQ_HEAD(cq_req_list, NvmeRequest) req_list;
    int32_t     virq;
    MSIMessage  msg;
    uint64_t    db_addr;
    uint64_t    db_addr_hva;
    uint64_t    eventidx_addr;
    uint64_t    eventidx_addr_hva;
    bool        is_active;
} NvmeCQueue;
```

User's host I/O arrives in NVMe format (using Logical Block Addressing (LBA)) to **nvme_poller().**

Inside the switch statement, the FemuCtrl controller checks whether the multi-poller mode is enabled.(default: not enabled)

The **nvme_poller()** runs in an infinite loop and waits until the dataplane starts. After the dataplane starts, the poller retrieves: **Submission Queue**, **Completion Queue**

It checks the is_active flag of both the SQ and CQ. If both are active, it calls the **nvme_process_sq_io()** function to handle I/O operations.

Once the I/O is completed by **ftl_thread()**, it invokes **nvme_process_cq_cpl()** to forward the result to the Completion Queue.

- **nvme_process_sq_io()**

```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;
}
```

```c
void nvme_update_sq_tail(NvmeSQueue *sq)
{
    if (sq->db_addr_hva) {
        sq->tail = *((uint32_t *)sq->db_addr_hva);
        return;
    }

    if (sq->db_addr) {
        nvme_addr_read(sq->ctrl, sq->db_addr, &sq->tail, sizeof(sq->tail));
    }
}
```

The first step in nvme_process_sq_io() is to call **nvme_update_sq_tail().** This function ensures that the tail pointer of the **Submission Queue** is correctly updated. And the nvme_process_sq_io() function fetches and processes the following commands from the submission queue until the submission queue is empty during the while loop.

If the fetched command is **continuous memory**: Calculate and copy the address of the command directly. If it is **non-continuous memory,** calculate and read the address of each command part.

After that, update the head pointer to move to the next command.

Retrieves the first request from req_list. save information such as start time, expire time, command ID, and opcode. The request is then passed to the **nvme_io_cmd()** function for further handling.

- **nvme_io_cmd()**

```
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;
    }
}
```

It executes commands based on the opcode of the received nvme command.

Depending on the opcode, it performs **flush**: flush the data from volatile cache to non volatile storage, **dsm**: data set management, **compare**: compare data blocks for correctness. **write_zeroes**: fill a specified block with zeroes, **write_uncor:** mark a block as uncorrectable, **default**: check if an

external command handler is registered. if the opcode is not supported, it returns an error.

• **femu_ring_enqueue()**

```
    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;
```

```
size_t femu_ring_enqueue(struct rte_ring *ring, void **objs, size_t count)
{
    return rte_ring_enqueue_bulk((struct rte_ring *)ring, objs, count, NULL);
}
```

If the command is successfully processed, **femu_ring_enqueue()** enqueues the request to the **to_ftl Queue**. The total number of processed commands is updated with each iteration.

• **ftl_thread()**

```
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;
```

Allocates **to_ftl** and **to_poller** queues to manage command flows.

```
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");
        }
```

In an infinite loop, when a request from nvme-process_sq_io() arrives in the **to_ftl** queue, **femu_ring_dequeue()** is used to dequeue the request.

```
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:
        break;
}
```

Depending on the cmd.opcode (e.g., read, write, dsm), it performs the corresponding operation and store the latency of each i/o operation.

```
            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;
}
```

Save the latency and update the expiration time for each request.

Requests completed through the **femu_ring_enqueue()** function are then enqueue in the **to_poller** queue. After that, check that the conditions to perform gc(garbage collection), and perform gc if the threshold value exceeds 75%.

These processes repeats until the program terminates.

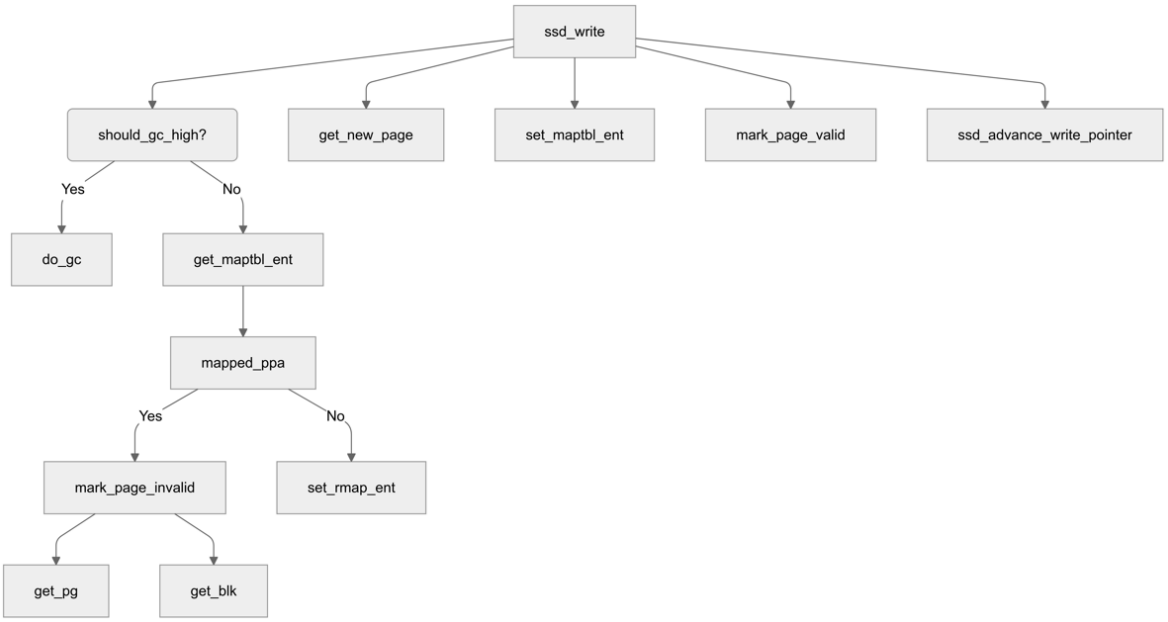
• **nvme_process_cq_cpl()**

```
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;
    }
}
```

nvme_process_cq_cpl() obtains the completed request through to_poller using **femu_ring_dequeue()** function. Thereafter, the request is inserted into the priority queue.

Pop() all requests from the priority queue to verify the completion time, update the expired request to the completion queue, insert the completed request back into the **submssion queue**, and call Interrupt Service Routine (ISR) if necessary.

- Write Mechanism Analysis



In the **ftl_thread(),** the opcode of the request (req->cmd.opcode) is checked.

If it matches NVME_CMD_WRITE, the **ssd_write()** function is executed.

```
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");
    ;
}
```

• **ssd_write()**

```
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;
```

```
typedef struct NvmeRequest {
    struct NvmeSQueue       *sq;
    struct NvmeCQueue       *cq;
    struct NvmeNamespace    *ns;
    uint16_t                status;
    uint64_t                slba;
    uint16_t                is_write;
    uint16_t                nlb;
    uint16_t                ctrl;
```

ssd_write() function takes ssd and NvmeRequest struct as parameters.

Starting LPN is calculated by dividing the starting LBA by the secs_per_pg.
Ending LPN is calculated by determining the LBA of the last logical block (lba+len-1) / secs_per_pg.

Since write operations are managed at the **page level**, it maps **sector-based requests (LBA) to page-based addresses (LPN)**.

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

It enters a while loop with **should_gc_high(ssd).**

If the GC threshold (95%) is exceeded, **GC(Garbage Collection)** is repeatedly performed to clean up the space occupied by victim lines.

```
// Iterate over each LPN in the given range to perform the write.
for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
    ppa = get_maptbl_ent(ssd, lpn);  // Retrieve the current PPA for the LPN.

    if (mapped_ppa(&ppa)) {  // If the page is already mapped, invalidate it.
        mark_page_invalid(ssd, &ppa);
        set_rmap_ent(ssd, INVALID_LPN, &ppa);  // Update reverse mapping.
    }
```

The function loops through the pages to be written, fetching the PPA for each LPN from the mapping table **with get_maptbl_ent().**

If a page is already mapped, it is marked as invalid using **mark_page_invalid()**, and the reverse mapping is updated with **set_rmap_ent().**

```
/* 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);
```

```
static struct ppa get_new_page(struct ssd *ssd)
{
    struct write_pointer *wpp = &ssd->wp;
    struct ppa ppa;
    ppa.ppa = 0;
    ppa.g.ch = wpp->ch;
    ppa.g.lun = wpp->lun;
    ppa.g.pg = wpp->pg;
    ppa.g.blk = wpp->blk;
    ppa.g.pl = wpp->pl;
    ftl_assert(ppa.g.pl == 0);

    return ppa;
}
```

A new page is allocated using **get_new_page()** and the PPA is returned. The mapping table is updated by **set_maptbl_ent().** The reverse mapping table is also updated using **set_rmap_ent().**

```
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++;
}
```

The newly allocated page is then set to a valid state using **mark_page_valid().**

In this functoin, page, block and line statuses are also updated accordingly, which plays a significant role in the Garbage Collection.

```
        ssd_advance_write_pointer(ssd);  // Move the write pointer.

        struct nand_cmd swr = {
            .type = USER_IO,  // Specify the command type as user I/O.
            .cmd = NAND_WRITE,
            .stime = req->stime
        };

        // Calculate the latency for the write and track the maximum latency.
        curlat = ssd_advance_status(ssd, &ppa, &swr);
        maxlat = (curlat > maxlat) ? curlat : maxlat;
    }

    return maxlat;  // Return the highest latency encountered.
}
```

• **ssd_advance_write_pointer()**

```c
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;
                /* 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++;
                }
                /* 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);
            }
        }
    }
}
```

ssd_advance_write_pointer() function moves channels, LUNs(=chips), and blocks sequentially and updates the next write location.

In FEMU, each blocks managed by **line**. When an I/O request is received, I/O is processed for the same block index, which distributes writes evenly across Channels, LUNs in parallel.

This function checks the status of a line and moves it to the **full line list**,
if all pages are valid and to **victim priority queue** if some pages are invalid.

• **ssd_advance_status()**

```
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;
}
```

It calculates the total latency based on the time difference between the start time and the completion time for each operation.

The **maximum latency** across all pages involved in the write is determined and returned.

- Read Mechanism Analysis

In the ftl_thread(), **ssd_read()** invoked by opcode **NVME_CMD_READ**.

```
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");
    ;
}
```

• **ssd_read()**

```
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;
        }
```

start_lpn and end_lpn are calculated by dividing the **LBA** by the number of sectors per page(secs_per_pg).

After that, Obtain the PPA for each LPN in a given range using the **get_maptbl_ent()** function.

If the PPA is invalid or not mapped, the function skips to the next LPN without.

```
        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;
}
```

nand_cmd is filled with read request status(type, cmd, time)

It calls ssd_advance_status() to caculate sublat and determine maxlat across all pages involved in the read.

- Garbage Collection Mechanism Analysis

• Should_gc(), should_gc_high: GC trigger condition

```
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);
}
```

```
static void ssd_init_params(struct ssdparams *spp, FemuCtrl *n)
{
 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);
 spp->enable_gc_delay = true;
```

GC is performed when **threshold** of the lines are already used or filled with invalid pages.

run-blackbox.sh

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

In run-blackbox shell scripts, gc_thres_pcent is 75, and gc_thres_pcent_high is 95

- **shoud_gc()** (low threshold): **75%** -> This means that only 25 percent or less of total lines can use space.

- **should_gc_high** (high threshold): **95**% -> This means that only 5 percent or less of total lines can use space.

• **Select_victim_line(): Victim Line Selection**

```c
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;
    }
```

If the do_gc() function is called by should_gc() or should_gc_high(), the **victim line** is selected.

```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;
}
```

It use **Priority Queue**(victim_line_pq) to select a victim line.

If the **ipc (invalid page cnt)** of the victim line is **less than the pages per line/8**, gc is not performed by returning NULL.

If victim_line is selected normally, remove it from pq to **pqueue_pop(),** initialize the pointer position of victim_line, reduce the victim_line_cnt, and return the selected victim line.

• **What is Priority of pq?**

```c
static void ssd_init_lines(struct ssd *ssd)
{
    struct ssdparams *spp = &ssd->sp;
    struct line_mgmt *lm = &ssd->lm;
    struct line *line;

    lm->tt_lines = spp->blks_per_pl;
    ftl_assert(lm->tt_lines == spp->tt_lines);
    lm->lines = g_malloc0(sizeof(struct line) * lm->tt_lines);

    QTAILQ_INIT(&lm->free_line_list);
    lm->victim_line_pq = pqueue_init(spp->tt_lines, victim_line_c
            victim_line_get_pri, victim_line_set_pri,
            victim_line_get_pos, victim_line_set_pos);
    QTAILQ_INIT(&lm->full_line_list);
```

```c
static inline int victim_line_cmp_pri(pqueue_pri_t next, pqueue_pri_t curr)
{
    return (next > curr);
}

static inline pqueue_pri_t victim_line_get_pri(void *a)
{
    return ((struct line *)a)->vpc;
}

static inline void victim_line_set_pri(void *a, pqueue_pri_t pri)
{
    ((struct line *)a)->vpc = pri;
}

static inline size_t victim_line_get_pos(void *a)
{
    return ((struct line *)a)->pos;
}

static inline void victim_line_set_pos(void *a, size_t pos)
{
    ((struct line *)a)->pos = pos;
}
```

• The **victim_line_get_pri()** function returns **vpc** from the line structure.

-> **A line with fewer valid pages** is selected as the victim line.

• **Clean_one_block(): Copy valid pages**

```c
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;
```

```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);
}
```

• Why get id of victim line to ppa.g.blk? (ppg.g.blk = victim->line)

Blocks on the same line are different chips, but they all have **the same block number**. Therefore, the ID of the victim line is stored so that the block of that line can be processed.

Clean_one_block() is performed while traversing all channels and the luns(=chips) within each channel to copy the valid pages to a new location.

If it is a valid page, read the page contents through **gc_read_page(),** and write the original page

information on another page through **gc_write_page().**

• gc_read_page():

```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);
    }
}
```

By default, gc_delay is turned on, so it goes inside the conditional statement. It then creates a nand_cmd structure to define the nand_read command, and since it is a garbage collection operation, the type is set to GC_IO. Finally, call ssd_advance_status() to calculate the read latency

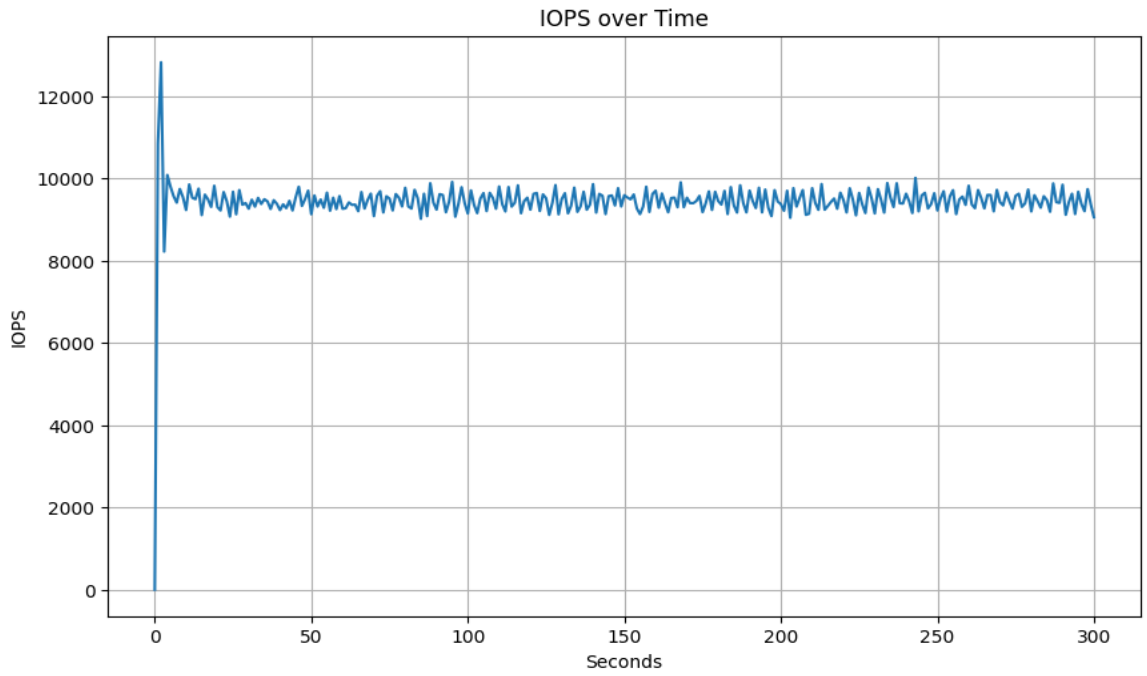
.• gc_write_page()

```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;
}
```

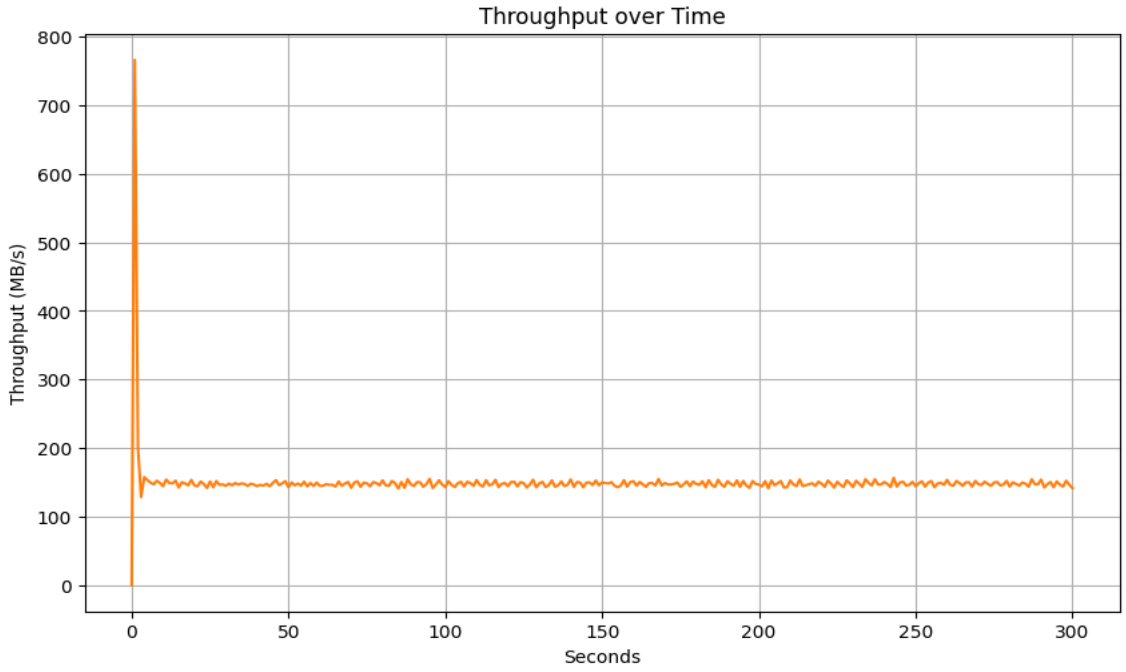
The gc_write_page() function moves a valid page from the victim block to a new location by allocating a new page with get_new_page(). It updates the mapping table (set_maptbl_ent()), reverse mapping table (set_rmap_ent()), and marks the new page as valid with mark_page_valid().
The write pointer is advanced using ssd_advance_write_pointer(), and if GC delay is enabled, the function updates the NAND status with ssd_advance_status().

2. Performance Analysis using benchmark (1GB SSD setting)

1-1. fio randrw: IOPS over time



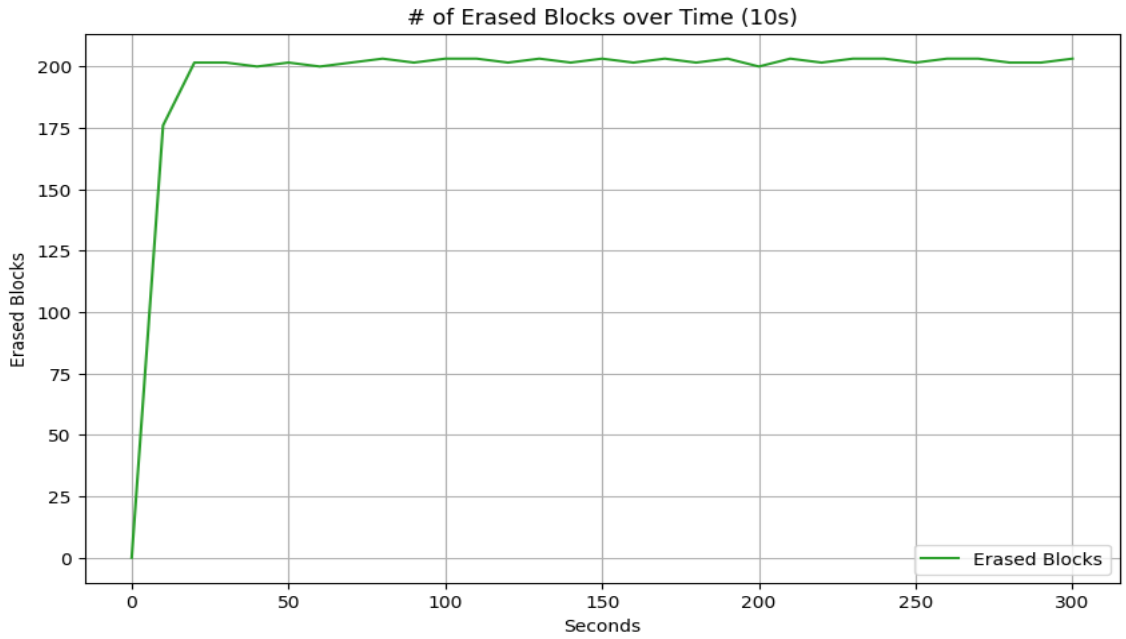
1-2. fio randrw: Throughput over time



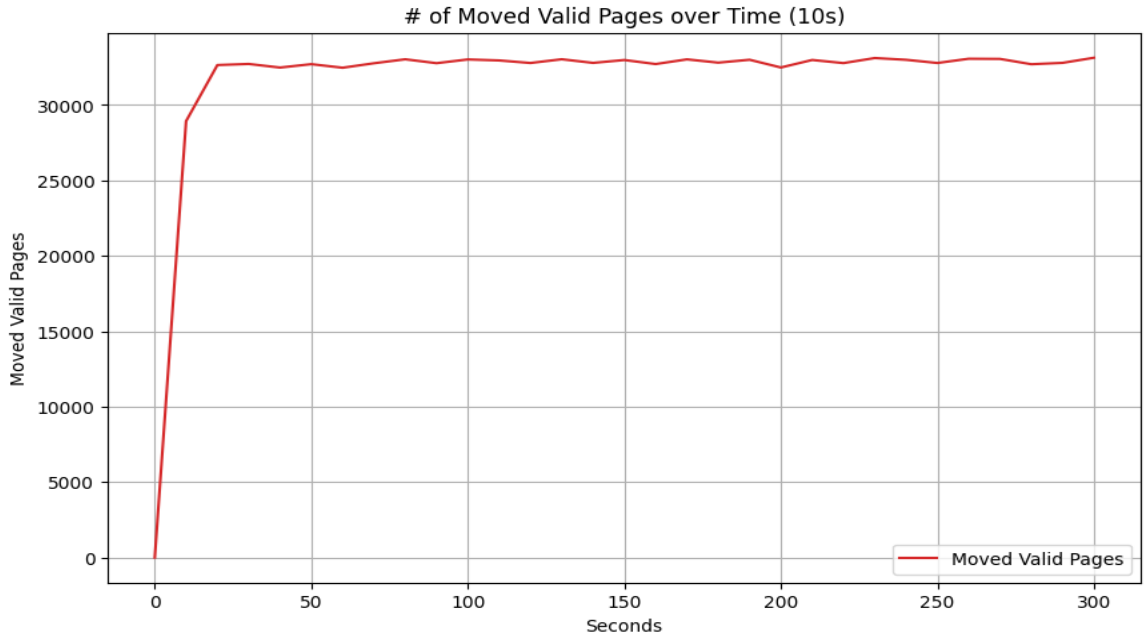
The write cliff phenomenon is observed as IOPS and throughput initially increase, then drop sharply, and finally stabilize at a lower level.

In FEMU, free lines fall below a threshold, garbage collection (GC) is triggered, causing write operations to be temporarily delayed or paused, resulting in a significant drop in performance.

1-3. fio randrw: # Erased blocks over time

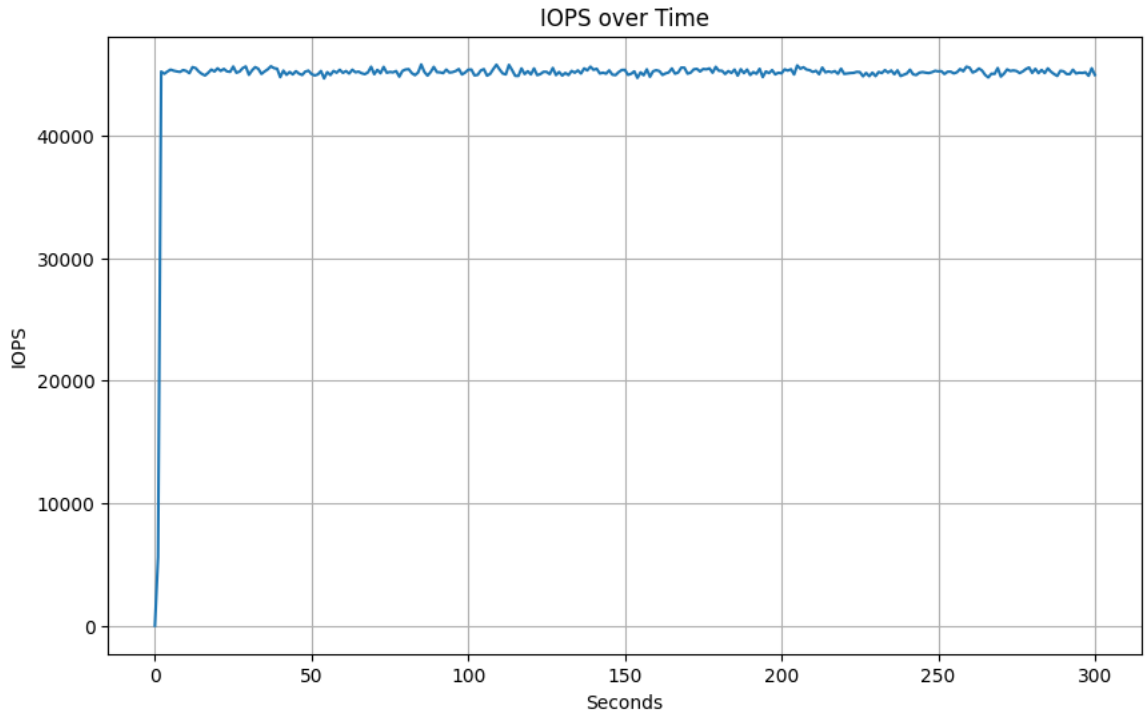


1-4. fio randrw: # valid pages moved over time

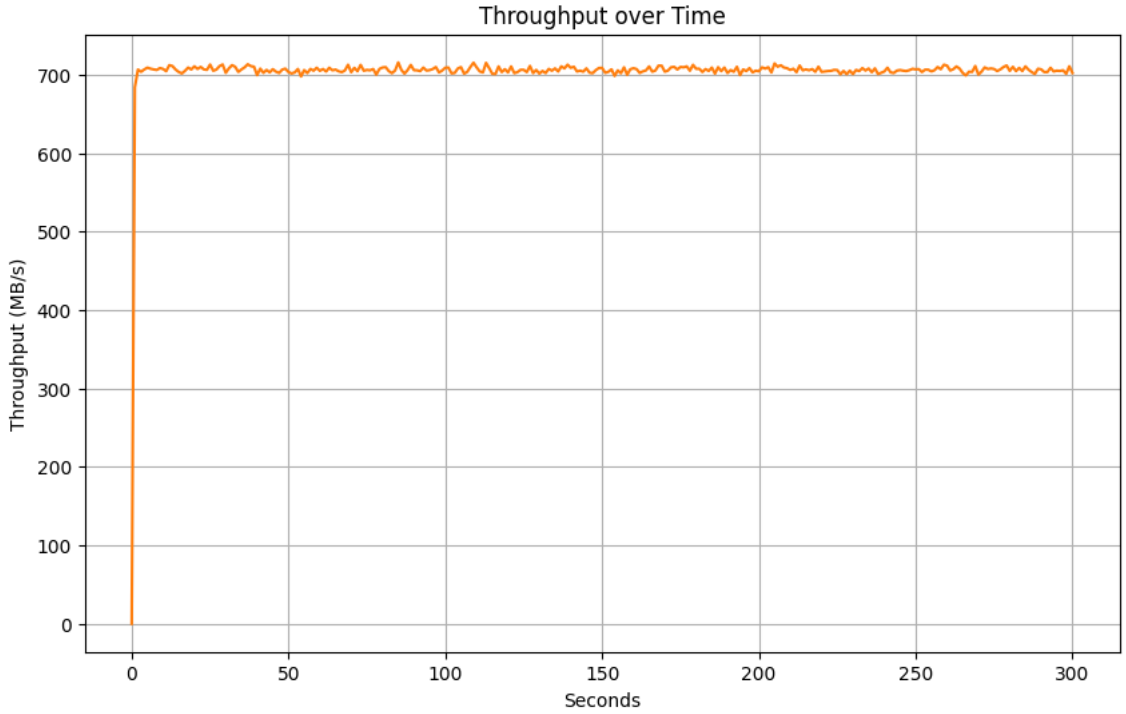


These graphs clearly illustrate the impact of garbage collection. Initially, GC is triggered, causing a sharp increase in both the number of Erased Blocks and Moved Valid Pages. Subsequently, GC continues at a steady rate, leading to the stabilization of both values.

2-1. fio randread: IOPS over time

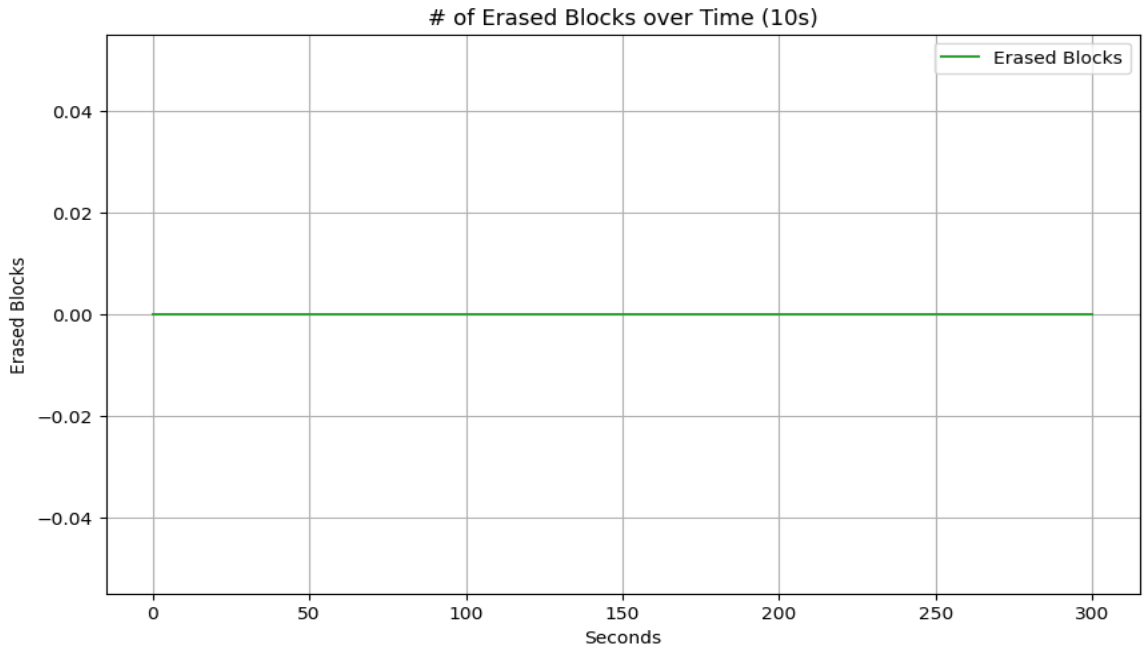


2-2. fio randread: Throughput over time

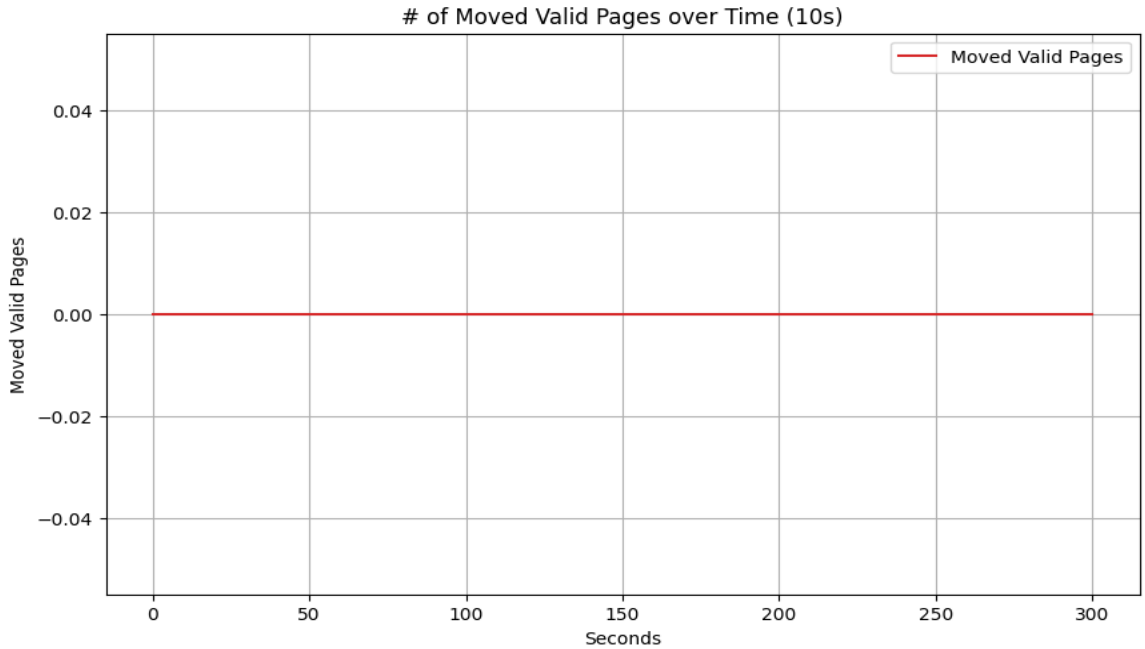


There is no significant drop or fluctuation, indicating that random read operations are unaffected by garbage collection or performance degradation.

2-3. fio randread: # Erased blocks over time



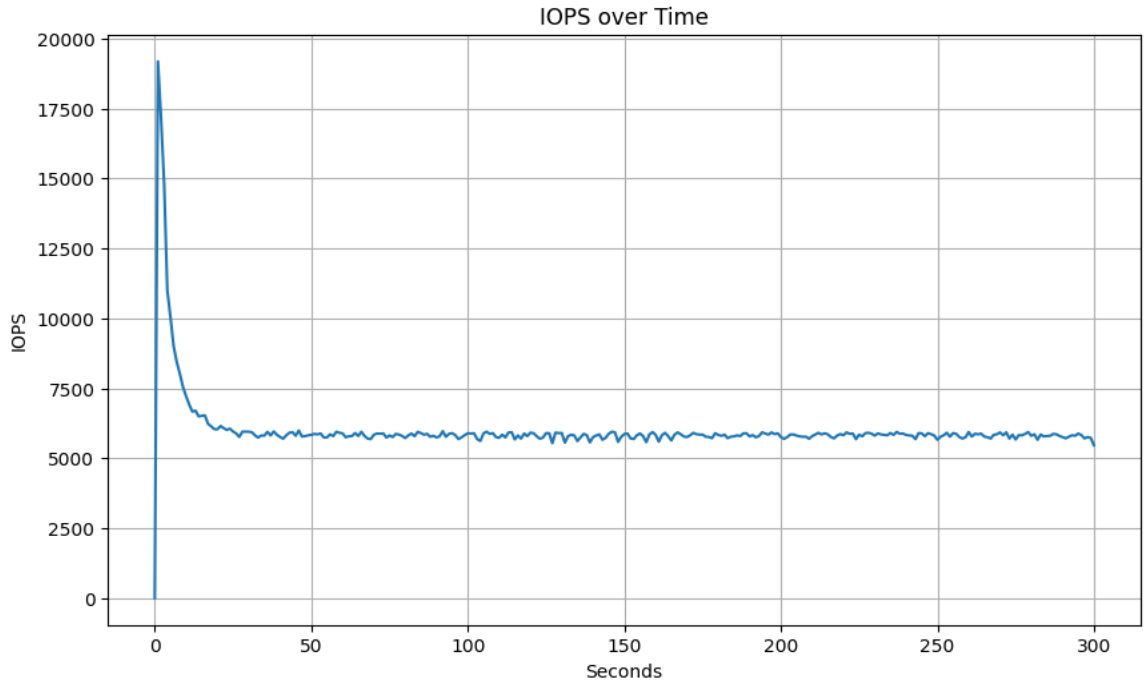
2-4. fio randread: # valid pages moved over time



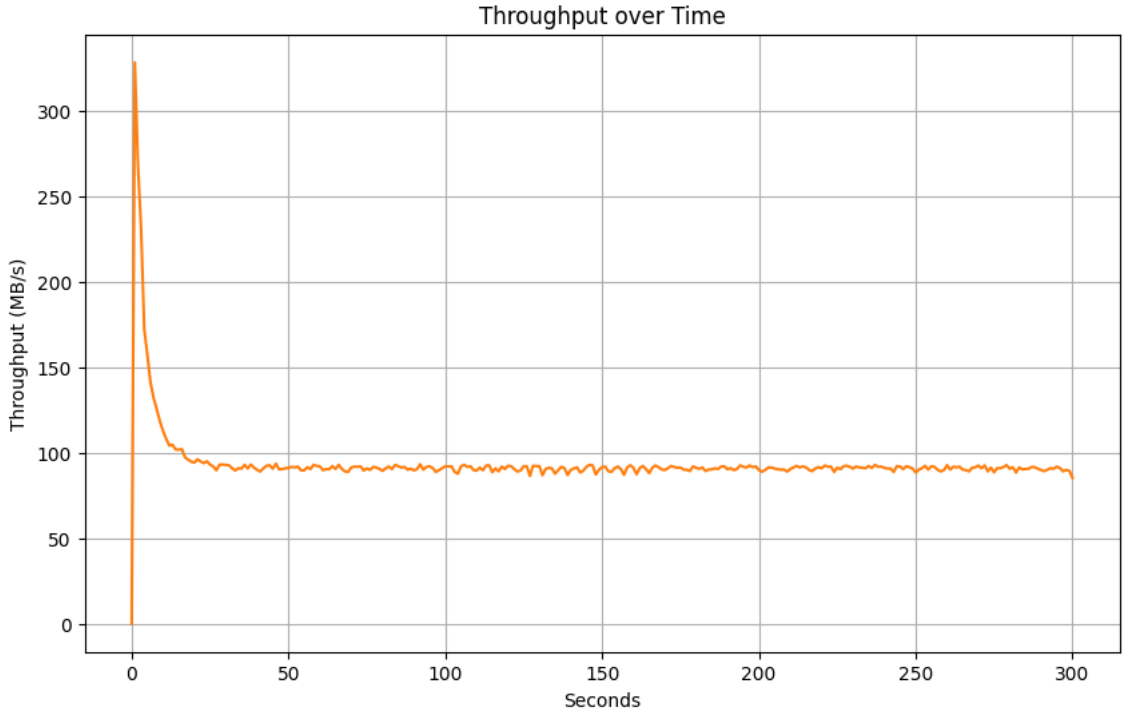
The graphs for erased blocks and moved valid pages during the read operation both remain at zero throughout the test.

This behavior is expected because read operations do not modify data, erase blocks, or trigger garbage collection. As a result, no blocks are erased, and no valid pages are moved during the read process, confirming stable read performance.

3-1. fio randwrite: IOPS over time

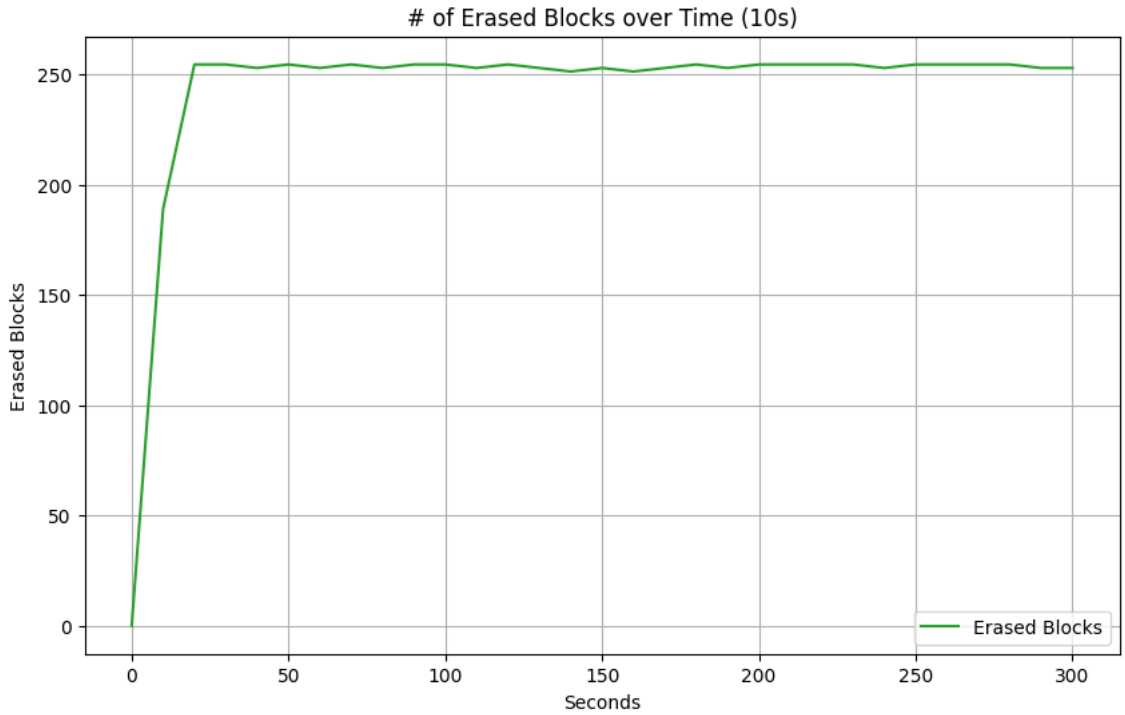


3-2. fio randwrite: Throughput over time

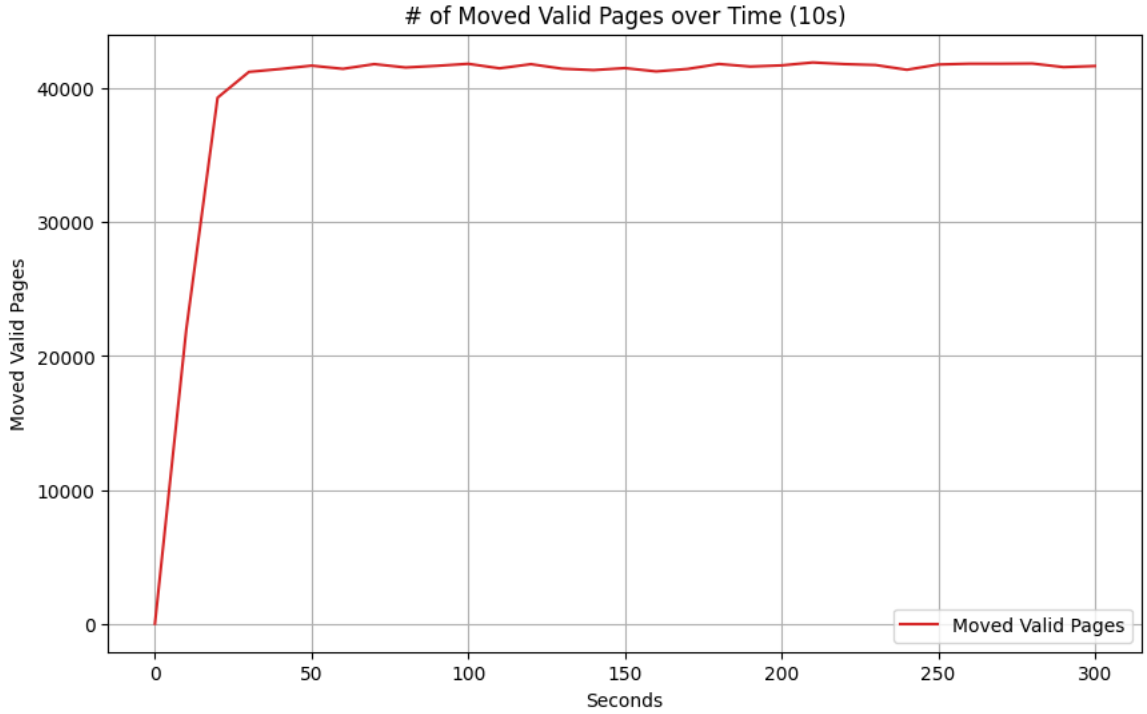


These graphs show write cliffs in IOPS and throughput. It can be observed that the stabilized IOPS and throughput values are lower compared to those in the randrw graphs. This is because the write operations were concentrated due to the randwrite workload (write 100%), leading to increased garbage collection activity and performance degradation.

3-3. fio randwrite: # Erased blocks over time

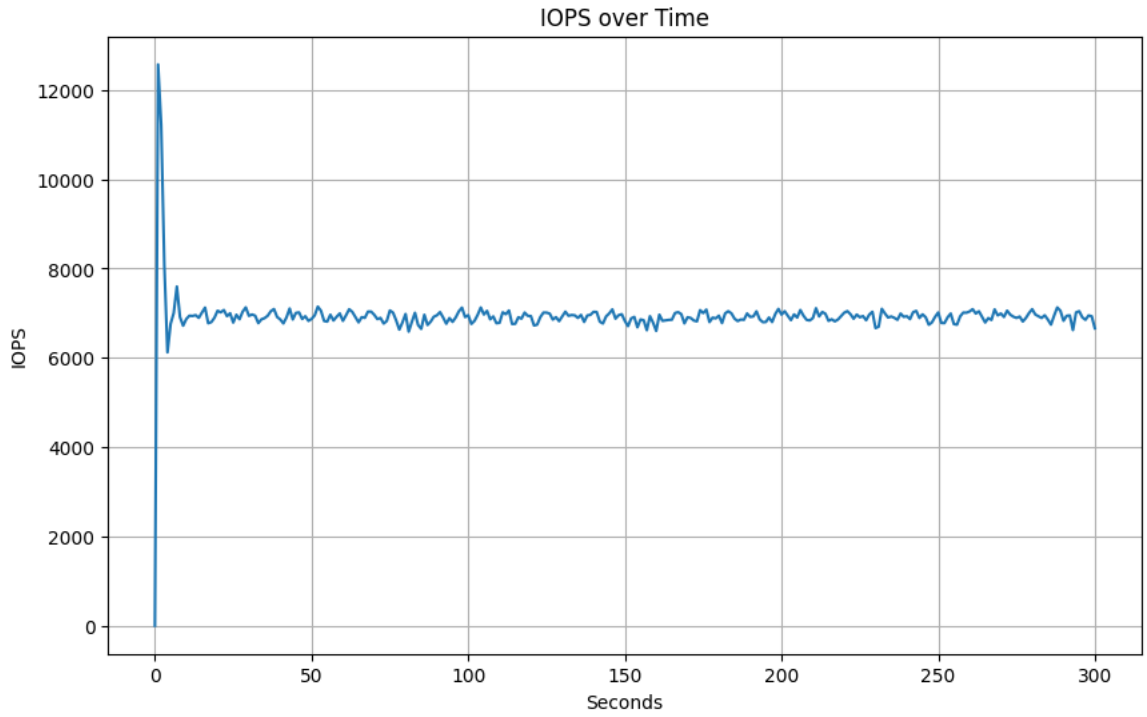


3-4. fio randwrite: # valid pages moved over time

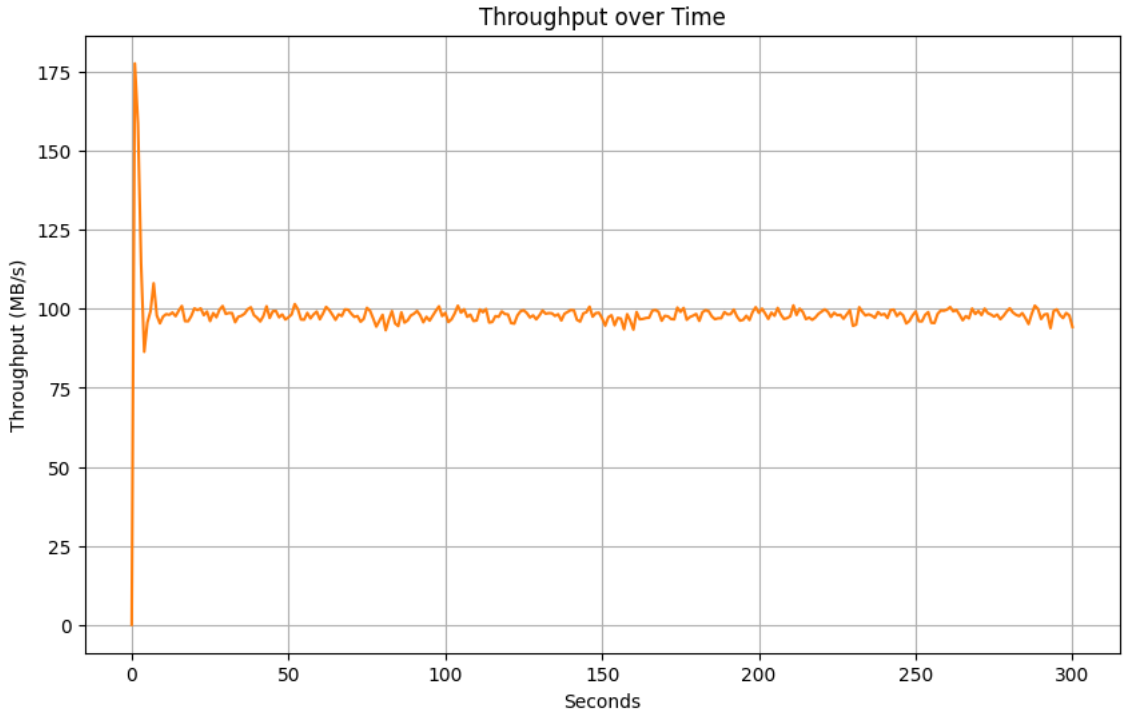


Similarly, the erased blocks and moved valid pages are observed to record higher values, which can be attributed to the write 100% workload.

4-1. sysbench(4threads): IOPS over time

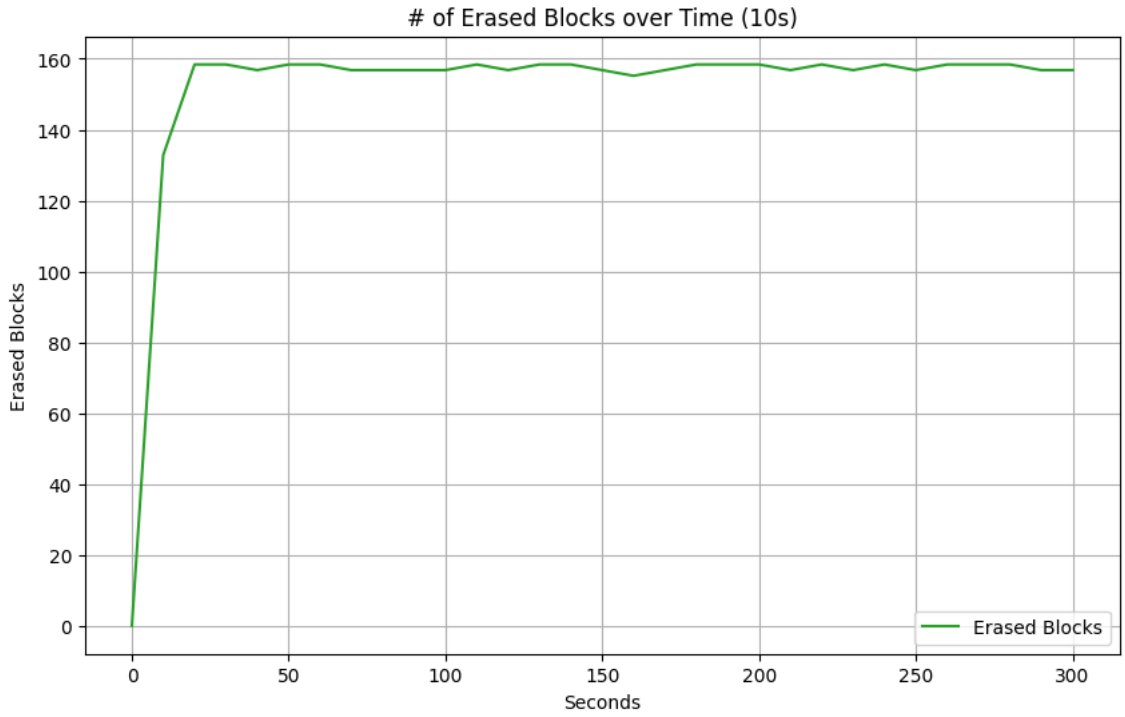


4-2. sysbench(4threads): Throughput over time

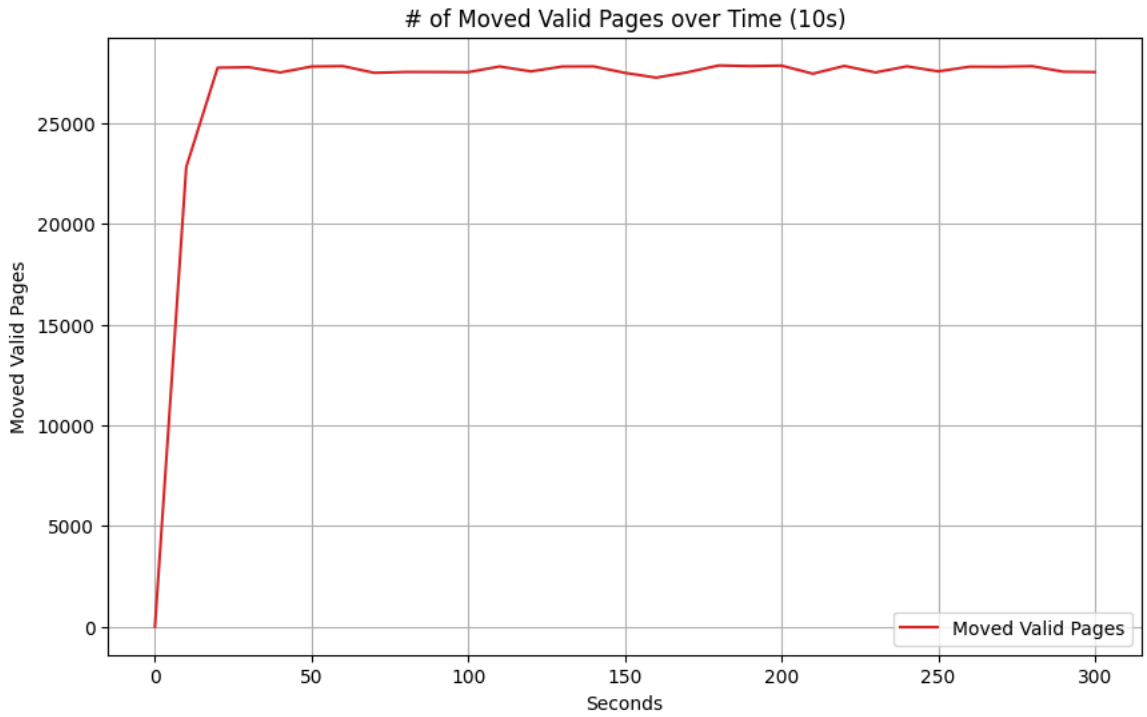


The write cliff phenomenon is observed, where IOPS and throughput initially surge, then drop sharply, and eventually stabilize at a lower level. This behavior, similar to FIO workloads, occurs due to garbage collection being triggered as free lines fall below a threshold.

4-3. sysbench(4threads): # Erased blocks over time

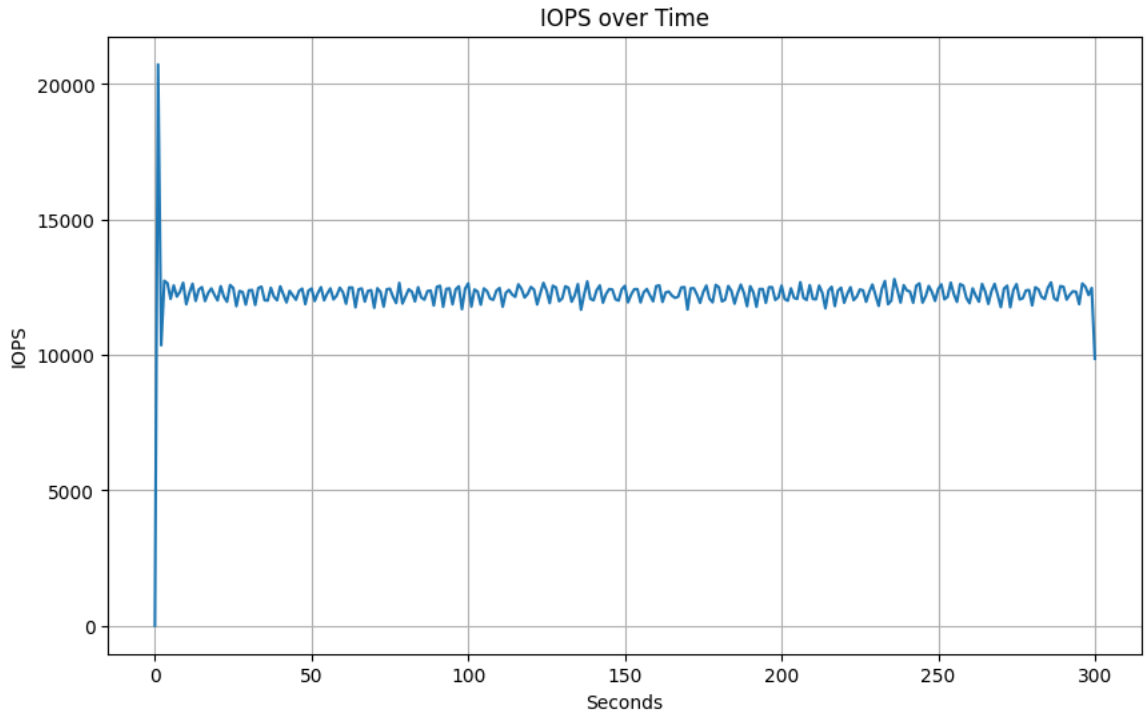


4-4. sysbench(4threads): # valid pages moved over time

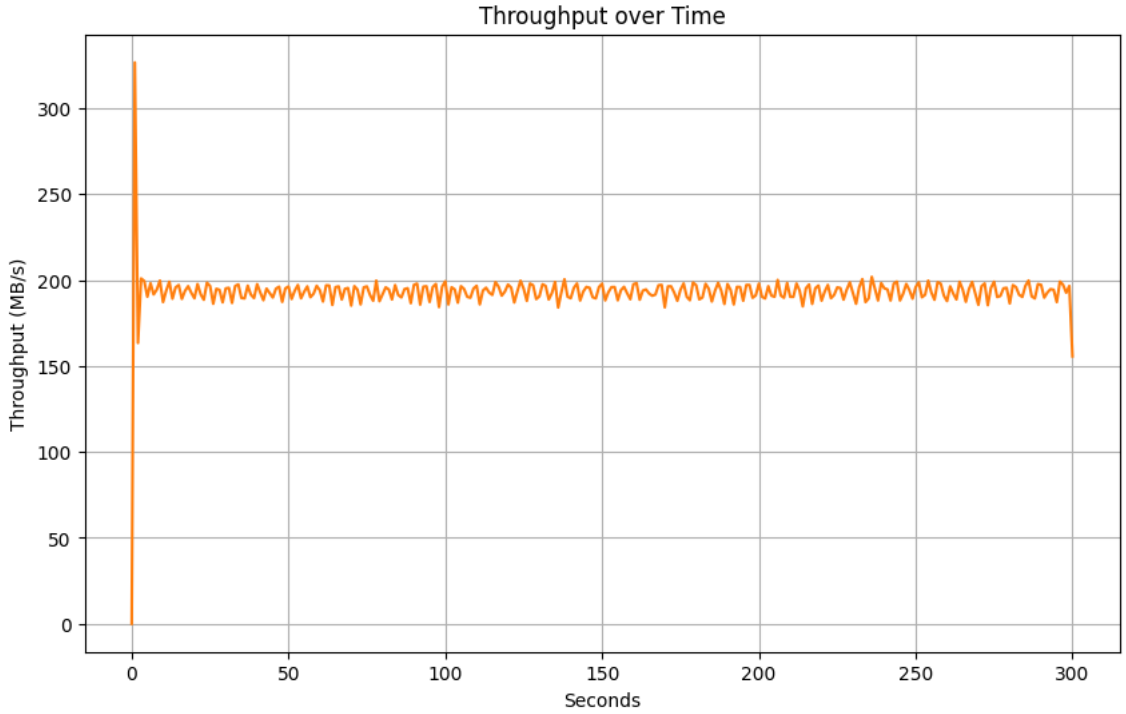


These graphs show a sharp increase in erased blocks and moved valid pages at the start, stabilizing over time. This aligns with the write cliff phenomenon, where garbage collection is triggered as free lines decrease. GC relocates valid pages and erases blocks, causing the earlier drop in IOPS and throughput due to temporary write delays.

5-1. sysbench(32threads): IOPS over time



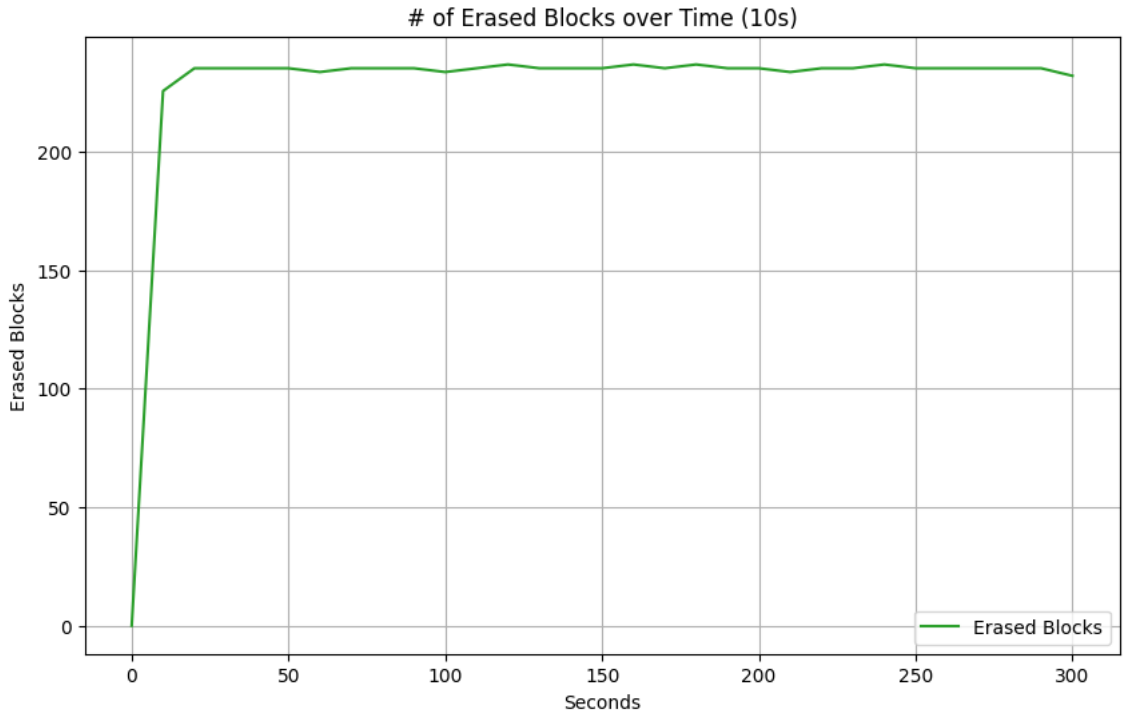
5-2. sysbench(32threads): Throughput over time



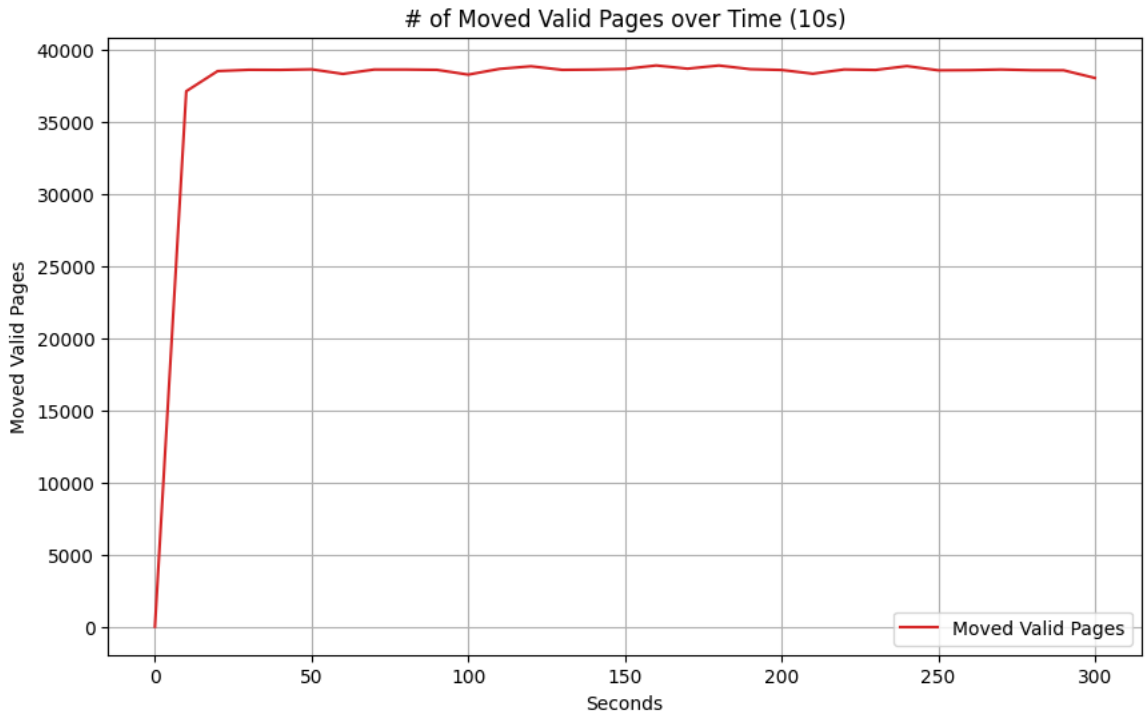
Both IOPS and throughput graphs show that the writing cliff still occurs.

Compared to the previous graph of sysbench thread=4, IOPS and throughput are observed to be higher.

5-3. sysbench(32hreads): # Erased blocks over time



5-4. sysbench(32threads): # valid pages moved over time



Compared to the previous graph of sysbench thread=4, number of erased blocks and number of moved valid pages are observed to be higher.

3. I/O Statistics Routine Implementation

- **Variables**

```c
typedef struct {
    double seconds;
    double throughput;
    uint64_t iops;
    uint64_t erased_block;
    uint64_t moved_valid_pages;
} stats;

uint64_t io_cnt = 0;
uint64_t erased_block = 0;
uint64_t moved_valid_pages = 0;
double throughput = 0.0;

static stats stats_buffer[BUFFER_SIZE];
static size_t stats_index = 0;
static timer_t timerid;
```

io_cnt: Store total number of I/O operations.

erased_block: Store count of erased blocks.

moved_valid_pages: Store count of moved valid pages.

throughput: Store read and write I/O throughput.

stats_buffer: Used to temporarily store performance statistics collected during the workload execution.

- **ssd_init()**

```c
void ssd_init(FemuCtrl *n)
{
    struct ssd *ssd = n->ssd;
    struct ssdparams *spp = &ssd->sp;

    ftl_assert(ssd);

    ssd_init_params(spp, n);

    /* initialize ssd internal layout architecture */
    ssd->ch = g_malloc0(sizeof(struct ssd_channel) * spp->nchs);
    for (int i = 0; i < spp->nchs; i++) {
        ssd_init_ch(&ssd->ch[i], spp);
    }

    /* initialize maptbl */
    ssd_init_maptbl(ssd);

    /* initialize rmap */
    ssd_init_rmap(ssd);

    /* initialize all the lines */
    ssd_init_lines(ssd);

    /* initialize write pointer, this is how we allocate new pages for writes */
    ssd_init_write_pointer(ssd);

    // initialize statistic buffer
    memset(stats_buffer, 0, sizeof(stats_buffer));

    qemu_thread_create(&ssd->ftl_thread, "FEMU-FTL-Thread", ftl_thread, n,
                       QEMU_THREAD_JOINABLE);
}
```

The stats_buffer is initialized in the ssd_init function to ensure it starts with a clean state before use.

• **ftl_thread()**

```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;
    init_timer();

    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);
                io_cnt++;
                throughput += (req->nlb * ssd->sp.secsz) / (1024.0 * 1024.0);
                break;
            case NVME_CMD_READ:
                lat = ssd_read(ssd, req);
                io_cnt++;
                throughput += (req->nlb * ssd->sp.secsz) / (1024.0 * 1024.0);
                break;
            case NVME_CMD_DSM:
                lat = 0;
                break;
            default:
                //ftl_err("FTL received unkown request type, ERROR\n");
                ;
            }
```

In the ftl_thread function, the ssd_write and ssd_read functions are called to handle write and read operations, respectively. After these function calls, the following actions occur:

io_cnt is incremented to count the total number of I/O operations.

throughput is updated to accumulate the amount of data processed in megabytes (MB).

• **init_timer()**

```c
void init_timer(void) {
    struct sigevent sev;
    struct itimerspec its;

    // Set up the timer to trigger a handler
    sev.sigev_notify = SIGEV_THREAD; // Notify via thread execution
    sev.sigev_notify_function = timer_handler; // Timer expiration handler
    sev.sigev_notify_attributes = NULL;
    sev.sigev_value.sival_ptr = NULL;

    if (timer_create(CLOCK_MONOTONIC, &sev, &timerid) == -1) {
        perror("timer_create failed");
        exit(EXIT_FAILURE);
    }

    // Create the timer with the specified settings
    // The timer starts in a state where it trigger the handler every 1 second
    its.it_value.tv_sec = 1;
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = 1;
    its.it_interval.tv_nsec = 0;

    if (timer_settime(timerid, 0, &its, NULL) == -1) {
        perror("timer_settime failed");
        exit(EXIT_FAILURE);
    }
}
```

Configures a timer to call the timer_handler function.

Starts the timer with an initial delay of 1 second and triggers the timer_handler() every 1 second thereafter.

• **timer_handler()**

```c
void timer_handler(union sigval sv) {
    // Tracks if workload is detected
    static bool workload_detected = false;
    static time_t start_time = 0; // Start time of the workload
    static int interval_count = 0; // Number of 1-second intervals processed

    // Detect workload when I/O count exceeds 3000 in 1 second
    if (!workload_detected && io_cnt > 3000) {
        workload_detected = true;
        printf("WorkLoad detected\n");
        start_time = time(NULL); // Record start time for logging

        // Record statistics in the buffer
        stats_buffer[stats_index].seconds = start_time;
        stats_buffer[stats_index].iops = io_cnt;
        stats_buffer[stats_index].throughput = throughput;
        stats_buffer[stats_index].moved_valid_pages = moved_valid_pages;
        stats_buffer[stats_index++].erased_block = erased_block;

        // Reset counters for the next interval
        io_cnt = 0;
        throughput = 0.0;
        moved_valid_pages = 0;
        erased_block = 0;
        return;
    }
    // If FIO workload is detected, log statistics every seconds.
    if (workload_detected) {
        ++interval_count;

        // Record statistics in the buffer
        stats_buffer[stats_index].seconds = difftime(time(NULL), st
        stats_buffer[stats_index].iops = io_cnt;
        stats_buffer[stats_index].throughput = throughput;
        stats_buffer[stats_index].moved_valid_pages = moved_valid_pages;
        stats_buffer[stats_index++].erased_block = erased_block;

        // Record statistics in the buffer
        io_cnt = 0;
        throughput = 0.0;
        moved_valid_pages = 0;
        erased_block = 0;

        // Stop logging after 310 intervals (310 seconds) because of sysbench prepare sequence
        if (interval_count >= 310) {
            printf("WorkLoad finished.\n");
            print_statistics(); // Log the recorded statistics
            timer_delete(timerid); // Delete the timer to stop further logging
        }
    }
}
```

```c
void print_statistics(void) {
    FILE *log_file = fopen("statistics.csv", "a");
    fprintf(log_file, "seconds,iops,throughput,erased_block,moved_valid_pages\n");
    for (int i = 0; i < stats_index; i++) {
        fprintf(log_file, "%.0f, %lu, %.2f, %lu, %lu\n",
            stats_buffer[i].seconds,
            stats_buffer[i].iops,
            stats_buffer[i].throughput,
            stats_buffer[i].erased_block,
            stats_buffer[i].moved_valid_pages);
    }
    fflush(log_file);
    fclose(log_file);
}
```

The timer_handler function logs statistics every second when I/O exceeds 3000 operations in a second: It Starts recording when io_cnt > 3000 and tracks the start time.

Every second, it stores the elapsed time, IOPS, throughput, erased blocks, and moved valid pages into stats_buffer.

After logging, the counters are reset for the next interval.

It stops logging after 310 seconds (310 intervals), prints the collected statistics, and deletes the timer. This includes an additional 10 seconds due to the prepare stage of sysbench, extending the logging duration from 300 seconds to 310 seconds.

• **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);
            erased_block++;
            if (spp->enable_gc_delay) {
```

In the do_gc function, the erased_block counter is incremented after the clean_one_block function is called and the block is marked as free.

• **clean_one_block**

```
/* 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);
            moved_valid_pages++;
            cnt++;
        }
    }

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

clean_one_block function, the moved_valid_pages counter is incremented after garbage collection write page process.