Internal Mechanisms and Performance Evaluation of FEMU SSD Emulator

Jihwan Moon

System Programming

Wonil Choi

October 30, 2024

1 Introduction

In recent years, Solid State Drives (SSDs) have emerged as a critical component in the storage device sector, driven by the need for faster and more efficient storage solutions. Extensive research is being conducted to enhance and optimize SSD performance, highlighting the growing need for SSD emulators that support such studies.

FEMU is an NVMe SSD emulator based on the open-source *QEMU* platform, capable of simulating various SSD operations. [1]

In this report, a source code analysis of FEMU's internal mechanisms is performed, focusing on the read, write, and garbage collection (GC) operations in Blackbox mode (BBSSD).

Custom I/O routines were implemented to collect necessary metrics for performance measurement, and performance evaluations were conducted under various workloads using *fio* and *sysbench*. [2, 3]

Through this process, the impact of GC on SSD performance is quantitatively analyzed, providing a deeper understanding of FEMU's operational principles.

2 Methodology

2.1 Source Code Analysis Method

FEMU is a *QEMU*-based NVMe SSD emulator that supports various types of SSD emulation. This report specifically analyzes the read, write, and GC mechanisms in Blackbox mode by examining the source code in bbssd/ftl.c and nvme-io.c located under the hw/femu directory.

2.2 Benchmark Tools and Experimental Environment Setup

```
# 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,

# Latency in nanoseconds
pg_rd_lat=40000 # page read latency
pg_wr_lat=200000 # page write latency
blk_er_lat=2000000 # block erase latency
ch_xfer_lat=0 # channel transfer time, ignored for now

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

Figure 2.1: FEMU Blackbox SSD configuration

Performance measurements were conducted using the I/O benchmarking tools *fio* and *sysbench*. A total of five benchmarks were performed, each executed for 300 seconds across four types of metrics. FEMU was executed in an *Ubuntu 24.04* on *WSL 2*, with the SSD configured to 4GB (3GB, 25% over-provisioning). Metric collection was achieved by modifying the FEMU source code to add routines for measuring metrics and printing the results.

2.3 Data Analysis Techniques

The collected metric data was visualized using the *matplotlib* module in *Python* to understand how the read, write, and GC mechanisms work, as well as the impact of GC on SSD performance.

3 Code Analysis of FEMU SSD Emulator

3.1 Structure

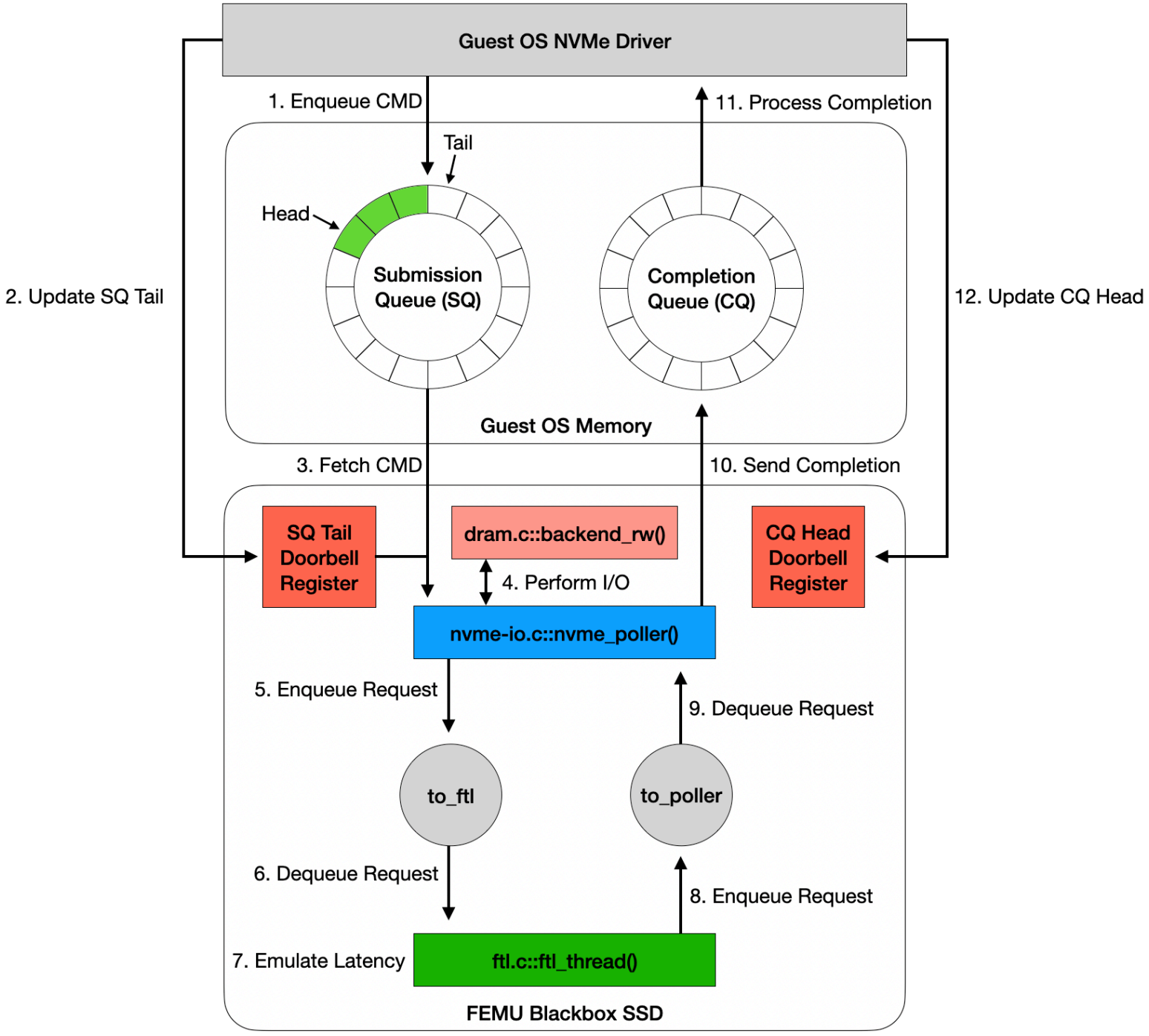

Figure 3.1: Flow of command execution in FEMU Blackbox SSD

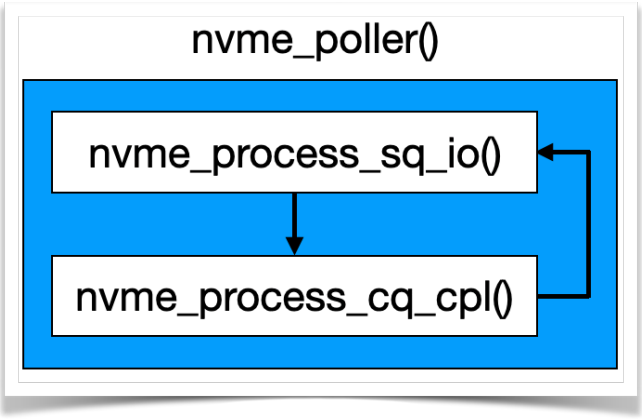

Figure 3.2: Brief workflow of nvme_poller

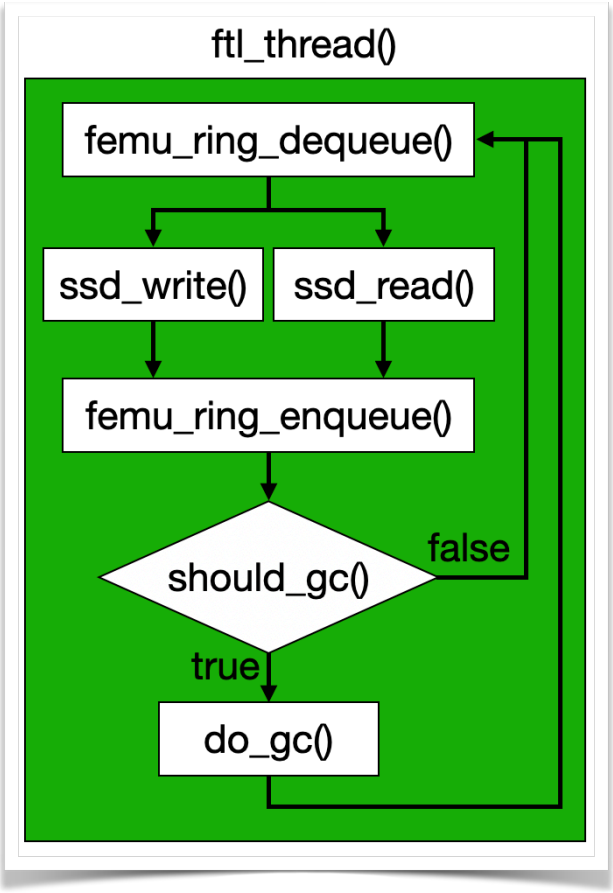

Figure 3.3: Brief workflow of ftl_thread

In the FEMU Blackbox SSD, when an I/O request generated by the guest OS is sent to the NVMe driver, it is converted into an NVMe command according to the NVMe specification and then inserted into the submission queue (SQ). Following this, the SQ tail doorbell register is updated. Next, nvme_poller() reads the SQ tail doorbell register, dequeueing a command from the SQ to perform I/O operations via DRAM by calling backend_rw(). Afterward, the request is placed into the to_ftl queue.

From this point, processing is handled in ftl_thread(), where a request is dequeued from to_ftl, its latency is simulated, and the completed request is placed into the to_poller queue.

The process then transitions back to nvme_poller(), which dequeues a request from to_poller, puts it into the completion queue (CQ), and notifies the NVMe driver that the I/O operation has been completed.

The NVMe driver retrieves a completed request from the CQ to determine whether it was successfully completed or if an error occurred. It updates the CQ head doorbell register and returns the result of the completed request back to the application. This flow repeats throughout the lifecycle of FEMU.

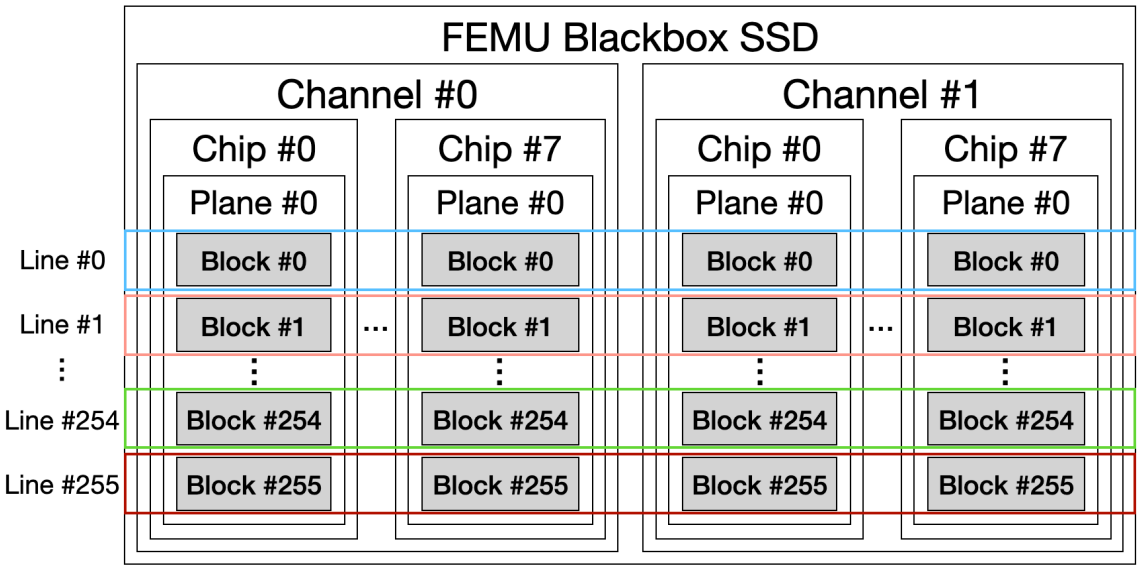

Figure 3.4: Architecture of FEMU Blackbox SSD used in this experiment

The structure of the FEMU used in this experiment is as shown above. According to the configuration, there are 256 pages per block, 256 blocks per plane, 1 plane per chip, 8 chips per channel, and a total of 2 channels. With multiple channels and chips, both channel-level and chip-level parallelism are supported. Additionally, FEMU groups blocks with the same index into a single logical unit called a line to efficiently process I/O and GC.

3.2 Read/Write Operation

```c
void *nvme_poller(void *arg)
{
    FemuCtrl *n = ((NvmePollerThreadArgument *)arg)->n;
    int index = ((NvmePollerThreadArgument *)arg)->index;
    int i;
    setup_timer(); // Sets up the timer

    switch (n->multipoller_enabled) // Default is 0
    {
    case 1: // This path is not taken
        while (1)…
        break;
    default: // multipoller disabled
        while (1)
        {
            if ((!n->dataplane_started))
            {
                usleep(1000);
                continue;
            }

            for (i = 1; i <= n->nr_io_queues; i++) // Iterates over the I/O queues
            {
                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); // Processes the SQ
                }
            }
            nvme_process_cq_cpl(n, index); // Processes the CQ
        }
        break;
    }

    return NULL;
}
```

Figure 3.5: Main logic of nvme_poller() in nvme-io.c

The function above serves as an interface connecting the SQ, CQ, and FTL (File Translation Layer).

It runs in an infinite loop, processing both the SQ and CQ.

```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);     // Updates the SQ tail from the doorbell register
    while (!(nvme_sq_empty(sq))) // Repeats while the SQ is not empty
    {
        if (sq->phys_contig) // If the SQ is physically contiguous (default is 1)
        {
            addr = sq->dma_addr + sq->head * n->sqe_size;                          // Gets the address of the head
            nvme_copy_cmd(&cmd, (void *)&(((NvmeCmd *)sq->dma_addr_hva)[sq->head])); // Reads a command from the SQ
        }
        else // This path is not taken⋯
        nvme_inc_sq_head(sq); // Increments the SQ head for the next command

        req = QTAILQ_FIRST(&sq->req_list);        // Gets a request
        QTAILQ_REMOVE(&sq->req_list, req, entry); // Removes the request from the list
        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) // This path is not taken⋯

        status = nvme_io_cmd(n, &cmd, req); // Performs an I/O operation
        if (1 && status == NVME_SUCCESS)
        {
            req->status = status;

            int rc = femu_ring_enqueue(n->to_ftl[index_poller], (void *)&req, 1); // Enqueues the request to to_ftl
            if (rc != 1)
            {
                femu_err("enqueue failed, ret=%d\n", rc);
            }
        }
        else if (status == NVME_SUCCESS) // This path is not taken⋯
        else
        {
            femu_err("Error IO processed!\n");
        }

        processed++;
    }

    nvme_update_sq_eventidx(sq); // Updates the SQ event index
    sq->completed += processed;
}
```

Figure 3.6: Main logic of nvme_process_sq_io() in nvme-io.c

In the function above, the SQ tail doorbell register is read to update the SQ tail. A loop runs while the SQ is not empty, fetching a command from the SQ. Afterward, the SQ head is incremented, and the request is retrieved from the SQ's req_list and set before calling nvme_io_cmd() to perform the I/O operation. nvme_io_cmd() internally calls nvme_rw(), which calls backend_rw() in dram.c to handle the actual I/O. After that, the request is placed into the to_ftl queue, and once the loop ends, the SQ event index is updated.

```c
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;

    ++io_count;              // Increments the I/O count
    throughput += data_size; // Accumulates throughput

    req->is_write = (rw->opcode == NVME_CMD_WRITE) ? 1 : 0; // Whether the request is a write

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

    if (nvme_map_prp(&req->qsg, &req->iov, prp1, prp2, data_size, n)) // Maps the PRP to the request
    {
        nvme_set_error_page(n, req->sq->sqid, cmd->cid, NVME_INVALID_FIELD,
                            offsetof(NvmeRwCmd, prp1), 0, ns->id); // Sets an error page
        return NVME_INVALID_FIELD | NVME_DNR;
    }

    assert((nlb << data_shift) == req->qsg.size); // Asserts that the size of the request is correct

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

    ret = backend_rw(n->mbe, &req->qsg, &data_offset, req->is_write); // Reads/writes data to/from DRAM
    if (!ret)
    {
        return NVME_SUCCESS;
    }

    return NVME_DNR;
}
```

Figure 3.7: Main logic of nvme_rw() in nvme-io.c

In the code above, backend_rw() is called to perform reading or writing data from/to the DRAM.

```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);
    }

    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    while (1)
    {
        for (i = 1; i <= n->nr_pollers; i++) // Iterates over the pollers (default nr_pollers is 1)
        {
            if (!ssd->to_ftl[i] || !femu_ring_count(ssd->to_ftl[i])) // If the queue does not exist or is empty
                continue;

            rc = femu_ring_dequeue(ssd->to_ftl[i], (void *)&req, 1); // Dequeues a request from the queue
            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); // Simulates the write operation
                break;
            case NVME_CMD_READ:
                lat = ssd_read(ssd, req); // Simulates the read operation
                break;
            case NVME_CMD_DSM:
                lat = 0;
                break;
            default:
                // ftl_err("FTL received unkown request type, ERROR\n");
                ;
            }

            req->reqlat = lat;       // Sets the latency of the request
            req->expire_time += lat; // Updates the expiration time of the request

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

            if (should_gc(ssd)) // If GC is required
            {
                do_gc(ssd, false); // Performs GC
            }
        }
    }

    return NULL;
}
```

Figure 3.8: Main logic of ftl_thread() in ftl.c

The function above runs in an infinite loop, fetching requests from to_ftl and simulating read/write operations based on the opcode. It then sets the latency and expiration time for each request, places it into to_poller, and performs GC if necessary.

```c
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++) // Iterates over the LPNs
    {
        ppa = get_maptbl_ent(ssd, lpn);                 // Gets the PPA using the LPN from the mapping table
        if (!mapped_ppa(&ppa) || !valid_ppa(ssd, &ppa)) // If the PPA is not mapped or invalid
        {
            // 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); // Calculates the latency of the NAND command
        maxlat = (sublat > maxlat) ? sublat : maxlat; // Updates maxlat
    }

    return maxlat; // Returns the maximum latency
}
```

Figure 3.9: Main logic of ssd_read() in ftl.c

In the code above, it iterates over the pages to be read, retrieves the PPA for each LPN from the mapping table, and calculates the NAND command latency. It returns the max latency upon completing the iteration.

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

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

    while (should_gc_high(ssd)) // Repeats while high threshold GC is required
    {
        r = do_gc(ssd, true); // Performs GC
        if (r == -1)          // If there is no victim line
            break;
    }

    for (lpn = start_lpn; lpn <= end_lpn; lpn++) // Iterates over the LPNs
    {
        ppa = get_maptbl_ent(ssd, lpn); // Gets the PPA using the LPN from the mapping table
        if (mapped_ppa(&ppa))           // If the PPA is mapped
        {
            /* update old page information first */
            mark_page_invalid(ssd, &ppa);         // Marks the page as invalid
            set_rmap_ent(ssd, INVALID_LPN, &ppa); // Removes the mapping of the PPA from the LPN in the reverse mapping table
        }

        /* new write */
        ppa = get_new_page(ssd); // Gets a new page
        /* update maptbl */
        set_maptbl_ent(ssd, lpn, &ppa); // Sets LPN->PPA mapping in the mapping table
        /* update rmap */
        set_rmap_ent(ssd, lpn, &ppa); // Sets PPA->LPN mapping in the reverse mapping table

        mark_page_valid(ssd, &ppa); // Marks the page as valid

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

        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); // Calculates the latency of the NAND command
        maxlat = (curlat > maxlat) ? curlat : maxlat; // Updates maxlat
    }

    return maxlat; // Returns the maximum latency
}
```

Figure 3.10: Main logic of ssd_write() in ftl.c

In the code above, high-threshold GC is performed while required before beginning the write process. The function iterates over the pages to be written, retrieving the PPA for each LPN from the mapping table. If a page is already mapped, the page is invalidated. A new page is then retrieved, its PPA and LPN are mapped, marked as valid, and the write pointer is advanced. The latency for each command is calculated, and the maximum latency is returned upon completing the iteration.

```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;            // Sets the start time
        lun->next_lun_avail_time = nand_stime + spp->pg_rd_lat; // Sets the next available time for the LUN
        lat = lun->next_lun_avail_time - cmd_stime;             // Calculates the latency
#if 0 // This path is not taken
#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; // Sets the start time
        if (ncmd->type == USER_IO)
        {
            lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat; // Sets the next available time for the LUN
        }
        else
        {
            lun->next_lun_avail_time = nand_stime + spp->pg_wr_lat; // Sets the next available time for the LUN
        }
        lat = lun->next_lun_avail_time - cmd_stime; // Calculates the latency

#if 0 // This path is not taken
#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;             // Sets the start time
        lun->next_lun_avail_time = nand_stime + spp->blk_er_lat; // Sets the next available time for the LUN
        lat = lun->next_lun_avail_time - cmd_stime;              // Calculates the latency
        break;

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

    return lat; // Returns the latency
}
```

Figure 3.11: Main logic of ssd_advance_status() in ftl.c

In the code above, the LUN's next available time and the latency are calculated based on whether it is currently in use.

```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); // Checks if the channel index is within the valid range
    wpp->ch++;                      // Increments the channel index
    if (wpp->ch == spp->nchs)       // If the channel index is out of range
    {
        wpp->ch = 0;                            // Resets the channel index
        check_addr(wpp->lun, spp->luns_per_ch); // Checks if the LUN index is within the valid range
        wpp->lun++;                             // Increments the LUN index
        /* in this case, we should go to next lun */
        if (wpp->lun == spp->luns_per_ch) // If the LUN index is out of range
        {
            wpp->lun = 0; // Resets the LUN index
            /* go to next page in the block */
            check_addr(wpp->pg, spp->pgs_per_blk); // Checks if the page index is within the valid range
            wpp->pg++;                             // Increments the page index
            if (wpp->pg == spp->pgs_per_blk)       // If the page index is out of range
            {
                wpp->pg = 0; // Resets the page index
                /* move current line to {victim,full} line list */
                if (wpp->curline->vpc == spp->pgs_per_line) // If all pages in the current line are valid
                {
                    /* 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); // Inserts the current line at the tail of full_line_list
                    lm->full_line_cnt++;                                          // Increments the full line count
                }
                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); // Inserts the current line into victim_line_pq
                    lm->victim_line_cnt++;                           // Increments the victim line count
                }
                /* current line is used up, pick another empty line */
                check_addr(wpp->blk, spp->blks_per_pl); // Checks if the block index is within the valid range
                wpp->curline = NULL;
                wpp->curline = get_next_free_line(ssd); // Sets the current line to the next free line
                if (!wpp->curline)
                {
                    /* TODO */
                    abort();
                }
                wpp->blk = wpp->curline->id;            // Sets the block index to the ID of the current line
                check_addr(wpp->blk, spp->blks_per_pl); // Checks if the block index is within the valid range
                /* 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);
            }
        }
    }
}
```

Figure 3.12: Main logic of ssd_advance_write_pointer() in ftl.c

The code above determines the next position for writing. When this function is called, it first changes the channel most frequently, followed by the LUN, and then the page. For instance, if there are 2 channels, 2 LUNs, and 2 pages, with the i-th channel, the j-th LUN, and the k-th page represented by 'ijk', these indices change as follows with each function call: 000, 100, 010, 110, 001, 101, 011, 111. If all pages in the current line are filled, the function checks whether all pages in that line are valid. If they are, the current line is added to full_line_list; if not, it is inserted into victim_line_pq. After that, the next free line is retrieved and designated as the current line.

```c
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)) // Default is true
    {
        rp = n->to_poller[index_poller];
    }

    while (femu_ring_count(rp)) // Repeats while to_poller is not empty
    {
        req = NULL;
        rc = femu_ring_dequeue(rp, (void *)&req, 1); // Dequeues a request from to_poller
        if (rc != 1)
        {
            femu_err("dequeue from to_poller request failed\n");
        }
        assert(req);

        pqueue_insert(pq, req); // Inserts the request into the priority queue, with priority determined by expiration time
    }
    while ((req = pqueue_peek(pq))) // Repeats while the priority queue is not empty
    {
        now = qemu_clock_get_ns(QEMU_CLOCK_REALTIME);
        if (now < req->expire_time) // If expire_time has not passed yet
        {
            break;
        }

        cq = n->cq[req->sq->sqid];
        if (!cq->is_active)
            continue;
        nvme_post_cqe(cq, req);                             // Posts a request to the CQ
        QTAILQ_INSERT_TAIL(&req->sq->req_list, req, entry); // Inserts the request into req_list
        pqueue_pop(pq);                                     // Pops the processed request
        processed++;
        n->nr_tt_ios++;

        if (now - req->expire_time >= 20000) // If the request is delayed by more than 20us
        {
            n->nr_tt_late_ios++;
            if (n->print_log) // This path is not taken…
        }
        n->should_isr[req->sq->sqid] = true; // Sets the flag to notify the I/O completion
    }

    if (processed == 0)
        return;

    switch (n->multipoller_enabled) // Default is 0
    {
    case 1: // This path is not taken
        nvme_isr_notify_io(n->cq[index_poller]);
        break;
    default:                                    // multipoller disabled
        for (i = 1; i <= n->nr_io_queues; i++) // Iterates over the I/O queues
        {
            if (n->should_isr[i])
            {
                nvme_isr_notify_io(n->cq[i]); // Notifies the I/O completion
                n->should_isr[i] = false;
            }
        }
        break;
    }
}
```

Figure 3.13: Main logic of nvme_process_cq_cpl() in nvme-io.c

In the function above, a loop runs while to_poller is not empty, dequeueing requests and adding them to the priority queue. After that, it peeks the request with the earliest expiration time; if the time has not yet passed, the loop breaks. If there are completed requests, they are added to the CQ, and the requests are removed from the priority queue. Once all requests in the priority queue have been processed, a loop iterates over the I/O queues to notify the completion.

3.2 Garbage Collection Mechanisms

In FEMU, before and after performing a write operation, it checks whether GC is needed and performs GC if required.

```
static inline bool should_gc(struct ssd *ssd)
{
    // Checks if the free line count is less than or equal to the GC threshold(default is 25%)
    return (ssd->lm.free_line_cnt <= ssd->sp.gc_thres_lines);
}
```

Figure 3.14: Main logic of should_gc() in ftl.c

```
static inline bool should_gc_high(struct ssd *ssd)
{
    // Checks if the free line count is less than or equal to the high GC threshold(default is 5%)
    return (ssd->lm.free_line_cnt <= ssd->sp.gc_thres_lines_high);
}
```

Figure 3.15: Main logic of should_gc_high() in ftl.c

Whether GC is required is determined by the two functions above. These return true if the number of free lines in the SSD is below the GC threshold.

```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); // Selects the victim line
    if (!victim_line)
    {
        return -1;
    }

    ppa.g.blk = victim_line->id; // Sets the block index to the id of the victim line
    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++) // Iterates over the channels
    {
        for (lun = 0; lun < spp->luns_per_ch; lun++) // Iterates over the LUNs
        {
            // Sets the channel, LUN, and plane indices
            ppa.g.ch = ch;
            ppa.g.lun = lun;
            ppa.g.pl = 0;
            lunp = get_lun(ssd, &ppa);  // Returns the pointer to the LUN
            clean_one_block(ssd, &ppa); // Cleans the specified block
            ++erased_blocks;            // Increments the erased block count
            mark_block_free(ssd, &ppa); // Marks the block as free

            if (spp->enable_gc_delay) // If GC delay is enabled (default is true)
            {
                struct nand_cmd gce;
                gce.type = GC_IO;
                gce.cmd = NAND_ERASE;
                gce.stime = 0;
                ssd_advance_status(ssd, &ppa, &gce); // Updates the status of the SSD
            }

            lunp->gc_endtime = lunp->next_lun_avail_time; // Updates gc_endtime of the LUN
        }
    }

    /* update line status */
    mark_line_free(ssd, &ppa); // Marks the line as free

    return 0;
}
```

Figure 3.16: Main logic of do_gc() in ftl.c

In the function above, a victim line is selected and all channels along with their respective LUNs are iterated over to clean blocks and mark them as free. And gc_endtime of the LUN is updated. Once all loops are complete, the victim line is marked as free.

```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); // Gets the line with the lowest valid page count from the priority queue
    if (!victim_line)
    {
        return NULL;
    }

    // If GC is not forced and the invalid page count in the victim line is less than 1/8 of the total pages in the line
    if (!force && victim_line->ipc < ssd->sp.pgs_per_line / 8)
    {
        return NULL;
    }

    pqueue_pop(lm->victim_line_pq); // Pops the victim line from the queue
    victim_line->pos = 0;           // Resets pos of the victim line used in victim_line_pq
    lm->victim_line_cnt--;

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

Figure 3.17: Main logic of select_victim_line() in ftl.c

In the code above, the line with the lowest valid page count is retrieved from the priority queue. If force is false and the invalid page count is less than one-eighth of the line's total pages, it returns NULL.

Next, the victim line is popped from the priority queue and returned.

```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++) // Iterates over the pages in the block
    {
        ppa->g.pg = pg;
        pg_iter = get_pg(ssd, ppa); // Gets the pointer to the page
        /* there shouldn't be any free page in victim blocks */
        ftl_assert(pg_iter->status != PG_FREE); // Asserts that the page is either valid or invalid
        if (pg_iter->status == PG_VALID)        // If the page is valid, it needs to be moved
        {
            ++pages_moved;          // Increments the valid page count moved
            gc_read_page(ssd, ppa); // Simulates the read operation
            /* delay the maptbl update until "write" happens */
            gc_write_page(ssd, ppa); // Simulates the write operation
            cnt++;
        }
    }

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

Figure 3.18: Main logic of clean_one_block() in ftl.c

In the code above, all pages in the block are iterated and each valid page is moved to another block.

```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) // If GC delay is enabled (default is true)
    {
        struct nand_cmd gcr;
        gcr.type = GC_IO;
        gcr.cmd = NAND_READ;
        gcr.stime = 0;
        ssd_advance_status(ssd, ppa, &gcr); // Updates the status of the SSD
    }
}
```

Figure 3.19: Main logic of gc_read_page() in ftl.c

In the code above, ssd_advance_status() is called, if enabled, to simulate the GC delay. It delays the next available time of the LUN by the page read latency.

```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); // Gets the LPN using the PPA from the reverse mapping table

    ftl_assert(valid_lpn(ssd, lpn));
    new_ppa = get_new_page(ssd); // Gets a new page
    /* update maptbl */
    set_maptbl_ent(ssd, lpn, &new_ppa); // Sets LPN->PPA mapping in the mapping table
    /* update rmap */
    set_rmap_ent(ssd, lpn, &new_ppa); // Sets PPA->LPN mapping in the reverse mapping table

    mark_page_valid(ssd, &new_ppa); // Marks the page as valid

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

    if (ssd->sp.enable_gc_delay) // If GC delay is enabled (default is true)
    {
        struct nand_cmd gcw;
        gcw.type = GC_IO;
        gcw.cmd = NAND_WRITE;
        gcw.stime = 0;
        ssd_advance_status(ssd, &new_ppa, &gcw); // Updates the status of the SSD
    }

    /* advance per-ch gc_endtime as well */
#if 0 // This path is not taken⋯
#endif

    new_lun = get_lun(ssd, &new_ppa);                   // Returns the pointer to the LUN
    new_lun->gc_endtime = new_lun->next_lun_avail_time; // Updates gc_endtime of the LUN

    return 0;
}
```

Figure 3.20: Main logic of gc_write_page() in ftl.c

In the code above, the LPN is retrieved from the reverse mapping table using the PPA of the page to be moved. Then, a new page is retrieved, mapped to the LPN, and the write pointer is advanced. After that, the GC delay is emulated, and the gc_endtime for the LUN of the new page is set.

4 Implementation of Custom I/O Routines

To collect the metrics necessary for the experiment by running benchmarks, modifications to the FEMU source code were required. Routines for collecting and printing out metrics were added to nvme-io.c and ftl.c.

```c
static void setup_timer(void)
{
    struct sigaction sa;
    struct itimerval timer;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &print_statistics; // Sets the signal handler
    sigaction(SIGALRM, &sa, NULL);

    timer.it_value.tv_sec = 1; // Sets the timer to 1 second
    timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 1; // Sets the interval to 1 second
    timer.it_interval.tv_usec = 0;

    setitimer(ITIMER_REAL, &timer, NULL); // Sets the timer
}
```

Figure 4.1: Implementation of setup_timer() in nvme-io.c

This is the code that sets up a timer to call a function that outputs metrics every second.

```c
static void print_statistics(int signum)
{
    static int seconds_counter = 0;
    if (io_count == 0)
        return;
    time_t now;
    struct tm *timeinfo;
    char time_str[20];

    time(&now);
    timeinfo = localtime(&now);

    strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", timeinfo);
    double throughput_mb = (double)throughput / (1024 * 1024); // Converts throughput to MB/s

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

    if (++seconds_counter >= 10)
    {
        femu_log("[%s] Erased blocks: %lu, Valid pages moved: %lu\n", time_str, erased_blocks, pages_moved);
        erased_blocks = 0;
        seconds_counter = 0;
        pages_moved = 0;
    }

    io_count = 0;
    throughput = 0;
}
```

Figure 4.2: Implementation of print_statistics() in nvme-io.c

Using the function above, IOPS and throughput are printed every second, while the number of erased blocks and valid pages moved is printed every 10 seconds.

```
void *nvme_poller(void *arg)
{
    FemuCtrl *n = ((NvmePollerThreadArgument *)arg)->n;
    int index = ((NvmePollerThreadArgument *)arg)->index;
    int i;
    setup_timer(); // Sets up the timer
```

Figure 4.3: Routine for setting up the timer in nvme-io.c

Setting up the timer is done in the code above.

```
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;

    ++io_count;              // Increments the I/O count
    throughput += data_size; // Accumulates throughput

    req->is_write = (rw->opcode == NVME_CMD_WRITE) ? 1 : 0; // Whether the request is a write
```

Figure 4.4: Routine for accumulating IOPS and throughput in nvme-io.c

I/O count and throughput are accumulated in the code above.

```
static int do_gc(struct ssd *ssd, bool force)
    /* copy back valid data */
    for (ch = 0; ch < spp->nchs; ch++) // Iterates over the channels
    {
        for (lun = 0; lun < spp->luns_per_ch; lun++) // Iterates over the LUNs
        {
            // Sets the channel, LUN, and plane indices
            ppa.g.ch = ch;
            ppa.g.lun = lun;
            ppa.g.pl = 0;
            lunp = get_lun(ssd, &ppa);  // Returns the pointer to the LUN
            clean_one_block(ssd, &ppa); // Cleans the specified block
            ++erased_blocks;            // Increments the erased block count
            mark_block_free(ssd, &ppa); // Marks the block as free
```

Figure 4.5: Routine for accumulating # of erased blocks in ftl.c

# of erased blocks is accumulated in the code above.

```
/* 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++) // Iterates over the pages in the block
    {
        ppa->g.pg = pg;
        pg_iter = get_pg(ssd, ppa); // Gets the pointer to the page
        /* there shouldn't be any free page in victim blocks */
        ftl_assert(pg_iter->status != PG_FREE); // Asserts that the page is either valid or invalid
        if (pg_iter->status == PG_VALID)        // If the page is valid, it needs to be moved
        {
            ++pages_moved;          // Increments the valid page count moved
            gc_read_page(ssd, ppa); // Simulates the read operation
            /* delay the maptbl update until "write" happens */
            gc_write_page(ssd, ppa); // Simulates the write operation
            cnt++;
        }
    }

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

Figure 4.6: Routine for accumulating # of valid pages moved in ftl.c

# of valid pages moved is accumulated in the code above.

5 Performance Evaluation

To conduct performance evaluation, FEMU was modified with the custom routines mentioned above and rebuilt. Then, *fio* and *sysbench* were used to run five workloads across four metrics.
The details are as follows.

Metrics: IOPS, Throughput, # of erased blocks, # of valid pages moved
Workloads: 3 *fio* workloads (read 100%, read 50%/write 50%, write 100%), 2 *sysbench* workloads (4 threads, 32 threads)

Command for Workload (i):

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

Command for Workload (ii):

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

Command for Workload (iii):

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

Command for Workload (iv):

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

Command for Workload (v):

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

• Workload (i) - fio read 100%

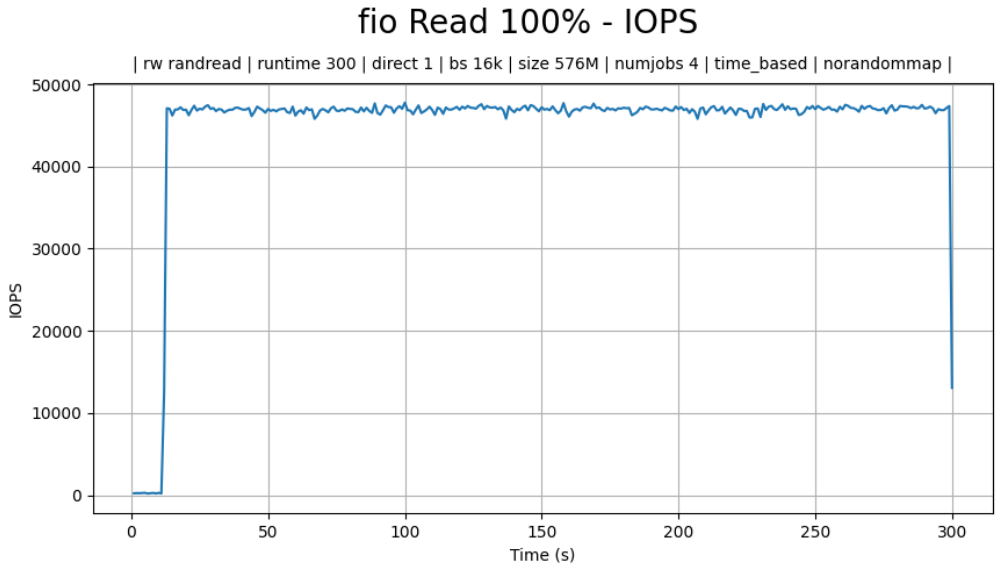

Figure 5.1: IOPS results for *fio* (read 100% workload)

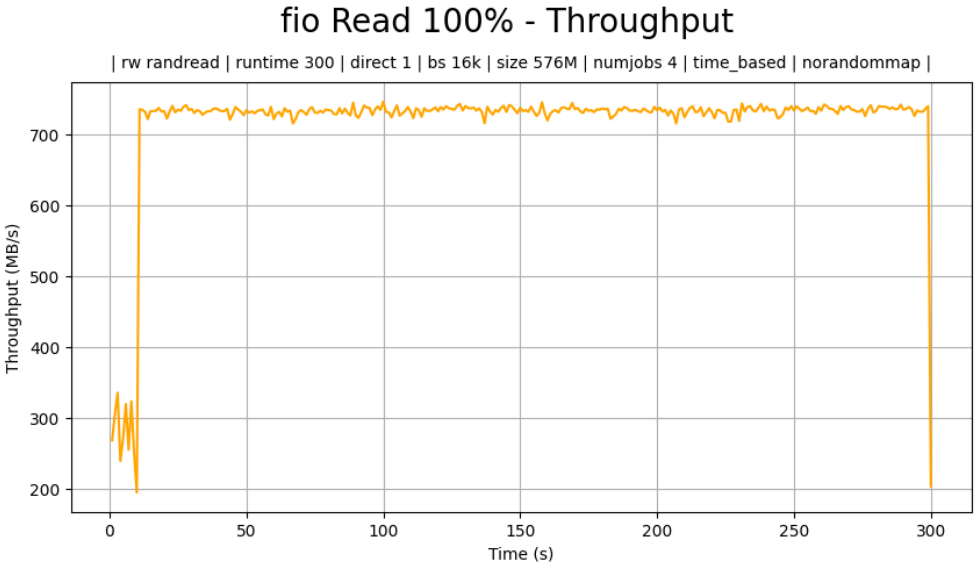

Figure 5.2: Throughput results for *fio* (read 100% workload)

Since this is a read 100% workload, write cliffs are not observed in IOPS and throughput, and both graphs show consistent performance levels without degradation.

• Workload (i) - fio read 100%

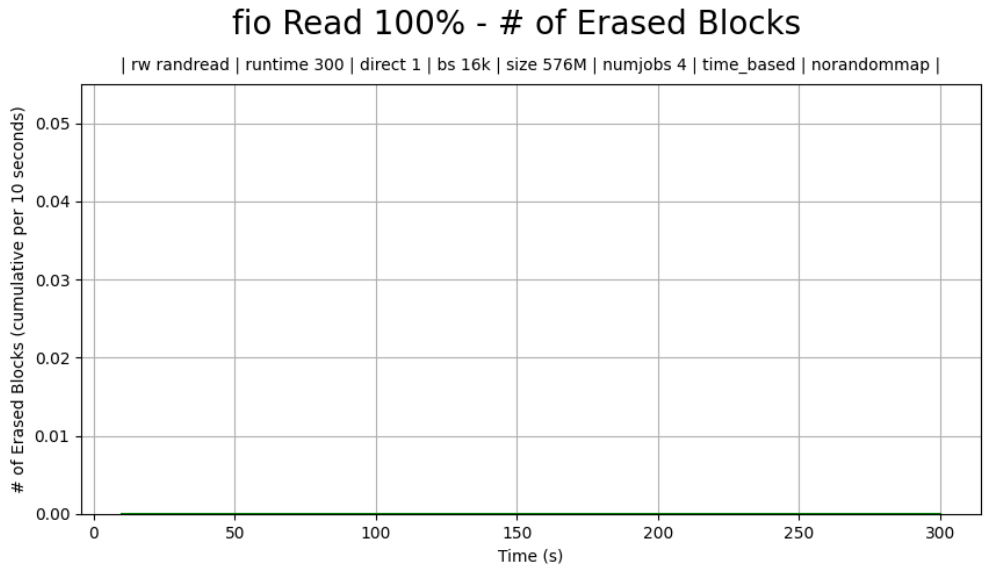

Figure 5.3: # of erased blocks results for *fio* (read 100% workload)

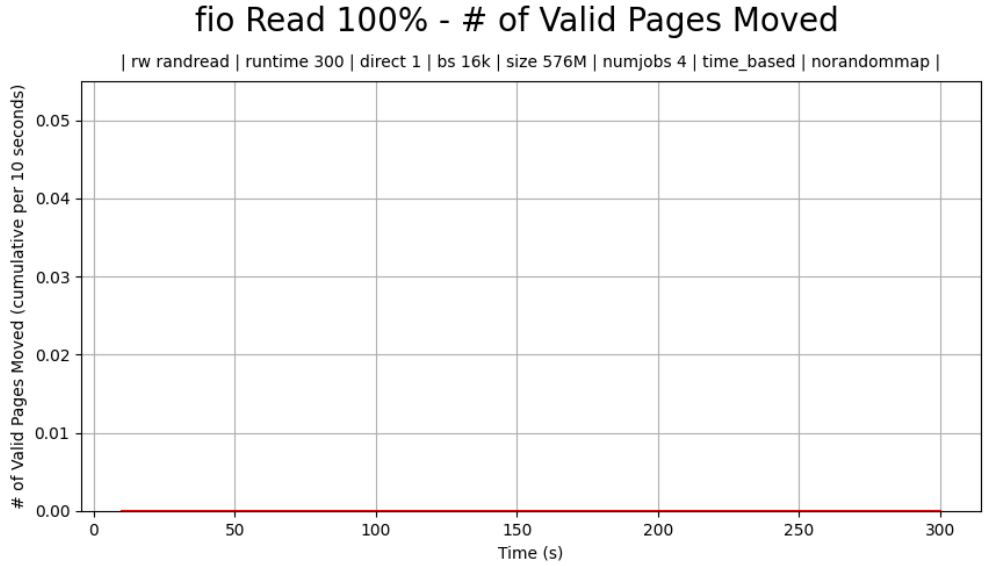

Figure 5.4: # of valid pages moved results for *fio* (read 100% workload)

Both graphs above show zero values throughout the test duration, as these metrics are related to GC that occurs during writes, which does not happen with a read 100% workload.

• Workload (ii) - fio read 50%/write 50%

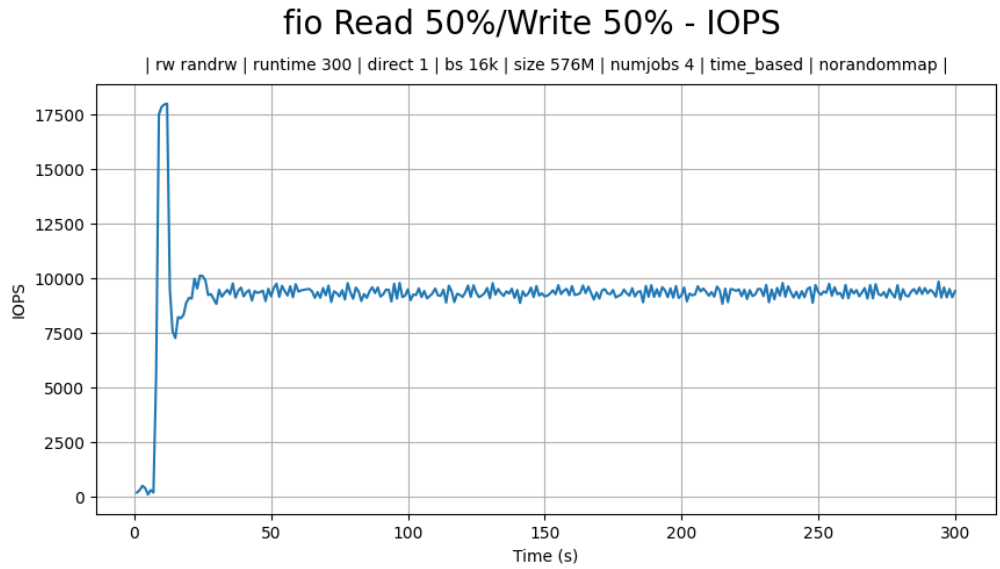

Figure 5.5: IOPS results for *fio* (read 50%/write 50% workload)

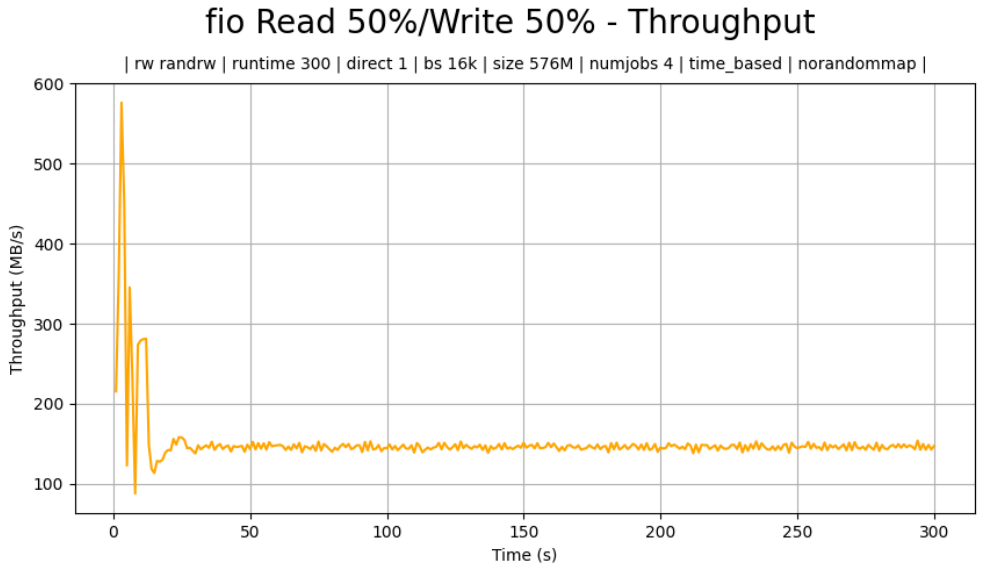

Figure 5.6: Throughput results for *fio* (read 50%/write 50% workload)

Based on the graphs above, it can be observed that both IOPS and throughput reach a peak, followed by a sharp decline, and then stabilize at a plateau, indicating the occurrence of a write cliff.

This phenomenon occurs because continuous writes reduce the number of free lines in the SSD below a certain threshold, triggering GC and temporarily halting ongoing I/O operations to perform the GC.

• Workload (ii) - fio read 50%/write 50%

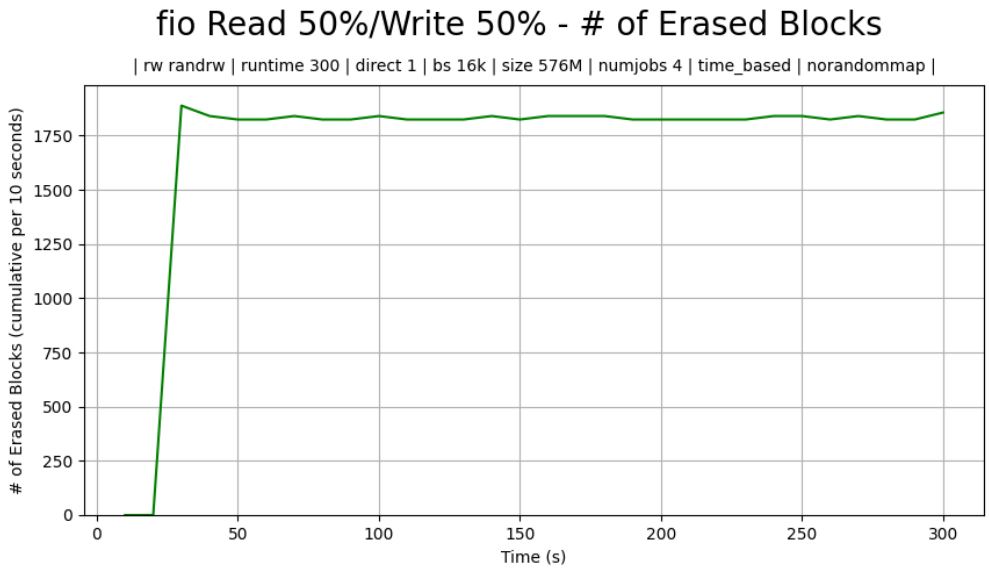

Figure 5.7: # of erased blocks results for *fio* (read 50%/write 50% workload)

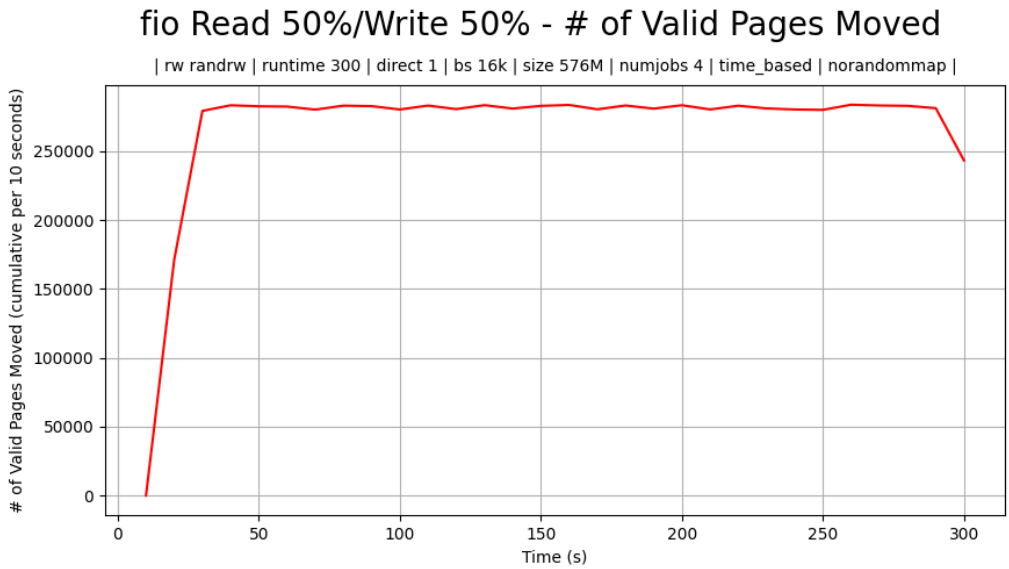

Figure 5.8: # of valid pages moved results for *fio* (read 50%/write 50% workload)

Based on the graphs above, both metrics show a sharp increase shortly after the benchmark begins, followed by a stable plateau.

This occurs because continuous writes reduce the number of free lines below the GC threshold, invoking GC.

• Workload (iii) - fio write 100%

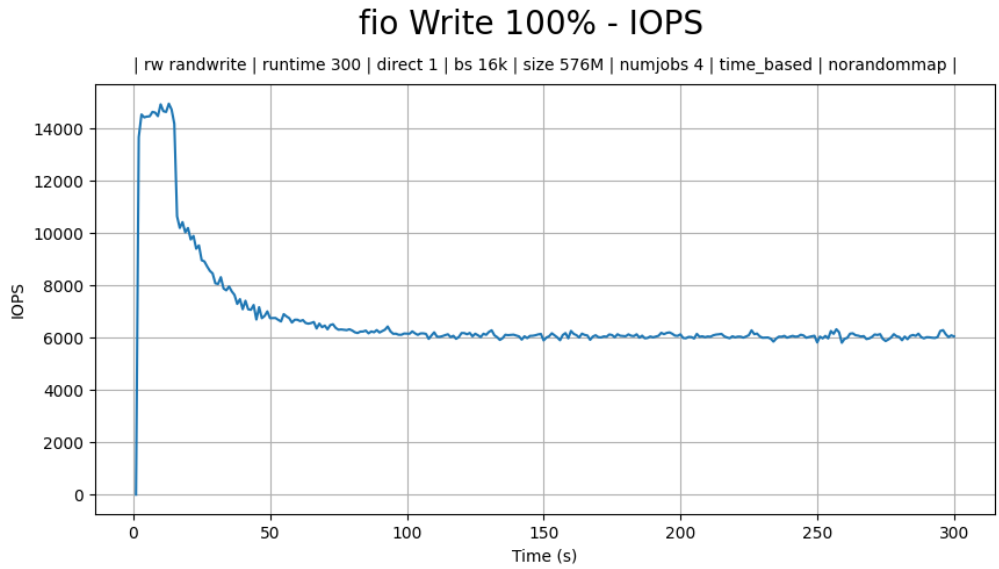

Figure 5.9: IOPS results for *fio* (write 100% workload)

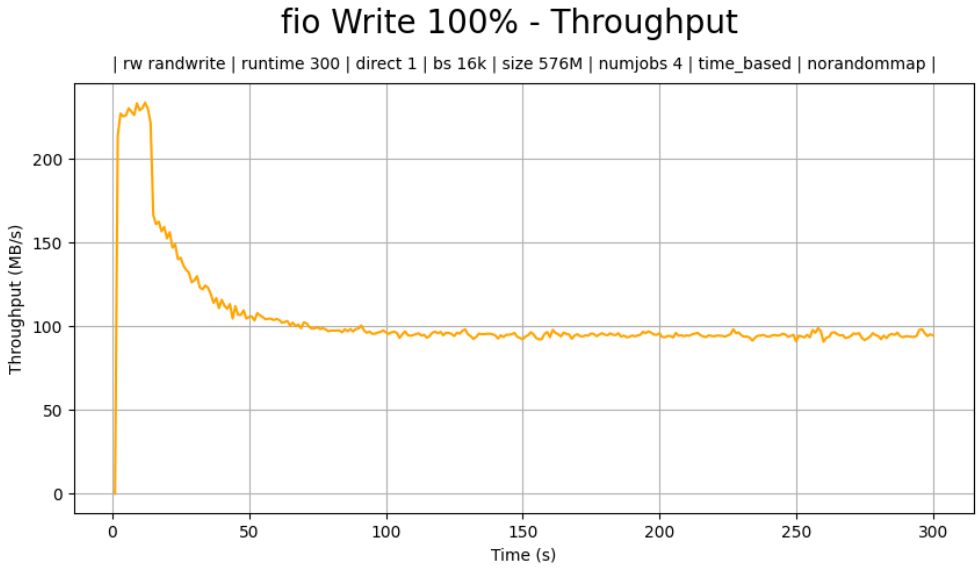

Figure 5.10: Throughput results for *fio* (write 100% workload)

Both graphs above exhibit write cliffs caused by GC.

Additionally, IOPS and throughput at write 100% show lower values compared to those at 50% read/50% write, likely due to the higher frequency of write operations.

• Workload (iii) - fio write 100%

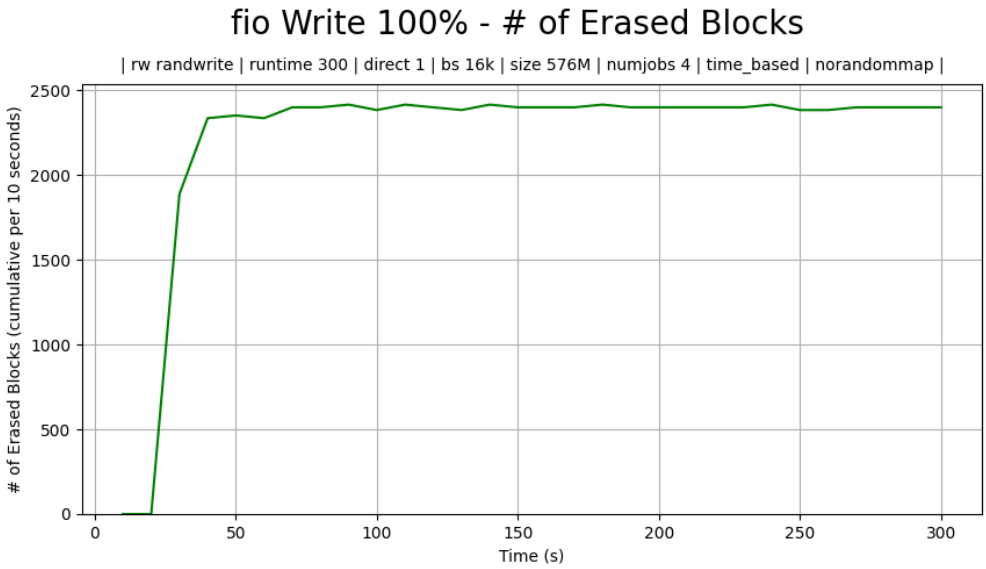

Figure 5.11: # of erased blocks results for *fio* (write 100% workload)

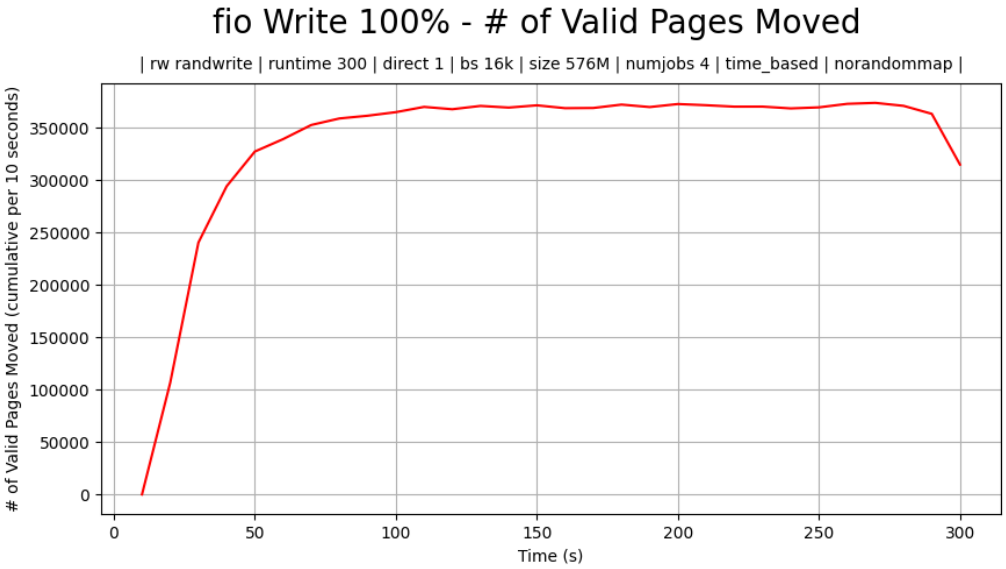

Figure 5.12: # of valid pages moved results for *fio* (write 100% workload)

Contrary to the previous case, the graphs above show both metrics at write 100% as higher than those at 50% read/50% write, likely due to the increased frequency of writes.

• Workload (iv) - sysbench 4 threads

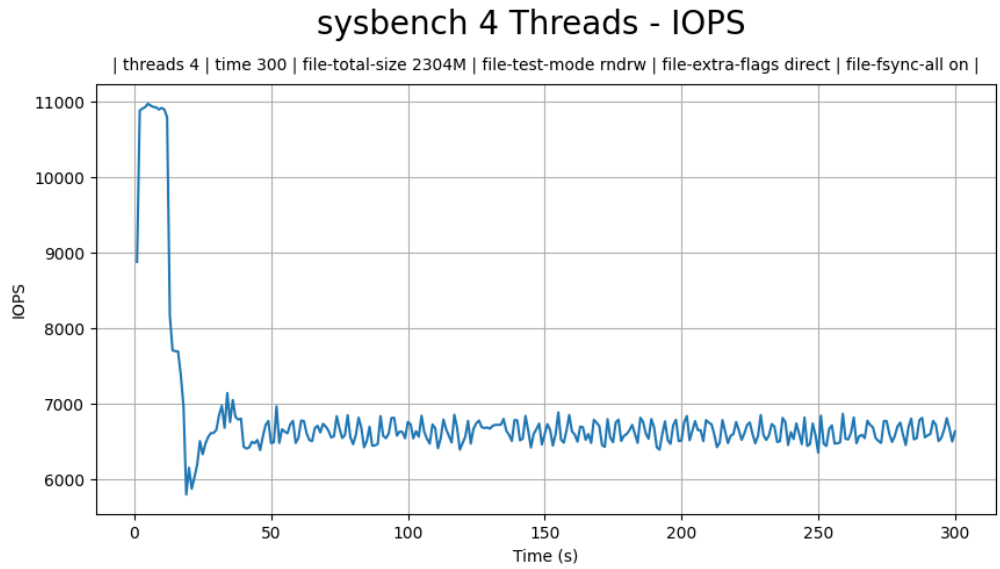

Figure 5.13: IOPS results for *sysbench* (4 threads workload)

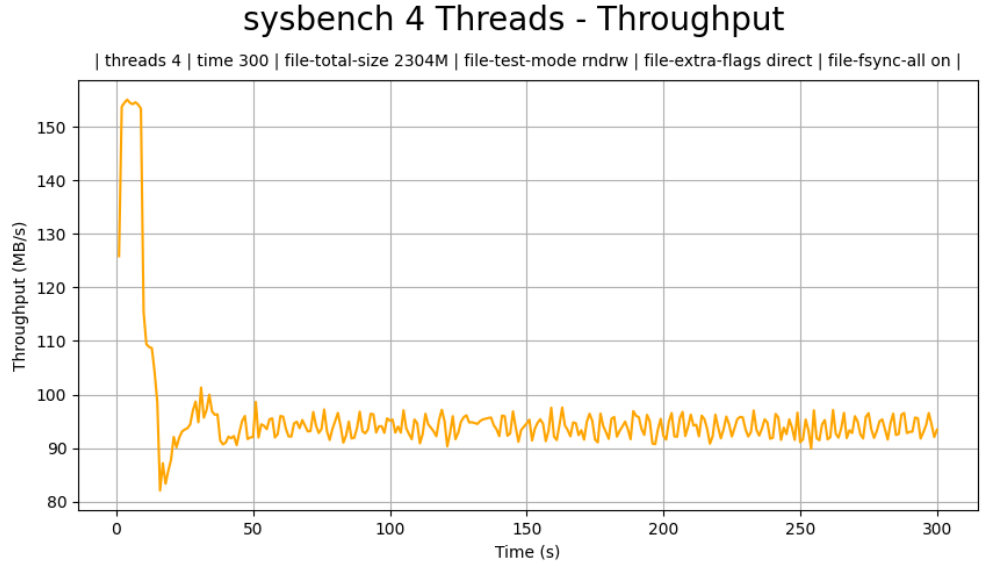

Figure 5.14: Throughput results for *sysbench* (4 threads workload)

In the graphs above, IOPS and throughput also show a sharp drop due to GC, followed by stabilization at a certain level.

• Workload (iv) - sysbench 4 threads

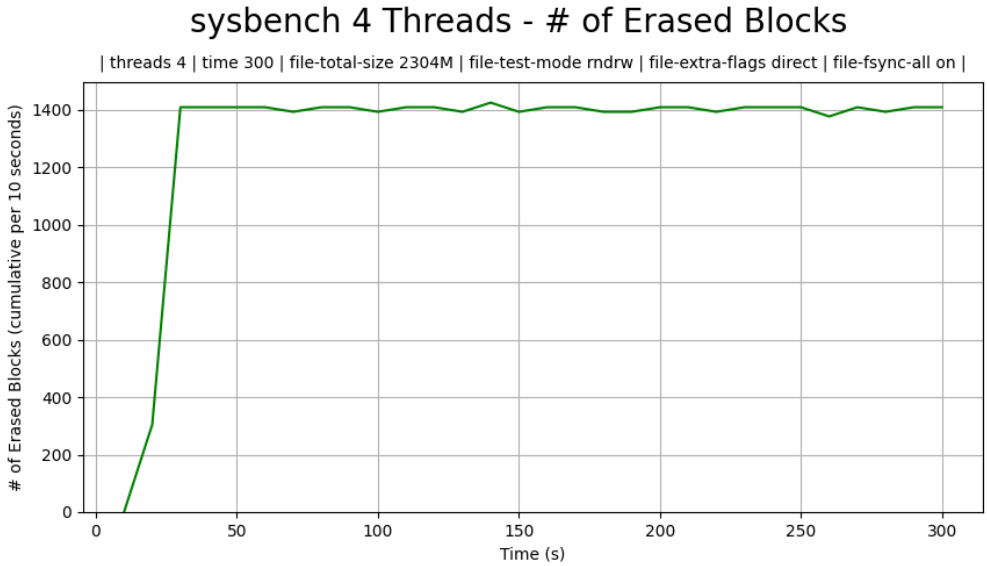

Figure 5.15: # of erased blocks results for *sysbench* (4 threads workload)

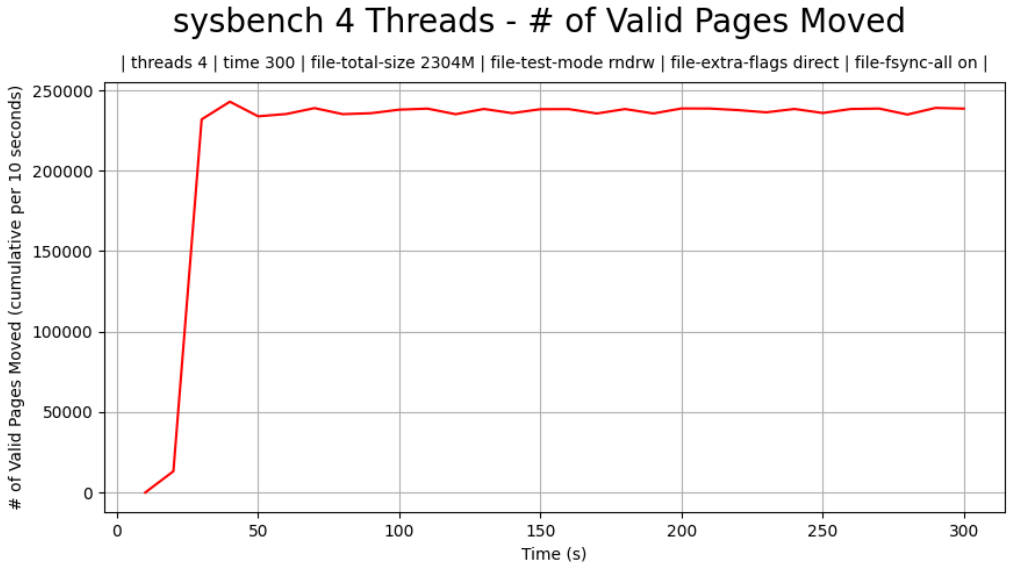

Figure 5.16: # of valid pages moved results for *sysbench* (4 threads workload)

Similarly, in the graphs above, both metrics rise sharply as GC is performed and then stabilize at a certain level.

• Workload (v) - sysbench 32 threads

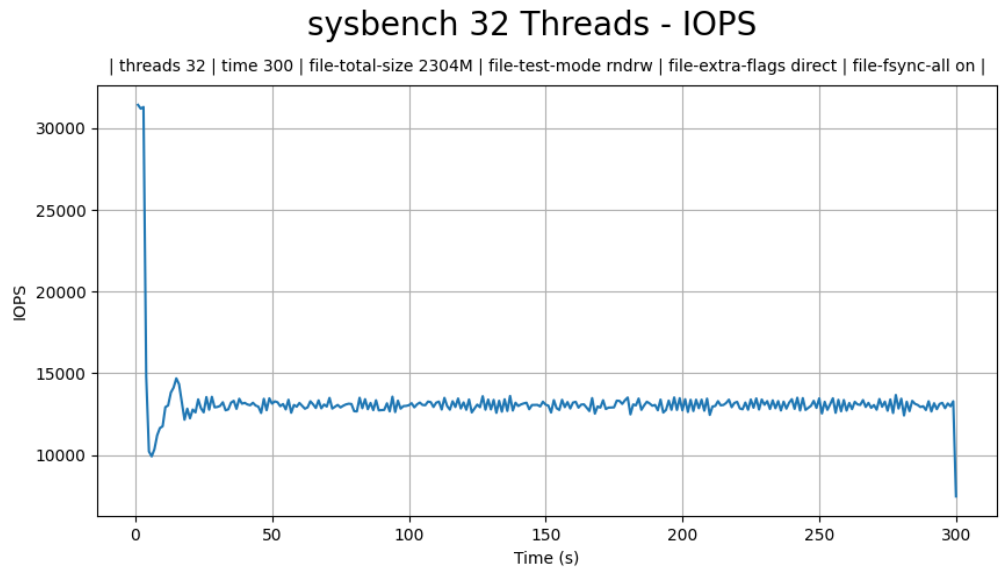

Figure 5.17: IOPS results for *sysbench* (32 threads workload)

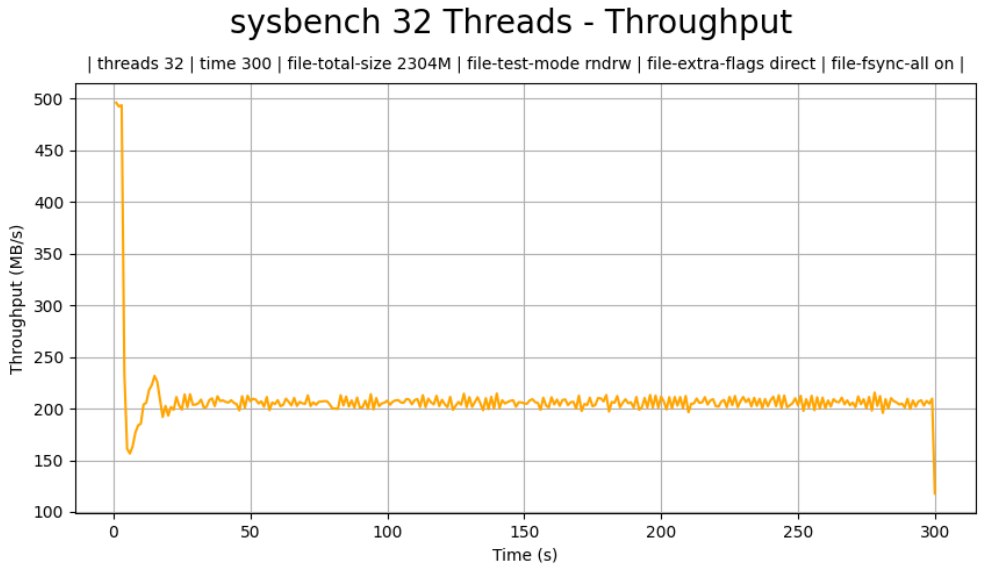

Figure 5.18: Throughput results for *sysbench* (32 threads workload)

With 32 threads, the workload shows significantly higher IOPS and throughput than with 4 threads, likely due to more I/O generated within the same timeframe. However, despite an 8-fold increase in threads, the metrics did not scale proportionally, likely due to hardware limitations of the computer running the guest OS or performance constraints in the FEMU SSD configuration.

• Workload (v) - sysbench 32 threads

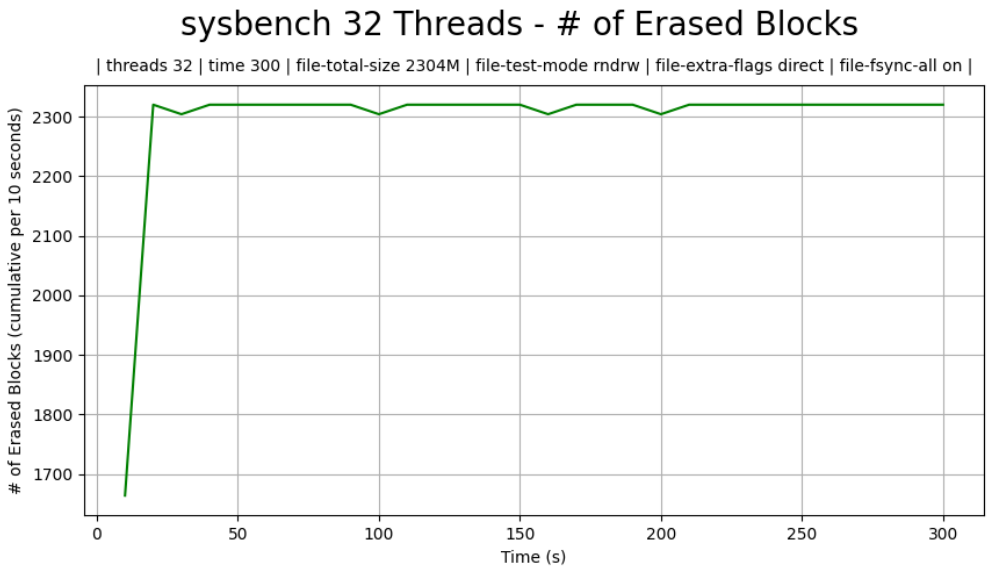

Figure 5.19: # of erased blocks results for *sysbench* (32 threads workload)

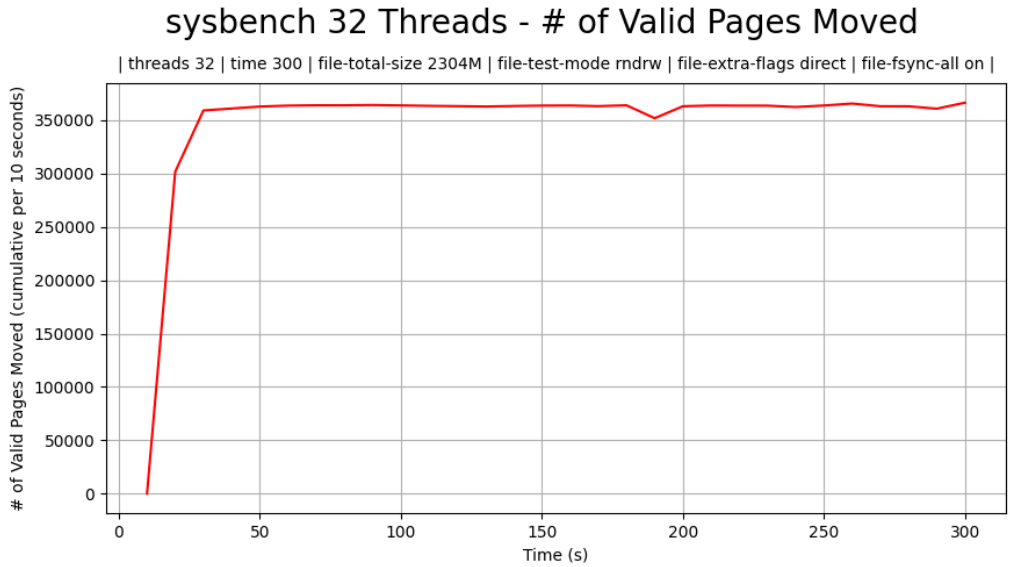

Figure 5.20: # of valid pages moved results for *sysbench* (32 threads workload)

For the same reason, increasing the worker threads generated more I/O operations, leading to a higher volume of writes and, as a result, higher metrics than the 4-thread configuration.

6 Conclusion

Up to this point, the FEMU source code was analyzed to understand how read, write, and GC operations work internally within the FEMU Blackbox SSD. Additionally, *fio* and *sysbench* were employed to generate I/O across various workloads and evaluate the impact of GC on SSD performance.

Through this analysis, the quantitative impact of the GC mechanism on SSD performance was verified. Notably, significant decreases in IOPS and throughput during GC events (write cliff) were observed, illustrating its critical effect on overall SSD performance.

Furthermore, insights were gained into how FEMU processes I/O and performs GC on a line-by-line basis, as well as how it determines the position of the next page to be written.
Based on this understanding, implementing hot/cold data separation in the future is expected to improve GC performance.

In conclusion, a comprehensive understanding of the GC operation and its impact on FEMU Blackbox SSD performance provides valuable foundational insights for future research, potentially contributing to SSD performance optimization and GC efficiency.

References

[1] https://github.com/MoatLab/FEMU

[2] https://github.com/axboe/fio

[3] https://github.com/akopytov/sysbench