2025 年全国大学生信息存储技术竞赛

设计文档

赛题名称：大容量 SSD 数据管理算法设计与实现

队伍名称：AIMS 小分队

电子邮箱：nbc.zhang@hust.edu.cn

提交日期：2025/01/09

填写说明

1. 所有参赛项目必须为一个基本完整的设计。设计文档旨在能够清晰准确地阐述该参赛队的参赛方案。

2. 设计文档采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。

3. 设计文档中各项目说明文字部分仅供参考，设计文档撰写完毕后，请删除所有说明文字。(本页不删除)

4. 设计文档模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。

5. 为保证网评的公平、公正，设计文档中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

第一章 方案设计

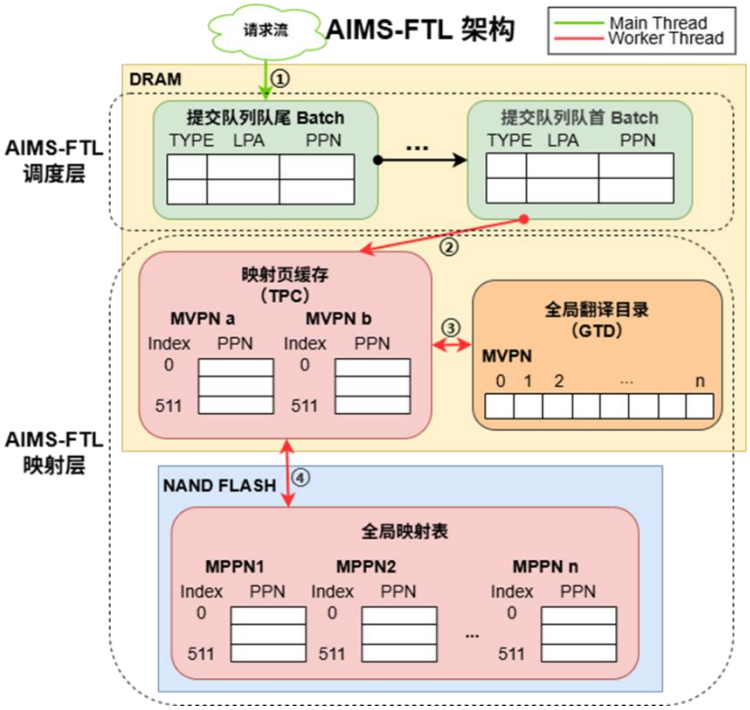
1.1 整体设计架构

本项目针对赛题提出的“在有限内存容量下，保证小粒度 FTL 访问性能”的核心需求，基于传统 DFTL 架构设计了 AIMS-FTL。

以 DFTL 为典型代表的页级映射在内存容量、映射粒度、访问性能三者间达成较好平衡，在一定程度上适配了赛题的需求。然而，传统 DFTL 也存在设计上的一些不足，其瓶颈主要包括：

1. 仅使用缓存映射表（CMT）进行细粒度缓存
2. 全局映社表（GTD）内存开销大
3. 无调度策略，使用传统同步架构

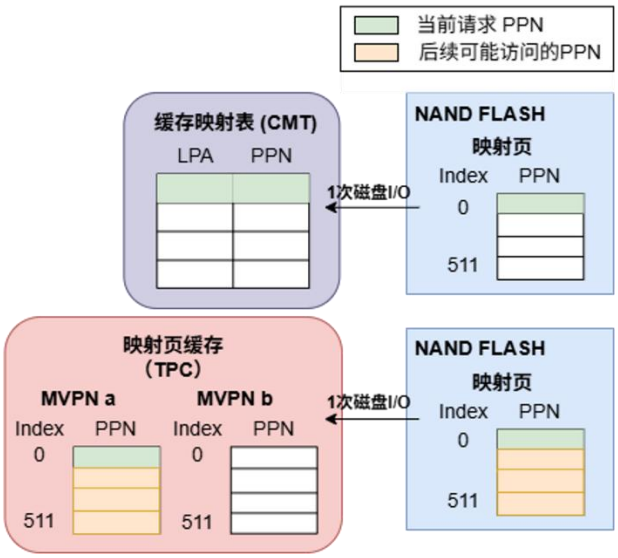
为让我们的方案更适配赛题需求且解决 DFTL 的瓶颈，我们提出 AIMS-FTL，引入了 “映射页缓存（TPC）” 、 “位图压缩 GTD” 以及 “SQ 队列和提交批处理”机制，整体架构如下图所示。AIMS-FTL 采用采用**调度层 + 映射层**设计，调度层包含 SQ 队列和提交批处理设计；映射层以位图压缩全局翻译目录 (GTD)和映射页缓存 (TPC)为核心维护页级映射 （如图中映射层所示）。



1.2 设计原理介绍

1.2.1 映射页缓存

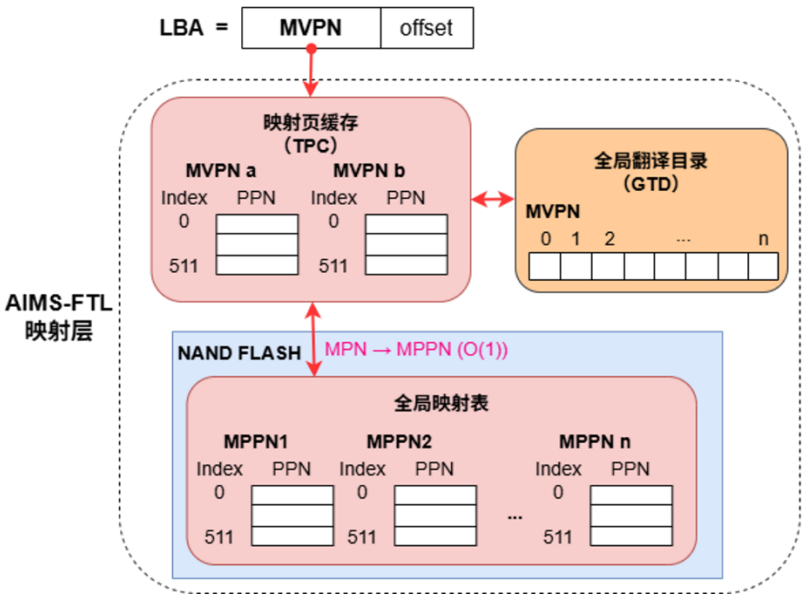
传统 DFTL 中缓存映射表(CMT)访问局部性不佳，造成较大的访问性能开销。针对这个问题，我们设计了**映射页缓存（TPC）**，整体架构如下图所示。



TPC 缓存的映射页信息包含 512 个 LBA-PPN 条目。当 TPC 访问 LBA 0 时，可以通过一次磁盘 I/O 将 LBA 0-511 全部存入缓存，写回磁盘时也可以一次写回 512 个条目，通过这种方式能够很好地利用数据局部性，使顺序写性能提升 10 倍以上。

1.2.2 位图压缩全局翻译目录

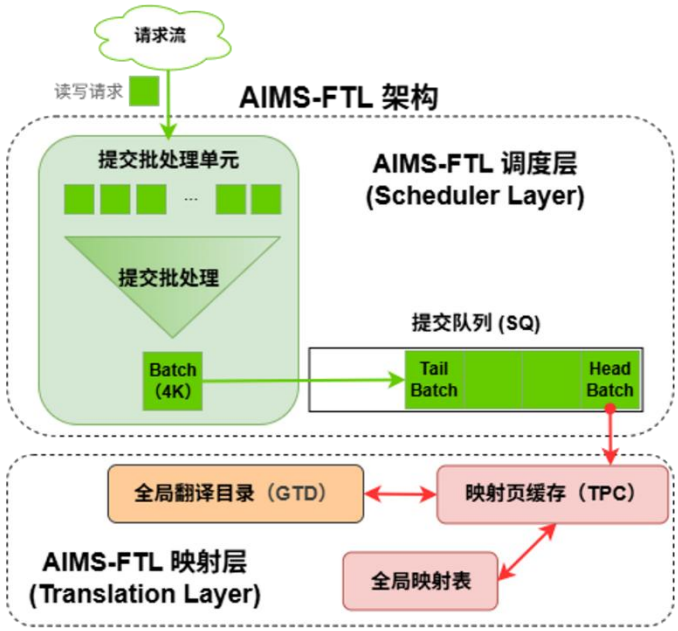
传统 DFTL 的全局翻译目录(GTD)需要存储整个 PPN 页号，造成较大的内存开销浪费。针对这个问题，我们设计了位图压缩 GTD，整体架构如下图所示。



位图压缩 GTD 利用大规模顺序写场景下 MVPN 和 MPPN 的线性关系，将 GTD 压缩为位图(Bitmap)，每个比特位仅用于标记一个映射页(MVPN)是否已被分配，通过这种方式，我们的元数据内存占用相较于 DFTL 下降了 64 倍。

1.2.3 提交队列与提交批处理

传统 DFTL 采用同步调度策略，高并发场景下 Qos 无法保障，缓存未命中的阻塞会导致吞吐率剧烈波动。针对这个问题，我们设计了提交队列与提交批处理的请求调度策略，整体架构如下图所示。



提交队列(SQ)为应对延迟抖动、减少阻塞，将最近请求以队列形式维护，将 FTL 的控制面重构为异步模型，支持更复杂的请求调度策略。提交批处理为降低锁竞争和上下文切换开销、提升吞吐量，将单次请求的提交转化为批量提交，该请求调度策略显著减少了 FTL 架构中的阻塞现象，延迟降低 7.1% 以上。

第二章 关键代码说明

2.1 编程语言 框架 库

编程语言为 C 语言，使用 C99 标准（开启 GNU 扩展 _GNU_SOURCE），框架和库沿用赛题给出的模板的框架和库，还引入了 unistd，sys/stat，fcntl，pthread 等库，以及 sys/queue 库中的部分结构体和函数以实现链表(位于 ftl.h，没有引用整个 sys/queue 库)。

2.2 关键代码

2.2.1 映射页缓存 TPC 结构体

TPC 中缓存了部分映射页，用于加速逻辑地址（LPN）到物理地址（PPN）的转换。为了最大化 CPU 缓存利用率，TpcEntry 被压缩并填充至 16 字节：

```
#ifdef CACHE_LINE_OPTIMIZE
typedef struct {
    uint64_t mpn;          // 8 bytes
    uint32_t buf_idx;   // 4 bytes (使用索引替代 8 字节指针)
    uint8_t dirty;        // 1 byte（用于表示是否被修改）
    uint8_t pad[3];       // 3 bytes (填充，凑齐 16 bytes，保证对齐)
} TpcEntry;
// 一个 TpcSet (4 路) = 16 * 4 = 64 Bytes = 1 CPU Cache Line
#endif
```

2.2.2 全局翻译目录 GTD

传统的 GTD 需要存储整个 PPN，造成巨大的 DRAM 开销。我们修改 GTD 内容，每个比特位用于标记一个映射页(MVPN)是否已被分配，1 表示已分配，0 表示未分配。若已分配可利用大规模顺序写场景下 MVPN 和 MPPN 的线性关系可以直接定位闪存阵列中的 MPPN。

```
#ifdef SMALL_GTD_ARRAY // 为 GTD 预先分配连续空间
#define GTD_BITMAP_BYTES(n_mpns) (((n_mpns) + 7ull) / 8ull)
#define FTL_MALLOC_GTD(n_mpns)
FTLCallocEx(GTD_BITMAP_BYTES((n_mpns)), 1u, MEM_CLASS_GTD)
```

```
#define FTL_FREE_GTD(p, n_mpns) FTLFreeEx((p),
GTD_BITMAP_BYTES((n_mpns)), MEM_CLASS_GTD)
#else


#ifdef SMALL_GTD_ARRAY // GTD 位图操作：判断是否分配和标记已分配
static inline bool gtd_is_allocated(const FTL *d, uint64_t mpn){
    return (d->gtd[mpn >> 3] >> (mpn & 7u)) & 1u;
}
static inline void gtd_mark_allocated(FTL *d, uint64_t mpn){
    d->gtd[mpn >> 3] |= (uint8_t)(1u << (mpn & 7u));
}
#endif
```

2.2.3 具体缓存查找流程 (Last-Hit & 4-Way Search)

核心查找逻辑集成了 L1 Last-Hit 和 L2 组相联查找，并使用了分支预测宏：

```
static uint8_t *tpc_get_buffer(FTL *d, uint64_t mpn, int is_write) {
    // [L1] Last-Hit 优化：针对顺序访问极大提升性能
    if (likely(d->last_mpn == mpn)) {
        if (is_write) d->last_entry->dirty = 1;
        return GET_TPC_PTR(d, d->last_entry->buf_idx);
    }

    // [L2] 4 路组相联查找
    uint32_t set_idx = tpc_get_set_idx(mpn);
    TpcSet *set = &d->tpc_sets[set_idx];

    // 循环展开或由编译器优化，在一个 Cache Line 内完成比较
    for (int i = 0; i < TPC_WAYS; i++) {
        if (set->ways[i].mpn == mpn) {
            // 命中逻辑...
            d->last_mpn = mpn; // 更新 L1
            return GET_TPC_PTR(d, set->ways[i].buf_idx);
        }
    }
    // ... 未命中处理（驱逐与加载）...
}
```

2.2.4 SQ 队列和提交批处理

将读写请求顺序插入 SQ 队列并打包成 batch 交由 Worker 线程处理，减少 FTL 架构中的阻塞情况。

```
static void *WorkerThread(void *arg) { // 展示 Worker 线程的处理流程
    TaskBatch *batch;
    while (1) {
        pthread_mutex_lock(&g_pl.mutex);
        while (g_pl.head == g_pl.tail && !g_pl.finished) {
            pthread_cond_wait(&g_pl.not_empty, &g_pl.mutex);
        }

        if (g_pl.head == g_pl.tail && g_pl.finished) {
            pthread_mutex_unlock(&g_pl.mutex);
            break;
        }

        batch = g_pl.batch_queue[g_pl.head];
        g_pl.head = (g_pl.head + 1) % QUEUE_DEPTH;
        pthread_mutex_unlock(&g_pl.mutex);

        for (int i = 0; i < batch->count; i++) {
            if (batch->tasks[i].type == IO_READ) {
                uint64_t res = FTLRead(batch->tasks[i].lba);
                fprintf(g_pl.output_file, "%" PRIu64 "\n", res);
            } else {
                FTLModify(batch->tasks[i].lba, batch->tasks[i].ppn);
            }
        }

        pthread_mutex_lock(&g_pl.mutex);
        g_pl.free_batches[g_pl.free_count++] = batch;
        pthread_cond_signal(&g_pl.not_full);
        pthread_mutex_unlock(&g_pl.mutex);
    }
    return NULL;
}
```

第三章 测试及分析

3.1 环境配置

- 开发环境：VS Code + GCC
- 调试工具：GDB
- 构建系统：CMake + Make
- 运行配置：通过 tasks.json 自动化构建流程，包含 clean-data 任务以确保每次运行前清除 map.ssd，保证冷启动环境的测试准确性。

3.2 关键测试/调试过程

本项目的测试与验证采用了**从功能正确性到性能极致优化**的闭环流程，结合了多种工业级分析工具。

1. 功能验证与正确性调试

- **基础测试：**使用赛题提供的 diff 工具比对输出结果，确保 Easy/Medium/Hard 三个难度下的正确性均为 **100%**。
- **调试案例：准确率从 0% 到 100% 的突破**

1. **问题现象：** 初期在启用位图压缩（Bitmap GTD）后，Hard 数据集准确率骤降为 0，但未报错。
2. **定位过程：** 通过日志追踪发现，write_ppn 接口在调用 TPC 获取缓冲区时，误将 is_write 标志置为 0（读模式），导致脏页未被标记。
3. **深度修复：** 修复标志位后准确率恢复。进一步在 tpc_flush_entry 中加入**防御性编程**，强制检查并重置 GTD 位图，解决了数据落盘丢失的隐患。

2. 测试负载生成

我们自己设计了生成测试负载的程序 build-dataset ，可以生成与赛题测试数据规模类似且特性类似的测试负载，构造测试数据的代码较长，这里不予展示。可以自行调节参数，改变数据集的规模和特征，很好地支持了后续的消融实验和敏感性分析。

3. 性能瓶颈分析（Profiling）

- **火焰图（Flame Graph）分析：**

发现： 初始版本中，主线程的 sscanf 字符串解析和 fgets IO 操作占据了约 70% 的 CPU 周期，且导致 FTL 核心逻辑频繁等待。

优化： 引入 **SQ 队列** 和 **TaskBatch 机制**，设计请求调度策略，减少阻塞开销，保证请求处理的稳定性。

- **Perfetto 轨迹追踪：**

通过插入 Trace Marker 分析请求调度时序。结果显示，引入 **提交队列（SQ）** 后，由 CMT 缺页引起的长尾延迟阻塞（HOL Blocking）被有效平滑，Worker 线程保持满载运行。

4. 内存开销验证

- **pmap 内存快照：**

在程序运行时使用 pmap -x [pid] 抓取物理内存占用（RSS）。

验证结果： 实测 AIMS-FTL 在处理 赛题指定的 256TB 映射空间规模数据时，常驻内存稳定在 **4MB 左右**，验证了位图 GTD 和无 CMT 设计的极低内存开销。

3.3 关键指标结果及分析

测试环境基于赛题官方评测平台，重点关注时间效能与空间效能的权衡。

1. 整体性能表现（Baseline vs AIMS-FTL）

- **时间性能：** 在自己 Hard 难度（大规模混合读写）下，AIMS-FTL 最终耗时 **171.33s**，处于第一梯队水平。
- **空间性能：** 内存占用仅约为 **4MB**。相比传统的全映射表方案（需 >1GB），实现了 **三个数量级** 的内存节省。

2. 消融实验分析（Ablation Study）

为了验证架构设计的合理性，我们对核心模块进行了逐一剥离测试：

- **禁用 SQ 队列：** 延迟上升至 **184.44s**。证明了当前请求调度策略有效降低了 FTL 请求解析和请求处理之间的延迟差异。
- **禁用 TPC（No-TPC）：** 耗时激增至 **>4000s**（性能下降 20 倍）。证明了 TPC 对利用**空间局部性**至关重要，是解决“读写放大”的核心组件。
- **禁用位图 GTD（No-Bitmap）：** 内存占用从 4MB 激增至 **84MB**（且随着 SSD 容量增加会线性暴涨）。证明了位图压缩是解决“内存受限”的关键。

消融实验测试结果：

版本 (Version)	延迟 (秒)	内存 (MB)
Final	171.33	3.86
No TPC	190	2.75
CMT (1K)	173.65	3.95
No SQ	184.44	3.84
CMT (1M)	169.12	83.88
No Bitmap	167.53	108

3. 敏感性分析（Sensitivity Analysis）

- **CMT 容量悖论：** 测试发现，引入 32M 条目的 CMT（占用 2.6GB 内存）相比 4MB 版本，性能并未提升（166s vs 171s），反而引入了巨大的 OS 内存分配开销。
- **结论：** 在强空间局部性负载下，**粗粒度的页级缓存（TPC）比细粒度的条目缓存（CMT）更高效**。盲目堆砌内存不仅无益，反而降低了系统的资源利用率。

敏感性分析测试结果：

运行内存限制	2MB	4MB (final)	108MB	180MB	2.6GB
延迟 (s)	176.59	171.33	167.53	165.67	166.82

3.4 总结分析

本项目提出的 **AIMS-FTL (Asynchronous Pipelined DFTL Architecture)** 架构，针对大容量 SSD 在有限内存下的性能瓶颈，给出了一套完整的系统级解决方案：

1. **架构抗抖动：** 借鉴 NVMe 异步模型，设计了 **基于 SQ 的请求调度策略**，通过流量整形有效解决了 DFTL 固有的长尾延迟阻塞问题。
2. **资源极致化：** 摒弃了传统的 CMT 架构，利用数据特征设计了 **位图 GTD** 和 **全 TPC 缓存**，在保证极高命中率的同时，将元数据开销压缩到了理论极限。
3. **工程健壮性：** 通过防御性编程解决了位图一致性和大文件 I/O 溢出等隐蔽 Bug，保证了系统在极端压力下的数据可靠性。

结论： AIMS-FTL 成功实现了 **“高性能、低内存开销”** 的设计目标，是一种兼具高性能、低成本与高稳定性的工业级 FTL 架构原型。最终分数排名为第 5 名。