

新型文件系统

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

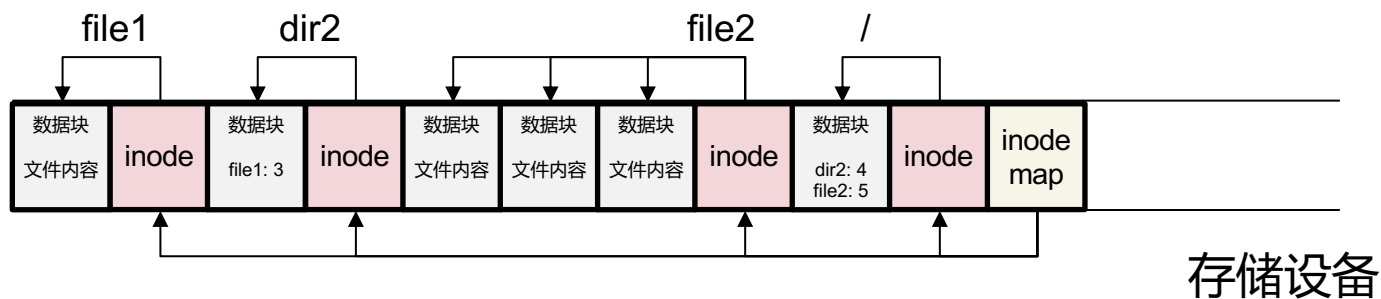
<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

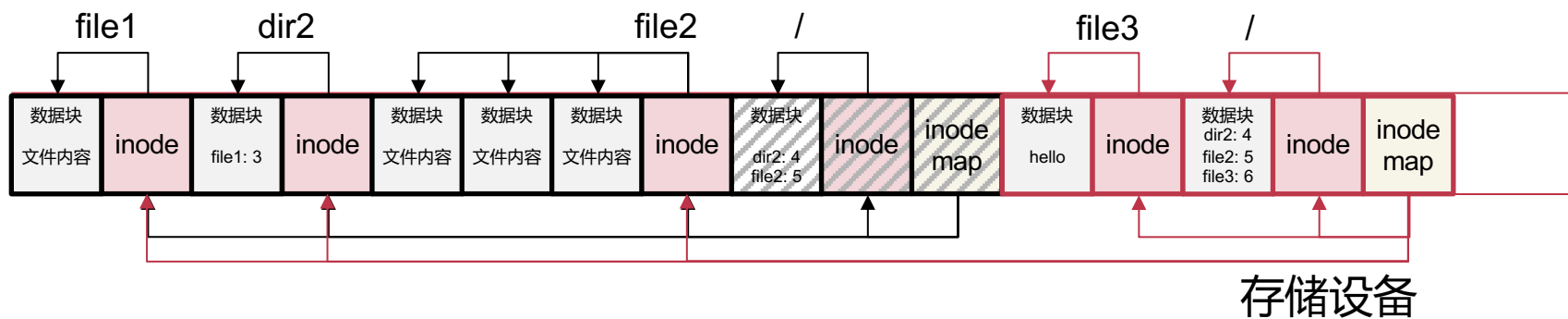
Review : LFS

- 一个日志文件系统
 - 有4个inode，位置记录在inode map中
 - 对应4个文件分别为：/，/dir2，/file2，/dir2/file1



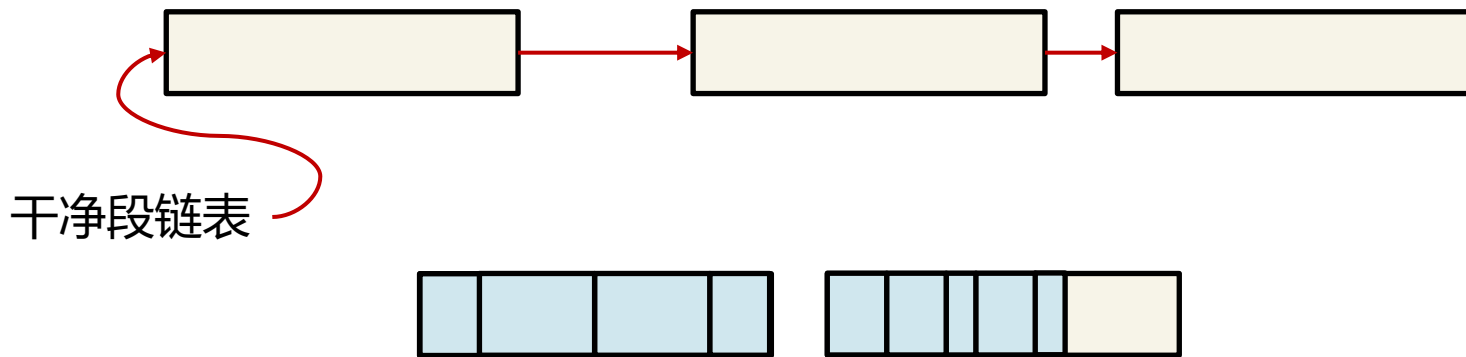
Review : 创建文件举例

- echo hello > /file3
 - 创建文件
 - 修改文件数据



Review : 段清理

1. 将一些段读入内存中准备清理
2. 识别出有效数据
3. 将有效数据整理后写入到干净段中 (对应拷贝方法)
4. 标记被清理的段为干净



Review : 闪存盘的性质

- **非对称的读写与擦除操作**

- 页 (page) 是读写单元 (8-16KB)
- 块 (block) 是擦除单元 (4-8MB)

- **Program/Erase cycles**

- 写入前需要先擦除
- 每个块被擦除的次数是有限的

- **随机访问性能**

- 没有寻道时间
- 随机访问的速度提升，但仍与顺序访问有一定差距

Review : 闪存盘的性质

- **磨损均衡**

- 频繁写入同一个块会造成写穿问题
- 将写入操作均匀的分摊在整个设备

- **多通道**

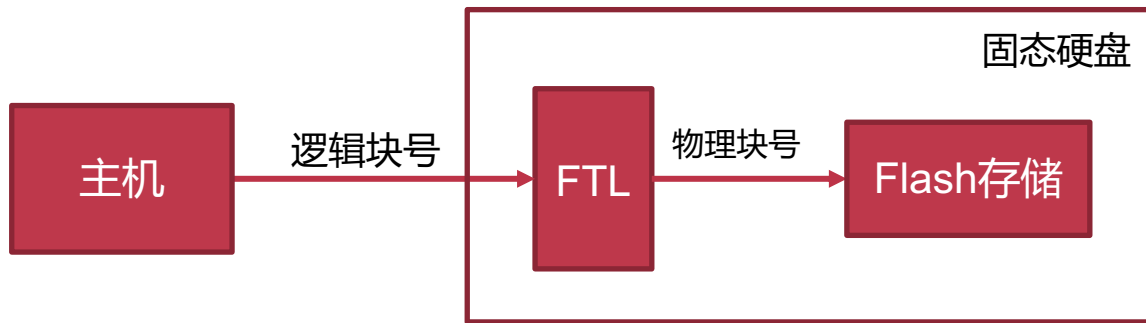
- 高并行性

- **异质Cell**

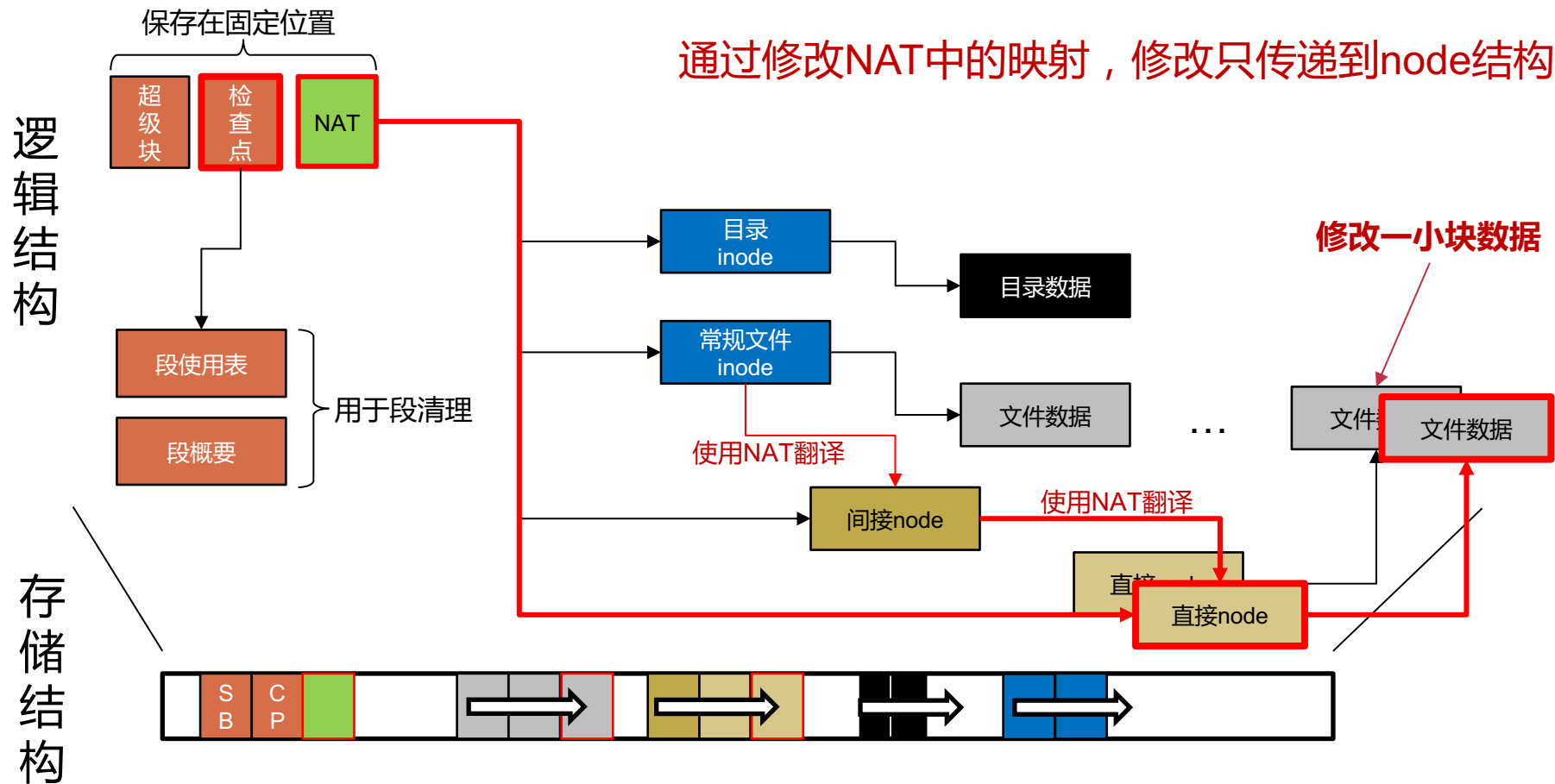
- 存储1到4个比特：SLC、MLC、TLC、QLC

Flash Translation Layer (FTL)

- 逻辑地址到物理地址的转换
 - 对外使用逻辑地址
 - 内部使用物理地址
 - 可软件实现，也可以固件实现
 - 用于垃圾回收、数据迁移、磨损均衡（wear-levelling）等



Review : F2FS的改进 : NAT



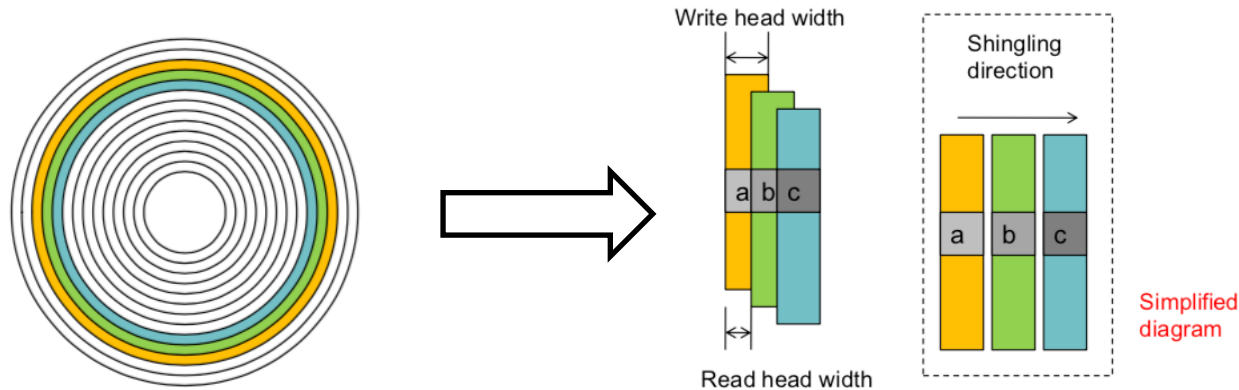
Review : 多Log写入

- 按热度将结构分类
 - 每个类型和热度对应一个log
 - 默认打开6个log
 - 用户可进一步配置
- 根据硬件信息可以进一步调整
 - 调整zone、section大小
 - 与硬件GC单元对齐等

类型	热度	对象
Node	热	目录的直接node块 (包括inode块)
	温	常规文件的间接node块
	冷	间接node块
Data	热	存放目录项的数据块
	温	常规文件的数据块
	冷	被清理过程移动的数据块 用户指定的冷数据块 多媒体文件的数据块

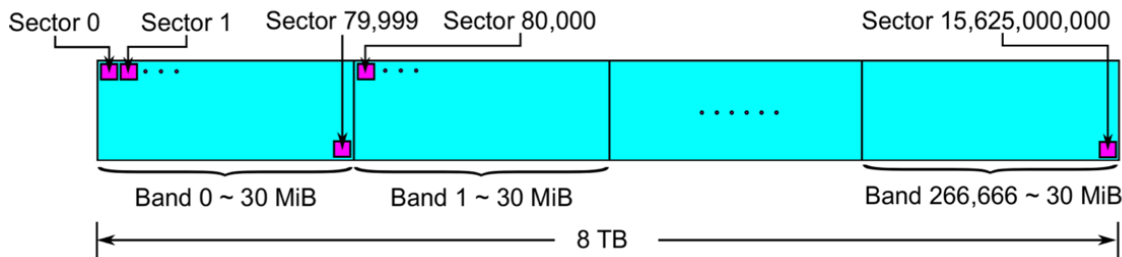
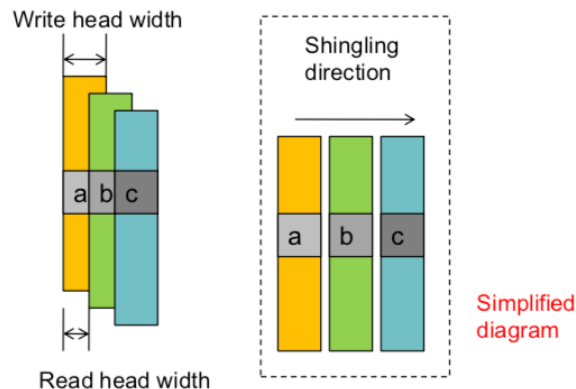
Review : 瓦式磁盘

- 传统磁盘密度难以提升
 - 写磁头的宽度难以减小
- 瓦式磁盘将磁道重叠，提升存储密度
 - 减小读磁头的宽度



瓦式磁盘的问题：随机写

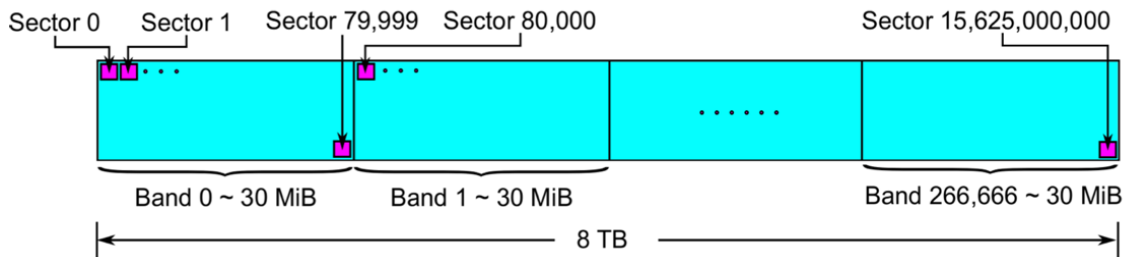
- 随机写会覆盖后面磁道的数据
 - 只能顺序写入
- 避免整个磁盘只能顺序写入
 - 磁盘划分成多个Band，Band间增大距离
 - 每个Band内必须顺序写入



Logical view of an 8 TB statically mapped drive-managed SMR disk

Band内随机写怎么办？

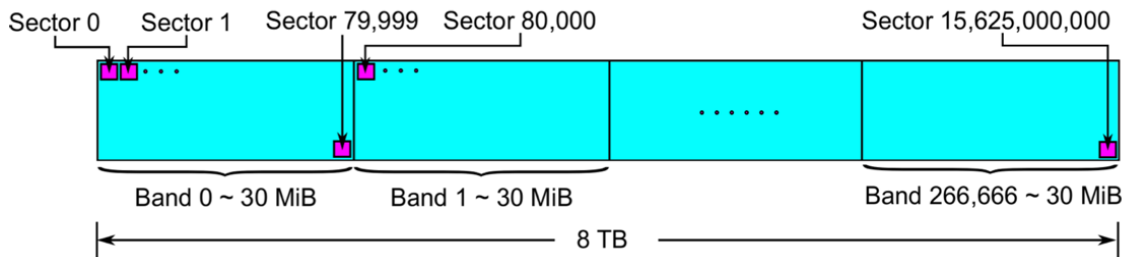
- Band大小 (30MB) > 块设备读写粒度 (4KB)



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法一：多次拷贝

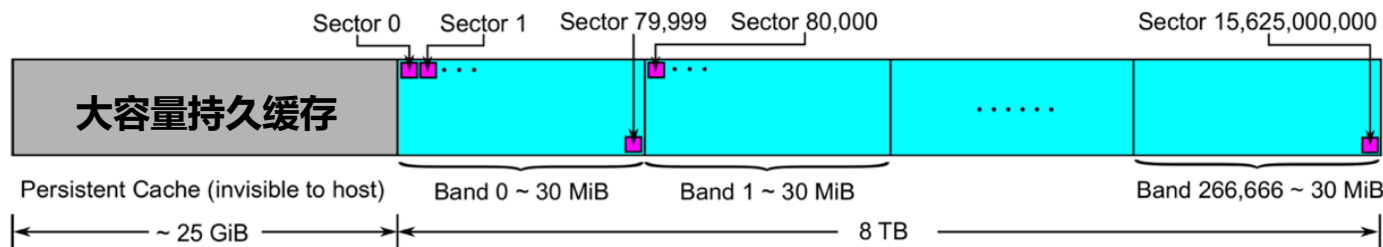
- 修改Band X中的4KB数据
 1. 找到空闲Band Y
 2. 从Band X的数据拷贝到Band Y，拷贝时将4KB修改写入
 3. 将Band Y中的数据拷贝回Band X
- 4KB随机写 → 120MB访问



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法二：缓存+动态映射

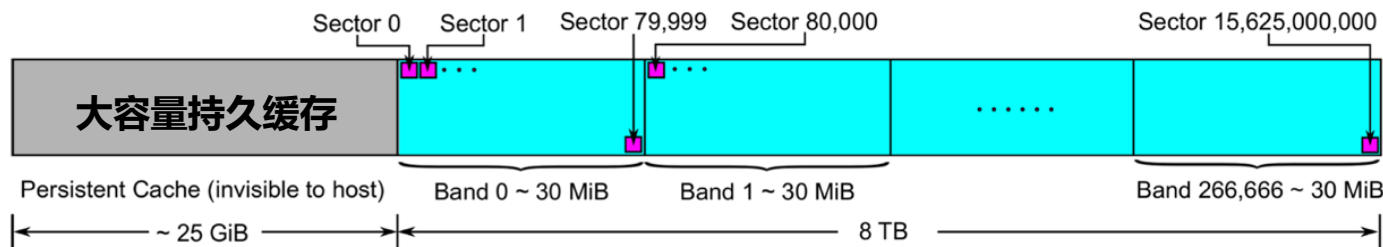
- 大容量持久缓存
 - 在磁盘头部预留的区域，磁道不重叠，可随机写入
 - 给固件（STL）单独使用，外部不可见
- 动态映射：Shingle Translation Layer (STL)
 - 从外部（逻辑）地址到内部（物理）地址的映射



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法二：缓存+动态映射

- 修改Band X中的4KB数据
 1. 将修改写入缓存，标记Band X为dirty
 2. 修改STL映射（让原位置指向持久化缓存）
 3. 空闲时，根据缓存内容，将 dirty Band进行清理
- 4KB随机写 → 修改4KB缓存



Logical view of an 8 TB statically mapped drive-managed SMR disk

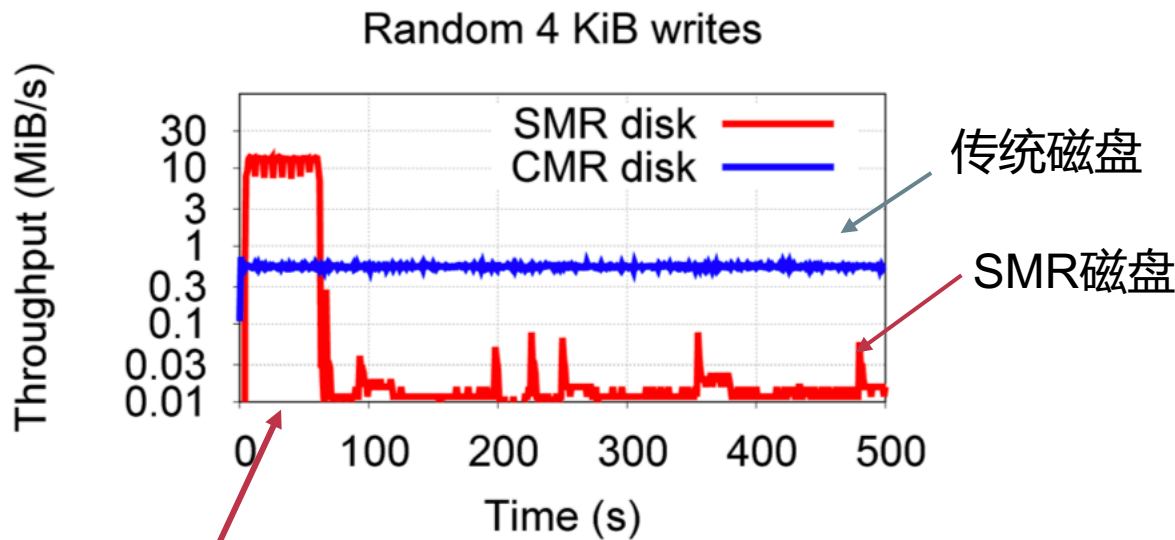
瓦式磁盘种类

SMR磁盘种类	接口	随机写处理方法
Drive-managed SMR (DM-SMR)	普通块设备接口	固件进行缓存和清理
Host-aware SMR (HA-SMR)	特殊指令接口	固件进行缓存和清理
Host-managed SMR (HM-SMR)	特殊指令接口	必须顺序写，随机写请求被拒绝

无需修改软件，可直接替换传统磁盘！

DM-SMR上使用Ext4

- 当随机写入时，Ext4吞吐量非常低！

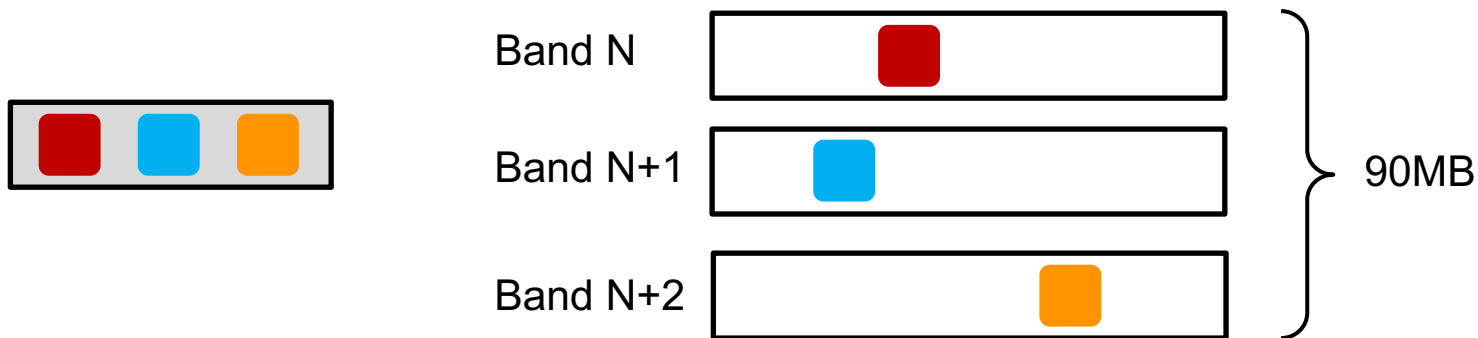


思考一下：为何这段时间内SMR磁盘上吞吐量很高？

如何改进Ext4来适应瓦式磁盘呢？

- **以DM-SMR磁盘为目标**
 - HM-SMR和HA-SMR需要文件系统的设计新的文件系统
 - 在成熟的Ext4上优化：消除元数据写回造成的随机写入
- **修改量小**
 - Ext4 + journaling 代码 ~50,000 行
 - 仅仅修改 ~40 行代码，新文件中添加 ~600 LOC 代码
- **效果显著**
 - 元数据修改较少时(<1%)，在SMR磁盘上有1.7-5.4倍性能提升
 - 有大量元数据修改时，在SMR和普通磁盘上，有2-13倍性能提升
 - 特定场景下达到40倍性能提升

观察：持久缓存对吞吐量的影响



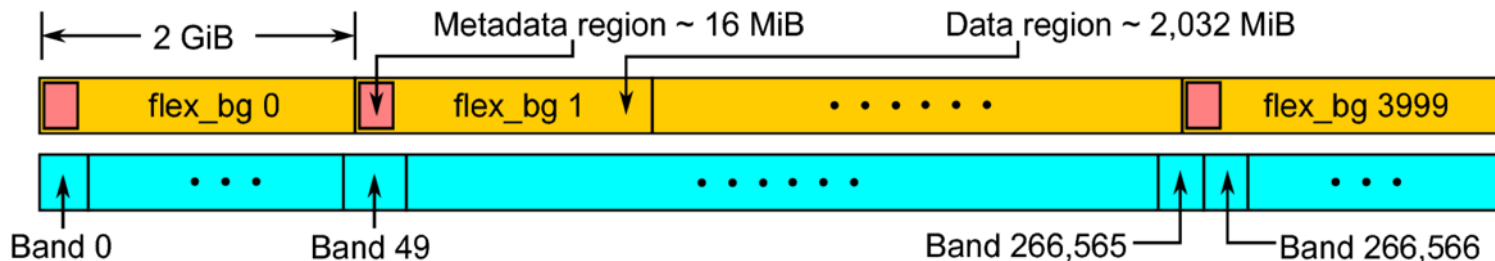
若持久化缓存中的写比较分散，清理时需要清理大量Band
假设清理1个Band需要1秒，吞吐量为1个随机写/秒



若三个随机写在一个Band中，则只需清理一个Band；吞吐量为3个随机写/秒
随机写的跨度 → 脏band数量 → 清理时的工作量 → 吞吐量

然而：Ext4的元数据非常分散

- Ext4使用块组 (flex_bg) 将文件系统分成多个区域
 - 每个块组前16MB用来保存元数据，其余保存数据
- 每次数据修改产生多处元数据的修改
- 8TB分区上有4,000个块组，元数据分散在4,000个Band！
- 分散的元数据随机写 → 脏band数量↓ → 清理工作的负担↓ → 吞吐量↑

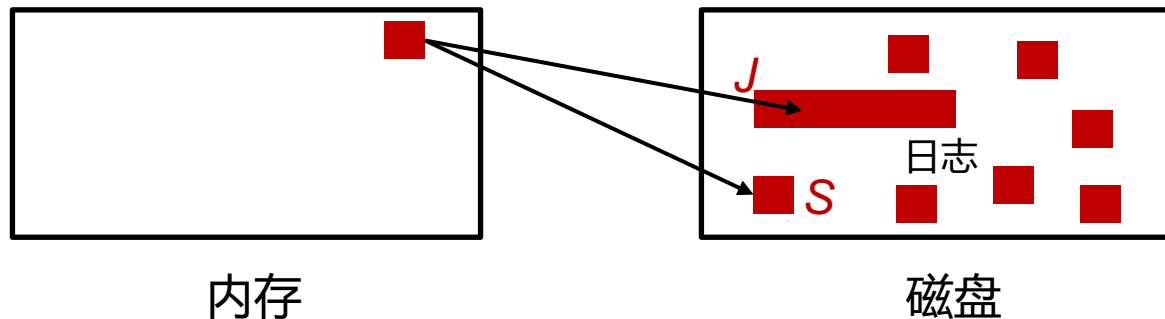


Ext4 partition laid on top of an 8 TB SMR disk

回顾：Ext4上的元数据写回

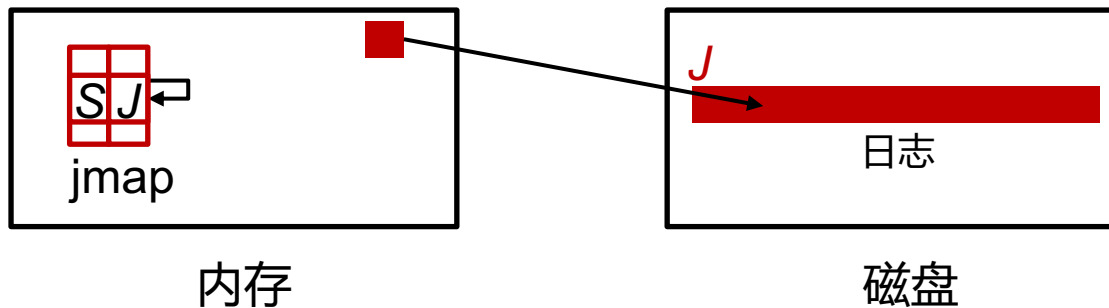
- Ext4使用JBD2记录元数据日志
 - 128MB的日志区域
 - 1. JBD2首先将元数据写入日志区域 J ，标记元数据为脏
 - 2. 脏元数据在日志提交后被写回到其应有位置 S

频繁的元数据写回，造成大量的分散随机写，降低吞吐量



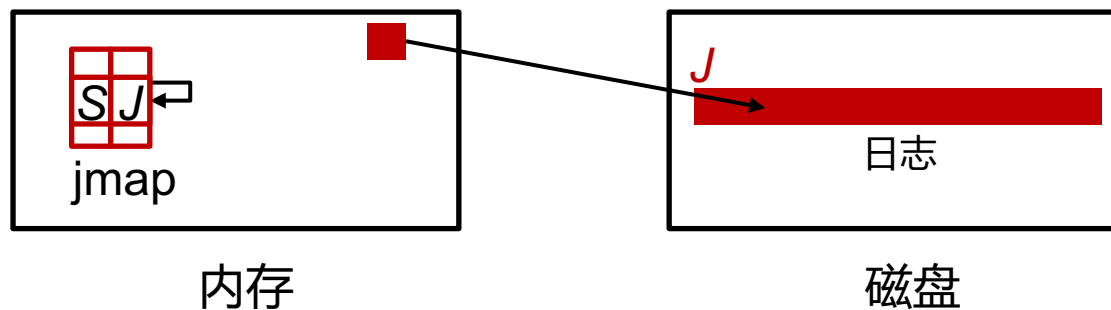
回到问题：Ext4上的元数据分散

- 修改磁盘布局需要大规模修改Ext4
 - 人力成本、新增Bug、破坏原有功能.....
- 怎么办？
- 引入Indirection：以LFS形式增加一个元数据缓存



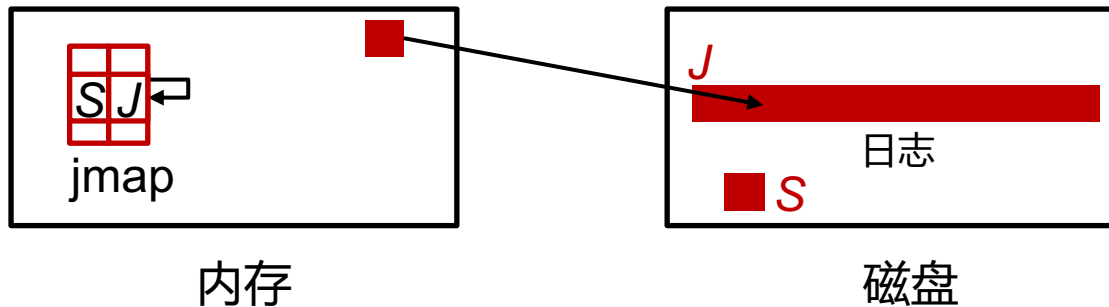
解决方法：以LFS形式增加一个元数据缓存

- 以LFS形式维护10GB日志空间作为元数据缓存
 - JBD2首先将元数据写入日志区域 J ，将元数据标记为clean（无需写回）
 - JBD2在内存中的jmap中将 S 映射到 J
- Indirection: 元数据访问需要通过jmap进行一次地址转换



日志满了怎么办？

- 日志空间清理
 - 无效的元数据（被新修改覆盖过的元数据）可以直接被回收
 - 对于冷元数据，可将其写回到Ext4中其原本的位置S
 - 热元数据继续保留在日志中
- 挂载FS时，读取日志，恢复出jmap



实现：对Ext4的修改

修改的~40行代码：将请求转给jbd2

```
- submit_bh(READ | REQ_META | REQ_PRIO, bh);  
+ jbd2_submit_bh(journal, READ | REQ_META | REQ_PRIO, bh, __func__);
```

新增文件：jmap.c (维护jmap)

```
+void jbd2_submit_bh(journal_t *journal, int rw, struct buffer_head *bh,  
+                  const char *func)  
{  
+    sector_t fsblk = bh->b_blocknr;  
+    ...  
+    je = jbd2_jmap_lookup(journal, fsblk, func);  
+    if (!je) {  
+        submit_bh(rw, bh);  
+        return;  
+    }  
+    ...  
+    logblk = je->mapping.logblk;  
+    read_block_from_log(journal, rw, bh, logblk);  
+}
```

拦截Ext4对block的请求

查找jmap

不在jmap中，直接访问磁盘对应block

根据jmap中记录的地址，在日志中读取block

实现：对Ext4的修改

新增文件：cleaner.c（维护10GB日志空间）

```
+static void do_clean_batch(struct work_struct *work) 将10GB日志空间的元数据写回到其在Ext4中的原本位置
+{
+    struct cleaner_ctx *ctx;
+    handle_t *handle = NULL;
+    int nr_live, err;
+    ...
+    nr_live = find_live_blocks(ctx);
+    if (nr_live == 0)
+        goto done;
+    ...
+    read_live_blocks(ctx, nr_live);
+    handle = jbd2_journal_start(ctx->journal, nr_live);
+    ...
+    attach_live_blocks(ctx, handle, nr_live);
+    err = jbd2_journal_stop(handle);
+done:
+    ...
+}
```

扫描日志中有效块，并加入ctx中临时保存

新的JBD2原子更新

将ctx中的有效块加入到JBD2原子更新中

结束JBD2原子更新

Non-volatile Memory (NVM)

非易失性内存

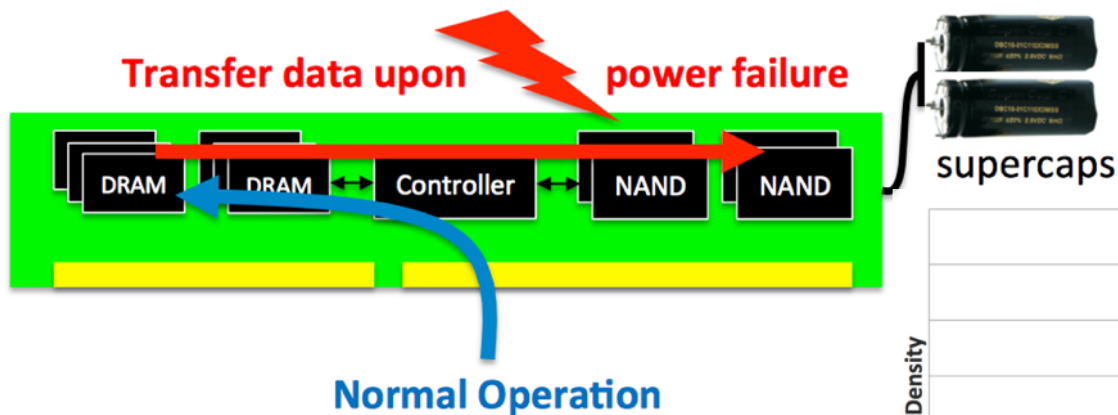
非易失性内存

如果内存里面的数据重启后还在，岂不是很棒？

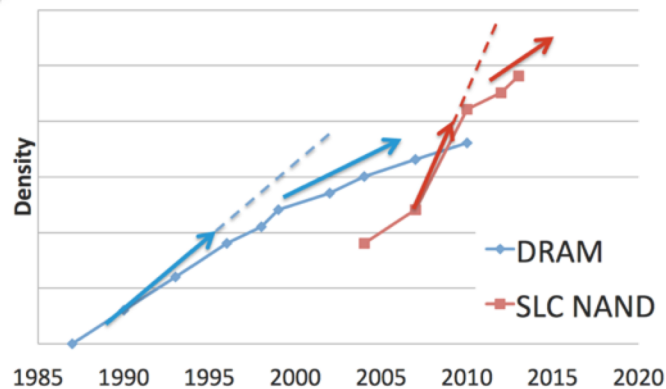
思考一下：会有哪些好处？

NVDIMM

- 在内存条上加上Flash和超级电容
 - 平时数据在DRAM中；断电后转移到Flash中持久保存

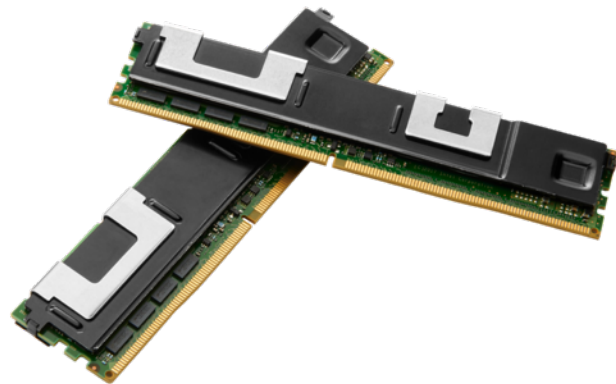


- 容量很难再提升

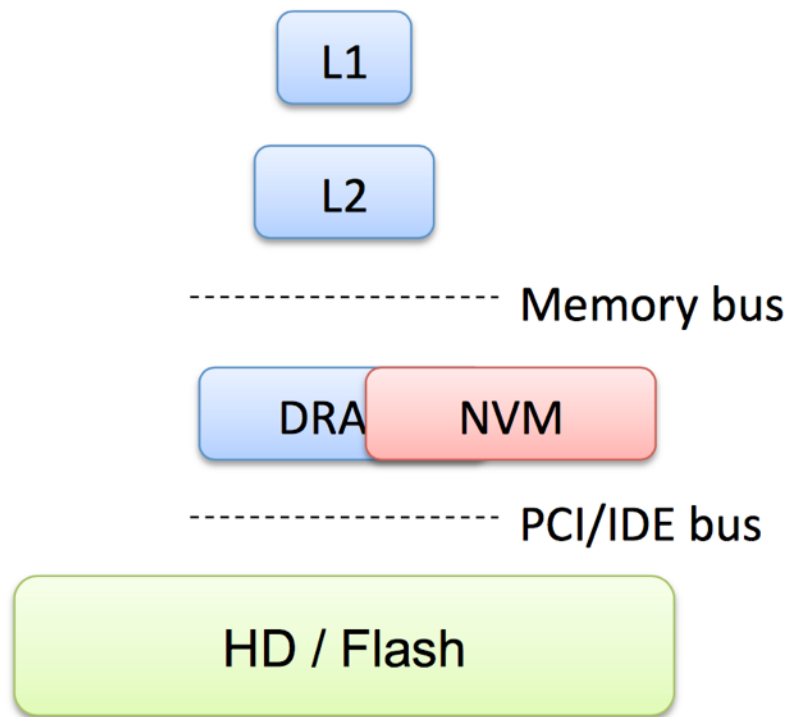


Intel Optane DC Persistent Memory

- ✓ 内存接口
- ✓ 字节寻址
- ✓ 持久保存数据
- ✓ 高密度 (512GB/DIMM)
- ✓ 需要磨损均衡，但耐磨度比NAND好10倍
- ✓ 比DRAM慢十倍以内，比NAND快1000倍



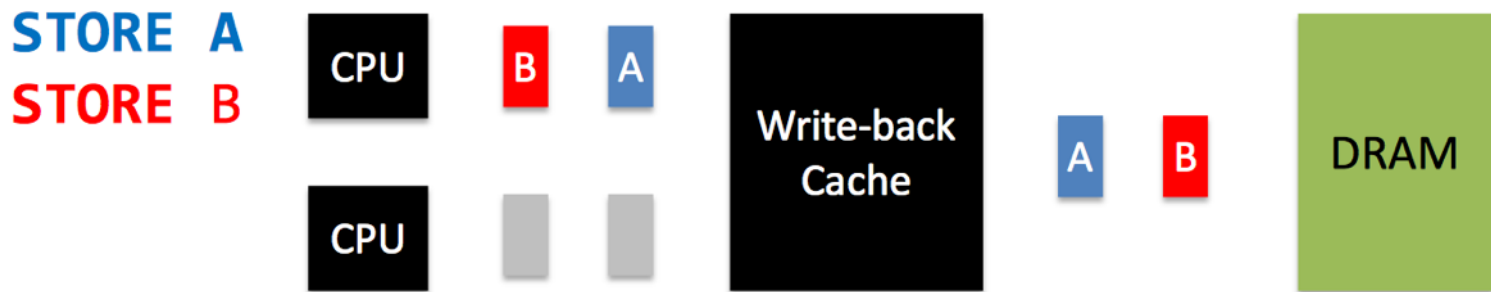
非易失性内存带来的新问题



	DISK	NVM
数据缓存位置	内存	CPU 缓存
写回	软件控制	硬件控制

内存写入顺序

- Writeback模式的CPU缓存
 - 提升性能
 - 会打乱内存写入顺序



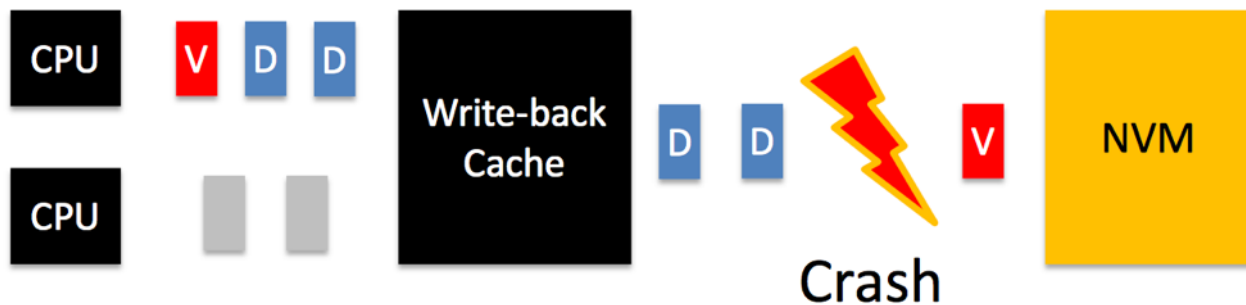
非易失性内存写入顺序

- 考虑持久性，写入顺序很重要

STORE data[0] = 0xF00D

STORE data[1] = 0xBEEF

STORE valid = 1



非易失性内存写入顺序

- 考虑持久性，写入顺序很重要

```
STORE data[0] = 0xF00D  
STORE data[1] = 0xBEEF  
STORE valid = 1
```

CPU

CPU

Write-back
Cache

data

valid

NVM

GARBAGE

1

错误的写入顺序在恢复时将垃圾数据视为有效数据！

思考🤔：有哪些解决方法？

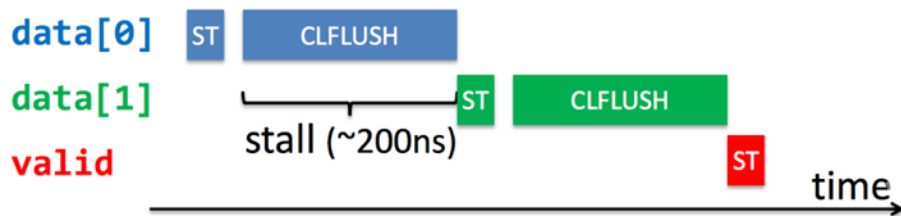
- 关闭CPU缓存？
- 使用Write-through模式的缓存？
- 每次写入后刷除整个缓存？

使用CLFLUSH保证顺序

- 使用CLFLUSH指令将数据逐出缓存，以保证顺序

```
STORE data[0] = 0xF00D
STORE data[1] = 0xBEEF
CLFLUSH data[0]
CLFLUSH data[1]
STORE valid = 1
```

- 但是CLFLUSH
 - 顺序执行，阻塞CPU流水线
 - 会将cacheline无效化

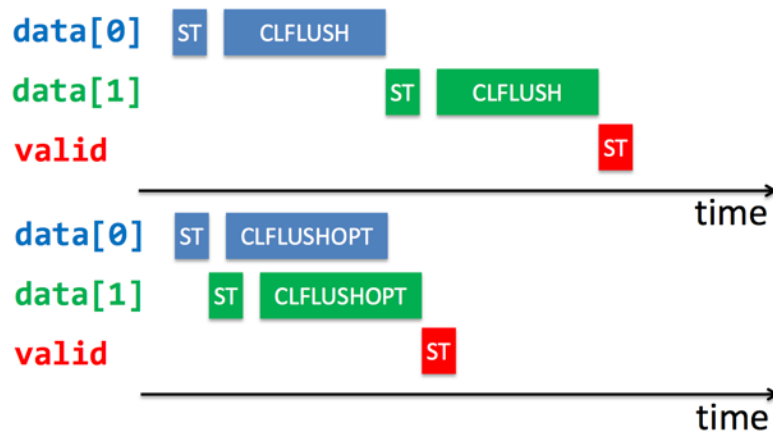



Intel x86 拓展指令集

- CLFLUSHOPT
 - 可并行执行的CLFLUSH
 - 需要sfence保证顺序

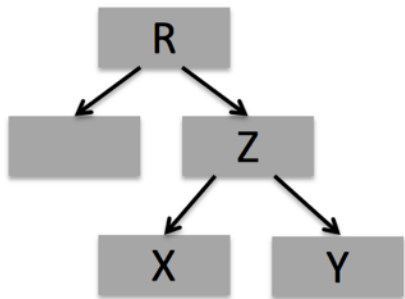
```
STORE data[0] = 0xF00D
STORE data[1] = 0xBEEF
CLFLUSHOPT data[0]
CLFLUSHOPT data[1]
SFENCE // explicit ordering point
STORE valid = 1
```

Implicit orderings

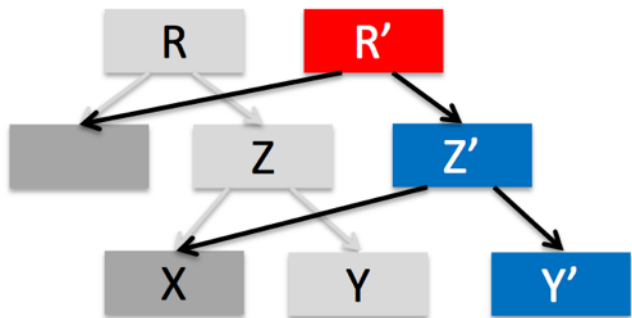


- CLWB
 - CLFLUSHOPT的基础上，不会将cacheline无效化

举例：NVM上的写时复制

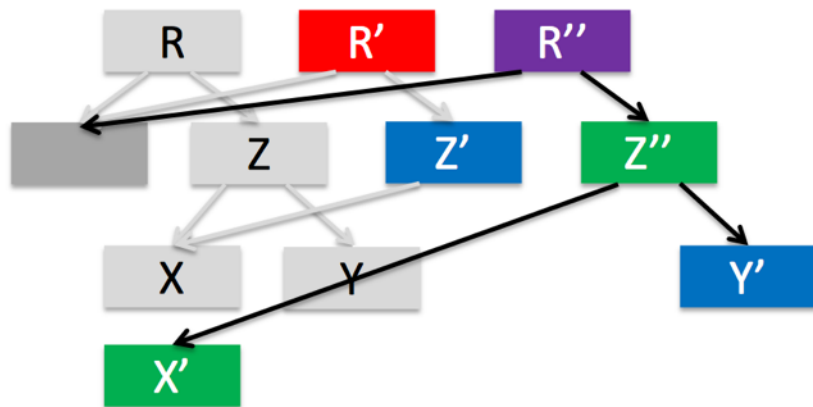


举例：NVM上的写时复制



STORE Y'
STORE Z'
CLWB Y'
CLWB Z'
SFENCE
STORE R'
CLWB R'

举例：NVM上的写时复制



STORE Y'
STORE Z'
CLWB Y'
CLWB Z'
SFENCE
STORE R'
CLWB R'
SFENCE
STORE X'
STORE Z'',
CLWB X'
CLWB Z'',
SFENCE
STORE R''
CLWB R''

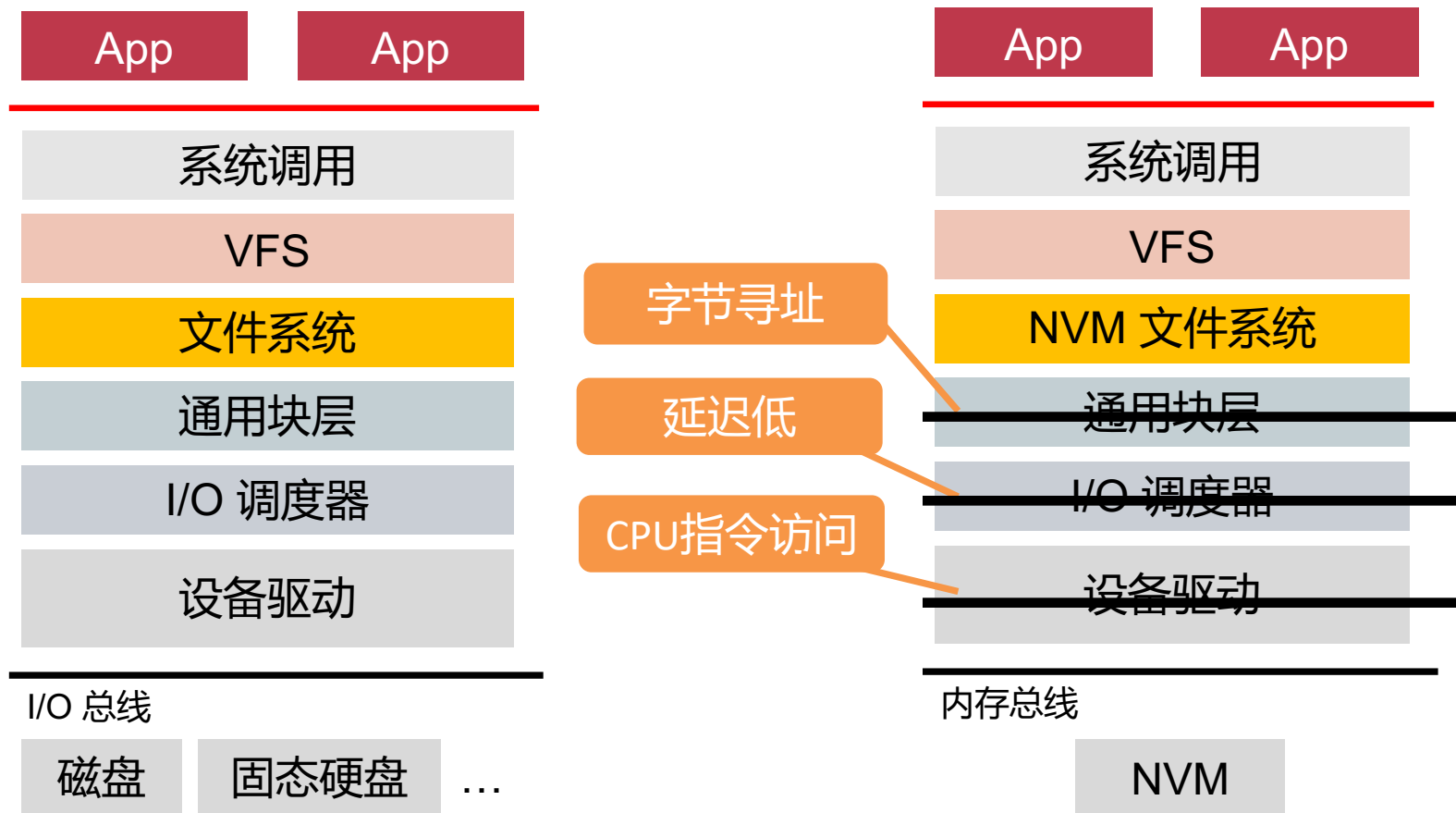
Non-volatile Memory File System

非易失性内存文件系统

非易失性内存改变存储栈



非易失性内存改变存储栈



一致性技术与非易失性内存文件系统

- 原子指令：ALL
- 写时复制：BPFS^[SOSP '09], PMFS^[EuroSys '14], NOVA^[FAST '16]
- 日志 (Journaling): PMFS, NOVA
- Log-structured: NOVA
- Soft updates: SoupFS^[USENIX ATC '17]

PMFS

- 为NVM和体系结构优化
 - 多种原子更新技术
- 允许应用直接访问NVM
 - DAX mmap
- Wild writes保护

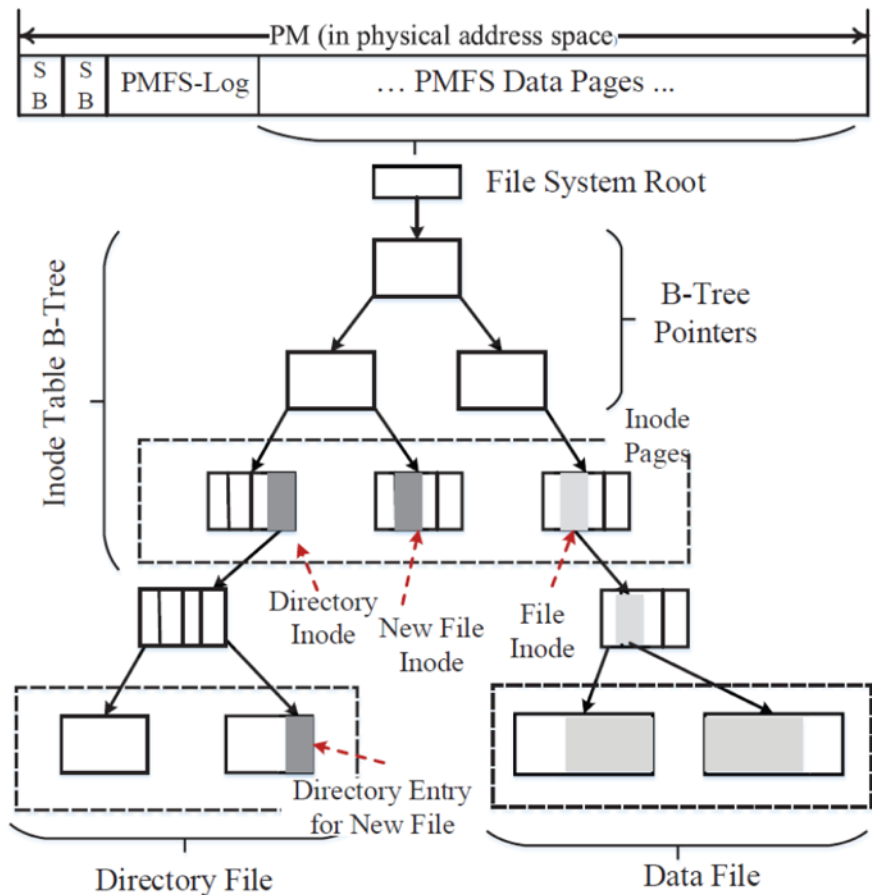


Figure 3: PMFS data layout

PMFS中的一致性保证

- 现有方法

- 写时复制 (Shadow Paging)
- 日志
- Log-structured updates

用于文件数据更新

用于元数据更新，如inode

- NVM专有的方法

- 原子指令更新

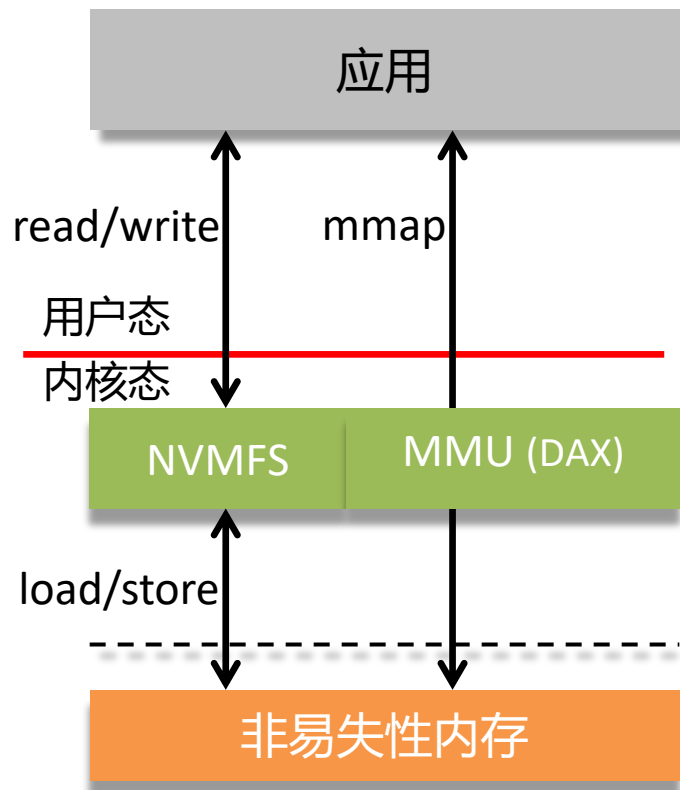
用于小修改

拓展的原子指令更新

- 8字节更新
 - CPU原本就支持8字节的原子更新
 - 更新inode的访问时间
- 16字节更新
 - 使用 *cmpxchg16b* 指令
 - 同时更新inode中的文件大小和修改时间
- 64字节更新
 - 使用硬件事务内存 (HTM)
 - 更新inode中的多个数据

让应用直接访问NVM

- DAX (direct access)
 - 文件mmap时
 - 通过建立页表映射
 - 将数据页映射给应用



防止NVM上的wild writes

- 程序Bug产生的wild writes会破坏NVM上的数据
- Supervisor Mode Access Protection (SMAP)
 - 防止内核错误地修改用户内存
- Write windows (PMFS提出)
 - 挂载时，NVM映射为只读
 - 写入时，CR0.WP临时设置为0，内核可以修改只读映射

	User	Kernel
User	Process Isolation	SMAP
Kernel	Privilege Levels	Write windows

Table 2: Overview of PM Write Protection