1. 写操作：之前文档里的考虑欠妥，这里需要重新思考。
2. 无论是更新写还是新写，都先放到缓存中，缓存满时，需要选择部分 subpages 拼接成 page 写回闪存，写回的目的地有两种选择，数据区和日志区；

写入日志区或者数据区的判断标准，有何不同

1. 我们有两个层次的映射，数据区的 page-level mapping table (LPN-PPN) 和日志区的 subpage-level mapping table (LSPN[4]-PPN)；

* 前者是按照 LPN 从 0 到 N-1 的顺序，可以看作一个数组 pagemap[N]，数组的索引表示 LPN，存储的内容是该 LPN 对应的 PPN；
* 后者是按照 LSPN 的更新顺序。

在 flashsim 里的 nand\_blk 结构里存在以下变量，

struct sect\_state sect[SECT\_NUM\_PER\_BLK];  // Logical Sector Number

记录每个物理扇区对应的逻辑扇区号，所以 subpage-level mapping 需要记录到 LSPN-PSPN 这一粒度，不能仅仅记录 LSPN-PPN，因为 一个 PPN 内存放多个 LSPN（8KB PPN 存放 2 个 LSPN，16KB PPN 存放 4 个 LSPN），比如 PPN 0 可能存放 LSPN 0（对应的扇区for(i=0; i<sect\_num\_per\_subpage;i++){N\* sect\_num\_per\_subpage+i）, LSPN 100, LSPN 200, LSPN 160，那么该 PPN 对应的逻辑扇区应该是 logical sector number [0~7], [100~107], [200~207], [160~167]。

也就是一个页4个子页，一个子页8个扇区，所以这边物理页（32个扇区）对应逻辑扇区 应该是[0~7],[800~807],[1600~1607], [1280~1287]

1. 多种情况的写入：

新写只会发生在 warmflash 的过程（在 disksim\_main.c）中，统计参数不计入正式的仿真。因为warmflash 的具体过程是将 trace 先跑一遍，所有请求都置为写，这样在正式的仿真过程中，mapping table就已经建立起来了，flash处于某种已使用的状态，而不是空盘，你可以假设逻辑空间上已存满数据，后续到来的写都是更新写。

假设page是16KB，

Warmflash过程：

* full page 新写：在缓存中淘汰某个脏LSPN 写回flash时（脏LSPN是指写过的，干净LSPN是指读操作读到缓存中未被更新过的，对于干净的LSPN，直接踢出缓存即可），发现与该 LSPN同属一个 page 的对齐、连续的4个subpages都在缓存中；

脏LSPN是指缓存中和闪存中数据不一致的，也就是写过的。缓存中是正确最新的，闪存中是无效过时的。

发现缓存中有连续属于一个page的4个子页，则将这4个子页合成一个页直接写回free\_page\_no（当前空闲页），这是full page 新写。

* partial page 新写：在缓存中淘汰某个脏LSPN 写回flash时，发现与该 LSPN同属一个 page 的4个subpages并不都在缓存中；

由于现在写的时候，会先对子页进行4KB对齐，所以不存在子页的部分写，只有页的部分写。

部分写 需要先读出该LSPN所在的源数据页（该子页在源数据页是无效的），将要写回的子页（缓存中新的内容）替换，如果该源数据页中还存在其他无效子页，是继续在缓存中寻找新子页还是直接不理 写回？这个可以斟酌下。

那么这里需要理一下 子页在 缓存、数据页、日志页中可能存在的状态和有效无效的情况，以及相应读写的数据流过程。

正式仿真过程：

* full page 更新：在缓存中淘汰某个 LSPN 写回flash时，发现与该 LSPN同属一个 page 的对齐、连续的4个subpages都在缓存中；
* partial page 更新：在缓存中淘汰某个脏LSPN 写回flash时，发现与该 LSPN同属一个 page 的4个subpages并不都在缓存中。

注意：这里面涉及到一些设计点。

1. 当从缓存淘汰某个脏LSPN（通常是LRU端的）时，有三种选择：

* 将缓存中与它同属一个page的所有LSPNs都拼接到一起写回flash；

如果能凑成full page 写，可以不经过日志区，直接写到数据区，节省日志区空间，减少日志页和数据页的合并；如果是partial page写，能够减少合并日志页和数据页过程中的数据页读，但是对于访问很热的 LSPN，如果提前将它写回flash，那么它很可能会马上又被更新，flash上刚写回的数据成了无效数据。

* 将LRU端的4个LSPNs拼凑成一个page写回；

日志页

* 介于前两者之间，比如对于同属一个page的4个LSPNs都在缓存，采取第一种；对于有3个LSPNs在缓存，先下发一次flash读，将缺少的LSPN读取出来，然后将4个LSPNs构成full page写，写到数据区，不经过日志区；对于有2个或1个LSPN在缓存，将LRU端的4个LSPNs拼凑成一个page写回。

1. 如果采取第一种或者第三种选择，那么对于缓存中非淘汰但是与淘汰的LSPN同属一页其它LSPNs，有两种选择：

* 将其状态从脏变为干净，不踢出缓存；
* 跟淘汰的 LSPN一样，踢出缓存。

可以对LRU链表 设置冷、热 区域，如果其他LSPN处于热区域，则不踢出缓存（缓存中是干净的，闪存中与其一致），如果其他LSPN处于冷区域，则跟淘汰的LSPN一样，踢出缓存。

1. 引入日志块存在一个严重的问题：

引入日志块，会导致日志块的回收，在极端情况下，回收开销会非常大，不可接受。假如每个block包含128个page，每个page包含4个minipage，每个minipage分属不同的 LPN，那么合并该日志块需要合并128×4 个逻辑数据页，至少产生128×4个页读取和写入，而且该日志块中的 数据不一定最新，即更新版本的数据在其它日志块中，那么还需要读取其它日志页，页读取操作更多。合并完成后，还需要擦除该日志块。

考虑到子页映射表记录了日志页的使用情况，可以不区分数据块和日志块，让数据页和日志页可以混合存储于一个块中，做日志页级别的回收，一方面降低合并开销，另一方面将日志页的合并与垃圾回收操作解耦。

除了两个映射表，还需要记录闪存上物理页的有效/无效状态，包括数据页的有效/无效，日志页内子页的有效/无效，这两个状态表也存放于缓存。

对于数据页，有效无效是一个页的状态，对于日志页，有效无效是一个子页为单位的状态。（在原有代码以扇区标记，但是同有效单位下扇区状态都是一致的）。

建议: 直接在上层维护一个状态表，大小为子页的数目，

从日志区的子页映射表可以推导有多少物理页用于做日志，限制这个页数即可限制子页映射表，也即日志区的大小。当日志页的数量到达最大阈值时，扫描整个子页映射表，首先寻找所有的数据都无效的 PPN，将这样的PPNs置为无效，然后在子映射表中删除这些PPNs对应的表项，即从日志区删除；若没有这样的日志页PPN， 则选择无效 LSPN最多的PPN或者最老的PPN。对该PPN中的每一个有效LSPN，找到与它对应的原 LPN，将原LPN的4个LSPNs都更新至最新版本，将该LPN写到新的数据页中，然后将原数据页PPN和从其它日志页读取的最新的 LSPNs置为无效。最后，将该日志页PPN置无效，删掉对应的子页映射条目。

日志页中有部分有效LSPN时，将有效数据写到新的数据页中，（合并日志页和数据页），然后将该日志页置无效。

设置一个计数器来记录日志页的数量，以此来限制日志区的大小。

子页映射表条目：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PPN | LSPN[0] | LSPN[1] | LSPN[2] | LSPN[3] |

并且是有一个新的日志页就添加新条目，大小MAP\_MAX来限制，所以日志页和子页映射表entry 是同增同减的，如果该entry 记录的4个LSPN的子页都是无效了，则该日志页无效，可以回收。