FTL Write Booster Design

一、功能说明

Write Booster Buffer 是 FTL 提供的一片高性能存储空间（SLC），在用户指定的特定数据写入时，使用该空间进行存储，以提供更好的存储速度和用户体验。

二、空间分配

目前，FTL 已有的空间分配策略为：优先为用户提供 SLC 空间(slc_cache & dynamic slc)，当 SLC 空间耗尽时，再使用 TLC 空间进行存储。

即，当前的空间分配策略不区分用户数据的类型，采用先来先分配的原则。

在开启 Write Booster 功能后，FTL 需要识别用户数据的类型，对于写入 booster buffer 中的数据，在其配置容量可用的情况下，需要使用 SLC 进行存储。

FTL 提供的 Write Booster 容量相关的接口:

1. 设置 write booster buffer 容量：

由用户配置，FE 通知 FTL， FTL 根据配置项中的最大 write booster buffer size 以及当前 pool 中实际可用的 SLC 空间，判决是否可以配置成功；

若剩余 SLC 空间满足需求，将该 SLC 空间配额预留给 writeBooster，优先使用 SLC_POOL 中的空间，然后使用 TLC POOL 中 dynamic SLC 的空间。剩余的空间为非 write booster 空间

（该空间中也可以包含 SLC）。

2. 查询 write booster buffer 当前容量：

FTL 支持查询当前 write booster buffer 的总体容量，默认情况下该容量等于配置容量。当 NAND 空间不足时，非 write booster 空间可以抢占 write booster 空间，

即， write booster buffer 的当前容量会相应减少，同时会减少同等容量的 write booster available buffer。

3. 查询 write booster buffer 可用容量：

FTL 支持查询 write booster buffer 可用容量，可用容量会随着写入 write booster buffer 的数据增多而相应的减少，同时也会受 2 中提到的因素减少。

当可用容量为 0 时，后续的 IO 将不再写入 write booster buffer。

4. 查询 write booster buffer life：

FTL 支持查询 write booster buffer 生命周期（百分比展示）。用 buffer 中已使用的 SLC 和 TLC 的各自生命周期加权后等到实时的生命周期。

三、 SPB 分配

目前，USER LU 只有一个 ASPB 用于提供 HOST IO 的写入。使能 Write Booster 功能后，存在 HOST IO 同时写入 Write Booster Buffer 空间和非 Write Booster Buffer 空间。

因此，USER LU 需要新增一个 ASPB 存储 Write Booster Buffer 的写入。则，相应需要新增的功能有：

1. 新增 Write Booster PageAllocate 管理：

用于管理 Write Booster ASPB 上的 page 分配；

2. 新增 Write Booster Spb apply 管理:

用于管理 Write Booster Spb 从 POOL 中的分配逻辑， 支持从 SLC POOL 和 TLC POOL 中进行 SPB 的分配

3. 新增 Write Booster Rlut 管理：

用于记录 Write Booster Aspb 的 P2L 信息。

四、IO 路径

1. HOST 写：

a. FE WriteNode 上新增标记，用于识别 Write Booster 写和非 Write Booster 写：

规则： 同一个 WriteNode 内只能有一种类型的 SubNode。

前后两个 WriteNode 如果为不同类型，前面一个 WriteNode 需要进行 force_Flush（设置为 FUA）

两种类型的 WriteNode 之前需要进行冲突检测

b. UserLu 根据 WriteNode 的类型，分别写入 WriteBooster 的 ASPB 和 normal ASPB，同时将 mapping 写入各自的 RLUT：

那么会存在一个问题：同一个 LAA 可能会同时存在于两个 RLUT entry 当中，就会涉及到新旧数据的问题。如果处理

该问题的方案如下：

1）、 WriteBooster 的 Rlut entry 直接写入 mapping;

2）、Normal 的 Rlut entry 在 insert mapping 时，需要 搜索 WriteBooster 的处于 busy 状态的 Rlut entry， 将所有相同的 LAA 都设置为无效值，

即需要保证 WriteBooster Rlut entry 中是最新的 mapping。

2. CONVERT

当 RLUT entry 写满，或者有 trim 命令时，会触发 RLUT entry 的 convert。 新增 WriteBooster Rlut entry 后，在触发 convert 流程时，需要同时

触发 WriteBooster Rlut entry 和对应的 Normal Rlut entry，且需要按照先执行 Normal Rlut entry 的 convert， 再执行 WriteBooster Rlut entry 的

convert 的顺序。

而当前 Rlut entry 的 partical-convert 逻辑只适用于 trim 触发的 convert， 并且 partical 与 full entry 的 convert 并没有前后关系。且当前的 full entry

之前的 convert 并没有要求顺序执行，且存在 full、convert 等多个 queue。 为了适应 WriteBooster Rlut entry 的需求，需要重构 RLUT entry 进行

convert 的逻辑：

1）去掉 Rlut full queue， partical queue， 保留 convert queue，添加 convert entry node 结构：

```
typedef struct _RlutConvertNode
{
    RlutEntry_t *Entry;
    U32  EntryStart;
    U32  EntryEnd;
    QTAILQ_ENTRY(_RlutConvertNode) link;
}RlutConvertNode;
```

Node 中包含待 convert 的 entry， 此次 convert 的范围。

同时， Rlut entry 中需要添加记录 convert 信息的结构：

```
typedef struct _RlutConvertInfo_t
{
    U8 ConvertIdx;
    U16 EntryStart[RLUT_ENTRY_MAX_CONVERT_NUM];
    U16 EntryEnd[RLUT_ENTRY_MAX_CONVERT_NUM];
    U32 ConvertSn[RLUT_ENTRY_MAX_CONVERT_NUM];
}RlutConvertInfo_t;
```

每个 Rlut entry 可以被触发多少 convert，需要记录多笔 convert 信息，使用 convertIdx 来记录；

EntryStart 和 EntryEnd 是每次 convert 的范围；

convertSn 是一个全局唯一的 SN，用于标记此次 convert 的序列，用于 recover 时，恢复 convert queue。

2） 当 WriteBooster Rlut entry 写满时，需要将 WriteBooster Rlut entry flush 到 NAND 上，

生成 WriteBooster Rlut entry 的 convert 节点后加入 convert queue，若 Normal Rlut entry 中存在未 convert 的 mapping， 需要

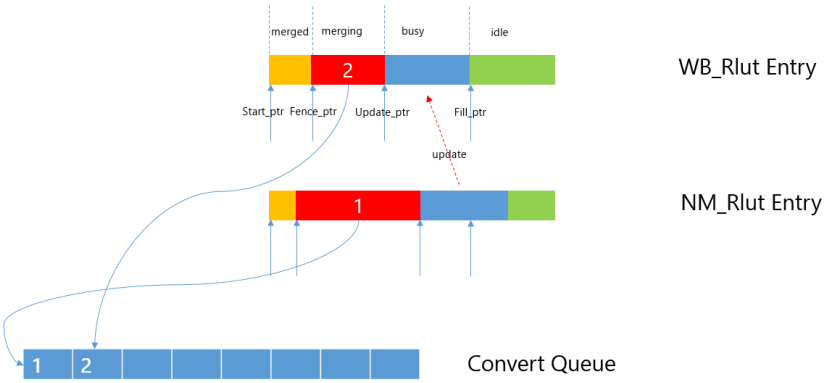
按照相同的操作先处理处理 Normal Rlut entry；

3） 当 Normal Rlut entry 写满时，需要将 Normal Rlut entry flush 到 NAND 上，生成 Normal Rlut entry 的 convert 节点后

加入 convert queue，若 WriteBooster Rlut entry 中存在未 convert 的 mapping， 需要按照相同的操作再处理 WriteBooster Rlut entry；

4）当收到 trim 命令时，需要将两种 Rlut entry 都加入到 convert queue 中。为避免大量 trim 命令触发添加过多的 convert node，

trim 命令触发 convert 的机制修改为当 convert queue 为空时，同时 trim record 不为 0，再执行该操作。



3. Host 读

Host 读执行 lookup mapping 操作时，查找 RLUT 的顺序修改为：

1） 查询 WriteBooster Rlut entry 的 busy 区域；

2） 查询 Normal Rlut entry 的 busy 区域；

3） 倒序查询 convert queue 中的 Rlut entry 的 merging 区域。