BTT - Block Translation Table

1. Introduction

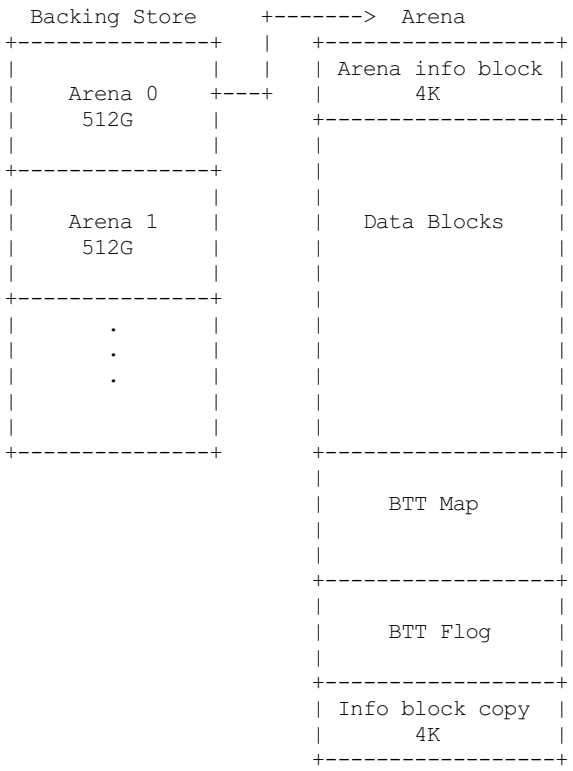
Persistent memory based storage is able to perform IO at byte (or more accurately, cache line) granularity. However, we often want to expose such storage as traditional block devices. The block drivers for persistent memory will do exactly this. However, they do not provide any atomicity guarantees. Traditional SSDs typically provide protection against torn sectors in hardware, using stored energy in capacitors to complete in-flight block writes, or perhaps in firmware. We don't have this luxury with persistent memory - if a write is in progress, and we experience a power failure, the block will contain a mix of old and new data. Applications may not be prepared to handle such a scenario.

The Block Translation Table (BTT) provides atomic sector update semantics for persistent memory devices, so that applications that rely on sector writes not being torn can continue to do so. The BTT manifests itself as a stacked block device, and reserves a portion of the underlying storage for its metadata. At the heart of it, is an indirection table that re-maps all the blocks on the volume. It can be thought of as an extremely simple file system that only provides atomic sector updates.

2. Static Layout

The underlying storage on which a BTT can be laid out is not limited in any way. The BTT, however, splits the available space into chunks of up to 512 GiB, called "Arenas".

Each arena follows the same layout for its metadata, and all references in an arena are internal to it (with the exception of one field that points to the next arena). The following depicts the "On-disk" metadata layout:



3. Theory of Operation

a. The BTT Map

The map is a simple lookup/indirection table that maps an LBA to an internal block. Each map entry is 32 bits. The two most significant bits are special flags, and the remaining form the internal block number.

Bit	Description

Bit	Description

31 - 30	Error and Zero flags - Used in the following way:
	
	`== ==  ==================================================`
	`31 30  Description`
	`== ==  ==================================================`
	`0  0   Initial state. Reads return zeroes; Premap = Postmap`
	`0  1   Zero state: Reads return zeroes`
	`1  0   Error state: Reads fail; Writes clear 'E' bit`
	`1  1   Normal Block â€“ has valid postmap`
	`== ==  ==================================================`

29 - 0	Mappings to internal 'postmap' blocks

Some of the terminology that will be subsequently used:

External LBA	LBA as made visible to upper layers.
ABA	Arena Block Address - Block offset/number within an arena
Premap ABA	The block offset into an arena, which was decided upon by range checking the External LBA
Postmap ABA	The block number in the "Data Blocks" area obtained after indirection from the map
nfree	The number of free blocks that are maintained at any given time. This is the number of concurrent writes that can happen to the arena.

For example, after adding a BTT, we surface a disk of 1024G. We get a read for the external LBA at 768G. This falls into the second arena, and of the 512G worth of blocks that this arena contributes, this block is at 256G. Thus, the premap ABA is 256G. We now refer to the map, and find out the mapping for block 'X' (256G) points to block 'Y', say '64'. Thus the postmap ABA is 64.

b. The BTT Flog

The BTT provides sector atomicity by making every write an "allocating write", i.e. Every write goes to a "free" block. A running list of free blocks is maintained in the form of the BTT flog. 'Flog' is a combination of the words "free list" and "log". The flog contains 'nfree' entries, and an entry contains:

lba	The premap ABA that is being written to
old_map	The old postmap ABA - after 'this' write completes, this will be a free block.
new_map	The new postmap ABA. The map will up updated to reflect this lba->postmap_aba mapping, but we log it here in case we have to recover.
seq	Sequence number to mark which of the 2 sections of this flog entry is valid/newest. It cycles between 01->10->11->01 (binary) under normal operation, with 00 indicating an uninitialized state.
lba'	alternate lba entry
old_map'	alternate old postmap entry
new_map'	alternate new postmap entry
seq'	alternate sequence number.

Each of the above fields is 32-bit, making one entry 32 bytes. Entries are also padded to 64 bytes to avoid cache line sharing or aliasing. Flog updates are done such that for any entry being written, it: a. overwrites the 'old' section in the entry based on sequence numbers b. writes the 'new' section such that the sequence number is written last.

c. The concept of lanes

While 'nfree' describes the number of concurrent IOs an arena can process concurrently, 'nlanes' is the number of IOs the BTT device as a whole can process:

```
nlanes = min(nfree, num_cpus)
```

A lane number is obtained at the start of any IO, and is used for indexing into all the on-disk and in-memory data structures for the duration of the IO. If there are more CPUs than the max number of available lanes, than lanes are protected by spinlocks.

d. In-memory data structure: Read Tracking Table (RTT)

Consider a case where we have two threads, one doing reads and the other, writes. We can hit a condition where the writer thread grabs a free block to do a new IO, but the (slow) reader thread is still reading from it. In other words, the reader consulted a map entry, and started reading the corresponding block. A writer started writing to the same external LBA, and finished the write updating the map for that external LBA to point to its new postmap ABA. At this point the internal, postmap block that the reader is (still) reading has been inserted into the list of free blocks. If another write comes in for the same LBA, it can grab this free block, and start writing to it, causing the reader to read incorrect data. To prevent this, we introduce the RTT.

The RTT is a simple, per arena table with 'nfree' entries. Every reader inserts into rtt[lane_number], the postmap ABA it is reading, and clears it after the read is complete. Every writer thread, after grabbing a free block, checks the RTT for its presence. If the postmap free block is in the RTT, it waits till the reader clears the RTT entry, and only then starts writing to it.

e. In-memory data structure: map locks

Consider a case where two writer threads are writing to the same LBA. There can be a race in the following sequence of steps:

```
free[lane] = map[premap_aba]
map[premap_aba] = postmap_aba
```

Both threads can update their respective free[lane] with the same old, freed postmap_aba. This has made the layout inconsistent by losing a free entry, and at the same time, duplicating another free entry for two lanes.

To solve this, we could have a single map lock (per arena) that has to be taken before performing the above sequence, but we feel that could be too contentious. Instead we use an array of (nfree) map_locks that is indexed by (premap_aba modulo nfree).

f. Reconstruction from the Flog

On startup, we analyze the BTT flog to create our list of free blocks. We walk through all the entries, and for each lane, of the set of two possible 'sections', we always look at the most recent one only (based on the sequence number). The reconstruction rules/steps are simple:

- Read map[log_entry.lba].
- If log_entry.new matches the map entry, then log_entry.old is free.
- If log_entry.new does not match the map entry, then log_entry.new is free. (This case can only be caused by power-fails/unsafe shutdowns)

g. Summarizing - Read and Write flows

Read:

1. Convert external LBA to arena number + pre-map ABA
2. Get a lane (and take lane_lock)
3. Read map to get the entry for this pre-map ABA
4. Enter post-map ABA into RTT[lane]
5. If TRIM flag set in map, return zeroes, and end IO (go to step 8)
6. If ERROR flag set in map, end IO with EIO (go to step 8)
7. Read data from this block
8. Remove post-map ABA entry from RTT[lane]
9. Release lane (and lane_lock)

Write:

1. Convert external LBA to Arena number + pre-map ABA
2. Get a lane (and take lane_lock)
3. Use lane to index into in-memory free list and obtain a new block, next flog index, next sequence number
4. Scan the RTT to check if free block is present, and spin/wait if it is.
5. Write data to this free block
6. Read map to get the existing post-map ABA entry for this pre-map ABA
7. Write flog entry: [premap_aba / old postmap_aba / new postmap_aba / seq_num]
8. Write new post-map ABA into map.
9. Write old post-map entry into the free list
10. Calculate next sequence number and write into the free list entry
11. Release lane (and lane_lock)

4. Error Handling

An arena would be in an error state if any of the metadata is corrupted irrecoverably, either due to a bug or a media error. The following conditions indicate an error:

- Info block checksum does not match (and recovering from the copy also fails)
- All internal available blocks are not uniquely and entirely addressed by the sum of mapped blocks and free blocks (from the BTT flog).
- Rebuilding free list from the flog reveals missing/duplicate/impossible entries
- A map entry is out of bounds

If any of these error conditions are encountered, the arena is put into a read only state using a flag in the info block.

5. Usage

The BTT can be set up on any disk (namespace) exposed by the libnvdimm subsystem (pmem, or blk mode). The easiest way to set up such a namespace is using the 'ndctl' utility [1]:

For example, the ndctl command line to setup a btt with a 4k sector size is:

```
ndctl create-namespace -f -e namespace0.0 -m sector -l 4k
```

See ndctl create-namespace --help for more options.

[1]: https://github.com/pmem/ndctl