Guidance for writing policies

Try to keep transactionality out of it. The core is careful to avoid asking about anything that is migrating. This is a pain, but makes it easier to write the policies.

Mappings are loaded into the policy at construction time.

Every bio that is mapped by the target is referred to the policy. The policy can return a simple HIT or MISS or issue a migration.

Currently there's no way for the policy to issue background work, e.g. to start writing back dirty blocks that are going to be evicted soon.

Because we map bios, rather than requests it's easy for the policy to get fooled by many small bios. For this reason the core target issues periodic ticks to the policy. It's suggested that the policy doesn't update states (eg, hit counts) for a block more than once for each tick. The core ticks by watching bios complete, and so trying to see when the io scheduler has let the ios run.

Overview of supplied cache replacement policies

multiqueue (mq)

This policy is now an alias for smq (see below).

The following tunables are accepted, but have no effect:

```
'sequential_threshold <#nr_sequential_ios>'
'random_threshold <#nr_random_ios>'
'read_promote_adjustment <value>'
'write_promote_adjustment <value>'
'discard_promote_adjustment <value>'
```

Stochastic multiqueue (smq)

This policy is the default.

The stochastic multi-queue (smq) policy addresses some of the problems with the multiqueue (mq) policy.

The smq policy (vs mq) offers the promise of less memory utilization, improved performance and increased adaptability in the face of changing workloads. smq also does not have any cumbersome tuning knobs.

Users may switch from "mq" to "smq" simply by appropriately reloading a DM table that is using the cache target. Doing so will cause all of the mq policy's hints to be dropped. Also, performance of the cache may degrade slightly until smq recalculates the origin device's hotspots that should be cached.

Memory usage

The mq policy used a lot of memory; 88 bytes per cache block on a 64 bit machine.

smq uses 28bit indexes to implement its data structures rather than pointers. It avoids storing an explicit hit count for each block. It has a 'hotspot' queue, rather than a pre-cache, which uses a quarter of the entries (each hotspot block covers a larger area than a single cache block).

All this means smq uses ~25bytes per cache block. Still a lot of memory, but a substantial improvement nontheless.

Level balancing

mq placed entries in different levels of the multiqueue structures based on their hit count (~ln(hit count)). This meant the bottom levels generally had the most entries, and the top ones had very few. Having unbalanced levels like this reduced the efficacy of the multiqueue.

smq does not maintain a hit count, instead it swaps hit entries with the least recently used entry from the level above. The overall ordering being a side effect of this stochastic process. With this scheme we can decide how many entries occupy each multiqueue level, resulting in better promotion/demotion decisions.

Adaptability: The mq policy maintained a hit count for each cache block. For a different block to get promoted to the cache its hit count has to exceed the lowest currently in the cache. This meant it could take a long time for the cache to adapt between varying IO patterns.

smq doesn't maintain hit counts, so a lot of this problem just goes away. In addition it tracks performance of the hotspot queue, which is used to decide which blocks to promote. If the hotspot queue is performing badly then it starts moving entries more quickly between levels. This lets it adapt to new IO patterns very quickly.

Performance

Testing smq shows substantially better performance than mq.

cleaner

The cleaner writes back all dirty blocks in a cache to decommission it.

Examples

The syntax for a table is:

```
cache <metadata dev> <cache dev> <origin dev> <block size>
<#feature_args> [<feature arg>]*
<policy> <#policy_args> [<policy arg>]*
```

The syntax to send a message using the dmsetup command is:

```
dmsetup message <mapped device> 0 sequential_threshold 1024
dmsetup message <mapped device> 0 random_threshold 8
```

Using dmsetup:

```
dmsetup create blah --table "0 268435456 cache /dev/sdb /dev/sdc \
    /dev/sdd 512 0 mq 4 sequential_threshold 1024 random_threshold 8"
creates a 128GB large mapped device named 'blah' with the
sequential threshold set to 1024 and the random_threshold set to 8.
```