A block layer cache (bcache)

Say you've got a big slow raid 6, and an ssd or three. Wouldn't it be nice if you could use them as cache... Hence bcache.

The bcache wiki can be found at:
https://bcache.evilpiepirate.org
This is the git repository of bcache-tools:
https://git.kernel.org/pub/scm/linux/kernel/git/colyli/bcache-tools.git/
The latest bcache kernel code can be found from mainline Linux kernel:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/

It's designed around the performance characteristics of SSDs - it only allocates in erase block sized buckets, and it uses a hybrid btree/log to track cached extents (which can be anywhere from a single sector to the bucket size). It's designed to avoid random writes at all costs; it fills up an erase block sequentially, then issues a discard before reusing it.

Both writethrough and writeback caching are supported. Writeback defaults to off, but can be switched on and off arbitrarily at runtime. Bcache goes to great lengths to protect your data - it reliably handles unclean shutdown. (It doesn't even have a notion of a clean shutdown; bcache simply doesn't return writes as completed until they're on stable storage).

Writeback caching can use most of the cache for buffering writes - writing dirty data to the backing device is always done sequentially, scanning from the start to the end of the index.

Since random IO is what SSDs excel at, there generally won't be much benefit to caching large sequential IO. Bcache detects sequential IO and skips it; it also keeps a rolling average of the IO sizes per task, and as long as the average is above the cutoff it will skip all IO from that task - instead of caching the first 512k after every seek. Backups and large file copies should thus entirely bypass the cache.

In the event of a data IO error on the flash it will try to recover by reading from disk or invalidating cache entries. For unrecoverable errors (meta data or dirty data), caching is automatically disabled; if dirty data was present in the cache it first disables writeback caching and waits for all dirty data to be flushed.

Getting started: You'll need bcache util from the bcache-tools repository. Both the cache device and backing device must be formatted before use:

```
bcache make -B /dev/sdb
bcache make -C /dev/sdc
```

bcache make has the ability to format multiple devices at the same time - if you format your backing devices and cache device at the same time, you won't have to manually attach:

```
bcache make -B /dev/sda /dev/sdb -C /dev/sdc
```

If your bcache-tools is not updated to latest version and does not have the unified *bcache* utility, you may use the legacy *make-bcache* utility to format bcache device with same -B and -C parameters.

bcache-tools now ships udev rules, and bcache devices are known to the kernel immediately. Without udev, you can manually register devices like this:

```
echo /dev/sdb > /sys/fs/bcache/register
echo /dev/sdc > /sys/fs/bcache/register
```

Registering the backing device makes the bcache device show up in /dev; you can now format it and use it as normal. But the first time using a new bcache device, it'll be running in passthrough mode until you attach it to a cache. If you are thinking about using bcache later, it is recommended to setup all your slow devices as bcache backing devices without a cache, and you can choose to add a caching device later. See 'ATTACHING' section below.

The devices show up as:

```
/dev/bcache<N>
```

As well as (with udev):

```
/dev/bcache/by-uuid/<uuid>
/dev/bcache/by-label/<label>
```

To get started:

```
mkfs.ext4 /dev/bcache0
mount /dev/bcache0 /mnt
```

You can control bcache devices through sysfs at /sys/block/bcache<N>/bcache . You can also control them through /sys/fs//bcache/<cset-uuid>/ .

Cache devices are managed as sets; multiple caches per set isn't supported yet but will allow for mirroring of metadata and dirty data in the future. Your new cache set shows up as /sys/fs/bcache/<UUID>

Attaching

After your cache device and backing device are registered, the backing device must be attached to your cache set to enable caching. Attaching a backing device to a cache set is done thusly, with the UUID of the cache set in /sys/fs/bcache:

```
echo <CSET-UUID> > /sys/block/bcache0/bcache/attach
```

This only has to be done once. The next time you reboot, just reregister all your bcache devices. If a backing device has data in a cache somewhere, the /dev/bcache<N> device won't be created until the cache shows up - particularly important if you have writeback caching turned on.

If you're booting up and your cache device is gone and never coming back, you can force run the backing device:

```
echo 1 > /sys/block/sdb/bcache/running
```

(You need to use /sys/block/sdb (or whatever your backing device is called), not /sys/block/bcache0, because bcache0 doesn't exist yet. If you're using a partition, the bcache directory would be at /sys/block/sdb/sdb2/bcache)

The backing device will still use that cache set if it shows up in the future, but all the cached data will be invalidated. If there was dirty data in the cache, don't expect the filesystem to be recoverable - you will have massive filesystem corruption, though ext4's fsck does work miracles.

Error Handling

Bcache tries to transparently handle IO errors to/from the cache device without affecting normal operation; if it sees too many errors (the threshold is configurable, and defaults to 0) it shuts down the cache device and switches all the backing devices to passthrough mode.

- For reads from the cache, if they error we just retry the read from the backing device.
- For writethrough writes, if the write to the cache errors we just switch to invalidating the data at that lba in the cache (i.e. the same thing we do for a write that bypasses the cache)
- For writeback writes, we currently pass that error back up to the filesystem/userspace. This could be improved - we could retry it as a write that skips the cache so we don't have to error the write.
- When we detach, we first try to flush any dirty data (if we were running in writeback mode). It currently doesn't do anything intelligent if it fails to read some of the dirty data, though.

Howto/cookbook

A. Starting a bcache with a missing caching device

If registering the backing device doesn't help, it's already there, you just need to force it to run without the cache:

```
host:~# echo /dev/sdb1 > /sys/fs/bcache/register
```

```
[  119.844831] bcache: register_bcache() error opening /dev/sdb1: device already registered
```

Next, you try to register your caching device if it's present. However if it's absent, or registration fails for some reason, you can still start your bcache without its cache, like so:

```
host:/sys/block/sdb/sdb1/bcache# echo 1 > running
```

Note that this may cause data loss if you were running in writeback mode.

B. Bcache does not find its cache:

```
host:/sys/block/md5/bcache# echo 0226553a-37cf-41d5-b3ce-8b1e944543a8 > attach
[ 1933.455082] bcache: bch_cached_dev_attach() Couldn't find uuid for md5 in set
[ 1933.478179] bcache: __cached_dev_store() Can't attach 0226553a-37cf-41d5-b3ce-8b1e944543a8
[ 1933.478179] : cache set not found
```

In this case, the caching device was simply not registered at boot or disappeared and came back, and needs to be (re-)registered:

```
host:/sys/block/md5/bcache# echo /dev/sdh2 > /sys/fs/bcache/register
```

C. Corrupt bcache crashes the kernel at device registration time:

This should never happen. If it does happen, then you have found a bug! Please report it to the bcache development list: linux-bcache@vger.kernel.org

Be sure to provide as much information that you can including kernel dmesg output if available so that we may assist.

D. Recovering data without bcache:

If bcache is not available in the kernel, a filesystem on the backing device is still available at an 8KiB offset. So either via a loopdev of the backing device created with --offset 8K, or any value defined by --data-offset when you originally formatted bcache with *bcache make*.

For example:

```
losetup -o 8192 /dev/loop0 /dev/your_bcache_backing_dev
```

This should present your unmodified backing device data in /dev/loop0

If your cache is in writethrough mode, then you can safely discard the cache device without loosing data.

E. Wiping a cache device

```
host:~# wipefs -a /dev/sdh2
16 bytes were erased at offset 0x1018 (bcache)
they were: c6 85 73 f6 4e 1a 45 ca 82 65 f5 7f 48 ba 6d 81
```

After you boot back with bcache enabled, you recreate the cache and attach it:

```
host:~# bcache make -C /dev/sdh2
UUID:                   7be7e175-8f4c-4f99-94b2-9c904d227045
Set UUID:               5bc072a8-ab17-446d-9744-e247949913c1
version:                0
nbuckets:               106874
block_size:             1
bucket_size:            1024
nr_in_set:              1
nr_this_dev:            0
first_bucket:           1
[  650.511912] bcache: run_cache_set() invalidating existing data
[  650.549228] bcache: register_cache() registered cache device sdh2
```

start backing device with missing cache:

```
host:/sys/block/md5/bcache# echo 1 > running
```

attach new cache:

```
host:/sys/block/md5/bcache# echo 5bc072a8-ab17-446d-9744-e247949913c1 > attach
[  865.276616] bcache: bch_cached_dev_attach() Caching md5 as bcache0 on set 5bc072a8-ab17-446d-9744-e247949913c1
```

F. Remove or replace a caching device:

```
host:/sys/block/sda/sda7/bcache# echo 1 > detach
[  695.872542] bcache: cached_dev_detach_finish() Caching disabled for sda7

host:~# wipefs -a /dev/nvme0n1p4
wipefs: error: /dev/nvme0n1p4: probing initialization failed: Device or resource busy
Ooops, it's disabled, but not unregistered, so it's still protected
```

We need to go and unregister it:

```
host:/sys/fs/bcache/b7ba27a1-2398-4649-8ae3-0959f57ba128# ls -l cache0
lrwxrwxrwx 1 root root 0 Feb 25 18:33 cache0 -> ../../../devices/pci0000:00/0000:00:1d.0/0000:70:00.0/nvme/nvme0/nvme0n1/nvme0n1p4/bcache
host:/sys/fs/bcache/b7ba27a1-2398-4649-8ae3-0959f57ba128# echo 1 > stop
kernel: [  917.041908] bcache: cache_set_free() Cache set b7ba27a1-2398-4649-8ae3-0959f57ba128 unregistered
```

Now we can wipe it:

```
host:~# wipefs -a /dev/nvme0n1p4
/dev/nvme0n1p4: 16 bytes were erased at offset 0x00001018 (bcache): c6 85 73 f6 4e 1a 45 ca 82 65 f5 7f 48 ba 6d 81
```

G. dm-crypt and bcache

First setup bcache unencrypted and then install dmcrypt on top of /dev/bcache<N> This will work faster than if you dmcrypt both the backing and caching devices and then install bcache on top. [benchmarks?]

H. Stop/free a registered bcache to wipe and/or recreate it

Suppose that you need to free up all bcache references so that you can fdisk run and re-register a changed partition table, which won't work if there are any active backing or caching devices left on it:

1. Is it present in /dev/bcache* ? (there are times where it won't be)

   If so, it's easy:

   ```
   host:/sys/block/bcache0/bcache# echo 1 > stop
   ```

2. But if your backing device is gone, this won't work:

   ```
   host:/sys/block/bcache0# cd bcache
   bash: cd: bcache: No such file or directory
   ```

   In this case, you may have to unregister the dmcrypt block device that references this bcache to free it up:

   ```
   host:~# dmsetup remove oldds1
   bcache: bcache_device_free() bcache0 stopped
   bcache: cache_set_free() Cache set 5bc072a8-ab17-446d-9744-e247949913c1 unregistered
   ```

   This causes the backing bcache to be removed from /sys/fs/bcache and then it can be reused. This would be true of any block device stacking where bcache is a lower device.

3. In other cases, you can also look in /sys/fs/bcache/:

   ```
   host:/sys/fs/bcache# ls -l */{cache?,bdev?}
   lrwxrwxrwx 1 root root 0 Mar  5 09:39 0226553a-37cf-41d5-b3ce-8b1e944543a8/bdev1 -> ../../../devices/virtual/block/dm-1/bcache/
   lrwxrwxrwx 1 root root 0 Mar  5 09:39 0226553a-37cf-41d5-b3ce-8b1e944543a8/cache0 -> ../../../devices/virtual/block/dm-4/bcache/
   lrwxrwxrwx 1 root root 0 Mar  5 09:39 5bc072a8-ab17-446d-9744-e247949913c1/cache0 -> ../../../devices/pci0000:00/0000:00:01.0/0000
   ```

   The device names will show which UUID is relevant, cd in that directory and stop the cache:

   ```
   host:/sys/fs/bcache/5bc072a8-ab17-446d-9744-e247949913c1# echo 1 > stop
   ```

   This will free up bcache references and let you reuse the partition for other purposes.

Troubleshooting performance

Bcache has a bunch of config options and tunables. The defaults are intended to be reasonable for typical desktop and server workloads, but they're not what you want for getting the best possible numbers when benchmarking.

- Backing device alignment

  The default metadata size in bcache is 8k. If your backing device is RAID based, then be sure to align this by a multiple of your stride width using *bcache make --data-offset*. If you intend to expand your disk array in the future, then multiply a series of primes by your raid stripe size to get the disk multiples that you would like.

  For example: If you have a 64k stripe size, then the following offset would provide alignment for many common RAID5 data spindle counts:

  ```
  64k * 2*2*2*3*3*5*7 bytes = 161280k
  ```

  That space is wasted, but for only 157.5MB you can grow your RAID 5 volume to the following data-spindle counts without re-aligning:

  ```
  3,4,5,6,7,8,9,10,12,14,15,18,20,21 ...
  ```

- Bad write performance

  If write performance is not what you expected, you probably wanted to be running in writeback mode, which isn't the default (not due to a lack of maturity, but simply because in writeback mode you'll lose data if something happens to your SSD):

  ```
  # echo writeback > /sys/block/bcache0/bcache/cache_mode
  ```

- Bad performance, or traffic not going to the SSD that you'd expect

  By default, bcache doesn't cache everything. It tries to skip sequential IO - because you really want to be caching the random IO, and if you copy a 10 gigabyte file you probably don't want that pushing 10 gigabytes of randomly accessed data out of your cache.

  But if you want to benchmark reads from cache, and you start out with fio writing an 8 gigabyte test file - so you want to disable that:

  ```
  # echo 0 > /sys/block/bcache0/bcache/sequential_cutoff
  ```

  To set it back to the default (4 mb), do:

  ```
  # echo 4M > /sys/block/bcache0/bcache/sequential_cutoff
  ```

- Traffic's still going to the spindle/still getting cache misses

  In the real world, SSDs don't always keep up with disks - particularly with slower SSDs, many disks being cached by one SSD, or mostly sequential IO. So you want to avoid being bottlenecked by the SSD and having it slow everything down.

  To avoid that bcache tracks latency to the cache device, and gradually throttles traffic if the latency exceeds a threshold (it does this by cranking down the sequential bypass).

  You can disable this if you need to by setting the thresholds to 0:

  ```
  # echo 0 > /sys/fs/bcache/<cache set>/congested_read_threshold_us
  # echo 0 > /sys/fs/bcache/<cache set>/congested_write_threshold_us
  ```

  The default is 2000 us (2 milliseconds) for reads, and 20000 for writes.

- Still getting cache misses, of the same data

  One last issue that sometimes trips people up is actually an old bug, due to the way cache coherency is handled for cache misses. If a btree node is full, a cache miss won't be able to insert a key for the new data and the data won't be written to the cache.

  In practice this isn't an issue because as soon as a write comes along it'll cause the btree node to be split, and you need almost no write traffic for this to not show up enough to be noticeable (especially since bcache's btree nodes are huge and index large regions of the device). But when you're benchmarking, if you're trying to warm the cache by reading a bunch of data and there's no other traffic - that can be a problem.

  Solution: warm the cache by doing writes, or use the testing branch (there's a fix for the issue there).

Sysfs - backing device

Available at /sys/block/<bdev>/bcache, /sys/block/bcache*/bcache and (if attached) /sys/fs/bcache/<cset-uuid>/bdev*

attach

Echo the UUID of a cache set to this file to enable caching.

cache_mode

Can be one of either writethrough, writeback, writearound or none.

clear_stats

Writing to this file resets the running total stats (not the day/hour/5 minute decaying versions).

detach

Write to this file to detach from a cache set. If there is dirty data in the cache, it will be flushed first.

dirty_data

Amount of dirty data for this backing device in the cache. Continuously updated unlike the cache set's version, but may be slightly off.

label

Name of underlying device.

readahead

Size of readahead that should be performed. Defaults to 0. If set to e.g. 1M, it will round cache miss reads up to that size, but without overlapping existing cache entries.

running

1 if bcache is running (i.e. whether the /dev/bcache device exists, whether it's in passthrough mode or caching).

sequential_cutoff

A sequential IO will bypass the cache once it passes this threshold; the most recent 128 IOs are tracked so sequential IO can be detected even when it isn't all done at once.

sequential_merge

If non zero, bcache keeps a list of the last 128 requests submitted to compare against all new requests to determine which new requests are sequential continuations of previous requests for the purpose of determining sequential cutoff. This is necessary if the sequential cutoff value is greater than the maximum acceptable sequential size for any single request.

state

The backing device can be in one of four different states:

no cache: Has never been attached to a cache set.

clean: Part of a cache set, and there is no cached dirty data.

dirty: Part of a cache set, and there is cached dirty data.

inconsistent: The backing device was forcibly run by the user when there was dirty data cached but the cache set was

unavailable; whatever data was on the backing device has likely been corrupted.

stop
: Write to this file to shut down the bcache device and close the backing device.

writeback_delay
: When dirty data is written to the cache and it previously did not contain any, waits some number of seconds before initiating writeback. Defaults to 30.

writeback_percent
: If nonzero, bcache tries to keep around this percentage of the cache dirty by throttling background writeback and using a PD controller to smoothly adjust the rate.

writeback_rate
: Rate in sectors per second - if writeback_percent is nonzero, background writeback is throttled to this rate. Continuously adjusted by bcache but may also be set by the user.

writeback_running
: If off, writeback of dirty data will not take place at all. Dirty data will still be added to the cache until it is mostly full; only meant for benchmarking. Defaults to on.

Sysfs - backing device stats

There are directories with these numbers for a running total, as well as versions that decay over the past day, hour and 5 minutes; they're also aggregated in the cache set directory as well.

bypassed
: Amount of IO (both reads and writes) that has bypassed the cache

cache_hits, cache_misses, cache_hit_ratio
: Hits and misses are counted per individual IO as bcache sees them; a partial hit is counted as a miss.

cache_bypass_hits, cache_bypass_misses
: Hits and misses for IO that is intended to skip the cache are still counted, but broken out here.

cache_miss_collisions
: Counts instances where data was going to be inserted into the cache from a cache miss, but raced with a write and data was already present (usually 0 since the synchronization for cache misses was rewritten)

cache_readaheads
: Count of times readahead occurred.

Sysfs - cache set

Available at /sys/fs/bcache/<cset-uuid>

average_key_size
: Average data per key in the btree.

bdev<0..n>
: Symlink to each of the attached backing devices.

block_size
: Block size of the cache devices.

btree_cache_size
: Amount of memory currently used by the btree cache

bucket_size
: Size of buckets

cache<0..n>
: Symlink to each of the cache devices comprising this cache set.

cache_available_percent
: Percentage of cache device which doesn't contain dirty data, and could potentially be used for writeback. This doesn't mean this space isn't used for clean cached data; the unused statistic (in priority_stats) is typically much lower.

clear_stats
: Clears the statistics associated with this cache

dirty_data
: Amount of dirty data is in the cache (updated when garbage collection runs).

flash_vol_create
: Echoing a size to this file (in human readable units, k/M/G) creates a thinly provisioned volume backed by the cache set.

io_error_halflife, io_error_limit
: These determines how many errors we accept before disabling the cache. Each error is decayed by the half life (in # ios). If the decaying count reaches io_error_limit dirty data is written out and the cache is disabled.

journal_delay_ms
: Journal writes will delay for up to this many milliseconds, unless a cache flush happens sooner. Defaults to 100.

root_usage_percent
: Percentage of the root btree node in use. If this gets too high the node will split, increasing the tree depth.

stop
: Write to this file to shut down the cache set - waits until all attached backing devices have been shut down.

tree_depth
: Depth of the btree (A single node btree has depth 0).

unregister
: Detaches all backing devices and closes the cache devices; if dirty data is present it will disable writeback caching and wait for it to be flushed.

Sysfs - cache set internal

This directory also exposes timings for a number of internal operations, with separate files for average duration, average frequency, last occurrence and max duration: garbage collection, btree read, btree node sorts and btree splits.

active_journal_entries
: Number of journal entries that are newer than the index.

btree_nodes
: Total nodes in the btree.

btree_used_percent
: Average fraction of btree in use.

bset_tree_stats
: Statistics about the auxiliary search trees

btree_cache_max_chain
: Longest chain in the btree node cache's hash table

cache_read_races
: Counts instances where while data was being read from the cache, the bucket was reused and invalidated - i.e. where the pointer was stale after the read completed. When this occurs the data is reread from the backing device.

trigger_gc
: Writing to this file forces garbage collection to run.

Sysfs - Cache device

Available at /sys/block/<cdev>/bcache

block_size
: Minimum granularity of writes - should match hardware sector size.

btree_written
: Sum of all btree writes, in (kilo/mega/giga) bytes

bucket_size
: Size of buckets

cache_replacement_policy
: One of either lru, fifo or random.

discard
: Boolean; if on a discard/TRIM will be issued to each bucket before it is reused. Defaults to off, since SATA TRIM is an unqueued command (and thus slow).

freelist_percent
: Size of the freelist as a percentage of nbuckets. Can be written to to increase the number of buckets kept on the freelist, which lets you artificially reduce the size of the cache at runtime. Mostly for testing purposes (i.e. testing how different size caches affect your hit rate), but since buckets are discarded when they move on to the freelist will also make the SSD's garbage collection easier by effectively giving it more reserved space.

io_errors
: Number of errors that have occurred, decayed by io_error_halflife.

metadata_written
: Sum of all non data writes (btree writes and all other metadata).

nbuckets
: Total buckets in this cache

priority_stats
: Statistics about how recently data in the cache has been accessed. This can reveal your working set size. Unused is the percentage of the cache that doesn't contain any data. Metadata is bcache's metadata overhead. Average is the average priority of cache buckets. Next is a list of quantiles with the priority threshold of each.

written
: Sum of all data that has been written to the cache; comparison with btree_written gives the amount of write inflation in bcache.