ZFS L2ARC

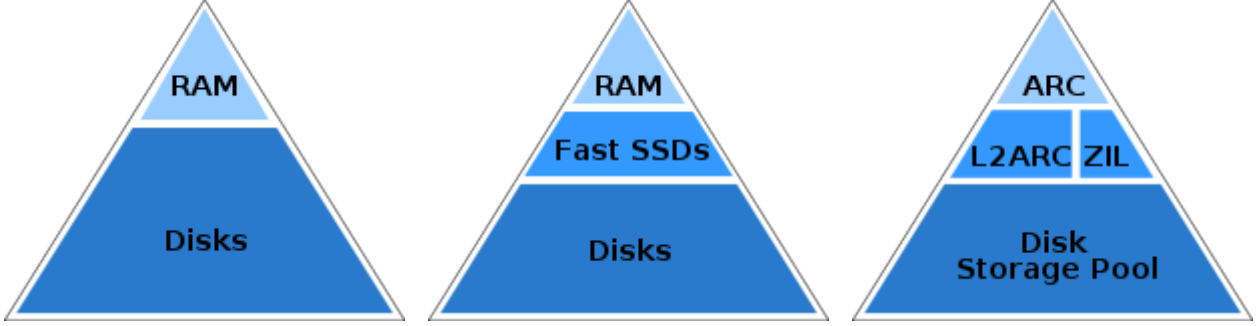
22 Jul 2008

I originally posted this at http://blogs.sun.com/brendan/entry/test.

An exciting new ZFS feature has now become publicly known: the second level ARC, or L2ARC. I've been busy with its development for over a year, however, this is my first chance to post about it. This post will show a quick example and answer some basic questions.

Background in a nutshell

The "ARC" is the ZFS main memory cache (in DRAM), which can be accessed with sub microsecond latency. An ARC read miss would normally read from disk, at millisecond latency (especially random reads). The L2ARC sits in-between, extending the main memory cache using fast storage devices, such as flash memory based SSDs (solid state disks).



old model *new model* *with ZFS*

Some example sizes to put this into perspective, from a lab machine named "walu":

Layer	*Medium*	*Total Capacity*
ARC	DRAM	128 Gbytes
L2ARC	6 x SSDs	550 Gbytes
Storage Pool	44 Disks	17.44 Tbytes (mirrored)

For this server, the L2ARC allows around 650 Gbytes to be stored in the total ZFS cache (ARC + L2ARC), rather than just DRAM with about 120 Gbytes.

A previous ZFS feature (the ZIL) allowed you to add SSD disks as log devices to improve write performance. This means ZFS provides two dimensions for adding flash memory to the file system stack: the L2ARC for random reads, and the ZIL for writes.

Adam has been the mastermind behind our flash memory efforts, and has written an excellent article in Communications of the ACM about flash memory based storage in ZFS; for more background, check it out.

L2ARC Example

To illustrate the L2ARC with an example, I'll use "walu", a medium-sized server in our test lab, which was briefly described above. Its ZFS pool of 44 x 7200 RPM disks is configured as a 2-way mirror, to provide both good reliability and performance. It also has 6 SSDs, which I'll add to the ZFS pool as L2ARC devices (or "cache devices").

I should note: this is an example of L2ARC operation, not a demonstration of the maximum performance that we can achieve (the SSDs I'm using here aren't the fastest I've ever used, nor the largest.)

20 clients access walu over NFSv3, and execute a random read workload with an 8 Kbyte record size across 500 Gbytes of files (which is also its working set).

1) Disks only

Since the 500 Gbytes of working set is larger than walu's 128 Gbytes of DRAM, the disks must service many requests. One way to grasp how this workload is performing is to examine the IOPS that the ZFS pool delivers:

```
walu# zpool iostat pool_0 30
               capacity     operations    bandwidth
pool         used  avail   read  write   read  write


------------------------------------------------------


  pool_0      8.38T  9.06T     95      4   762K  29.1K
  pool_0      8.38T  9.06T  1.87K     15  15.0M  30.3K
  pool_0      8.38T  9.06T  1.88K      3  15.1M  20.4K
  pool_0      8.38T  9.06T  1.89K     16  15.1M  39.3K
  pool_0      8.38T  9.06T  1.89K      4  15.1M  23.8K
  [...]
```

The pool is pulling about 1.89K ops/sec, which would require about 42 ops per disk of this pool. To examine how this is delivered by the disks, we can either use `zpool iostat` or the original `iostat`:

```
walu# iostat -xnz 10
[...trimmed first output...]
                    extended device statistics

     r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    43.9    0.0  351.5    0.0  0.0  0.4    0.0   10.0   0  34 c0t5000CCA215C46459d0

    47.6    0.0  381.1    0.0  0.0  0.5    0.0    9.8   0  36 c0t5000CCA215C4521Dd0
    42.7    0.0  349.9    0.0  0.0  0.4    0.0   10.1   0  35 c0t5000CCA215C45F89d0
    41.4    0.0  331.5    0.0  0.0  0.4    0.0    9.6   0  32 c0t5000CCA215C42A4Cd0
    45.6    0.0  365.1    0.0  0.0  0.4    0.0    9.2   0  34 c0t5000CCA215C45541d0
    45.0    0.0  360.3    0.0  0.0  0.4    0.0    9.4   0  34 c0t5000CCA215C458F1d0
    42.9    0.0  343.5    0.0  0.0  0.4    0.0    9.9   0  33 c0t5000CCA215C450E3d0
    44.9    0.0  359.5    0.0  0.0  0.4    0.0    9.3   0  35 c0t5000CCA215C45323d0
    45.9    0.0  367.5    0.0  0.0  0.5    0.0   10.1   0  37 c0t5000CCA215C4505Dd0
[...etc...]
```

`iostat` is interesting as it lists the service times: `wsvc_t` + `asvc_t`. These I/Os are taking on average between 9 and 10 milliseconds to complete, which the client application will usually suffer as latency. This time will be due to the random read nature of this workload: each I/O must wait as the disk heads seek and the disk platter rotates.

Another way to understand this performance is to examine the total NFSv3 ops delivered by this system (these days I use a GUI to monitor NFSv3 ops, but for this blog post I'll hammer `nfsstat` into printing something concise):

```
walu# nfsstat -v 3 1 | sed '/^Server NFSv3/,/^[0-9]/!d'
[...]
Server NFSv3:
calls     badcalls
2260      0
Server NFSv3:
calls     badcalls
2306      0
Server NFSv3:
calls     badcalls
2239      0
[...]
```

That's about 2.27K ops/sec for NFSv3; I'd expect 1.89K of that to be what our pool was delivering, and the rest are cache hits out of DRAM, which is warm at this point.

2) L2ARC devices

Now the 6 SSDs are added as L2ARC cache devices:

```
walu# zpool add pool_0 cache c7t0d0 c7t1d0 c8t0d0 c8t1d0 c9t0d0 c9t1d0
```

And we wait until the L2ARC is warm.

Time passes ...

Several hours later the cache devices have warmed up enough to satisfy most I/Os which miss main memory. The combined 'capacity/used' column for the cache devices shows that our 500 Gbytes of working set now exists on those 6 SSDs:

```
walu# zpool iostat -v pool_0 30
[...]
                            capacity     operations    bandwidth
pool                      used  avail   read  write   read  write


------------------------  -----  -----  -----  -----  -----  -----


  pool_0                  8.38T  9.06T     30     14   245K  31.9K
    mirror                 421G   507G      1      0  9.44K      0
      c0t5000CCA216CCB905d0    -      -      0      0  4.08K      0
      c0t5000CCA216CCB74Cd0    -      -      0      0  5.36K      0
    mirror                 416G   512G      0      0  7.66K      0
      c0t5000CCA216CCB919d0    -      -      0      0  4.34K      0
      c0t5000CCA216CCB763d0    -      -      0      0  3.32K      0
  [... 40 disks truncated ...]
  cache                       -      -      -      -      -      -
    c7t0d0                84.5G  8.63G  2.63K      0  21.1M  11.4K
    c7t1d0                84.7G  8.43G  2.62K      0  21.0M      0
    c8t0d0                84.5G  8.68G  2.61K      0  20.9M      0
    c8t1d0                84.8G  8.34G  2.64K      0  21.1M      0
    c9t0d0                84.3G  8.81G  2.63K      0  21.0M      0
    c9t1d0                84.2G  8.91G  2.63K      0  21.0M  1.53K


------------------------  -----  -----  -----  -----  -----  -----
```

The pool_0 disks are still serving some requests (in this output 30 ops/sec) but the bulk of the reads are being serviced by the L2ARC cache devices, each providing around 2.6K ops/sec. The total delivered by this ZFS pool is 15.8K ops/sec (pool disks + L2ARC devices), about **8.4x faster** than with disks alone.

This is confirmed by the delivered NFSv3 ops:

```
walu# nfsstat -v 3 1 | sed '/^Server NFSv3/,/^[0-9]/!d'
[...]
Server NFSv3:
calls      badcalls

18729      0

Server NFSv3:
calls      badcalls

18762      0

Server NFSv3:
calls      badcalls

19000      0

[...]
```

walu is now delivering 18.7K ops/sec, which is 8.3x faster than without the L2ARC.

However, the real win for the client applications is that of read *latency*; the disk-only iostat output showed our average was between 9 and 10 milliseconds, the L2ARC cache devices are delivering the following:

```
walu# iostat -xnz 10
                    extended device statistics

    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
[...]
 2665.0    0.4 21317.2    0.0  0.7  0.7    0.2    0.2  39  67 c9t0d0
 2668.1    0.5 21342.0    3.2  0.6  0.7    0.2    0.2  38  66 c9t1d0
 2665.4    0.4 21320.4    0.0  0.7  0.7    0.3    0.3  42  69 c8t0d0
 2683.6    0.4 21465.9    0.0  0.7  0.7    0.3    0.3  41  68 c8t1d0
 2660.7    0.6 21295.6    3.2  0.6  0.6    0.2    0.2  36  65 c7t1d0
 2650.7    0.4 21202.8    0.0  0.6  0.6    0.2    0.2  36  64 c7t0d0
```

Our average service time is between 0.4 and 0.6 ms (`wsvt_t` + `asvc_t` columns), which is about **20x faster** than what the disks were delivering.

What this means ...

An 8.3x improvement for 8 Kbyte random IOPS across a 500 Gbyte working set is impressive, as is improving storage I/O latency by 20x.

But this isn't really about the numbers, which will become dated (these SSDs were manufactured in July 2008, by a supplier who is providing us with bigger and faster SSDs every month).

What's important is that ZFS can make intelligent use of fast storage technology, in different roles to maximize their benefit. When you hear of new SSDs with incredible ops/sec performance, picture them as your L2ARC; or if it were great write throughput, picture them as your ZIL.

The example above was to show that the L2ARC can deliver, over NFS, whatever these SSDs could do. And these SSDs are being used as a second level cache, in-between main memory and disk, to achieve the best price/performance.

Questions

I recently spoke to a customer about the L2ARC and they asked a few questions which may be useful to repeat here:

What is L2ARC?

> The L2ARC is best pictured as a cache layer in-between main memory and disk, using flash memory based SSDs or other fast devices as storage. It holds non-dirty ZFS data, and is currently intended to improve the performance of random read workloads.

Isn't flash memory unreliable? What have you done about that?

> It's getting much better, but we have designed the L2ARC to handle errors safely. The data stored on the L2ARC is checksummed, and if the checksum is wrong or the SSD reports an error, we defer that read to the original pool of disks. Enough errors and the L2ARC device will offline itself. I've even yanked out busy L2ARC devices on live systems as part of testing, and everything continues to run.

Aren't SSDs really expensive?

> They used to be, but their price/performance has now reached the point where it makes sense to start using them in the coming months. See Adam's ACM article for more details about price/performance.

What about writes – isn't flash memory slow to write to?

> The L2ARC is coded to write to the cache devices asynchronously, so write latency doesn't affect system performance. This allows us to use "read-bias" SSDs for the L2ARC, which have the best read latency (and slow write latency).

What's bad about the L2ARC?

> It was designed to either improve performance or do nothing, so there isn't anything that should be bad. To explain what I mean by do nothing: if you use the L2ARC for a streaming or sequential workload, then the L2ARC will mostly ignore it and not cache it. This is because the default L2ARC settings assume you are using current SSD devices, where caching random read workloads is most favourable; with future SSDs (or other storage technology), we can use the L2ARC for streaming workloads as well.

Internals

If anyone is interested, I wrote a summary of L2ARC internals as a block comment in usr/src/uts/common/fs/zfs/arc.c, which is also surrounded by the actual implementation code. The block comment is below (see the source for the latest version), and is an excellent reference for how it really works:

```
/*
 * Level 2 ARC
 *
 * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
 * It uses dedicated storage devices to hold cached data, which are populated
 * using large infrequent writes.  The main role of this cache is to boost
 * the performance of random read workloads.  The intended L2ARC devices
 * include short-stroked disks, solid state disks, and other media with
 * substantially faster read latency than disk.
 *
 *                 +-----------------------+
 *                 |         ARC           |
 *                 +-----------------------+
 *                    |         ^     ^
 *                    |         |     |
 *     l2arc_feed_thread()    arc_read()
 *                    |         |     |
 *                    |  l2arc read   |
 *                    V         |     |
 *               +---------------+     |
 *               |     L2ARC     |     |
 *               +---------------+     |
 *                    |    ^           |
 *          l2arc_write() |           |
 *                    |    |           |
 *                    V    |           |
 *                 +-------+      +-------+
 *                 | vdev  |      | vdev  |
 *                 | cache |      | cache |
 *                 +-------+      +-------+
 *                 +=========+     .-----.
 *                 :  L2ARC  :    |-__-/
 *                 : devices :    | Disks |
 *                 +=========+    `-__-'
 *
 * Read requests are satisfied from the following sources, in order:
 *
 *      1) ARC
 *      2) vdev cache of L2ARC devices
 *      3) L2ARC devices
 *      4) vdev cache of disks
 *      5) disks
 *
 * Some L2ARC device types exhibit extremely slow write performance.
 * To accommodate for this there are some significant differences between
 * the L2ARC and traditional cache design:
 *
 * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
 * the ARC behave as usual, freeing buffers and placing headers on ghost
 * lists.  The ARC does not send buffers to the L2ARC during eviction as
 * this would add inflated write latencies for all ARC memory pressure.
 *
 * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
 * It does this by periodically scanning buffers from the eviction-end of
 * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
 * not already there.  It scans until a headroom of buffers is satisfied,
 * which itself is a buffer for ARC eviction.  The thread that does this is
 * l2arc_feed_thread(), illustrated below; example sizes are included to
 * provide a better sense of ratio than this diagram:
 *
 *             head -->                        tail
 *              +---------------------+----------+
 *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
 *              +---------------------+----------+   |   o L2ARC eligible
 *      ARC_mru |:#::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
 *              +---------------------+----------+   |
 *                   15.9 Gbytes      ^ 32 Mbytes    |
 *                                  headroom         |
 *                                             l2arc_feed_thread()
 *                                                   |
 *                       l2arc write hand <--[oooo]--'
 *                                   |           8 Mbyte
 *                                   |          write max
 *                                   V
 *                +==================================+
 *      L2ARC dev |####|#|###|###|    |####| ... |
 *                +==================================+
 *                                32 Gbytes
 *
 * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
 * evicted, then the L2ARC has cached a buffer much sooner than it probably
 * needed to, potentially wasting L2ARC device bandwidth and storage.  It is
 * safe to say that this is an uncommon case, since buffers at the end of
 * the ARC lists have moved there due to inactivity.
 *
 * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
 * then the L2ARC simply misses copying some buffers.  This serves as a
 * pressure valve to prevent heavy read workloads from both stalling the ARC
 * with waits and clogging the L2ARC with writes.  This also helps prevent
 * the potential for the L2ARC to churn if it attempts to cache content too
 * quickly, such as during backups of the entire pool.
 *
 * 5. After system boot and before the ARC has filled main memory, there are
 * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
```

```
 * no evictions from the ARC and so the tails of the ARC_mru and ARC_mfu
 * lists can remain mostly static.  Instead of searching from tail of these
 * lists as pictured, the l2arc_feed_thread() will search from the list heads
 * for eligible buffers, greatly increasing its chance of finding them.
 *
 * The L2ARC device write speed is also boosted during this time so that
 * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
 * there are no L2ARC reads, and no fear of degrading read performance
 * through increased writes.
 *
 * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
 * the vdev queue can aggregate them into larger and fewer writes.  Each
 * device is written to in a rotor fashion, sweeping writes through
 * available space then repeating.
 *
 * 7. The L2ARC does not store dirty content.  It never needs to flush
 * write buffers back to disk based storage.
 *
 * 8. If an ARC buffer is written (and dirtied) which also exists in the
 * L2ARC, the now stale L2ARC buffer is immediately dropped.
 *
 * The performance of the L2ARC can be tweaked by a number of tunables, which
 * may be necessary for different workloads:
 *
 *      l2arc_write_max         max write bytes per interval
 *      l2arc_write_boost       extra write bytes during device warmup
 *      l2arc_noprefetch        skip caching prefetched buffers
 *      l2arc_headroom          number of max device writes to precache
 *      l2arc_feed_secs         seconds between L2ARC writing
 *
 * Tunables may be removed or added as future performance improvements are
 * integrated, and also may become zpool properties.
 */
```

Jonathan Schwartz (our CEO) recently linked to this block comment in a blog entry about flash memory, to show that ZFS can incorporate flash into the storage hierarchy, and here is the *actual implementation*.

Copyright 2017 Brendan Gregg.
About this blog