

```
*****
147489 Wed Aug 7 13:35:00 2013
new/usr/src/uts/common/fs/zfs/arc.c
3995 Memory leak of compressed buffers in l2arc_write_done
3997 ZFS L2ARC default behavior should allow reading while writing
*****
```

```
1 /*
2 * CDDL HEADER START
3 *
4 * The contents of this file are subject to the terms of the
5 * Common Development and Distribution License (the "License").
6 * You may not use this file except in compliance with the License.
7 *
8 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 */
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
24 * Copyright (c) 2013 by Delphix. All rights reserved.
25 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
26 */

28 /*
29 * DVA-based Adjustable Replacement Cache
30 *
31 * While much of the theory of operation used here is
32 * based on the self-tuning, low overhead replacement cache
33 * presented by Megiddo and Modha at FAST 2003, there are some
34 * significant differences:
35 *
36 * 1. The Megiddo and Modha model assumes any page is evictable.
37 * Pages in its cache cannot be "locked" into memory. This makes
38 * the eviction algorithm simple: evict the last page in the list.
39 * This also make the performance characteristics easy to reason
40 * about. Our cache is not so simple. At any given moment, some
41 * subset of the blocks in the cache are un-evictable because we
42 * have handed out a reference to them. Blocks are only evictable
43 * when there are no external references active. This makes
44 * eviction far more problematic: we choose to evict the evictable
45 * blocks that are the "lowest" in the list.
46 *
47 * There are times when it is not possible to evict the requested
48 * space. In these circumstances we are unable to adjust the cache
49 * size. To prevent the cache growing unbounded at these times we
50 * implement a "cache throttle" that slows the flow of new data
51 * into the cache until we can make space available.
52 *
53 * 2. The Megiddo and Modha model assumes a fixed cache size.
54 * Pages are evicted when the cache is full and there is a cache
55 * miss. Our model has a variable sized cache. It grows with
56 * high use, but also tries to react to memory pressure from the
57 * operating system: decreasing its size when system memory is
58 * tight.
59 *
60 * 3. The Megiddo and Modha model assumes a fixed page size. All
```

```
61 * elements of the cache are therefore exactly the same size. So
62 * when adjusting the cache size following a cache miss, its simply
63 * a matter of choosing a single page to evict. In our model, we
64 * have variable sized cache blocks (ranging from 512 bytes to
65 * 128K bytes). We therefore choose a set of blocks to evict to make
66 * space for a cache miss that approximates as closely as possible
67 * the space used by the new block.
68 *
69 * See also: "ARC: A Self-Tuning, Low Overhead Replacement Cache"
70 * by N. Megiddo & D. Modha, FAST 2003
71 */

73 /*
74 * The locking model:
75 *
76 * A new reference to a cache buffer can be obtained in two
77 * ways: 1) via a hash table lookup using the DVA as a key,
78 * or 2) via one of the ARC lists. The arc_read() interface
79 * uses method 1, while the internal arc algorithms for
80 * adjusting the cache use method 2. We therefore provide two
81 * types of locks: 1) the hash table lock array, and 2) the
82 * arc list locks.
83 *
84 * Buffers do not have their own mutexes, rather they rely on the
85 * hash table mutexes for the bulk of their protection (i.e. most
86 * fields in the arc_buf_hdr_t are protected by these mutexes).
87 *
88 * buf_hash_find() returns the appropriate mutex (held) when it
89 * locates the requested buffer in the hash table. It returns
90 * NULL for the mutex if the buffer was not in the table.
91 *
92 * buf_hash_remove() expects the appropriate hash mutex to be
93 * already held before it is invoked.
94 *
95 * Each arc state also has a mutex which is used to protect the
96 * buffer list associated with the state. When attempting to
97 * obtain a hash table lock while holding an arc list lock you
98 * must use: mutex_tryenter() to avoid deadlock. Also note that
99 * the active state mutex must be held before the ghost state mutex.
100 *
101 * Arc buffers may have an associated eviction callback function.
102 * This function will be invoked prior to removing the buffer (e.g.
103 * in arc_do_user_evicts()). Note however that the data associated
104 * with the buffer may be evicted prior to the callback. The callback
105 * must be made with *no locks held* (to prevent deadlock). Additionally,
106 * the users of callbacks must ensure that their private data is
107 * protected from simultaneous callbacks from arc_buf_evict()
108 * and arc_do_user_evicts().
109 *
110 * Note that the majority of the performance stats are manipulated
111 * with atomic operations.
112 *
113 * The L2ARC uses the l2arc_buflist_mtx global mutex for the following:
114 *
115 * - L2ARC buflist creation
116 * - L2ARC buflist eviction
117 * - L2ARC write completion, which walks L2ARC buflists
118 * - ARC header destruction, as it removes from L2ARC buflists
119 * - ARC header release, as it removes from L2ARC buflists
120 *
121 * Please note that if you first grab the l2arc_buflist_mtx, you can't do a
122 * mutex_enter on a buffer's hash_lock anymore due to lock inversion. To grab
123 * the hash_lock you must use mutex_tryenter and possibly deal with the buffer
124 * not being available (due to e.g. some other thread holding it while trying
125 * to unconditionally grab the l2arc_buflist_mtx which you are holding). The
126 * inverse situation (first grab hash_lock, then l2arc_buflist_mtx) is safe.
```

```

127 /*
128 #include <sys/spa.h>
129 #include <sys/zio.h>
130 #include <sys/zio_compress.h>
131 #include <sys/zfs_context.h>
132 #include <sys/arc.h>
133 #include <sys/refcount.h>
134 #include <sys/vdev.h>
135 #include <sys/vdev_impl.h>
136 #ifndef _KERNEL
137 #include <sys/vmsystm.h>
138 #include <vm/anon.h>
139 #include <sys/fs/swapnode.h>
140 #include <sys/dnlc.h>
141 #endif
142 #endif
143 #include <sys/callb.h>
144 #include <sys/kstat.h>
145 #include <zfs_fletcher.h>

147 #ifndef _KERNEL
148 /* set with ZFS_DEBUG=watch, to enable watchpoints on frozen buffers */
149 boolean_t arc_watch = B_FALSE;
150 int arc_procfid;
151 #endif

153 static kmutex_t      arc_reclaim_thr_lock;
154 static kcondvar_t    arc_reclaim_thr_cv;     /* used to signal reclaim thr */
155 static uint8_t       arc_thread_exit;

157 extern int zfs_write_limit_shift;
158 extern uint64_t zfs_write_limit_max;
159 extern kmutex_t zfs_write_limit_lock;

161 #define ARC_REDUCE_DNLC_PERCENT 3
162 uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

164 typedef enum arc_reclaim_strategy {
165     ARC_RECLAIM_AGGR,          /* Aggressive reclaim strategy */
166     ARC_RECLAIM_CONS,         /* Conservative reclaim strategy */
167 } arc_reclaim_strategy_t;
unchanged_portion_omitted

595 static buf_hash_table_t buf_hash_table;

597 #define BUF_HASH_INDEX(sp, dva, birth) \
598     (buf_hash(sp, dva, birth) & buf_hash_table.ht_mask)
599 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
600 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
601 #define HDR_LOCK(hdr) \
602     (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_sp, &hdr->b_dva, hdr->b_birth)))

604 uint64_t zfs_crc64_table[256];

606 /*
607 * Level 2 ARC
608 */

610 #define L2ARC_WRITE_SIZE      (8 * 1024 * 1024)      /* initial write max */
611 #define L2ARC_HEADROOM        2                      /* num of writes */
612 /*
613 * If we discover during ARC scan any buffers to be compressed, we boost
614 * our headroom for the next scanning cycle by this percentage multiple.
615 */
616 #define L2ARC_HEADROOM_BOOST  200
617 #define L2ARC_FEED_SECS       1                      /* caching interval secs */

```

```

618 #define L2ARC_FEED_MIN_MS      200                  /* min caching interval ms */
619
620 #define l2arc_writes_sent      ARCSTAT(arcstat_l2_writes_sent)
621 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)
622
623 /* L2ARC Performance Tunables */
624 uint64_t l2arc_write_max = L2ARC_WRITE_SIZE;      /* default max write size */
625 uint64_t l2arc_write_boost = L2ARC_WRITE_SIZE;    /* extra write during warmup */
626 uint64_t l2arc_headroom = L2ARC_HEADROOM;         /* number of dev writes */
627 uint64_t l2arc_headroom_boost = L2ARC_HEADROOM_BOOST;
628 uint64_t l2arc_feed_secs = L2ARC_FEED_SECS;       /* interval seconds */
629 uint64_t l2arc_feed_min_ms = L2ARC_FEED_MIN_MS;   /* min interval milliseconds */
630 boolean_t l2arc_noprefetch = B_TRUE;              /* don't cache prefetch bufs */
631 boolean_t l2arc_feed_again = B_TRUE;               /* turbo warmup */
632 boolean_t l2arc_norw = B_FALSE;                   /* no reads during writes */
633 boolean_t l2arc_norw = B_TRUE;                   /* no reads during writes */

634 /*
635 * L2ARC Internals
636 */
637 typedef struct l2arc_dev {
638     vdev_t           vdev;                /* vdev */
639     spa_t            spa;                 /* spa */
640     uint64_t         l2ad_hand;           /* next write location */
641     uint64_t         l2ad_start;          /* first addr on device */
642     uint64_t         l2ad_end;             /* last addr on device */
643     uint64_t         l2ad_evict;          /* last addr eviction reached */
644     boolean_t        l2ad_first;           /* first sweep through */
645     boolean_t        l2ad_writing;         /* currently writing */
646     list_t           l2ad_buflist;         /* buffer list */
647     list_node_t      l2ad_node;            /* device list node */
648 } l2arc_dev_t;
unchanged_portion_omitted

649 struct l2arc_buf_hdr {
650     /* protected by arc_buf_hdr mutex */
651     l2arc_dev_t      b_dev;                /* L2ARC device */
652     uint64_t         b_daddr;              /* disk address, offset byte */
653     /* compression applied to buffer data */
654     enum zio_compress b_compress;
655     /* real alloc'd buffer size depending on b_compress applied */
656     int              b_asize;
657     /* temporary buffer holder for in-flight compressed data */
658     void             *b_tmp_cdata;
659 };
unchanged_portion_omitted

660 static kmutex_t l2arc_feed_thr_lock;
661 static kcondvar_t l2arc_feed_thr_cv;
662 static uint8_t l2arc_thread_exit;

663 static void l2arc_read_done(zio_t *zio);
664 static void l2arc_hdr_stat_add(void);
665 static void l2arc_hdr_stat_remove(void);

700 static boolean_t l2arc_compress_buf(void *in_data, uint64_t in_sz,
701                                     void **out_data, uint64_t *out_sz, enum zio_compress *compress);
702 static boolean_t l2arc_decompress_buf(l2arc_buf_hdr_t *l2hdr);
703     enum zio_compress c);
704 static void l2arc_release_cdata_buf(arc_buf_hdr_t *ab);

705 static uint64_t
706 buf_hash(uint64_t spa, const dva_t *dva, uint64_t birth)
707 {
708     uint8_t *vdva = (uint8_t *)dva;

```

```

709     uint64_t crc = -1ULL;
710     int i;
712 
713     ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);
714 
715     for (i = 0; i < sizeof(dva_t); i++)
716         crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ vdva[i]) & 0xFF];
717 
718     crc ^= (spa>>8) ^ birth;
719 
720 } unchanged_portion_omitted
4121 /*
4122 * Free buffers that were tagged for destruction.
4123 */
4124 static void
4125 l2arc_do_free_on_write(void)
4126 {
4127     list_t *buflist;
4128     l2arc_data_free_t *df, *df_prev;
4129 
4130     mutex_enter(&l2arc_free_on_write_mtx);
4131     buflist = l2arc_free_on_write();
4132 
4133     for (df = list_tail(buflist); df; df = df_prev) {
4134         df_prev = list_prev(buflist, df);
4135         ASSERT(df->l2df_data != NULL);
4136         ASSERT(df->l2df_func != NULL);
4137         df->l2df_func(df->l2df_data, df->l2df_size);
4138         list_remove(buflist, df);
4139         kmem_free(df, sizeof(l2arc_data_free_t));
4140     }
4141 
4142     mutex_exit(&l2arc_free_on_write_mtx);
4143 }

4145 /*
4146 * A write to a cache device has completed. Update all headers to allow
4147 * reads from these buffers to begin.
4148 */
4149 static void
4150 l2arc_write_done(zio_t *zio)
4151 {
4152     l2arc_write_callback_t *cb;
4153     l2arc_dev_t *dev;
4154     list_t *buflist;
4155     arc_buf_hdr_t *head, *ab;
4156     arc_buf_hdr_t *head, *ab, *ab_prev;
4157     l2arc_buf_hdr_t *abl2;
4158     kmutex_t *hash_lock;
4159 
4160     struct defer_done_entry {
4161         arc_buf_hdr_t *dde_buf;
4162         list_node_t dde_node;
4163     } *dde, *dde_next;
4164     list_t defer_done_list;
4165 
4166     cb = zio->io_private;
4167     ASSERT(cb != NULL);
4168     dev = cb->l2wcb_dev;
4169     ASSERT(dev != NULL);
4170     head = cb->l2wcb_head;
4171     ASSERT(head != NULL);
4172 
```

```

4169     buflist = dev->l2ad_buflist;
4170     ASSERT(buflist != NULL);
4171     DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4172                   l2arc_write_callback_t *, cb);
4173 
4174     if (zio->io_error != 0)
4175         ARCSTAT_BUMP(arcstat_l2_writes_error);
4176 
4177     mutex_enter(&l2arc_buflist_mtx);
4178 
4179     /*
4180      * All writes completed, or an error was hit.
4181      */
4182     list_create(&defer_done_list, sizeof(*dde),
4183                 offsetof(struct defer_done_entry, dde_node));
4184     for (ab = list_prev(buflist, head); ab; ab = list_prev(buflist, ab)) {
4185         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4186             ab_prev = list_prev(buflist, ab);
4187 
4188             hash_lock = HDR_LOCK(ab);
4189             if (!mutex_tryenter(hash_lock)) {
4190                 /*
4191                  * Can't pause here to grab hash_lock while also holding
4192                  * l2arc_buflist_mtx, so place the buffers on a temporary
4193                  * thread-local list for later processing.
4194                  * This buffer misses out. It may be in a stage
4195                  * of eviction. Its ARC_L2_WRITING flag will be
4196                  * left set, denying reads to this buffer.
4197                  */
4198             dde = kmalloc(sizeof(*dde), KM_SLEEP);
4199             dde->dde_buf = ab;
4200             list_insert_tail(&defer_done_list, dde);
4201             ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
4202             continue;
4203         }
4204 
4205         atomic_inc_64(&l2arc_writes_done);
4206         list_remove(buflist, head);
4207         kmem_cache_free(hdr_cache, head);
4208         mutex_exit(&l2arc_buflist_mtx);
4209         abl2 = ab->b_l2hdr;
4210 
4211         /*
4212          * Now process the buffers. We're not holding l2arc_buflist_mtx
4213          * anymore, so we can do a regular mutex_enter on the hash_lock.
4214          * Release the temporary compressed buffer as soon as possible.
4215          */
4216         for (dde = list_head(&defer_done_list); dde != NULL; dde = dde_next) {
4217             kmutex_t *hash_lock;
4218             if (abl2->b_compress != ZIO_COMPRESS_OFF)
4219                 l2arc_release_cdata_buf(ab);
4220 
4221             dde_next = list_next(&defer_done_list, dde);
4222             ab = dde->dde_buf;
4223             hash_lock = HDR_LOCK(ab);
4224 
4225             mutex_enter(hash_lock);
4226 
4227             if (zio->io_error != 0) {
4228                 /*
4229                  * Error - drop L2ARC entry.
4230                  */
4231                 l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;
4232                 mutex_enter(&l2arc_buflist_mtx);
4233                 list_remove(buflist, ab);
4234                 mutex_exit(&l2arc_buflist_mtx);
4235             }
4236         }
4237     }
4238 
```

```

4221         ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4220         ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4222         ab->b_l2hdr = NULL;
4223         kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4224         kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4225         ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4226     }
4227
4228     /*
4229      * Allow ARC to begin reads to this L2ARC entry.
4230      */
4231     ab->b_flags &= ~ARC_L2_WRITING;
4232
4233     mutex_exit(hash_lock);
4234
4235     list_remove(&defer_done_list, dde);
4236     list_destroy(&defer_done_list);
4237
4238     atomic_inc_64(&l2arc_writes_done);
4239     list_remove(buclist, head);
4240     kmem_cache_free(hdr_cache, head);
4241 } unchanged_portion_omitted
4242 */
4243 * Evict buffers from the device write hand to the distance specified in
4244 * bytes. This distance may span populated buffers, it may span nothing.
4245 * This is clearing a region on the L2ARC device ready for writing.
4246 * If the 'all' boolean is set, every buffer is evicted.
4247 */
4248 static void
4249 l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
4250 {
4251     list_t *buclist;
4252     l2arc_buf_hdr_t *l2hdr;
4253     l2arc_buf_hdr_t *abl2;
4254     arc_buf_hdr_t *ab, *ab_prev;
4255     kmutex_t *hash_lock;
4256     uint64_t taddr;
4257
4258     buclist = dev->l2ad_buclist;
4259
4260     if (buclist == NULL)
4261         return;
4262
4263     if (!all && dev->l2ad_first) {
4264         /*
4265          * This is the first sweep through the device. There is
4266          * nothing to evict.
4267          */
4268         return;
4269     }
4270
4271     if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
4272         /*
4273          * When nearing the end of the device, evict to the end
4274          * before the device write hand jumps to the start.
4275          */
4276         taddr = dev->l2ad_end;
4277     } else {

```

```

4278         ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4279         ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4280         ab->b_l2hdr = NULL;
4281         kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4282         kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4283         ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4284     }
4285
4286     taddr = dev->l2ad_hand + distance;
4287 }
4288 DTRACE_PROBE4(l2arc_evict, l2arc_dev_t *, dev, list_t *, buclist,
4289 uint64_t, taddr, boolean_t, all);
4290
4291 top:
4292 mutex_enter(&l2arc_buclist_mtx);
4293 for (ab = list_tail(buclist); ab; ab = ab_prev) {
4294     ab_prev = list_prev(buclist, ab);
4295
4296     hash_lock = HDR_LOCK(ab);
4297     if (!mutex_tryenter(hash_lock)) {
4298         /*
4299          * Missed the hash lock. Retry.
4300          */
4301         ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
4302         mutex_exit(&l2arc_buclist_mtx);
4303         mutex_enter(hash_lock);
4304         mutex_exit(hash_lock);
4305         goto top;
4306     }
4307
4308     if (HDR_L2_WRITE_HEAD(ab)) {
4309         /*
4310          * We hit a write head node. Leave it for
4311          * l2arc_write_done().
4312          */
4313         list_remove(buclist, ab);
4314         mutex_exit(hash_lock);
4315         continue;
4316     }
4317
4318     if (!all && ab->b_l2hdr != NULL &&
4319         (ab->b_l2hdr->b_daddr > taddr ||
4320          ab->b_l2hdr->b_daddr < dev->l2ad_hand)) {
4321         /*
4322          * We've evicted to the target address,
4323          * or the end of the device.
4324          */
4325         mutex_exit(hash_lock);
4326         break;
4327     }
4328
4329     if (HDR_FREE_IN_PROGRESS(ab)) {
4330         /*
4331          * Already on the path to destruction.
4332          */
4333         mutex_exit(hash_lock);
4334         continue;
4335     }
4336
4337     if (ab->b_state == arc_l2c_only) {
4338         ASSERT(!HDR_L2_READING(ab));
4339         /*
4340          * This doesn't exist in the ARC. Destroy.
4341          * arc_hdr_destroy() will call list_remove()
4342          * and decrement arcstat_l2_size.
4343          */
4344         arc_change_state(arc_anon, ab, hash_lock);
4345         arc_hdr_destroy(ab);
4346     } else {
4347         /*
4348          * Invalidate issued or about to be issued
4349          * reads, since we may be about to write
4350          * over this location.
4351          */
4352     }

```

```

4463     if (HDR_L2_READING(ab)) {
4464         ARCSTAT_BUMP(arcstat_l2_evict_reading);
4465         ab->b_flags |= ARC_L2_EVICTED;
4466     }
4467
4468     /*
4469      * Tell ARC this no longer exists in L2ARC.
4470      */
4471     if (ab->b_l2hdr != NULL) {
4472         l2hdr = ab->b_l2hdr;
4473         ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4474         ab12 = ab->b_l2hdr;
4475         ARCSTAT_INCR(arcstat_l2_asize, -ab12->b_asize);
4476         ab->b_l2hdr = NULL;
4477         kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4478         kmem_free(ab12, sizeof (l2arc_buf_hdr_t));
4479         ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4480     }
4481     list_remove(buclist, ab);
4482
4483     /*
4484      * This may have been leftover after a
4485      * failed write.
4486      */
4487     ab->b_flags &= ~ARC_L2_WRITING;
4488 }
4489 mutex_exit(hash_lock);
4490 mutex_exit(&l2arc_buclist_mtx);
4491 vdev_space_update(dev->l2ad_vdev, -(taddr - dev->l2ad_evict), 0, 0);
4492 }

4493 /*
4494  * Find and write ARC buffers to the L2ARC device.
4495  *
4496  * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4497  * for reading until they have completed writing.
4498  * The headroom_boost is an in-out parameter used to maintain headroom boost
4499  * state between calls to this function.
4500  *
4501  * Returns the number of bytes actually written (which may be smaller than
4502  * the delta by which the device hand has changed due to alignment).
4503  */
4504 */
4505 static uint64_t
4506 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
4507     boolean_t *headroom_boost)
4508 {
4509     arc_buf_hdr_t *ab, *ab_prev, *head;
4510     list_t *list;
4511     uint64_t write_asize, write_psize, write_sz, headroom,
4512     buf_compress_minsz;
4513     void *buf_data;
4514     kmutex_t *list_lock;
4515     boolean_t full;
4516     l2arc_write_callback_t *cb;
4517     zio_t *pio, *wzio;
4518     uint64_t guid = spa_load_guid(spa);
4519     const boolean_t do_headroom_boost = *headroom_boost;
4520     struct defer_write_entry {
4521         arc_buf_hdr_t *dwe_buf;
4522         void *dwe_orig_data;
4523         uint64_t dwe_orig_size;
4524         list_node_t *dwe_node;
4525     } *dwe, *dwe_next;

```

```

4526     list_t defer_write_list;
4528     ASSERT(dev->l2ad_vdev != NULL);
4529
4530     /* Lower the flag now, we might want to raise it again later. */
4531     *headroom_boost = B_FALSE;
4532
4533     pio = NULL;
4534     write_sz = write_asize = write_psize = 0;
4535     full = B_FALSE;
4536     head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
4537     head->b_flags |= ARC_L2_WRITE_HEAD;
4538
4539     /*
4540      * We will want to try to compress buffers that are at least 2x the
4541      * device sector size.
4542      */
4543     buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;
4544
4545     /*
4546      * Copy buffers for L2ARC writing.
4547      */
4548     list_create(&defer_write_list, sizeof (*dwe),
4549                 offsetof(struct defer_write_entry, dwe_node));
4550     mutex_enter(&l2arc_buclist_mtx);
4551     for (int try = 0; try <= 3; try++) {
4552         uint64_t passed_sz = 0;
4553
4554         list = l2arc_list_locked(try, &list_lock);
4555
4556         /*
4557          * L2ARC fast warmup.
4558          *
4559          * Until the ARC is warm and starts to evict, read from the
4560          * head of the ARC lists rather than the tail.
4561          */
4562         if (arc_warm == B_FALSE)
4563             ab = list_head(list);
4564         else
4565             ab = list_tail(list);
4566
4567         headroom = target_sz * l2arc_headroom;
4568         if (do_headroom_boost)
4569             headroom = (headroom * l2arc_headroom_boost) / 100;
4570
4571         for (; ab; ab = ab->prev) {
4572             l2arc_buf_hdr_t *l2hdr;
4573             kmutex_t *hash_lock;
4574             uint64_t buf_sz;
4575
4576             if (arc_warm == B_FALSE)
4577                 ab_prev = list_next(list, ab);
4578             else
4579                 ab_prev = list_prev(list, ab);
4580
4581             hash_lock = HDR_LOCK(ab);
4582             if (!mutex_tryenter(hash_lock)) {
4583                 /*
4584                  * Skip this buffer rather than waiting.
4585                  */
4586                 continue;
4587             }
4588             passed_sz += ab->b_size;
4589             if (passed_sz > headroom) {
4590                 /*

```

```
new/usr/src/uts/common/fs/zfs/arc.c

4591                         * Searched too far.
4592                         */
4593                         mutex_exit(hash_lock);
4594                         break;
4595                     }
4596
4597                 if (!l2arc_write_eligible(guid, ab)) {
4598                         mutex_exit(hash_lock);
4599                         continue;
4600                     }
4601
4602                 if ((write_sz + ab->b_size) > target_sz) {
4603                         full = B_TRUE;
4604                         mutex_exit(hash_lock);
4605                         break;
4606                     }
4607
4608                 if (pio == NULL) {
4609                         /*
4610                         * Insert a dummy header on the buflist so
4611                         * l2arc_write_done() can find where the
4612                         * write buffers begin without searching.
4613                         */
4614                         list_insert_head(dev->l2ad_buflist, head);
4615
4616                         cb = kmalloc(sizeof(l2arc_write_callback_t), KM_SLEEP);
4617                         cb->l2wcb_dev = dev;
4618                         cb->l2wcb_head = head;
4619                         pio = zio_root(spa, l2arc_write_done, cb,
4620                                         ZIO_FLAG_CANFAIL);
4621                     }
4622
4623                     /*
4624                         * Create and add a new L2ARC header.
4625                         */
4626                     l2hdr = kmalloc(sizeof(l2arc_buf_hdr_t), KM_SLEEP);
4627                     l2hdr->b_dev = dev;
4628                     ab->b_flags |= ARC_L2_WRITING;
4629                     l2hdr->b_compress = ZIO_COMPRESS_OFF;
4630                     l2hdr->b_asize = ab->b_size;
4631
4632                     /*
4633                         * Temporarily stash the buffer in defer_write_entries.
4634                         * Temporarily stash the data buffer in b_tmp_cdata.
4635                         * The subsequent write step will pick it up from
4636                         * there. This is because we can't access ab->b_buf
4637                         * there. This is because can't access ab->b_buf
4638                         * without holding the hash_lock, which we in turn
4639                         * can't access without holding the ARC list locks
4640                         * while walking the ARC lists (we want to avoid
4641                         * holding these locks during compression/writing).
4642                         * (which we want to avoid during compression/writing).
4643                     */
4644                     dwe = kmalloc(sizeof(*dwe), KM_SLEEP);
4645                     dwe->dwe_buf = ab;
4646                     dwe->dwe_orig_data = ab->b_buf->b_data;
4647                     dwe->dwe_orig_size = ab->b_size;
4648                     l2hdr->b_compress = ZIO_COMPRESS_OFF;
4649                     l2hdr->b_asize = ab->b_size;
4650                     l2hdr->b_tmp_cdata = ab->b_buf->b_data;
4651
4652                     buf_sz = ab->b_size;
4653                     ab->b_l2hdr = l2hdr;
4654
4655                     list_insert_head(dev->l2ad_buflist, ab);
```

```
new/usr/src/uts/common/fs/zfs/arc.c
4650                                         list_insert_tail(&defer_write_list, dwe);
4652
4653                                         /*
4654                                         * Compute and store the buffer cksum before
4655                                         * writing. On debug the cksum is verified first.
4656                                         */
4657                                         arc_cksum_verify(ab->b_buf);
4658                                         arc_cksum_compute(ab->b_buf, B_TRUE);
4659                                         mutex_exit(hash_lock);
4660                                         write_sz += dwe->dwe_orig_size;
4661                                         write_sz += buf_sz;
4662                                     }
4663
4664                                     mutex_exit(list_lock);
4665
4666                                     if (full == B_TRUE)
4667                                         break;
4668
4669                                     /* No buffers selected for writing? */
4670                                     if (pio == NULL) {
4671                                         ASSERT0(write_sz);
4672                                         mutex_exit(&l2arc_buclist_mtx);
4673                                         kmem_cache_free(hdr_cache, head);
4674                                         list_destroy(&defer_write_list);
4675                                         return (0);
4676                                     }
4677
4678                                     mutex_exit(&l2arc_buclist_mtx);
4679
4680                                     /*
4681                                     * Now start writing the buffers. We're starting at the write head
4682                                     * and work backwards, retracing the course of the buffer selector
4683                                     * loop above.
4684                                     */
4685                                     for (dwe = list_head(&defer_write_list); dwe != NULL; dwe = dwe_next) {
4686                                         for (ab = list_prev(dev->l2ad_buclist, head); ab;
4687                                             ab = list_prev(dev->l2ad_buclist, ab)) {
4688                                             l2arc_buf_hdr_t *l2hdr;
4689                                             uint64_t buf_sz;
4690
4691                                             dwe_next = list_next(&defer_write_list, dwe);
4692                                             ab = dwe->dwe_buf;
4693
4694                                             /*
4695                                             * Accessing ab->b_l2hdr without locking is safe here because
4696                                             * we're holding the l2arc_buclist_mtx and no other thread will
4697                                             * ever directly modify the L2 fields. In particular ab->b_buf
4698                                             * may be invalid by now due to ARC eviction.
4699                                             * We shouldn't need to lock the buffer here, since we flagged
4700                                             * it as ARC_L2_WRITING in the previous step, but we must take
4701                                             * care to only access its L2 cache parameters. In particular,
4702                                             * ab->b_buf may be invalid by now due to ARC eviction.
4703                                             */
4704                                             l2hdr = ab->b_l2hdr;
4705                                             l2hdr->b_daddr = dev->l2ad_hand;
4706
4707                                             if ((ab->b_flags & ARC_L2COMPRESS) &&
4708                                                 l2hdr->b_asize >= buf_compress_minsz &&
4709                                                 l2arc_compress_buf(dwe->dwe_orig_data, dwe->dwe_orig_size,
4710                                                 &buf_data, &buf_sz, &l2hdr->b_compress)) {
4711                                                 l2hdr->b_asize >= buf_compress_minsz) {
4712                                                 if (l2arc_compress_buf(l2hdr)) {
4713                                                 /*
4714                                                 */
4715                                             }
```

```

4707             * If compression succeeded, enable headroom
4708             * boost on the next scan cycle.
4709             */
4710             headroom_boost = B_TRUE;
4711             l2hdr->b_asize = buf_sz;
4712         } else {
4713             buf_data = dwe->dwe_orig_data;
4714             buf_sz = dwe->dwe_orig_size;
4715             l2hdr->b_asize = dwe->dwe_orig_size;
4716         }
4717     }
4718
4719     /*
4720      * Pick up the buffer data we had previously stashed away
4721      * (and now potentially also compressed).
4722      */
4723     buf_data = l2hdr->b_tmp_cdata;
4724     buf_sz = l2hdr->b_asize;
4725
4726     /* Compression may have squashed the buffer to zero length. */
4727     if (buf_sz != 0) {
4728         uint64_t buf_p_sz;
4729
4730         wzio = zio_write_phys(pio, dev->l2ad_vdev,
4731                               dev->l2ad_hand, l2hdr->b_asize, buf_data,
4732                               ZIO_CHECKSUM_OFF, NULL, NULL,
4733                               ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL,
4734                               B_FALSE);
4735         dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
4736         NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
4737         ZIO_FLAG_CANFAIL, B_FALSE);
4738
4739         DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
4740                       zio_t *, wzio);
4741         (void) zio_nowait(wzio);
4742
4743         write_asize += l2hdr->b_asize;
4744         write_asize += buf_sz;
4745         /*
4746          * Keep the clock hand suitably device-aligned.
4747          */
4748         buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4749         write_psize += buf_p_sz;
4750         dev->l2ad_hand += buf_p_sz;
4751     }
4752
4753     list_remove(&defer_write_list, dwe);
4754     kmem_free(dwe, sizeof (*dwe));
4755 }
4756
4757 list_destroy(&defer_write_list);
4758 mutex_exit(&l2arc_buclist_mtx);
4759
4760 ASSERT3U(write_asize, <=, target_sz);
4761 ARSTAT_BUMP(arcstat_l2_writes_sent);
4762 ARSTAT_INCR(arcstat_l2_write_bytes, write_asize);
4763 ARSTAT_INCR(arcstat_l2_size, write_sz);
4764 ARSTAT_INCR(arcstat_l2_asize, write_asize);
4765 vdev_space_update(dev->l2ad_vdev, write_psize, 0, 0);
4766
4767 /*
4768  * Bump device hand to the device start if it is approaching the end.
4769  * l2arc_evict() will already have evicted ahead for this case.
4770  */
4771 if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
4772     vdev_space_update(dev->l2ad_vdev,

```

```

4760             dev->l2ad_end - dev->l2ad_hand, 0, 0);
4761             dev->l2ad_hand = dev->l2ad_start;
4762             dev->l2ad_evict = dev->l2ad_start;
4763             dev->l2ad_first = B_FALSE;
4764         }
4765
4766         dev->l2ad_writing = B_TRUE;
4767         (void) zio_wait(pio);
4768         dev->l2ad_writing = B_FALSE;
4769
4770     return (write_asize);
4771 }
4772
4773 /*
4774  * Compresses an L2ARC buffer.
4775  * The data to be compressed is in in_data and its size in in_sz. This routine
4776  * tries to compress the data and depending on the compression result there
4777  * are three possible outcomes:
4778  * *) The buffer was incompressible. The function returns with B_FALSE and
4779  * does nothing else.
4780  * The data to be compressed must be prefilled in l2hdr->b_tmp_cdata and its
4781  * size in l2hdr->b_asize. This routine tries to compress the data and
4782  * depending on the compression result there are three possible outcomes:
4783  * *) The buffer was incompressible. The original l2hdr contents were left
4784  * untouched and are ready for writing to an L2 device.
4785  * *) The buffer was all-zeros, so there is no need to write it to an L2
4786  * device. To indicate this situation, the *out_data is set to NULL,
4787  * *out_sz is set to zero, *compress is set to ZIO_COMPRESS_EMPTY and
4788  * the function returns B_TRUE.
4789  * *) Compression succeeded and *out_data was set to point to a buffer holding
4790  * the compressed data buffer, *out_sz was set to indicate the output size,
4791  * *compress was set to the appropriate compression algorithm and B_TRUE is
4792  * returned. Once writing is done the buffer will be automatically freed by
4793  * l2arc_do_free_on_write().
4794  * device. To indicate this situation b_tmp_cdata is NULL'ed, b_asize is
4795  * set to zero and b_compress is set to ZIO_COMPRESS_EMPTY.
4796  * *) Compression succeeded and b_tmp_cdata was replaced with a temporary
4797  * data buffer which holds the compressed data to be written, and b_asize
4798  * tells us how much data there is. b_compress is set to the appropriate
4799  * compression algorithm. Once writing is done, invoke
4800  * l2arc_release_cdata_buf on this l2hdr to free this temporary buffer.
4801
4802 */
4803
4804 Returns B_TRUE if compression succeeded, or B_FALSE if it didn't (the
4805 * buffer was incompressible).
4806
4807 static boolean_t
4808 l2arc_compress_buf(void *in_data, uint64_t in_sz, void **out_data,
4809                     uint64_t *out_sz, enum zio_compress *compress)
4810 {
4811     l2arc_compress_buf(l2arc_buf_hdr_t *l2hdr)
4812     {
4813         void *cdata;
4814         size_t csize, len;
4815
4816         cdata = zio_data_buf_alloc(in_sz);
4817         *out_sz = zio_compress_data(ZIO_COMPRESS_LZ4, in_data, cdata, in_sz);
4818         ASSERT(l2hdr->b_compress == ZIO_COMPRESS_OFF);
4819         ASSERT(l2hdr->b_tmp_cdata != NULL);
4820
4821         if (*out_sz == 0) {
4822             /* Zero block, indicate that there's nothing to write. */
4823             zio_data_buf_free(cdata, in_sz);
4824             *compress = ZIO_COMPRESS_EMPTY;
4825             *out_data = NULL;
4826             len = l2hdr->b_asize;
4827             cdata = zio_data_buf_alloc(len);
4828             csize = zio_compress_data(ZIO_COMPRESS_LZ4, l2hdr->b_tmp_cdata,

```

```

4766         cdata, l2hdr->b_asize);
4767
4768     if (csize == 0) {
4769         /* zero block, indicate that there's nothing to write */
4770         zio_data_buf_free(cdata, len);
4771         l2hdr->b_compress = ZIO_COMPRESS_EMPTY;
4772         l2hdr->b_asize = 0;
4773         l2hdr->b_tmp_cdata = NULL;
4774         ARCSTAT_BUMP(arcstat_l2_compress_zeros);
4775         return (B_TRUE);
4776     } else if (*out_sz > 0 && *out_sz < in_sz) {
4777     } else if (csize > 0 && csize < len) {
4778         /*
4779          * Compression succeeded, we'll keep the cdata around for
4780          * writing and release it after writing.
4781          * writing and release it afterwards.
4782         */
4783         l2arc_data_free_t *df;
4784
4785         *compress = ZIO_COMPRESS_LZ4;
4786         *out_data = cdata;
4787
4788         df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
4789         df->l2df_data = cdata;
4790         df->l2df_size = *out_sz;
4791         df->l2df_func = zio_data_buf_free;
4792         mutex_enter(&l2arc_free_on_write_mtx);
4793         list_insert_head(l2arc_free_on_write, df);
4794         mutex_exit(&l2arc_free_on_write_mtx);
4795
4796         l2hdr->b_compress = ZIO_COMPRESS_LZ4;
4797         l2hdr->b_asize = csize;
4798         l2hdr->b_tmp_cdata = cdata;
4799         ARCSTAT_BUMP(arcstat_l2_compress_successes);
4800         ARCSTAT_BUMP(arcstat_l2_free_on_write);
4801         return (B_TRUE);
4802     } else {
4803         /*
4804          * Compression failed, release the compressed buffer.
4805          * l2hdr will be left unmodified.
4806         */
4807         zio_data_buf_free(cdata, in_sz);
4808         zio_data_buf_free(cdata, len);
4809         ARCSTAT_BUMP(arcstat_l2_compress_failures);
4810         return (B_FALSE);
4811     }
4812
4813     unchanged_portion_omitted
4814
4815 */
4816 * Releases the temporary b_tmp_cdata buffer in an l2arc header structure.
4817 * This buffer serves as a temporary holder of compressed data while
4818 * the buffer entry is being written to an l2arc device. Once that is
4819 * done, we can dispose of it.
4820 */
4821 static void
4822 l2arc_release_cdata_buf(arc_buf_hdr_t *ab)
4823 {
4824     l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;
4825
4826     if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4827         /*
4828          * If the data was compressed, then we've allocated a
4829          * temporary buffer for it, so now we need to release it.
4830         */
4831         ASSERT(l2hdr->b_tmp_cdata != NULL);
4832
4833
4834
4835 }
```

```

4876             zio_data_buf_free(l2hdr->b_tmp_cdata, ab->b_size);
4877         }
4878         l2hdr->b_tmp_cdata = NULL;
4879     }
4880
4881     /*
4882      * This thread feeds the L2ARC at regular intervals. This is the beating
4883      * heart of the L2ARC.
4884      */
4885     static void
4886     l2arc_feed_thread(void)
4887     {
4888         callb_cpr_t cpr;
4889         l2arc_dev_t *dev;
4890         spa_t *spa;
4891         uint64_t size, wrote;
4892         clock_t begin, next = ddi_get_lbolt();
4893         boolean_t headroom_boost = B_FALSE;
4894
4895         CALLB_CPR_INIT(&cpr, &l2arc_feed_thr_lock, callb_generic_cpr, FTAG);
4896
4897         mutex_enter(&l2arc_feed_thr_lock);
4898
4899         while (l2arc_thread_exit == 0) {
4900             CALLB_CPR_SAFE_BEGIN(&cpr);
4901             (void) cv_timedwait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock,
4902                                 next);
4903             CALLB_CPR_SAFE_END(&cpr, &l2arc_feed_thr_lock);
4904             next = ddi_get_lbolt() + hz;
4905
4906             /*
4907              * Quick check for L2ARC devices.
4908              */
4909             mutex_enter(&l2arc_dev_mtx);
4910             if (l2arc_ndev == 0) {
4911                 mutex_exit(&l2arc_dev_mtx);
4912                 continue;
4913             }
4914             mutex_exit(&l2arc_dev_mtx);
4915             begin = ddi_get_lbolt();
4916
4917             /*
4918              * This selects the next l2arc device to write to, and in
4919              * doing so the next spa to feed from: dev->l2ad_spa. This
4920              * will return NULL if there are now no l2arc devices or if
4921              * they are all faulted.
4922              */
4923             if (dev = l2arc_dev_get_next()) == NULL)
4924                 continue;
4925
4926             /*
4927              * If a device is returned, its spa's config lock is also
4928              * held to prevent device removal. l2arc_dev_get_next()
4929              * will grab and release l2arc_dev_mtx.
4930              */
4931             spa = dev->l2ad_spa;
4932             ASSERT(spa != NULL);
4933
4934             /*
4935              * If the pool is read-only then force the feed thread to
4936              * sleep a little longer.
4937              */
4938             if (!spa_writeable(spa)) {
4939                 next = ddi_get_lbolt() + 5 * l2arc_feed_secs * hz;
4940                 spa_config_exit(spa, SCL_L2ARC, dev);
4941                 continue;
4942             }
4943
4944             /*
4945              * Write the data to the device.
4946              */
4947             l2arc_data_free(df);
4948             df->l2df_data = NULL;
4949             df->l2df_size = 0;
4950             df->l2df_func = zio_data_buf_free;
4951             mutex_enter(&l2arc_free_on_write_mtx);
4952             list_insert_head(l2arc_free_on_write, df);
4953             mutex_exit(&l2arc_free_on_write_mtx);
4954
4955             /*
4956              * If the data was compressed, then we've allocated a
4957              * temporary buffer for it, so now we need to release it.
4958              */
4959             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4960                 l2arc_release_cdata_buf(ab);
4961             }
4962
4963             /*
4964              * If the data was compressed, then we've allocated a
4965              * temporary buffer for it, so now we need to release it.
4966              */
4967             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4968                 l2arc_release_cdata_buf(ab);
4969             }
4970
4971             /*
4972              * If the data was compressed, then we've allocated a
4973              * temporary buffer for it, so now we need to release it.
4974              */
4975             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4976                 l2arc_release_cdata_buf(ab);
4977             }
4978
4979             /*
4980              * If the data was compressed, then we've allocated a
4981              * temporary buffer for it, so now we need to release it.
4982              */
4983             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4984                 l2arc_release_cdata_buf(ab);
4985             }
4986
4987             /*
4988              * If the data was compressed, then we've allocated a
4989              * temporary buffer for it, so now we need to release it.
4990              */
4991             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4992                 l2arc_release_cdata_buf(ab);
4993             }
4994
4995             /*
4996              * If the data was compressed, then we've allocated a
4997              * temporary buffer for it, so now we need to release it.
4998              */
4999             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5000                 l2arc_release_cdata_buf(ab);
5001             }
5002
5003             /*
5004              * If the data was compressed, then we've allocated a
5005              * temporary buffer for it, so now we need to release it.
5006              */
5007             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5008                 l2arc_release_cdata_buf(ab);
5009             }
5010
5011             /*
5012              * If the data was compressed, then we've allocated a
5013              * temporary buffer for it, so now we need to release it.
5014              */
5015             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5016                 l2arc_release_cdata_buf(ab);
5017             }
5018
5019             /*
5020              * If the data was compressed, then we've allocated a
5021              * temporary buffer for it, so now we need to release it.
5022              */
5023             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5024                 l2arc_release_cdata_buf(ab);
5025             }
5026
5027             /*
5028              * If the data was compressed, then we've allocated a
5029              * temporary buffer for it, so now we need to release it.
5030              */
5031             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5032                 l2arc_release_cdata_buf(ab);
5033             }
5034
5035             /*
5036              * If the data was compressed, then we've allocated a
5037              * temporary buffer for it, so now we need to release it.
5038              */
5039             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5040                 l2arc_release_cdata_buf(ab);
5041             }
5042
5043             /*
5044              * If the data was compressed, then we've allocated a
5045              * temporary buffer for it, so now we need to release it.
5046              */
5047             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5048                 l2arc_release_cdata_buf(ab);
5049             }
5050
5051             /*
5052              * If the data was compressed, then we've allocated a
5053              * temporary buffer for it, so now we need to release it.
5054              */
5055             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5056                 l2arc_release_cdata_buf(ab);
5057             }
5058
5059             /*
5060              * If the data was compressed, then we've allocated a
5061              * temporary buffer for it, so now we need to release it.
5062              */
5063             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5064                 l2arc_release_cdata_buf(ab);
5065             }
5066
5067             /*
5068              * If the data was compressed, then we've allocated a
5069              * temporary buffer for it, so now we need to release it.
5070              */
5071             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5072                 l2arc_release_cdata_buf(ab);
5073             }
5074
5075             /*
5076              * If the data was compressed, then we've allocated a
5077              * temporary buffer for it, so now we need to release it.
5078              */
5079             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5080                 l2arc_release_cdata_buf(ab);
5081             }
5082
5083             /*
5084              * If the data was compressed, then we've allocated a
5085              * temporary buffer for it, so now we need to release it.
5086              */
5087             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5088                 l2arc_release_cdata_buf(ab);
5089             }
5090
5091             /*
5092              * If the data was compressed, then we've allocated a
5093              * temporary buffer for it, so now we need to release it.
5094              */
5095             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5096                 l2arc_release_cdata_buf(ab);
5097             }
5098
5099             /*
5100              * If the data was compressed, then we've allocated a
5101              * temporary buffer for it, so now we need to release it.
5102              */
5103             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5104                 l2arc_release_cdata_buf(ab);
5105             }
5106
5107             /*
5108              * If the data was compressed, then we've allocated a
5109              * temporary buffer for it, so now we need to release it.
5110              */
5111             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5112                 l2arc_release_cdata_buf(ab);
5113             }
5114
5115             /*
5116              * If the data was compressed, then we've allocated a
5117              * temporary buffer for it, so now we need to release it.
5118              */
5119             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5120                 l2arc_release_cdata_buf(ab);
5121             }
5122
5123             /*
5124              * If the data was compressed, then we've allocated a
5125              * temporary buffer for it, so now we need to release it.
5126              */
5127             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5128                 l2arc_release_cdata_buf(ab);
5129             }
5130
5131             /*
5132              * If the data was compressed, then we've allocated a
5133              * temporary buffer for it, so now we need to release it.
5134              */
5135             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5136                 l2arc_release_cdata_buf(ab);
5137             }
5138
5139             /*
5140              * If the data was compressed, then we've allocated a
5141              * temporary buffer for it, so now we need to release it.
5142              */
5143             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5144                 l2arc_release_cdata_buf(ab);
5145             }
5146
5147             /*
5148              * If the data was compressed, then we've allocated a
5149              * temporary buffer for it, so now we need to release it.
5150              */
5151             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5152                 l2arc_release_cdata_buf(ab);
5153             }
5154
5155             /*
5156              * If the data was compressed, then we've allocated a
5157              * temporary buffer for it, so now we need to release it.
5158              */
5159             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5160                 l2arc_release_cdata_buf(ab);
5161             }
5162
5163             /*
5164              * If the data was compressed, then we've allocated a
5165              * temporary buffer for it, so now we need to release it.
5166              */
5167             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5168                 l2arc_release_cdata_buf(ab);
5169             }
5170
5171             /*
5172              * If the data was compressed, then we've allocated a
5173              * temporary buffer for it, so now we need to release it.
5174              */
5175             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5176                 l2arc_release_cdata_buf(ab);
5177             }
5178
5179             /*
5180              * If the data was compressed, then we've allocated a
5181              * temporary buffer for it, so now we need to release it.
5182              */
5183             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5184                 l2arc_release_cdata_buf(ab);
5185             }
5186
5187             /*
5188              * If the data was compressed, then we've allocated a
5189              * temporary buffer for it, so now we need to release it.
5190              */
5191             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5192                 l2arc_release_cdata_buf(ab);
5193             }
5194
5195             /*
5196              * If the data was compressed, then we've allocated a
5197              * temporary buffer for it, so now we need to release it.
5198              */
5199             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5200                 l2arc_release_cdata_buf(ab);
5201             }
5202
5203             /*
5204              * If the data was compressed, then we've allocated a
5205              * temporary buffer for it, so now we need to release it.
5206              */
5207             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5208                 l2arc_release_cdata_buf(ab);
5209             }
5210
5211             /*
5212              * If the data was compressed, then we've allocated a
5213              * temporary buffer for it, so now we need to release it.
5214              */
5215             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5216                 l2arc_release_cdata_buf(ab);
5217             }
5218
5219             /*
5220              * If the data was compressed, then we've allocated a
5221              * temporary buffer for it, so now we need to release it.
5222              */
5223             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5224                 l2arc_release_cdata_buf(ab);
5225             }
5226
5227             /*
5228              * If the data was compressed, then we've allocated a
5229              * temporary buffer for it, so now we need to release it.
5230              */
5231             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5232                 l2arc_release_cdata_buf(ab);
5233             }
5234
5235             /*
5236              * If the data was compressed, then we've allocated a
5237              * temporary buffer for it, so now we need to release it.
5238              */
5239             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5240                 l2arc_release_cdata_buf(ab);
5241             }
5242
5243             /*
5244              * If the data was compressed, then we've allocated a
5245              * temporary buffer for it, so now we need to release it.
5246              */
5247             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5248                 l2arc_release_cdata_buf(ab);
5249             }
5250
5251             /*
5252              * If the data was compressed, then we've allocated a
5253              * temporary buffer for it, so now we need to release it.
5254              */
5255             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5256                 l2arc_release_cdata_buf(ab);
5257             }
5258
5259             /*
5260              * If the data was compressed, then we've allocated a
5261              * temporary buffer for it, so now we need to release it.
5262              */
5263             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5264                 l2arc_release_cdata_buf(ab);
5265             }
5266
5267             /*
5268              * If the data was compressed, then we've allocated a
5269              * temporary buffer for it, so now we need to release it.
5270              */
5271             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5272                 l2arc_release_cdata_buf(ab);
5273             }
5274
5275             /*
5276              * If the data was compressed, then we've allocated a
5277              * temporary buffer for it, so now we need to release it.
5278              */
5279             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5280                 l2arc_release_cdata_buf(ab);
5281             }
5282
5283             /*
5284              * If the data was compressed, then we've allocated a
5285              * temporary buffer for it, so now we need to release it.
5286              */
5287             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5288                 l2arc_release_cdata_buf(ab);
5289             }
5290
5291             /*
5292              * If the data was compressed, then we've allocated a
5293              * temporary buffer for it, so now we need to release it.
5294              */
5295             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5296                 l2arc_release_cdata_buf(ab);
5297             }
5298
5299             /*
5300              * If the data was compressed, then we've allocated a
5301              * temporary buffer for it, so now we need to release it.
5302              */
5303             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5304                 l2arc_release_cdata_buf(ab);
5305             }
5306
5307             /*
5308              * If the data was compressed, then we've allocated a
5309              * temporary buffer for it, so now we need to release it.
5310              */
5311             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5312                 l2arc_release_cdata_buf(ab);
5313             }
5314
5315             /*
5316              * If the data was compressed, then we've allocated a
5317              * temporary buffer for it, so now we need to release it.
5318              */
5319             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5320                 l2arc_release_cdata_buf(ab);
5321             }
5322
5323             /*
5324              * If the data was compressed, then we've allocated a
5325              * temporary buffer for it, so now we need to release it.
5326              */
5327             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5328                 l2arc_release_cdata_buf(ab);
5329             }
5330
5331             /*
5332              * If the data was compressed, then we've allocated a
5333              * temporary buffer for it, so now we need to release it.
5334              */
5335             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5336                 l2arc_release_cdata_buf(ab);
5337             }
5338
5339             /*
5340              * If the data was compressed, then we've allocated a
5341              * temporary buffer for it, so now we need to release it.
5342              */
5343             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5344                 l2arc_release_cdata_buf(ab);
5345             }
5346
5347             /*
5348              * If the data was compressed, then we've allocated a
5349              * temporary buffer for it, so now we need to release it.
5350              */
5351             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5352                 l2arc_release_cdata_buf(ab);
5353             }
5354
5355             /*
5356              * If the data was compressed, then we've allocated a
5357              * temporary buffer for it, so now we need to release it.
5358              */
5359             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5360                 l2arc_release_cdata_buf(ab);
5361             }
5362
5363             /*
5364              * If the data was compressed, then we've allocated a
5365              * temporary buffer for it, so now we need to release it.
5366              */
5367             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5368                 l2arc_release_cdata_buf(ab);
5369             }
5370
5371             /*
5372              * If the data was compressed, then we've allocated a
5373              * temporary buffer for it, so now we need to release it.
5374              */
5375             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5376                 l2arc_release_cdata_buf(ab);
5377             }
5378
5379             /*
5380              * If the data was compressed, then we've allocated a
5381              * temporary buffer for it, so now we need to release it.
5382              */
5383             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5384                 l2arc_release_cdata_buf(ab);
5385             }
5386
5387             /*
5388              * If the data was compressed, then we've allocated a
5389              * temporary buffer for it, so now we need to release it.
5390              */
5391             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5392                 l2arc_release_cdata_buf(ab);
5393             }
5394
5395             /*
5396              * If the data was compressed, then we've allocated a
5397              * temporary buffer for it, so now we need to release it.
5398              */
5399             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5400                 l2arc_release_cdata_buf(ab);
5401             }
5402
5403             /*
5404              * If the data was compressed, then we've allocated a
5405              * temporary buffer for it, so now we need to release it.
5406              */
5407             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5408                 l2arc_release_cdata_buf(ab);
5409             }
5410
5411             /*
5412              * If the data was compressed, then we've allocated a
5413              * temporary buffer for it, so now we need to release it.
5414              */
5415             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5416                 l2arc_release_cdata_buf(ab);
5417             }
5418
5419             /*
5420              * If the data was compressed, then we've allocated a
5421              * temporary buffer for it, so now we need to release it.
5422              */
5423             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5424                 l2arc_release_cdata_buf(ab);
5425             }
5426
5427             /*
5428              * If the data was compressed, then we've allocated a
5429              * temporary buffer for it, so now we need to release it.
5430              */
5431             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5432                 l2arc_release_cdata_buf(ab);
5433             }
5434
5435             /*
5436              * If the data was compressed, then we've allocated a
5437              * temporary buffer for it, so now we need to release it.
5438              */
5439             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5440                 l2arc_release_cdata_buf(ab);
5441             }
5442
5443             /*
5444              * If the data was compressed, then we've allocated a
5445              * temporary buffer for it, so now we need to release it.
5446              */
5447             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5448                 l2arc_release_cdata_buf(ab);
5449             }
5450
5451             /*
5452              * If the data was compressed, then we've allocated a
5453              * temporary buffer for it, so now we need to release it.
5454              */
5455             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5456                 l2arc_release_cdata_buf(ab);
5457             }
5458
5459             /*
5460              * If the data was compressed, then we've allocated a
5461              * temporary buffer for it, so now we need to release it.
5462              */
5463             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5464                 l2arc_release_cdata_buf(ab);
5465             }
5466
5467             /*
5468              * If the data was compressed, then we've allocated a
5469              * temporary buffer for it, so now we need to release it.
5470              */
5471             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5472                 l2arc_release_cdata_buf(ab);
5473             }
5474
5475             /*
5476              * If the data was compressed, then we've allocated a
5477              * temporary buffer for it, so now we need to release it.
5478              */
5479             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5480                 l2arc_release_cdata_buf(ab);
5481             }
5482
5483             /*
5484              * If the data was compressed, then we've allocated a
5485              * temporary buffer for it, so now we need to release it.
5486              */
5487             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5488                 l2arc_release_cdata_buf(ab);
5489             }
5490
5491             /*
5492              * If the data was compressed, then we've allocated a
5493              * temporary buffer for it, so now we need to release it.
5494              */
5495             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5496                 l2arc_release_cdata_buf(ab);
5497             }
5498
5499             /*
5500              * If the data was compressed, then we've allocated a
5501              * temporary buffer for it, so now we need to release it.
5502              */
5503             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5504                 l2arc_release_cdata_buf(ab);
5505             }
5506
5507             /*
5508              * If the data was compressed, then we've allocated a
5509              * temporary buffer for it, so now we need to release it.
5510              */
5511             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5512                 l2arc_release_cdata_buf(ab);
5513             }
5514
5515             /*
5516              * If the data was compressed, then we've allocated a
5517              * temporary buffer for it, so now we need to release it.
5518              */
5519             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5520                 l2arc_release_cdata_buf(ab);
5521             }
5522
5523             /*
5524              * If the data was compressed, then we've allocated a
5525              * temporary buffer for it, so now we need to release it.
5526              */
5527             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5528                 l2arc_release_cdata_buf(ab);
5529             }
5530
5531             /*
5532              * If the data was compressed, then we've allocated a
5533              * temporary buffer for it, so now we need to release it.
5534              */
5535             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5536                 l2arc_release_cdata_buf(ab);
5537             }
5538
5539             /*
5540              * If the data was compressed, then we've allocated a
5541              * temporary buffer for it, so now we need to release it.
5542              */
5543             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5544                 l2arc_release_cdata_buf(ab);
5545             }
5546
5547             /*
5548              * If the data was compressed, then we've allocated a
5549              * temporary buffer for it, so now we need to release it.
5550              */
5551             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5552                 l2arc_release_cdata_buf(ab);
5553             }
5554
5555             /*
5556              * If the data was compressed, then we've allocated a
5557              * temporary buffer for it, so now we need to release it.
5558              */
5559             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5560                 l2arc_release_cdata_buf(ab);
5561             }
5562
5563             /*
5564              * If the data was compressed, then we've allocated a
5565              * temporary buffer for it, so now we need to release it.
5566              */
5567             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5568                 l2arc_release_cdata_buf(ab);
5569             }
5570
5571             /*
5572              * If the data was compressed, then we've allocated a
5573              * temporary buffer for it, so now we need to release it.
5574              */
5575             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5576                 l2arc_release_cdata_buf(ab);
5577             }
5578
5579             /*
5580              * If the data was compressed, then we've allocated a
5581              * temporary buffer for it, so now we need to release it.
5582              */
5583             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5584                 l2arc_release_cdata_buf(ab);
5585             }
5586
5587             /*
5588              * If the data was compressed, then we've allocated a
5589              * temporary buffer for it, so now we need to release it.
5590              */
5591             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5592                 l2arc_release_cdata_buf(ab);
5593             }
5594
5595             /*
5596              * If the data was compressed, then we've allocated a
5597              * temporary buffer for it, so now we need to release it.
5598              */
5599             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5600                 l2arc_release_cdata_buf(ab);
5601             }
5602
5603             /*
5604              * If the data was compressed, then we've allocated a
5605              * temporary buffer for it, so now we need to release it.
5606              */
5607             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5608                 l2arc_release_cdata_buf(ab);
5609             }
5610
5611             /*
5612              * If the data was compressed, then we've allocated a
5613              * temporary buffer for it, so now we need to release it.
5614              */
5615             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5616                 l2arc_release_cdata_buf(ab);
5617             }
5618
5619             /*
5620              * If the data was compressed, then we've allocated a
5621              * temporary buffer for it, so now we need to release it.
5622              */
5623             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5624                 l2arc_release_cdata_buf(ab);
5625             }
5626
5627             /*
5628              * If the data was compressed, then we've allocated a
5629              * temporary buffer for it, so now we need to release it.
5630              */
5631             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5632                 l2arc_release_cdata_buf(ab);
5633             }
5634
5635             /*
5636              * If the data was compressed, then we've allocated a
5637              * temporary buffer for it, so now we need to release it.
5638              */
5639             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5640                 l2arc_release_cdata_buf(ab);
5641             }
5642
5643             /*
5644              * If the data was compressed, then we've allocated a
5645              * temporary buffer for it, so now we need to release it.
5646              */
5647             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5648                 l2arc_release_cdata_buf(ab);
5649             }
5650
5651             /*
5652              * If the data was compressed, then we've allocated a
5653              * temporary buffer for it, so now we need to release it.
5654              */
5655             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5656                 l2arc_release_cdata_buf(ab);
5657             }
5658
5659             /*
5660              * If the data was compressed, then we've allocated a
5661              * temporary buffer for it, so now we need to release it.
5662              */
5663             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5664                 l2arc_release_cdata_buf(ab);
5665             }
5666
5667             /*
5668              * If the data was compressed, then we've allocated a
5669              * temporary buffer for it, so now we need to release it.
5670              */
5671             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5672                 l2arc_release_cdata_buf(ab);
5673             }
5674
5675             /*
5676              * If the data was compressed, then we've allocated a
5677              * temporary buffer for it, so now we need to release it.
5678              */
5679             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5680                 l2arc_release_cdata_buf(ab);
5681             }
5682
5683             /*
5684              * If the data was compressed, then we've allocated a
5685              * temporary buffer for it, so now we need to release it.
5686              */
5687             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5688                 l2arc_release_cdata_buf(ab);
5689             }
5690
5691             /*
5692              * If the data was compressed, then we've allocated a
5693              * temporary buffer for it, so now we need to release it.
5694              */
5695             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5696                 l2arc_release_cdata_buf(ab);
5697             }
5698
5699             /*
5700              * If the data was compressed, then we've allocated a
5701              * temporary buffer for it, so now we need to release it.
5702              */
5703             if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5704                 l2arc_release_cdata_buf(ab);
5705             }
5706
5707             /*
5708              * If the data was compressed, then we've allocated a
5709              * temporary buffer for it
```

```
4961      /*
4962       * Avoid contributing to memory pressure.
4963       */
4964     if (arc_reclaim_needed()) {
4965         ARCSTAT_BUMP(arcstat_l2_abort_lowmem);
4966         spa_config_exit(spa, SCL_L2ARC, dev);
4967         continue;
4968     }
4970     ARCSTAT_BUMP(arcstat_l2_feeds);
4972     size = l2arc_write_size();
4974     /*
4975      * Evict L2ARC buffers that will be overwritten.
4976      */
4977     l2arc_evict(dev, size, B_FALSE);
4979     /*
4980      * Write ARC buffers.
4981      */
4982     wrote = l2arc_write_buffers(spa, dev, size, &headroom_boost);
4984     /*
4985      * Calculate interval between writes.
4986      */
4987     next = l2arc_write_interval(begin, size, wrote);
4988     spa_config_exit(spa, SCL_L2ARC, dev);
4989 }
4991 l2arc_thread_exit = 0;
4992 cv_broadcast(&l2arc_feed_thr_cv);
4993 CALLB_CPR_EXIT(&cpr);           /* drops l2arc_feed_thr_lock */
4994 thread_exit();
4995 }
```

unchanged\_portion\_omitted