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

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



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

/*
 * The locking model:
 *
 * A new reference to a cache buffer can be obtained in two
 * ways: 1) via a hash table lookup using the DVA as a key,
 * or 2) via one of the ARC lists.  The arc_read() interface
 * uses method 1, while the internal arc algorithms for
 * adjusting the cache use method 2.  We therefore provide two
 * types of locks: 1) the hash table lock array, and 2) the
 * arc list locks.
 *
 * Buffers do not have their own mutexes, rather they rely on the
 * hash table mutexes for the bulk of their protection (i.e. most
 * fields in the arc_buf_hdr_t are protected by these mutexes).
 *
 * buf_hash_find() returns the appropriate mutex (held) when it
 * locates the requested buffer in the hash table.  It returns
 * NULL for the mutex if the buffer was not in the table.
 *
 * buf_hash_remove() expects the appropriate hash mutex to be
 * already held before it is invoked.
 *
 * Each arc state also has a mutex which is used to protect the
 * buffer list associated with the state.  When attempting to
 * obtain a hash table lock while holding an arc list lock you
 * must use: mutex_tryenter() to avoid deadlock.  Also note that
 * the active state mutex must be held before the ghost state mutex.
 *
 * Arc buffers may have an associated eviction callback function.
 * This function will be invoked prior to removing the buffer (e.g.
 * in arc_do_user_evicts()).  Note however that the data associated
 * with the buffer may be evicted prior to the callback.  The callback
 * must be made with *no locks held* (to prevent deadlock).  Additionally,
 * the users of callbacks must ensure that their private data is
 * protected from simultaneous callbacks from arc_buf_evict()
 * and arc_do_user_evicts().
 *
 * Note that the majority of the performance stats are manipulated
 * with atomic operations.
 *
 * The L2ARC uses the l2arc_buflist_mtx global mutex for the following:
 *
 *      - L2ARC buflist creation
 *      - L2ARC buflist eviction
 *      - L2ARC write completion, which walks L2ARC buflists
 *      - ARC header destruction, as it removes from L2ARC buflists
 *      - ARC header release, as it removes from L2ARC buflists
 *
 * Please note that if you first grab the l2arc_buflist_mtx, you can't do a
 * mutex_enter on a buffer's hash_lock anymore due to lock inversion. To grab
 * the hash_lock you must use mutex_tryenter and possibly deal with the buffer
 * not being available (due to e.g. some other thread holding it while trying
 * to unconditionally grab the l2arc_buflist_mtx which you are holding). The
 * inverse situation (first grab hash_lock, then l2arc_buflist_mtx) is safe.
```

```
 127  */

 129 #include <sys/spa.h>
 130 #include <sys/zio.h>
 131 #include <sys/zio_compress.h>
 132 #include <sys/zfs_context.h>
 133 #include <sys/arc.h>
 134 #include <sys/refcount.h>
 135 #include <sys/vdev.h>
 136 #include <sys/vdev_impl.h>
 137 #ifdef _KERNEL
 138 #include <sys/vmsystm.h>
 139 #include <vm/anon.h>
 140 #include <sys/fs/swapnode.h>
 141 #include <sys/dnlc.h>
 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_procfd;
 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(spa, dva, birth) \
 598         (buf_hash(spa, 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_spa, &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 */

 620 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
 621 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

 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 */
 625 boolean_t l2arc_norw = B_TRUE;                  /* no reads during writes */

 634 /*
 635  * L2ARC Internals
 636  */
 637 typedef struct l2arc_dev {
 638         vdev_t                  *l2ad_vdev;     /* vdev */
 639         spa_t                   *l2ad_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_

 674 struct l2arc_buf_hdr {
 675         /* protected by arc_buf_hdr  mutex */
 676         l2arc_dev_t             *b_dev;         /* L2ARC device */
 677         uint64_t                b_daddr;        /* disk address, offset byte */
 678         /* compression applied to buffer data */
 679         enum zio_compress       b_compress;
 680         /* real alloc'd buffer size depending on b_compress applied */
 681         int                     b_asize;
 675         /* temporary buffer holder for in-flight compressed data */
 676         void                    *b_tmp_cdata;
 682 };
______unchanged_portion_omitted_

 692 static kmutex_t l2arc_feed_thr_lock;
 693 static kcondvar_t l2arc_feed_thr_cv;
 694 static uint8_t l2arc_thread_exit;

 696 static void l2arc_read_done(zio_t *zio);
 697 static void l2arc_hdr_stat_add(void);
 698 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);
 695 static boolean_t l2arc_compress_buf(l2arc_buf_hdr_t *l2hdr);
 702 static void l2arc_decompress_zio(zio_t *zio, arc_buf_hdr_t *hdr,
 703     enum zio_compress c);
 698 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         ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);

 714         for (i = 0; i < sizeof (dva_t); i++)
 715                 crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ vdva[i]) & 0xFF];

 717         crc ^= (spa>>8) ^ birth;

 719         return (crc);
 720 }
_____unchanged_portion_omitted_

4121 /*
4122  * Free buffers that were tagged for destruction.
4123  */
4124 static void
4125 l2arc_do_free_on_write(void)
4120 l2arc_do_free_on_write()
4126 {
4127         list_t *buflist;
4128         l2arc_data_free_t *df, *df_prev;

4130         mutex_enter(&l2arc_free_on_write_mtx);
4131         buflist = l2arc_free_on_write;

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         }

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;
4150         arc_buf_hdr_t *head, *ab, *ab_prev;
4151         l2arc_buf_hdr_t *abl2;
4152         kmutex_t *hash_lock;

4157         struct defer_done_entry {
4158                 arc_buf_hdr_t *dde_buf;
4159                 list_node_t dde_node;
4160         } *dde, *dde_next;
4161         list_t defer_done_list;

4163         cb = zio->io_private;
4164         ASSERT(cb != NULL);
4165         dev = cb->l2wcb_dev;
4166         ASSERT(dev != NULL);
4167         head = cb->l2wcb_head;
4168         ASSERT(head != NULL);
```



```
4169         buflist = dev->l2ad_buflist;
4170         ASSERT(buflist != NULL);
4171         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4172             l2arc_write_callback_t *, cb);

4174         if (zio->io_error != 0)
4175                 ARCSTAT_BUMP(arcstat_l2_writes_error);

4177         mutex_enter(&l2arc_buflist_mtx);

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)) {
4173         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4174                 ab_prev = list_prev(buflist, ab);

4176                 hash_lock = HDR_LOCK(ab);
4177                 if (!mutex_tryenter(hash_lock)) {
4185                 /*
4186                  * Can't pause here to grab hash_lock while also holding
4187                  * l2arc_buflist_mtx, so place the buffers on a temporary
4188                  * thread-local list for later processing.
4179                  * This buffer misses out.  It may be in a stage
4180                  * of eviction.  Its ARC_L2_WRITING flag will be
4181                  * left set, denying reads to this buffer.
4189                  */
4190                 dde = kmem_alloc(sizeof (*dde), KM_SLEEP);
4191                 dde->dde_buf = ab;
4192                 list_insert_tail(&defer_done_list, dde);
4183                 ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
4184                 continue;
4193         }

4195         atomic_inc_64(&l2arc_writes_done);
4196         list_remove(buflist, head);
4197         kmem_cache_free(hdr_cache, head);
4198         mutex_exit(&l2arc_buflist_mtx);
4187                 abl2 = ab->b_l2hdr;

4200         /*
4201          * Now process the buffers. We're not holding l2arc_buflist_mtx
4202          * anymore, so we can do a regular mutex_enter on the hash_lock.
4190                  * Release the temporary compressed buffer as soon as possible.
4203          */
4204         for (dde = list_head(&defer_done_list); dde != NULL; dde = dde_next) {
4205                 kmutex_t *hash_lock;
4192                 if (abl2->b_compress != ZIO_COMPRESS_OFF)
4193                         l2arc_release_cdata_buf(ab);

4207                 dde_next = list_next(&defer_done_list, dde);
4208                 ab = dde->dde_buf;
4209                 hash_lock = HDR_LOCK(ab);

4211                 mutex_enter(hash_lock);

4213                 if (zio->io_error != 0) {
4214                         /*
4215                          * Error - drop L2ARC entry.
4216                          */
4217                         l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;
4218                         mutex_enter(&l2arc_buflist_mtx);
4219                         list_remove(buflist, ab);
4220                         mutex_exit(&l2arc_buflist_mtx);
```

```
4221                        ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4200                        ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4222                        ab->b_l2hdr = NULL;
4223                        kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4202                        kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4224                        ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4225                }

4227                /*
4228                 * Allow ARC to begin reads to this L2ARC entry.
4229                 */
4230                ab->b_flags &= ~ARC_L2_WRITING;

4232                mutex_exit(hash_lock);

4234                list_remove(&defer_done_list, dde);
4235        }
4236        list_destroy(&defer_done_list);

4214        atomic_inc_64(&l2arc_writes_done);
4215        list_remove(buflist, head);
4216        kmem_cache_free(hdr_cache, head);
4217        mutex_exit(&l2arc_buflist_mtx);

4238        l2arc_do_free_on_write();

4240        kmem_free(cb, sizeof (l2arc_write_callback_t));
4241 }
_____unchanged_portion_omitted_

4362 /*
4363  * Evict buffers from the device write hand to the distance specified in
4364  * bytes.  This distance may span populated buffers, it may span nothing.
4365  * This is clearing a region on the L2ARC device ready for writing.
4366  * If the 'all' boolean is set, every buffer is evicted.
4367  */
4368 static void
4369 l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
4370 {
4371        list_t *buflist;
4372        l2arc_buf_hdr_t *l2hdr;
4353        l2arc_buf_hdr_t *abl2;
4373        arc_buf_hdr_t *ab, *ab_prev;
4374        kmutex_t *hash_lock;
4375        uint64_t taddr;

4377        buflist = dev->l2ad_buflist;

4379        if (buflist == NULL)
4380                return;

4382        if (!all && dev->l2ad_first) {
4383                /*
4384                 * This is the first sweep through the device.  There is
4385                 * nothing to evict.
4386                 */
4387                return;
4388        }

4390        if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
4391                /*
4392                 * When nearing the end of the device, evict to the end
4393                 * before the device write hand jumps to the start.
4394                 */
4395                taddr = dev->l2ad_end;
4396        } else {
```



```
4397                taddr = dev->l2ad_hand + distance;
4398        }
4399        DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
4400            uint64_t, taddr, boolean_t, all);

4402 top:
4403        mutex_enter(&l2arc_buflist_mtx);
4404        for (ab = list_tail(buflist); ab; ab = ab_prev) {
4405                ab_prev = list_prev(buflist, ab);

4407                hash_lock = HDR_LOCK(ab);
4408                if (!mutex_tryenter(hash_lock)) {
4409                        /*
4410                         * Missed the hash lock.  Retry.
4411                         */
4412                        ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
4413                        mutex_exit(&l2arc_buflist_mtx);
4414                        mutex_enter(hash_lock);
4415                        mutex_exit(hash_lock);
4416                        goto top;
4417                }

4419                if (HDR_L2_WRITE_HEAD(ab)) {
4420                        /*
4421                         * We hit a write head node.  Leave it for
4422                         * l2arc_write_done().
4423                         */
4424                        list_remove(buflist, ab);
4425                        mutex_exit(hash_lock);
4426                        continue;
4427                }

4429                if (!all && ab->b_l2hdr != NULL &&
4430                    (ab->b_l2hdr->b_daddr > taddr ||
4431                    ab->b_l2hdr->b_daddr < dev->l2ad_hand)) {
4432                        /*
4433                         * We've evicted to the target address,
4434                         * or the end of the device.
4435                         */
4436                        mutex_exit(hash_lock);
4437                        break;
4438                }

4440                if (HDR_FREE_IN_PROGRESS(ab)) {
4441                        /*
4442                         * Already on the path to destruction.
4443                         */
4444                        mutex_exit(hash_lock);
4445                        continue;
4446                }

4448                if (ab->b_state == arc_l2c_only) {
4449                        ASSERT(!HDR_L2_READING(ab));
4450                        /*
4451                         * This doesn't exist in the ARC.  Destroy.
4452                         * arc_hdr_destroy() will call list_remove()
4453                         * and decrement arcstat_l2_size.
4454                         */
4455                        arc_change_state(arc_anon, ab, hash_lock);
4456                        arc_hdr_destroy(ab);
4457                } else {
4458                        /*
4459                         * Invalidate issued or about to be issued
4460                         * reads, since we may be about to write
4461                         * over this location.
4462                         */
```

```
4463                        if (HDR_L2_READING(ab)) {
4464                                ARCSTAT_BUMP(arcstat_l2_evict_reading);
4465                                ab->b_flags |= ARC_L2_EVICTED;
4466                        }

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);
4453                                abl2 = ab->b_l2hdr;
4454                                ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4474                                ab->b_l2hdr = NULL;
4475                                kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4456                                kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4476                                ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4477                        }
4478                        list_remove(buflist, ab);

4480                        /*
4481                         * This may have been leftover after a
4482                         * failed write.
4483                         */
4484                        ab->b_flags &= ~ARC_L2_WRITING;
4485                }
4486                mutex_exit(hash_lock);
4487        }
4488        mutex_exit(&l2arc_buflist_mtx);

4490        vdev_space_update(dev->l2ad_vdev, -(taddr - dev->l2ad_evict), 0, 0);
4491        dev->l2ad_evict = taddr;
4492 }

4494 /*
4495  * Find and write ARC buffers to the L2ARC device.
4496  *
4497  * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4498  * for reading until they have completed writing.
4499  * The headroom_boost is an in-out parameter used to maintain headroom boost
4500  * state between calls to this function.
4501  *
4502  * Returns the number of bytes actually written (which may be smaller than
4503  * the delta by which the device hand has changed due to alignment).
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);

4530        /* Lower the flag now, we might want to raise it again later. */
4531        *headroom_boost = B_FALSE;

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;

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;

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_buflist_mtx);
4551        for (int try = 0; try <= 3; try++) {
4552                uint64_t passed_sz = 0;

4554                list = l2arc_list_locked(try, &list_lock);

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);

4567                headroom = target_sz * l2arc_headroom;
4568                if (do_headroom_boost)
4569                        headroom = (headroom * l2arc_headroom_boost) / 100;

4571                for (; ab; ab = ab_prev) {
4572                        l2arc_buf_hdr_t *l2hdr;
4573                        kmutex_t *hash_lock;
4546                        uint64_t buf_sz;

4575                        if (arc_warm == B_FALSE)
4576                                ab_prev = list_next(list, ab);
4577                        else
4578                                ab_prev = list_prev(list, ab);

4580                        hash_lock = HDR_LOCK(ab);
4581                        if (!mutex_tryenter(hash_lock)) {
4582                                /*
4583                                 * Skip this buffer rather than waiting.
4584                                 */
4585                                continue;
4586                        }

4588                        passed_sz += ab->b_size;
4589                        if (passed_sz > headroom) {
4590                                /*
```

```
4591                                     * Searched too far.
4592                                     */
4593                                    mutex_exit(hash_lock);
4594                                    break;
4595                            }

4597                            if (!l2arc_write_eligible(guid, ab)) {
4598                                    mutex_exit(hash_lock);
4599                                    continue;
4600                            }

4602                            if ((write_sz + ab->b_size) > target_sz) {
4603                                    full = B_TRUE;
4604                                    mutex_exit(hash_lock);
4605                                    break;
4606                            }

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);

4616                                    cb = kmem_alloc(
4617                                        sizeof (l2arc_write_callback_t), KM_SLEEP);
4618                                    cb->l2wcb_dev = dev;
4619                                    cb->l2wcb_head = head;
4620                                    pio = zio_root(spa, l2arc_write_done, cb,
4621                                        ZIO_FLAG_CANFAIL);
4622                            }

4624                            /*
4625                             * Create and add a new L2ARC header.
4626                             */
4627                            l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
4628                            l2hdr->b_dev = dev;
4629                            ab->b_flags |= ARC_L2_WRITING;
4630                            l2hdr->b_compress = ZIO_COMPRESS_OFF;
4631                            l2hdr->b_asize = ab->b_size;

4633                            /*
4634                             * Temporarily stash the buffer in defer_write_entries.
4605                             * 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
4607                             * there. This is because can't access ab->b_buf
4637                             * without holding the hash_lock, which we in turn
4638                             * can't access without holding the ARC list locks
4639                             * while walking the ARC lists (we want to avoid
4640                             * holding these locks during compression/writing).
4610                             * (which we want to avoid during compression/writing).
4641                             */
4642                            dwe = kmem_alloc(sizeof (*dwe), KM_SLEEP);
4643                            dwe->dwe_buf = ab;
4644                            dwe->dwe_orig_data = ab->b_buf->b_data;
4645                            dwe->dwe_orig_size = ab->b_size;
4612                            l2hdr->b_compress = ZIO_COMPRESS_OFF;
4613                            l2hdr->b_asize = ab->b_size;
4614                            l2hdr->b_tmp_cdata = ab->b_buf->b_data;

4616                            buf_sz = ab->b_size;
4647                            ab->b_l2hdr = l2hdr;

4649                            list_insert_head(dev->l2ad_buflist, ab);
```



```
4650                            list_insert_tail(&defer_write_list, dwe);

4652                            /*
4653                             * Compute and store the buffer cksum before
4654                             * writing.  On debug the cksum is verified first.
4655                             */
4656                            arc_cksum_verify(ab->b_buf);
4657                            arc_cksum_compute(ab->b_buf, B_TRUE);

4659                            mutex_exit(hash_lock);

4661                            write_sz += dwe->dwe_orig_size;
4630                            write_sz += buf_sz;
4662                    }

4664                    mutex_exit(list_lock);

4666                    if (full == B_TRUE)
4667                            break;
4668            }

4670            /* No buffers selected for writing? */
4671            if (pio == NULL) {
4672                    ASSERT0(write_sz);
4673                    mutex_exit(&l2arc_buflist_mtx);
4674                    kmem_cache_free(hdr_cache, head);
4675                    list_destroy(&defer_write_list);
4676                    return (0);
4677            }

4679            mutex_exit(&l2arc_buflist_mtx);

4681            /*
4682             * Now start writing the buffers. We're starting at the write head
4683             * and work backwards, retracing the course of the buffer selector
4684             * loop above.
4685             */
4686            for (dwe = list_head(&defer_write_list); dwe != NULL; dwe = dwe_next) {
4652            for (ab = list_prev(dev->l2ad_buflist, head); ab;
4653                ab = list_prev(dev->l2ad_buflist, ab)) {
4687                    l2arc_buf_hdr_t *l2hdr;
4688                    uint64_t buf_sz;

4690                    dwe_next = list_next(&defer_write_list, dwe);
4691                    ab = dwe->dwe_buf;

4693                    /*
4694                     * Accessing ab->b_l2hdr without locking is safe here because
4695                     * we're holding the l2arc_buflist_mtx and no other thread will
4696                     * ever directly modify the L2 fields. In particular ab->b_buf
4697                     * may be invalid by now due to ARC eviction.
4658                     * We shouldn't need to lock the buffer here, since we flagged
4659                     * it as ARC_L2_WRITING in the previous step, but we must take
4660                     * care to only access its L2 cache parameters. In particular,
4661                     * ab->b_buf may be invalid by now due to ARC eviction.
4698                     */
4699                    l2hdr = ab->b_l2hdr;
4700                    l2hdr->b_daddr = dev->l2ad_hand;

4702                    if ((ab->b_flags & ARC_L2COMPRESS) &&
4703                        l2hdr->b_asize >= buf_compress_minsz &&
4704                        l2arc_compress_buf(dwe->dwe_orig_data, dwe->dwe_orig_size,
4705                        &buf_data, &buf_sz, &l2hdr->b_compress)) {
4667                        l2hdr->b_asize >= buf_compress_minsz) {
4668                            if (l2arc_compress_buf(l2hdr)) {
4706                            /*
```

```
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                }
4675                }

4677                /*
4678                 * Pick up the buffer data we had previously stashed away
4679                 * (and now potentially also compressed).
4680                 */
4681                buf_data = l2hdr->b_tmp_cdata;
4682                buf_sz = l2hdr->b_asize;

4718                /* Compression may have squashed the buffer to zero length. */
4719                if (buf_sz != 0) {
4720                        uint64_t buf_p_sz;

4722                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
4723                            dev->l2ad_hand, l2hdr->b_asize, buf_data,
4724                            ZIO_CHECKSUM_OFF, NULL, NULL,
4725                            ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL,
4726                            B_FALSE);
4689                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
4690                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
4691                            ZIO_FLAG_CANFAIL, B_FALSE);

4728                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
4729                            zio_t *, wzio);
4730                        (void) zio_nowait(wzio);

4732                        write_asize += l2hdr->b_asize;
4697                        write_asize += buf_sz;
4733                        /*
4734                         * Keep the clock hand suitably device-aligned.
4735                         */
4736                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4737                        write_psize += buf_p_sz;
4738                        dev->l2ad_hand += buf_p_sz;
4739                }

4741                list_remove(&defer_write_list, dwe);
4742                kmem_free(dwe, sizeof (*dwe));
4743        }

4745        list_destroy(&defer_write_list);
4707        mutex_exit(&l2arc_buflist_mtx);

4747        ASSERT3U(write_asize, <=, target_sz);
4748        ARCSTAT_BUMP(arcstat_l2_writes_sent);
4749        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
4750        ARCSTAT_INCR(arcstat_l2_size, write_sz);
4751        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
4752        vdev_space_update(dev->l2ad_vdev, write_psize, 0, 0);

4754        /*
4755         * Bump device hand to the device start if it is approaching the end.
4756         * l2arc_evict() will already have evicted ahead for this case.
4757         */
4758        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
4759                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        }

4766        dev->l2ad_writing = B_TRUE;
4767        (void) zio_wait(pio);
4768        dev->l2ad_writing = B_FALSE;

4770        return (write_asize);
4771 }

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.
4737  * The data to be compressed must be prefilled in l2hdr->b_tmp_cdata and its
4738  * size in l2hdr->b_asize. This routine tries to compress the data and
4739  * depending on the compression result there are three possible outcomes:
4740  * *) The buffer was incompressible. The original l2hdr contents were left
4741  *    untouched and are ready for writing to an L2 device.
4780  * *) The buffer was all-zeros, so there is no need to write it to an L2
4781  *    device. To indicate this situation, the *out_data is set to NULL,
4782  *    *out_sz is set to zero, *compress is set to ZIO_COMPRESS_EMPTY and
4783  *    the function returns B_TRUE.
4784  * *) Compression succeeded and *out_data was set to point to a buffer holding
4785  *    the compressed data buffer, *out_sz was set to indicate the output size,
4786  *    *compress was set to the appropriate compression algorithm and B_TRUE is
4787  *    returned. Once writing is done the buffer will be automatically freed by
4788  *    l2arc_do_free_on_write().
4743  *    device. To indicate this situation b_tmp_cdata is NULL'ed, b_asize is
4744  *    set to zero and b_compress is set to ZIO_COMPRESS_EMPTY.
4745  * *) Compression succeeded and b_tmp_cdata was replaced with a temporary
4746  *    data buffer which holds the compressed data to be written, and b_asize
4747  *    tells us how much data there is. b_compress is set to the appropriate
4748  *    compression algorithm. Once writing is done, invoke
4749  *    l2arc_release_cdata_buf on this l2hdr to free this temporary buffer.
4750  *
4751  * Returns B_TRUE if compression succeeded, or B_FALSE if it didn't (the
4752  * buffer was incompressible).
4789  */
4790 static boolean_t
4791 l2arc_compress_buf(void *in_data, uint64_t in_sz, void **out_data,
4792     uint64_t *out_sz, enum zio_compress *compress)
4755 l2arc_compress_buf(l2arc_buf_hdr_t *l2hdr)
4793 {
4794        void *cdata;
4758        size_t csize, len;

4796        cdata = zio_data_buf_alloc(in_sz);
4797        *out_sz = zio_compress_data(ZIO_COMPRESS_LZ4, in_data, cdata, in_sz);
4760        ASSERT(l2hdr->b_compress == ZIO_COMPRESS_OFF);
4761        ASSERT(l2hdr->b_tmp_cdata != NULL);

4799        if (*out_sz == 0) {
4800                /* Zero block, indicate that there's nothing to write. */
4801                zio_data_buf_free(cdata, in_sz);
4802                *compress = ZIO_COMPRESS_EMPTY;
4803                *out_data = NULL;
4763        len = l2hdr->b_asize;
4764        cdata = zio_data_buf_alloc(len);
4765        csize = zio_compress_data(ZIO_COMPRESS_LZ4, l2hdr->b_tmp_cdata,
```

```
4766             cdata, l2hdr->b_asize);

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;
4804                 ARCSTAT_BUMP(arcstat_l2_compress_zeros);
4805                 return (B_TRUE);
4806         } else if (*out_sz > 0 && *out_sz < in_sz) {
4776         } else if (csize > 0 && csize < len) {
4807                 /*
4808                  * Compression succeeded, we'll keep the cdata around for
4809                  * writing and release it after writing.
4779                  * writing and release it afterwards.
4810                  */
4811                 l2arc_data_free_t *df;

4813                 *compress = ZIO_COMPRESS_LZ4;
4814                 *out_data = cdata;

4816                 df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
4817                 df->l2df_data = cdata;
4818                 df->l2df_size = *out_sz;
4819                 df->l2df_func = zio_data_buf_free;
4820                 mutex_enter(&l2arc_free_on_write_mtx);
4821                 list_insert_head(l2arc_free_on_write, df);
4822                 mutex_exit(&l2arc_free_on_write_mtx);

4781                 l2hdr->b_compress = ZIO_COMPRESS_LZ4;
4782                 l2hdr->b_asize = csize;
4783                 l2hdr->b_tmp_cdata = cdata;
4824                 ARCSTAT_BUMP(arcstat_l2_compress_successes);
4825                 ARCSTAT_BUMP(arcstat_l2_free_on_write);
4826                 return (B_TRUE);
4827         } else {
4828                 /*
4829                  * Compression failed, release the compressed buffer.
4789                  * l2hdr will be left unmodified.
4830                  */
4831                 zio_data_buf_free(cdata, in_sz);
4791                 zio_data_buf_free(cdata, len);
4832                 ARCSTAT_BUMP(arcstat_l2_compress_failures);
4833                 return (B_FALSE);
4834         }
4835 }
```

_____unchanged_portion_omitted_

```
4859 /*
4860  * Releases the temporary b_tmp_cdata buffer in an l2arc header structure.
4861  * This buffer serves as a temporary holder of compressed data while
4862  * the buffer entry is being written to an l2arc device. Once that is
4863  * done, we can dispose of it.
4864  */
4865 static void
4866 l2arc_release_cdata_buf(arc_buf_hdr_t *ab)
4867 {
4868         l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;

4870         if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
4871                 /*
4872                  * If the data was compressed, then we've allocated a
4873                  * temporary buffer for it, so now we need to release it.
4874                  */
4875                 ASSERT(l2hdr->b_tmp_cdata != NULL);
```



```
4876                 zio_data_buf_free(l2hdr->b_tmp_cdata, ab->b_size);
4877         }
4878         l2hdr->b_tmp_cdata = NULL;
4879 }

4899 /*
4900  * This thread feeds the L2ARC at regular intervals.  This is the beating
4901  * heart of the L2ARC.
4902  */
4903 static void
4904 l2arc_feed_thread(void)
4905 {
4906         callb_cpr_t cpr;
4907         l2arc_dev_t *dev;
4908         spa_t *spa;
4909         uint64_t size, wrote;
4910         clock_t begin, next = ddi_get_lbolt();
4911         boolean_t headroom_boost = B_FALSE;

4913         CALLB_CPR_INIT(&cpr, &l2arc_feed_thr_lock, callb_generic_cpr, FTAG);

4915         mutex_enter(&l2arc_feed_thr_lock);

4917         while (l2arc_thread_exit == 0) {
4918                 CALLB_CPR_SAFE_BEGIN(&cpr);
4919                 (void) cv_timedwait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock,
4920                     next);
4921                 CALLB_CPR_SAFE_END(&cpr, &l2arc_feed_thr_lock);
4922                 next = ddi_get_lbolt() + hz;

4924                 /*
4925                  * Quick check for L2ARC devices.
4926                  */
4927                 mutex_enter(&l2arc_dev_mtx);
4928                 if (l2arc_ndev == 0) {
4929                         mutex_exit(&l2arc_dev_mtx);
4930                         continue;
4931                 }
4932                 mutex_exit(&l2arc_dev_mtx);
4933                 begin = ddi_get_lbolt();

4935                 /*
4936                  * This selects the next l2arc device to write to, and in
4937                  * doing so the next spa to feed from: dev->l2ad_spa.   This
4938                  * will return NULL if there are now no l2arc devices or if
4939                  * they are all faulted.
4940                  *
4941                  * If a device is returned, its spa's config lock is also
4942                  * held to prevent device removal.  l2arc_dev_get_next()
4943                  * will grab and release l2arc_dev_mtx.
4944                  */
4945                 if ((dev = l2arc_dev_get_next()) == NULL)
4946                         continue;

4948                 spa = dev->l2ad_spa;
4949                 ASSERT(spa != NULL);

4951                 /*
4952                  * If the pool is read-only then force the feed thread to
4953                  * sleep a little longer.
4954                  */
4955                 if (!spa_writeable(spa)) {
4956                         next = ddi_get_lbolt() + 5 * l2arc_feed_secs * hz;
4957                         spa_config_exit(spa, SCL_L2ARC, dev);
4958                         continue;
4959                 }
```

```
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_***