```
**********************************************************
  146303 Tue Aug  6 16:31:03 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
**********************************************************
______unchanged_portion_omitted_

 234 /* The 6 states: */
 235 static arc_state_t ARC_anon;
 236 static arc_state_t ARC_mru;
 237 static arc_state_t ARC_mru_ghost;
 238 static arc_state_t ARC_mfu;
 239 static arc_state_t ARC_mfu_ghost;
 240 static arc_state_t ARC_l2c_only;

 242 typedef struct arc_stats {
 243         kstat_named_t arcstat_hits;
 244         kstat_named_t arcstat_misses;
 245         kstat_named_t arcstat_demand_data_hits;
 246         kstat_named_t arcstat_demand_data_misses;
 247         kstat_named_t arcstat_demand_metadata_hits;
 248         kstat_named_t arcstat_demand_metadata_misses;
 249         kstat_named_t arcstat_prefetch_data_hits;
 250         kstat_named_t arcstat_prefetch_data_misses;
 251         kstat_named_t arcstat_prefetch_metadata_hits;
 252         kstat_named_t arcstat_prefetch_metadata_misses;
 253         kstat_named_t arcstat_mru_hits;
 254         kstat_named_t arcstat_mru_ghost_hits;
 255         kstat_named_t arcstat_mfu_hits;
 256         kstat_named_t arcstat_mfu_ghost_hits;
 257         kstat_named_t arcstat_deleted;
 258         kstat_named_t arcstat_recycle_miss;
 259         /*
 260          * Number of buffers that could not be evicted because the hash lock
 261          * was held by another thread.  The lock may not necessarily be held
 262          * by something using the same buffer, since hash locks are shared
 263          * by multiple buffers.
 264          */
 265         kstat_named_t arcstat_mutex_miss;
 266         /*
 267          * Number of buffers skipped because they have I/O in progress, are
 268          * indirect prefetch buffers that have not lived long enough, or are
 269          * not from the spa we're trying to evict from.
 270          */
 271         kstat_named_t arcstat_evict_skip;
 272         kstat_named_t arcstat_evict_l2_cached;
 273         kstat_named_t arcstat_evict_l2_eligible;
 274         kstat_named_t arcstat_evict_l2_ineligible;
 275         kstat_named_t arcstat_hash_elements;
 276         kstat_named_t arcstat_hash_elements_max;
 277         kstat_named_t arcstat_hash_collisions;
 278         kstat_named_t arcstat_hash_chains;
 279         kstat_named_t arcstat_hash_chain_max;
 280         kstat_named_t arcstat_p;
 281         kstat_named_t arcstat_c;
 282         kstat_named_t arcstat_c_min;
 283         kstat_named_t arcstat_c_max;
 284         kstat_named_t arcstat_size;
 285         kstat_named_t arcstat_hdr_size;
 286         kstat_named_t arcstat_data_size;
 287         kstat_named_t arcstat_other_size;
 288         kstat_named_t arcstat_l2_hits;
 289         kstat_named_t arcstat_l2_misses;
 290         kstat_named_t arcstat_l2_feeds;
 291         kstat_named_t arcstat_l2_rw_clash;
```



```
 292         kstat_named_t arcstat_l2_read_bytes;
 293         kstat_named_t arcstat_l2_write_bytes;
 294         kstat_named_t arcstat_l2_writes_sent;
 295         kstat_named_t arcstat_l2_writes_done;
 296         kstat_named_t arcstat_l2_writes_error;
 297         kstat_named_t arcstat_l2_writes_hdr_miss;
 297         kstat_named_t arcstat_l2_evict_lock_retry;
 298         kstat_named_t arcstat_l2_evict_reading;
 299         kstat_named_t arcstat_l2_free_on_write;
 300         kstat_named_t arcstat_l2_abort_lowmem;
 301         kstat_named_t arcstat_l2_cksum_bad;
 302         kstat_named_t arcstat_l2_io_error;
 303         kstat_named_t arcstat_l2_size;
 304         kstat_named_t arcstat_l2_asize;
 305         kstat_named_t arcstat_l2_hdr_size;
 306         kstat_named_t arcstat_l2_compress_successes;
 307         kstat_named_t arcstat_l2_compress_zeros;
 308         kstat_named_t arcstat_l2_compress_failures;
 309         kstat_named_t arcstat_memory_throttle_count;
 310         kstat_named_t arcstat_duplicate_buffers;
 311         kstat_named_t arcstat_duplicate_buffers_size;
 312         kstat_named_t arcstat_duplicate_reads;
 313         kstat_named_t arcstat_meta_used;
 314         kstat_named_t arcstat_meta_limit;
 315         kstat_named_t arcstat_meta_max;
 316 } arc_stats_t;

 318 static arc_stats_t arc_stats = {
 319         { "hits",                       KSTAT_DATA_UINT64 },
 320         { "misses",                     KSTAT_DATA_UINT64 },
 321         { "demand_data_hits",           KSTAT_DATA_UINT64 },
 322         { "demand_data_misses",         KSTAT_DATA_UINT64 },
 323         { "demand_metadata_hits",       KSTAT_DATA_UINT64 },
 324         { "demand_metadata_misses",     KSTAT_DATA_UINT64 },
 325         { "prefetch_data_hits",         KSTAT_DATA_UINT64 },
 326         { "prefetch_data_misses",       KSTAT_DATA_UINT64 },
 327         { "prefetch_metadata_hits",     KSTAT_DATA_UINT64 },
 328         { "prefetch_metadata_misses",   KSTAT_DATA_UINT64 },
 329         { "mru_hits",                   KSTAT_DATA_UINT64 },
 330         { "mru_ghost_hits",             KSTAT_DATA_UINT64 },
 331         { "mfu_hits",                   KSTAT_DATA_UINT64 },
 332         { "mfu_ghost_hits",             KSTAT_DATA_UINT64 },
 333         { "deleted",                    KSTAT_DATA_UINT64 },
 334         { "recycle_miss",               KSTAT_DATA_UINT64 },
 335         { "mutex_miss",                 KSTAT_DATA_UINT64 },
 336         { "evict_skip",                 KSTAT_DATA_UINT64 },
 337         { "evict_l2_cached",            KSTAT_DATA_UINT64 },
 338         { "evict_l2_eligible",          KSTAT_DATA_UINT64 },
 339         { "evict_l2_ineligible",        KSTAT_DATA_UINT64 },
 340         { "hash_elements",              KSTAT_DATA_UINT64 },
 341         { "hash_elements_max",          KSTAT_DATA_UINT64 },
 342         { "hash_collisions",            KSTAT_DATA_UINT64 },
 343         { "hash_chains",                KSTAT_DATA_UINT64 },
 344         { "hash_chain_max",             KSTAT_DATA_UINT64 },
 345         { "p",                          KSTAT_DATA_UINT64 },
 346         { "c",                          KSTAT_DATA_UINT64 },
 347         { "c_min",                      KSTAT_DATA_UINT64 },
 348         { "c_max",                      KSTAT_DATA_UINT64 },
 349         { "size",                       KSTAT_DATA_UINT64 },
 350         { "hdr_size",                   KSTAT_DATA_UINT64 },
 351         { "data_size",                  KSTAT_DATA_UINT64 },
 352         { "other_size",                 KSTAT_DATA_UINT64 },
 353         { "l2_hits",                    KSTAT_DATA_UINT64 },
 354         { "l2_misses",                  KSTAT_DATA_UINT64 },
 355         { "l2_feeds",                   KSTAT_DATA_UINT64 },
 356         { "l2_rw_clash",                KSTAT_DATA_UINT64 },
```

```
357         { "l2_read_bytes",              KSTAT_DATA_UINT64 },
358         { "l2_write_bytes",             KSTAT_DATA_UINT64 },
359         { "l2_writes_sent",             KSTAT_DATA_UINT64 },
360         { "l2_writes_done",             KSTAT_DATA_UINT64 },
361         { "l2_writes_error",            KSTAT_DATA_UINT64 },
363         { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
362         { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
363         { "l2_evict_reading",           KSTAT_DATA_UINT64 },
364         { "l2_free_on_write",           KSTAT_DATA_UINT64 },
365         { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
366         { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
367         { "l2_io_error",                KSTAT_DATA_UINT64 },
368         { "l2_size",                    KSTAT_DATA_UINT64 },
369         { "l2_asize",                   KSTAT_DATA_UINT64 },
370         { "l2_hdr_size",                KSTAT_DATA_UINT64 },
371         { "l2_compress_successes",      KSTAT_DATA_UINT64 },
372         { "l2_compress_zeros",          KSTAT_DATA_UINT64 },
373         { "l2_compress_failures",       KSTAT_DATA_UINT64 },
374         { "memory_throttle_count",      KSTAT_DATA_UINT64 },
375         { "duplicate_buffers",          KSTAT_DATA_UINT64 },
376         { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
377         { "duplicate_reads",            KSTAT_DATA_UINT64 },
378         { "arc_meta_used",              KSTAT_DATA_UINT64 },
379         { "arc_meta_limit",             KSTAT_DATA_UINT64 },
380         { "arc_meta_max",               KSTAT_DATA_UINT64 }
381 };
```

*unchanged_portion_omitted_*

```
586 static buf_hash_table_t buf_hash_table;

588 #define BUF_HASH_INDEX(spa, dva, birth) \
589         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
590 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
591 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
592 #define HDR_LOCK(hdr) \
593         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))

595 uint64_t zfs_crc64_table[256];

597 /*
598  * Level 2 ARC
599  */

601 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
602 #define L2ARC_HEADROOM          2                       /* num of writes */
603 /*
604  * If we discover during ARC scan any buffers to be compressed, we boost
605  * our headroom for the next scanning cycle by this percentage multiple.
606  */
607 #define L2ARC_HEADROOM_BOOST    200
608 #define L2ARC_FEED_SECS         1               /* caching interval secs */
609 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

611 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
612 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

614 /* L2ARC Performance Tunables */
615 uint64_t l2arc_write_max = L2ARC_WRITE_SIZE;    /* default max write size */
616 uint64_t l2arc_write_boost = L2ARC_WRITE_SIZE;  /* extra write during warmup */
617 uint64_t l2arc_headroom = L2ARC_HEADROOM;       /* number of dev writes */
618 uint64_t l2arc_headroom_boost = L2ARC_HEADROOM_BOOST;
619 uint64_t l2arc_feed_secs = L2ARC_FEED_SECS;     /* interval seconds */
620 uint64_t l2arc_feed_min_ms = L2ARC_FEED_MIN_MS; /* min interval milliseconds */
621 boolean_t l2arc_noprefetch = B_TRUE;            /* don't cache prefetch bufs */
622 boolean_t l2arc_feed_again = B_TRUE;            /* turbo warmup */
623 boolean_t l2arc_norw = B_FALSE;                 /* no reads during writes */
```



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

625 /*
626  * L2ARC Internals
627  */
628 typedef struct l2arc_dev {
629         vdev_t                  *l2ad_vdev;     /* vdev */
630         spa_t                   *l2ad_spa;      /* spa */
631         uint64_t                l2ad_hand;      /* next write location */
632         uint64_t                l2ad_start;     /* first addr on device */
633         uint64_t                l2ad_end;       /* last addr on device */
634         uint64_t                l2ad_evict;     /* last addr eviction reached */
635         boolean_t               l2ad_first;     /* first sweep through */
636         boolean_t               l2ad_writing;   /* currently writing */
637         list_t                  *l2ad_buflist;  /* buffer list */
638         list_node_t             l2ad_node;      /* device list node */
639 } l2arc_dev_t;
```

*unchanged_portion_omitted_*

```
4138 /*
4139  * A write to a cache device has completed.  Update all headers to allow
4140  * reads from these buffers to begin.
4141  */
4142 static void
4143 l2arc_write_done(zio_t *zio)
4144 {
4145         l2arc_write_callback_t *cb;
4146         l2arc_dev_t *dev;
4147         list_t *buflist;
4148         arc_buf_hdr_t *head, *ab, *ab_prev;
4149         l2arc_buf_hdr_t *l2hdr;
4151         l2arc_buf_hdr_t *abl2;
4150         kmutex_t *hash_lock;

4152         cb = zio->io_private;
4153         ASSERT(cb != NULL);
4154         dev = cb->l2wcb_dev;
4155         ASSERT(dev != NULL);
4156         head = cb->l2wcb_head;
4157         ASSERT(head != NULL);
4158         buflist = dev->l2ad_buflist;
4159         ASSERT(buflist != NULL);
4160         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4161             l2arc_write_callback_t *, cb);

4163         if (zio->io_error != 0)
4164                 ARCSTAT_BUMP(arcstat_l2_writes_error);

4166         mutex_enter(&l2arc_buflist_mtx);

4168         /*
4169          * All writes completed, or an error was hit.
4170          */
4171         dev->l2ad_writing = B_FALSE;
4172         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4173                 ab_prev = list_prev(buflist, ab);

4175                 hash_lock = HDR_LOCK(ab);
4176                 mutex_enter(hash_lock);
4177                 if (!mutex_tryenter(hash_lock)) {
4178                         /*
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.
4182                          */
4183                         ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
```

```
4184                            continue;
4185                    }

4178                    l2hdr = ab->b_l2hdr;
4187                    abl2 = ab->b_l2hdr;

4180                    /*
4181                     * Release the temporary compressed buffer as soon as possible.
4182                     */
4183                    if (l2hdr->b_compress != ZIO_COMPRESS_OFF)
4192                    if (abl2->b_compress != ZIO_COMPRESS_OFF)
4184                            l2arc_release_cdata_buf(ab);

4186                    if (zio->io_error != 0) {
4187                            /*
4188                             * Error - drop L2ARC entry.
4189                             */
4190                            list_remove(buflist, ab);
4191                            ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4200                            ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4192                            ab->b_l2hdr = NULL;
4193                            kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4202                            kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4194                            ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4195                    }

4197                    /*
4198                     * Allow ARC to begin reads to this L2ARC entry.
4199                     */
4200                    ab->b_flags &= ~ARC_L2_WRITING;

4202                    mutex_exit(hash_lock);
4203            }

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);

4210            l2arc_do_free_on_write();

4212            kmem_free(cb, sizeof (l2arc_write_callback_t));
4213    }
______unchanged_portion_omitted_

4334    /*
4335     * Evict buffers from the device write hand to the distance specified in
4336     * bytes.  This distance may span populated buffers, it may span nothing.
4337     * This is clearing a region on the L2ARC device ready for writing.
4338     * If the 'all' boolean is set, every buffer is evicted.
4339     */
4340    static void
4341    l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
4342    {
4343            list_t *buflist;
4344            l2arc_buf_hdr_t *l2hdr;
4353            l2arc_buf_hdr_t *abl2;
4345            arc_buf_hdr_t *ab, *ab_prev;
4346            kmutex_t *hash_lock;
4347            uint64_t taddr;

4349            buflist = dev->l2ad_buflist;

4351            if (buflist == NULL)
4352                    return;
```



```
4354            if (!all && dev->l2ad_first) {
4355                    /*
4356                     * This is the first sweep through the device.  There is
4357                     * nothing to evict.
4358                     */
4359                    return;
4360            }

4362            if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
4363                    /*
4364                     * When nearing the end of the device, evict to the end
4365                     * before the device write hand jumps to the start.
4366                     */
4367                    taddr = dev->l2ad_end;
4368            } else {
4369                    taddr = dev->l2ad_hand + distance;
4370            }
4371            DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
4372                uint64_t, taddr, boolean_t, all);

4374    top:
4375            mutex_enter(&l2arc_buflist_mtx);
4376            for (ab = list_tail(buflist); ab; ab = ab_prev) {
4377                    ab_prev = list_prev(buflist, ab);

4379                    hash_lock = HDR_LOCK(ab);
4380                    if (!mutex_tryenter(hash_lock)) {
4381                            /*
4382                             * Missed the hash lock.  Retry.
4383                             */
4384                            ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
4385                            mutex_exit(&l2arc_buflist_mtx);
4386                            mutex_enter(hash_lock);
4387                            mutex_exit(hash_lock);
4388                            goto top;
4389                    }

4391                    if (HDR_L2_WRITE_HEAD(ab)) {
4392                            /*
4393                             * We hit a write head node.  Leave it for
4394                             * l2arc_write_done().
4395                             */
4396                            list_remove(buflist, ab);
4397                            mutex_exit(hash_lock);
4398                            continue;
4399                    }

4401                    if (!all && ab->b_l2hdr != NULL &&
4402                        (ab->b_l2hdr->b_daddr > taddr ||
4403                        ab->b_l2hdr->b_daddr < dev->l2ad_hand)) {
4404                            /*
4405                             * We've evicted to the target address,
4406                             * or the end of the device.
4407                             */
4408                            mutex_exit(hash_lock);
4409                            break;
4410                    }

4412                    if (HDR_FREE_IN_PROGRESS(ab)) {
4413                            /*
4414                             * Already on the path to destruction.
4415                             */
4416                            mutex_exit(hash_lock);
4417                            continue;
4418                    }
```

```
4420                 if (ab->b_state == arc_l2c_only) {
4421                         ASSERT(!HDR_L2_READING(ab));
4422                         /*
4423                          * This doesn't exist in the ARC.  Destroy.
4424                          * arc_hdr_destroy() will call list_remove()
4425                          * and decrement arcstat_l2_size.
4426                          */
4427                         arc_change_state(arc_anon, ab, hash_lock);
4428                         arc_hdr_destroy(ab);
4429                 } else {
4430                         /*
4431                          * Invalidate issued or about to be issued
4432                          * reads, since we may be about to write
4433                          * over this location.
4434                          */
4435                         if (HDR_L2_READING(ab)) {
4436                                 ARCSTAT_BUMP(arcstat_l2_evict_reading);
4437                                 ab->b_flags |= ARC_L2_EVICTED;
4438                         }
4440                         /*
4441                          * Tell ARC this no longer exists in L2ARC.
4442                          */
4443                         if (ab->b_l2hdr != NULL) {
4444                                 l2hdr = ab->b_l2hdr;
4445                                 ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4453                                 abl2 = ab->b_l2hdr;
4454                                 ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4446                                 ab->b_l2hdr = NULL;
4447                                 kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
4456                                 kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4448                                 ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4449                         }
4450                         list_remove(buflist, ab);
4452                         /*
4453                          * This may have been leftover after a
4454                          * failed write.
4455                          */
4456                         ab->b_flags &= ~ARC_L2_WRITING;
4457                 }
4458                 mutex_exit(hash_lock);
4459         }
4460         mutex_exit(&l2arc_buflist_mtx);
4462         vdev_space_update(dev->l2ad_vdev, -(taddr - dev->l2ad_evict), 0, 0);
4463         dev->l2ad_evict = taddr;
4464 }
4466 /*
4467  * Find and write ARC buffers to the L2ARC device.
4468  *
4469  * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4470  * for reading until they have completed writing.
4471  * The headroom_boost is an in-out parameter used to maintain headroom boost
4472  * state between calls to this function.
4473  *
4474  * Returns the number of bytes actually written (which may be smaller than
4475  * the delta by which the device hand has changed due to alignment).
4476  */
4477 static uint64_t
4478 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
4479     boolean_t *headroom_boost)
4480 {
4481         arc_buf_hdr_t *ab, *ab_prev, *head;
4482         list_t *list;
```



```
4483         uint64_t write_asize, write_psize, write_sz, headroom,
4484             buf_compress_minsz;
4485         void *buf_data;
4486         kmutex_t *list_lock;
4487         boolean_t full;
4488         l2arc_write_callback_t *cb;
4489         zio_t *pio, *wzio;
4490         uint64_t guid = spa_load_guid(spa);
4491         const boolean_t do_headroom_boost = *headroom_boost;
4493         ASSERT(dev->l2ad_vdev != NULL);
4495         /* Lower the flag now, we might want to raise it again later. */
4496         *headroom_boost = B_FALSE;
4498         pio = NULL;
4499         write_sz = write_asize = write_psize = 0;
4500         full = B_FALSE;
4501         head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
4502         head->b_flags |= ARC_L2_WRITE_HEAD;
4504         /*
4505          * We will want to try to compress buffers that are at least 2x the
4506          * device sector size.
4507          */
4508         buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;
4510         /*
4511          * Copy buffers for L2ARC writing.
4512          */
4513         mutex_enter(&l2arc_buflist_mtx);
4514         for (int try = 0; try <= 3; try++) {
4515                 uint64_t passed_sz = 0;
4517                 list = l2arc_list_locked(try, &list_lock);
4519                 /*
4520                  * L2ARC fast warmup.
4521                  *
4522                  * Until the ARC is warm and starts to evict, read from the
4523                  * head of the ARC lists rather than the tail.
4524                  */
4525                 if (arc_warm == B_FALSE)
4526                         ab = list_head(list);
4527                 else
4528                         ab = list_tail(list);
4530                 headroom = target_sz * l2arc_headroom;
4531                 if (do_headroom_boost)
4532                         headroom = (headroom * l2arc_headroom_boost) / 100;
4534                 for (; ab; ab = ab_prev) {
4535                         l2arc_buf_hdr_t *l2hdr;
4536                         kmutex_t *hash_lock;
4537                         uint64_t buf_sz;
4539                         if (arc_warm == B_FALSE)
4540                                 ab_prev = list_next(list, ab);
4541                         else
4542                                 ab_prev = list_prev(list, ab);
4544                         hash_lock = HDR_LOCK(ab);
4545                         if (!mutex_tryenter(hash_lock)) {
4546                                 /*
4547                                  * Skip this buffer rather than waiting.
4548                                  */
```

```
4549                            continue;
4550                    }

4552                    passed_sz += ab->b_size;
4553                    if (passed_sz > headroom) {
4554                            /*
4555                             * Searched too far.
4556                             */
4557                            mutex_exit(hash_lock);
4558                            break;
4559                    }

4561                    if (!l2arc_write_eligible(guid, ab)) {
4562                            mutex_exit(hash_lock);
4563                            continue;
4564                    }

4566                    if ((write_sz + ab->b_size) > target_sz) {
4567                            full = B_TRUE;
4568                            mutex_exit(hash_lock);
4569                            break;
4570                    }

4572                    if (pio == NULL) {
4573                            /*
4574                             * Insert a dummy header on the buflist so
4575                             * l2arc_write_done() can find where the
4576                             * write buffers begin without searching.
4577                             */
4578                            list_insert_head(dev->l2ad_buflist, head);

4580                            cb = kmem_alloc(
4581                                sizeof (l2arc_write_callback_t), KM_SLEEP);
4582                            cb->l2wcb_dev = dev;
4583                            cb->l2wcb_head = head;
4584                            pio = zio_root(spa, l2arc_write_done, cb,
4585                                ZIO_FLAG_CANFAIL);
4586                    }

4588                    /*
4589                     * Create and add a new L2ARC header.
4590                     */
4591                    l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
4592                    l2hdr->b_dev = dev;
4593                    ab->b_flags |= ARC_L2_WRITING;

4595                    /*
4596                     * Temporarily stash the data buffer in b_tmp_cdata.
4597                     * The subsequent write step will pick it up from
4598                     * there. This is because can't access ab->b_buf
4599                     * without holding the hash_lock, which we in turn
4600                     * can't access without holding the ARC list locks
4601                     * (which we want to avoid during compression/writing).
4602                     */
4603                    l2hdr->b_compress = ZIO_COMPRESS_OFF;
4604                    l2hdr->b_asize = ab->b_size;
4605                    l2hdr->b_tmp_cdata = ab->b_buf->b_data;

4607                    buf_sz = ab->b_size;
4608                    ab->b_l2hdr = l2hdr;

4610                    list_insert_head(dev->l2ad_buflist, ab);

4612                    /*
4613                     * Compute and store the buffer cksum before
4614                     * writing.  On debug the cksum is verified first.
```



```
4615                             */
4616                            arc_cksum_verify(ab->b_buf);
4617                            arc_cksum_compute(ab->b_buf, B_TRUE);

4619                            mutex_exit(hash_lock);

4621                            write_sz += buf_sz;
4622                    }

4624                    mutex_exit(list_lock);

4626                    if (full == B_TRUE)
4627                            break;
4628            }

4630            /* No buffers selected for writing? */
4631            if (pio == NULL) {
4632                    ASSERT0(write_sz);
4633                    mutex_exit(&l2arc_buflist_mtx);
4634                    kmem_cache_free(hdr_cache, head);
4635                    return (0);
4636            }

4638            /*
4639             * Now start writing the buffers. We're starting at the write head
4640             * and work backwards, retracing the course of the buffer selector
4641             * loop above.
4642             */
4643            for (ab = list_prev(dev->l2ad_buflist, head); ab;
4644                ab = list_prev(dev->l2ad_buflist, ab)) {
4645                    l2arc_buf_hdr_t *l2hdr;
4646                    uint64_t buf_sz;

4648                    /*
4649                     * We shouldn't need to lock the buffer here, since we flagged
4650                     * it as ARC_L2_WRITING in the previous step, but we must take
4651                     * care to only access its L2 cache parameters. In particular,
4652                     * ab->b_buf may be invalid by now due to ARC eviction.
4653                     */
4654                    l2hdr = ab->b_l2hdr;
4655                    l2hdr->b_daddr = dev->l2ad_hand;

4657                    if ((ab->b_flags & ARC_L2COMPRESS) &&
4658                        l2hdr->b_asize >= buf_compress_minsz) {
4659                            if (l2arc_compress_buf(l2hdr)) {
4660                                    /*
4661                                     * If compression succeeded, enable headroom
4662                                     * boost on the next scan cycle.
4663                                     */
4664                                    *headroom_boost = B_TRUE;
4665                            }
4666                    }

4668                    /*
4669                     * Pick up the buffer data we had previously stashed away
4670                     * (and now potentially also compressed).
4671                     */
4672                    buf_data = l2hdr->b_tmp_cdata;
4673                    buf_sz = l2hdr->b_asize;

4675                    /* Compression may have squashed the buffer to zero length. */
4676                    if (buf_sz != 0) {
4677                            uint64_t buf_p_sz;

4679                            wzio = zio_write_phys(pio, dev->l2ad_vdev,
4680                                dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
```

```
4681                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
4682                            ZIO_FLAG_CANFAIL, B_FALSE);

4684                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
4685                            zio_t *, wzio);
4686                        (void) zio_nowait(wzio);

4688                        write_asize += buf_sz;
4689                        /*
4690                         * Keep the clock hand suitably device-aligned.
4691                         */
4692                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4693                        write_psize += buf_p_sz;
4694                        dev->l2ad_hand += buf_p_sz;
4695                }
4696        }

4698        mutex_exit(&l2arc_buflist_mtx);

4700        ASSERT3U(write_asize, <=, target_sz);
4701        ARCSTAT_BUMP(arcstat_l2_writes_sent);
4702        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
4703        ARCSTAT_INCR(arcstat_l2_size, write_sz);
4704        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
4705        vdev_space_update(dev->l2ad_vdev, write_psize, 0, 0);

4707        /*
4708         * Bump device hand to the device start if it is approaching the end.
4709         * l2arc_evict() will already have evicted ahead for this case.
4710         */
4711        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
4712                vdev_space_update(dev->l2ad_vdev,
4713                    dev->l2ad_end - dev->l2ad_hand, 0, 0);
4714                dev->l2ad_hand = dev->l2ad_start;
4715                dev->l2ad_evict = dev->l2ad_start;
4716                dev->l2ad_first = B_FALSE;
4717        }

4719        /* dev->l2ad_writing will be lowered in the zio done callback */
4720        dev->l2ad_writing = B_TRUE;
4721        (void) zio_wait(pio);
4722        ASSERT(dev->l2ad_writing == B_FALSE);
4730        dev->l2ad_writing = B_FALSE;

4724        return (write_asize);
4725 }
```

_____***unchanged_portion_omitted_***