```
**********************************************************
  150510 Wed Oct  8 22:17:06 2014
new/usr/src/uts/common/fs/zfs/arc.c
5222 l2arc compression buffers "leak"
Author:         Andriy Gapon <avg@FreeBSD.org>
Reviewed by:    Saso Kiselkov <skiselkov.ml@gmail.com>
Reviewed by:    Xin Li <delphij@FreeBSD.org>
**********************************************************
______unchanged_portion_omitted_

 244 /* The 6 states: */
 245 static arc_state_t ARC_anon;
 246 static arc_state_t ARC_mru;
 247 static arc_state_t ARC_mru_ghost;
 248 static arc_state_t ARC_mfu;
 249 static arc_state_t ARC_mfu_ghost;
 250 static arc_state_t ARC_l2c_only;

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



```
 300         kstat_named_t arcstat_l2_feeds;
 301         kstat_named_t arcstat_l2_rw_clash;
 302         kstat_named_t arcstat_l2_read_bytes;
 303         kstat_named_t arcstat_l2_write_bytes;
 304         kstat_named_t arcstat_l2_writes_sent;
 305         kstat_named_t arcstat_l2_writes_done;
 306         kstat_named_t arcstat_l2_writes_error;
 307         kstat_named_t arcstat_l2_writes_hdr_miss;
 308         kstat_named_t arcstat_l2_evict_lock_retry;
 309         kstat_named_t arcstat_l2_evict_reading;
 310         kstat_named_t arcstat_l2_free_on_write;
 311         kstat_named_t arcstat_l2_cdata_free_on_write;
 312 #endif /* ! codereview */
 313         kstat_named_t arcstat_l2_abort_lowmem;
 314         kstat_named_t arcstat_l2_cksum_bad;
 315         kstat_named_t arcstat_l2_io_error;
 316         kstat_named_t arcstat_l2_size;
 317         kstat_named_t arcstat_l2_asize;
 318         kstat_named_t arcstat_l2_hdr_size;
 319         kstat_named_t arcstat_l2_compress_successes;
 320         kstat_named_t arcstat_l2_compress_zeros;
 321         kstat_named_t arcstat_l2_compress_failures;
 322         kstat_named_t arcstat_memory_throttle_count;
 323         kstat_named_t arcstat_duplicate_buffers;
 324         kstat_named_t arcstat_duplicate_buffers_size;
 325         kstat_named_t arcstat_duplicate_reads;
 326         kstat_named_t arcstat_meta_used;
 327         kstat_named_t arcstat_meta_limit;
 328         kstat_named_t arcstat_meta_max;
 329 } arc_stats_t;

 331 static arc_stats_t arc_stats = {
 332         { "hits",                       KSTAT_DATA_UINT64 },
 333         { "misses",                     KSTAT_DATA_UINT64 },
 334         { "demand_data_hits",           KSTAT_DATA_UINT64 },
 335         { "demand_data_misses",         KSTAT_DATA_UINT64 },
 336         { "demand_metadata_hits",       KSTAT_DATA_UINT64 },
 337         { "demand_metadata_misses",     KSTAT_DATA_UINT64 },
 338         { "prefetch_data_hits",         KSTAT_DATA_UINT64 },
 339         { "prefetch_data_misses",       KSTAT_DATA_UINT64 },
 340         { "prefetch_metadata_hits",     KSTAT_DATA_UINT64 },
 341         { "prefetch_metadata_misses",   KSTAT_DATA_UINT64 },
 342         { "mru_hits",                   KSTAT_DATA_UINT64 },
 343         { "mru_ghost_hits",             KSTAT_DATA_UINT64 },
 344         { "mfu_hits",                   KSTAT_DATA_UINT64 },
 345         { "mfu_ghost_hits",             KSTAT_DATA_UINT64 },
 346         { "deleted",                    KSTAT_DATA_UINT64 },
 347         { "recycle_miss",               KSTAT_DATA_UINT64 },
 348         { "mutex_miss",                 KSTAT_DATA_UINT64 },
 349         { "evict_skip",                 KSTAT_DATA_UINT64 },
 350         { "evict_l2_cached",            KSTAT_DATA_UINT64 },
 351         { "evict_l2_eligible",          KSTAT_DATA_UINT64 },
 352         { "evict_l2_ineligible",        KSTAT_DATA_UINT64 },
 353         { "hash_elements",              KSTAT_DATA_UINT64 },
 354         { "hash_elements_max",          KSTAT_DATA_UINT64 },
 355         { "hash_collisions",            KSTAT_DATA_UINT64 },
 356         { "hash_chains",                KSTAT_DATA_UINT64 },
 357         { "hash_chain_max",             KSTAT_DATA_UINT64 },
 358         { "p",                          KSTAT_DATA_UINT64 },
 359         { "c",                          KSTAT_DATA_UINT64 },
 360         { "c_min",                      KSTAT_DATA_UINT64 },
 361         { "c_max",                      KSTAT_DATA_UINT64 },
 362         { "size",                       KSTAT_DATA_UINT64 },
 363         { "hdr_size",                   KSTAT_DATA_UINT64 },
 364         { "data_size",                  KSTAT_DATA_UINT64 },
 365         { "other_size",                 KSTAT_DATA_UINT64 },
```

```
366         { "l2_hits",                    KSTAT_DATA_UINT64 },
367         { "l2_misses",                  KSTAT_DATA_UINT64 },
368         { "l2_feeds",                   KSTAT_DATA_UINT64 },
369         { "l2_rw_clash",                KSTAT_DATA_UINT64 },
370         { "l2_read_bytes",              KSTAT_DATA_UINT64 },
371         { "l2_write_bytes",             KSTAT_DATA_UINT64 },
372         { "l2_writes_sent",             KSTAT_DATA_UINT64 },
373         { "l2_writes_done",             KSTAT_DATA_UINT64 },
374         { "l2_writes_error",            KSTAT_DATA_UINT64 },
375         { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
376         { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
377         { "l2_evict_reading",           KSTAT_DATA_UINT64 },
378         { "l2_free_on_write",           KSTAT_DATA_UINT64 },
379         { "l2_cdata_free_on_write",     KSTAT_DATA_UINT64 },
380 #endif /* ! codereview */
381         { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
382         { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
383         { "l2_io_error",                KSTAT_DATA_UINT64 },
384         { "l2_size",                    KSTAT_DATA_UINT64 },
385         { "l2_asize",                   KSTAT_DATA_UINT64 },
386         { "l2_hdr_size",                KSTAT_DATA_UINT64 },
387         { "l2_compress_successes",      KSTAT_DATA_UINT64 },
388         { "l2_compress_zeros",          KSTAT_DATA_UINT64 },
389         { "l2_compress_failures",       KSTAT_DATA_UINT64 },
390         { "memory_throttle_count",      KSTAT_DATA_UINT64 },
391         { "duplicate_buffers",          KSTAT_DATA_UINT64 },
392         { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
393         { "duplicate_reads",            KSTAT_DATA_UINT64 },
394         { "arc_meta_used",              KSTAT_DATA_UINT64 },
395         { "arc_meta_limit",             KSTAT_DATA_UINT64 },
396         { "arc_meta_max",               KSTAT_DATA_UINT64 }
397 };

399 #define ARCSTAT(stat)   (arc_stats.stat.value.ui64)

401 #define ARCSTAT_INCR(stat, val) \
402         atomic_add_64(&arc_stats.stat.value.ui64, (val))

404 #define ARCSTAT_BUMP(stat)      ARCSTAT_INCR(stat, 1)
405 #define ARCSTAT_BUMPDOWN(stat)  ARCSTAT_INCR(stat, -1)

407 #define ARCSTAT_MAX(stat, val) {                                        \
408         uint64_t m;                                                    \
409         while ((val) > (m = arc_stats.stat.value.ui64) &&              \
410             (m != atomic_cas_64(&arc_stats.stat.value.ui64, m, (val)))) \
411                 continue;                                              \
412 }

414 #define ARCSTAT_MAXSTAT(stat) \
415         ARCSTAT_MAX(stat##_max, arc_stats.stat.value.ui64)

417 /*
418  * We define a macro to allow ARC hits/misses to be easily broken down by
419  * two separate conditions, giving a total of four different subtypes for
420  * each of hits and misses (so eight statistics total).
421  */
422 #define ARCSTAT_CONDSTAT(cond1, stat1, notstat1, cond2, stat2, notstat2, stat) \
423         if (cond1) {                                                   \
424                 if (cond2) {                                           \
425                         ARCSTAT_BUMP(arcstat_##stat1##_##stat2##_##stat); \
426                 } else {                                               \
427                         ARCSTAT_BUMP(arcstat_##stat1##_##notstat2##_##stat); \
428                 }                                                      \
429         } else {                                                       \
430                 if (cond2) {                                           \
431                         ARCSTAT_BUMP(arcstat_##notstat1##_##stat2##_##stat); \
```



```
432                 } else {                                               \
433                         ARCSTAT_BUMP(arcstat_##notstat1##_##notstat2##_##stat);\
434                 }                                                      \
435         }

437 kstat_t                 *arc_ksp;
438 static arc_state_t      *arc_anon;
439 static arc_state_t      *arc_mru;
440 static arc_state_t      *arc_mru_ghost;
441 static arc_state_t      *arc_mfu;
442 static arc_state_t      *arc_mfu_ghost;
443 static arc_state_t      *arc_l2c_only;

445 /*
446  * There are several ARC variables that are critical to export as kstats --
447  * but we don't want to have to grovel around in the kstat whenever we wish to
448  * manipulate them.  For these variables, we therefore define them to be in
449  * terms of the statistic variable.  This assures that we are not introducing
450  * the possibility of inconsistency by having shadow copies of the variables,
451  * while still allowing the code to be readable.
452  */
453 #define arc_size        ARCSTAT(arcstat_size)   /* actual total arc size */
454 #define arc_p           ARCSTAT(arcstat_p)      /* target size of MRU */
455 #define arc_c           ARCSTAT(arcstat_c)      /* target size of cache */
456 #define arc_c_min       ARCSTAT(arcstat_c_min)  /* min target cache size */
457 #define arc_c_max       ARCSTAT(arcstat_c_max)  /* max target cache size */
458 #define arc_meta_limit  ARCSTAT(arcstat_meta_limit) /* max size for metadata */
459 #define arc_meta_used   ARCSTAT(arcstat_meta_used) /* size of metadata */
460 #define arc_meta_max    ARCSTAT(arcstat_meta_max) /* max size of metadata */

462 #define L2ARC_IS_VALID_COMPRESS(_c_) \
463         ((_c_) == ZIO_COMPRESS_LZ4 || (_c_) == ZIO_COMPRESS_EMPTY)

465 static int              arc_no_grow;    /* Don't try to grow cache size */
466 static uint64_t         arc_tempreserve;
467 static uint64_t         arc_loaned_bytes;

469 typedef struct l2arc_buf_hdr l2arc_buf_hdr_t;

471 typedef struct arc_callback arc_callback_t;

473 struct arc_callback {
474         void                    *acb_private;
475         arc_done_func_t         *acb_done;
476         arc_buf_t               *acb_buf;
477         zio_t                   *acb_zio_dummy;
478         arc_callback_t          *acb_next;
479 };

481 typedef struct arc_write_callback arc_write_callback_t;

483 struct arc_write_callback {
484         void            *awcb_private;
485         arc_done_func_t *awcb_ready;
486         arc_done_func_t *awcb_physdone;
487         arc_done_func_t *awcb_done;
488         arc_buf_t       *awcb_buf;
489 };

491 struct arc_buf_hdr {
492         /* protected by hash lock */
493         dva_t                   b_dva;
494         uint64_t                b_birth;
495         uint64_t                b_cksum0;

497         kmutex_t                b_freeze_lock;
```

```
498         zio_cksum_t             *b_freeze_cksum;
499         void                    *b_thawed;

501         arc_buf_hdr_t           *b_hash_next;
502         arc_buf_t               *b_buf;
503         uint32_t                b_flags;
504         uint32_t                b_datacnt;

506         arc_callback_t          *b_acb;
507         kcondvar_t              b_cv;

509         /* immutable */
510         arc_buf_contents_t      b_type;
511         uint64_t                b_size;
512         uint64_t                b_spa;

514         /* protected by arc state mutex */
515         arc_state_t             *b_state;
516         list_node_t             b_arc_node;

518         /* updated atomically */
519         clock_t                 b_arc_access;

521         /* self protecting */
522         refcount_t              b_refcnt;

524         l2arc_buf_hdr_t         *b_l2hdr;
525         list_node_t             b_l2node;
526 };

528 static arc_buf_t *arc_eviction_list;
529 static kmutex_t arc_eviction_mtx;
530 static arc_buf_hdr_t arc_eviction_hdr;
531 static void arc_get_data_buf(arc_buf_t *buf);
532 static void arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock);
533 static int arc_evict_needed(arc_buf_contents_t type);
534 static void arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes);
535 static void arc_buf_watch(arc_buf_t *buf);

537 static boolean_t l2arc_write_eligible(uint64_t spa_guid, arc_buf_hdr_t *ab);

539 #define GHOST_STATE(state)      \
540         ((state) == arc_mru_ghost || (state) == arc_mfu_ghost ||        \
541         (state) == arc_l2c_only)

543 /*
544  * Private ARC flags.  These flags are private ARC only flags that will show up
545  * in b_flags in the arc_hdr_buf_t.  Some flags are publicly declared, and can
546  * be passed in as arc_flags in things like arc_read.  However, these flags
547  * should never be passed and should only be set by ARC code.  When adding new
548  * public flags, make sure not to smash the private ones.
549  */

551 #define ARC_IN_HASH_TABLE       (1 << 9)        /* this buffer is hashed */
552 #define ARC_IO_IN_PROGRESS      (1 << 10)       /* I/O in progress for buf */
553 #define ARC_IO_ERROR            (1 << 11)       /* I/O failed for buf */
554 #define ARC_FREED_IN_READ       (1 << 12)       /* buf freed while in read */
555 #define ARC_BUF_AVAILABLE       (1 << 13)       /* block not in active use */
556 #define ARC_INDIRECT            (1 << 14)       /* this is an indirect block */
557 #define ARC_FREE_IN_PROGRESS    (1 << 15)       /* hdr about to be freed */
558 #define ARC_L2_WRITING          (1 << 16)       /* L2ARC write in progress */
559 #define ARC_L2_EVICTED          (1 << 17)       /* evicted during I/O */
560 #define ARC_L2_WRITE_HEAD       (1 << 18)       /* head of write list */

562 #define HDR_IN_HASH_TABLE(hdr)  ((hdr)->b_flags & ARC_IN_HASH_TABLE)
563 #define HDR_IO_IN_PROGRESS(hdr) ((hdr)->b_flags & ARC_IO_IN_PROGRESS)
```



```
564 #define HDR_IO_ERROR(hdr)       ((hdr)->b_flags & ARC_IO_ERROR)
565 #define HDR_PREFETCH(hdr)       ((hdr)->b_flags & ARC_PREFETCH)
566 #define HDR_FREED_IN_READ(hdr)  ((hdr)->b_flags & ARC_FREED_IN_READ)
567 #define HDR_BUF_AVAILABLE(hdr)  ((hdr)->b_flags & ARC_BUF_AVAILABLE)
568 #define HDR_FREE_IN_PROGRESS(hdr)       ((hdr)->b_flags & ARC_FREE_IN_PROGRESS)
569 #define HDR_L2CACHE(hdr)        ((hdr)->b_flags & ARC_L2CACHE)
570 #define HDR_L2_READING(hdr)     ((hdr)->b_flags & ARC_IO_IN_PROGRESS && \
571                                     (hdr)->b_l2hdr != NULL)
572 #define HDR_L2_WRITING(hdr)     ((hdr)->b_flags & ARC_L2_WRITING)
573 #define HDR_L2_EVICTED(hdr)     ((hdr)->b_flags & ARC_L2_EVICTED)
574 #define HDR_L2_WRITE_HEAD(hdr)  ((hdr)->b_flags & ARC_L2_WRITE_HEAD)

576 /*
577  * Other sizes
578  */

580 #define HDR_SIZE ((int64_t)sizeof (arc_buf_hdr_t))
581 #define L2HDR_SIZE ((int64_t)sizeof (l2arc_buf_hdr_t))

583 /*
584  * Hash table routines
585  */

587 #define HT_LOCK_PAD     64

589 struct ht_lock {
590         kmutex_t        ht_lock;
591 #ifdef _KERNEL
592         unsigned char   pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
593 #endif
594 };

596 #define BUF_LOCKS 256
597 typedef struct buf_hash_table {
598         uint64_t ht_mask;
599         arc_buf_hdr_t **ht_table;
600         struct ht_lock ht_locks[BUF_LOCKS];
601 } buf_hash_table_t;

603 static buf_hash_table_t buf_hash_table;

605 #define BUF_HASH_INDEX(spa, dva, birth) \
606         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
607 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
608 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
609 #define HDR_LOCK(hdr) \
610         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))

612 uint64_t zfs_crc64_table[256];

614 /*
615  * Level 2 ARC
616  */

618 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
619 #define L2ARC_HEADROOM          2                       /* num of writes */
620 /*
621  * If we discover during ARC scan any buffers to be compressed, we boost
622  * our headroom for the next scanning cycle by this percentage multiple.
623  */
624 #define L2ARC_HEADROOM_BOOST    200
625 #define L2ARC_FEED_SECS         1               /* caching interval secs */
626 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

628 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
629 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)
```

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

/*
 * L2ARC Internals
 */
typedef struct l2arc_dev {
        vdev_t                  *l2ad_vdev;     /* vdev */
        spa_t                   *l2ad_spa;      /* spa */
        uint64_t                l2ad_hand;      /* next write location */
        uint64_t                l2ad_start;     /* first addr on device */
        uint64_t                l2ad_end;       /* last addr on device */
        uint64_t                l2ad_evict;     /* last addr eviction reached */
        boolean_t               l2ad_first;     /* first sweep through */
        boolean_t               l2ad_writing;   /* currently writing */
        list_t                  *l2ad_buflist;  /* buffer list */
        list_node_t             l2ad_node;      /* device list node */
} l2arc_dev_t;

static list_t L2ARC_dev_list;                   /* device list */
static list_t *l2arc_dev_list;                  /* device list pointer */
static kmutex_t l2arc_dev_mtx;                  /* device list mutex */
static l2arc_dev_t *l2arc_dev_last;             /* last device used */
static kmutex_t l2arc_buflist_mtx;              /* mutex for all buflists */
static list_t L2ARC_free_on_write;              /* free after write buf list */
static list_t *l2arc_free_on_write;             /* free after write list ptr */
static kmutex_t l2arc_free_on_write_mtx;        /* mutex for list */
static uint64_t l2arc_ndev;                     /* number of devices */

typedef struct l2arc_read_callback {
        arc_buf_t               *l2rcb_buf;             /* read buffer */
        spa_t                   *l2rcb_spa;             /* spa */
        blkptr_t                l2rcb_bp;               /* original blkptr */
        zbookmark_phys_t        l2rcb_zb;               /* original bookmark */
        int                     l2rcb_flags;            /* original flags */
        enum zio_compress       l2rcb_compress;         /* applied compress */
} l2arc_read_callback_t;

typedef struct l2arc_write_callback {
        l2arc_dev_t     *l2wcb_dev;             /* device info */
        arc_buf_hdr_t   *l2wcb_head;            /* head of write buflist */
} l2arc_write_callback_t;

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

typedef struct l2arc_data_free {
        /* protected by l2arc_free_on_write_mtx */
```



```
        void            *l2df_data;
        size_t          l2df_size;
        void            (*l2df_func)(void *, size_t);
        list_node_t     l2df_list_node;
} l2arc_data_free_t;

static kmutex_t l2arc_feed_thr_lock;
static kcondvar_t l2arc_feed_thr_cv;
static uint8_t l2arc_thread_exit;

static void l2arc_read_done(zio_t *zio);
static void l2arc_hdr_stat_add(void);
static void l2arc_hdr_stat_remove(void);

static boolean_t l2arc_compress_buf(l2arc_buf_hdr_t *l2hdr);
static void l2arc_decompress_zio(zio_t *zio, arc_buf_hdr_t *hdr,
    enum zio_compress c);
static void l2arc_release_cdata_buf(arc_buf_hdr_t *ab);

static uint64_t
buf_hash(uint64_t spa, const dva_t *dva, uint64_t birth)
{
        uint8_t *vdva = (uint8_t *)dva;
        uint64_t crc = -1ULL;
        int i;

        ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);

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

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

        return (crc);
}

#define BUF_EMPTY(buf)                                          \
        ((buf)->b_dva.dva_word[0] == 0 &&                       \
        (buf)->b_dva.dva_word[1] == 0 &&                        \
        (buf)->b_cksum0 == 0)

#define BUF_EQUAL(spa, dva, birth, buf)                         \
        ((buf)->b_dva.dva_word[0] == (dva)->dva_word[0]) &&     \
        ((buf)->b_dva.dva_word[1] == (dva)->dva_word[1]) &&     \
        ((buf)->b_birth == birth) && ((buf)->b_spa == spa)

static void
buf_discard_identity(arc_buf_hdr_t *hdr)
{
        hdr->b_dva.dva_word[0] = 0;
        hdr->b_dva.dva_word[1] = 0;
        hdr->b_birth = 0;
        hdr->b_cksum0 = 0;
}

static arc_buf_hdr_t *
buf_hash_find(uint64_t spa, const blkptr_t *bp, kmutex_t **lockp)
{
        const dva_t *dva = BP_IDENTITY(bp);
        uint64_t birth = BP_PHYSICAL_BIRTH(bp);
        uint64_t idx = BUF_HASH_INDEX(spa, dva, birth);
        kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
        arc_buf_hdr_t *buf;

        mutex_enter(hash_lock);
        for (buf = buf_hash_table.ht_table[idx]; buf != NULL;
```

```
762             buf = buf->b_hash_next) {
763                 if (BUF_EQUAL(spa, dva, birth, buf)) {
764                         *lockp = hash_lock;
765                         return (buf);
766                 }
767         }
768         mutex_exit(hash_lock);
769         *lockp = NULL;
770         return (NULL);
771 }
772
773 /*
774  * Insert an entry into the hash table.  If there is already an element
775  * equal to elem in the hash table, then the already existing element
776  * will be returned and the new element will not be inserted.
777  * Otherwise returns NULL.
778  */
779 static arc_buf_hdr_t *
780 buf_hash_insert(arc_buf_hdr_t *buf, kmutex_t **lockp)
781 {
782         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
783         kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
784         arc_buf_hdr_t *fbuf;
785         uint32_t i;
786
787         ASSERT(!DVA_IS_EMPTY(&buf->b_dva));
788         ASSERT(buf->b_birth != 0);
789         ASSERT(!HDR_IN_HASH_TABLE(buf));
790         *lockp = hash_lock;
791         mutex_enter(hash_lock);
792         for (fbuf = buf_hash_table.ht_table[idx], i = 0; fbuf != NULL;
793             fbuf = fbuf->b_hash_next, i++) {
794                 if (BUF_EQUAL(buf->b_spa, &buf->b_dva, buf->b_birth, fbuf))
795                         return (fbuf);
796         }
797
798         buf->b_hash_next = buf_hash_table.ht_table[idx];
799         buf_hash_table.ht_table[idx] = buf;
800         buf->b_flags |= ARC_IN_HASH_TABLE;
801
802         /* collect some hash table performance data */
803         if (i > 0) {
804                 ARCSTAT_BUMP(arcstat_hash_collisions);
805                 if (i == 1)
806                         ARCSTAT_BUMP(arcstat_hash_chains);
807
808                 ARCSTAT_MAX(arcstat_hash_chain_max, i);
809         }
810
811         ARCSTAT_BUMP(arcstat_hash_elements);
812         ARCSTAT_MAXSTAT(arcstat_hash_elements);
813
814         return (NULL);
815 }
816
817 static void
818 buf_hash_remove(arc_buf_hdr_t *buf)
819 {
820         arc_buf_hdr_t *fbuf, **bufp;
821         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
822
823         ASSERT(MUTEX_HELD(BUF_HASH_LOCK(idx)));
824         ASSERT(HDR_IN_HASH_TABLE(buf));
825
826         bufp = &buf_hash_table.ht_table[idx];
827         while ((fbuf = *bufp) != buf) {
```



```
828                 ASSERT(fbuf != NULL);
829                 bufp = &fbuf->b_hash_next;
830         }
831         *bufp = buf->b_hash_next;
832         buf->b_hash_next = NULL;
833         buf->b_flags &= ~ARC_IN_HASH_TABLE;
834
835         /* collect some hash table performance data */
836         ARCSTAT_BUMPDOWN(arcstat_hash_elements);
837
838         if (buf_hash_table.ht_table[idx] &&
839             buf_hash_table.ht_table[idx]->b_hash_next == NULL)
840                 ARCSTAT_BUMPDOWN(arcstat_hash_chains);
841 }
842
843 /*
844  * Global data structures and functions for the buf kmem cache.
845  */
846 static kmem_cache_t *hdr_cache;
847 static kmem_cache_t *buf_cache;
848
849 static void
850 buf_fini(void)
851 {
852         int i;
853
854         kmem_free(buf_hash_table.ht_table,
855             (buf_hash_table.ht_mask + 1) * sizeof (void *));
856         for (i = 0; i < BUF_LOCKS; i++)
857                 mutex_destroy(&buf_hash_table.ht_locks[i].ht_lock);
858         kmem_cache_destroy(hdr_cache);
859         kmem_cache_destroy(buf_cache);
860 }
861
862 /*
863  * Constructor callback - called when the cache is empty
864  * and a new buf is requested.
865  */
866 /* ARGSUSED */
867 static int
868 hdr_cons(void *vbuf, void *unused, int kmflag)
869 {
870         arc_buf_hdr_t *buf = vbuf;
871
872         bzero(buf, sizeof (arc_buf_hdr_t));
873         refcount_create(&buf->b_refcnt);
874         cv_init(&buf->b_cv, NULL, CV_DEFAULT, NULL);
875         mutex_init(&buf->b_freeze_lock, NULL, MUTEX_DEFAULT, NULL);
876         arc_space_consume(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
877
878         return (0);
879 }
880
881 /* ARGSUSED */
882 static int
883 buf_cons(void *vbuf, void *unused, int kmflag)
884 {
885         arc_buf_t *buf = vbuf;
886
887         bzero(buf, sizeof (arc_buf_t));
888         mutex_init(&buf->b_evict_lock, NULL, MUTEX_DEFAULT, NULL);
889         arc_space_consume(sizeof (arc_buf_t), ARC_SPACE_HDRS);
890
891         return (0);
892 }
```

```
894 /*
895  * Destructor callback - called when a cached buf is
896  * no longer required.
897  */
898 /* ARGSUSED */
899 static void
900 hdr_dest(void *vbuf, void *unused)
901 {
902         arc_buf_hdr_t *buf = vbuf;

904         ASSERT(BUF_EMPTY(buf));
905         refcount_destroy(&buf->b_refcnt);
906         cv_destroy(&buf->b_cv);
907         mutex_destroy(&buf->b_freeze_lock);
908         arc_space_return(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
909 }

911 /* ARGSUSED */
912 static void
913 buf_dest(void *vbuf, void *unused)
914 {
915         arc_buf_t *buf = vbuf;

917         mutex_destroy(&buf->b_evict_lock);
918         arc_space_return(sizeof (arc_buf_t), ARC_SPACE_HDRS);
919 }

921 /*
922  * Reclaim callback -- invoked when memory is low.
923  */
924 /* ARGSUSED */
925 static void
926 hdr_recl(void *unused)
927 {
928         dprintf("hdr_recl called\n");
929         /*
930          * umem calls the reclaim func when we destroy the buf cache,
931          * which is after we do arc_fini().
932          */
933         if (!arc_dead)
934                 cv_signal(&arc_reclaim_thr_cv);
935 }

937 static void
938 buf_init(void)
939 {
940         uint64_t *ct;
941         uint64_t hsize = 1ULL << 12;
942         int i, j;

944         /*
945          * The hash table is big enough to fill all of physical memory
946          * with an average block size of zfs_arc_average_blocksize (default 8K).
947          * By default, the table will take up
948          * totalmem * sizeof(void*) / 8K (1MB per GB with 8-byte pointers).
949          */
950         while (hsize * zfs_arc_average_blocksize < physmem * PAGESIZE)
951                 hsize <<= 1;
952 retry:
953         buf_hash_table.ht_mask = hsize - 1;
954         buf_hash_table.ht_table =
955             kmem_zalloc(hsize * sizeof (void*), KM_NOSLEEP);
956         if (buf_hash_table.ht_table == NULL) {
957                 ASSERT(hsize > (1ULL << 8));
958                 hsize >>= 1;
959                 goto retry;
```



```
960         }

962         hdr_cache = kmem_cache_create("arc_buf_hdr_t", sizeof (arc_buf_hdr_t),
963             0, hdr_cons, hdr_dest, hdr_recl, NULL, NULL, 0);
964         buf_cache = kmem_cache_create("arc_buf_t", sizeof (arc_buf_t),
965             0, buf_cons, buf_dest, NULL, NULL, NULL, 0);

967         for (i = 0; i < 256; i++)
968                 for (ct = zfs_crc64_table + i, *ct = i, j = 8; j > 0; j--)
969                         *ct = (*ct >> 1) ^ (-(*ct & 1) & ZFS_CRC64_POLY);

971         for (i = 0; i < BUF_LOCKS; i++) {
972                 mutex_init(&buf_hash_table.ht_locks[i].ht_lock,
973                     NULL, MUTEX_DEFAULT, NULL);
974         }
975 }

977 #define ARC_MINTIME     (hz>>4) /* 62 ms */

979 static void
980 arc_cksum_verify(arc_buf_t *buf)
981 {
982         zio_cksum_t zc;

984         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
985                 return;

987         mutex_enter(&buf->b_hdr->b_freeze_lock);
988         if (buf->b_hdr->b_freeze_cksum == NULL ||
989             (buf->b_hdr->b_flags & ARC_IO_ERROR)) {
990                 mutex_exit(&buf->b_hdr->b_freeze_lock);
991                 return;
992         }
993         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
994         if (!ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc))
995                 panic("buffer modified while frozen!");
996         mutex_exit(&buf->b_hdr->b_freeze_lock);
997 }

999 static int
1000 arc_cksum_equal(arc_buf_t *buf)
1001 {
1002         zio_cksum_t zc;
1003         int equal;

1005         mutex_enter(&buf->b_hdr->b_freeze_lock);
1006         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
1007         equal = ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc);
1008         mutex_exit(&buf->b_hdr->b_freeze_lock);

1010         return (equal);
1011 }

1013 static void
1014 arc_cksum_compute(arc_buf_t *buf, boolean_t force)
1015 {
1016         if (!force && !(zfs_flags & ZFS_DEBUG_MODIFY))
1017                 return;

1019         mutex_enter(&buf->b_hdr->b_freeze_lock);
1020         if (buf->b_hdr->b_freeze_cksum != NULL) {
1021                 mutex_exit(&buf->b_hdr->b_freeze_lock);
1022                 return;
1023         }
1024         buf->b_hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
1025         fletcher_2_native(buf->b_data, buf->b_hdr->b_size,
```

```
1026             buf->b_hdr->b_freeze_cksum);
1027         mutex_exit(&buf->b_hdr->b_freeze_lock);
1028         arc_buf_watch(buf);
1029 }

1031 #ifndef _KERNEL
1032 typedef struct procctl {
1033         long cmd;
1034         prwatch_t prwatch;
1035 } procctl_t;
1036 #endif

1038 /* ARGSUSED */
1039 static void
1040 arc_buf_unwatch(arc_buf_t *buf)
1041 {
1042 #ifndef _KERNEL
1043         if (arc_watch) {
1044                 int result;
1045                 procctl_t ctl;
1046                 ctl.cmd = PCWATCH;
1047                 ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1048                 ctl.prwatch.pr_size = 0;
1049                 ctl.prwatch.pr_wflags = 0;
1050                 result = write(arc_procfd, &ctl, sizeof (ctl));
1051                 ASSERT3U(result, ==, sizeof (ctl));
1052         }
1053 #endif
1054 }

1056 /* ARGSUSED */
1057 static void
1058 arc_buf_watch(arc_buf_t *buf)
1059 {
1060 #ifndef _KERNEL
1061         if (arc_watch) {
1062                 int result;
1063                 procctl_t ctl;
1064                 ctl.cmd = PCWATCH;
1065                 ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1066                 ctl.prwatch.pr_size = buf->b_hdr->b_size;
1067                 ctl.prwatch.pr_wflags = WA_WRITE;
1068                 result = write(arc_procfd, &ctl, sizeof (ctl));
1069                 ASSERT3U(result, ==, sizeof (ctl));
1070         }
1071 #endif
1072 }

1074 void
1075 arc_buf_thaw(arc_buf_t *buf)
1076 {
1077         if (zfs_flags & ZFS_DEBUG_MODIFY) {
1078                 if (buf->b_hdr->b_state != arc_anon)
1079                         panic("modifying non-anon buffer!");
1080                 if (buf->b_hdr->b_flags & ARC_IO_IN_PROGRESS)
1081                         panic("modifying buffer while i/o in progress!");
1082                 arc_cksum_verify(buf);
1083         }

1085         mutex_enter(&buf->b_hdr->b_freeze_lock);
1086         if (buf->b_hdr->b_freeze_cksum != NULL) {
1087                 kmem_free(buf->b_hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1088                 buf->b_hdr->b_freeze_cksum = NULL;
1089         }

1091         if (zfs_flags & ZFS_DEBUG_MODIFY) {
```



```
1092                 if (buf->b_hdr->b_thawed)
1093                         kmem_free(buf->b_hdr->b_thawed, 1);
1094                 buf->b_hdr->b_thawed = kmem_alloc(1, KM_SLEEP);
1095         }

1097         mutex_exit(&buf->b_hdr->b_freeze_lock);

1099         arc_buf_unwatch(buf);
1100 }

1102 void
1103 arc_buf_freeze(arc_buf_t *buf)
1104 {
1105         kmutex_t *hash_lock;

1107         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
1108                 return;

1110         hash_lock = HDR_LOCK(buf->b_hdr);
1111         mutex_enter(hash_lock);

1113         ASSERT(buf->b_hdr->b_freeze_cksum != NULL ||
1114             buf->b_hdr->b_state == arc_anon);
1115         arc_cksum_compute(buf, B_FALSE);
1116         mutex_exit(hash_lock);

1118 }

1120 static void
1121 add_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
1122 {
1123         ASSERT(MUTEX_HELD(hash_lock));

1125         if ((refcount_add(&ab->b_refcnt, tag) == 1) &&
1126             (ab->b_state != arc_anon)) {
1127                 uint64_t delta = ab->b_size * ab->b_datacnt;
1128                 list_t *list = &ab->b_state->arcs_list[ab->b_type];
1129                 uint64_t *size = &ab->b_state->arcs_lsize[ab->b_type];

1131                 ASSERT(!MUTEX_HELD(&ab->b_state->arcs_mtx));
1132                 mutex_enter(&ab->b_state->arcs_mtx);
1133                 ASSERT(list_link_active(&ab->b_arc_node));
1134                 list_remove(list, ab);
1135                 if (GHOST_STATE(ab->b_state)) {
1136                         ASSERT0(ab->b_datacnt);
1137                         ASSERT3P(ab->b_buf, ==, NULL);
1138                         delta = ab->b_size;
1139                 }
1140                 ASSERT(delta > 0);
1141                 ASSERT3U(*size, >=, delta);
1142                 atomic_add_64(size, -delta);
1143                 mutex_exit(&ab->b_state->arcs_mtx);
1144                 /* remove the prefetch flag if we get a reference */
1145                 if (ab->b_flags & ARC_PREFETCH)
1146                         ab->b_flags &= ~ARC_PREFETCH;
1147         }
1148 }

1150 static int
1151 remove_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
1152 {
1153         int cnt;
1154         arc_state_t *state = ab->b_state;

1156         ASSERT(state == arc_anon || MUTEX_HELD(hash_lock));
1157         ASSERT(!GHOST_STATE(state));
```

```
1159         if (((cnt = refcount_remove(&ab->b_refcnt, tag)) == 0) &&
1160             (state != arc_anon)) {
1161                 uint64_t *size = &state->arcs_lsize[ab->b_type];

1163                 ASSERT(!MUTEX_HELD(&state->arcs_mtx));
1164                 mutex_enter(&state->arcs_mtx);
1165                 ASSERT(!list_link_active(&ab->b_arc_node));
1166                 list_insert_head(&state->arcs_list[ab->b_type], ab);
1167                 ASSERT(ab->b_datacnt > 0);
1168                 atomic_add_64(size, ab->b_size * ab->b_datacnt);
1169                 mutex_exit(&state->arcs_mtx);
1170         }
1171         return (cnt);
1172 }

1174 /*
1175  * Move the supplied buffer to the indicated state.  The mutex
1176  * for the buffer must be held by the caller.
1177  */
1178 static void
1179 arc_change_state(arc_state_t *new_state, arc_buf_hdr_t *ab, kmutex_t *hash_lock)
1180 {
1181         arc_state_t *old_state = ab->b_state;
1182         int64_t refcnt = refcount_count(&ab->b_refcnt);
1183         uint64_t from_delta, to_delta;

1185         ASSERT(MUTEX_HELD(hash_lock));
1186         ASSERT3P(new_state, !=, old_state);
1187         ASSERT(refcnt == 0 || ab->b_datacnt > 0);
1188         ASSERT(ab->b_datacnt == 0 || !GHOST_STATE(new_state));
1189         ASSERT(ab->b_datacnt <= 1 || old_state != arc_anon);

1191         from_delta = to_delta = ab->b_datacnt * ab->b_size;

1193         /*
1194          * If this buffer is evictable, transfer it from the
1195          * old state list to the new state list.
1196          */
1197         if (refcnt == 0) {
1198                 if (old_state != arc_anon) {
1199                         int use_mutex = !MUTEX_HELD(&old_state->arcs_mtx);
1200                         uint64_t *size = &old_state->arcs_lsize[ab->b_type];

1202                         if (use_mutex)
1203                                 mutex_enter(&old_state->arcs_mtx);

1205                         ASSERT(list_link_active(&ab->b_arc_node));
1206                         list_remove(&old_state->arcs_list[ab->b_type], ab);

1208                         /*
1209                          * If prefetching out of the ghost cache,
1210                          * we will have a non-zero datacnt.
1211                          */
1212                         if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
1213                                 /* ghost elements have a ghost size */
1214                                 ASSERT(ab->b_buf == NULL);
1215                                 from_delta = ab->b_size;
1216                         }
1217                         ASSERT3U(*size, >=, from_delta);
1218                         atomic_add_64(size, -from_delta);

1220                         if (use_mutex)
1221                                 mutex_exit(&old_state->arcs_mtx);
1222                 }
1223                 if (new_state != arc_anon) {
```



```
1224                         int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
1225                         uint64_t *size = &new_state->arcs_lsize[ab->b_type];

1227                         if (use_mutex)
1228                                 mutex_enter(&new_state->arcs_mtx);

1230                         list_insert_head(&new_state->arcs_list[ab->b_type], ab);

1232                         /* ghost elements have a ghost size */
1233                         if (GHOST_STATE(new_state)) {
1234                                 ASSERT(ab->b_datacnt == 0);
1235                                 ASSERT(ab->b_buf == NULL);
1236                                 to_delta = ab->b_size;
1237                         }
1238                         atomic_add_64(size, to_delta);

1240                         if (use_mutex)
1241                                 mutex_exit(&new_state->arcs_mtx);
1242                 }
1243         }

1245         ASSERT(!BUF_EMPTY(ab));
1246         if (new_state == arc_anon && HDR_IN_HASH_TABLE(ab))
1247                 buf_hash_remove(ab);

1249         /* adjust state sizes */
1250         if (to_delta)
1251                 atomic_add_64(&new_state->arcs_size, to_delta);
1252         if (from_delta) {
1253                 ASSERT3U(old_state->arcs_size, >=, from_delta);
1254                 atomic_add_64(&old_state->arcs_size, -from_delta);
1255         }
1256         ab->b_state = new_state;

1258         /* adjust l2arc hdr stats */
1259         if (new_state == arc_l2c_only)
1260                 l2arc_hdr_stat_add();
1261         else if (old_state == arc_l2c_only)
1262                 l2arc_hdr_stat_remove();
1263 }

1265 void
1266 arc_space_consume(uint64_t space, arc_space_type_t type)
1267 {
1268         ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);

1270         switch (type) {
1271         case ARC_SPACE_DATA:
1272                 ARCSTAT_INCR(arcstat_data_size, space);
1273                 break;
1274         case ARC_SPACE_OTHER:
1275                 ARCSTAT_INCR(arcstat_other_size, space);
1276                 break;
1277         case ARC_SPACE_HDRS:
1278                 ARCSTAT_INCR(arcstat_hdr_size, space);
1279                 break;
1280         case ARC_SPACE_L2HDRS:
1281                 ARCSTAT_INCR(arcstat_l2_hdr_size, space);
1282                 break;
1283         }

1285         ARCSTAT_INCR(arcstat_meta_used, space);
1286         atomic_add_64(&arc_size, space);
1287 }

1289 void
```

```
1290 arc_space_return(uint64_t space, arc_space_type_t type)
1291 {
1292         ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);

1294         switch (type) {
1295         case ARC_SPACE_DATA:
1296                 ARCSTAT_INCR(arcstat_data_size, -space);
1297                 break;
1298         case ARC_SPACE_OTHER:
1299                 ARCSTAT_INCR(arcstat_other_size, -space);
1300                 break;
1301         case ARC_SPACE_HDRS:
1302                 ARCSTAT_INCR(arcstat_hdr_size, -space);
1303                 break;
1304         case ARC_SPACE_L2HDRS:
1305                 ARCSTAT_INCR(arcstat_l2_hdr_size, -space);
1306                 break;
1307         }

1309         ASSERT(arc_meta_used >= space);
1310         if (arc_meta_max < arc_meta_used)
1311                 arc_meta_max = arc_meta_used;
1312         ARCSTAT_INCR(arcstat_meta_used, -space);
1313         ASSERT(arc_size >= space);
1314         atomic_add_64(&arc_size, -space);
1315 }

1317 void *
1318 arc_data_buf_alloc(uint64_t size)
1319 {
1320         if (arc_evict_needed(ARC_BUFC_DATA))
1321                 cv_signal(&arc_reclaim_thr_cv);
1322         atomic_add_64(&arc_size, size);
1323         return (zio_data_buf_alloc(size));
1324 }

1326 void
1327 arc_data_buf_free(void *buf, uint64_t size)
1328 {
1329         zio_data_buf_free(buf, size);
1330         ASSERT(arc_size >= size);
1331         atomic_add_64(&arc_size, -size);
1332 }

1334 arc_buf_t *
1335 arc_buf_alloc(spa_t *spa, int size, void *tag, arc_buf_contents_t type)
1336 {
1337         arc_buf_hdr_t *hdr;
1338         arc_buf_t *buf;

1340         ASSERT3U(size, >, 0);
1341         hdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
1342         ASSERT(BUF_EMPTY(hdr));
1343         hdr->b_size = size;
1344         hdr->b_type = type;
1345         hdr->b_spa = spa_load_guid(spa);
1346         hdr->b_state = arc_anon;
1347         hdr->b_arc_access = 0;
1348         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1349         buf->b_hdr = hdr;
1350         buf->b_data = NULL;
1351         buf->b_efunc = NULL;
1352         buf->b_private = NULL;
1353         buf->b_next = NULL;
1354         hdr->b_buf = buf;
1355         arc_get_data_buf(buf);
```



```
1356         hdr->b_datacnt = 1;
1357         hdr->b_flags = 0;
1358         ASSERT(refcount_is_zero(&hdr->b_refcnt));
1359         (void) refcount_add(&hdr->b_refcnt, tag);

1361         return (buf);
1362 }

1364 static char *arc_onloan_tag = "onloan";

1366 /*
1367  * Loan out an anonymous arc buffer. Loaned buffers are not counted as in
1368  * flight data by arc_tempreserve_space() until they are "returned". Loaned
1369  * buffers must be returned to the arc before they can be used by the DMU or
1370  * freed.
1371  */
1372 arc_buf_t *
1373 arc_loan_buf(spa_t *spa, int size)
1374 {
1375         arc_buf_t *buf;

1377         buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);

1379         atomic_add_64(&arc_loaned_bytes, size);
1380         return (buf);
1381 }

1383 /*
1384  * Return a loaned arc buffer to the arc.
1385  */
1386 void
1387 arc_return_buf(arc_buf_t *buf, void *tag)
1388 {
1389         arc_buf_hdr_t *hdr = buf->b_hdr;

1391         ASSERT(buf->b_data != NULL);
1392         (void) refcount_add(&hdr->b_refcnt, tag);
1393         (void) refcount_remove(&hdr->b_refcnt, arc_onloan_tag);

1395         atomic_add_64(&arc_loaned_bytes, -hdr->b_size);
1396 }

1398 /* Detach an arc_buf from a dbuf (tag) */
1399 void
1400 arc_loan_inuse_buf(arc_buf_t *buf, void *tag)
1401 {
1402         arc_buf_hdr_t *hdr;

1404         ASSERT(buf->b_data != NULL);
1405         hdr = buf->b_hdr;
1406         (void) refcount_add(&hdr->b_refcnt, arc_onloan_tag);
1407         (void) refcount_remove(&hdr->b_refcnt, tag);
1408         buf->b_efunc = NULL;
1409         buf->b_private = NULL;

1411         atomic_add_64(&arc_loaned_bytes, hdr->b_size);
1412 }

1414 static arc_buf_t *
1415 arc_buf_clone(arc_buf_t *from)
1416 {
1417         arc_buf_t *buf;
1418         arc_buf_hdr_t *hdr = from->b_hdr;
1419         uint64_t size = hdr->b_size;

1421         ASSERT(hdr->b_state != arc_anon);
```

```
1423         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1424         buf->b_hdr = hdr;
1425         buf->b_data = NULL;
1426         buf->b_efunc = NULL;
1427         buf->b_private = NULL;
1428         buf->b_next = hdr->b_buf;
1429         hdr->b_buf = buf;
1430         arc_get_data_buf(buf);
1431         bcopy(from->b_data, buf->b_data, size);

1433         /*
1434          * This buffer already exists in the arc so create a duplicate
1435          * copy for the caller.  If the buffer is associated with user data
1436          * then track the size and number of duplicates.  These stats will be
1437          * updated as duplicate buffers are created and destroyed.
1438          */
1439         if (hdr->b_type == ARC_BUFC_DATA) {
1440                 ARCSTAT_BUMP(arcstat_duplicate_buffers);
1441                 ARCSTAT_INCR(arcstat_duplicate_buffers_size, size);
1442         }
1443         hdr->b_datacnt += 1;
1444         return (buf);
1445 }

1447 void
1448 arc_buf_add_ref(arc_buf_t *buf, void* tag)
1449 {
1450         arc_buf_hdr_t *hdr;
1451         kmutex_t *hash_lock;

1453         /*
1454          * Check to see if this buffer is evicted.  Callers
1455          * must verify b_data != NULL to know if the add_ref
1456          * was successful.
1457          */
1458         mutex_enter(&buf->b_evict_lock);
1459         if (buf->b_data == NULL) {
1460                 mutex_exit(&buf->b_evict_lock);
1461                 return;
1462         }
1463         hash_lock = HDR_LOCK(buf->b_hdr);
1464         mutex_enter(hash_lock);
1465         hdr = buf->b_hdr;
1466         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
1467         mutex_exit(&buf->b_evict_lock);

1469         ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);
1470         add_reference(hdr, hash_lock, tag);
1471         DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
1472         arc_access(hdr, hash_lock);
1473         mutex_exit(hash_lock);
1474         ARCSTAT_BUMP(arcstat_hits);
1475         ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
1476             demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
1477             data, metadata, hits);
1478 }

1480 static void
1481 arc_buf_free_on_write(void *data, size_t size,
1482     void (*free_func)(void *, size_t))
1483 {
1484         l2arc_data_free_t *df;

1486         df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
1487         df->l2df_data = data;
```



```
1488         df->l2df_size = size;
1489         df->l2df_func = free_func;
1490         mutex_enter(&l2arc_free_on_write_mtx);
1491         list_insert_head(l2arc_free_on_write, df);
1492         mutex_exit(&l2arc_free_on_write_mtx);
1493 }

1495 #endif /* ! codereview */
1496 /*
1497  * Free the arc data buffer.  If it is an l2arc write in progress,
1498  * the buffer is placed on l2arc_free_on_write to be freed later.
1499  */
1500 static void
1501 arc_buf_data_free(arc_buf_t *buf, void (*free_func)(void *, size_t))
1502 {
1503         arc_buf_hdr_t *hdr = buf->b_hdr;

1505         if (HDR_L2_WRITING(hdr)) {
1506                 arc_buf_free_on_write(buf->b_data, hdr->b_size, free_func);
 311                 l2arc_data_free_t *df;
 312                 df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
 313                 df->l2df_data = buf->b_data;
 314                 df->l2df_size = hdr->b_size;
 315                 df->l2df_func = free_func;
 316                 mutex_enter(&l2arc_free_on_write_mtx);
 317                 list_insert_head(l2arc_free_on_write, df);
 318                 mutex_exit(&l2arc_free_on_write_mtx);
1507                 ARCSTAT_BUMP(arcstat_l2_free_on_write);
1508         } else {
1509                 free_func(buf->b_data, hdr->b_size);
1510         }
1511 }

1513 /*
1514  * Free up buf->b_data and if 'remove' is set, then pull the
1515  * arc_buf_t off of the the arc_buf_hdr_t's list and free it.
1516  */
1517 static void
1518 arc_buf_l2_cdata_free(arc_buf_hdr_t *hdr)
1519 {
1520         l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;

1522         ASSERT(MUTEX_HELD(&l2arc_buflist_mtx));

1524         if (l2hdr->b_tmp_cdata == NULL)
1525                 return;

1527         ASSERT(HDR_L2_WRITING(hdr));
1528         arc_buf_free_on_write(l2hdr->b_tmp_cdata, hdr->b_size,
1529             zio_data_buf_free);
1530         ARCSTAT_BUMP(arcstat_l2_cdata_free_on_write);
1531         l2hdr->b_tmp_cdata = NULL;
1532 }

1534 static void
1535 #endif /* ! codereview */
1536 arc_buf_destroy(arc_buf_t *buf, boolean_t recycle, boolean_t remove)
1537 {
1538         arc_buf_t **bufp;

1540         /* free up data associated with the buf */
1541         if (buf->b_data) {
1542                 arc_state_t *state = buf->b_hdr->b_state;
1543                 uint64_t size = buf->b_hdr->b_size;
1544                 arc_buf_contents_t type = buf->b_hdr->b_type;
```

```
1546                arc_cksum_verify(buf);
1547                arc_buf_unwatch(buf);

1549                if (!recycle) {
1550                        if (type == ARC_BUFC_METADATA) {
1551                                arc_buf_data_free(buf, zio_buf_free);
1552                                arc_space_return(size, ARC_SPACE_DATA);
1553                        } else {
1554                                ASSERT(type == ARC_BUFC_DATA);
1555                                arc_buf_data_free(buf, zio_data_buf_free);
1556                                ARCSTAT_INCR(arcstat_data_size, -size);
1557                                atomic_add_64(&arc_size, -size);
1558                        }
1559                }
1560                if (list_link_active(&buf->b_hdr->b_arc_node)) {
1561                        uint64_t *cnt = &state->arcs_lsize[type];

1563                        ASSERT(refcount_is_zero(&buf->b_hdr->b_refcnt));
1564                        ASSERT(state != arc_anon);

1566                        ASSERT3U(*cnt, >=, size);
1567                        atomic_add_64(cnt, -size);
1568                }
1569                ASSERT3U(state->arcs_size, >=, size);
1570                atomic_add_64(&state->arcs_size, -size);
1571                buf->b_data = NULL;

1573                /*
1574                 * If we're destroying a duplicate buffer make sure
1575                 * that the appropriate statistics are updated.
1576                 */
1577                if (buf->b_hdr->b_datacnt > 1 &&
1578                    buf->b_hdr->b_type == ARC_BUFC_DATA) {
1579                        ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
1580                        ARCSTAT_INCR(arcstat_duplicate_buffers_size, -size);
1581                }
1582                ASSERT(buf->b_hdr->b_datacnt > 0);
1583                buf->b_hdr->b_datacnt -= 1;
1584        }

1586        /* only remove the buf if requested */
1587        if (!remove)
1588                return;

1590        /* remove the buf from the hdr list */
1591        for (bufp = &buf->b_hdr->b_buf; *bufp != buf; bufp = &(*bufp)->b_next)
1592                continue;
1593        *bufp = buf->b_next;
1594        buf->b_next = NULL;

1596        ASSERT(buf->b_efunc == NULL);

1598        /* clean up the buf */
1599        buf->b_hdr = NULL;
1600        kmem_cache_free(buf_cache, buf);
1601 }

1603 static void
1604 arc_hdr_destroy(arc_buf_hdr_t *hdr)
1605 {
1606        ASSERT(refcount_is_zero(&hdr->b_refcnt));
1607        ASSERT3P(hdr->b_state, ==, arc_anon);
1608        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
1609        l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;

1611        if (l2hdr != NULL) {
```



```
1612                boolean_t buflist_held = MUTEX_HELD(&l2arc_buflist_mtx);
1613                /*
1614                 * To prevent arc_free() and l2arc_evict() from
1615                 * attempting to free the same buffer at the same time,
1616                 * a FREE_IN_PROGRESS flag is given to arc_free() to
1617                 * give it priority.  l2arc_evict() can't destroy this
1618                 * header while we are waiting on l2arc_buflist_mtx.
1619                 *
1620                 * The hdr may be removed from l2ad_buflist before we
1621                 * grab l2arc_buflist_mtx, so b_l2hdr is rechecked.
1622                 */
1623                if (!buflist_held) {
1624                        mutex_enter(&l2arc_buflist_mtx);
1625                        l2hdr = hdr->b_l2hdr;
1626                }

1628                if (l2hdr != NULL) {
1629                        list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
1630                        arc_buf_l2_cdata_free(hdr);
1631 #endif /* ! codereview */
1632                        ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);
1633                        ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
1634                        vdev_space_update(l2hdr->b_dev->l2ad_vdev,
1635                            -l2hdr->b_asize, 0, 0);
1636                        kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
1637                        if (hdr->b_state == arc_l2c_only)
1638                                l2arc_hdr_stat_remove();
1639                        hdr->b_l2hdr = NULL;
1640                }

1642                if (!buflist_held)
1643                        mutex_exit(&l2arc_buflist_mtx);
1644        }

1646        if (!BUF_EMPTY(hdr)) {
1647                ASSERT(!HDR_IN_HASH_TABLE(hdr));
1648                buf_discard_identity(hdr);
1649        }
1650        while (hdr->b_buf) {
1651                arc_buf_t *buf = hdr->b_buf;

1653                if (buf->b_efunc) {
1654                        mutex_enter(&arc_eviction_mtx);
1655                        mutex_enter(&buf->b_evict_lock);
1656                        ASSERT(buf->b_hdr != NULL);
1657                        arc_buf_destroy(hdr->b_buf, FALSE, FALSE);
1658                        hdr->b_buf = buf->b_next;
1659                        buf->b_hdr = &arc_eviction_hdr;
1660                        buf->b_next = arc_eviction_list;
1661                        arc_eviction_list = buf;
1662                        mutex_exit(&buf->b_evict_lock);
1663                        mutex_exit(&arc_eviction_mtx);
1664                } else {
1665                        arc_buf_destroy(hdr->b_buf, FALSE, TRUE);
1666                }
1667        }
1668        if (hdr->b_freeze_cksum != NULL) {
1669                kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1670                hdr->b_freeze_cksum = NULL;
1671        }
1672        if (hdr->b_thawed) {
1673                kmem_free(hdr->b_thawed, 1);
1674                hdr->b_thawed = NULL;
1675        }

1677        ASSERT(!list_link_active(&hdr->b_arc_node));
```

```
1678         ASSERT3P(hdr->b_hash_next, ==, NULL);
1679         ASSERT3P(hdr->b_acb, ==, NULL);
1680         kmem_cache_free(hdr_cache, hdr);
1681 }

1683 void
1684 arc_buf_free(arc_buf_t *buf, void *tag)
1685 {
1686         arc_buf_hdr_t *hdr = buf->b_hdr;
1687         int hashed = hdr->b_state != arc_anon;

1689         ASSERT(buf->b_efunc == NULL);
1690         ASSERT(buf->b_data != NULL);

1692         if (hashed) {
1693                 kmutex_t *hash_lock = HDR_LOCK(hdr);

1695                 mutex_enter(hash_lock);
1696                 hdr = buf->b_hdr;
1697                 ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

1699                 (void) remove_reference(hdr, hash_lock, tag);
1700                 if (hdr->b_datacnt > 1) {
1701                         arc_buf_destroy(buf, FALSE, TRUE);
1702                 } else {
1703                         ASSERT(buf == hdr->b_buf);
1704                         ASSERT(buf->b_efunc == NULL);
1705                         hdr->b_flags |= ARC_BUF_AVAILABLE;
1706                 }
1707                 mutex_exit(hash_lock);
1708         } else if (HDR_IO_IN_PROGRESS(hdr)) {
1709                 int destroy_hdr;
1710                 /*
1711                  * We are in the middle of an async write.  Don't destroy
1712                  * this buffer unless the write completes before we finish
1713                  * decrementing the reference count.
1714                  */
1715                 mutex_enter(&arc_eviction_mtx);
1716                 (void) remove_reference(hdr, NULL, tag);
1717                 ASSERT(refcount_is_zero(&hdr->b_refcnt));
1718                 destroy_hdr = !HDR_IO_IN_PROGRESS(hdr);
1719                 mutex_exit(&arc_eviction_mtx);
1720                 if (destroy_hdr)
1721                         arc_hdr_destroy(hdr);
1722         } else {
1723                 if (remove_reference(hdr, NULL, tag) > 0)
1724                         arc_buf_destroy(buf, FALSE, TRUE);
1725                 else
1726                         arc_hdr_destroy(hdr);
1727         }
1728 }

1730 boolean_t
1731 arc_buf_remove_ref(arc_buf_t *buf, void* tag)
1732 {
1733         arc_buf_hdr_t *hdr = buf->b_hdr;
1734         kmutex_t *hash_lock = HDR_LOCK(hdr);
1735         boolean_t no_callback = (buf->b_efunc == NULL);

1737         if (hdr->b_state == arc_anon) {
1738                 ASSERT(hdr->b_datacnt == 1);
1739                 arc_buf_free(buf, tag);
1740                 return (no_callback);
1741         }

1743         mutex_enter(hash_lock);
```



```
1744         hdr = buf->b_hdr;
1745         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
1746         ASSERT(hdr->b_state != arc_anon);
1747         ASSERT(buf->b_data != NULL);

1749         (void) remove_reference(hdr, hash_lock, tag);
1750         if (hdr->b_datacnt > 1) {
1751                 if (no_callback)
1752                         arc_buf_destroy(buf, FALSE, TRUE);
1753         } else if (no_callback) {
1754                 ASSERT(hdr->b_buf == buf && buf->b_next == NULL);
1755                 ASSERT(buf->b_efunc == NULL);
1756                 hdr->b_flags |= ARC_BUF_AVAILABLE;
1757         }
1758         ASSERT(no_callback || hdr->b_datacnt > 1 ||
1759             refcount_is_zero(&hdr->b_refcnt));
1760         mutex_exit(hash_lock);
1761         return (no_callback);
1762 }

1764 int
1765 arc_buf_size(arc_buf_t *buf)
1766 {
1767         return (buf->b_hdr->b_size);
1768 }

1770 /*
1771  * Called from the DMU to determine if the current buffer should be
1772  * evicted. In order to ensure proper locking, the eviction must be initiated
1773  * from the DMU. Return true if the buffer is associated with user data and
1774  * duplicate buffers still exist.
1775  */
1776 boolean_t
1777 arc_buf_eviction_needed(arc_buf_t *buf)
1778 {
1779         arc_buf_hdr_t *hdr;
1780         boolean_t evict_needed = B_FALSE;

1782         if (zfs_disable_dup_eviction)
1783                 return (B_FALSE);

1785         mutex_enter(&buf->b_evict_lock);
1786         hdr = buf->b_hdr;
1787         if (hdr == NULL) {
1788                 /*
1789                  * We are in arc_do_user_evicts(); let that function
1790                  * perform the eviction.
1791                  */
1792                 ASSERT(buf->b_data == NULL);
1793                 mutex_exit(&buf->b_evict_lock);
1794                 return (B_FALSE);
1795         } else if (buf->b_data == NULL) {
1796                 /*
1797                  * We have already been added to the arc eviction list;
1798                  * recommend eviction.
1799                  */
1800                 ASSERT3P(hdr, ==, &arc_eviction_hdr);
1801                 mutex_exit(&buf->b_evict_lock);
1802                 return (B_TRUE);
1803         }

1805         if (hdr->b_datacnt > 1 && hdr->b_type == ARC_BUFC_DATA)
1806                 evict_needed = B_TRUE;

1808         mutex_exit(&buf->b_evict_lock);
1809         return (evict_needed);
```

```
1810 }
1812 /*
1813  * Evict buffers from list until we've removed the specified number of
1814  * bytes.  Move the removed buffers to the appropriate evict state.
1815  * If the recycle flag is set, then attempt to "recycle" a buffer:
1816  * - look for a buffer to evict that is `bytes' long.
1817  * - return the data block from this buffer rather than freeing it.
1818  * This flag is used by callers that are trying to make space for a
1819  * new buffer in a full arc cache.
1820  *
1821  * This function makes a "best effort".  It skips over any buffers
1822  * it can't get a hash_lock on, and so may not catch all candidates.
1823  * It may also return without evicting as much space as requested.
1824  */
1825 static void *
1826 arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle,
1827     arc_buf_contents_t type)
1828 {
1829         arc_state_t *evicted_state;
1830         uint64_t bytes_evicted = 0, skipped = 0, missed = 0;
1831         arc_buf_hdr_t *ab, *ab_prev = NULL;
1832         list_t *list = &state->arcs_list[type];
1833         kmutex_t *hash_lock;
1834         boolean_t have_lock;
1835         void *stolen = NULL;
1836         arc_buf_hdr_t marker = { 0 };
1837         int count = 0;
1839         ASSERT(state == arc_mru || state == arc_mfu);
1841         evicted_state = (state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;
1843         mutex_enter(&state->arcs_mtx);
1844         mutex_enter(&evicted_state->arcs_mtx);
1846         for (ab = list_tail(list); ab; ab = ab_prev) {
1847                 ab_prev = list_prev(list, ab);
1848                 /* prefetch buffers have a minimum lifespan */
1849                 if (HDR_IO_IN_PROGRESS(ab) ||
1850                     (spa && ab->b_spa != spa) ||
1851                     (ab->b_flags & (ARC_PREFETCH|ARC_INDIRECT) &&
1852                     ddi_get_lbolt() - ab->b_arc_access <
1853                     arc_min_prefetch_lifespan)) {
1854                         skipped++;
1855                         continue;
1856                 }
1857                 /* "lookahead" for better eviction candidate */
1858                 if (recycle && ab->b_size != bytes &&
1859                     ab_prev && ab_prev->b_size == bytes)
1860                         continue;
1862                 /* ignore markers */
1863                 if (ab->b_spa == 0)
1864                         continue;
1866                 /*
1867                  * It may take a long time to evict all the bufs requested.
1868                  * To avoid blocking all arc activity, periodically drop
1869                  * the arcs_mtx and give other threads a chance to run
1870                  * before reacquiring the lock.
1871                  *
1872                  * If we are looking for a buffer to recycle, we are in
1873                  * the hot code path, so don't sleep.
1874                  */
1875                 if (!recycle && count++ > arc_evict_iterations) {
```



```
1876                         list_insert_after(list, ab, &marker);
1877                         mutex_exit(&evicted_state->arcs_mtx);
1878                         mutex_exit(&state->arcs_mtx);
1879                         kpreempt(KPREEMPT_SYNC);
1880                         mutex_enter(&state->arcs_mtx);
1881                         mutex_enter(&evicted_state->arcs_mtx);
1882                         ab_prev = list_prev(list, &marker);
1883                         list_remove(list, &marker);
1884                         count = 0;
1885                         continue;
1886                 }
1888                 hash_lock = HDR_LOCK(ab);
1889                 have_lock = MUTEX_HELD(hash_lock);
1890                 if (have_lock || mutex_tryenter(hash_lock)) {
1891                         ASSERT0(refcount_count(&ab->b_refcnt));
1892                         ASSERT(ab->b_datacnt > 0);
1893                         while (ab->b_buf) {
1894                                 arc_buf_t *buf = ab->b_buf;
1895                                 if (!mutex_tryenter(&buf->b_evict_lock)) {
1896                                         missed += 1;
1897                                         break;
1898                                 }
1899                                 if (buf->b_data) {
1900                                         bytes_evicted += ab->b_size;
1901                                         if (recycle && ab->b_type == type &&
1902                                             ab->b_size == bytes &&
1903                                             !HDR_L2_WRITING(ab)) {
1904                                                 stolen = buf->b_data;
1905                                                 recycle = FALSE;
1906                                         }
1907                                 }
1908                                 if (buf->b_efunc) {
1909                                         mutex_enter(&arc_eviction_mtx);
1910                                         arc_buf_destroy(buf,
1911                                             buf->b_data == stolen, FALSE);
1912                                         ab->b_buf = buf->b_next;
1913                                         buf->b_hdr = &arc_eviction_hdr;
1914                                         buf->b_next = arc_eviction_list;
1915                                         arc_eviction_list = buf;
1916                                         mutex_exit(&arc_eviction_mtx);
1917                                         mutex_exit(&buf->b_evict_lock);
1918                                 } else {
1919                                         mutex_exit(&buf->b_evict_lock);
1920                                         arc_buf_destroy(buf,
1921                                             buf->b_data == stolen, TRUE);
1922                                 }
1923                         }
1925                         if (ab->b_l2hdr) {
1926                                 ARCSTAT_INCR(arcstat_evict_l2_cached,
1927                                     ab->b_size);
1928                         } else {
1929                                 if (l2arc_write_eligible(ab->b_spa, ab)) {
1930                                         ARCSTAT_INCR(arcstat_evict_l2_eligible,
1931                                             ab->b_size);
1932                                 } else {
1933                                         ARCSTAT_INCR(
1934                                             arcstat_evict_l2_ineligible,
1935                                             ab->b_size);
1936                                 }
1937                         }
1939                         if (ab->b_datacnt == 0) {
1940                                 arc_change_state(evicted_state, ab, hash_lock);
1941                                 ASSERT(HDR_IN_HASH_TABLE(ab));
```

```
1942                                ab->b_flags |= ARC_IN_HASH_TABLE;
1943                                ab->b_flags &= ~ARC_BUF_AVAILABLE;
1944                                DTRACE_PROBE1(arc__evict, arc_buf_hdr_t *, ab);
1945                        }
1946                        if (!have_lock)
1947                                mutex_exit(hash_lock);
1948                        if (bytes >= 0 && bytes_evicted >= bytes)
1949                                break;
1950                } else {
1951                        missed += 1;
1952                }
1953        }

1955        mutex_exit(&evicted_state->arcs_mtx);
1956        mutex_exit(&state->arcs_mtx);

1958        if (bytes_evicted < bytes)
1959                dprintf("only evicted %lld bytes from %x",
1960                    (longlong_t)bytes_evicted, state);

1962        if (skipped)
1963                ARCSTAT_INCR(arcstat_evict_skip, skipped);

1965        if (missed)
1966                ARCSTAT_INCR(arcstat_mutex_miss, missed);

1968        /*
1969         * Note: we have just evicted some data into the ghost state,
1970         * potentially putting the ghost size over the desired size.  Rather
1971         * that evicting from the ghost list in this hot code path, leave
1972         * this chore to the arc_reclaim_thread().
1973         */

1975        return (stolen);
1976 }

1978 /*
1979  * Remove buffers from list until we've removed the specified number of
1980  * bytes.  Destroy the buffers that are removed.
1981  */
1982 static void
1983 arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes)
1984 {
1985        arc_buf_hdr_t *ab, *ab_prev;
1986        arc_buf_hdr_t marker = { 0 };
1987        list_t *list = &state->arcs_list[ARC_BUFC_DATA];
1988        kmutex_t *hash_lock;
1989        uint64_t bytes_deleted = 0;
1990        uint64_t bufs_skipped = 0;
1991        int count = 0;

1993        ASSERT(GHOST_STATE(state));
1994 top:
1995        mutex_enter(&state->arcs_mtx);
1996        for (ab = list_tail(list); ab; ab = ab_prev) {
1997                ab_prev = list_prev(list, ab);
1998                if (ab->b_type > ARC_BUFC_NUMTYPES)
1999                        panic("invalid ab=%p", (void *)ab);
2000                if (spa && ab->b_spa != spa)
2001                        continue;

2003                /* ignore markers */
2004                if (ab->b_spa == 0)
2005                        continue;

2007                hash_lock = HDR_LOCK(ab);
```



```
2008                /* caller may be trying to modify this buffer, skip it */
2009                if (MUTEX_HELD(hash_lock))
2010                        continue;

2012                /*
2013                 * It may take a long time to evict all the bufs requested.
2014                 * To avoid blocking all arc activity, periodically drop
2015                 * the arcs_mtx and give other threads a chance to run
2016                 * before reacquiring the lock.
2017                 */
2018                if (count++ > arc_evict_iterations) {
2019                        list_insert_after(list, ab, &marker);
2020                        mutex_exit(&state->arcs_mtx);
2021                        kpreempt(KPREEMPT_SYNC);
2022                        mutex_enter(&state->arcs_mtx);
2023                        ab_prev = list_prev(list, &marker);
2024                        list_remove(list, &marker);
2025                        count = 0;
2026                        continue;
2027                }
2028                if (mutex_tryenter(hash_lock)) {
2029                        ASSERT(!HDR_IO_IN_PROGRESS(ab));
2030                        ASSERT(ab->b_buf == NULL);
2031                        ARCSTAT_BUMP(arcstat_deleted);
2032                        bytes_deleted += ab->b_size;

2034                        if (ab->b_l2hdr != NULL) {
2035                                /*
2036                                 * This buffer is cached on the 2nd Level ARC;
2037                                 * don't destroy the header.
2038                                 */
2039                                arc_change_state(arc_l2c_only, ab, hash_lock);
2040                                mutex_exit(hash_lock);
2041                        } else {
2042                                arc_change_state(arc_anon, ab, hash_lock);
2043                                mutex_exit(hash_lock);
2044                                arc_hdr_destroy(ab);
2045                        }

2047                        DTRACE_PROBE1(arc__delete, arc_buf_hdr_t *, ab);
2048                        if (bytes >= 0 && bytes_deleted >= bytes)
2049                                break;
2050                } else if (bytes < 0) {
2051                        /*
2052                         * Insert a list marker and then wait for the
2053                         * hash lock to become available. Once its
2054                         * available, restart from where we left off.
2055                         */
2056                        list_insert_after(list, ab, &marker);
2057                        mutex_exit(&state->arcs_mtx);
2058                        mutex_enter(hash_lock);
2059                        mutex_exit(hash_lock);
2060                        mutex_enter(&state->arcs_mtx);
2061                        ab_prev = list_prev(list, &marker);
2062                        list_remove(list, &marker);
2063                } else {
2064                        bufs_skipped += 1;
2065                }

2067        }
2068        mutex_exit(&state->arcs_mtx);

2070        if (list == &state->arcs_list[ARC_BUFC_DATA] &&
2071            (bytes < 0 || bytes_deleted < bytes)) {
2072                list = &state->arcs_list[ARC_BUFC_METADATA];
2073                goto top;
```

```
2074         }
2076         if (bufs_skipped) {
2077                 ARCSTAT_INCR(arcstat_mutex_miss, bufs_skipped);
2078                 ASSERT(bytes >= 0);
2079         }
2081         if (bytes_deleted < bytes)
2082                 dprintf("only deleted %lld bytes from %p",
2083                     (longlong_t)bytes_deleted, state);
2084 }
2086 static void
2087 arc_adjust(void)
2088 {
2089         int64_t adjustment, delta;
2091         /*
2092          * Adjust MRU size
2093          */
2095         adjustment = MIN((int64_t)(arc_size - arc_c),
2096             (int64_t)(arc_anon->arcs_size + arc_mru->arcs_size + arc_meta_used -
2097             arc_p));
2099         if (adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_DATA] > 0) {
2100                 delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_DATA], adjustment);
2101                 (void) arc_evict(arc_mru, NULL, delta, FALSE, ARC_BUFC_DATA);
2102                 adjustment -= delta;
2103         }
2105         if (adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_METADATA] > 0) {
2106                 delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_METADATA], adjustment);
2107                 (void) arc_evict(arc_mru, NULL, delta, FALSE,
2108                     ARC_BUFC_METADATA);
2109         }
2111         /*
2112          * Adjust MFU size
2113          */
2115         adjustment = arc_size - arc_c;
2117         if (adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_DATA] > 0) {
2118                 delta = MIN(adjustment, arc_mfu->arcs_lsize[ARC_BUFC_DATA]);
2119                 (void) arc_evict(arc_mfu, NULL, delta, FALSE, ARC_BUFC_DATA);
2120                 adjustment -= delta;
2121         }
2123         if (adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_METADATA] > 0) {
2124                 int64_t delta = MIN(adjustment,
2125                     arc_mfu->arcs_lsize[ARC_BUFC_METADATA]);
2126                 (void) arc_evict(arc_mfu, NULL, delta, FALSE,
2127                     ARC_BUFC_METADATA);
2128         }
2130         /*
2131          * Adjust ghost lists
2132          */
2134         adjustment = arc_mru->arcs_size + arc_mru_ghost->arcs_size - arc_c;
2136         if (adjustment > 0 && arc_mru_ghost->arcs_size > 0) {
2137                 delta = MIN(arc_mru_ghost->arcs_size, adjustment);
2138                 arc_evict_ghost(arc_mru_ghost, NULL, delta);
2139         }
```



```
2141         adjustment =
2142             arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size - arc_c;
2144         if (adjustment > 0 && arc_mfu_ghost->arcs_size > 0) {
2145                 delta = MIN(arc_mfu_ghost->arcs_size, adjustment);
2146                 arc_evict_ghost(arc_mfu_ghost, NULL, delta);
2147         }
2148 }
2150 static void
2151 arc_do_user_evicts(void)
2152 {
2153         mutex_enter(&arc_eviction_mtx);
2154         while (arc_eviction_list != NULL) {
2155                 arc_buf_t *buf = arc_eviction_list;
2156                 arc_eviction_list = buf->b_next;
2157                 mutex_enter(&buf->b_evict_lock);
2158                 buf->b_hdr = NULL;
2159                 mutex_exit(&buf->b_evict_lock);
2160                 mutex_exit(&arc_eviction_mtx);
2162                 if (buf->b_efunc != NULL)
2163                         VERIFY0(buf->b_efunc(buf->b_private));
2165                 buf->b_efunc = NULL;
2166                 buf->b_private = NULL;
2167                 kmem_cache_free(buf_cache, buf);
2168                 mutex_enter(&arc_eviction_mtx);
2169         }
2170         mutex_exit(&arc_eviction_mtx);
2171 }
2173 /*
2174  * Flush all *evictable* data from the cache for the given spa.
2175  * NOTE: this will not touch "active" (i.e. referenced) data.
2176  */
2177 void
2178 arc_flush(spa_t *spa)
2179 {
2180         uint64_t guid = 0;
2182         if (spa)
2183                 guid = spa_load_guid(spa);
2185         while (list_head(&arc_mru->arcs_list[ARC_BUFC_DATA])) {
2186                 (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_DATA);
2187                 if (spa)
2188                         break;
2189         }
2190         while (list_head(&arc_mru->arcs_list[ARC_BUFC_METADATA])) {
2191                 (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_METADATA);
2192                 if (spa)
2193                         break;
2194         }
2195         while (list_head(&arc_mfu->arcs_list[ARC_BUFC_DATA])) {
2196                 (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_DATA);
2197                 if (spa)
2198                         break;
2199         }
2200         while (list_head(&arc_mfu->arcs_list[ARC_BUFC_METADATA])) {
2201                 (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_METADATA);
2202                 if (spa)
2203                         break;
2204         }
```

```
2206         arc_evict_ghost(arc_mru_ghost, guid, -1);
2207         arc_evict_ghost(arc_mfu_ghost, guid, -1);

2209         mutex_enter(&arc_reclaim_thr_lock);
2210         arc_do_user_evicts();
2211         mutex_exit(&arc_reclaim_thr_lock);
2212         ASSERT(spa || arc_eviction_list == NULL);
2213 }

2215 void
2216 arc_shrink(void)
2217 {
2218         if (arc_c > arc_c_min) {
2219                 uint64_t to_free;

2221 #ifdef _KERNEL
2222                 to_free = MAX(arc_c >> arc_shrink_shift, ptob(needfree));
2223 #else
2224                 to_free = arc_c >> arc_shrink_shift;
2225 #endif
2226                 if (arc_c > arc_c_min + to_free)
2227                         atomic_add_64(&arc_c, -to_free);
2228                 else
2229                         arc_c = arc_c_min;

2231                 atomic_add_64(&arc_p, -(arc_p >> arc_shrink_shift));
2232                 if (arc_c > arc_size)
2233                         arc_c = MAX(arc_size, arc_c_min);
2234                 if (arc_p > arc_c)
2235                         arc_p = (arc_c >> 1);
2236                 ASSERT(arc_c >= arc_c_min);
2237                 ASSERT((int64_t)arc_p >= 0);
2238         }

2240         if (arc_size > arc_c)
2241                 arc_adjust();
2242 }

2244 /*
2245  * Determine if the system is under memory pressure and is asking
2246  * to reclaim memory. A return value of 1 indicates that the system
2247  * is under memory pressure and that the arc should adjust accordingly.
2248  */
2249 static int
2250 arc_reclaim_needed(void)
2251 {
2252         uint64_t extra;

2254 #ifdef _KERNEL

2256         if (needfree)
2257                 return (1);

2259         /*
2260          * take 'desfree' extra pages, so we reclaim sooner, rather than later
2261          */
2262         extra = desfree;

2264         /*
2265          * check that we're out of range of the pageout scanner.  It starts to
2266          * schedule paging if freemem is less than lotsfree and needfree.
2267          * lotsfree is the high-water mark for pageout, and needfree is the
2268          * number of needed free pages.  We add extra pages here to make sure
2269          * the scanner doesn't start up while we're freeing memory.
2270          */
2271         if (freemem < lotsfree + needfree + extra)
```



```
2272                 return (1);

2274         /*
2275          * check to make sure that swapfs has enough space so that anon
2276          * reservations can still succeed. anon_resvmem() checks that the
2277          * availrmem is greater than swapfs_minfree, and the number of reserved
2278          * swap pages.  We also add a bit of extra here just to prevent
2279          * circumstances from getting really dire.
2280          */
2281         if (availrmem < swapfs_minfree + swapfs_reserve + extra)
2282                 return (1);

2284         /*
2285          * Check that we have enough availrmem that memory locking (e.g., via
2286          * mlock(3C) or memcntl(2)) can still succeed.  (pages_pp_maximum
2287          * stores the number of pages that cannot be locked; when availrmem
2288          * drops below pages_pp_maximum, page locking mechanisms such as
2289          * page_pp_lock() will fail.)
2290          */
2291         if (availrmem <= pages_pp_maximum)
2292                 return (1);

2294 #if defined(__i386)
2295         /*
2296          * If we're on an i386 platform, it's possible that we'll exhaust the
2297          * kernel heap space before we ever run out of available physical
2298          * memory.  Most checks of the size of the heap_area compare against
2299          * tune.t_minarmem, which is the minimum available real memory that we
2300          * can have in the system.  However, this is generally fixed at 25 pages
2301          * which is so low that it's useless.  In this comparison, we seek to
2302          * calculate the total heap-size, and reclaim if more than 3/4ths of the
2303          * heap is allocated.  (Or, in the calculation, if less than 1/4th is
2304          * free)
2305          */
2306         if (vmem_size(heap_arena, VMEM_FREE) <
2307             (vmem_size(heap_arena, VMEM_FREE | VMEM_ALLOC) >> 2))
2308                 return (1);
2309 #endif

2311         /*
2312          * If zio data pages are being allocated out of a separate heap segment,
2313          * then enforce that the size of available vmem for this arena remains
2314          * above about 1/16th free.
2315          *
2316          * Note: The 1/16th arena free requirement was put in place
2317          * to aggressively evict memory from the arc in order to avoid
2318          * memory fragmentation issues.
2319          */
2320         if (zio_arena != NULL &&
2321             vmem_size(zio_arena, VMEM_FREE) <
2322             (vmem_size(zio_arena, VMEM_ALLOC) >> 4))
2323                 return (1);
2324 #else
2325         if (spa_get_random(100) == 0)
2326                 return (1);
2327 #endif
2328         return (0);
2329 }

2331 static void
2332 arc_kmem_reap_now(arc_reclaim_strategy_t strat)
2333 {
2334         size_t                  i;
2335         kmem_cache_t            *prev_cache = NULL;
2336         kmem_cache_t            *prev_data_cache = NULL;
2337         extern kmem_cache_t     *zio_buf_cache[];
```

```
2338         extern kmem_cache_t     *zio_data_buf_cache[];
2339         extern kmem_cache_t     *range_seg_cache;

2341 #ifdef _KERNEL
2342         if (arc_meta_used >= arc_meta_limit) {
2343                 /*
2344                  * We are exceeding our meta-data cache limit.
2345                  * Purge some DNLC entries to release holds on meta-data.
2346                  */
2347                 dnlc_reduce_cache((void *)(uintptr_t)arc_reduce_dnlc_percent);
2348         }
2349 #if defined(__i386)
2350         /*
2351          * Reclaim unused memory from all kmem caches.
2352          */
2353         kmem_reap();
2354 #endif
2355 #endif

2357         /*
2358          * An aggressive reclamation will shrink the cache size as well as
2359          * reap free buffers from the arc kmem caches.
2360          */
2361         if (strat == ARC_RECLAIM_AGGR)
2362                 arc_shrink();

2364         for (i = 0; i < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT; i++) {
2365                 if (zio_buf_cache[i] != prev_cache) {
2366                         prev_cache = zio_buf_cache[i];
2367                         kmem_cache_reap_now(zio_buf_cache[i]);
2368                 }
2369                 if (zio_data_buf_cache[i] != prev_data_cache) {
2370                         prev_data_cache = zio_data_buf_cache[i];
2371                         kmem_cache_reap_now(zio_data_buf_cache[i]);
2372                 }
2373         }
2374         kmem_cache_reap_now(buf_cache);
2375         kmem_cache_reap_now(hdr_cache);
2376         kmem_cache_reap_now(range_seg_cache);

2378         /*
2379          * Ask the vmem areana to reclaim unused memory from its
2380          * quantum caches.
2381          */
2382         if (zio_arena != NULL && strat == ARC_RECLAIM_AGGR)
2383                 vmem_qcache_reap(zio_arena);
2384 }

2386 static void
2387 arc_reclaim_thread(void)
2388 {
2389         clock_t                 growtime = 0;
2390         arc_reclaim_strategy_t  last_reclaim = ARC_RECLAIM_CONS;
2391         callb_cpr_t             cpr;

2393         CALLB_CPR_INIT(&cpr, &arc_reclaim_thr_lock, callb_generic_cpr, FTAG);

2395         mutex_enter(&arc_reclaim_thr_lock);
2396         while (arc_thread_exit == 0) {
2397                 if (arc_reclaim_needed()) {

2399                         if (arc_no_grow) {
2400                                 if (last_reclaim == ARC_RECLAIM_CONS) {
2401                                         last_reclaim = ARC_RECLAIM_AGGR;
2402                                 } else {
2403                                         last_reclaim = ARC_RECLAIM_CONS;
```



```
2404                                 }
2405                         } else {
2406                                 arc_no_grow = TRUE;
2407                                 last_reclaim = ARC_RECLAIM_AGGR;
2408                                 membar_producer();
2409                         }

2411                         /* reset the growth delay for every reclaim */
2412                         growtime = ddi_get_lbolt() + (arc_grow_retry * hz);

2414                         arc_kmem_reap_now(last_reclaim);
2415                         arc_warm = B_TRUE;

2417                 } else if (arc_no_grow && ddi_get_lbolt() >= growtime) {
2418                         arc_no_grow = FALSE;
2419                 }

2421                 arc_adjust();

2423                 if (arc_eviction_list != NULL)
2424                         arc_do_user_evicts();

2426                 /* block until needed, or one second, whichever is shorter */
2427                 CALLB_CPR_SAFE_BEGIN(&cpr);
2428                 (void) cv_timedwait(&arc_reclaim_thr_cv,
2429                     &arc_reclaim_thr_lock, (ddi_get_lbolt() + hz));
2430                 CALLB_CPR_SAFE_END(&cpr, &arc_reclaim_thr_lock);
2431         }

2433         arc_thread_exit = 0;
2434         cv_broadcast(&arc_reclaim_thr_cv);
2435         CALLB_CPR_EXIT(&cpr);           /* drops arc_reclaim_thr_lock */
2436         thread_exit();
2437 }

2439 /*
2440  * Adapt arc info given the number of bytes we are trying to add and
2441  * the state that we are comming from.  This function is only called
2442  * when we are adding new content to the cache.
2443  */
2444 static void
2445 arc_adapt(int bytes, arc_state_t *state)
2446 {
2447         int mult;
2448         uint64_t arc_p_min = (arc_c >> arc_p_min_shift);

2450         if (state == arc_l2c_only)
2451                 return;

2453         ASSERT(bytes > 0);
2454         /*
2455          * Adapt the target size of the MRU list:
2456          *      - if we just hit in the MRU ghost list, then increase
2457          *        the target size of the MRU list.
2458          *      - if we just hit in the MFU ghost list, then increase
2459          *        the target size of the MFU list by decreasing the
2460          *        target size of the MRU list.
2461          */
2462         if (state == arc_mru_ghost) {
2463                 mult = ((arc_mru_ghost->arcs_size >= arc_mfu_ghost->arcs_size) ?
2464                     1 : (arc_mfu_ghost->arcs_size/arc_mru_ghost->arcs_size));
2465                 mult = MIN(mult, 10); /* avoid wild arc_p adjustment */

2467                 arc_p = MIN(arc_c - arc_p_min, arc_p + bytes * mult);
2468         } else if (state == arc_mfu_ghost) {
2469                 uint64_t delta;
```

```
2471                 mult = ((arc_mfu_ghost->arcs_size >= arc_mru_ghost->arcs_size) ?
2472                     1 : (arc_mru_ghost->arcs_size/arc_mfu_ghost->arcs_size));
2473                 mult = MIN(mult, 10);
2475                 delta = MIN(bytes * mult, arc_p);
2476                 arc_p = MAX(arc_p_min, arc_p - delta);
2477         }
2478         ASSERT((int64_t)arc_p >= 0);
2480         if (arc_reclaim_needed()) {
2481                 cv_signal(&arc_reclaim_thr_cv);
2482                 return;
2483         }
2485         if (arc_no_grow)
2486                 return;
2488         if (arc_c >= arc_c_max)
2489                 return;
2491         /*
2492          * If we're within (2 * maxblocksize) bytes of the target
2493          * cache size, increment the target cache size
2494          */
2495         if (arc_size > arc_c - (2ULL << SPA_MAXBLOCKSHIFT)) {
2496                 atomic_add_64(&arc_c, (int64_t)bytes);
2497                 if (arc_c > arc_c_max)
2498                         arc_c = arc_c_max;
2499                 else if (state == arc_anon)
2500                         atomic_add_64(&arc_p, (int64_t)bytes);
2501                 if (arc_p > arc_c)
2502                         arc_p = arc_c;
2503         }
2504         ASSERT((int64_t)arc_p >= 0);
2505 }
2507 /*
2508  * Check if the cache has reached its limits and eviction is required
2509  * prior to insert.
2510  */
2511 static int
2512 arc_evict_needed(arc_buf_contents_t type)
2513 {
2514         if (type == ARC_BUFC_METADATA && arc_meta_used >= arc_meta_limit)
2515                 return (1);
2517         if (arc_reclaim_needed())
2518                 return (1);
2520         return (arc_size > arc_c);
2521 }
2523 /*
2524  * The buffer, supplied as the first argument, needs a data block.
2525  * So, if we are at cache max, determine which cache should be victimized.
2526  * We have the following cases:
2527  *
2528  * 1. Insert for MRU, p > sizeof(arc_anon + arc_mru) ->
2529  * In this situation if we're out of space, but the resident size of the MFU is
2530  * under the limit, victimize the MFU cache to satisfy this insertion request.
2531  *
2532  * 2. Insert for MRU, p <= sizeof(arc_anon + arc_mru) ->
2533  * Here, we've used up all of the available space for the MRU, so we need to
2534  * evict from our own cache instead.  Evict from the set of resident MRU
2535  * entries.
```



```
2536  *
2537  * 3. Insert for MFU (c - p) > sizeof(arc_mfu) ->
2538  * c minus p represents the MFU space in the cache, since p is the size of the
2539  * cache that is dedicated to the MRU.  In this situation there's still space on
2540  * the MFU side, so the MRU side needs to be victimized.
2541  *
2542  * 4. Insert for MFU (c - p) < sizeof(arc_mfu) ->
2543  * MFU's resident set is consuming more space than it has been allotted.  In
2544  * this situation, we must victimize our own cache, the MFU, for this insertion.
2545  */
2546 static void
2547 arc_get_data_buf(arc_buf_t *buf)
2548 {
2549         arc_state_t             *state = buf->b_hdr->b_state;
2550         uint64_t                size = buf->b_hdr->b_size;
2551         arc_buf_contents_t      type = buf->b_hdr->b_type;
2553         arc_adapt(size, state);
2555         /*
2556          * We have not yet reached cache maximum size,
2557          * just allocate a new buffer.
2558          */
2559         if (!arc_evict_needed(type)) {
2560                 if (type == ARC_BUFC_METADATA) {
2561                         buf->b_data = zio_buf_alloc(size);
2562                         arc_space_consume(size, ARC_SPACE_DATA);
2563                 } else {
2564                         ASSERT(type == ARC_BUFC_DATA);
2565                         buf->b_data = zio_data_buf_alloc(size);
2566                         ARCSTAT_INCR(arcstat_data_size, size);
2567                         atomic_add_64(&arc_size, size);
2568                 }
2569                 goto out;
2570         }
2572         /*
2573          * If we are prefetching from the mfu ghost list, this buffer
2574          * will end up on the mru list; so steal space from there.
2575          */
2576         if (state == arc_mfu_ghost)
2577                 state = buf->b_hdr->b_flags & ARC_PREFETCH ? arc_mru : arc_mfu;
2578         else if (state == arc_mru_ghost)
2579                 state = arc_mru;
2581         if (state == arc_mru || state == arc_anon) {
2582                 uint64_t mru_used = arc_anon->arcs_size + arc_mru->arcs_size;
2583                 state = (arc_mfu->arcs_lsize[type] >= size &&
2584                     arc_p > mru_used) ? arc_mfu : arc_mru;
2585         } else {
2586                 /* MFU cases */
2587                 uint64_t mfu_space = arc_c - arc_p;
2588                 state =  (arc_mru->arcs_lsize[type] >= size &&
2589                     mfu_space > arc_mfu->arcs_size) ? arc_mru : arc_mfu;
2590         }
2591         if ((buf->b_data = arc_evict(state, NULL, size, TRUE, type)) == NULL) {
2592                 if (type == ARC_BUFC_METADATA) {
2593                         buf->b_data = zio_buf_alloc(size);
2594                         arc_space_consume(size, ARC_SPACE_DATA);
2595                 } else {
2596                         ASSERT(type == ARC_BUFC_DATA);
2597                         buf->b_data = zio_data_buf_alloc(size);
2598                         ARCSTAT_INCR(arcstat_data_size, size);
2599                         atomic_add_64(&arc_size, size);
2600                 }
2601                 ARCSTAT_BUMP(arcstat_recycle_miss);
```

```
2602         }
2603         ASSERT(buf->b_data != NULL);
2604 out:
2605         /*
2606          * Update the state size.  Note that ghost states have a
2607          * "ghost size" and so don't need to be updated.
2608          */
2609         if (!GHOST_STATE(buf->b_hdr->b_state)) {
2610                 arc_buf_hdr_t *hdr = buf->b_hdr;

2612                 atomic_add_64(&hdr->b_state->arcs_size, size);
2613                 if (list_link_active(&hdr->b_arc_node)) {
2614                         ASSERT(refcount_is_zero(&hdr->b_refcnt));
2615                         atomic_add_64(&hdr->b_state->arcs_lsize[type], size);
2616                 }
2617                 /*
2618                  * If we are growing the cache, and we are adding anonymous
2619                  * data, and we have outgrown arc_p, update arc_p
2620                  */
2621                 if (arc_size < arc_c && hdr->b_state == arc_anon &&
2622                     arc_anon->arcs_size + arc_mru->arcs_size > arc_p)
2623                         arc_p = MIN(arc_c, arc_p + size);
2624         }
2625 }

2627 /*
2628  * This routine is called whenever a buffer is accessed.
2629  * NOTE: the hash lock is dropped in this function.
2630  */
2631 static void
2632 arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock)
2633 {
2634         clock_t now;

2636         ASSERT(MUTEX_HELD(hash_lock));

2638         if (buf->b_state == arc_anon) {
2639                 /*
2640                  * This buffer is not in the cache, and does not
2641                  * appear in our "ghost" list.  Add the new buffer
2642                  * to the MRU state.
2643                  */

2645                 ASSERT(buf->b_arc_access == 0);
2646                 buf->b_arc_access = ddi_get_lbolt();
2647                 DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2648                 arc_change_state(arc_mru, buf, hash_lock);

2650         } else if (buf->b_state == arc_mru) {
2651                 now = ddi_get_lbolt();

2653                 /*
2654                  * If this buffer is here because of a prefetch, then either:
2655                  * - clear the flag if this is a "referencing" read
2656                  *   (any subsequent access will bump this into the MFU state).
2657                  * or
2658                  * - move the buffer to the head of the list if this is
2659                  *   another prefetch (to make it less likely to be evicted).
2660                  */
2661                 if ((buf->b_flags & ARC_PREFETCH) != 0) {
2662                         if (refcount_count(&buf->b_refcnt) == 0) {
2663                                 ASSERT(list_link_active(&buf->b_arc_node));
2664                         } else {
2665                                 buf->b_flags &= ~ARC_PREFETCH;
2666                                 ARCSTAT_BUMP(arcstat_mru_hits);
2667                         }
```



```
2668                         buf->b_arc_access = now;
2669                         return;
2670                 }

2672                 /*
2673                  * This buffer has been "accessed" only once so far,
2674                  * but it is still in the cache. Move it to the MFU
2675                  * state.
2676                  */
2677                 if (now > buf->b_arc_access + ARC_MINTIME) {
2678                         /*
2679                          * More than 125ms have passed since we
2680                          * instantiated this buffer.  Move it to the
2681                          * most frequently used state.
2682                          */
2683                         buf->b_arc_access = now;
2684                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2685                         arc_change_state(arc_mfu, buf, hash_lock);
2686                 }
2687                 ARCSTAT_BUMP(arcstat_mru_hits);
2688         } else if (buf->b_state == arc_mru_ghost) {
2689                 arc_state_t     *new_state;
2690                 /*
2691                  * This buffer has been "accessed" recently, but
2692                  * was evicted from the cache.  Move it to the
2693                  * MFU state.
2694                  */

2696                 if (buf->b_flags & ARC_PREFETCH) {
2697                         new_state = arc_mru;
2698                         if (refcount_count(&buf->b_refcnt) > 0)
2699                                 buf->b_flags &= ~ARC_PREFETCH;
2700                         DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2701                 } else {
2702                         new_state = arc_mfu;
2703                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2704                 }

2706                 buf->b_arc_access = ddi_get_lbolt();
2707                 arc_change_state(new_state, buf, hash_lock);

2709                 ARCSTAT_BUMP(arcstat_mru_ghost_hits);
2710         } else if (buf->b_state == arc_mfu) {
2711                 /*
2712                  * This buffer has been accessed more than once and is
2713                  * still in the cache.  Keep it in the MFU state.
2714                  *
2715                  * NOTE: an add_reference() that occurred when we did
2716                  * the arc_read() will have kicked this off the list.
2717                  * If it was a prefetch, we will explicitly move it to
2718                  * the head of the list now.
2719                  */
2720                 if ((buf->b_flags & ARC_PREFETCH) != 0) {
2721                         ASSERT(refcount_count(&buf->b_refcnt) == 0);
2722                         ASSERT(list_link_active(&buf->b_arc_node));
2723                 }
2724                 ARCSTAT_BUMP(arcstat_mfu_hits);
2725                 buf->b_arc_access = ddi_get_lbolt();
2726         } else if (buf->b_state == arc_mfu_ghost) {
2727                 arc_state_t     *new_state = arc_mfu;
2728                 /*
2729                  * This buffer has been accessed more than once but has
2730                  * been evicted from the cache.  Move it back to the
2731                  * MFU state.
2732                  */
```

```
2734                 if (buf->b_flags & ARC_PREFETCH) {
2735                         /*
2736                          * This is a prefetch access...
2737                          * move this block back to the MRU state.
2738                          */
2739                         ASSERT0(refcount_count(&buf->b_refcnt));
2740                         new_state = arc_mru;
2741                 }

2743                 buf->b_arc_access = ddi_get_lbolt();
2744                 DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2745                 arc_change_state(new_state, buf, hash_lock);

2747                 ARCSTAT_BUMP(arcstat_mfu_ghost_hits);
2748         } else if (buf->b_state == arc_l2c_only) {
2749                 /*
2750                  * This buffer is on the 2nd Level ARC.
2751                  */

2753                 buf->b_arc_access = ddi_get_lbolt();
2754                 DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2755                 arc_change_state(arc_mfu, buf, hash_lock);
2756         } else {
2757                 ASSERT(!"invalid arc state");
2758         }
2759 }

2761 /* a generic arc_done_func_t which you can use */
2762 /* ARGSUSED */
2763 void
2764 arc_bcopy_func(zio_t *zio, arc_buf_t *buf, void *arg)
2765 {
2766         if (zio == NULL || zio->io_error == 0)
2767                 bcopy(buf->b_data, arg, buf->b_hdr->b_size);
2768         VERIFY(arc_buf_remove_ref(buf, arg));
2769 }

2771 /* a generic arc_done_func_t */
2772 void
2773 arc_getbuf_func(zio_t *zio, arc_buf_t *buf, void *arg)
2774 {
2775         arc_buf_t **bufp = arg;
2776         if (zio && zio->io_error) {
2777                 VERIFY(arc_buf_remove_ref(buf, arg));
2778                 *bufp = NULL;
2779         } else {
2780                 *bufp = buf;
2781                 ASSERT(buf->b_data);
2782         }
2783 }

2785 static void
2786 arc_read_done(zio_t *zio)
2787 {
2788         arc_buf_hdr_t   *hdr;
2789         arc_buf_t       *buf;
2790         arc_buf_t       *abuf;  /* buffer we're assigning to callback */
2791         kmutex_t        *hash_lock = NULL;
2792         arc_callback_t  *callback_list, *acb;
2793         int             freeable = FALSE;

2795         buf = zio->io_private;
2796         hdr = buf->b_hdr;

2798         /*
2799          * The hdr was inserted into hash-table and removed from lists
```



```
2800          * prior to starting I/O.  We should find this header, since
2801          * it's in the hash table, and it should be legit since it's
2802          * not possible to evict it during the I/O.  The only possible
2803          * reason for it not to be found is if we were freed during the
2804          * read.
2805          */
2806         if (HDR_IN_HASH_TABLE(hdr)) {
2807                 ASSERT3U(hdr->b_birth, ==, BP_PHYSICAL_BIRTH(zio->io_bp));
2808                 ASSERT3U(hdr->b_dva.dva_word[0], ==,
2809                     BP_IDENTITY(zio->io_bp)->dva_word[0]);
2810                 ASSERT3U(hdr->b_dva.dva_word[1], ==,
2811                     BP_IDENTITY(zio->io_bp)->dva_word[1]);

2813                 arc_buf_hdr_t *found = buf_hash_find(hdr->b_spa, zio->io_bp,
2814                     &hash_lock);

2816                 ASSERT((found == NULL && HDR_FREED_IN_READ(hdr) &&
2817                     hash_lock == NULL) ||
2818                     (found == hdr &&
2819                     DVA_EQUAL(&hdr->b_dva, BP_IDENTITY(zio->io_bp))) ||
2820                     (found == hdr && HDR_L2_READING(hdr)));
2821         }

2823         hdr->b_flags &= ~ARC_L2_EVICTED;
2824         if (l2arc_noprefetch && (hdr->b_flags & ARC_PREFETCH))
2825                 hdr->b_flags &= ~ARC_L2CACHE;

2827         /* byteswap if necessary */
2828         callback_list = hdr->b_acb;
2829         ASSERT(callback_list != NULL);
2830         if (BP_SHOULD_BYTESWAP(zio->io_bp) && zio->io_error == 0) {
2831                 dmu_object_byteswap_t bswap =
2832                     DMU_OT_BYTESWAP(BP_GET_TYPE(zio->io_bp));
2833                 arc_byteswap_func_t *func = BP_GET_LEVEL(zio->io_bp) > 0 ?
2834                     byteswap_uint64_array :
2835                     dmu_ot_byteswap[bswap].ob_func;
2836                 func(buf->b_data, hdr->b_size);
2837         }

2839         arc_cksum_compute(buf, B_FALSE);
2840         arc_buf_watch(buf);

2842         if (hash_lock && zio->io_error == 0 && hdr->b_state == arc_anon) {
2843                 /*
2844                  * Only call arc_access on anonymous buffers.  This is because
2845                  * if we've issued an I/O for an evicted buffer, we've already
2846                  * called arc_access (to prevent any simultaneous readers from
2847                  * getting confused).
2848                  */
2849                 arc_access(hdr, hash_lock);
2850         }

2852         /* create copies of the data buffer for the callers */
2853         abuf = buf;
2854         for (acb = callback_list; acb; acb = acb->acb_next) {
2855                 if (acb->acb_done) {
2856                         if (abuf == NULL) {
2857                                 ARCSTAT_BUMP(arcstat_duplicate_reads);
2858                                 abuf = arc_buf_clone(buf);
2859                         }
2860                         acb->acb_buf = abuf;
2861                         abuf = NULL;
2862                 }
2863         }
2864         hdr->b_acb = NULL;
2865         hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
```

```
2866         ASSERT(!HDR_BUF_AVAILABLE(hdr));
2867         if (abuf == buf) {
2868                 ASSERT(buf->b_efunc == NULL);
2869                 ASSERT(hdr->b_datacnt == 1);
2870                 hdr->b_flags |= ARC_BUF_AVAILABLE;
2871         }

2873         ASSERT(refcount_is_zero(&hdr->b_refcnt) || callback_list != NULL);

2875         if (zio->io_error != 0) {
2876                 hdr->b_flags |= ARC_IO_ERROR;
2877                 if (hdr->b_state != arc_anon)
2878                         arc_change_state(arc_anon, hdr, hash_lock);
2879                 if (HDR_IN_HASH_TABLE(hdr))
2880                         buf_hash_remove(hdr);
2881                 freeable = refcount_is_zero(&hdr->b_refcnt);
2882         }

2884         /*
2885          * Broadcast before we drop the hash_lock to avoid the possibility
2886          * that the hdr (and hence the cv) might be freed before we get to
2887          * the cv_broadcast().
2888          */
2889         cv_broadcast(&hdr->b_cv);

2891         if (hash_lock) {
2892                 mutex_exit(hash_lock);
2893         } else {
2894                 /*
2895                  * This block was freed while we waited for the read to
2896                  * complete.  It has been removed from the hash table and
2897                  * moved to the anonymous state (so that it won't show up
2898                  * in the cache).
2899                  */
2900                 ASSERT3P(hdr->b_state, ==, arc_anon);
2901                 freeable = refcount_is_zero(&hdr->b_refcnt);
2902         }

2904         /* execute each callback and free its structure */
2905         while ((acb = callback_list) != NULL) {
2906                 if (acb->acb_done)
2907                         acb->acb_done(zio, acb->acb_buf, acb->acb_private);

2909                 if (acb->acb_zio_dummy != NULL) {
2910                         acb->acb_zio_dummy->io_error = zio->io_error;
2911                         zio_nowait(acb->acb_zio_dummy);
2912                 }

2914                 callback_list = acb->acb_next;
2915                 kmem_free(acb, sizeof (arc_callback_t));
2916         }

2918         if (freeable)
2919                 arc_hdr_destroy(hdr);
2920 }

2922 /*
2923  * "Read" the block at the specified DVA (in bp) via the
2924  * cache.  If the block is found in the cache, invoke the provided
2925  * callback immediately and return.  Note that the 'zio' parameter
2926  * in the callback will be NULL in this case, since no IO was
2927  * required.  If the block is not in the cache pass the read request
2928  * on to the spa with a substitute callback function, so that the
2929  * requested block will be added to the cache.
2930  *
2931  * If a read request arrives for a block that has a read in-progress,
```



```
2932  * either wait for the in-progress read to complete (and return the
2933  * results); or, if this is a read with a "done" func, add a record
2934  * to the read to invoke the "done" func when the read completes,
2935  * and return; or just return.
2936  *
2937  * arc_read_done() will invoke all the requested "done" functions
2938  * for readers of this block.
2939  */
2940 int
2941 arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
2942     void *private, zio_priority_t priority, int zio_flags, uint32_t *arc_flags,
2943     const zbookmark_phys_t *zb)
2944 {
2945         arc_buf_hdr_t *hdr = NULL;
2946         arc_buf_t *buf = NULL;
2947         kmutex_t *hash_lock = NULL;
2948         zio_t *rzio;
2949         uint64_t guid = spa_load_guid(spa);

2951         ASSERT(!BP_IS_EMBEDDED(bp) ||
2952             BPE_GET_ETYPE(bp) == BP_EMBEDDED_TYPE_DATA);

2954 top:
2955         if (!BP_IS_EMBEDDED(bp)) {
2956                 /*
2957                  * Embedded BP's have no DVA and require no I/O to "read".
2958                  * Create an anonymous arc buf to back it.
2959                  */
2960                 hdr = buf_hash_find(guid, bp, &hash_lock);
2961         }

2963         if (hdr != NULL && hdr->b_datacnt > 0) {

2965                 *arc_flags |= ARC_CACHED;

2967                 if (HDR_IO_IN_PROGRESS(hdr)) {

2969                         if (*arc_flags & ARC_WAIT) {
2970                                 cv_wait(&hdr->b_cv, hash_lock);
2971                                 mutex_exit(hash_lock);
2972                                 goto top;
2973                         }
2974                         ASSERT(*arc_flags & ARC_NOWAIT);

2976                         if (done) {
2977                                 arc_callback_t  *acb = NULL;

2979                                 acb = kmem_zalloc(sizeof (arc_callback_t),
2980                                     KM_SLEEP);
2981                                 acb->acb_done = done;
2982                                 acb->acb_private = private;
2983                                 if (pio != NULL)
2984                                         acb->acb_zio_dummy = zio_null(pio,
2985                                             spa, NULL, NULL, NULL, zio_flags);

2987                                 ASSERT(acb->acb_done != NULL);
2988                                 acb->acb_next = hdr->b_acb;
2989                                 hdr->b_acb = acb;
2990                                 add_reference(hdr, hash_lock, private);
2991                                 mutex_exit(hash_lock);
2992                                 return (0);
2993                         }
2994                         mutex_exit(hash_lock);
2995                         return (0);
2996                 }
```

```
2998                ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);

3000                if (done) {
3001                        add_reference(hdr, hash_lock, private);
3002                        /*
3003                         * If this block is already in use, create a new
3004                         * copy of the data so that we will be guaranteed
3005                         * that arc_release() will always succeed.
3006                         */
3007                        buf = hdr->b_buf;
3008                        ASSERT(buf);
3009                        ASSERT(buf->b_data);
3010                        if (HDR_BUF_AVAILABLE(hdr)) {
3011                                ASSERT(buf->b_efunc == NULL);
3012                                hdr->b_flags &= ~ARC_BUF_AVAILABLE;
3013                        } else {
3014                                buf = arc_buf_clone(buf);
3015                        }

3017                } else if (*arc_flags & ARC_PREFETCH &&
3018                    refcount_count(&hdr->b_refcnt) == 0) {
3019                        hdr->b_flags |= ARC_PREFETCH;
3020                }
3021                DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
3022                arc_access(hdr, hash_lock);
3023                if (*arc_flags & ARC_L2CACHE)
3024                        hdr->b_flags |= ARC_L2CACHE;
3025                if (*arc_flags & ARC_L2COMPRESS)
3026                        hdr->b_flags |= ARC_L2COMPRESS;
3027                mutex_exit(hash_lock);
3028                ARCSTAT_BUMP(arcstat_hits);
3029                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
3030                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
3031                    data, metadata, hits);

3033                if (done)
3034                        done(NULL, buf, private);
3035        } else {
3036                uint64_t size = BP_GET_LSIZE(bp);
3037                arc_callback_t *acb;
3038                vdev_t *vd = NULL;
3039                uint64_t addr = 0;
3040                boolean_t devw = B_FALSE;
3041                enum zio_compress b_compress = ZIO_COMPRESS_OFF;
3042                uint64_t b_asize = 0;

3044                if (hdr == NULL) {
3045                        /* this block is not in the cache */
3046                        arc_buf_hdr_t *exists = NULL;
3047                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
3048                        buf = arc_buf_alloc(spa, size, private, type);
3049                        hdr = buf->b_hdr;
3050                        if (!BP_IS_EMBEDDED(bp)) {
3051                                hdr->b_dva = *BP_IDENTITY(bp);
3052                                hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
3053                                hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
3054                                exists = buf_hash_insert(hdr, &hash_lock);
3055                        }
3056                        if (exists != NULL) {
3057                                /* somebody beat us to the hash insert */
3058                                mutex_exit(hash_lock);
3059                                buf_discard_identity(hdr);
3060                                (void) arc_buf_remove_ref(buf, private);
3061                                goto top; /* restart the IO request */
3062                        }
3063                        /* if this is a prefetch, we don't have a reference */
```



```
3064                        if (*arc_flags & ARC_PREFETCH) {
3065                                (void) remove_reference(hdr, hash_lock,
3066                                    private);
3067                                hdr->b_flags |= ARC_PREFETCH;
3068                        }
3069                        if (*arc_flags & ARC_L2CACHE)
3070                                hdr->b_flags |= ARC_L2CACHE;
3071                        if (*arc_flags & ARC_L2COMPRESS)
3072                                hdr->b_flags |= ARC_L2COMPRESS;
3073                        if (BP_GET_LEVEL(bp) > 0)
3074                                hdr->b_flags |= ARC_INDIRECT;
3075                } else {
3076                        /* this block is in the ghost cache */
3077                        ASSERT(GHOST_STATE(hdr->b_state));
3078                        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
3079                        ASSERT0(refcount_count(&hdr->b_refcnt));
3080                        ASSERT(hdr->b_buf == NULL);

3082                        /* if this is a prefetch, we don't have a reference */
3083                        if (*arc_flags & ARC_PREFETCH)
3084                                hdr->b_flags |= ARC_PREFETCH;
3085                        else
3086                                add_reference(hdr, hash_lock, private);
3087                        if (*arc_flags & ARC_L2CACHE)
3088                                hdr->b_flags |= ARC_L2CACHE;
3089                        if (*arc_flags & ARC_L2COMPRESS)
3090                                hdr->b_flags |= ARC_L2COMPRESS;
3091                        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
3092                        buf->b_hdr = hdr;
3093                        buf->b_data = NULL;
3094                        buf->b_efunc = NULL;
3095                        buf->b_private = NULL;
3096                        buf->b_next = NULL;
3097                        hdr->b_buf = buf;
3098                        ASSERT(hdr->b_datacnt == 0);
3099                        hdr->b_datacnt = 1;
3100                        arc_get_data_buf(buf);
3101                        arc_access(hdr, hash_lock);
3102                }

3104                ASSERT(!GHOST_STATE(hdr->b_state));

3106                acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
3107                acb->acb_done = done;
3108                acb->acb_private = private;

3110                ASSERT(hdr->b_acb == NULL);
3111                hdr->b_acb = acb;
3112                hdr->b_flags |= ARC_IO_IN_PROGRESS;

3114                if (hdr->b_l2hdr != NULL &&
3115                    (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
3116                        devw = hdr->b_l2hdr->b_dev->l2ad_writing;
3117                        addr = hdr->b_l2hdr->b_daddr;
3118                        b_compress = hdr->b_l2hdr->b_compress;
3119                        b_asize = hdr->b_l2hdr->b_asize;
3120                        /*
3121                         * Lock out device removal.
3122                         */
3123                        if (vdev_is_dead(vd) ||
3124                            !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
3125                                vd = NULL;
3126                }

3128                if (hash_lock != NULL)
3129                        mutex_exit(hash_lock);
```

```
3131                 /*
3132                  * At this point, we have a level 1 cache miss.  Try again in
3133                  * L2ARC if possible.
3134                  */
3135                 ASSERT3U(hdr->b_size, ==, size);
3136                 DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
3137                     uint64_t, size, zbookmark_phys_t *, zb);
3138                 ARCSTAT_BUMP(arcstat_misses);
3139                 ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
3140                     demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
3141                     data, metadata, misses);
3142
3143                 if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
3144                         /*
3145                          * Read from the L2ARC if the following are true:
3146                          * 1. The L2ARC vdev was previously cached.
3147                          * 2. This buffer still has L2ARC metadata.
3148                          * 3. This buffer isn't currently writing to the L2ARC.
3149                          * 4. The L2ARC entry wasn't evicted, which may
3150                          *    also have invalidated the vdev.
3151                          * 5. This isn't prefetch and l2arc_noprefetch is set.
3152                          */
3153                         if (hdr->b_l2hdr != NULL &&
3154                             !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
3155                             !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
3156                                 l2arc_read_callback_t *cb;
3157
3158                                 DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
3159                                 ARCSTAT_BUMP(arcstat_l2_hits);
3160
3161                                 cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
3162                                     KM_SLEEP);
3163                                 cb->l2rcb_buf = buf;
3164                                 cb->l2rcb_spa = spa;
3165                                 cb->l2rcb_bp = *bp;
3166                                 cb->l2rcb_zb = *zb;
3167                                 cb->l2rcb_flags = zio_flags;
3168                                 cb->l2rcb_compress = b_compress;
3169
3170                                 ASSERT(addr >= VDEV_LABEL_START_SIZE &&
3171                                     addr + size < vd->vdev_psize -
3172                                     VDEV_LABEL_END_SIZE);
3173
3174                                 /*
3175                                  * l2arc read.  The SCL_L2ARC lock will be
3176                                  * released by l2arc_read_done().
3177                                  * Issue a null zio if the underlying buffer
3178                                  * was squashed to zero size by compression.
3179                                  */
3180                                 if (b_compress == ZIO_COMPRESS_EMPTY) {
3181                                         rzio = zio_null(pio, spa, vd,
3182                                             l2arc_read_done, cb,
3183                                             zio_flags | ZIO_FLAG_DONT_CACHE |
3184                                             ZIO_FLAG_CANFAIL |
3185                                             ZIO_FLAG_DONT_PROPAGATE |
3186                                             ZIO_FLAG_DONT_RETRY);
3187                                 } else {
3188                                         rzio = zio_read_phys(pio, vd, addr,
3189                                             b_asize, buf->b_data,
3190                                             ZIO_CHECKSUM_OFF,
3191                                             l2arc_read_done, cb, priority,
3192                                             zio_flags | ZIO_FLAG_DONT_CACHE |
3193                                             ZIO_FLAG_CANFAIL |
3194                                             ZIO_FLAG_DONT_PROPAGATE |
3195                                             ZIO_FLAG_DONT_RETRY, B_FALSE);
```



```
3196                                 }
3197                                 DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
3198                                     zio_t *, rzio);
3199                                 ARCSTAT_INCR(arcstat_l2_read_bytes, b_asize);
3200
3201                                 if (*arc_flags & ARC_NOWAIT) {
3202                                         zio_nowait(rzio);
3203                                         return (0);
3204                                 }
3205
3206                                 ASSERT(*arc_flags & ARC_WAIT);
3207                                 if (zio_wait(rzio) == 0)
3208                                         return (0);
3209
3210                                 /* l2arc read error; goto zio_read() */
3211                         } else {
3212                                 DTRACE_PROBE1(l2arc__miss,
3213                                     arc_buf_hdr_t *, hdr);
3214                                 ARCSTAT_BUMP(arcstat_l2_misses);
3215                                 if (HDR_L2_WRITING(hdr))
3216                                         ARCSTAT_BUMP(arcstat_l2_rw_clash);
3217                                 spa_config_exit(spa, SCL_L2ARC, vd);
3218                         }
3219                 } else {
3220                         if (vd != NULL)
3221                                 spa_config_exit(spa, SCL_L2ARC, vd);
3222                         if (l2arc_ndev != 0) {
3223                                 DTRACE_PROBE1(l2arc__miss,
3224                                     arc_buf_hdr_t *, hdr);
3225                                 ARCSTAT_BUMP(arcstat_l2_misses);
3226                         }
3227                 }
3228
3229                 rzio = zio_read(pio, spa, bp, buf->b_data, size,
3230                     arc_read_done, buf, priority, zio_flags, zb);
3231
3232                 if (*arc_flags & ARC_WAIT)
3233                         return (zio_wait(rzio));
3234
3235                 ASSERT(*arc_flags & ARC_NOWAIT);
3236                 zio_nowait(rzio);
3237         }
3238         return (0);
3239 }
3240
3241 void
3242 arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private)
3243 {
3244         ASSERT(buf->b_hdr != NULL);
3245         ASSERT(buf->b_hdr->b_state != arc_anon);
3246         ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt) || func == NULL);
3247         ASSERT(buf->b_efunc == NULL);
3248         ASSERT(!HDR_BUF_AVAILABLE(buf->b_hdr));
3249
3250         buf->b_efunc = func;
3251         buf->b_private = private;
3252 }
3253
3254 /*
3255  * Notify the arc that a block was freed, and thus will never be used again.
3256  */
3257 void
3258 arc_freed(spa_t *spa, const blkptr_t *bp)
3259 {
3260         arc_buf_hdr_t *hdr;
3261         kmutex_t *hash_lock;
```

```
3262         uint64_t guid = spa_load_guid(spa);

3264         ASSERT(!BP_IS_EMBEDDED(bp));

3266         hdr = buf_hash_find(guid, bp, &hash_lock);
3267         if (hdr == NULL)
3268                 return;
3269         if (HDR_BUF_AVAILABLE(hdr)) {
3270                 arc_buf_t *buf = hdr->b_buf;
3271                 add_reference(hdr, hash_lock, FTAG);
3272                 hdr->b_flags &= ~ARC_BUF_AVAILABLE;
3273                 mutex_exit(hash_lock);

3275                 arc_release(buf, FTAG);
3276                 (void) arc_buf_remove_ref(buf, FTAG);
3277         } else {
3278                 mutex_exit(hash_lock);
3279         }

3281 }

3283 /*
3284  * Clear the user eviction callback set by arc_set_callback(), first calling
3285  * it if it exists.  Because the presence of a callback keeps an arc_buf cached
3286  * clearing the callback may result in the arc_buf being destroyed.  However,
3287  * it will not result in the *last* arc_buf being destroyed, hence the data
3288  * will remain cached in the ARC. We make a copy of the arc buffer here so
3289  * that we can process the callback without holding any locks.
3290  *
3291  * It's possible that the callback is already in the process of being cleared
3292  * by another thread.  In this case we can not clear the callback.
3293  *
3294  * Returns B_TRUE if the callback was successfully called and cleared.
3295  */
3296 boolean_t
3297 arc_clear_callback(arc_buf_t *buf)
3298 {
3299         arc_buf_hdr_t *hdr;
3300         kmutex_t *hash_lock;
3301         arc_evict_func_t *efunc = buf->b_efunc;
3302         void *private = buf->b_private;

3304         mutex_enter(&buf->b_evict_lock);
3305         hdr = buf->b_hdr;
3306         if (hdr == NULL) {
3307                 /*
3308                  * We are in arc_do_user_evicts().
3309                  */
3310                 ASSERT(buf->b_data == NULL);
3311                 mutex_exit(&buf->b_evict_lock);
3312                 return (B_FALSE);
3313         } else if (buf->b_data == NULL) {
3314                 /*
3315                  * We are on the eviction list; process this buffer now
3316                  * but let arc_do_user_evicts() do the reaping.
3317                  */
3318                 buf->b_efunc = NULL;
3319                 mutex_exit(&buf->b_evict_lock);
3320                 VERIFY0(efunc(private));
3321                 return (B_TRUE);
3322         }
3323         hash_lock = HDR_LOCK(hdr);
3324         mutex_enter(hash_lock);
3325         hdr = buf->b_hdr;
3326         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
```



```
3328         ASSERT3U(refcount_count(&hdr->b_refcnt), <, hdr->b_datacnt);
3329         ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);

3331         buf->b_efunc = NULL;
3332         buf->b_private = NULL;

3334         if (hdr->b_datacnt > 1) {
3335                 mutex_exit(&buf->b_evict_lock);
3336                 arc_buf_destroy(buf, FALSE, TRUE);
3337         } else {
3338                 ASSERT(buf == hdr->b_buf);
3339                 hdr->b_flags |= ARC_BUF_AVAILABLE;
3340                 mutex_exit(&buf->b_evict_lock);
3341         }

3343         mutex_exit(hash_lock);
3344         VERIFY0(efunc(private));
3345         return (B_TRUE);
3346 }

3348 /*
3349  * Release this buffer from the cache, making it an anonymous buffer.  This
3350  * must be done after a read and prior to modifying the buffer contents.
3351  * If the buffer has more than one reference, we must make
3352  * a new hdr for the buffer.
3353  */
3354 void
3355 arc_release(arc_buf_t *buf, void *tag)
3356 {
3357         arc_buf_hdr_t *hdr;
3358         kmutex_t *hash_lock = NULL;
3359         l2arc_buf_hdr_t *l2hdr;
3360         uint64_t buf_size;

3362         /*
3363          * It would be nice to assert that if it's DMU metadata (level >
3364          * 0 || it's the dnode file), then it must be syncing context.
3365          * But we don't know that information at this level.
3366          */

3368         mutex_enter(&buf->b_evict_lock);
3369         hdr = buf->b_hdr;

3371         /* this buffer is not on any list */
3372         ASSERT(refcount_count(&hdr->b_refcnt) > 0);

3374         if (hdr->b_state == arc_anon) {
3375                 /* this buffer is already released */
3376                 ASSERT(buf->b_efunc == NULL);
3377         } else {
3378                 hash_lock = HDR_LOCK(hdr);
3379                 mutex_enter(hash_lock);
3380                 hdr = buf->b_hdr;
3381                 ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
3382         }

3384         l2hdr = hdr->b_l2hdr;
3385         if (l2hdr) {
3386                 mutex_enter(&l2arc_buflist_mtx);
3387                 arc_buf_l2_cdata_free(hdr);
3388 #endif /* ! codereview */
3389                 hdr->b_l2hdr = NULL;
3390                 list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
3391         }
3392         buf_size = hdr->b_size;
```

```
3394         /*
3395          * Do we have more than one buf?
3396          */
3397         if (hdr->b_datacnt > 1) {
3398                 arc_buf_hdr_t *nhdr;
3399                 arc_buf_t **bufp;
3400                 uint64_t blksz = hdr->b_size;
3401                 uint64_t spa = hdr->b_spa;
3402                 arc_buf_contents_t type = hdr->b_type;
3403                 uint32_t flags = hdr->b_flags;

3405                 ASSERT(hdr->b_buf != buf || buf->b_next != NULL);
3406                 /*
3407                  * Pull the data off of this hdr and attach it to
3408                  * a new anonymous hdr.
3409                  */
3410                 (void) remove_reference(hdr, hash_lock, tag);
3411                 bufp = &hdr->b_buf;
3412                 while (*bufp != buf)
3413                         bufp = &(*bufp)->b_next;
3414                 *bufp = buf->b_next;
3415                 buf->b_next = NULL;

3417                 ASSERT3U(hdr->b_state->arcs_size, >=, hdr->b_size);
3418                 atomic_add_64(&hdr->b_state->arcs_size, -hdr->b_size);
3419                 if (refcount_is_zero(&hdr->b_refcnt)) {
3420                         uint64_t *size = &hdr->b_state->arcs_lsize[hdr->b_type];
3421                         ASSERT3U(*size, >=, hdr->b_size);
3422                         atomic_add_64(size, -hdr->b_size);
3423                 }

3425                 /*
3426                  * We're releasing a duplicate user data buffer, update
3427                  * our statistics accordingly.
3428                  */
3429                 if (hdr->b_type == ARC_BUFC_DATA) {
3430                         ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
3431                         ARCSTAT_INCR(arcstat_duplicate_buffers_size,
3432                             -hdr->b_size);
3433                 }
3434                 hdr->b_datacnt -= 1;
3435                 arc_cksum_verify(buf);
3436                 arc_buf_unwatch(buf);

3438                 mutex_exit(hash_lock);

3440                 nhdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
3441                 nhdr->b_size = blksz;
3442                 nhdr->b_spa = spa;
3443                 nhdr->b_type = type;
3444                 nhdr->b_buf = buf;
3445                 nhdr->b_state = arc_anon;
3446                 nhdr->b_arc_access = 0;
3447                 nhdr->b_flags = flags & ARC_L2_WRITING;
3448                 nhdr->b_l2hdr = NULL;
3449                 nhdr->b_datacnt = 1;
3450                 nhdr->b_freeze_cksum = NULL;
3451                 (void) refcount_add(&nhdr->b_refcnt, tag);
3452                 buf->b_hdr = nhdr;
3453                 mutex_exit(&buf->b_evict_lock);
3454                 atomic_add_64(&arc_anon->arcs_size, blksz);
3455         } else {
3456                 mutex_exit(&buf->b_evict_lock);
3457                 ASSERT(refcount_count(&hdr->b_refcnt) == 1);
3458                 ASSERT(!list_link_active(&hdr->b_arc_node));
3459                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
```



```
3460                 if (hdr->b_state != arc_anon)
3461                         arc_change_state(arc_anon, hdr, hash_lock);
3462                 hdr->b_arc_access = 0;
3463                 if (hash_lock)
3464                         mutex_exit(hash_lock);

3466                 buf_discard_identity(hdr);
3467                 arc_buf_thaw(buf);
3468         }
3469         buf->b_efunc = NULL;
3470         buf->b_private = NULL;

3472         if (l2hdr) {
3473                 ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
3474                 vdev_space_update(l2hdr->b_dev->l2ad_vdev,
3475                     -l2hdr->b_asize, 0, 0);
3476                 kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
3477                 ARCSTAT_INCR(arcstat_l2_size, -buf_size);
3478                 mutex_exit(&l2arc_buflist_mtx);
3479         }
3480 }

3482 int
3483 arc_released(arc_buf_t *buf)
3484 {
3485         int released;

3487         mutex_enter(&buf->b_evict_lock);
3488         released = (buf->b_data != NULL && buf->b_hdr->b_state == arc_anon);
3489         mutex_exit(&buf->b_evict_lock);
3490         return (released);
3491 }

3493 #ifdef ZFS_DEBUG
3494 int
3495 arc_referenced(arc_buf_t *buf)
3496 {
3497         int referenced;

3499         mutex_enter(&buf->b_evict_lock);
3500         referenced = (refcount_count(&buf->b_hdr->b_refcnt));
3501         mutex_exit(&buf->b_evict_lock);
3502         return (referenced);
3503 }
3504 #endif

3506 static void
3507 arc_write_ready(zio_t *zio)
3508 {
3509         arc_write_callback_t *callback = zio->io_private;
3510         arc_buf_t *buf = callback->awcb_buf;
3511         arc_buf_hdr_t *hdr = buf->b_hdr;

3513         ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt));
3514         callback->awcb_ready(zio, buf, callback->awcb_private);

3516         /*
3517          * If the IO is already in progress, then this is a re-write
3518          * attempt, so we need to thaw and re-compute the cksum.
3519          * It is the responsibility of the callback to handle the
3520          * accounting for any re-write attempt.
3521          */
3522         if (HDR_IO_IN_PROGRESS(hdr)) {
3523                 mutex_enter(&hdr->b_freeze_lock);
3524                 if (hdr->b_freeze_cksum != NULL) {
3525                         kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
```

```
3526                        hdr->b_freeze_cksum = NULL;
3527                }
3528                mutex_exit(&hdr->b_freeze_lock);
3529        }
3530        arc_cksum_compute(buf, B_FALSE);
3531        hdr->b_flags |= ARC_IO_IN_PROGRESS;
3532 }

3534 /*
3535  * The SPA calls this callback for each physical write that happens on behalf
3536  * of a logical write.  See the comment in dbuf_write_physdone() for details.
3537  */
3538 static void
3539 arc_write_physdone(zio_t *zio)
3540 {
3541        arc_write_callback_t *cb = zio->io_private;
3542        if (cb->awcb_physdone != NULL)
3543                cb->awcb_physdone(zio, cb->awcb_buf, cb->awcb_private);
3544 }

3546 static void
3547 arc_write_done(zio_t *zio)
3548 {
3549        arc_write_callback_t *callback = zio->io_private;
3550        arc_buf_t *buf = callback->awcb_buf;
3551        arc_buf_hdr_t *hdr = buf->b_hdr;

3553        ASSERT(hdr->b_acb == NULL);

3555        if (zio->io_error == 0) {
3556                if (BP_IS_HOLE(zio->io_bp) || BP_IS_EMBEDDED(zio->io_bp)) {
3557                        buf_discard_identity(hdr);
3558                } else {
3559                        hdr->b_dva = *BP_IDENTITY(zio->io_bp);
3560                        hdr->b_birth = BP_PHYSICAL_BIRTH(zio->io_bp);
3561                        hdr->b_cksum0 = zio->io_bp->blk_cksum.zc_word[0];
3562                }
3563        } else {
3564                ASSERT(BUF_EMPTY(hdr));
3565        }

3567        /*
3568         * If the block to be written was all-zero or compressed enough to be
3569         * embedded in the BP, no write was performed so there will be no
3570         * dva/birth/checksum.  The buffer must therefore remain anonymous
3571         * (and uncached).
3572         */
3573        if (!BUF_EMPTY(hdr)) {
3574                arc_buf_hdr_t *exists;
3575                kmutex_t *hash_lock;

3577                ASSERT(zio->io_error == 0);

3579                arc_cksum_verify(buf);

3581                exists = buf_hash_insert(hdr, &hash_lock);
3582                if (exists) {
3583                        /*
3584                         * This can only happen if we overwrite for
3585                         * sync-to-convergence, because we remove
3586                         * buffers from the hash table when we arc_free().
3587                         */
3588                        if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
3589                                if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
3590                                        panic("bad overwrite, hdr=%p exists=%p",
3591                                            (void *)hdr, (void *)exists);
```



```
3592                                ASSERT(refcount_is_zero(&exists->b_refcnt));
3593                                arc_change_state(arc_anon, exists, hash_lock);
3594                                mutex_exit(hash_lock);
3595                                arc_hdr_destroy(exists);
3596                                exists = buf_hash_insert(hdr, &hash_lock);
3597                                ASSERT3P(exists, ==, NULL);
3598                        } else if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
3599                                /* nopwrite */
3600                                ASSERT(zio->io_prop.zp_nopwrite);
3601                                if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
3602                                        panic("bad nopwrite, hdr=%p exists=%p",
3603                                            (void *)hdr, (void *)exists);
3604                        } else {
3605                                /* Dedup */
3606                                ASSERT(hdr->b_datacnt == 1);
3607                                ASSERT(hdr->b_state == arc_anon);
3608                                ASSERT(BP_GET_DEDUP(zio->io_bp));
3609                                ASSERT(BP_GET_LEVEL(zio->io_bp) == 0);
3610                        }
3611                }
3612                hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
3613                /* if it's not anon, we are doing a scrub */
3614                if (!exists && hdr->b_state == arc_anon)
3615                        arc_access(hdr, hash_lock);
3616                mutex_exit(hash_lock);
3617        } else {
3618                hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
3619        }

3621        ASSERT(!refcount_is_zero(&hdr->b_refcnt));
3622        callback->awcb_done(zio, buf, callback->awcb_private);

3624        kmem_free(callback, sizeof (arc_write_callback_t));
3625 }

3627 zio_t *
3628 arc_write(zio_t *pio, spa_t *spa, uint64_t txg,
3629     blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, boolean_t l2arc_compress,
3630     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *physdone,
3631     arc_done_func_t *done, void *private, zio_priority_t priority,
3632     int zio_flags, const zbookmark_phys_t *zb)
3633 {
3634        arc_buf_hdr_t *hdr = buf->b_hdr;
3635        arc_write_callback_t *callback;
3636        zio_t *zio;

3638        ASSERT(ready != NULL);
3639        ASSERT(done != NULL);
3640        ASSERT(!HDR_IO_ERROR(hdr));
3641        ASSERT((hdr->b_flags & ARC_IO_IN_PROGRESS) == 0);
3642        ASSERT(hdr->b_acb == NULL);
3643        if (l2arc)
3644                hdr->b_flags |= ARC_L2CACHE;
3645        if (l2arc_compress)
3646                hdr->b_flags |= ARC_L2COMPRESS;
3647        callback = kmem_zalloc(sizeof (arc_write_callback_t), KM_SLEEP);
3648        callback->awcb_ready = ready;
3649        callback->awcb_physdone = physdone;
3650        callback->awcb_done = done;
3651        callback->awcb_private = private;
3652        callback->awcb_buf = buf;

3654        zio = zio_write(pio, spa, txg, bp, buf->b_data, hdr->b_size, zp,
3655            arc_write_ready, arc_write_physdone, arc_write_done, callback,
3656            priority, zio_flags, zb);
```

```
3658         return (zio);
3659 }

3661 static int
3662 arc_memory_throttle(uint64_t reserve, uint64_t txg)
3663 {
3664 #ifdef _KERNEL
3665         uint64_t available_memory = ptob(freemem);
3666         static uint64_t page_load = 0;
3667         static uint64_t last_txg = 0;

3669 #if defined(__i386)
3670         available_memory =
3671             MIN(available_memory, vmem_size(heap_arena, VMEM_FREE));
3672 #endif

3674         if (freemem > physmem * arc_lotsfree_percent / 100)
3675                 return (0);

3677         if (txg > last_txg) {
3678                 last_txg = txg;
3679                 page_load = 0;
3680         }
3681         /*
3682          * If we are in pageout, we know that memory is already tight,
3683          * the arc is already going to be evicting, so we just want to
3684          * continue to let page writes occur as quickly as possible.
3685          */
3686         if (curproc == proc_pageout) {
3687                 if (page_load > MAX(ptob(minfree), available_memory) / 4)
3688                         return (SET_ERROR(ERESTART));
3689                 /* Note: reserve is inflated, so we deflate */
3690                 page_load += reserve / 8;
3691                 return (0);
3692         } else if (page_load > 0 && arc_reclaim_needed()) {
3693                 /* memory is low, delay before restarting */
3694                 ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
3695                 return (SET_ERROR(EAGAIN));
3696         }
3697         page_load = 0;
3698 #endif
3699         return (0);
3700 }

3702 void
3703 arc_tempreserve_clear(uint64_t reserve)
3704 {
3705         atomic_add_64(&arc_tempreserve, -reserve);
3706         ASSERT((int64_t)arc_tempreserve >= 0);
3707 }

3709 int
3710 arc_tempreserve_space(uint64_t reserve, uint64_t txg)
3711 {
3712         int error;
3713         uint64_t anon_size;

3715         if (reserve > arc_c/4 && !arc_no_grow)
3716                 arc_c = MIN(arc_c_max, reserve * 4);
3717         if (reserve > arc_c)
3718                 return (SET_ERROR(ENOMEM));

3720         /*
3721          * Don't count loaned bufs as in flight dirty data to prevent long
3722          * network delays from blocking transactions that are ready to be
3723          * assigned to a txg.
```



```
3724          */
3725         anon_size = MAX((int64_t)(arc_anon->arcs_size - arc_loaned_bytes), 0);

3727         /*
3728          * Writes will, almost always, require additional memory allocations
3729          * in order to compress/encrypt/etc the data.  We therefore need to
3730          * make sure that there is sufficient available memory for this.
3731          */
3732         error = arc_memory_throttle(reserve, txg);
3733         if (error != 0)
3734                 return (error);

3736         /*
3737          * Throttle writes when the amount of dirty data in the cache
3738          * gets too large.  We try to keep the cache less than half full
3739          * of dirty blocks so that our sync times don't grow too large.
3740          * Note: if two requests come in concurrently, we might let them
3741          * both succeed, when one of them should fail.  Not a huge deal.
3742          */

3744         if (reserve + arc_tempreserve + anon_size > arc_c / 2 &&
3745             anon_size > arc_c / 4) {
3746                 dprintf("failing, arc_tempreserve=%lluK anon_meta=%lluK "
3747                     "anon_data=%lluK tempreserve=%lluK arc_c=%lluK\n",
3748                     arc_tempreserve>>10,
3749                     arc_anon->arcs_lsize[ARC_BUFC_METADATA]>>10,
3750                     arc_anon->arcs_lsize[ARC_BUFC_DATA]>>10,
3751                     reserve>>10, arc_c>>10);
3752                 return (SET_ERROR(ERESTART));
3753         }
3754         atomic_add_64(&arc_tempreserve, reserve);
3755         return (0);
3756 }

3758 void
3759 arc_init(void)
3760 {
3761         mutex_init(&arc_reclaim_thr_lock, NULL, MUTEX_DEFAULT, NULL);
3762         cv_init(&arc_reclaim_thr_cv, NULL, CV_DEFAULT, NULL);

3764         /* Convert seconds to clock ticks */
3765         arc_min_prefetch_lifespan = 1 * hz;

3767         /* Start out with 1/8 of all memory */
3768         arc_c = physmem * PAGESIZE / 8;

3770 #ifdef _KERNEL
3771         /*
3772          * On architectures where the physical memory can be larger
3773          * than the addressable space (intel in 32-bit mode), we may
3774          * need to limit the cache to 1/8 of VM size.
3775          */
3776         arc_c = MIN(arc_c, vmem_size(heap_arena, VMEM_ALLOC | VMEM_FREE) / 8);
3777 #endif

3779         /* set min cache to 1/32 of all memory, or 64MB, whichever is more */
3780         arc_c_min = MAX(arc_c / 4, 64<<20);
3781         /* set max to 3/4 of all memory, or all but 1GB, whichever is more */
3782         if (arc_c * 8 >= 1<<30)
3783                 arc_c_max = (arc_c * 8) - (1<<30);
3784         else
3785                 arc_c_max = arc_c_min;
3786         arc_c_max = MAX(arc_c * 6, arc_c_max);

3788         /*
3789          * Allow the tunables to override our calculations if they are
```

```
3790          * reasonable (ie. over 64MB)
3791          */
3792         if (zfs_arc_max > 64<<20 && zfs_arc_max < physmem * PAGESIZE)
3793                 arc_c_max = zfs_arc_max;
3794         if (zfs_arc_min > 64<<20 && zfs_arc_min <= arc_c_max)
3795                 arc_c_min = zfs_arc_min;

3797         arc_c = arc_c_max;
3798         arc_p = (arc_c >> 1);

3800         /* limit meta-data to 1/4 of the arc capacity */
3801         arc_meta_limit = arc_c_max / 4;

3803         /* Allow the tunable to override if it is reasonable */
3804         if (zfs_arc_meta_limit > 0 && zfs_arc_meta_limit <= arc_c_max)
3805                 arc_meta_limit = zfs_arc_meta_limit;

3807         if (arc_c_min < arc_meta_limit / 2 && zfs_arc_min == 0)
3808                 arc_c_min = arc_meta_limit / 2;

3810         if (zfs_arc_grow_retry > 0)
3811                 arc_grow_retry = zfs_arc_grow_retry;

3813         if (zfs_arc_shrink_shift > 0)
3814                 arc_shrink_shift = zfs_arc_shrink_shift;

3816         if (zfs_arc_p_min_shift > 0)
3817                 arc_p_min_shift = zfs_arc_p_min_shift;

3819         /* if kmem_flags are set, lets try to use less memory */
3820         if (kmem_debugging())
3821                 arc_c = arc_c / 2;
3822         if (arc_c < arc_c_min)
3823                 arc_c = arc_c_min;

3825         arc_anon = &ARC_anon;
3826         arc_mru = &ARC_mru;
3827         arc_mru_ghost = &ARC_mru_ghost;
3828         arc_mfu = &ARC_mfu;
3829         arc_mfu_ghost = &ARC_mfu_ghost;
3830         arc_l2c_only = &ARC_l2c_only;
3831         arc_size = 0;

3833         mutex_init(&arc_anon->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3834         mutex_init(&arc_mru->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3835         mutex_init(&arc_mru_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3836         mutex_init(&arc_mfu->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3837         mutex_init(&arc_mfu_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3838         mutex_init(&arc_l2c_only->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);

3840         list_create(&arc_mru->arcs_list[ARC_BUFC_METADATA],
3841             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3842         list_create(&arc_mru->arcs_list[ARC_BUFC_DATA],
3843             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3844         list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA],
3845             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3846         list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA],
3847             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3848         list_create(&arc_mfu->arcs_list[ARC_BUFC_METADATA],
3849             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3850         list_create(&arc_mfu->arcs_list[ARC_BUFC_DATA],
3851             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3852         list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA],
3853             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3854         list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA],
3855             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
```



```
3856         list_create(&arc_l2c_only->arcs_list[ARC_BUFC_METADATA],
3857             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3858         list_create(&arc_l2c_only->arcs_list[ARC_BUFC_DATA],
3859             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));

3861         buf_init();

3863         arc_thread_exit = 0;
3864         arc_eviction_list = NULL;
3865         mutex_init(&arc_eviction_mtx, NULL, MUTEX_DEFAULT, NULL);
3866         bzero(&arc_eviction_hdr, sizeof (arc_buf_hdr_t));

3868         arc_ksp = kstat_create("zfs", 0, "arcstats", "misc", KSTAT_TYPE_NAMED,
3869             sizeof (arc_stats) / sizeof (kstat_named_t), KSTAT_FLAG_VIRTUAL);

3871         if (arc_ksp != NULL) {
3872                 arc_ksp->ks_data = &arc_stats;
3873                 kstat_install(arc_ksp);
3874         }

3876         (void) thread_create(NULL, 0, arc_reclaim_thread, NULL, 0, &p0,
3877             TS_RUN, minclsyspri);

3879         arc_dead = FALSE;
3880         arc_warm = B_FALSE;

3882         /*
3883          * Calculate maximum amount of dirty data per pool.
3884          *
3885          * If it has been set by /etc/system, take that.
3886          * Otherwise, use a percentage of physical memory defined by
3887          * zfs_dirty_data_max_percent (default 10%) with a cap at
3888          * zfs_dirty_data_max_max (default 4GB).
3889          */
3890         if (zfs_dirty_data_max == 0) {
3891                 zfs_dirty_data_max = physmem * PAGESIZE *
3892                     zfs_dirty_data_max_percent / 100;
3893                 zfs_dirty_data_max = MIN(zfs_dirty_data_max,
3894                     zfs_dirty_data_max_max);
3895         }
3896 }

3898 void
3899 arc_fini(void)
3900 {
3901         mutex_enter(&arc_reclaim_thr_lock);
3902         arc_thread_exit = 1;
3903         while (arc_thread_exit != 0)
3904                 cv_wait(&arc_reclaim_thr_cv, &arc_reclaim_thr_lock);
3905         mutex_exit(&arc_reclaim_thr_lock);

3907         arc_flush(NULL);

3909         arc_dead = TRUE;

3911         if (arc_ksp != NULL) {
3912                 kstat_delete(arc_ksp);
3913                 arc_ksp = NULL;
3914         }

3916         mutex_destroy(&arc_eviction_mtx);
3917         mutex_destroy(&arc_reclaim_thr_lock);
3918         cv_destroy(&arc_reclaim_thr_cv);

3920         list_destroy(&arc_mru->arcs_list[ARC_BUFC_METADATA]);
3921         list_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA]);
```

```
3922         list_destroy(&arc_mfu->arcs_list[ARC_BUFC_METADATA]);
3923         list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA]);
3924         list_destroy(&arc_mru->arcs_list[ARC_BUFC_DATA]);
3925         list_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA]);
3926         list_destroy(&arc_mfu->arcs_list[ARC_BUFC_DATA]);
3927         list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA]);

3929         mutex_destroy(&arc_anon->arcs_mtx);
3930         mutex_destroy(&arc_mru->arcs_mtx);
3931         mutex_destroy(&arc_mru_ghost->arcs_mtx);
3932         mutex_destroy(&arc_mfu->arcs_mtx);
3933         mutex_destroy(&arc_mfu_ghost->arcs_mtx);
3934         mutex_destroy(&arc_l2c_only->arcs_mtx);

3936         buf_fini();

3938         ASSERT(arc_loaned_bytes == 0);
3939 }

3941 /*
3942  * Level 2 ARC
3943  *
3944  * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
3945  * It uses dedicated storage devices to hold cached data, which are populated
3946  * using large infrequent writes.  The main role of this cache is to boost
3947  * the performance of random read workloads.  The intended L2ARC devices
3948  * include short-stroked disks, solid state disks, and other media with
3949  * substantially faster read latency than disk.
3950  *
3951  *                 +-----------------------+
3952  *                 |         ARC           |
3953  *                 +-----------------------+
3954  *                    |         ^     ^
3955  *                    |         |     |
3956  *      l2arc_feed_thread()    arc_read()
3957  *                    |         |     |
3958  *                    |  l2arc read   |
3959  *                    V         |     |
3960  *               +---------------+    |
3961  *               |     L2ARC     |    |
3962  *               +---------------+    |
3963  *                   |    ^           |
3964  *          l2arc_write() |           |
3965  *                   |    |           |
3966  *                   V    |           |
3967  *                 +-------+      +-------+
3968  *                 | vdev  |      | vdev  |
3969  *                 | cache |      | cache |
3970  *                 +-------+      +-------+
3971  *                 +=========+     .-----.
3972  *                 :  L2ARC  :    |-_____-|
3973  *                 : devices :    | Disks |
3974  *                 +=========+    `-_____-'
3975  *
3976  * Read requests are satisfied from the following sources, in order:
3977  *
3978  *      1) ARC
3979  *      2) vdev cache of L2ARC devices
3980  *      3) L2ARC devices
3981  *      4) vdev cache of disks
3982  *      5) disks
3983  *
3984  * Some L2ARC device types exhibit extremely slow write performance.
3985  * To accommodate for this there are some significant differences between
3986  * the L2ARC and traditional cache design:
3987  *
```



```
3988  * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
3989  * the ARC behave as usual, freeing buffers and placing headers on ghost
3990  * lists.  The ARC does not send buffers to the L2ARC during eviction as
3991  * this would add inflated write latencies for all ARC memory pressure.
3992  *
3993  * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
3994  * It does this by periodically scanning buffers from the eviction-end of
3995  * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
3996  * not already there. It scans until a headroom of buffers is satisfied,
3997  * which itself is a buffer for ARC eviction. If a compressible buffer is
3998  * found during scanning and selected for writing to an L2ARC device, we
3999  * temporarily boost scanning headroom during the next scan cycle to make
4000  * sure we adapt to compression effects (which might significantly reduce
4001  * the data volume we write to L2ARC). The thread that does this is
4002  * l2arc_feed_thread(), illustrated below; example sizes are included to
4003  * provide a better sense of ratio than this diagram:
4004  *
4005  *             head -->                        tail
4006  *              +---------------------+----------+
4007  *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
4008  *              +---------------------+----------+   |   o L2ARC eligible
4009  *      ARC_mru |:#::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
4010  *              +---------------------+----------+   |
4011  *                   15.9 Gbytes      ^ 32 Mbytes    |
4012  *                                 headroom          |
4013  *                                          l2arc_feed_thread()
4014  *                                                   |
4015  *                       l2arc write hand <--[oooo]--'
4016  *                                 |           8 Mbyte
4017  *                                 |          write max
4018  *                                 V
4019  *                +==============================+
4020  *      L2ARC dev |####|#|###|###|    |####| ... |
4021  *                +==============================+
4022  *                               32 Gbytes
4023  *
4024  * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
4025  * evicted, then the L2ARC has cached a buffer much sooner than it probably
4026  * needed to, potentially wasting L2ARC device bandwidth and storage.  It is
4027  * safe to say that this is an uncommon case, since buffers at the end of
4028  * the ARC lists have moved there due to inactivity.
4029  *
4030  * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
4031  * then the L2ARC simply misses copying some buffers.  This serves as a
4032  * pressure valve to prevent heavy read workloads from both stalling the ARC
4033  * with waits and clogging the L2ARC with writes.  This also helps prevent
4034  * the potential for the L2ARC to churn if it attempts to cache content too
4035  * quickly, such as during backups of the entire pool.
4036  *
4037  * 5. After system boot and before the ARC has filled main memory, there are
4038  * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
4039  * lists can remain mostly static.  Instead of searching from tail of these
4040  * lists as pictured, the l2arc_feed_thread() will search from the list heads
4041  * for eligible buffers, greatly increasing its chance of finding them.
4042  *
4043  * The L2ARC device write speed is also boosted during this time so that
4044  * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
4045  * there are no L2ARC reads, and no fear of degrading read performance
4046  * through increased writes.
4047  *
4048  * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
4049  * the vdev queue can aggregate them into larger and fewer writes.  Each
4050  * device is written to in a rotor fashion, sweeping writes through
4051  * available space then repeating.
4052  *
4053  * 7. The L2ARC does not store dirty content.  It never needs to flush
```

```
 * write buffers back to disk based storage.
 *
 * 8. If an ARC buffer is written (and dirtied) which also exists in the
 * L2ARC, the now stale L2ARC buffer is immediately dropped.
 *
 * The performance of the L2ARC can be tweaked by a number of tunables, which
 * may be necessary for different workloads:
 *
 *      l2arc_write_max         max write bytes per interval
 *      l2arc_write_boost       extra write bytes during device warmup
 *      l2arc_noprefetch        skip caching prefetched buffers
 *      l2arc_headroom          number of max device writes to precache
 *      l2arc_headroom_boost    when we find compressed buffers during ARC
 *                              scanning, we multiply headroom by this
 *                              percentage factor for the next scan cycle,
 *                              since more compressed buffers are likely to
 *                              be present
 *      l2arc_feed_secs         seconds between L2ARC writing
 *
 * Tunables may be removed or added as future performance improvements are
 * integrated, and also may become zpool properties.
 *
 * There are three key functions that control how the L2ARC warms up:
 *
 *      l2arc_write_eligible()  check if a buffer is eligible to cache
 *      l2arc_write_size()      calculate how much to write
 *      l2arc_write_interval()  calculate sleep delay between writes
 *
 * These three functions determine what to write, how much, and how quickly
 * to send writes.
 */

static boolean_t
l2arc_write_eligible(uint64_t spa_guid, arc_buf_hdr_t *ab)
{
        /*
         * A buffer is *not* eligible for the L2ARC if it:
         * 1. belongs to a different spa.
         * 2. is already cached on the L2ARC.
         * 3. has an I/O in progress (it may be an incomplete read).
         * 4. is flagged not eligible (zfs property).
         */
        if (ab->b_spa != spa_guid || ab->b_l2hdr != NULL ||
            HDR_IO_IN_PROGRESS(ab) || !HDR_L2CACHE(ab))
                return (B_FALSE);

        return (B_TRUE);
}

static uint64_t
l2arc_write_size(void)
{
        uint64_t size;

        /*
         * Make sure our globals have meaningful values in case the user
         * altered them.
         */
        size = l2arc_write_max;
        if (size == 0) {
                cmn_err(CE_NOTE, "Bad value for l2arc_write_max, value must "
                    "be greater than zero, resetting it to the default (%d)",
                    L2ARC_WRITE_SIZE);
                size = l2arc_write_max = L2ARC_WRITE_SIZE;
        }
```



```
        if (arc_warm == B_FALSE)
                size += l2arc_write_boost;

        return (size);

}

static clock_t
l2arc_write_interval(clock_t began, uint64_t wanted, uint64_t wrote)
{
        clock_t interval, next, now;

        /*
         * If the ARC lists are busy, increase our write rate; if the
         * lists are stale, idle back.  This is achieved by checking
         * how much we previously wrote - if it was more than half of
         * what we wanted, schedule the next write much sooner.
         */
        if (l2arc_feed_again && wrote > (wanted / 2))
                interval = (hz * l2arc_feed_min_ms) / 1000;
        else
                interval = hz * l2arc_feed_secs;

        now = ddi_get_lbolt();
        next = MAX(now, MIN(now + interval, began + interval));

        return (next);
}

static void
l2arc_hdr_stat_add(void)
{
        ARCSTAT_INCR(arcstat_l2_hdr_size, HDR_SIZE + L2HDR_SIZE);
        ARCSTAT_INCR(arcstat_hdr_size, -HDR_SIZE);
}

static void
l2arc_hdr_stat_remove(void)
{
        ARCSTAT_INCR(arcstat_l2_hdr_size, -(HDR_SIZE + L2HDR_SIZE));
        ARCSTAT_INCR(arcstat_hdr_size, HDR_SIZE);
}

/*
 * Cycle through L2ARC devices.  This is how L2ARC load balances.
 * If a device is returned, this also returns holding the spa config lock.
 */
static l2arc_dev_t *
l2arc_dev_get_next(void)
{
        l2arc_dev_t *first, *next = NULL;

        /*
         * Lock out the removal of spas (spa_namespace_lock), then removal
         * of cache devices (l2arc_dev_mtx).  Once a device has been selected,
         * both locks will be dropped and a spa config lock held instead.
         */
        mutex_enter(&spa_namespace_lock);
        mutex_enter(&l2arc_dev_mtx);

        /* if there are no vdevs, there is nothing to do */
        if (l2arc_ndev == 0)
                goto out;

        first = NULL;
        next = l2arc_dev_last;
```

```
4186         do {
4187                 /* loop around the list looking for a non-faulted vdev */
4188                 if (next == NULL) {
4189                         next = list_head(l2arc_dev_list);
4190                 } else {
4191                         next = list_next(l2arc_dev_list, next);
4192                         if (next == NULL)
4193                                 next = list_head(l2arc_dev_list);
4194                 }

4196                 /* if we have come back to the start, bail out */
4197                 if (first == NULL)
4198                         first = next;
4199                 else if (next == first)
4200                         break;

4202         } while (vdev_is_dead(next->l2ad_vdev));

4204         /* if we were unable to find any usable vdevs, return NULL */
4205         if (vdev_is_dead(next->l2ad_vdev))
4206                 next = NULL;

4208         l2arc_dev_last = next;

4210 out:
4211         mutex_exit(&l2arc_dev_mtx);

4213         /*
4214          * Grab the config lock to prevent the 'next' device from being
4215          * removed while we are writing to it.
4216          */
4217         if (next != NULL)
4218                 spa_config_enter(next->l2ad_spa, SCL_L2ARC, next, RW_READER);
4219         mutex_exit(&spa_namespace_lock);

4221         return (next);
4222 }

4224 /*
4225  * Free buffers that were tagged for destruction.
4226  */
4227 static void
4228 l2arc_do_free_on_write()
4229 {
4230         list_t *buflist;
4231         l2arc_data_free_t *df, *df_prev;

4233         mutex_enter(&l2arc_free_on_write_mtx);
4234         buflist = l2arc_free_on_write;

4236         for (df = list_tail(buflist); df; df = df_prev) {
4237                 df_prev = list_prev(buflist, df);
4238                 ASSERT(df->l2df_data != NULL);
4239                 ASSERT(df->l2df_func != NULL);
4240                 df->l2df_func(df->l2df_data, df->l2df_size);
4241                 list_remove(buflist, df);
4242                 kmem_free(df, sizeof (l2arc_data_free_t));
4243         }

4245         mutex_exit(&l2arc_free_on_write_mtx);
4246 }

4248 /*
4249  * A write to a cache device has completed.  Update all headers to allow
4250  * reads from these buffers to begin.
4251  */
```



```
4252 static void
4253 l2arc_write_done(zio_t *zio)
4254 {
4255         l2arc_write_callback_t *cb;
4256         l2arc_dev_t *dev;
4257         list_t *buflist;
4258         arc_buf_hdr_t *head, *ab, *ab_prev;
4259         l2arc_buf_hdr_t *abl2;
4260         kmutex_t *hash_lock;
4261         int64_t bytes_dropped = 0;

4263         cb = zio->io_private;
4264         ASSERT(cb != NULL);
4265         dev = cb->l2wcb_dev;
4266         ASSERT(dev != NULL);
4267         head = cb->l2wcb_head;
4268         ASSERT(head != NULL);
4269         buflist = dev->l2ad_buflist;
4270         ASSERT(buflist != NULL);
4271         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4272             l2arc_write_callback_t *, cb);

4274         if (zio->io_error != 0)
4275                 ARCSTAT_BUMP(arcstat_l2_writes_error);

4277         mutex_enter(&l2arc_buflist_mtx);

4279         /*
4280          * All writes completed, or an error was hit.
4281          */
4282         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4283                 ab_prev = list_prev(buflist, ab);
4284                 abl2 = ab->b_l2hdr;

4286                 /*
4287                  * Release the temporary compressed buffer as soon as possible.
4288                  */
4289                 if (abl2->b_compress != ZIO_COMPRESS_OFF)
4290                         l2arc_release_cdata_buf(ab);

4292                 hash_lock = HDR_LOCK(ab);
4293                 if (!mutex_tryenter(hash_lock)) {
4294                         /*
4295                          * This buffer misses out.  It may be in a stage
4296                          * of eviction.  Its ARC_L2_WRITING flag will be
4297                          * left set, denying reads to this buffer.
4298                          */
4299                         ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
4300                         continue;
4301                 }

4303                 if (zio->io_error != 0) {
4304                         /*
4305                          * Error - drop L2ARC entry.
4306                          */
4307                         list_remove(buflist, ab);
4308                         ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4309                         bytes_dropped += abl2->b_asize;
4310                         ab->b_l2hdr = NULL;
4311                         kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4312                         ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4313                 }

4315                 /*
4316                  * Allow ARC to begin reads to this L2ARC entry.
4317                  */
```

```
4318                 ab->b_flags &= ~ARC_L2_WRITING;
4320                 mutex_exit(hash_lock);
4321         }
4323         atomic_inc_64(&l2arc_writes_done);
4324         list_remove(buflist, head);
4325         kmem_cache_free(hdr_cache, head);
4326         mutex_exit(&l2arc_buflist_mtx);
4328         vdev_space_update(dev->l2ad_vdev, -bytes_dropped, 0, 0);
4330         l2arc_do_free_on_write();
4332         kmem_free(cb, sizeof (l2arc_write_callback_t));
4333 }
4335 /*
4336  * A read to a cache device completed.  Validate buffer contents before
4337  * handing over to the regular ARC routines.
4338  */
4339 static void
4340 l2arc_read_done(zio_t *zio)
4341 {
4342         l2arc_read_callback_t *cb;
4343         arc_buf_hdr_t *hdr;
4344         arc_buf_t *buf;
4345         kmutex_t *hash_lock;
4346         int equal;
4348         ASSERT(zio->io_vd != NULL);
4349         ASSERT(zio->io_flags & ZIO_FLAG_DONT_PROPAGATE);
4351         spa_config_exit(zio->io_spa, SCL_L2ARC, zio->io_vd);
4353         cb = zio->io_private;
4354         ASSERT(cb != NULL);
4355         buf = cb->l2rcb_buf;
4356         ASSERT(buf != NULL);
4358         hash_lock = HDR_LOCK(buf->b_hdr);
4359         mutex_enter(hash_lock);
4360         hdr = buf->b_hdr;
4361         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
4363         /*
4364          * If the buffer was compressed, decompress it first.
4365          */
4366         if (cb->l2rcb_compress != ZIO_COMPRESS_OFF)
4367                 l2arc_decompress_zio(zio, hdr, cb->l2rcb_compress);
4368         ASSERT(zio->io_data != NULL);
4370         /*
4371          * Check this survived the L2ARC journey.
4372          */
4373         equal = arc_cksum_equal(buf);
4374         if (equal && zio->io_error == 0 && !HDR_L2_EVICTED(hdr)) {
4375                 mutex_exit(hash_lock);
4376                 zio->io_private = buf;
4377                 zio->io_bp_copy = cb->l2rcb_bp; /* XXX fix in L2ARC 2.0 */
4378                 zio->io_bp = &zio->io_bp_copy;  /* XXX fix in L2ARC 2.0 */
4379                 arc_read_done(zio);
4380         } else {
4381                 mutex_exit(hash_lock);
4382                 /*
4383                  * Buffer didn't survive caching.  Increment stats and
```



```
4384                  * reissue to the original storage device.
4385                  */
4386                 if (zio->io_error != 0) {
4387                         ARCSTAT_BUMP(arcstat_l2_io_error);
4388                 } else {
4389                         zio->io_error = SET_ERROR(EIO);
4390                 }
4391                 if (!equal)
4392                         ARCSTAT_BUMP(arcstat_l2_cksum_bad);
4394                 /*
4395                  * If there's no waiter, issue an async i/o to the primary
4396                  * storage now.  If there *is* a waiter, the caller must
4397                  * issue the i/o in a context where it's OK to block.
4398                  */
4399                 if (zio->io_waiter == NULL) {
4400                         zio_t *pio = zio_unique_parent(zio);
4402                         ASSERT(!pio || pio->io_child_type == ZIO_CHILD_LOGICAL);
4404                         zio_nowait(zio_read(pio, cb->l2rcb_spa, &cb->l2rcb_bp,
4405                             buf->b_data, zio->io_size, arc_read_done, buf,
4406                             zio->io_priority, cb->l2rcb_flags, &cb->l2rcb_zb));
4407                 }
4408         }
4410         kmem_free(cb, sizeof (l2arc_read_callback_t));
4411 }
4413 /*
4414  * This is the list priority from which the L2ARC will search for pages to
4415  * cache.  This is used within loops (0..3) to cycle through lists in the
4416  * desired order.  This order can have a significant effect on cache
4417  * performance.
4418  *
4419  * Currently the metadata lists are hit first, MFU then MRU, followed by
4420  * the data lists.  This function returns a locked list, and also returns
4421  * the lock pointer.
4422  */
4423 static list_t *
4424 l2arc_list_locked(int list_num, kmutex_t **lock)
4425 {
4426         list_t *list = NULL;
4428         ASSERT(list_num >= 0 && list_num <= 3);
4430         switch (list_num) {
4431         case 0:
4432                 list = &arc_mfu->arcs_list[ARC_BUFC_METADATA];
4433                 *lock = &arc_mfu->arcs_mtx;
4434                 break;
4435         case 1:
4436                 list = &arc_mru->arcs_list[ARC_BUFC_METADATA];
4437                 *lock = &arc_mru->arcs_mtx;
4438                 break;
4439         case 2:
4440                 list = &arc_mfu->arcs_list[ARC_BUFC_DATA];
4441                 *lock = &arc_mfu->arcs_mtx;
4442                 break;
4443         case 3:
4444                 list = &arc_mru->arcs_list[ARC_BUFC_DATA];
4445                 *lock = &arc_mru->arcs_mtx;
4446                 break;
4447         }
4449         ASSERT(!(MUTEX_HELD(*lock)));
```

```
4450         mutex_enter(*lock);
4451         return (list);
4452 }

4454 /*
4455  * Evict buffers from the device write hand to the distance specified in
4456  * bytes.  This distance may span populated buffers, it may span nothing.
4457  * This is clearing a region on the L2ARC device ready for writing.
4458  * If the 'all' boolean is set, every buffer is evicted.
4459  */
4460 static void
4461 l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
4462 {
4463         list_t *buflist;
4464         l2arc_buf_hdr_t *abl2;
4465         arc_buf_hdr_t *ab, *ab_prev;
4466         kmutex_t *hash_lock;
4467         uint64_t taddr;
4468         int64_t bytes_evicted = 0;

4470         buflist = dev->l2ad_buflist;

4472         if (buflist == NULL)
4473                 return;

4475         if (!all && dev->l2ad_first) {
4476                 /*
4477                  * This is the first sweep through the device.  There is
4478                  * nothing to evict.
4479                  */
4480                 return;
4481         }

4483         if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
4484                 /*
4485                  * When nearing the end of the device, evict to the end
4486                  * before the device write hand jumps to the start.
4487                  */
4488                 taddr = dev->l2ad_end;
4489         } else {
4490                 taddr = dev->l2ad_hand + distance;
4491         }
4492         DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
4493             uint64_t, taddr, boolean_t, all);

4495 top:
4496         mutex_enter(&l2arc_buflist_mtx);
4497         for (ab = list_tail(buflist); ab; ab = ab_prev) {
4498                 ab_prev = list_prev(buflist, ab);

4500                 hash_lock = HDR_LOCK(ab);
4501                 if (!mutex_tryenter(hash_lock)) {
4502                         /*
4503                          * Missed the hash lock.  Retry.
4504                          */
4505                         ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
4506                         mutex_exit(&l2arc_buflist_mtx);
4507                         mutex_enter(hash_lock);
4508                         mutex_exit(hash_lock);
4509                         goto top;
4510                 }

4512                 if (HDR_L2_WRITE_HEAD(ab)) {
4513                         /*
4514                          * We hit a write head node.  Leave it for
4515                          * l2arc_write_done().
```



```
4516                          */
4517                         list_remove(buflist, ab);
4518                         mutex_exit(hash_lock);
4519                         continue;
4520                 }

4522                 if (!all && ab->b_l2hdr != NULL &&
4523                     (ab->b_l2hdr->b_daddr > taddr ||
4524                     ab->b_l2hdr->b_daddr < dev->l2ad_hand)) {
4525                         /*
4526                          * We've evicted to the target address,
4527                          * or the end of the device.
4528                          */
4529                         mutex_exit(hash_lock);
4530                         break;
4531                 }

4533                 if (HDR_FREE_IN_PROGRESS(ab)) {
4534                         /*
4535                          * Already on the path to destruction.
4536                          */
4537                         mutex_exit(hash_lock);
4538                         continue;
4539                 }

4541                 if (ab->b_state == arc_l2c_only) {
4542                         ASSERT(!HDR_L2_READING(ab));
4543                         /*
4544                          * This doesn't exist in the ARC.  Destroy.
4545                          * arc_hdr_destroy() will call list_remove()
4546                          * and decrement arcstat_l2_size.
4547                          */
4548                         arc_change_state(arc_anon, ab, hash_lock);
4549                         arc_hdr_destroy(ab);
4550                 } else {
4551                         /*
4552                          * Invalidate issued or about to be issued
4553                          * reads, since we may be about to write
4554                          * over this location.
4555                          */
4556                         if (HDR_L2_READING(ab)) {
4557                                 ARCSTAT_BUMP(arcstat_l2_evict_reading);
4558                                 ab->b_flags |= ARC_L2_EVICTED;
4559                         }

4561                         /*
4562                          * Tell ARC this no longer exists in L2ARC.
4563                          */
4564                         if (ab->b_l2hdr != NULL) {
4565                                 abl2 = ab->b_l2hdr;
4566                                 ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4567                                 bytes_evicted += abl2->b_asize;
4568                                 ab->b_l2hdr = NULL;
4569                                 /*
4570                                  * We are destroying l2hdr, so ensure that
4571                                  * its compressed buffer, if any, is not leaked.
4572                                  */
4573                                 ASSERT(abl2->b_tmp_cdata == NULL);
4574 #endif /* ! codereview */
4575                                 kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4576                                 ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4577                         }
4578                         list_remove(buflist, ab);

4580                         /*
4581                          * This may have been leftover after a
```

```
4582                         * failed write.
4583                         */
4584                        ab->b_flags &= ~ARC_L2_WRITING;
4585                }
4586                mutex_exit(hash_lock);
4587        }
4588        mutex_exit(&l2arc_buflist_mtx);

4590        vdev_space_update(dev->l2ad_vdev, -bytes_evicted, 0, 0);
4591        dev->l2ad_evict = taddr;
4592 }

4594 /*
4595  * Find and write ARC buffers to the L2ARC device.
4596  *
4597  * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4598  * for reading until they have completed writing.
4599  * The headroom_boost is an in-out parameter used to maintain headroom boost
4600  * state between calls to this function.
4601  *
4602  * Returns the number of bytes actually written (which may be smaller than
4603  * the delta by which the device hand has changed due to alignment).
4604  */
4605 static uint64_t
4606 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
4607     boolean_t *headroom_boost)
4608 {
4609        arc_buf_hdr_t *ab, *ab_prev, *head;
4610        list_t *list;
4611        uint64_t write_asize, write_psize, write_sz, headroom,
4612            buf_compress_minsz;
4613        void *buf_data;
4614        kmutex_t *list_lock;
4615        boolean_t full;
4616        l2arc_write_callback_t *cb;
4617        zio_t *pio, *wzio;
4618        uint64_t guid = spa_load_guid(spa);
4619        const boolean_t do_headroom_boost = *headroom_boost;

4621        ASSERT(dev->l2ad_vdev != NULL);

4623        /* Lower the flag now, we might want to raise it again later. */
4624        *headroom_boost = B_FALSE;

4626        pio = NULL;
4627        write_sz = write_asize = write_psize = 0;
4628        full = B_FALSE;
4629        head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
4630        head->b_flags |= ARC_L2_WRITE_HEAD;

4632        /*
4633         * We will want to try to compress buffers that are at least 2x the
4634         * device sector size.
4635         */
4636        buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

4638        /*
4639         * Copy buffers for L2ARC writing.
4640         */
4641        mutex_enter(&l2arc_buflist_mtx);
4642        for (int try = 0; try <= 3; try++) {
4643                uint64_t passed_sz = 0;

4645                list = l2arc_list_locked(try, &list_lock);

4647                /*
```



```
4648                 * L2ARC fast warmup.
4649                 *
4650                 * Until the ARC is warm and starts to evict, read from the
4651                 * head of the ARC lists rather than the tail.
4652                 */
4653                if (arc_warm == B_FALSE)
4654                        ab = list_head(list);
4655                else
4656                        ab = list_tail(list);

4658                headroom = target_sz * l2arc_headroom;
4659                if (do_headroom_boost)
4660                        headroom = (headroom * l2arc_headroom_boost) / 100;

4662                for (; ab; ab = ab_prev) {
4663                        l2arc_buf_hdr_t *l2hdr;
4664                        kmutex_t *hash_lock;
4665                        uint64_t buf_sz;

4667                        if (arc_warm == B_FALSE)
4668                                ab_prev = list_next(list, ab);
4669                        else
4670                                ab_prev = list_prev(list, ab);

4672                        hash_lock = HDR_LOCK(ab);
4673                        if (!mutex_tryenter(hash_lock)) {
4674                                /*
4675                                 * Skip this buffer rather than waiting.
4676                                 */
4677                                continue;
4678                        }

4680                        passed_sz += ab->b_size;
4681                        if (passed_sz > headroom) {
4682                                /*
4683                                 * Searched too far.
4684                                 */
4685                                mutex_exit(hash_lock);
4686                                break;
4687                        }

4689                        if (!l2arc_write_eligible(guid, ab)) {
4690                                mutex_exit(hash_lock);
4691                                continue;
4692                        }

4694                        if ((write_sz + ab->b_size) > target_sz) {
4695                                full = B_TRUE;
4696                                mutex_exit(hash_lock);
4697                                break;
4698                        }

4700                        if (pio == NULL) {
4701                                /*
4702                                 * Insert a dummy header on the buflist so
4703                                 * l2arc_write_done() can find where the
4704                                 * write buffers begin without searching.
4705                                 */
4706                                list_insert_head(dev->l2ad_buflist, head);

4708                                cb = kmem_alloc(
4709                                    sizeof (l2arc_write_callback_t), KM_SLEEP);
4710                                cb->l2wcb_dev = dev;
4711                                cb->l2wcb_head = head;
4712                                pio = zio_root(spa, l2arc_write_done, cb,
4713                                    ZIO_FLAG_CANFAIL);
```

```
4714                        }
4716                        /*
4717                         * Create and add a new L2ARC header.
4718                         */
4719                        l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
4720                        l2hdr->b_dev = dev;
4721                        ab->b_flags |= ARC_L2_WRITING;
4723                        /*
4724                         * Temporarily stash the data buffer in b_tmp_cdata.
4725                         * The subsequent write step will pick it up from
4726                         * there. This is because can't access ab->b_buf
4727                         * without holding the hash_lock, which we in turn
4728                         * can't access without holding the ARC list locks
4729                         * (which we want to avoid during compression/writing).
4730                         */
4731                        l2hdr->b_compress = ZIO_COMPRESS_OFF;
4732                        l2hdr->b_asize = ab->b_size;
4733                        l2hdr->b_tmp_cdata = ab->b_buf->b_data;
4735                        buf_sz = ab->b_size;
4736                        ab->b_l2hdr = l2hdr;
4738                        list_insert_head(dev->l2ad_buflist, ab);
4740                        /*
4741                         * Compute and store the buffer cksum before
4742                         * writing.  On debug the cksum is verified first.
4743                         */
4744                        arc_cksum_verify(ab->b_buf);
4745                        arc_cksum_compute(ab->b_buf, B_TRUE);
4747                        mutex_exit(hash_lock);
4749                        write_sz += buf_sz;
4750                }
4752                mutex_exit(list_lock);
4754                if (full == B_TRUE)
4755                        break;
4756        }
4758        /* No buffers selected for writing? */
4759        if (pio == NULL) {
4760                ASSERT0(write_sz);
4761                mutex_exit(&l2arc_buflist_mtx);
4762                kmem_cache_free(hdr_cache, head);
4763                return (0);
4764        }
4766        /*
4767         * Now start writing the buffers. We're starting at the write head
4768         * and work backwards, retracing the course of the buffer selector
4769         * loop above.
4770         */
4771        for (ab = list_prev(dev->l2ad_buflist, head); ab;
4772            ab = list_prev(dev->l2ad_buflist, ab)) {
4773                l2arc_buf_hdr_t *l2hdr;
4774                uint64_t buf_sz;
4776                /*
4777                 * We shouldn't need to lock the buffer here, since we flagged
4778                 * it as ARC_L2_WRITING in the previous step, but we must take
4779                 * care to only access its L2 cache parameters. In particular,
```



```
4780                 * ab->b_buf may be invalid by now due to ARC eviction.
4781                 */
4782                l2hdr = ab->b_l2hdr;
4783                l2hdr->b_daddr = dev->l2ad_hand;
4785                if ((ab->b_flags & ARC_L2COMPRESS) &&
4786                    l2hdr->b_asize >= buf_compress_minsz) {
4787                        if (l2arc_compress_buf(l2hdr)) {
4788                                /*
4789                                 * If compression succeeded, enable headroom
4790                                 * boost on the next scan cycle.
4791                                 */
4792                                *headroom_boost = B_TRUE;
4793                        }
4794                }
4796                /*
4797                 * Pick up the buffer data we had previously stashed away
4798                 * (and now potentially also compressed).
4799                 */
4800                buf_data = l2hdr->b_tmp_cdata;
4801                buf_sz = l2hdr->b_asize;
4803                /*
4804                 * If the data has not been compressed, then clear b_tmp_cdata
4805                 * to make sure that it points only to a temporary compression
4806                 * buffer.
4807                 */
4808                if (!L2ARC_IS_VALID_COMPRESS(l2hdr->b_compress))
4809                        l2hdr->b_tmp_cdata = NULL;
4811 #endif /* ! codereview */
4812                /* Compression may have squashed the buffer to zero length. */
4813                if (buf_sz != 0) {
4814                        uint64_t buf_p_sz;
4816                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
4817                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
4818                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
4819                            ZIO_FLAG_CANFAIL, B_FALSE);
4821                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
4822                            zio_t *, wzio);
4823                        (void) zio_nowait(wzio);
4825                        write_asize += buf_sz;
4826                        /*
4827                         * Keep the clock hand suitably device-aligned.
4828                         */
4829                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4830                        write_psize += buf_p_sz;
4831                        dev->l2ad_hand += buf_p_sz;
4832                }
4833        }
4835        mutex_exit(&l2arc_buflist_mtx);
4837        ASSERT3U(write_asize, <=, target_sz);
4838        ARCSTAT_BUMP(arcstat_l2_writes_sent);
4839        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
4840        ARCSTAT_INCR(arcstat_l2_size, write_sz);
4841        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
4842        vdev_space_update(dev->l2ad_vdev, write_asize, 0, 0);
4844        /*
4845         * Bump device hand to the device start if it is approaching the end.
```

```
4846          * l2arc_evict() will already have evicted ahead for this case.
4847          */
4848         if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
4849                 dev->l2ad_hand = dev->l2ad_start;
4850                 dev->l2ad_evict = dev->l2ad_start;
4851                 dev->l2ad_first = B_FALSE;
4852         }

4854         dev->l2ad_writing = B_TRUE;
4855         (void) zio_wait(pio);
4856         dev->l2ad_writing = B_FALSE;

4858         return (write_asize);
4859 }

4861 /*
4862  * Compresses an L2ARC buffer.
4863  * The data to be compressed must be prefilled in l2hdr->b_tmp_cdata and its
4864  * size in l2hdr->b_asize. This routine tries to compress the data and
4865  * depending on the compression result there are three possible outcomes:
4866  * *) The buffer was incompressible. The original l2hdr contents were left
4867  *    untouched and are ready for writing to an L2 device.
4868  * *) The buffer was all-zeros, so there is no need to write it to an L2
4869  *    device. To indicate this situation b_tmp_cdata is NULL'ed, b_asize is
4870  *    set to zero and b_compress is set to ZIO_COMPRESS_EMPTY.
4871  * *) Compression succeeded and b_tmp_cdata was replaced with a temporary
4872  *    data buffer which holds the compressed data to be written, and b_asize
4873  *    tells us how much data there is. b_compress is set to the appropriate
4874  *    compression algorithm. Once writing is done, invoke
4875  *    l2arc_release_cdata_buf on this l2hdr to free this temporary buffer.
4876  *
4877  * Returns B_TRUE if compression succeeded, or B_FALSE if it didn't (the
4878  * buffer was incompressible).
4879  */
4880 static boolean_t
4881 l2arc_compress_buf(l2arc_buf_hdr_t *l2hdr)
4882 {
4883         void *cdata;
4884         size_t csize, len, rounded;

4886         ASSERT(l2hdr->b_compress == ZIO_COMPRESS_OFF);
4887         ASSERT(l2hdr->b_tmp_cdata != NULL);

4889         len = l2hdr->b_asize;
4890         cdata = zio_data_buf_alloc(len);
4891         csize = zio_compress_data(ZIO_COMPRESS_LZ4, l2hdr->b_tmp_cdata,
4892             cdata, l2hdr->b_asize);

4894         rounded = P2ROUNDUP(csize, (size_t)SPA_MINBLOCKSIZE);
4895         if (rounded > csize) {
4896                 bzero((char *)cdata + csize, rounded - csize);
4897                 csize = rounded;
4898         }

4900         if (csize == 0) {
4901                 /* zero block, indicate that there's nothing to write */
4902                 zio_data_buf_free(cdata, len);
4903                 l2hdr->b_compress = ZIO_COMPRESS_EMPTY;
4904                 l2hdr->b_asize = 0;
4905                 l2hdr->b_tmp_cdata = NULL;
4906                 ARCSTAT_BUMP(arcstat_l2_compress_zeros);
4907                 return (B_TRUE);
4908         } else if (csize > 0 && csize < len) {
4909                 /*
4910                  * Compression succeeded, we'll keep the cdata around for
4911                  * writing and release it afterwards.
```



```
4912                  */
4913                 l2hdr->b_compress = ZIO_COMPRESS_LZ4;
4914                 l2hdr->b_asize = csize;
4915                 l2hdr->b_tmp_cdata = cdata;
4916                 ARCSTAT_BUMP(arcstat_l2_compress_successes);
4917                 return (B_TRUE);
4918         } else {
4919                 /*
4920                  * Compression failed, release the compressed buffer.
4921                  * l2hdr will be left unmodified.
4922                  */
4923                 zio_data_buf_free(cdata, len);
4924                 ARCSTAT_BUMP(arcstat_l2_compress_failures);
4925                 return (B_FALSE);
4926         }
4927 }

4929 /*
4930  * Decompresses a zio read back from an l2arc device. On success, the
4931  * underlying zio's io_data buffer is overwritten by the uncompressed
4932  * version. On decompression error (corrupt compressed stream), the
4933  * zio->io_error value is set to signal an I/O error.
4934  *
4935  * Please note that the compressed data stream is not checksummed, so
4936  * if the underlying device is experiencing data corruption, we may feed
4937  * corrupt data to the decompressor, so the decompressor needs to be
4938  * able to handle this situation (LZ4 does).
4939  */
4940 static void
4941 l2arc_decompress_zio(zio_t *zio, arc_buf_hdr_t *hdr, enum zio_compress c)
4942 {
4943         ASSERT(L2ARC_IS_VALID_COMPRESS(c));

4945         if (zio->io_error != 0) {
4946                 /*
4947                  * An io error has occured, just restore the original io
4948                  * size in preparation for a main pool read.
4949                  */
4950                 zio->io_orig_size = zio->io_size = hdr->b_size;
4951                 return;
4952         }

4954         if (c == ZIO_COMPRESS_EMPTY) {
4955                 /*
4956                  * An empty buffer results in a null zio, which means we
4957                  * need to fill its io_data after we're done restoring the
4958                  * buffer's contents.
4959                  */
4960                 ASSERT(hdr->b_buf != NULL);
4961                 bzero(hdr->b_buf->b_data, hdr->b_size);
4962                 zio->io_data = zio->io_orig_data = hdr->b_buf->b_data;
4963         } else {
4964                 ASSERT(zio->io_data != NULL);
4965                 /*
4966                  * We copy the compressed data from the start of the arc buffer
4967                  * (the zio_read will have pulled in only what we need, the
4968                  * rest is garbage which we will overwrite at decompression)
4969                  * and then decompress back to the ARC data buffer. This way we
4970                  * can minimize copying by simply decompressing back over the
4971                  * original compressed data (rather than decompressing to an
4972                  * aux buffer and then copying back the uncompressed buffer,
4973                  * which is likely to be much larger).
4974                  */
4975                 uint64_t csize;
4976                 void *cdata;
```

```
4978                 csize = zio->io_size;
4979                 cdata = zio_data_buf_alloc(csize);
4980                 bcopy(zio->io_data, cdata, csize);
4981                 if (zio_decompress_data(c, cdata, zio->io_data, csize,
4982                     hdr->b_size) != 0)
4983                         zio->io_error = EIO;
4984                 zio_data_buf_free(cdata, csize);
4985         }

4987         /* Restore the expected uncompressed IO size. */
4988         zio->io_orig_size = zio->io_size = hdr->b_size;
4989 }

4991 /*
4992  * Releases the temporary b_tmp_cdata buffer in an l2arc header structure.
4993  * This buffer serves as a temporary holder of compressed data while
4994  * the buffer entry is being written to an l2arc device. Once that is
4995  * done, we can dispose of it.
4996  */
4997 static void
4998 l2arc_release_cdata_buf(arc_buf_hdr_t *ab)
4999 {
5000         l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;

5002         ASSERT(L2ARC_IS_VALID_COMPRESS(l2hdr->b_compress));
5003         if (l2hdr->b_compress != ZIO_COMPRESS_EMPTY) {
 330         if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
5004                 /*
5005                  * If the data was compressed, then we've allocated a
5006                  * temporary buffer for it, so now we need to release it.
5007                  */
5008                 ASSERT(l2hdr->b_tmp_cdata != NULL);
5009                 zio_data_buf_free(l2hdr->b_tmp_cdata, ab->b_size);
5010                 l2hdr->b_tmp_cdata = NULL;
5011         } else {
5012                 ASSERT(l2hdr->b_tmp_cdata == NULL);
5013 #endif /* ! codereview */
5014         }
 337         l2hdr->b_tmp_cdata = NULL;
5015 }
```

_____*unchanged_portion_omitted_*