```
**********************************************************
  192861 Mon Dec  9 16:07:35 2013
new/usr/src/uts/common/fs/zfs/arc.c
3525 Persistent L2ARC
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

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



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

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

#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/zio_compress.h>
#include <sys/zfs_context.h>
#include <sys/arc.h>
```

```
 128 #include <sys/refcount.h>
 129 #include <sys/vdev.h>
 130 #include <sys/vdev_impl.h>
 131 #include <sys/dsl_pool.h>
 132 #ifdef _KERNEL
 133 #include <sys/vmsystm.h>
 134 #include <vm/anon.h>
 135 #include <sys/fs/swapnode.h>
 136 #include <sys/dnlc.h>
 137 #endif
 138 #include <sys/callb.h>
 139 #include <sys/kstat.h>
 140 #include <zfs_fletcher.h>
 141 #include <sys/byteorder.h>
 142 #include <sys/spa_impl.h>

 144 #ifndef _KERNEL
 145 /* set with ZFS_DEBUG=watch, to enable watchpoints on frozen buffers */
 146 boolean_t arc_watch = B_FALSE;
 147 int arc_procfd;
 148 #endif

 150 static kmutex_t         arc_reclaim_thr_lock;
 151 static kcondvar_t       arc_reclaim_thr_cv;     /* used to signal reclaim thr */
 152 static uint8_t          arc_thread_exit;

 154 #define ARC_REDUCE_DNLC_PERCENT 3
 155 uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

 157 typedef enum arc_reclaim_strategy {
 158         ARC_RECLAIM_AGGR,               /* Aggressive reclaim strategy */
 159         ARC_RECLAIM_CONS                /* Conservative reclaim strategy */
 160 } arc_reclaim_strategy_t;
_____unchanged_portion_omitted_

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

 253 typedef struct arc_stats {
 254         kstat_named_t arcstat_hits;
 255         kstat_named_t arcstat_misses;
 256         kstat_named_t arcstat_demand_data_hits;
 257         kstat_named_t arcstat_demand_data_misses;
 258         kstat_named_t arcstat_demand_metadata_hits;
 259         kstat_named_t arcstat_demand_metadata_misses;
 260         kstat_named_t arcstat_prefetch_data_hits;
 261         kstat_named_t arcstat_prefetch_data_misses;
 262         kstat_named_t arcstat_prefetch_metadata_hits;
 263         kstat_named_t arcstat_prefetch_metadata_misses;
 264         kstat_named_t arcstat_mru_hits;
 265         kstat_named_t arcstat_mru_ghost_hits;
 266         kstat_named_t arcstat_mfu_hits;
 267         kstat_named_t arcstat_mfu_ghost_hits;
 268         kstat_named_t arcstat_deleted;
 269         kstat_named_t arcstat_recycle_miss;
 270         /*
 271          * Number of buffers that could not be evicted because the hash lock
 272          * was held by another thread.  The lock may not necessarily be held
 273          * by something using the same buffer, since hash locks are shared
 274          * by multiple buffers.
 275          */
```



```
 276         kstat_named_t arcstat_mutex_miss;
 277         /*
 278          * Number of buffers skipped because they have I/O in progress, are
 279          * indirect prefetch buffers that have not lived long enough, or are
 280          * not from the spa we're trying to evict from.
 281          */
 282         kstat_named_t arcstat_evict_skip;
 283         kstat_named_t arcstat_evict_l2_cached;
 284         kstat_named_t arcstat_evict_l2_eligible;
 285         kstat_named_t arcstat_evict_l2_ineligible;
 286         kstat_named_t arcstat_hash_elements;
 287         kstat_named_t arcstat_hash_elements_max;
 288         kstat_named_t arcstat_hash_collisions;
 289         kstat_named_t arcstat_hash_chains;
 290         kstat_named_t arcstat_hash_chain_max;
 291         kstat_named_t arcstat_p;
 292         kstat_named_t arcstat_c;
 293         kstat_named_t arcstat_c_min;
 294         kstat_named_t arcstat_c_max;
 295         kstat_named_t arcstat_size;
 296         kstat_named_t arcstat_hdr_size;
 297         kstat_named_t arcstat_data_size;
 298         kstat_named_t arcstat_other_size;
 299         kstat_named_t arcstat_l2_hits;
 300         kstat_named_t arcstat_l2_misses;
 301         kstat_named_t arcstat_l2_feeds;
 302         kstat_named_t arcstat_l2_rw_clash;
 303         kstat_named_t arcstat_l2_read_bytes;
 304         kstat_named_t arcstat_l2_write_bytes;
 305         kstat_named_t arcstat_l2_writes_sent;
 306         kstat_named_t arcstat_l2_writes_done;
 307         kstat_named_t arcstat_l2_writes_error;
 308         kstat_named_t arcstat_l2_writes_hdr_miss;
 309         kstat_named_t arcstat_l2_evict_lock_retry;
 310         kstat_named_t arcstat_l2_evict_reading;
 311         kstat_named_t arcstat_l2_free_on_write;
 312         kstat_named_t arcstat_l2_abort_lowmem;
 313         kstat_named_t arcstat_l2_cksum_bad;
 314         kstat_named_t arcstat_l2_io_error;
 315         kstat_named_t arcstat_l2_size;
 316         kstat_named_t arcstat_l2_asize;
 317         kstat_named_t arcstat_l2_hdr_size;
 318         kstat_named_t arcstat_l2_compress_successes;
 319         kstat_named_t arcstat_l2_compress_zeros;
 320         kstat_named_t arcstat_l2_compress_failures;
 321         kstat_named_t arcstat_l2_log_blk_writes;
 322         kstat_named_t arcstat_l2_log_blk_avg_size;
 323         kstat_named_t arcstat_l2_data_to_meta_ratio;
 324         kstat_named_t arcstat_l2_rebuild_successes;
 325         kstat_named_t arcstat_l2_rebuild_abort_unsupported;
 326         kstat_named_t arcstat_l2_rebuild_abort_timeout;
 327         kstat_named_t arcstat_l2_rebuild_abort_io_errors;
 328         kstat_named_t arcstat_l2_rebuild_abort_cksum_errors;
 329         kstat_named_t arcstat_l2_rebuild_abort_loop_errors;
 330         kstat_named_t arcstat_l2_rebuild_abort_lowmem;
 331         kstat_named_t arcstat_l2_rebuild_size;
 332         kstat_named_t arcstat_l2_rebuild_bufs;
 333         kstat_named_t arcstat_l2_rebuild_bufs_precached;
 334         kstat_named_t arcstat_l2_rebuild_psize;
 335         kstat_named_t arcstat_l2_rebuild_log_blks;
 336         kstat_named_t arcstat_memory_throttle_count;
 337         kstat_named_t arcstat_duplicate_buffers;
 338         kstat_named_t arcstat_duplicate_buffers_size;
 339         kstat_named_t arcstat_duplicate_reads;
 340         kstat_named_t arcstat_meta_used;
 341         kstat_named_t arcstat_meta_limit;
```

```
342         kstat_named_t arcstat_meta_max;
343 } arc_stats_t;

345 static arc_stats_t arc_stats = {
346         { "hits",                       KSTAT_DATA_UINT64 },
347         { "misses",                     KSTAT_DATA_UINT64 },
348         { "demand_data_hits",           KSTAT_DATA_UINT64 },
349         { "demand_data_misses",         KSTAT_DATA_UINT64 },
350         { "demand_metadata_hits",       KSTAT_DATA_UINT64 },
351         { "demand_metadata_misses",     KSTAT_DATA_UINT64 },
352         { "prefetch_data_hits",         KSTAT_DATA_UINT64 },
353         { "prefetch_data_misses",       KSTAT_DATA_UINT64 },
354         { "prefetch_metadata_hits",     KSTAT_DATA_UINT64 },
355         { "prefetch_metadata_misses",   KSTAT_DATA_UINT64 },
356         { "mru_hits",                   KSTAT_DATA_UINT64 },
357         { "mru_ghost_hits",             KSTAT_DATA_UINT64 },
358         { "mfu_hits",                   KSTAT_DATA_UINT64 },
359         { "mfu_ghost_hits",             KSTAT_DATA_UINT64 },
360         { "deleted",                    KSTAT_DATA_UINT64 },
361         { "recycle_miss",               KSTAT_DATA_UINT64 },
362         { "mutex_miss",                 KSTAT_DATA_UINT64 },
363         { "evict_skip",                 KSTAT_DATA_UINT64 },
364         { "evict_l2_cached",            KSTAT_DATA_UINT64 },
365         { "evict_l2_eligible",          KSTAT_DATA_UINT64 },
366         { "evict_l2_ineligible",        KSTAT_DATA_UINT64 },
367         { "hash_elements",              KSTAT_DATA_UINT64 },
368         { "hash_elements_max",          KSTAT_DATA_UINT64 },
369         { "hash_collisions",            KSTAT_DATA_UINT64 },
370         { "hash_chains",                KSTAT_DATA_UINT64 },
371         { "hash_chain_max",             KSTAT_DATA_UINT64 },
372         { "p",                          KSTAT_DATA_UINT64 },
373         { "c",                          KSTAT_DATA_UINT64 },
374         { "c_min",                      KSTAT_DATA_UINT64 },
375         { "c_max",                      KSTAT_DATA_UINT64 },
376         { "size",                       KSTAT_DATA_UINT64 },
377         { "hdr_size",                   KSTAT_DATA_UINT64 },
378         { "data_size",                  KSTAT_DATA_UINT64 },
379         { "other_size",                 KSTAT_DATA_UINT64 },
380         { "l2_hits",                    KSTAT_DATA_UINT64 },
381         { "l2_misses",                  KSTAT_DATA_UINT64 },
382         { "l2_feeds",                   KSTAT_DATA_UINT64 },
383         { "l2_rw_clash",                KSTAT_DATA_UINT64 },
384         { "l2_read_bytes",              KSTAT_DATA_UINT64 },
385         { "l2_write_bytes",             KSTAT_DATA_UINT64 },
386         { "l2_writes_sent",             KSTAT_DATA_UINT64 },
387         { "l2_writes_done",             KSTAT_DATA_UINT64 },
388         { "l2_writes_error",            KSTAT_DATA_UINT64 },
389         { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
390         { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
391         { "l2_evict_reading",           KSTAT_DATA_UINT64 },
392         { "l2_free_on_write",           KSTAT_DATA_UINT64 },
393         { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
394         { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
395         { "l2_io_error",                KSTAT_DATA_UINT64 },
396         { "l2_size",                    KSTAT_DATA_UINT64 },
397         { "l2_asize",                   KSTAT_DATA_UINT64 },
398         { "l2_hdr_size",                KSTAT_DATA_UINT64 },
399         { "l2_compress_successes",      KSTAT_DATA_UINT64 },
400         { "l2_compress_zeros",          KSTAT_DATA_UINT64 },
401         { "l2_compress_failures",       KSTAT_DATA_UINT64 },
402         { "l2_log_blk_writes",          KSTAT_DATA_UINT64 },
403         { "l2_log_blk_avg_size",        KSTAT_DATA_UINT64 },
404         { "l2_data_to_meta_ratio",      KSTAT_DATA_UINT64 },
405         { "l2_rebuild_successes",       KSTAT_DATA_UINT64 },
406         { "l2_rebuild_unsupported",     KSTAT_DATA_UINT64 },
407         { "l2_rebuild_timeout",         KSTAT_DATA_UINT64 },
```



```
408         { "l2_rebuild_io_errors",       KSTAT_DATA_UINT64 },
409         { "l2_rebuild_cksum_errors",    KSTAT_DATA_UINT64 },
410         { "l2_rebuild_loop_errors",     KSTAT_DATA_UINT64 },
411         { "l2_rebuild_lowmem",          KSTAT_DATA_UINT64 },
412         { "l2_rebuild_psize",           KSTAT_DATA_UINT64 },
413         { "l2_rebuild_bufs",            KSTAT_DATA_UINT64 },
414         { "l2_rebuild_bufs_precached",  KSTAT_DATA_UINT64 },
415         { "l2_rebuild_size",            KSTAT_DATA_UINT64 },
416         { "l2_rebuild_log_blks",        KSTAT_DATA_UINT64 },
417         { "memory_throttle_count",      KSTAT_DATA_UINT64 },
418         { "duplicate_buffers",          KSTAT_DATA_UINT64 },
419         { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
420         { "duplicate_reads",            KSTAT_DATA_UINT64 },
421         { "arc_meta_used",              KSTAT_DATA_UINT64 },
422         { "arc_meta_limit",             KSTAT_DATA_UINT64 },
423         { "arc_meta_max",               KSTAT_DATA_UINT64 }
424 };
```

*unchanged_portion_omitted_*

```
441 #define ARCSTAT_MAXSTAT(stat) \
442         ARCSTAT_MAX(stat##_max, arc_stats.stat.value.ui64)

444 /*
445  * We define a macro to allow ARC hits/misses to be easily broken down by
446  * two separate conditions, giving a total of four different subtypes for
447  * each of hits and misses (so eight statistics total).
448  */
449 #define ARCSTAT_CONDSTAT(cond1, stat1, notstat1, cond2, stat2, notstat2, stat) \
450         if (cond1) {                                                    \
451                 if (cond2) {                                            \
452                         ARCSTAT_BUMP(arcstat_##stat1##_##stat2##_##stat); \
453                 } else {                                                \
454                         ARCSTAT_BUMP(arcstat_##stat1##_##notstat2##_##stat); \
455                 }                                                       \
456         } else {                                                        \
457                 if (cond2) {                                            \
458                         ARCSTAT_BUMP(arcstat_##notstat1##_##stat2##_##stat); \
459                 } else {                                                \
460                         ARCSTAT_BUMP(arcstat_##notstat1##_##notstat2##_##stat);\
461                 }                                                       \
462         }

464 /*
465  * This macro allows us to use kstats as floating averages. Each time we
466  * update this kstat, we first factor it and the update value by
467  * ARCSTAT_AVG_FACTOR to shrink the new value's contribution to the overall
468  * average. This macro assumes that integer loads and stores are atomic, but
469  * is not safe for multiple writers updating the kstat in parallel (only the
470  * last writer's update will remain).
471  */
472 #define ARCSTAT_F_AVG_FACTOR    3
473 #define ARCSTAT_F_AVG(stat, value) \
474         do { \
475                 uint64_t x = ARCSTAT(stat); \
476                 x = x - x / ARCSTAT_F_AVG_FACTOR + \
477                     (value) / ARCSTAT_F_AVG_FACTOR; \
478                 ARCSTAT(stat) = x; \
479                 _NOTE(NOTREACHED) \
480                 _NOTE(CONSTCOND) \
481         } while (0)

483 kstat_t                 *arc_ksp;
484 static arc_state_t      *arc_anon;
485 static arc_state_t      *arc_mru;
486 static arc_state_t      *arc_mru_ghost;
487 static arc_state_t      *arc_mfu;
```

```
 488 static arc_state_t      *arc_mfu_ghost;
 489 static arc_state_t      *arc_l2c_only;

 491 /*
 492  * There are several ARC variables that are critical to export as kstats --
 493  * but we don't want to have to grovel around in the kstat whenever we wish to
 494  * manipulate them.  For these variables, we therefore define them to be in
 495  * terms of the statistic variable.  This assures that we are not introducing
 496  * the possibility of inconsistency by having shadow copies of the variables,
 497  * while still allowing the code to be readable.
 498  */
 499 #define arc_size        ARCSTAT(arcstat_size)   /* actual total arc size */
 500 #define arc_p           ARCSTAT(arcstat_p)      /* target size of MRU */
 501 #define arc_c           ARCSTAT(arcstat_c)      /* target size of cache */
 502 #define arc_c_min       ARCSTAT(arcstat_c_min)  /* min target cache size */
 503 #define arc_c_max       ARCSTAT(arcstat_c_max)  /* max target cache size */
 504 #define arc_meta_limit  ARCSTAT(arcstat_meta_limit) /* max size for metadata */
 505 #define arc_meta_used   ARCSTAT(arcstat_meta_used) /* size of metadata */
 506 #define arc_meta_max    ARCSTAT(arcstat_meta_max) /* max size of metadata */

 508 #define L2ARC_IS_VALID_COMPRESS(_c_) \
 509         ((_c_) == ZIO_COMPRESS_LZ4 || (_c_) == ZIO_COMPRESS_EMPTY)

 511 static int              arc_no_grow;    /* Don't try to grow cache size */
 512 static uint64_t         arc_tempreserve;
 513 static uint64_t         arc_loaned_bytes;

 515 typedef struct l2arc_buf_hdr l2arc_buf_hdr_t;

 517 typedef struct arc_callback arc_callback_t;

 519 struct arc_callback {
 520         void                    *acb_private;
 521         arc_done_func_t         *acb_done;
 522         arc_buf_t               *acb_buf;
 523         zio_t                   *acb_zio_dummy;
 524         arc_callback_t          *acb_next;
 525 };
_____unchanged_portion_omitted_

 649 static buf_hash_table_t buf_hash_table;

 651 #define BUF_HASH_INDEX(spa, dva, birth) \
 652         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
 653 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
 654 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
 655 #define HDR_LOCK(hdr) \
 656         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))

 658 uint64_t zfs_crc64_table[256];

 660 /*
 661  * Level 2 ARC
 662  */

 664 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
 665 #define L2ARC_HEADROOM          2                       /* num of writes */
 666 /*
 667  * If we discover during ARC scan any buffers to be compressed, we boost
 668  * our headroom for the next scanning cycle by this percentage multiple.
 669  */
 670 #define L2ARC_HEADROOM_BOOST    200
 671 #define L2ARC_FEED_SECS         1               /* caching interval secs */
 672 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

 674 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
```



```
 675 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

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

 688 /*
 689  * L2ARC Internals
 690  */
 691 typedef struct l2arc_dev l2arc_dev_t;
 640 typedef struct l2arc_dev {
 641         vdev_t                  *l2ad_vdev;     /* vdev */
 642         spa_t                   *l2ad_spa;      /* spa */
 643         uint64_t                l2ad_hand;      /* next write location */
 644         uint64_t                l2ad_start;     /* first addr on device */
 645         uint64_t                l2ad_end;       /* last addr on device */
 646         uint64_t                l2ad_evict;     /* last addr eviction reached */
 647         boolean_t               l2ad_first;     /* first sweep through */
 648         boolean_t               l2ad_writing;   /* currently writing */
 649         list_t                  *l2ad_buflist;  /* buffer list */
 650         list_node_t             l2ad_node;      /* device list node */
 651 } l2arc_dev_t;

 692 static list_t L2ARC_dev_list;                   /* device list */
 693 static list_t *l2arc_dev_list;                  /* device list pointer */
 694 static kmutex_t l2arc_dev_mtx;                  /* device list mutex */
 695 static l2arc_dev_t *l2arc_dev_last;             /* last device used */
 696 static kmutex_t l2arc_buflist_mtx;              /* mutex for all buflists */
 697 static list_t L2ARC_free_on_write;              /* free after write buf list */
 698 static list_t *l2arc_free_on_write;             /* free after write list ptr */
 699 static kmutex_t l2arc_free_on_write_mtx;        /* mutex for list */
 700 static uint64_t l2arc_ndev;                     /* number of devices */

 702 typedef struct l2arc_read_callback {
 703         arc_buf_t               *l2rcb_buf;             /* read buffer */
 704         spa_t                   *l2rcb_spa;             /* spa */
 705         blkptr_t                l2rcb_bp;               /* original blkptr */
 706         zbookmark_t             l2rcb_zb;               /* original bookmark */
 707         int                     l2rcb_flags;            /* original flags */
 708         enum zio_compress       l2rcb_compress;         /* applied compress */
 709 } l2arc_read_callback_t;

 711 typedef struct l2arc_write_callback {
 712         l2arc_dev_t     *l2wcb_dev;             /* device info */
 713         arc_buf_hdr_t   *l2wcb_head;            /* head of write buflist */
 714         /* list of in-flight l2arc_log_blk_buf_t's */
 715         list_t          l2wcb_log_blk_buf_list;
 716 } l2arc_write_callback_t;
_____unchanged_portion_omitted_

 738 static kmutex_t l2arc_feed_thr_lock;
 739 static kcondvar_t l2arc_feed_thr_cv;
 740 static uint8_t l2arc_thread_exit;

 742 static void l2arc_read_done(zio_t *zio);
 743 static void l2arc_hdr_stat_add(boolean_t from_arc);
 702 static void l2arc_hdr_stat_add(void);
 744 static void l2arc_hdr_stat_remove(void);
 745 static l2arc_dev_t *l2arc_vdev_get(vdev_t *vd);
```

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

752 enum {
753         L2ARC_DEV_HDR_EVICT_FIRST = (1 << 0)    /* mirror of l2ad_first */
754 };

756 /*
757  * Pointer used in persistent L2ARC (for pointing to log blocks & ARC buffers).
758  */
759 typedef struct l2arc_log_blk_ptr {
760         uint64_t        l2lbp_daddr;    /* device address of log */
761         /*
762          * l2lbp_prop is the same format as the blk_prop in blkptr_t:
763          *      * logical size (in sectors)
764          *      * physical (compressed) size (in sectors)
765          *      * compression algorithm (we always LZ4-compress l2arc logs)
766          *      * checksum algorithm (used for l2lbp_cksum)
767          *      * object type & level (unused for now)
768          */
769         uint64_t        l2lbp_prop;
770         zio_cksum_t     l2lbp_cksum;    /* fletcher4 of log */
771 } l2arc_log_blk_ptr_t;

773 /*
774  * The persistent L2ARC device header.
775  */
776 typedef struct l2arc_dev_hdr_phys {
777         uint64_t        l2dh_magic;
778         zio_cksum_t     l2dh_self_cksum;        /* fletcher4 of fields below */

780         /*
781          * Global L2ARC device state and metadata.
782          */
783         uint64_t        l2dh_spa_guid;
784         uint64_t        l2dh_evict_tail;        /* current evict pointer */
785         uint64_t        l2dh_alloc_space;       /* vdev space alloc status */
786         uint64_t        l2dh_flags;             /* l2arc_dev_hdr_flags_t */

788         /*
789          * Start of log block chain. [0] -> newest log, [1] -> one older (used
790          * for initiating prefetch).
791          */
792         l2arc_log_blk_ptr_t     l2dh_start_lbps[2];

794         const uint64_t  l2dh_pad[43];           /* pad to 512 bytes */
795 } l2arc_dev_hdr_phys_t;
796 CTASSERT(sizeof (l2arc_dev_hdr_phys_t) == SPA_MINBLOCKSIZE);

798 /*
799  * A single ARC buffer header entry in a l2arc_log_blk_phys_t.
800  */
801 typedef struct l2arc_log_ent_phys {
802         dva_t                   l2le_dva;       /* dva of buffer */
803         uint64_t                l2le_birth;     /* birth txg of buffer */
804         uint64_t                l2le_cksum0;
805         zio_cksum_t             l2le_freeze_cksum;
806         /*
807          * l2le_prop is the same format as the blk_prop in blkptr_t:
808          *      * logical size (in sectors)
809          *      * physical (compressed) size (in sectors)
810          *      * compression algorithm
811          *      * checksum algorithm (used for cksum0)
```



```
812          *      * object type & level (used to restore arc_buf_contents_t)
813          */
814         uint64_t                l2le_prop;
815         uint64_t                l2le_daddr;     /* buf location on l2dev */
816         const uint64_t          l2le_pad[6];    /* resv'd for future use */
817 } l2arc_log_ent_phys_t;

819 /*
820  * These design limits give us the following overhead (before compression):
821  *      avg_blk_sz      overhead
822  *      1k              12.51 %
823  *      2k               6.26 %
824  *      4k               3.13 %
825  *      8k               1.56 %
826  *      16k              0.78 %
827  *      32k              0.39 %
828  *      64k              0.20 %
829  *      128k             0.10 %
830  * Compression should be able to sequeeze these down by about a factor of 2x.
831  */
832 #define L2ARC_LOG_BLK_SIZE                      (128 * 1024)    /* 128k */
833 #define L2ARC_LOG_BLK_HEADER_LEN                (128)
834 #define L2ARC_LOG_BLK_ENTRIES                   /* 1023 entries */      \
835         ((L2ARC_LOG_BLK_SIZE - L2ARC_LOG_BLK_HEADER_LEN) /              \
836         sizeof (l2arc_log_ent_phys_t))
837 /*
838  * Maximum amount of data in an l2arc log block (used to terminate rebuilding
839  * before we hit the write head and restore potentially corrupted blocks).
840  */
841 #define L2ARC_LOG_BLK_MAX_PAYLOAD_SIZE  \
842         (SPA_MAXBLOCKSIZE * L2ARC_LOG_BLK_ENTRIES)
843 /*
844  * For the persistency and rebuild algorithms to operate reliably we need
845  * the L2ARC device to at least be able to hold 3 full log blocks (otherwise
846  * excessive log block looping might confuse the log chain end detection).
847  * Under normal circumstances this is not a problem, since this is somewhere
848  * around only 400 MB.
849  */
850 #define L2ARC_PERSIST_MIN_SIZE  (3 * L2ARC_LOG_BLK_MAX_PAYLOAD_SIZE)

852 /*
853  * A log block of up to 1023 ARC buffer log entries, chained into the
854  * persistent L2ARC metadata linked list.
855  */
856 typedef struct l2arc_log_blk_phys {
857         /* Header - see L2ARC_LOG_BLK_HEADER_LEN above */
858         uint64_t                l2lb_magic;
859         l2arc_log_blk_ptr_t     l2lb_back2_lbp; /* back 2 steps in chain */
860         uint64_t                l2lb_pad[9];    /* resv'd for future use */
861         /* Payload */
862         l2arc_log_ent_phys_t    l2lb_entries[L2ARC_LOG_BLK_ENTRIES];
863 } l2arc_log_blk_phys_t;

865 CTASSERT(sizeof (l2arc_log_blk_phys_t) == L2ARC_LOG_BLK_SIZE);
866 CTASSERT(offsetof(l2arc_log_blk_phys_t, l2lb_entries) -
867     offsetof(l2arc_log_blk_phys_t, l2lb_magic) == L2ARC_LOG_BLK_HEADER_LEN);

869 /*
870  * These structures hold in-flight l2arc_log_blk_phys_t's as they're being
871  * written to the L2ARC device. They may be compressed, hence the uint8_t[].
872  */
873 typedef struct l2arc_log_blk_buf {
874         uint8_t         l2lbb_log_blk[sizeof (l2arc_log_blk_phys_t)];
875         list_node_t     l2lbb_node;
876 } l2arc_log_blk_buf_t;
```

```
878 /* Macros for the manipulation fields in the blk_prop format of blkptr_t */
879 #define BLKPROP_GET_LSIZE(_obj, _field)         \
880         BF64_GET_SB((_obj)->_field, 0, 16, SPA_MINBLOCKSHIFT, 1)
881 #define BLKPROP_SET_LSIZE(_obj, _field, x)      \
882         BF64_SET_SB((_obj)->_field, 0, 16, SPA_MINBLOCKSHIFT, 1, x)
883 #define BLKPROP_GET_PSIZE(_obj, _field)         \
884         BF64_GET_SB((_obj)->_field, 16, 16, SPA_MINBLOCKSHIFT, 1)
885 #define BLKPROP_SET_PSIZE(_obj, _field, x)      \
886         BF64_SET_SB((_obj)->_field, 16, 16, SPA_MINBLOCKSHIFT, 1, x)
887 #define BLKPROP_GET_COMPRESS(_obj, _field)      \
888         BF64_GET((_obj)->_field, 32, 8)
889 #define BLKPROP_SET_COMPRESS(_obj, _field, x)   \
890         BF64_SET((_obj)->_field, 32, 8, x)
891 #define BLKPROP_GET_CHECKSUM(_obj, _field)      \
892         BF64_GET((_obj)->_field, 40, 8)
893 #define BLKPROP_SET_CHECKSUM(_obj, _field, x)   \
894         BF64_SET((_obj)->_field, 40, 8, x)
895 #define BLKPROP_GET_TYPE(_obj, _field)          \
896         BF64_GET((_obj)->_field, 48, 8)
897 #define BLKPROP_SET_TYPE(_obj, _field, x)       \
898         BF64_SET((_obj)->_field, 48, 8, x)

900 /* Macros for manipulating a l2arc_log_blk_ptr_t->l2lbp_prop field */
901 #define LBP_GET_LSIZE(_add)             BLKPROP_GET_LSIZE(_add, l2lbp_prop)
902 #define LBP_SET_LSIZE(_add, x)          BLKPROP_SET_LSIZE(_add, l2lbp_prop, x)
903 #define LBP_GET_PSIZE(_add)             BLKPROP_GET_PSIZE(_add, l2lbp_prop)
904 #define LBP_SET_PSIZE(_add, x)          BLKPROP_SET_PSIZE(_add, l2lbp_prop, x)
905 #define LBP_GET_COMPRESS(_add)          BLKPROP_GET_COMPRESS(_add, l2lbp_prop)
906 #define LBP_SET_COMPRESS(_add, x)       BLKPROP_SET_COMPRESS(_add, l2lbp_prop, \
907     x)
908 #define LBP_GET_CHECKSUM(_add)          BLKPROP_GET_CHECKSUM(_add, l2lbp_prop)
909 #define LBP_SET_CHECKSUM(_add, x)       BLKPROP_SET_CHECKSUM(_add, l2lbp_prop, \
910     x)
911 #define LBP_GET_TYPE(_add)              BLKPROP_GET_TYPE(_add, l2lbp_prop)
912 #define LBP_SET_TYPE(_add, x)           BLKPROP_SET_TYPE(_add, l2lbp_prop, x)

914 /* Macros for manipulating a l2arc_log_ent_phys_t->l2le_prop field */
915 #define LE_GET_LSIZE(_le)       BLKPROP_GET_LSIZE(_le, l2le_prop)
916 #define LE_SET_LSIZE(_le, x)    BLKPROP_SET_LSIZE(_le, l2le_prop, x)
917 #define LE_GET_PSIZE(_le)       BLKPROP_GET_PSIZE(_le, l2le_prop)
918 #define LE_SET_PSIZE(_le, x)    BLKPROP_SET_PSIZE(_le, l2le_prop, x)
919 #define LE_GET_COMPRESS(_le)    BLKPROP_GET_COMPRESS(_le, l2le_prop)
920 #define LE_SET_COMPRESS(_le, x) BLKPROP_SET_COMPRESS(_le, l2le_prop, x)
921 #define LE_GET_CHECKSUM(_le)    BLKPROP_GET_CHECKSUM(_le, l2le_prop)
922 #define LE_SET_CHECKSUM(_le, x) BLKPROP_SET_CHECKSUM(_le, l2le_prop, x)
923 #define LE_GET_TYPE(_le)        BLKPROP_GET_TYPE(_le, l2le_prop)
924 #define LE_SET_TYPE(_le, x)     BLKPROP_SET_TYPE(_le, l2le_prop, x)

926 #define PTR_SWAP(x, y)          \
927         do {                    \
928                 void *tmp = (x);\
929                 x = y;          \
930                 y = tmp;        \
931                 _NOTE(CONSTCOND)\
932         } while (0)

934 #define L2ARC_DEV_HDR_MAGIC     0x12bab10c00000001LLU
935 #define L2ARC_LOG_BLK_MAGIC     0x120103b10c000001LLU
936 #define L2ARC_REBUILD_TIMEOUT   300     /* a rebuild may take at most 300s */

938 struct l2arc_dev {
939         vdev_t                  *l2ad_vdev;     /* vdev */
940         spa_t                   *l2ad_spa;      /* spa */
941         uint64_t                l2ad_hand;      /* next write location */
942         uint64_t                l2ad_start;     /* first addr on device */
943         uint64_t                l2ad_end;       /* last addr on device */
```



```
944         uint64_t                l2ad_evict;     /* last addr eviction reached */
945         boolean_t               l2ad_first;     /* first sweep through */
946         boolean_t               l2ad_writing;   /* currently writing */
947         list_t                  *l2ad_buflist;  /* buffer list */
948         list_node_t             l2ad_node;      /* device list node */
949         l2arc_dev_hdr_phys_t    l2ad_dev_hdr;   /* persistent device header */
950         l2arc_log_blk_phys_t    l2ad_log_blk;   /* currently open log block */
951         int                     l2ad_log_ent_idx; /* index into cur log blk */
952         /* number of bytes in current log block's payload */
953         uint64_t                l2ad_log_blk_payload_asize;
954         /* flag indicating whether a rebuild is scheduled or is going on */
955         boolean_t               l2ad_rebuild;
956 };

958 /*
959  * Performance tuning of L2ARC persistency:
960  *
961  * l2arc_rebuild_enabled : Controls whether L2ARC device adds (either at
962  *              pool import or when adding one manually later) will attempt
963  *              to rebuild L2ARC buffer contents. In special circumstances,
964  *              the administrator may want to set this to B_FALSE, if they
965  *              are having trouble importing a pool or attaching an L2ARC
966  *              device (e.g. the L2ARC device is slow to read in stored log
967  *              metadata, or the metadata has become somehow
968  *              fragmented/unusable).
969  * l2arc_rebuild_timeout : A hard timeout value on L2ARC rebuilding to help
970  *              avoid a slow L2ARC device from preventing pool import. If we
971  *              are not done rebuilding an L2ARC device by this time, we
972  *              stop the rebuild and return immediately.
973  */
974 boolean_t l2arc_rebuild_enabled = B_TRUE;
975 uint64_t l2arc_rebuild_timeout = L2ARC_REBUILD_TIMEOUT;

977 /*
978  * L2ARC persistency rebuild routines.
979  */
980 static void l2arc_dev_rebuild_start(l2arc_dev_t *dev);
981 static int l2arc_rebuild(l2arc_dev_t *dev);
982 static void l2arc_log_blk_restore(l2arc_dev_t *dev, uint64_t load_guid,
983     l2arc_log_blk_phys_t *lb, uint64_t lb_psize);
984 static void l2arc_hdr_restore(const l2arc_log_ent_phys_t *le,
985     l2arc_dev_t *dev, uint64_t guid);

987 /*
988  * L2ARC persistency read I/O routines.
989  */
990 static int l2arc_dev_hdr_read(l2arc_dev_t *dev, l2arc_dev_hdr_phys_t *hdr);
991 static int l2arc_log_blk_read(l2arc_dev_t *dev,
992     const l2arc_log_blk_ptr_t *this_lp, const l2arc_log_blk_ptr_t *next_lp,
993     l2arc_log_blk_phys_t *this_lb, l2arc_log_blk_phys_t *next_lb,
994     uint8_t *this_lb_buf, uint8_t *next_lb_buf,
995     zio_t *this_io, zio_t **next_io);
996 static boolean_t l2arc_log_blk_ptr_valid(l2arc_dev_t *dev,
997     const l2arc_log_blk_ptr_t *lp);
998 static zio_t *l2arc_log_blk_prefetch(vdev_t *vd,
999     const l2arc_log_blk_ptr_t *lp, uint8_t *lb_buf);
1000 static void l2arc_log_blk_prefetch_abort(zio_t *zio);

1002 /*
1003  * L2ARC persistency write I/O routines.
1004  */
1005 static void l2arc_dev_hdr_update(l2arc_dev_t *dev, zio_t *pio);
1006 static void l2arc_log_blk_commit(l2arc_dev_t *dev, zio_t *pio,
1007     l2arc_write_callback_t *cb);

1009 /*
```

```
 * L2ARC persistency auxilliary routines.
 */
static void l2arc_dev_hdr_checksum(const l2arc_dev_hdr_phys_t *hdr,
    zio_cksum_t *cksum);
static boolean_t l2arc_log_blk_insert(l2arc_dev_t *dev,
    const arc_buf_hdr_t *ab);
static inline boolean_t l2arc_range_check_overlap(uint64_t bottom,
    uint64_t top, uint64_t check);
static boolean_t l2arc_check_rebuild_timeout_hit(int64_t deadline);

static inline uint64_t
710 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);
}
```

_____*unchanged_portion_omitted_*

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

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

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

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

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

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

                        /*
                         * If prefetching out of the ghost cache,
                         * we will have a non-zero datacnt.
```



```
                         */
                        if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
                                /* ghost elements have a ghost size */
                                ASSERT(ab->b_buf == NULL);
                                from_delta = ab->b_size;
                        }
                        ASSERT3U(*size, >=, from_delta);
                        atomic_add_64(size, -from_delta);

                        if (use_mutex)
                                mutex_exit(&old_state->arcs_mtx);
                }
                if (new_state != arc_anon) {
                        int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
                        uint64_t *size = &new_state->arcs_lsize[ab->b_type];

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

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

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

                        if (use_mutex)
                                mutex_exit(&new_state->arcs_mtx);
                }
        }

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

        /* adjust state sizes */
        if (to_delta)
                atomic_add_64(&new_state->arcs_size, to_delta);
        if (from_delta) {
                ASSERT3U(old_state->arcs_size, >=, from_delta);
                atomic_add_64(&old_state->arcs_size, -from_delta);
        }
        ab->b_state = new_state;

        /* adjust l2arc hdr stats */
        if (new_state == arc_l2c_only)
                l2arc_hdr_stat_add(old_state != arc_anon);
1250            l2arc_hdr_stat_add();
        else if (old_state == arc_l2c_only)
                l2arc_hdr_stat_remove();
}
```

_____*unchanged_portion_omitted_*

```
/*
 * Allocates an empty arc_buf_hdr structure (lacking any data buffer).
 * This is used during l2arc reconstruction to make empty ARC buffers
 * which circumvent the regular disk->arc->l2arc path and instead come
 * into being in the reverse order, i.e. l2arc->arc->(disk).
 */
arc_buf_hdr_t *
arc_buf_hdr_alloc(uint64_t guid, int size, arc_buf_contents_t type)
{
        arc_buf_hdr_t *hdr;
```

```
1675         ASSERT3U(size, >, 0);
1676         hdr = kmem_cache_alloc(hdr_cache, KM_SLEEP);
1677         ASSERT(BUF_EMPTY(hdr));
1678         hdr->b_size = size;
1679         hdr->b_type = type;
1680         hdr->b_spa = guid;
1681         hdr->b_state = arc_anon;
1682         hdr->b_arc_access = 0;
1683         hdr->b_buf = NULL;
1684         hdr->b_datacnt = 0;
1685         hdr->b_flags = 0;
1686         ASSERT(refcount_is_zero(&hdr->b_refcnt));

1688         return (hdr);
1689 }

1691 static char *arc_onloan_tag = "onloan";

1693 /*
1694  * Loan out an anonymous arc buffer. Loaned buffers are not counted as in
1695  * flight data by arc_tempreserve_space() until they are "returned". Loaned
1696  * buffers must be returned to the arc before they can be used by the DMU or
1697  * freed.
1698  */
1699 arc_buf_t *
1700 arc_loan_buf(spa_t *spa, int size)
1701 {
1702         arc_buf_t *buf;

1704         buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);

1706         atomic_add_64(&arc_loaned_bytes, size);
1707         return (buf);
1708 }
______unchanged_portion_omitted_

1899 static void
1900 arc_hdr_destroy(arc_buf_hdr_t *hdr)
1901 {
1902         ASSERT(refcount_is_zero(&hdr->b_refcnt));
1903         ASSERT3P(hdr->b_state, ==, arc_anon);
1904         ASSERT(!HDR_IO_IN_PROGRESS(hdr));
1905         l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;

1907         if (l2hdr != NULL) {
1908                 boolean_t buflist_held = MUTEX_HELD(&l2arc_buflist_mtx);
1909                 /*
1910                  * To prevent arc_free() and l2arc_evict() from
1911                  * attempting to free the same buffer at the same time,
1912                  * a FREE_IN_PROGRESS flag is given to arc_free() to
1913                  * give it priority.  l2arc_evict() can't destroy this
1914                  * header while we are waiting on l2arc_buflist_mtx.
1915                  *
1916                  * The hdr may be removed from l2ad_buflist before we
1917                  * grab l2arc_buflist_mtx, so b_l2hdr is rechecked.
1918                  */
1919                 if (!buflist_held) {
1920                         mutex_enter(&l2arc_buflist_mtx);
1921                         l2hdr = hdr->b_l2hdr;
1922                 }

1924                 if (l2hdr != NULL) {
1925                         list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
1926                         ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);
1927                         ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
```



```
1928                         kmem_free(l2hdr, sizeof (*l2hdr));
1591                         kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
1929                         if (hdr->b_state == arc_l2c_only)
1930                                 l2arc_hdr_stat_remove();
1931                         hdr->b_l2hdr = NULL;
1932                 }

1934                 if (!buflist_held)
1935                         mutex_exit(&l2arc_buflist_mtx);
1936         }

1938         if (!BUF_EMPTY(hdr)) {
1939                 ASSERT(!HDR_IN_HASH_TABLE(hdr));
1940                 buf_discard_identity(hdr);
1941         }
1942         while (hdr->b_buf) {
1943                 arc_buf_t *buf = hdr->b_buf;

1945                 if (buf->b_efunc) {
1946                         mutex_enter(&arc_eviction_mtx);
1947                         mutex_enter(&buf->b_evict_lock);
1948                         ASSERT(buf->b_hdr != NULL);
1949                         arc_buf_destroy(hdr->b_buf, FALSE, FALSE);
1950                         hdr->b_buf = buf->b_next;
1951                         buf->b_hdr = &arc_eviction_hdr;
1952                         buf->b_next = arc_eviction_list;
1953                         arc_eviction_list = buf;
1954                         mutex_exit(&buf->b_evict_lock);
1955                         mutex_exit(&arc_eviction_mtx);
1956                 } else {
1957                         arc_buf_destroy(hdr->b_buf, FALSE, TRUE);
1958                 }
1959         }
1960         if (hdr->b_freeze_cksum != NULL) {
1961                 kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1962                 hdr->b_freeze_cksum = NULL;
1963         }
1964         if (hdr->b_thawed) {
1965                 kmem_free(hdr->b_thawed, 1);
1966                 hdr->b_thawed = NULL;
1967         }

1969         ASSERT(!list_link_active(&hdr->b_arc_node));
1970         ASSERT3P(hdr->b_hash_next, ==, NULL);
1971         ASSERT3P(hdr->b_acb, ==, NULL);
1972         kmem_cache_free(hdr_cache, hdr);
1973 }
______unchanged_portion_omitted_

3202 /*
3203  * "Read" the block at the specified DVA (in bp) via the
3204  * cache.  If the block is found in the cache, invoke the provided
3205  * callback immediately and return.  Note that the 'zio' parameter
3206  * in the callback will be NULL in this case, since no IO was
3207  * required.  If the block is not in the cache pass the read request
3208  * on to the spa with a substitute callback function, so that the
3209  * requested block will be added to the cache.
3210  *
3211  * If a read request arrives for a block that has a read in-progress,
3212  * either wait for the in-progress read to complete (and return the
3213  * results); or, if this is a read with a "done" func, add a record
3214  * to the read to invoke the "done" func when the read completes,
3215  * and return; or just return.
3216  *
3217  * arc_read_done() will invoke all the requested "done" functions
3218  * for readers of this block.
```

```
 */
int
arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
    void *private, zio_priority_t priority, int zio_flags, uint32_t *arc_flags,
    const zbookmark_t *zb)
{
        arc_buf_hdr_t *hdr;
        arc_buf_t *buf = NULL;
        kmutex_t *hash_lock;
        zio_t *rzio;
        uint64_t guid = spa_load_guid(spa);

top:
        hdr = buf_hash_find(guid, BP_IDENTITY(bp), BP_PHYSICAL_BIRTH(bp),
            &hash_lock);
        if (hdr && hdr->b_datacnt > 0) {

                *arc_flags |= ARC_CACHED;

                if (HDR_IO_IN_PROGRESS(hdr)) {

                        if (*arc_flags & ARC_WAIT) {
                                cv_wait(&hdr->b_cv, hash_lock);
                                mutex_exit(hash_lock);
                                goto top;
                        }
                        ASSERT(*arc_flags & ARC_NOWAIT);

                        if (done) {
                                arc_callback_t  *acb = NULL;

                                acb = kmem_zalloc(sizeof (arc_callback_t),
                                    KM_SLEEP);
                                acb->acb_done = done;
                                acb->acb_private = private;
                                if (pio != NULL)
                                        acb->acb_zio_dummy = zio_null(pio,
                                            spa, NULL, NULL, NULL, zio_flags);

                                ASSERT(acb->acb_done != NULL);
                                acb->acb_next = hdr->b_acb;
                                hdr->b_acb = acb;
                                add_reference(hdr, hash_lock, private);
                                mutex_exit(hash_lock);
                                return (0);
                        }
                        mutex_exit(hash_lock);
                        return (0);
                }

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

                if (done) {
                        add_reference(hdr, hash_lock, private);
                        /*
                         * If this block is already in use, create a new
                         * copy of the data so that we will be guaranteed
                         * that arc_release() will always succeed.
                         */
                        buf = hdr->b_buf;
                        ASSERT(buf);
                        ASSERT(buf->b_data);
                        if (HDR_BUF_AVAILABLE(hdr)) {
                                ASSERT(buf->b_efunc == NULL);
                                hdr->b_flags &= ~ARC_BUF_AVAILABLE;
                        } else {
```



```
                                buf = arc_buf_clone(buf);
                        }

                } else if (*arc_flags & ARC_PREFETCH &&
                    refcount_count(&hdr->b_refcnt) == 0) {
                        hdr->b_flags |= ARC_PREFETCH;
                }
                DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
                arc_access(hdr, hash_lock);
                if (*arc_flags & ARC_L2CACHE)
                        hdr->b_flags |= ARC_L2CACHE;
                if (*arc_flags & ARC_L2COMPRESS)
                        hdr->b_flags |= ARC_L2COMPRESS;
                mutex_exit(hash_lock);
                ARCSTAT_BUMP(arcstat_hits);
                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
                    data, metadata, hits);

                if (done)
                        done(NULL, buf, private);
        } else {
                uint64_t size = BP_GET_LSIZE(bp);
                arc_callback_t  *acb;
                vdev_t *vd = NULL;
                uint64_t addr = 0;
                boolean_t devw = B_FALSE;
                enum zio_compress b_compress = ZIO_COMPRESS_OFF;
                uint64_t b_asize = 0;

                if (hdr == NULL) {
                        /* this block is not in the cache */
                        arc_buf_hdr_t   *exists;
                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
                        buf = arc_buf_alloc(spa, size, private, type);
                        hdr = buf->b_hdr;
                        hdr->b_dva = *BP_IDENTITY(bp);
                        hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
                        hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
                        exists = buf_hash_insert(hdr, &hash_lock);
                        if (exists) {
                                /* somebody beat us to the hash insert */
                                mutex_exit(hash_lock);
                                buf_discard_identity(hdr);
                                (void) arc_buf_remove_ref(buf, private);
                                goto top; /* restart the IO request */
                        }
                        /* if this is a prefetch, we don't have a reference */
                        if (*arc_flags & ARC_PREFETCH) {
                                (void) remove_reference(hdr, hash_lock,
                                    private);
                                hdr->b_flags |= ARC_PREFETCH;
                        }
                        if (*arc_flags & ARC_L2CACHE)
                                hdr->b_flags |= ARC_L2CACHE;
                        if (*arc_flags & ARC_L2COMPRESS)
                                hdr->b_flags |= ARC_L2COMPRESS;
                        if (BP_GET_LEVEL(bp) > 0)
                                hdr->b_flags |= ARC_INDIRECT;
                } else {
                        /* this block is in the ghost cache */
                        ASSERT(GHOST_STATE(hdr->b_state));
                        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
                        ASSERT0(refcount_count(&hdr->b_refcnt));
                        ASSERT(hdr->b_buf == NULL);
```

```
3351                        /* if this is a prefetch, we don't have a reference */
3352                        if (*arc_flags & ARC_PREFETCH)
3353                                hdr->b_flags |= ARC_PREFETCH;
3354                        else
3355                                add_reference(hdr, hash_lock, private);
3356                        if (*arc_flags & ARC_L2CACHE)
3357                                hdr->b_flags |= ARC_L2CACHE;
3358                        if (*arc_flags & ARC_L2COMPRESS)
3359                                hdr->b_flags |= ARC_L2COMPRESS;
3360                        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
3361                        buf->b_hdr = hdr;
3362                        buf->b_data = NULL;
3363                        buf->b_efunc = NULL;
3364                        buf->b_private = NULL;
3365                        buf->b_next = NULL;
3366                        hdr->b_buf = buf;
3367                        ASSERT(hdr->b_datacnt == 0);
3368                        hdr->b_datacnt = 1;
3369                        arc_get_data_buf(buf);
3370                        arc_access(hdr, hash_lock);
3371                }

3373                ASSERT(!GHOST_STATE(hdr->b_state));

3375                acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
3376                acb->acb_done = done;
3377                acb->acb_private = private;

3379                ASSERT(hdr->b_acb == NULL);
3380                hdr->b_acb = acb;
3381                hdr->b_flags |= ARC_IO_IN_PROGRESS;

3383                if (hdr->b_l2hdr != NULL &&
3384                    (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
3385                        /*
3386                         * Need to stash these before letting go of hash_lock
3387                         */
3388                        devw = hdr->b_l2hdr->b_dev->l2ad_writing;
3389                        addr = hdr->b_l2hdr->b_daddr;
3390                        b_compress = hdr->b_l2hdr->b_compress;
3391                        b_asize = hdr->b_l2hdr->b_asize;
3392                        /*
3393                         * Lock out device removal.
3394                         */
3395                        if (vdev_is_dead(vd) ||
3396                            !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
3397                                vd = NULL;
3398                }

3400                mutex_exit(hash_lock);

3402                /*
3403                 * At this point, we have a level 1 cache miss.  Try again in
3404                 * L2ARC if possible.
3405                 */
3406                ASSERT3U(hdr->b_size, ==, size);
3407                DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
3408                    uint64_t, size, zbookmark_t *, zb);
3409                ARCSTAT_BUMP(arcstat_misses);
3410                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
3411                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
3412                    data, metadata, misses);

3414                if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
3415                        /*
3416                         * Read from the L2ARC if the following are true:
```



```
3417                         * 1. The L2ARC vdev was previously cached.
3418                         * 2. This buffer still has L2ARC metadata.
3419                         * 3. This buffer isn't currently writing to the L2ARC.
3420                         * 4. The L2ARC entry wasn't evicted, which may
3421                         *    also have invalidated the vdev.
3422                         * 5. This isn't prefetch and l2arc_noprefetch is set.
3423                         */
3424                        if (hdr->b_l2hdr != NULL &&
3425                            !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
3426                            !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
3427                                l2arc_read_callback_t *cb;

3429                                DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
3430                                ARCSTAT_BUMP(arcstat_l2_hits);

3432                                cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
3433                                    KM_SLEEP);
3434                                cb->l2rcb_buf = buf;
3435                                cb->l2rcb_spa = spa;
3436                                cb->l2rcb_bp = *bp;
3437                                cb->l2rcb_zb = *zb;
3438                                cb->l2rcb_flags = zio_flags;
3439                                cb->l2rcb_compress = b_compress;

3441                                ASSERT(addr >= VDEV_LABEL_START_SIZE &&
3442                                    addr + size < vd->vdev_psize -
3443                                    VDEV_LABEL_END_SIZE);

3445                                /*
3446                                 * l2arc read.  The SCL_L2ARC lock will be
3447                                 * released by l2arc_read_done().
3448                                 * Issue a null zio if the underlying buffer
3449                                 * was squashed to zero size by compression.
3450                                 */
3451                                if (b_compress == ZIO_COMPRESS_EMPTY) {
3452                                        rzio = zio_null(pio, spa, vd,
3453                                            l2arc_read_done, cb,
3454                                            zio_flags | ZIO_FLAG_DONT_CACHE |
3455                                            ZIO_FLAG_CANFAIL |
3456                                            ZIO_FLAG_DONT_PROPAGATE |
3457                                            ZIO_FLAG_DONT_RETRY);
3458                                } else {
3459                                        rzio = zio_read_phys(pio, vd, addr,
3460                                            b_asize, buf->b_data,
3461                                            ZIO_CHECKSUM_OFF,
3462                                            l2arc_read_done, cb, priority,
3463                                            zio_flags | ZIO_FLAG_DONT_CACHE |
3464                                            ZIO_FLAG_CANFAIL |
3465                                            ZIO_FLAG_DONT_PROPAGATE |
3466                                            ZIO_FLAG_DONT_RETRY, B_FALSE);
3467                                }
3468                                DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
3469                                    zio_t *, rzio);
3470                                ARCSTAT_INCR(arcstat_l2_read_bytes, b_asize);

3472                                if (*arc_flags & ARC_NOWAIT) {
3473                                        zio_nowait(rzio);
3474                                        return (0);
3475                                }

3477                                ASSERT(*arc_flags & ARC_WAIT);
3478                                if (zio_wait(rzio) == 0)
3479                                        return (0);

3481                                /* l2arc read error; goto zio_read() */
3482                        } else {
```

```
3483                                    DTRACE_PROBE1(l2arc__miss,
3484                                        arc_buf_hdr_t *, hdr);
3485                                    ARCSTAT_BUMP(arcstat_l2_misses);
3486                                    if (HDR_L2_WRITING(hdr))
3487                                            ARCSTAT_BUMP(arcstat_l2_rw_clash);
3488                                    spa_config_exit(spa, SCL_L2ARC, vd);
3489                            }
3490                    } else {
3491                            if (vd != NULL)
3492                                    spa_config_exit(spa, SCL_L2ARC, vd);
3493                            if (l2arc_ndev != 0) {
3494                                    DTRACE_PROBE1(l2arc__miss,
3495                                        arc_buf_hdr_t *, hdr);
3496                                    ARCSTAT_BUMP(arcstat_l2_misses);
3497                            }
3498                    }

3500                    rzio = zio_read(pio, spa, bp, buf->b_data, size,
3501                        arc_read_done, buf, priority, zio_flags, zb);

3503                    if (*arc_flags & ARC_WAIT)
3504                            return (zio_wait(rzio));

3506                    ASSERT(*arc_flags & ARC_NOWAIT);
3507                    zio_nowait(rzio);
3508            }
3509            return (0);
3510    }
_____unchanged_portion_omitted_

3637    /*
3638     * Release this buffer from the cache, making it an anonymous buffer.  This
3639     * must be done after a read and prior to modifying the buffer contents.
3640     * If the buffer has more than one reference, we must make
3641     * a new hdr for the buffer.
3642     */
3643    void
3644    arc_release(arc_buf_t *buf, void *tag)
3645    {
3646            arc_buf_hdr_t *hdr;
3647            kmutex_t *hash_lock = NULL;
3648            l2arc_buf_hdr_t *l2hdr;
3649            uint64_t buf_size;

3651            /*
3652             * It would be nice to assert that if it's DMU metadata (level >
3653             * 0 || it's the dnode file), then it must be syncing context.
3654             * But we don't know that information at this level.
3655             */

3657            mutex_enter(&buf->b_evict_lock);
3658            hdr = buf->b_hdr;

3660            /* this buffer is not on any list */
3661            ASSERT(refcount_count(&hdr->b_refcnt) > 0);

3663            if (hdr->b_state == arc_anon) {
3664                    /* this buffer is already released */
3665                    ASSERT(buf->b_efunc == NULL);
3666            } else {
3667                    hash_lock = HDR_LOCK(hdr);
3668                    mutex_enter(hash_lock);
3669                    hdr = buf->b_hdr;
3670                    ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
3671            }
```



```
3673            l2hdr = hdr->b_l2hdr;
3674            if (l2hdr) {
3675                    mutex_enter(&l2arc_buflist_mtx);
3676                    hdr->b_l2hdr = NULL;
3677                    list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
3678            }
3679            buf_size = hdr->b_size;

3681            /*
3682             * Do we have more than one buf?
3683             */
3684            if (hdr->b_datacnt > 1) {
3685                    arc_buf_hdr_t *nhdr;
3686                    arc_buf_t **bufp;
3687                    uint64_t blksz = hdr->b_size;
3688                    uint64_t spa = hdr->b_spa;
3689                    arc_buf_contents_t type = hdr->b_type;
3690                    uint32_t flags = hdr->b_flags;

3692                    ASSERT(hdr->b_buf != buf || buf->b_next != NULL);
3693                    /*
3694                     * Pull the data off of this hdr and attach it to
3695                     * a new anonymous hdr.
3696                     */
3697                    (void) remove_reference(hdr, hash_lock, tag);
3698                    bufp = &hdr->b_buf;
3699                    while (*bufp != buf)
3700                            bufp = &(*bufp)->b_next;
3701                    *bufp = buf->b_next;
3702                    buf->b_next = NULL;

3704                    ASSERT3U(hdr->b_state->arcs_size, >=, hdr->b_size);
3705                    atomic_add_64(&hdr->b_state->arcs_size, -hdr->b_size);
3706                    if (refcount_is_zero(&hdr->b_refcnt)) {
3707                            uint64_t *size = &hdr->b_state->arcs_lsize[hdr->b_type];
3708                            ASSERT3U(*size, >=, hdr->b_size);
3709                            atomic_add_64(size, -hdr->b_size);
3710                    }

3712                    /*
3713                     * We're releasing a duplicate user data buffer, update
3714                     * our statistics accordingly.
3715                     */
3716                    if (hdr->b_type == ARC_BUFC_DATA) {
3717                            ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
3718                            ARCSTAT_INCR(arcstat_duplicate_buffers_size,
3719                                -hdr->b_size);
3720                    }
3721                    hdr->b_datacnt -= 1;
3722                    arc_cksum_verify(buf);
3723                    arc_buf_unwatch(buf);

3725                    mutex_exit(hash_lock);

3727                    nhdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
3728                    nhdr->b_size = blksz;
3729                    nhdr->b_spa = spa;
3730                    nhdr->b_type = type;
3731                    nhdr->b_buf = buf;
3732                    nhdr->b_state = arc_anon;
3733                    nhdr->b_arc_access = 0;
3734                    nhdr->b_flags = flags & ARC_L2_WRITING;
3735                    nhdr->b_l2hdr = NULL;
3736                    nhdr->b_datacnt = 1;
3737                    nhdr->b_freeze_cksum = NULL;
3738                    (void) refcount_add(&nhdr->b_refcnt, tag);
```

```
3739                 buf->b_hdr = nhdr;
3740                 mutex_exit(&buf->b_evict_lock);
3741                 atomic_add_64(&arc_anon->arcs_size, blksz);
3742         } else {
3743                 mutex_exit(&buf->b_evict_lock);
3744                 ASSERT(refcount_count(&hdr->b_refcnt) == 1);
3745                 ASSERT(!list_link_active(&hdr->b_arc_node));
3746                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
3747                 if (hdr->b_state != arc_anon)
3748                         arc_change_state(arc_anon, hdr, hash_lock);
3749                 hdr->b_arc_access = 0;
3750                 if (hash_lock)
3751                         mutex_exit(hash_lock);

3753                 buf_discard_identity(hdr);
3754                 arc_buf_thaw(buf);
3755         }
3756         buf->b_efunc = NULL;
3757         buf->b_private = NULL;

3759         if (l2hdr) {
3760                 ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
3761                 kmem_free(l2hdr, sizeof (*l2hdr));
3421                 kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
3762                 ARCSTAT_INCR(arcstat_l2_size, -buf_size);
3763                 mutex_exit(&l2arc_buflist_mtx);
3764         }
3765 }
```

*_____unchanged_portion_omitted_*

```
4233 /*
4234  * Level 2 ARC
4235  *
4236  * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
4237  * It uses dedicated storage devices to hold cached data, which are populated
4238  * using large infrequent writes.  The main role of this cache is to boost
4239  * the performance of random read workloads.  The intended L2ARC devices
4240  * include short-stroked disks, solid state disks, and other media with
4241  * substantially faster read latency than disk.
4242  *
4243  *                 +-----------------------+
4244  *                 |         ARC           |
4245  *                 +-----------------------+
4246  *                    |         ^     ^
4247  *                    |         |     |
4248  *      l2arc_feed_thread()    arc_read()
4249  *                    |         |     |
4250  *                    |  l2arc read   |
4251  *                    V         |     |
4252  *               +---------------+    |
4253  *               |     L2ARC     |    |
4254  *               +---------------+    |
4255  *                   |    ^           |
4256  *          l2arc_write() |           |
4257  *                   |    |           |
4258  *                   V    |           |
4259  *                 +-------+      +-------+
4260  *                 | vdev  |      | vdev  |
4261  *                 | cache |      | cache |
4262  *                 +-------+      +-------+
4263  *                 +=========+     .-----.
4264  *                 :  L2ARC  :    |-_____-|
4265  *                 : devices :    | Disks |
4266  *                 +=========+    `-_____-'
4267  *
4268  * Read requests are satisfied from the following sources, in order:
```



```
4269  *
4270  *      1) ARC
4271  *      2) vdev cache of L2ARC devices
4272  *      3) L2ARC devices
4273  *      4) vdev cache of disks
4274  *      5) disks
4275  *
4276  * Some L2ARC device types exhibit extremely slow write performance.
4277  * To accommodate for this there are some significant differences between
4278  * the L2ARC and traditional cache design:
4279  *
4280  * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
4281  * the ARC behave as usual, freeing buffers and placing headers on ghost
4282  * lists.  The ARC does not send buffers to the L2ARC during eviction as
4283  * this would add inflated write latencies for all ARC memory pressure.
4284  *
4285  * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
4286  * It does this by periodically scanning buffers from the eviction-end of
4287  * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
4288  * not already there. It scans until a headroom of buffers is satisfied,
4289  * which itself is a buffer for ARC eviction. If a compressible buffer is
4290  * found during scanning and selected for writing to an L2ARC device, we
4291  * temporarily boost scanning headroom during the next scan cycle to make
4292  * sure we adapt to compression effects (which might significantly reduce
4293  * the data volume we write to L2ARC). The thread that does this is
4294  * l2arc_feed_thread(), illustrated below; example sizes are included to
4295  * provide a better sense of ratio than this diagram:
4296  *
4297  *             head -->                        tail
4298  *              +---------------------+----------+
4299  *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
4300  *              +---------------------+----------+   |   o L2ARC eligible
4301  *      ARC_mru |:#::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
4302  *              +---------------------+----------+   |
4303  *                   15.9 Gbytes      ^ 32 Mbytes    |
4304  *                                  headroom          |
4305  *                                             l2arc_feed_thread()
4306  *                                                    |
4307  *                         l2arc write hand <--[oooo]--'
4308  *                                 |           8 Mbyte
4309  *                                 |          write max
4310  *                                 V
4311  *               +==============================+
4312  *      L2ARC dev |####|#|###|###|    |####| ... |
4313  *                +==============================+
4314  *                            32 Gbytes
4315  *
4316  * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
4317  * evicted, then the L2ARC has cached a buffer much sooner than it probably
4318  * needed, potentially wasting L2ARC device bandwidth and storage.  It is
4319  * safe to say that this is an uncommon case, since buffers at the end of
4320  * the ARC lists have moved there due to inactivity.
4321  *
4322  * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
4323  * then the L2ARC simply misses copying some buffers.  This serves as a
4324  * pressure valve to prevent heavy read workloads from both stalling the ARC
4325  * with waits and clogging the L2ARC with writes.  This also helps prevent
4326  * the potential for the L2ARC to churn if it attempts to cache content too
4327  * quickly, such as during backups of the entire pool.
4328  *
4329  * 5. After system boot and before the ARC has filled main memory, there are
4330  * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
4331  * lists can remain mostly static.  Instead of searching from tail of these
4332  * lists as pictured, the l2arc_feed_thread() will search from the list heads
4333  * for eligible buffers, greatly increasing its chance of finding them.
4334  *
```

```
4335  * The L2ARC device write speed is also boosted during this time so that
4336  * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
4337  * there are no L2ARC reads, and no fear of degrading read performance
4338  * through increased writes.
4339  *
4340  * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
4341  * the vdev queue can aggregate them into larger and fewer writes.  Each
4342  * device is written to in a rotor fashion, sweeping writes through
4343  * available space then repeating.
4344  *
4345  * 7. The L2ARC does not store dirty content.  It never needs to flush
4346  * write buffers back to disk based storage.
4347  *
4348  * 8. If an ARC buffer is written (and dirtied) which also exists in the
4349  * L2ARC, the now stale L2ARC buffer is immediately dropped.
4350  *
4351  * The performance of the L2ARC can be tweaked by a number of tunables, which
4352  * may be necessary for different workloads:
4353  *
4354  *      l2arc_write_max         max write bytes per interval
4355  *      l2arc_write_boost       extra write bytes during device warmup
4356  *      l2arc_noprefetch        skip caching prefetched buffers
4357  *      l2arc_headroom          number of max device writes to precache
4358  *      l2arc_headroom_boost    when we find compressed buffers during ARC
4359  *                              scanning, we multiply headroom by this
4360  *                              percentage factor for the next scan cycle,
4361  *                              since more compressed buffers are likely to
4362  *                              be present
4363  *      l2arc_feed_secs         seconds between L2ARC writing
4364  *
4365  * Tunables may be removed or added as future performance improvements are
4366  * integrated, and also may become zpool properties.
4367  *
4368  * There are three key functions that control how the L2ARC warms up:
4369  *
4370  *      l2arc_write_eligible()  check if a buffer is eligible to cache
4371  *      l2arc_write_size()      calculate how much to write
4372  *      l2arc_write_interval()  calculate sleep delay between writes
4373  *
4374  * These three functions determine what to write, how much, and how quickly
4375  * to send writes.
4376  *
4377  * L2ARC persistency:
4378  *
4379  * When writing buffers to L2ARC, we periodically add some metadata to
4380  * make sure we can pick them up after reboot, thus dramatically reducing
4381  * the impact that any downtime has on the performance of storage systems
4382  * with large caches.
4383  *
4384  * The implementation works fairly simply by integrating the following two
4385  * modifications:
4386  *
4387  * *) Every now and then we mix in a piece of metadata (called a log block)
4388  *    into the L2ARC write. This allows us to understand what's been written,
4389  *    so that we can rebuild the arc_buf_hdr_t structures of the main ARC
4390  *    buffers. The log block also includes a "back-reference" pointer to the
4391  *    previous block, forming a back-linked list of blocks on the L2ARC device.
4392  *
4393  * *) We reserve SPA_MINBLOCKSIZE of space at the start of each L2ARC device
4394  *    for our header bookkeeping purposes. This contains a device header, which
4395  *    contains our top-level reference structures. We update it each time we
4396  *    write a new log block, so that we're able to locate it in the L2ARC
4397  *    device. If this write results in an inconsistent device header (e.g. due
4398  *    to power failure), we detect this by verifying the header's checksum
4399  *    and simply drop the entries from L2ARC.
4400  *
```



```
4401  * Implementation diagram:
4402  *
4403  * +=== L2ARC device (not to scale) ======================================+
4404  * |       ___________newest log block pointers_________                  |
4405  * |      /                                  \1 back   \latest            |
4406  * |     /                                    V        V                 |
4407  * ||L2 dev hdr |---|bufs |lb |bufs |lb |bufs |lb |bufs |lb |---(empty)---|
4408  * |                       ^       / ^       / ^       /                  |
4409  * |                       `-prev-'  `-prev-'  `-prev-'                   |
4410  * |                         lb        lb        lb                       |
4411  * +======================================================================+
4412  *
4413  * On-device data structures:
4414  *
4415  * L2ARC device header: l2arc_dev_hdr_phys_t
4416  * L2ARC log block:     l2arc_log_blk_phys_t
4417  *
4418  * L2ARC reconstruction:
4419  *
4420  * When writing data, we simply write in the standard rotary fashion,
4421  * evicting buffers as we go and simply writing new data over them (writing
4422  * a new log block every now and then). This obviously means that once we
4423  * loop around the end of the device, we will start cutting into an already
4424  * committed log block (and its referenced data buffers), like so:
4425  *
4426  *    current write head__       __old tail
4427  *                        \     /
4428  *                        V    V
4429  * <--|bufs |lb |bufs |lb |    |bufs |lb |bufs |lb |-->
4430  *                         ^    ^^^^^^^^^____________________________________
4431  *                         |                                                \
4432  *                   <<nextwrite>> may overwrite this blk and/or its bufs --'
4433  *
4434  * When importing the pool, we detect this situation and use it to stop
4435  * our scanning process (see l2arc_rebuild).
4436  *
4437  * There is one significant caveat to consider when rebuilding ARC contents
4438  * from an L2ARC device: what about invalidated buffers? Given the above
4439  * construction, we cannot update blocks which we've already written to amend
4440  * them to remove buffers which were invalidated. Thus, during reconstruction,
4441  * we might be populating the cache with buffers for data that's not on the
4442  * main pool anymore, or may have been overwritten!
4443  *
4444  * As it turns out, this isn't a problem. Every arc_read request includes
4445  * both the DVA and, crucially, the birth TXG of the BP the caller is
4446  * looking for. So even if the cache were populated by completely rotten
4447  * blocks for data that had been long deleted and/or overwritten, we'll
4448  * never actually return bad data from the cache, since the DVA with the
4449  * birth TXG uniquely identify a block in space and time - once created,
4450  * a block is immutable on disk. The worst thing we have done is wasted
4451  * some time and memory at l2arc rebuild to reconstruct outdated ARC
4452  * entries that will get dropped from the l2arc as it is being updated
4453  * with new blocks.
4454  */

4456 static boolean_t
4457 l2arc_write_eligible(uint64_t spa_guid, arc_buf_hdr_t *ab)
4458 {
4459         /*
4460          * A buffer is *not* eligible for the L2ARC if it:
4461          * 1. belongs to a different spa.
4462          * 2. is already cached on the L2ARC.
4463          * 3. has an I/O in progress (it may be an incomplete read).
4464          * 4. is flagged not eligible (zfs property).
4465          */
4466         if (ab->b_spa != spa_guid || ab->b_l2hdr != NULL ||
```

```
4467             HDR_IO_IN_PROGRESS(ab) || !HDR_L2CACHE(ab))
4468                 return (B_FALSE);

4470         return (B_TRUE);
4471 }
______unchanged_portion_omitted_

4519 static void
4520 l2arc_hdr_stat_add(boolean_t from_arc)
4102 l2arc_hdr_stat_add(void)
4521 {
4522         ARCSTAT_INCR(arcstat_l2_hdr_size, HDR_SIZE + L2HDR_SIZE);
4523         if (from_arc)
4524                 ARCSTAT_INCR(arcstat_hdr_size, -HDR_SIZE);
4525 }
______unchanged_portion_omitted_

4534 /*
4535  * Cycle through L2ARC devices.  This is how L2ARC load balances.
4536  * If a device is returned, this also returns holding the spa config lock.
4537  */
4538 static l2arc_dev_t *
4539 l2arc_dev_get_next(void)
4540 {
4541         l2arc_dev_t *first, *next = NULL;

4543         /*
4544          * Lock out the removal of spas (spa_namespace_lock), then removal
4545          * of cache devices (l2arc_dev_mtx).  Once a device has been selected,
4546          * both locks will be dropped and a spa config lock held instead.
4547          */
4548         mutex_enter(&spa_namespace_lock);
4549         mutex_enter(&l2arc_dev_mtx);

4551         /* if there are no vdevs, there is nothing to do */
4552         if (l2arc_ndev == 0)
4553                 goto out;

4555         first = NULL;
4556         next = l2arc_dev_last;
4557         do {
4558                 /*
4559                  * Loop around the list looking for a non-faulted vdev
4560                  * and one that isn't currently doing an L2ARC rebuild.
4561                  */
4139                 /* loop around the list looking for a non-faulted vdev */
4562                 if (next == NULL) {
4563                         next = list_head(l2arc_dev_list);
4564                 } else {
4565                         next = list_next(l2arc_dev_list, next);
4566                         if (next == NULL)
4567                                 next = list_head(l2arc_dev_list);
4568                 }

4570                 /* if we have come back to the start, bail out */
4571                 if (first == NULL)
4572                         first = next;
4573                 else if (next == first)
4574                         break;

4576         } while (vdev_is_dead(next->l2ad_vdev) || next->l2ad_rebuild);
4154         } while (vdev_is_dead(next->l2ad_vdev));

4578         /* if we were unable to find any usable vdevs, return NULL */
4579         if (vdev_is_dead(next->l2ad_vdev) || next->l2ad_rebuild)
4157         if (vdev_is_dead(next->l2ad_vdev))
```



```
4580                 next = NULL;

4582         l2arc_dev_last = next;

4584 out:
4585         mutex_exit(&l2arc_dev_mtx);

4587         /*
4588          * Grab the config lock to prevent the 'next' device from being
4589          * removed while we are writing to it.
4590          */
4591         if (next != NULL)
4592                 spa_config_enter(next->l2ad_spa, SCL_L2ARC, next, RW_READER);
4593         mutex_exit(&spa_namespace_lock);

4595         return (next);
4596 }
______unchanged_portion_omitted_

4622 /*
4623  * A write to a cache device has completed.  Update all headers to allow
4624  * reads from these buffers to begin.
4625  */
4626 static void
4627 l2arc_write_done(zio_t *zio)
4628 {
4629         l2arc_write_callback_t *cb;
4630         l2arc_dev_t *dev;
4631         list_t *buflist;
4632         arc_buf_hdr_t *head, *ab, *ab_prev;
4633         l2arc_buf_hdr_t *l2hdr;
4211         l2arc_buf_hdr_t *abl2;
4634         kmutex_t *hash_lock;
4635         l2arc_log_blk_buf_t *lb_buf;

4637         cb = zio->io_private;
4638         ASSERT(cb != NULL);
4639         dev = cb->l2wcb_dev;
4640         ASSERT(dev != NULL);
4641         head = cb->l2wcb_head;
4642         ASSERT(head != NULL);
4643         buflist = dev->l2ad_buflist;
4644         ASSERT(buflist != NULL);
4645         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4646             l2arc_write_callback_t *, cb);

4648         if (zio->io_error != 0)
4649                 ARCSTAT_BUMP(arcstat_l2_writes_error);

4651         mutex_enter(&l2arc_buflist_mtx);

4653         /*
4654          * All writes completed, or an error was hit.
4655          */
4656         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4657                 ab_prev = list_prev(buflist, ab);
4658                 l2hdr = ab->b_l2hdr;

4660                 /*
4661                  * Release the temporary compressed buffer as soon as possible.
4662                  */
4663                 if (l2hdr->b_compress != ZIO_COMPRESS_OFF)
4664                         l2arc_release_cdata_buf(ab);

4666                 hash_lock = HDR_LOCK(ab);
4667                 if (!mutex_tryenter(hash_lock)) {
```

```
4668                        /*
4669                         * This buffer misses out.  It may be in a stage
4670                         * of eviction.  Its ARC_L2_WRITING flag will be
4671                         * left set, denying reads to this buffer.
4672                         */
4673                        ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
4674                        continue;
4675                }

4247                abl2 = ab->b_l2hdr;

4249                /*
4250                 * Release the temporary compressed buffer as soon as possible.
4251                 */
4252                if (abl2->b_compress != ZIO_COMPRESS_OFF)
4253                        l2arc_release_cdata_buf(ab);

4677                if (zio->io_error != 0) {
4678                        /*
4679                         * Error - drop L2ARC entry.
4680                         */
4681                        list_remove(buflist, ab);
4682                        ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4260                        ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4683                        ab->b_l2hdr = NULL;
4684                        kmem_free(l2hdr, sizeof (*l2hdr));
4262                        kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4685                        ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4686                }

4688                /*
4689                 * Allow ARC to begin reads to this L2ARC entry.
4690                 */
4691                ab->b_flags &= ~ARC_L2_WRITING;

4693                mutex_exit(hash_lock);
4694        }

4696        atomic_inc_64(&l2arc_writes_done);
4697        list_remove(buflist, head);
4698        kmem_cache_free(hdr_cache, head);
4699        mutex_exit(&l2arc_buflist_mtx);

4701        l2arc_do_free_on_write();

4703        for (lb_buf = list_tail(&cb->l2wcb_log_blk_buf_list); lb_buf != NULL;
4704            lb_buf = list_tail(&cb->l2wcb_log_blk_buf_list)) {
4705                (void) list_remove_tail(&cb->l2wcb_log_blk_buf_list);
4706                kmem_free(lb_buf, sizeof (*lb_buf));
4707        }
4708        list_destroy(&cb->l2wcb_log_blk_buf_list);
4709        kmem_free(cb, sizeof (l2arc_write_callback_t));
4710 }
______unchanged_portion_omitted_

4831 /*
4832  * Calculates the maximum overhead of L2ARC metadata log blocks for a given
4833  * L2ARC write size. l2arc_evict and l2arc_write_buffers need to include this
4834  * overhead in processing to make sure there is enough headroom available
4835  * when writing buffers.
4836  */
4837 static inline uint64_t
4838 l2arc_log_blk_overhead(uint64_t write_sz)
4839 {
4840        return ((write_sz / SPA_MINBLOCKSIZE / L2ARC_LOG_BLK_ENTRIES) + 1) *
4841            L2ARC_LOG_BLK_SIZE;
```



```
4842 }

4844 /*
4845  * Evict buffers from the device write hand to the distance specified in
4846  * bytes.  This distance may span populated buffers, it may span nothing.
4847  * This is clearing a region on the L2ARC device ready for writing.
4848  * If the 'all' boolean is set, every buffer is evicted.
4849  */
4850 static void
4851 l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
4852 {
4853        list_t *buflist;
4854        l2arc_buf_hdr_t *l2hdr;
4413        l2arc_buf_hdr_t *abl2;
4855        arc_buf_hdr_t *ab, *ab_prev;
4856        kmutex_t *hash_lock;
4857        uint64_t taddr;

4859        buflist = dev->l2ad_buflist;

4861        if (buflist == NULL)
4862                return;

4864        if (!all && dev->l2ad_first) {
4865                /*
4866                 * This is the first sweep through the device.  There is
4867                 * nothing to evict.
4868                 */
4869                return;
4870        }

4872        /*
4873         * We need to add in the worst case scenario of log block overhead.
4874         */
4875        distance += l2arc_log_blk_overhead(distance);
4876        if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
4877                /*
4878                 * When nearing the end of the device, evict to the end
4879                 * before the device write hand jumps to the start.
4880                 */
4881                taddr = dev->l2ad_end;
4882        } else {
4883                taddr = dev->l2ad_hand + distance;
4884        }
4885        DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
4886            uint64_t, taddr, boolean_t, all);

4888 top:
4889        mutex_enter(&l2arc_buflist_mtx);
4890        for (ab = list_tail(buflist); ab; ab = ab_prev) {
4891                ab_prev = list_prev(buflist, ab);

4893                hash_lock = HDR_LOCK(ab);
4894                if (!mutex_tryenter(hash_lock)) {
4895                        /*
4896                         * Missed the hash lock.  Retry.
4897                         */
4898                        ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
4899                        mutex_exit(&l2arc_buflist_mtx);
4900                        mutex_enter(hash_lock);
4901                        mutex_exit(hash_lock);
4902                        goto top;
4903                }

4905                if (HDR_L2_WRITE_HEAD(ab)) {
4906                        /*
```

```
4907                         * We hit a write head node.  Leave it for
4908                         * l2arc_write_done().
4909                         */
4910                        list_remove(buflist, ab);
4911                        mutex_exit(hash_lock);
4912                        continue;
4913                }

4915                if (!all && ab->b_l2hdr != NULL &&
4916                    (ab->b_l2hdr->b_daddr > taddr ||
4917                    ab->b_l2hdr->b_daddr < dev->l2ad_hand)) {
4918                        /*
4919                         * We've evicted to the target address,
4920                         * or the end of the device.
4921                         */
4922                        mutex_exit(hash_lock);
4923                        break;
4924                }

4926                if (HDR_FREE_IN_PROGRESS(ab)) {
4927                        /*
4928                         * Already on the path to destruction.
4929                         */
4930                        mutex_exit(hash_lock);
4931                        continue;
4932                }

4934                if (ab->b_state == arc_l2c_only) {
4935                        ASSERT(!HDR_L2_READING(ab));
4936                        /*
4937                         * This doesn't exist in the ARC.  Destroy.
4938                         * arc_hdr_destroy() will call list_remove()
4939                         * and decrement arcstat_l2_size.
4940                         */
4941                        arc_change_state(arc_anon, ab, hash_lock);
4942                        arc_hdr_destroy(ab);
4943                } else {
4944                        /*
4945                         * Invalidate issued or about to be issued
4946                         * reads, since we may be about to write
4947                         * over this location.
4948                         */
4949                        if (HDR_L2_READING(ab)) {
4950                                ARCSTAT_BUMP(arcstat_l2_evict_reading);
4951                                ab->b_flags |= ARC_L2_EVICTED;
4952                        }

4954                        /*
4955                         * Tell ARC this no longer exists in L2ARC.
4956                         */
4957                        if (ab->b_l2hdr != NULL) {
4958                                l2hdr = ab->b_l2hdr;
4959                                ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
4513                                abl2 = ab->b_l2hdr;
4514                                ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4960                                ab->b_l2hdr = NULL;
4961                                kmem_free(l2hdr, sizeof (*l2hdr));
4516                                kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4962                                ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4963                        }
4964                        list_remove(buflist, ab);

4966                        /*
4967                         * This may have been leftover after a
4968                         * failed write.
4969                         */
```



```
4970                                ab->b_flags &= ~ARC_L2_WRITING;
4971                        }
4972                        mutex_exit(hash_lock);
4973                }
4974                mutex_exit(&l2arc_buflist_mtx);

4976                vdev_space_update(dev->l2ad_vdev, -(taddr - dev->l2ad_evict), 0, 0);
4977                dev->l2ad_evict = taddr;
4978        }

4980        /*
4981         * Find and write ARC buffers to the L2ARC device.
4982         *
4983         * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4984         * for reading until they have completed writing.
4985         * The headroom_boost is an in-out parameter used to maintain headroom boost
4986         * state between calls to this function.
4987         *
4988         * Returns the number of bytes actually written (which may be smaller than
4989         * the delta by which the device hand has changed due to alignment).
4990         */
4991        static uint64_t
4992        l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
4993            boolean_t *headroom_boost)
4994        {
4995                arc_buf_hdr_t *ab, *ab_prev, *head;
4996                list_t *list;
4997                /*
4998                 * These variables mean:
4999                 * - write_size: in-memory size of ARC buffers we've written (before
5000                 *      compression).
5001                 * - write_asize: actual on-disk size of ARC buffers we've written
5002                 *      (after compression).
5003                 * - write_aligned_asize: actual sum of space taken by ARC buffers
5004                 *      on the device (after compression and alignment, so that
5005                 *      every buffer starts on a multiple of the device block size).
5006                 * - headroom: L2ARC scanning headroom (we won't scan beyond this
5007                 *      distance from the list tail).
5008                 * - buf_compress_minsz: minimum in-memory ARC buffer size for us
5009                 *      to try compressing it.
5010                 */
5011                uint64_t write_size, write_asize, write_aligned_asize, headroom,
4552                uint64_t write_asize, write_psize, write_sz, headroom,
5012                    buf_compress_minsz;
5013                void *buf_data;
5014                kmutex_t *list_lock;
5015                boolean_t full;
5016                l2arc_write_callback_t *cb;
5017                zio_t *pio, *wzio;
5018                uint64_t guid = spa_load_guid(spa);
5019                const boolean_t do_headroom_boost = *headroom_boost;
5020                boolean_t dev_hdr_update = B_FALSE;

5022                ASSERT(dev->l2ad_vdev != NULL);

5024                /* Lower the flag now, we might want to raise it again later. */
5025                *headroom_boost = B_FALSE;

5027                pio = NULL;
5028                cb = NULL;
5029                write_size = write_asize = write_aligned_asize = 0;
4568                write_sz = write_asize = write_psize = 0;
5030                full = B_FALSE;
5031                head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
5032                head->b_flags |= ARC_L2_WRITE_HEAD;
```

```
5034         /*
5035          * We will want to try to compress buffers that are at least 2x the
5036          * device sector size.
5037          */
5038         buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

5040         /*
5041          * Copy buffers for L2ARC writing.
5042          */
5043         mutex_enter(&l2arc_buflist_mtx);
5044         for (int try = 0; try <= 3; try++) {
5045                 uint64_t passed_sz = 0;

5047                 list = l2arc_list_locked(try, &list_lock);

5049                 /*
5050                  * L2ARC fast warmup.
5051                  *
5052                  * Until the ARC is warm and starts to evict, read from the
5053                  * head of the ARC lists rather than the tail.
5054                  */
5055                 if (arc_warm == B_FALSE)
5056                         ab = list_head(list);
5057                 else
5058                         ab = list_tail(list);

5060                 headroom = target_sz * l2arc_headroom;
5061                 if (do_headroom_boost)
5062                         headroom = (headroom * l2arc_headroom_boost) / 100;

5064                 for (; ab; ab = ab_prev) {
5065                         l2arc_buf_hdr_t *l2hdr;
5066                         kmutex_t *hash_lock;
5067                         uint64_t buf_aligned_size;
4606                         uint64_t buf_sz;

5069                         if (arc_warm == B_FALSE)
5070                                 ab_prev = list_next(list, ab);
5071                         else
5072                                 ab_prev = list_prev(list, ab);

5074                         hash_lock = HDR_LOCK(ab);
5075                         if (!mutex_tryenter(hash_lock)) {
5076                                 /*
5077                                  * Skip this buffer rather than waiting.
5078                                  */
5079                                 continue;
5080                         }

5082                         /*
5083                          * When examining whether we've met our write target,
5084                          * we must always use the aligned size of the buffer,
5085                          * since that's the maximum amount of space a buffer
5086                          * can take up on the L2ARC device.
5087                          */
5088                         buf_aligned_size = vdev_psize_to_asize(dev->l2ad_vdev,
5089                             ab->b_size);
5090                         passed_sz += buf_aligned_size;
4621                         passed_sz += ab->b_size;
5091                         if (passed_sz > headroom) {
5092                                 /*
5093                                  * Searched too far.
5094                                  */
5095                                 mutex_exit(hash_lock);
5096                                 break;
5097                         }
```



```
5099                         if (!l2arc_write_eligible(guid, ab)) {
5100                                 mutex_exit(hash_lock);
5101                                 continue;
5102                         }

5104                         if ((write_size + buf_aligned_size) > target_sz) {
4635                         if ((write_sz + ab->b_size) > target_sz) {
5105                                 full = B_TRUE;
5106                                 mutex_exit(hash_lock);
5107                                 break;
5108                         }

5110                         if (pio == NULL) {
5111                                 /*
5112                                  * Insert a dummy header on the buflist so
5113                                  * l2arc_write_done() can find where the
5114                                  * write buffers begin without searching.
5115                                  */
5116                                 list_insert_head(dev->l2ad_buflist, head);

5118                                 cb = kmem_zalloc(
4649                                 cb = kmem_alloc(
5119                                     sizeof (l2arc_write_callback_t), KM_SLEEP);
5120                                 cb->l2wcb_dev = dev;
5121                                 cb->l2wcb_head = head;
5122                                 list_create(&cb->l2wcb_log_blk_buf_list,
5123                                     sizeof (l2arc_log_blk_buf_t),
5124                                     offsetof(l2arc_log_blk_buf_t, l2lbb_node));
5125                                 pio = zio_root(spa, l2arc_write_done, cb,
5126                                     ZIO_FLAG_CANFAIL);
5127                         }

5129                         /*
5130                          * Create and add a new L2ARC header.
5131                          */
5132                         l2hdr = kmem_zalloc(sizeof (*l2hdr), KM_SLEEP);
4660                         l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
5133                         l2hdr->b_dev = dev;
5134                         ab->b_flags |= ARC_L2_WRITING;

5136                         /*
5137                          * Temporarily stash the data buffer in b_tmp_cdata.
5138                          * The subsequent write step will pick it up from
5139                          * there. This is because can't access ab->b_buf
5140                          * without holding the hash_lock, which we in turn
5141                          * can't access without holding the ARC list locks
5142                          * (which we want to avoid during compression/writing).
5143                          */
5144                         l2hdr->b_compress = ZIO_COMPRESS_OFF;
5145                         l2hdr->b_asize = ab->b_size;
5146                         l2hdr->b_tmp_cdata = ab->b_buf->b_data;

4676                         buf_sz = ab->b_size;
5148                         ab->b_l2hdr = l2hdr;

5150                         list_insert_head(dev->l2ad_buflist, ab);

5152                         /*
5153                          * Compute and store the buffer cksum before
5154                          * writing.  On debug the cksum is verified first.
5155                          */
5156                         arc_cksum_verify(ab->b_buf);
5157                         arc_cksum_compute(ab->b_buf, B_TRUE);

5159                         mutex_exit(hash_lock);
```

```
5161                        write_size += buf_aligned_size;
4690                        write_sz += buf_sz;
5162                }

5164                mutex_exit(list_lock);

5166                if (full == B_TRUE)
5167                        break;
5168        }

5170        /* No buffers selected for writing? */
5171        if (pio == NULL) {
5172                ASSERT0(write_size);
4701                ASSERT0(write_sz);
5173                mutex_exit(&l2arc_buflist_mtx);
5174                kmem_cache_free(hdr_cache, head);
5175                return (0);
5176        }

5178        /*
5179         * Now start writing the buffers. We're starting at the write head
5180         * and work backwards, retracing the course of the buffer selector
5181         * loop above.
5182         */
5183        for (ab = list_prev(dev->l2ad_buflist, head); ab;
5184            ab = list_prev(dev->l2ad_buflist, ab)) {
5185                l2arc_buf_hdr_t *l2hdr;
5186                uint64_t buf_sz;

5188                /*
5189                 * We shouldn't need to lock the buffer here, since we flagged
5190                 * it as ARC_L2_WRITING in the previous step, but we must take
5191                 * care to only access its L2 cache parameters. In particular,
5192                 * ab->b_buf may be invalid by now due to ARC eviction.
5193                 */
5194                l2hdr = ab->b_l2hdr;
5195                l2hdr->b_daddr = dev->l2ad_hand;

5197                if ((ab->b_flags & ARC_L2COMPRESS) &&
5198                    l2hdr->b_asize >= buf_compress_minsz) {
5199                        if (l2arc_compress_buf(l2hdr)) {
5200                                /*
5201                                 * If compression succeeded, enable headroom
5202                                 * boost on the next scan cycle.
5203                                 */
5204                                *headroom_boost = B_TRUE;
5205                        }
5206                }

5208                /*
5209                 * Pick up the buffer data we had previously stashed away
5210                 * (and now potentially also compressed).
5211                 */
5212                buf_data = l2hdr->b_tmp_cdata;
5213                buf_sz = l2hdr->b_asize;

5215                /* Compression may have squashed the buffer to zero length. */
5216                if (buf_sz != 0) {
5217                        uint64_t buf_aligned_asize;
4746                        uint64_t buf_p_sz;

5219                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
5220                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
5221                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
5222                            ZIO_FLAG_CANFAIL, B_FALSE);
```



```
5224                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
5225                            zio_t *, wzio);
5226                        (void) zio_nowait(wzio);

5228                        write_asize += buf_sz;
5229                        /*
5230                         * Keep the clock hand suitably device-aligned.
5231                         */
5232                        buf_aligned_asize = vdev_psize_to_asize(dev->l2ad_vdev,
5233                            buf_sz);
5234                        write_aligned_asize += buf_aligned_asize;
5235                        dev->l2ad_hand += buf_aligned_asize;
5236                        ASSERT(dev->l2ad_hand <= dev->l2ad_evict ||
5237                            dev->l2ad_first);
4761                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4762                        write_psize += buf_p_sz;
4763                        dev->l2ad_hand += buf_p_sz;
5238                }

5240                if (l2arc_log_blk_insert(dev, ab)) {
5241                        l2arc_log_blk_commit(dev, pio, cb);
5242                        dev_hdr_update = B_TRUE;
5243                }
5244        }

5245        mutex_exit(&l2arc_buflist_mtx);

5247        if (dev_hdr_update)
5248                l2arc_dev_hdr_update(dev, pio);

5250        VERIFY3U(write_aligned_asize, <=, target_sz);
4769        ASSERT3U(write_asize, <=, target_sz);
5251        ARCSTAT_BUMP(arcstat_l2_writes_sent);
5252        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
5253        ARCSTAT_INCR(arcstat_l2_size, write_size);
5254        ARCSTAT_INCR(arcstat_l2_asize, write_aligned_asize);
5255        vdev_space_update(dev->l2ad_vdev, write_aligned_asize, 0, 0);
4772        ARCSTAT_INCR(arcstat_l2_size, write_sz);
4773        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
4774        vdev_space_update(dev->l2ad_vdev, write_psize, 0, 0);

5257        /*
5258         * Bump device hand to the device start if it is approaching the end.
5259         * l2arc_evict() will already have evicted ahead for this case.
5260         */
5261        if (dev->l2ad_hand + target_sz + l2arc_log_blk_overhead(target_sz) >=
5262            dev->l2ad_end) {
4780        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
5263                vdev_space_update(dev->l2ad_vdev,
5264                    dev->l2ad_end - dev->l2ad_hand, 0, 0);
5265                dev->l2ad_hand = dev->l2ad_start;
5266                dev->l2ad_evict = dev->l2ad_start;
5267                dev->l2ad_first = B_FALSE;
5268        }

5270        dev->l2ad_writing = B_TRUE;
5271        (void) zio_wait(pio);
5272        dev->l2ad_writing = B_FALSE;

5274        return (write_asize);
5275 }
_____unchanged_portion_omitted_

5521 boolean_t
5522 l2arc_vdev_present(vdev_t *vd)
```

```
5523 {
5524         return (l2arc_vdev_get(vd) != NULL);
5525 }

5527 static l2arc_dev_t *
5528 l2arc_vdev_get(vdev_t *vd)
5529 {
5530         l2arc_dev_t     *dev;
5531         boolean_t       held = MUTEX_HELD(&l2arc_dev_mtx);

5533         if (!held)
5534                 mutex_enter(&l2arc_dev_mtx);
5535         for (dev = list_head(l2arc_dev_list); dev != NULL;
5536             dev = list_next(l2arc_dev_list, dev)) {
5537                 if (dev->l2ad_vdev == vd)
5538                         break;
5539         }
5540         if (!held)
5541                 mutex_exit(&l2arc_dev_mtx);

5543         return (dev);
5052         return (dev != NULL);
5544 }

5546 /*
5547  * Add a vdev for use by the L2ARC.  By this point the spa has already
5548  * validated the vdev and opened it. The 'rebuild' flag indicates whether
5549  * we should attempt an L2ARC persistency rebuild.
5057  * validated the vdev and opened it.
5550  */
5551 void
5552 l2arc_add_vdev(spa_t *spa, vdev_t *vd, boolean_t rebuild)
5060 l2arc_add_vdev(spa_t *spa, vdev_t *vd)
5553 {
5554         l2arc_dev_t *adddev;

5556         ASSERT(!l2arc_vdev_present(vd));

5558         /*
5559          * Create a new l2arc device entry.
5560          */
5561         adddev = kmem_zalloc(sizeof (l2arc_dev_t), KM_SLEEP);
5562         adddev->l2ad_spa = spa;
5563         adddev->l2ad_vdev = vd;
5564         /* leave an extra SPA_MINBLOCKSIZE for l2arc device header */
5565         adddev->l2ad_start = VDEV_LABEL_START_SIZE + SPA_MINBLOCKSIZE;
5072         adddev->l2ad_start = VDEV_LABEL_START_SIZE;
5566         adddev->l2ad_end = VDEV_LABEL_START_SIZE + vdev_get_min_asize(vd);
5567         adddev->l2ad_hand = adddev->l2ad_start;
5568         adddev->l2ad_evict = adddev->l2ad_start;
5569         adddev->l2ad_first = B_TRUE;
5570         adddev->l2ad_writing = B_FALSE;

5572         /*
5573          * This is a list of all ARC buffers that are still valid on the
5574          * device.
5575          */
5576         adddev->l2ad_buflist = kmem_zalloc(sizeof (list_t), KM_SLEEP);
5577         list_create(adddev->l2ad_buflist, sizeof (arc_buf_hdr_t),
5578             offsetof(arc_buf_hdr_t, b_l2node));

5580         vdev_space_update(vd, 0, 0, adddev->l2ad_end - adddev->l2ad_hand);

5582         /*
5583          * Add device to global list
5584          */
```



```
5585         mutex_enter(&l2arc_dev_mtx);
5586         list_insert_head(l2arc_dev_list, adddev);
5587         atomic_inc_64(&l2arc_ndev);
5588         if (rebuild && l2arc_rebuild_enabled &&
5589             adddev->l2ad_end - adddev->l2ad_start > L2ARC_PERSIST_MIN_SIZE) {
5590                 /*
5591                  * Just mark the device as pending for a rebuild. We won't
5592                  * be starting a rebuild in line here as it would block pool
5593                  * import. Instead spa_load_impl will hand that off to an
5594                  * async task which will call l2arc_spa_rebuild_start.
5595                  */
5596                 adddev->l2ad_rebuild = B_TRUE;
5597         }
5598         mutex_exit(&l2arc_dev_mtx);
5599 }
______unchanged_portion_omitted_

5692 void
5693 l2arc_stop(void)
5694 {
5695         if (!(spa_mode_global & FWRITE))
5696                 return;

5698         mutex_enter(&l2arc_feed_thr_lock);
5699         cv_signal(&l2arc_feed_thr_cv);  /* kick thread out of startup */
5700         l2arc_thread_exit = 1;
5701         while (l2arc_thread_exit != 0)
5702                 cv_wait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock);
5703         mutex_exit(&l2arc_feed_thr_lock);
5704 }

5706 /*
5707  * Punches out rebuild threads for the L2ARC devices in a spa. This should
5708  * be called as one of the final steps of a pool import.
5709  */
5710 void
5711 l2arc_spa_rebuild_start(spa_t *spa)
5712 {
5713         l2arc_dev_t     *dev;
5714         /*
5715          * Locate the spa's l2arc devices and kick off rebuild threads.
5716          */
5717         mutex_enter(&l2arc_dev_mtx);
5718         for (int i = 0; i < spa->spa_l2cache.sav_count; i++) {
5719                 dev = l2arc_vdev_get(spa->spa_l2cache.sav_vdevs[i]);
5720                 ASSERT(dev != NULL);
5721                 if (dev->l2ad_rebuild) {
5722                         (void) thread_create(NULL, 0, l2arc_dev_rebuild_start,
5723                             dev, 0, &p0, TS_RUN, minclsyspri);
5724                 }
5725         }
5726         mutex_exit(&l2arc_dev_mtx);
5727 }

5729 /*
5730  * Main entry point for L2ARC rebuilding.
5731  */
5732 static void
5733 l2arc_dev_rebuild_start(l2arc_dev_t *dev)
5734 {
5735         spa_t *spa = dev->l2ad_spa;
5736         vdev_t *vd = dev->l2ad_vdev;

5738         /* Lock out device removal. */
5739         spa_config_enter(spa, SCL_L2ARC, vd, RW_READER);
5740         ASSERT(dev->l2ad_rebuild);
```

```
5741         (void) l2arc_rebuild(dev);
5742         dev->l2ad_rebuild = B_FALSE;
5743         spa_config_exit(spa, SCL_L2ARC, vd);
5744         thread_exit();
5745 }

5747 /*
5748  * This function implements the actual L2ARC metadata rebuild. It:
5749  *
5750  * 1) reads the device's header
5751  * 2) if a good device header is found, starts reading the log block chain
5752  * 3) restores each block's contents to memory (reconstructing arc_buf_hdr_t's)
5753  *
5754  * Operation stops under any of the following conditions:
5755  *
5756  * 1) We reach the end of the log blk chain (the back-reference in the blk is
5757  *    invalid or loops over our starting point).
5758  * 2) We encounter *any* error condition (cksum errors, io errors, looped
5759  *    blocks, etc.).
5760  * 3) The l2arc_rebuild_timeout is hit - this is a final resort to protect
5761  *    from making severely fragmented L2ARC log blocks or slow L2ARC devices
5762  *    prevent a machine from finishing a pool import (and thus letting the
5763  *    administrator take corrective action, e.g. by kicking the misbehaving
5764  *    L2ARC device out of the pool, or by reimporting the pool with L2ARC
5765  *    rebuilding disabled).
5766  */
5767 static int
5768 l2arc_rebuild(l2arc_dev_t *dev)
5769 {
5770         int                     err;
5771         l2arc_log_blk_phys_t    *this_lb, *next_lb;
5772         uint8_t                 *this_lb_buf, *next_lb_buf;
5773         zio_t                   *this_io = NULL, *next_io = NULL;
5774         int64_t                 deadline;
5775         l2arc_log_blk_ptr_t     lb_ptrs[2];
5776         boolean_t               first_pass;
5777         uint64_t                load_guid;

5779         load_guid = spa_load_guid(dev->l2ad_vdev->vdev_spa);
5780         deadline = ddi_get_lbolt64() + hz * l2arc_rebuild_timeout;
5781         /*
5782          * Device header processing phase.
5783          */
5784         if ((err = l2arc_dev_hdr_read(dev, &dev->l2ad_dev_hdr)) != 0) {
5785                 /* device header corrupted, start a new one */
5786                 bzero(&dev->l2ad_dev_hdr, sizeof (&dev->l2ad_dev_hdr));
5787                 return (err);
5788         }
5789         if (l2arc_check_rebuild_timeout_hit(deadline))
5790                 return (SET_ERROR(ETIMEDOUT));

5792         /* Retrieve the persistent L2ARC device state */
5793         dev->l2ad_evict = dev->l2ad_dev_hdr.l2dh_evict_tail;
5794         dev->l2ad_hand = vdev_psize_to_asize(dev->l2ad_vdev,
5795             dev->l2ad_dev_hdr.l2dh_start_lbps[0].l2lbp_daddr +
5796             LBP_GET_PSIZE(&dev->l2ad_dev_hdr.l2dh_start_lbps[0]));
5797         dev->l2ad_first = !!(dev->l2ad_dev_hdr.l2dh_flags &
5798             L2ARC_DEV_HDR_EVICT_FIRST);

5800         /* Prepare the rebuild processing state */
5801         bcopy(dev->l2ad_dev_hdr.l2dh_start_lbps, lb_ptrs, sizeof (lb_ptrs));
5802         this_lb = kmem_zalloc(sizeof (*this_lb), KM_SLEEP);
5803         next_lb = kmem_zalloc(sizeof (*next_lb), KM_SLEEP);
5804         this_lb_buf = kmem_zalloc(sizeof (l2arc_log_blk_phys_t), KM_SLEEP);
5805         next_lb_buf = kmem_zalloc(sizeof (l2arc_log_blk_phys_t), KM_SLEEP);
5806         first_pass = B_TRUE;
```



```
5808         /* Start the rebuild process */
5809         for (;;) {
5810                 if (!l2arc_log_blk_ptr_valid(dev, &lb_ptrs[0]))
5811                         /* We hit an invalid block address, end the rebuild. */
5812                         break;

5814                 if ((err = l2arc_log_blk_read(dev, &lb_ptrs[0], &lb_ptrs[1],
5815                     this_lb, next_lb, this_lb_buf, next_lb_buf,
5816                     this_io, &next_io)) != 0)
5817                         break;

5819                 /* Protection against infinite loops of log blocks. */
5820                 if (l2arc_range_check_overlap(lb_ptrs[1].l2lbp_daddr,
5821                     lb_ptrs[0].l2lbp_daddr,
5822                     dev->l2ad_dev_hdr.l2dh_start_lbps[0].l2lbp_daddr) &&
5823                     !first_pass) {
5824                         ARCSTAT_BUMP(arcstat_l2_rebuild_abort_loop_errors);
5825                         err = SET_ERROR(ELOOP);
5826                         break;
5827                 }

5829                 /*
5830                  * Our memory pressure valve. If the system is running low
5831                  * on memory, rather than swamping memory with new ARC buf
5832                  * hdrs, we opt not to rebuild the L2ARC. At this point,
5833                  * however, we have already set up our L2ARC dev to chain in
5834                  * new metadata log blk, so the user may choose to re-add the
5835                  * L2ARC dev at a later time to reconstruct it (when there's
5836                  * less memory pressure).
5837                  */
5838                 if (arc_reclaim_needed()) {
5839                         ARCSTAT_BUMP(arcstat_l2_rebuild_abort_lowmem);
5840                         cmn_err(CE_NOTE, "System running low on memory, "
5841                             "aborting L2ARC rebuild.");
5842                         err = SET_ERROR(ENOMEM);
5843                         break;
5844                 }

5846                 /*
5847                  * Now that we know that the next_lb checks out alright, we
5848                  * can start reconstruction from this lb - we can be sure
5849                  * that the L2ARC write hand has not yet reached any of our
5850                  * buffers.
5851                  */
5852                 l2arc_log_blk_restore(dev, load_guid, this_lb,
5853                     LBP_GET_PSIZE(&lb_ptrs[0]));

5855                 /*
5856                  * End of list detection. We can look ahead two steps in the
5857                  * blk chain and if the 2nd blk from this_lb dips below the
5858                  * initial chain starting point, then we know two things:
5859                  *      1) it can't be valid, and
5860                  *      2) the next_lb's ARC entries might have already been
5861                  *      partially overwritten and so we should stop before
5862                  *      we restore it
5863                  */
5864                 if (l2arc_range_check_overlap(
5865                     this_lb->l2lb_back2_lbp.l2lbp_daddr, lb_ptrs[0].l2lbp_daddr,
5866                     dev->l2ad_dev_hdr.l2dh_start_lbps[0].l2lbp_daddr) &&
5867                     !first_pass)
5868                         break;

5870                 /* log blk restored, continue with next one in the list */
5871                 lb_ptrs[0] = lb_ptrs[1];
5872                 lb_ptrs[1] = this_lb->l2lb_back2_lbp;
```

```
5873                 PTR_SWAP(this_lb, next_lb);
5874                 PTR_SWAP(this_lb_buf, next_lb_buf);
5875                 this_io = next_io;
5876                 next_io = NULL;
5877                 first_pass = B_FALSE;

5879                 if (l2arc_check_rebuild_timeout_hit(deadline)) {
5880                         err = SET_ERROR(ETIMEDOUT);
5881                         break;
5882                 }
5883         }
5884         if (next_io != NULL)
5885                 l2arc_log_blk_prefetch_abort(next_io);
5886         kmem_free(this_lb, sizeof (*this_lb));
5887         kmem_free(next_lb, sizeof (*next_lb));
5888         kmem_free(this_lb_buf, sizeof (l2arc_log_blk_phys_t));
5889         kmem_free(next_lb_buf, sizeof (l2arc_log_blk_phys_t));
5890         if (err == 0)
5891                 ARCSTAT_BUMP(arcstat_l2_rebuild_successes);

5893         return (err);
5894 }

5896 /*
5897  * Restores the payload of a log blk to ARC. This creates empty ARC hdr
5898  * entries which only contain an l2arc hdr, essentially restoring the
5899  * buffers to their L2ARC evicted state. This function also updates space
5900  * usage on the L2ARC vdev to make sure it tracks restored buffers.
5901  */
5902 static void
5903 l2arc_log_blk_restore(l2arc_dev_t *dev, uint64_t load_guid,
5904     l2arc_log_blk_phys_t *lb, uint64_t lb_psize)
5905 {
5906         uint64_t        size = 0, psize = 0;

5908         mutex_enter(&l2arc_buflist_mtx);

5910         for (int i = L2ARC_LOG_BLK_ENTRIES - 1; i >= 0; i--) {
5911                 /*
5912                  * Restore goes in the reverse direction to preserve correct
5913                  * temporal ordering of buffers in the l2ad_buflist.
5914                  */
5915                 l2arc_hdr_restore(&lb->l2lb_entries[i], dev, load_guid);
5916                 size += LE_GET_LSIZE(&lb->l2lb_entries[i]);
5917                 psize += LE_GET_PSIZE(&lb->l2lb_entries[i]);
5918         }
5919         mutex_exit(&l2arc_buflist_mtx);

5921         /*
5922          * Record rebuild stats:
5923          *      size            In-memory size of restored buffer data in ARC
5924          *      psize           Physical size of restored buffers in the L2ARC
5925          *      bufs            # of ARC buffer headers restored
5926          *      log_blks        # of L2ARC log entries processed during restore
5927          */
5928         ARCSTAT_INCR(arcstat_l2_rebuild_size, size);
5929         ARCSTAT_INCR(arcstat_l2_rebuild_psize, psize);
5930         ARCSTAT_INCR(arcstat_l2_rebuild_bufs, L2ARC_LOG_BLK_ENTRIES);
5931         ARCSTAT_BUMP(arcstat_l2_rebuild_log_blks);
5932         ARCSTAT_F_AVG(arcstat_l2_log_blk_avg_size, lb_psize);
5933         ARCSTAT_F_AVG(arcstat_l2_data_to_meta_ratio, psize / lb_psize);
5934         vdev_space_update(dev->l2ad_vdev, psize, 0, 0);
5935 }

5937 /*
5938  * Restores a single ARC buf hdr from a log block. The ARC buffer is put
```



```
5939  * into a state indicating that it has been evicted to L2ARC.
5940  */
5941 static void
5942 l2arc_hdr_restore(const l2arc_log_ent_phys_t *le, l2arc_dev_t *dev,
5943     uint64_t load_guid)
5944 {
5945         arc_buf_hdr_t   *hdr, *exists;
5946         kmutex_t        *hash_lock;
5947         arc_buf_contents_t      type = LE_GET_TYPE(le);
5948         l2arc_buf_hdr_t         *l2hdr;

5950         hdr = arc_buf_hdr_alloc(load_guid, LE_GET_LSIZE(le), type);
5951         hdr->b_dva = le->l2le_dva;
5952         hdr->b_birth = le->l2le_birth;
5953         hdr->b_cksum0 = le->l2le_cksum0;
5954         hdr->b_size = LE_GET_LSIZE(le);
5955         exists = buf_hash_insert(hdr, &hash_lock);
5956         if (exists) {
5957                 /* Buffer was already cached, no need to restore it. */
5958                 mutex_exit(hash_lock);
5959                 arc_hdr_destroy(hdr);
5960                 ARCSTAT_BUMP(arcstat_l2_rebuild_bufs_precached);
5961                 return;
5962         }
5963         hdr->b_flags = ARC_IN_HASH_TABLE | ARC_L2CACHE;
5964         if (LE_GET_COMPRESS(le) != ZIO_COMPRESS_OFF)
5965                 hdr->b_flags |= ARC_L2COMPRESS;
5966         mutex_enter(&hdr->b_freeze_lock);
5967         ASSERT(hdr->b_freeze_cksum == NULL);
5968         hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
5969         *hdr->b_freeze_cksum = le->l2le_freeze_cksum;
5970         mutex_exit(&hdr->b_freeze_lock);

5972         /* now rebuild the l2arc entry */
5973         ASSERT(hdr->b_l2hdr == NULL);
5974         l2hdr = kmem_zalloc(sizeof (*l2hdr), KM_SLEEP);
5975         l2hdr->b_dev = dev;
5976         l2hdr->b_daddr = le->l2le_daddr;
5977         l2hdr->b_asize = LE_GET_PSIZE(le);
5978         l2hdr->b_compress = LE_GET_COMPRESS(le);
5979         hdr->b_l2hdr = l2hdr;
5980         list_insert_tail(dev->l2ad_buflist, hdr);
5981         ARCSTAT_INCR(arcstat_l2_size, hdr->b_size);
5982         ARCSTAT_INCR(arcstat_l2_asize, l2hdr->b_asize);

5984         arc_change_state(arc_l2c_only, hdr, hash_lock);
5985         mutex_exit(hash_lock);
5986 }

5988 /*
5989  * Attempts to read the device header on the provided L2ARC device and writes
5990  * it to 'ub'. On success, this function returns 0, otherwise the appropriate
5991  * error code is returned.
5992  */
5993 static int
5994 l2arc_dev_hdr_read(l2arc_dev_t *dev, l2arc_dev_hdr_phys_t *hdr)
5995 {
5996         int             err;
5997         uint64_t        guid;
5998         zio_cksum_t     cksum;

6000         guid = spa_guid(dev->l2ad_vdev->vdev_spa);

6002         if ((err = zio_wait(zio_read_phys(NULL, dev->l2ad_vdev,
6003             VDEV_LABEL_START_SIZE, sizeof (*hdr), hdr,
6004             ZIO_CHECKSUM_OFF, NULL, NULL, ZIO_PRIORITY_ASYNC_READ,
```

```
6005             ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
6006             ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY, B_FALSE))) != 0) {
6007                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_io_errors);
6008                 return (err);
6009         }
6010 
6011         if (hdr->l2dh_magic == BSWAP_64(L2ARC_DEV_HDR_MAGIC))
6012                 byteswap_uint64_array(hdr, sizeof (*hdr));
6013 
6014         if (hdr->l2dh_magic != L2ARC_DEV_HDR_MAGIC ||
6015             hdr->l2dh_spa_guid != guid) {
6016                 /*
6017                  * Attempt to rebuild a device containing no actual dev hdr
6018                  * or containing a header from some other pool.
6019                  */
6020                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_unsupported);
6021                 return (SET_ERROR(ENOTSUP));
6022         }
6023 
6024         l2arc_dev_hdr_checksum(hdr, &cksum);
6025         if (!ZIO_CHECKSUM_EQUAL(hdr->l2dh_self_cksum, cksum)) {
6026                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_cksum_errors);
6027                 return (SET_ERROR(EINVAL));
6028         }
6029         if (hdr->l2dh_evict_tail < dev->l2ad_start ||
6030             hdr->l2dh_evict_tail >= dev->l2ad_end) {
6031                 /* Data in dev hdr is invalid for this device. */
6032                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_unsupported);
6033                 return (SET_ERROR(EINVAL));
6034         }
6035 
6036         return (0);
6037 }
6038 
6039 /*
6040  * Reads L2ARC log blocks from storage and validates their contents.
6041  *
6042  * This function implements a simple prefetcher to make sure that while
6043  * we're processing one buffer the L2ARC is already prefetching the next
6044  * one in the chain.
6045  *
6046  * The arguments this_lp and next_lp point to the current and next log blk
6047  * address in the block chain. Similarly, this_lb and next_lb hold the
6048  * l2arc_log_blk_phys_t's of the current and next L2ARC blk. The this_lb_buf
6049  * and next_lb_buf must be buffers of appropriate to hold a raw
6050  * l2arc_log_blk_phys_t (they are used as catch buffers for read ops prior
6051  * to buffer decompression).
6052  *
6053  * The 'this_io' and 'next_io' arguments are used for block prefetching.
6054  * When issuing the first blk IO during rebuild, you should pass NULL for
6055  * 'this_io'. This function will then issue a sync IO to read the block and
6056  * also issue an async IO to fetch the next block in the block chain. The
6057  * prefetch IO is returned in 'next_io'. On subsequent calls to this
6058  * function, pass the value returned in 'next_io' from the previous call
6059  * as 'this_io' and a fresh 'next_io' pointer to hold the next prefetch IO.
6060  * Prior to the call, you should initialize your 'next_io' pointer to be
6061  * NULL. If no prefetch IO was issued, the pointer is left set at NULL.
6062  *
6063  * On success, this function returns 0, otherwise it returns an appropriate
6064  * error code. On error the prefetching IO is aborted and cleared before
6065  * returning from this function. Therefore, if we return 'success', the
6066  * caller can assume that we have taken care of cleanup of prefetch IOs.
6067  */
6068 static int
6069 l2arc_log_blk_read(l2arc_dev_t *dev,
6070     const l2arc_log_blk_ptr_t *this_lbp, const l2arc_log_blk_ptr_t *next_lbp,
```



```
6071     l2arc_log_blk_phys_t *this_lb, l2arc_log_blk_phys_t *next_lb,
6072     uint8_t *this_lb_buf, uint8_t *next_lb_buf,
6073     zio_t *this_io, zio_t **next_io)
6074 {
6075         int err = 0;
6076         zio_cksum_t cksum;
6077 
6078         ASSERT(this_lbp != NULL && next_lbp != NULL);
6079         ASSERT(this_lb != NULL && next_lb != NULL);
6080         ASSERT(this_lb_buf != NULL && next_lb_buf != NULL);
6081         ASSERT(next_io != NULL && *next_io == NULL);
6082         ASSERT(l2arc_log_blk_ptr_valid(dev, this_lbp));
6083 
6084         /*
6085          * Check to see if we have issued the IO for this log blk in a
6086          * previous run. If not, this is the first call, so issue it now.
6087          */
6088         if (this_io == NULL) {
6089                 this_io = l2arc_log_blk_prefetch(dev->l2ad_vdev, this_lbp,
6090                     this_lb_buf);
6091         }
6092 
6093         /*
6094          * Peek to see if we can start issuing the next IO immediately.
6095          */
6096         if (l2arc_log_blk_ptr_valid(dev, next_lbp)) {
6097                 /*
6098                  * Start issuing IO for the next log blk early - this
6099                  * should help keep the L2ARC device busy while we
6100                  * decompress and restore this log blk.
6101                  */
6102                 *next_io = l2arc_log_blk_prefetch(dev->l2ad_vdev, next_lbp,
6103                     next_lb_buf);
6104         }
6105 
6106         /* Wait for the IO to read this log block to complete */
6107         if ((err = zio_wait(this_io)) != 0) {
6108                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_io_errors);
6109                 goto cleanup;
6110         }
6111 
6112         /* Make sure the buffer checks out */
6113         fletcher_4_native(this_lb_buf, LBP_GET_PSIZE(this_lbp), &cksum);
6114         if (!ZIO_CHECKSUM_EQUAL(cksum, this_lbp->l2lbp_cksum)) {
6115                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_cksum_errors);
6116                 err = SET_ERROR(EINVAL);
6117                 goto cleanup;
6118         }
6119 
6120         /* Now we can take our time decoding this buffer */
6121         switch (LBP_GET_COMPRESS(this_lbp)) {
6122         case ZIO_COMPRESS_OFF:
6123                 bcopy(this_lb_buf, this_lb, sizeof (*this_lb));
6124                 break;
6125         case ZIO_COMPRESS_LZ4:
6126                 if ((err = zio_decompress_data(LBP_GET_COMPRESS(this_lbp),
6127                     this_lb_buf, this_lb, LBP_GET_PSIZE(this_lbp),
6128                     sizeof (*this_lb))) != 0) {
6129                         err = SET_ERROR(EINVAL);
6130                         goto cleanup;
6131                 }
6132                 break;
6133         default:
6134                 err = SET_ERROR(EINVAL);
6135                 goto cleanup;
6136         }
```

```
6137         if (this_lb->l2lb_magic == BSWAP_64(L2ARC_LOG_BLK_MAGIC))
6138                 byteswap_uint64_array(this_lb, sizeof (*this_lb));
6139         if (this_lb->l2lb_magic != L2ARC_LOG_BLK_MAGIC) {
6140                 err = SET_ERROR(EINVAL);
6141                 goto cleanup;
6142         }
6143 cleanup:
6144         /* Abort an in-flight prefetch I/O in case of error */
6145         if (err != 0 && *next_io != NULL) {
6146                 l2arc_log_blk_prefetch_abort(*next_io);
6147                 *next_io = NULL;
6148         }
6149         return (err);
6150 }

6152 /*
6153  * Validates an L2ARC log blk address to make sure that it can be read
6154  * from the provided L2ARC device. Returns B_TRUE if the address is
6155  * within the device's bounds, or B_FALSE if not.
6156  */
6157 static boolean_t
6158 l2arc_log_blk_ptr_valid(l2arc_dev_t *dev, const l2arc_log_blk_ptr_t *lbp)
6159 {
6160         uint64_t psize = LBP_GET_PSIZE(lbp);
6161         uint64_t end = lbp->l2lbp_daddr + psize;

6163         /*
6164          * A log block is valid if all of the following conditions are true:
6165          * - it fits entirely between l2ad_start and l2ad_end
6166          * - it has a valid size
6167          * - it isn't anywhere between l2ad_hand and l2ad_evict (i.e. it
6168          *      doesn't sit in the evicted region)
6169          */
6170         return (lbp->l2lbp_daddr >= dev->l2ad_start && end < dev->l2ad_end &&
6171             psize != 0 && psize <= sizeof (l2arc_log_blk_phys_t) &&
6172             lbp->l2lbp_daddr > dev->l2ad_evict && end <= dev->l2ad_hand);
6173 }

6175 /*
6176  * Starts an asynchronous read IO to read a log block. This is used in log
6177  * block reconstruction to start reading the next block before we are done
6178  * decoding and reconstructing the current block, to keep the l2arc device
6179  * nice and hot with read IO to process.
6180  * The returned zio will contain a newly allocated memory buffers for the IO
6181  * data which should then be freed by the caller once the zio is no longer
6182  * needed (i.e. due to it having completed). If you wish to abort this
6183  * zio, you should do so using l2arc_log_blk_prefetch_abort, which takes
6184  * care of disposing of the allocated buffers correctly.
6185  */
6186 static zio_t *
6187 l2arc_log_blk_prefetch(vdev_t *vd, const l2arc_log_blk_ptr_t *lbp,
6188     uint8_t *lb_buf)
6189 {
6190         uint32_t psize;
6191         zio_t *pio;

6193         psize = LBP_GET_PSIZE(lbp);
6194         ASSERT(psize <= sizeof (l2arc_log_blk_phys_t));
6195         pio = zio_root(vd->vdev_spa, NULL, NULL, ZIO_FLAG_DONT_CACHE |
6196             ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE |
6197             ZIO_FLAG_DONT_RETRY);
6198         (void) zio_nowait(zio_read_phys(pio, vd, lbp->l2lbp_daddr, psize,
6199             lb_buf, ZIO_CHECKSUM_OFF, NULL, NULL, ZIO_PRIORITY_ASYNC_READ,
6200             ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
6201             ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY, B_FALSE));
```



```
6203         return (pio);
6204 }

6206 /*
6207  * Aborts a zio returned from l2arc_log_blk_prefetch and frees the data
6208  * buffers allocated for it.
6209  */
6210 static void
6211 l2arc_log_blk_prefetch_abort(zio_t *zio)
6212 {
6213         (void) zio_wait(zio);
6214 }

6216 /*
6217  * Creates a zio to update the device header on an l2arc device. The zio is
6218  * initiated as a child of 'pio'.
6219  */
6220 static void
6221 l2arc_dev_hdr_update(l2arc_dev_t *dev, zio_t *pio)
6222 {
6223         zio_t *wzio;
6224         vdev_stat_t st;
6225         l2arc_dev_hdr_phys_t *hdr = &dev->l2ad_dev_hdr;

6227         vdev_get_stats(dev->l2ad_vdev, &st);

6229         hdr->l2dh_magic = L2ARC_DEV_HDR_MAGIC;
6230         hdr->l2dh_spa_guid = spa_guid(dev->l2ad_vdev->vdev_spa);
6231         hdr->l2dh_evict_tail = dev->l2ad_evict;
6232         hdr->l2dh_alloc_space = st.vs_alloc;
6233         hdr->l2dh_flags = 0;
6234         if (dev->l2ad_first)
6235                 hdr->l2dh_flags |= L2ARC_DEV_HDR_EVICT_FIRST;

6237         /* checksum operation goes last */
6238         l2arc_dev_hdr_checksum(hdr, &hdr->l2dh_self_cksum);

6240         CTASSERT(sizeof (*hdr) >= SPA_MINBLOCKSIZE &&
6241             sizeof (*hdr) <= SPA_MAXBLOCKSIZE);
6242         wzio = zio_write_phys(pio, dev->l2ad_vdev, VDEV_LABEL_START_SIZE,
6243             sizeof (*hdr), hdr, ZIO_CHECKSUM_OFF, NULL,
6244             NULL, ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL, B_FALSE);
6245         DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
6246             zio_t *, wzio);
6247         (void) zio_nowait(wzio);
6248 }

6250 /*
6251  * Commits a log block to the L2ARC device. This routine is invoked from
6252  * l2arc_write_buffers when the log block fills up.
6253  * This function allocates some memory to temporarily hold the serialized
6254  * buffer to be written. This is then released in l2arc_write_done.
6255  */
6256 static void
6257 l2arc_log_blk_commit(l2arc_dev_t *dev, zio_t *pio,
6258     l2arc_write_callback_t *cb)
6259 {
6260         l2arc_log_blk_phys_t    *lb = &dev->l2ad_log_blk;
6261         uint64_t                psize, asize;
6262         l2arc_log_blk_buf_t     *lb_buf;
6263         zio_t                   *wzio;

6265         VERIFY(dev->l2ad_log_ent_idx == L2ARC_LOG_BLK_ENTRIES);

6267         /* link the buffer into the block chain */
6268         lb->l2lb_back2_lbp = dev->l2ad_dev_hdr.l2dh_start_lbps[1];
```

```
6269         lb->l2lb_magic = L2ARC_LOG_BLK_MAGIC;

6271         /* try to compress the buffer */
6272         lb_buf = kmem_zalloc(sizeof (*lb_buf), KM_SLEEP);
6273         list_insert_tail(&cb->l2wcb_log_blk_buf_list, lb_buf);
6274         VERIFY((psize = zio_compress_data(ZIO_COMPRESS_LZ4, lb,
6275             lb_buf->l2lbb_log_blk, sizeof (*lb))) != 0);

6277         /*
6278          * Update the start log blk pointer in the device header to point
6279          * to the log block we're about to write.
6280          */
6281         dev->l2ad_dev_hdr.l2dh_start_lbps[1] =
6282             dev->l2ad_dev_hdr.l2dh_start_lbps[0];
6283         dev->l2ad_dev_hdr.l2dh_start_lbps[0].l2lbp_daddr = dev->l2ad_hand;
6284         LBP_SET_LSIZE(&dev->l2ad_dev_hdr.l2dh_start_lbps[0], sizeof (*lb));
6285         LBP_SET_PSIZE(&dev->l2ad_dev_hdr.l2dh_start_lbps[0], psize);
6286         LBP_SET_CHECKSUM(&dev->l2ad_dev_hdr.l2dh_start_lbps[0],
6287             ZIO_CHECKSUM_FLETCHER_4);
6288         LBP_SET_TYPE(&dev->l2ad_dev_hdr.l2dh_start_lbps[0], 0);
6289         if (psize < sizeof (*lb)) {
6290                 /* compression succeeded */
6291                 LBP_SET_COMPRESS(&dev->l2ad_dev_hdr.l2dh_start_lbps[0],
6292                     ZIO_COMPRESS_LZ4);
6293         } else {
6294                 /* compression failed */
6295                 bcopy(lb, lb_buf->l2lbb_log_blk, sizeof (*lb));
6296                 LBP_SET_COMPRESS(&dev->l2ad_dev_hdr.l2dh_start_lbps[0],
6297                     ZIO_COMPRESS_OFF);
6298         }
6299         /* checksum what we're about to write */
6300         fletcher_4_native(lb_buf->l2lbb_log_blk, psize,
6301             &dev->l2ad_dev_hdr.l2dh_start_lbps[0].l2lbp_cksum);

6303         /* perform the write itself */
6304         CTASSERT(L2ARC_LOG_BLK_SIZE >= SPA_MINBLOCKSIZE &&
6305             L2ARC_LOG_BLK_SIZE <= SPA_MAXBLOCKSIZE);
6306         wzio = zio_write_phys(pio, dev->l2ad_vdev, dev->l2ad_hand,
6307             psize, lb_buf->l2lbb_log_blk, ZIO_CHECKSUM_OFF, NULL, NULL,
6308             ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL, B_FALSE);
6309         DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev, zio_t *, wzio);
6310         (void) zio_nowait(wzio);

6312         /* realign the device hand */
6313         asize = vdev_psize_to_asize(dev->l2ad_vdev, psize);
6314         dev->l2ad_hand += asize;
6315         VERIFY(dev->l2ad_hand <= dev->l2ad_evict || dev->l2ad_first);
6316         vdev_space_update(dev->l2ad_vdev, asize, 0, 0);

6318         /* bump the kstats */
6319         ARCSTAT_INCR(arcstat_l2_write_bytes, psize);
6320         ARCSTAT_BUMP(arcstat_l2_log_blk_writes);
6321         ARCSTAT_F_AVG(arcstat_l2_log_blk_avg_size, asize);
6322         ARCSTAT_F_AVG(arcstat_l2_data_to_meta_ratio,
6323             dev->l2ad_log_blk_payload_asize / asize);

6325         dev->l2ad_log_ent_idx = dev->l2ad_log_blk_payload_asize = 0;
6326 }

6328 /*
6329  * Computes the checksum of 'hdr' and stores it in 'cksum'.
6330  */
6331 static void
6332 l2arc_dev_hdr_checksum(const l2arc_dev_hdr_phys_t *hdr, zio_cksum_t *cksum)
6333 {
6334         fletcher_4_native((uint8_t *)hdr +
```



```
6335             offsetof(l2arc_dev_hdr_phys_t, l2dh_spa_guid),
6336             sizeof (*hdr) - offsetof(l2arc_dev_hdr_phys_t, l2dh_spa_guid),
6337             cksum);
6338 }

6340 /*
6341  * Inserts ARC buffer 'ab' into the current L2ARC log blk on the device.
6342  * The buffer being inserted must be present in L2ARC.
6343  * Returns B_TRUE if the L2ARC log blk is full and needs to be committed
6344  * to L2ARC, or B_FALSE if it still has room for more ARC buffers.
6345  */
6346 static boolean_t
6347 l2arc_log_blk_insert(l2arc_dev_t *dev, const arc_buf_hdr_t *ab)
6348 {
6349         l2arc_log_blk_phys_t *lb = &dev->l2ad_log_blk;
6350         l2arc_log_ent_phys_t *le;
6351         const l2arc_buf_hdr_t *l2hdr = ab->b_l2hdr;
6352         int index = dev->l2ad_log_ent_idx++;

6354         ASSERT(l2hdr != NULL);
6355         ASSERT(index < L2ARC_LOG_BLK_ENTRIES);

6357         le = &lb->l2lb_entries[index];
6358         bzero(le, sizeof (*le));
6359         le->l2le_dva = ab->b_dva;
6360         le->l2le_birth = ab->b_birth;
6361         le->l2le_cksum0 = ab->b_cksum0;
6362         le->l2le_daddr = l2hdr->b_daddr;
6363         LE_SET_LSIZE(le, ab->b_size);
6364         LE_SET_PSIZE(le, l2hdr->b_asize);
6365         LE_SET_COMPRESS(le, l2hdr->b_compress);
6366         le->l2le_freeze_cksum = *ab->b_freeze_cksum;
6367         LE_SET_CHECKSUM(le, ZIO_CHECKSUM_FLETCHER_2);
6368         LE_SET_TYPE(le, ab->b_type);
6369         dev->l2ad_log_blk_payload_asize += l2hdr->b_asize;

6371         return (dev->l2ad_log_ent_idx == L2ARC_LOG_BLK_ENTRIES);
6372 }

6374 /*
6375  * Checks whether a given L2ARC device address sits in a time-sequential
6376  * range. The trick here is that the L2ARC is a rotary buffer, so we can't
6377  * just do a range comparison, we need to handle the situation in which the
6378  * range wraps around the end of the L2ARC device. Arguments:
6379  *      bottom  Lower end of the range to check (written to earlier).
6380  *      top     Upper end of the range to check (written to later).
6381  *      check   The address for which we want to determine if it sits in
6382  *              between the top and bottom.
6383  *
6384  * The 3-way conditional below represents the following cases:
6385  *
6386  *      bottom < top : Sequentially ordered case:
6387  *        <check>--------+-------------------+
6388  *                       |  (overlap here?)  |
6389  *       L2ARC dev       V                   V
6390  *       |---------------<bottom>============<top>--------------|
6391  *
6392  *      bottom > top: Looped-around case:
6393  *                            <check>--------+------------------+
6394  *                                           |  (overlap here?) |
6395  *       L2ARC dev                           V                  V
6396  *       |===============<top>--------------<bottom>===========|
6397  *       ^               ^
6398  *       |  (or here?)   |
6399  *       +---------------+---------<check>
6400  *
```

```
6401  *       top == bottom : Just a single address comparison.
6402  */
6403 static inline boolean_t
6404 l2arc_range_check_overlap(uint64_t bottom, uint64_t top, uint64_t check)
6405 {
6406         if (bottom < top)
6407                 return (bottom <= check && check <= top);
6408         else if (bottom > top)
6409                 return (check <= top || bottom <= check);
6410         else
6411                 return (check == top);
6412 }

6414 /*
6415  * Checks whether a rebuild timeout deadline has been hit and if it has,
6416  * increments the appropriate error counters.
6417  */
6418 static boolean_t
6419 l2arc_check_rebuild_timeout_hit(int64_t deadline)
6420 {
6421         if (deadline != 0 && deadline < ddi_get_lbolt64()) {
6422                 ARCSTAT_BUMP(arcstat_l2_rebuild_abort_timeout);
6423                 cmn_err(CE_WARN, "L2ARC rebuild is taking too long, "
6424                     "dropping remaining L2ARC metadata.");
6425                 return (B_TRUE);
6426         } else {
6427                 return (B_FALSE);
6428         }
6429 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  176145 Mon Dec  9 16:07:36 2013
new/usr/src/uts/common/fs/zfs/spa.c
3525 Persistent L2ARC
**********************************************************
______unchanged_portion_omitted_

1409 /*
1410  * Load (or re-load) the current list of vdevs describing the active l2cache for
1411  * this pool.  When this is called, we have some form of basic information in
1412  * 'spa_l2cache.sav_config'.  We parse this into vdevs, try to open them, and
1413  * then re-generate a more complete list including status information.
1414  * Devices which are already active have their details maintained, and are
1415  * not re-opened.
1416  */
1417 static void
1418 spa_load_l2cache(spa_t *spa)
1419 {
1420         nvlist_t **l2cache;
1421         uint_t nl2cache;
1422         int i, j, oldnvdevs;
1423         uint64_t guid;
1424         vdev_t *vd, **oldvdevs, **newvdevs;
1425         spa_aux_vdev_t *sav = &spa->spa_l2cache;

1427         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

1429         if (sav->sav_config != NULL) {
1430                 VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
1431                     ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
1432                 newvdevs = kmem_alloc(nl2cache * sizeof (void *), KM_SLEEP);
1433         } else {
1434                 nl2cache = 0;
1435                 newvdevs = NULL;
1436         }

1438         oldvdevs = sav->sav_vdevs;
1439         oldnvdevs = sav->sav_count;
1440         sav->sav_vdevs = NULL;
1441         sav->sav_count = 0;

1443         /*
1444          * Process new nvlist of vdevs.
1445          */
1446         for (i = 0; i < nl2cache; i++) {
1447                 VERIFY(nvlist_lookup_uint64(l2cache[i], ZPOOL_CONFIG_GUID,
1448                     &guid) == 0);

1450                 newvdevs[i] = NULL;
1451                 for (j = 0; j < oldnvdevs; j++) {
1452                         vd = oldvdevs[j];
1453                         if (vd != NULL && guid == vd->vdev_guid) {
1454                                 /*
1455                                  * Retain previous vdev for add/remove ops.
1456                                  */
1457                                 newvdevs[i] = vd;
1458                                 oldvdevs[j] = NULL;
1459                                 break;
1460                         }
1461                 }

1463                 if (newvdevs[i] == NULL) {
1464                         /*
1465                          * Create new vdev
1466                          */
1467                         VERIFY(spa_config_parse(spa, &vd, l2cache[i], NULL, 0,
```



```
1468                             VDEV_ALLOC_L2CACHE) == 0);
1469                         ASSERT(vd != NULL);
1470                         newvdevs[i] = vd;

1472                         /*
1473                          * Commit this vdev as an l2cache device,
1474                          * even if it fails to open.
1475                          */
1476                         spa_l2cache_add(vd);

1478                         vd->vdev_top = vd;
1479                         vd->vdev_aux = sav;

1481                         spa_l2cache_activate(vd);

1483                         if (vdev_open(vd) != 0)
1484                                 continue;

1486                         (void) vdev_validate_aux(vd);

1488                         if (!vdev_is_dead(vd)) {
1489                                 boolean_t do_rebuild = B_FALSE;

1491                                 (void) nvlist_lookup_boolean_value(l2cache[i],
1492                                     ZPOOL_CONFIG_L2CACHE_PERSISTENT,
1493                                     &do_rebuild);
1494                                 l2arc_add_vdev(spa, vd, do_rebuild);
1488                         if (!vdev_is_dead(vd))
1489                                 l2arc_add_vdev(spa, vd);
1495                         }
1496                 }
1497         }

1499         /*
1500          * Purge vdevs that were dropped
1501          */
1502         for (i = 0; i < oldnvdevs; i++) {
1503                 uint64_t pool;

1505                 vd = oldvdevs[i];
1506                 if (vd != NULL) {
1507                         ASSERT(vd->vdev_isl2cache);

1509                         if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
1510                             pool != 0ULL && l2arc_vdev_present(vd))
1511                                 l2arc_remove_vdev(vd);
1512                         vdev_clear_stats(vd);
1513                         vdev_free(vd);
1514                 }
1515         }

1517         if (oldvdevs)
1518                 kmem_free(oldvdevs, oldnvdevs * sizeof (void *));

1520         if (sav->sav_config == NULL)
1521                 goto out;

1523         sav->sav_vdevs = newvdevs;
1524         sav->sav_count = (int)nl2cache;

1526         /*
1527          * Recompute the stashed list of l2cache devices, with status
1528          * information this time.
1529          */
1530         VERIFY(nvlist_remove(sav->sav_config, ZPOOL_CONFIG_L2CACHE,
1531             DATA_TYPE_NVLIST_ARRAY) == 0);
```

```
1533         l2cache = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
1534         for (i = 0; i < sav->sav_count; i++)
1535                 l2cache[i] = vdev_config_generate(spa,
1536                     sav->sav_vdevs[i], B_TRUE, VDEV_CONFIG_L2CACHE);
1537         VERIFY(nvlist_add_nvlist_array(sav->sav_config,
1538             ZPOOL_CONFIG_L2CACHE, l2cache, sav->sav_count) == 0);
1539 out:
1540         for (i = 0; i < sav->sav_count; i++)
1541                 nvlist_free(l2cache[i]);
1542         if (sav->sav_count)
1543                 kmem_free(l2cache, sav->sav_count * sizeof (void *));
1544 }
_____unchanged_portion_omitted_

2076 /*
2077  * Load an existing storage pool, using the pool's builtin spa_config as a
2078  * source of configuration information.
2079  */
2080 static int
2081 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2082     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2083     char **ereport)
2084 {
2085         int error = 0;
2086         nvlist_t *nvroot = NULL;
2087         nvlist_t *label;
2088         vdev_t *rvd;
2089         uberblock_t *ub = &spa->spa_uberblock;
2090         uint64_t children, config_cache_txg = spa->spa_config_txg;
2091         int orig_mode = spa->spa_mode;
2092         int parse;
2093         uint64_t obj;
2094         boolean_t missing_feat_write = B_FALSE;

2096         /*
2097          * If this is an untrusted config, access the pool in read-only mode.
2098          * This prevents things like resilvering recently removed devices.
2099          */
2100         if (!mosconfig)
2101                 spa->spa_mode = FREAD;

2103         ASSERT(MUTEX_HELD(&spa_namespace_lock));

2105         spa->spa_load_state = state;

2107         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2108                 return (SET_ERROR(EINVAL));

2110         parse = (type == SPA_IMPORT_EXISTING ?
2111             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

2113         /*
2114          * Create "The Godfather" zio to hold all async IOs
2115          */
2116         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2117             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

2119         /*
2120          * Parse the configuration into a vdev tree.  We explicitly set the
2121          * value that will be returned by spa_version() since parsing the
2122          * configuration requires knowing the version number.
2123          */
2124         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2125         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2126         spa_config_exit(spa, SCL_ALL, FTAG);
```



```
2128         if (error != 0)
2129                 return (error);

2131         ASSERT(spa->spa_root_vdev == rvd);

2133         if (type != SPA_IMPORT_ASSEMBLE) {
2134                 ASSERT(spa_guid(spa) == pool_guid);
2135         }

2137         /*
2138          * Try to open all vdevs, loading each label in the process.
2139          */
2140         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2141         error = vdev_open(rvd);
2142         spa_config_exit(spa, SCL_ALL, FTAG);
2143         if (error != 0)
2144                 return (error);

2146         /*
2147          * We need to validate the vdev labels against the configuration that
2148          * we have in hand, which is dependent on the setting of mosconfig. If
2149          * mosconfig is true then we're validating the vdev labels based on
2150          * that config.  Otherwise, we're validating against the cached config
2151          * (zpool.cache) that was read when we loaded the zfs module, and then
2152          * later we will recursively call spa_load() and validate against
2153          * the vdev config.
2154          *
2155          * If we're assembling a new pool that's been split off from an
2156          * existing pool, the labels haven't yet been updated so we skip
2157          * validation for now.
2158          */
2159         if (type != SPA_IMPORT_ASSEMBLE) {
2160                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2161                 error = vdev_validate(rvd, mosconfig);
2162                 spa_config_exit(spa, SCL_ALL, FTAG);

2164                 if (error != 0)
2165                         return (error);

2167                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2168                         return (SET_ERROR(ENXIO));
2169         }

2171         /*
2172          * Find the best uberblock.
2173          */
2174         vdev_uberblock_load(rvd, ub, &label);

2176         /*
2177          * If we weren't able to find a single valid uberblock, return failure.
2178          */
2179         if (ub->ub_txg == 0) {
2180                 nvlist_free(label);
2181                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2182         }

2184         /*
2185          * If the pool has an unsupported version we can't open it.
2186          */
2187         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2188                 nvlist_free(label);
2189                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2190         }

2192         if (ub->ub_version >= SPA_VERSION_FEATURES) {
```

```
2193                nvlist_t *features;

2195                /*
2196                 * If we weren't able to find what's necessary for reading the
2197                 * MOS in the label, return failure.
2198                 */
2199                if (label == NULL || nvlist_lookup_nvlist(label,
2200                    ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2201                        nvlist_free(label);
2202                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2203                            ENXIO));
2204                }

2206                /*
2207                 * Update our in-core representation with the definitive values
2208                 * from the label.
2209                 */
2210                nvlist_free(spa->spa_label_features);
2211                VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2212        }

2214        nvlist_free(label);

2216        /*
2217         * Look through entries in the label nvlist's features_for_read. If
2218         * there is a feature listed there which we don't understand then we
2219         * cannot open a pool.
2220         */
2221        if (ub->ub_version >= SPA_VERSION_FEATURES) {
2222                nvlist_t *unsup_feat;

2224                VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2225                    0);

2227                for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2228                    NULL); nvp != NULL;
2229                    nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2230                        if (!zfeature_is_supported(nvpair_name(nvp))) {
2231                                VERIFY(nvlist_add_string(unsup_feat,
2232                                    nvpair_name(nvp), "") == 0);
2233                        }
2234                }

2236                if (!nvlist_empty(unsup_feat)) {
2237                        VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2238                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2239                        nvlist_free(unsup_feat);
2240                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2241                            ENOTSUP));
2242                }

2244                nvlist_free(unsup_feat);
2245        }

2247        /*
2248         * If the vdev guid sum doesn't match the uberblock, we have an
2249         * incomplete configuration.  We first check to see if the pool
2250         * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2251         * If it is, defer the vdev_guid_sum check till later so we
2252         * can handle missing vdevs.
2253         */
2254        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2255            &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2256            rvd->vdev_guid_sum != ub->ub_guid_sum)
2257                return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));
```



```
2259        if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2260                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2261                spa_try_repair(spa, config);
2262                spa_config_exit(spa, SCL_ALL, FTAG);
2263                nvlist_free(spa->spa_config_splitting);
2264                spa->spa_config_splitting = NULL;
2265        }

2267        /*
2268         * Initialize internal SPA structures.
2269         */
2270        spa->spa_state = POOL_STATE_ACTIVE;
2271        spa->spa_ubsync = spa->spa_uberblock;
2272        spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2273            TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2274        spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2275            spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2276        spa->spa_claim_max_txg = spa->spa_first_txg;
2277        spa->spa_prev_software_version = ub->ub_software_version;

2279        error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2280        if (error)
2281                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2282        spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2284        if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2285                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2287        if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2288                boolean_t missing_feat_read = B_FALSE;
2289                nvlist_t *unsup_feat, *enabled_feat;

2291                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2292                    &spa->spa_feat_for_read_obj) != 0) {
2293                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2294                }

2296                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2297                    &spa->spa_feat_for_write_obj) != 0) {
2298                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2299                }

2301                if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2302                    &spa->spa_feat_desc_obj) != 0) {
2303                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2304                }

2306                enabled_feat = fnvlist_alloc();
2307                unsup_feat = fnvlist_alloc();

2309                if (!spa_features_check(spa, B_FALSE,
2310                    unsup_feat, enabled_feat))
2311                        missing_feat_read = B_TRUE;

2313                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2314                        if (!spa_features_check(spa, B_TRUE,
2315                            unsup_feat, enabled_feat)) {
2316                                missing_feat_write = B_TRUE;
2317                        }
2318                }

2320                fnvlist_add_nvlist(spa->spa_load_info,
2321                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2323                if (!nvlist_empty(unsup_feat)) {
2324                        fnvlist_add_nvlist(spa->spa_load_info,
```

```
2325                                ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2326                }
2327
2328                fnvlist_free(enabled_feat);
2329                fnvlist_free(unsup_feat);
2330
2331                if (!missing_feat_read) {
2332                        fnvlist_add_boolean(spa->spa_load_info,
2333                            ZPOOL_CONFIG_CAN_RDONLY);
2334                }
2335
2336                /*
2337                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
2338                 * twofold: to determine whether the pool is available for
2339                 * import in read-write mode and (if it is not) whether the
2340                 * pool is available for import in read-only mode. If the pool
2341                 * is available for import in read-write mode, it is displayed
2342                 * as available in userland; if it is not available for import
2343                 * in read-only mode, it is displayed as unavailable in
2344                 * userland. If the pool is available for import in read-only
2345                 * mode but not read-write mode, it is displayed as unavailable
2346                 * in userland with a special note that the pool is actually
2347                 * available for open in read-only mode.
2348                 *
2349                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2350                 * missing a feature for write, we must first determine whether
2351                 * the pool can be opened read-only before returning to
2352                 * userland in order to know whether to display the
2353                 * abovementioned note.
2354                 */
2355                if (missing_feat_read || (missing_feat_write &&
2356                    spa_writeable(spa))) {
2357                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2358                            ENOTSUP));
2359                }
2360        }
2361
2362        spa->spa_is_initializing = B_TRUE;
2363        error = dsl_pool_open(spa->spa_dsl_pool);
2364        spa->spa_is_initializing = B_FALSE;
2365        if (error != 0)
2366                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2367
2368        if (!mosconfig) {
2369                uint64_t hostid;
2370                nvlist_t *policy = NULL, *nvconfig;
2371
2372                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2373                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2374
2375                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2376                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2377                        char *hostname;
2378                        unsigned long myhostid = 0;
2379
2380                        VERIFY(nvlist_lookup_string(nvconfig,
2381                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);
2382
2383 #ifdef  _KERNEL
2384                        myhostid = zone_get_hostid(NULL);
2385 #else   /* _KERNEL */
2386                        /*
2387                         * We're emulating the system's hostid in userland, so
2388                         * we can't use zone_get_hostid().
2389                         */
2390                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
```



```
2391 #endif  /* _KERNEL */
2392                        if (hostid != 0 && myhostid != 0 &&
2393                            hostid != myhostid) {
2394                                nvlist_free(nvconfig);
2395                                cmn_err(CE_WARN, "pool '%s' could not be "
2396                                    "loaded as it was last accessed by "
2397                                    "another system (host: %s hostid: 0x%lx). "
2398                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2399                                    spa_name(spa), hostname,
2400                                    (unsigned long)hostid);
2401                                return (SET_ERROR(EBADF));
2402                        }
2403                }
2404                if (nvlist_lookup_nvlist(spa->spa_config,
2405                    ZPOOL_REWIND_POLICY, &policy) == 0)
2406                        VERIFY(nvlist_add_nvlist(nvconfig,
2407                            ZPOOL_REWIND_POLICY, policy) == 0);
2408
2409                spa_config_set(spa, nvconfig);
2410                spa_unload(spa);
2411                spa_deactivate(spa);
2412                spa_activate(spa, orig_mode);
2413
2414                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2415        }
2416
2417        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2418                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2419        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2420        if (error != 0)
2421                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2422
2423        /*
2424         * Load the bit that tells us to use the new accounting function
2425         * (raid-z deflation).  If we have an older pool, this will not
2426         * be present.
2427         */
2428        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2429        if (error != 0 && error != ENOENT)
2430                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2431
2432        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2433            &spa->spa_creation_version);
2434        if (error != 0 && error != ENOENT)
2435                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2436
2437        /*
2438         * Load the persistent error log.  If we have an older pool, this will
2439         * not be present.
2440         */
2441        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2442        if (error != 0 && error != ENOENT)
2443                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2444
2445        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2446            &spa->spa_errlog_scrub);
2447        if (error != 0 && error != ENOENT)
2448                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2449
2450        /*
2451         * Load the history object.  If we have an older pool, this
2452         * will not be present.
2453         */
2454        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2455        if (error != 0 && error != ENOENT)
2456                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
```

```
2458         /*
2459          * If we're assembling the pool from the split-off vdevs of
2460          * an existing pool, we don't want to attach the spares & cache
2461          * devices.
2462          */
2463
2464         /*
2465          * Load any hot spares for this pool.
2466          */
2467         error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2468         if (error != 0 && error != ENOENT)
2469                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2470         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2471                 ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2472                 if (load_nvlist(spa, spa->spa_spares.sav_object,
2473                     &spa->spa_spares.sav_config) != 0)
2474                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2475
2476                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2477                 spa_load_spares(spa);
2478                 spa_config_exit(spa, SCL_ALL, FTAG);
2479         } else if (error == 0) {
2480                 spa->spa_spares.sav_sync = B_TRUE;
2481         }
2482
2483         /*
2484          * Load any level 2 ARC devices for this pool.
2485          */
2486         error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2487             &spa->spa_l2cache.sav_object);
2488         if (error != 0 && error != ENOENT)
2489                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2490         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2491                 ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2492                 if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2493                     &spa->spa_l2cache.sav_config) != 0)
2494                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2495
2496                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2497                 spa_load_l2cache(spa);
2498                 spa_config_exit(spa, SCL_ALL, FTAG);
2499         } else if (error == 0) {
2500                 spa->spa_l2cache.sav_sync = B_TRUE;
2501         }
2502
2503         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
2504
2505         error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2506         if (error && error != ENOENT)
2507                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2508
2509         if (error == 0) {
2510                 uint64_t autoreplace;
2511
2512                 spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2513                 spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2514                 spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2515                 spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2516                 spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2517                 spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2518                     &spa->spa_dedup_ditto);
2519
2520                 spa->spa_autoreplace = (autoreplace != 0);
2521         }
```



```
2523         /*
2524          * If the 'autoreplace' property is set, then post a resource notifying
2525          * the ZFS DE that it should not issue any faults for unopenable
2526          * devices.  We also iterate over the vdevs, and post a sysevent for any
2527          * unopenable vdevs so that the normal autoreplace handler can take
2528          * over.
2529          */
2530         if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2531                 spa_check_removed(spa->spa_root_vdev);
2532                 /*
2533                  * For the import case, this is done in spa_import(), because
2534                  * at this point we're using the spare definitions from
2535                  * the MOS config, not necessarily from the userland config.
2536                  */
2537                 if (state != SPA_LOAD_IMPORT) {
2538                         spa_aux_check_removed(&spa->spa_spares);
2539                         spa_aux_check_removed(&spa->spa_l2cache);
2540                 }
2541         }
2542
2543         /*
2544          * Load the vdev state for all toplevel vdevs.
2545          */
2546         vdev_load(rvd);
2547
2548         /*
2549          * Propagate the leaf DTLs we just loaded all the way up the tree.
2550          */
2551         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2552         vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2553         spa_config_exit(spa, SCL_ALL, FTAG);
2554
2555         /*
2556          * Load the DDTs (dedup tables).
2557          */
2558         error = ddt_load(spa);
2559         if (error != 0)
2560                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2561
2562         spa_update_dspace(spa);
2563
2564         /*
2565          * Validate the config, using the MOS config to fill in any
2566          * information which might be missing.  If we fail to validate
2567          * the config then declare the pool unfit for use. If we're
2568          * assembling a pool from a split, the log is not transferred
2569          * over.
2570          */
2571         if (type != SPA_IMPORT_ASSEMBLE) {
2572                 nvlist_t *nvconfig;
2573
2574                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2575                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2576
2577                 if (!spa_config_valid(spa, nvconfig)) {
2578                         nvlist_free(nvconfig);
2579                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2580                             ENXIO));
2581                 }
2582                 nvlist_free(nvconfig);
2583
2584                 /*
2585                  * Now that we've validated the config, check the state of the
2586                  * root vdev.  If it can't be opened, it indicates one or
2587                  * more toplevel vdevs are faulted.
2588                  */
```

```
2589                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2590                        return (SET_ERROR(ENXIO));

2592                if (spa_check_logs(spa)) {
2593                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2594                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2595                }
2596        }

2598        if (missing_feat_write) {
2599                ASSERT(state == SPA_LOAD_TRYIMPORT);

2601                /*
2602                 * At this point, we know that we can open the pool in
2603                 * read-only mode but not read-write mode. We now have enough
2604                 * information and can return to userland.
2605                 */
2606                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2607        }

2609        /*
2610         * We've successfully opened the pool, verify that we're ready
2611         * to start pushing transactions.
2612         */
2613        if (state != SPA_LOAD_TRYIMPORT) {
2614                if (error = spa_load_verify(spa))
2615                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2616                            error));
2617        }

2619        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2620            spa->spa_load_max_txg == UINT64_MAX)) {
2621                dmu_tx_t *tx;
2622                int need_update = B_FALSE;

2624                ASSERT(state != SPA_LOAD_TRYIMPORT);

2626                /*
2627                 * Claim log blocks that haven't been committed yet.
2628                 * This must all happen in a single txg.
2629                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2630                 * invoked from zil_claim_log_block()'s i/o done callback.
2631                 * Price of rollback is that we abandon the log.
2632                 */
2633                spa->spa_claiming = B_TRUE;

2635                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2636                    spa_first_txg(spa));
2637                (void) dmu_objset_find(spa_name(spa),
2638                    zil_claim, tx, DS_FIND_CHILDREN);
2639                dmu_tx_commit(tx);

2641                spa->spa_claiming = B_FALSE;

2643                spa_set_log_state(spa, SPA_LOG_GOOD);
2644                spa->spa_sync_on = B_TRUE;
2645                txg_sync_start(spa->spa_dsl_pool);

2647                /*
2648                 * Wait for all claims to sync.  We sync up to the highest
2649                 * claimed log block birth time so that claimed log blocks
2650                 * don't appear to be from the future.  spa_claim_max_txg
2651                 * will have been set for us by either zil_check_log_chain()
2652                 * (invoked from spa_check_logs()) or zil_claim() above.
2653                 */
2654                txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);
```



```
2656                /*
2657                 * If the config cache is stale, or we have uninitialized
2658                 * metaslabs (see spa_vdev_add()), then update the config.
2659                 *
2660                 * If this is a verbatim import, trust the current
2661                 * in-core spa_config and update the disk labels.
2662                 */
2663                if (config_cache_txg != spa->spa_config_txg ||
2664                    state == SPA_LOAD_IMPORT ||
2665                    state == SPA_LOAD_RECOVER ||
2666                    (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2667                        need_update = B_TRUE;

2669                for (int c = 0; c < rvd->vdev_children; c++)
2670                        if (rvd->vdev_child[c]->vdev_ms_array == 0)
2671                                need_update = B_TRUE;

2673                /*
2674                 * Update the config cache asychronously in case we're the
2675                 * root pool, in which case the config cache isn't writable yet.
2676                 */
2677                if (need_update)
2678                        spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2680                /*
2681                 * Check all DTLs to see if anything needs resilvering.
2682                 */
2683                if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2684                    vdev_resilver_needed(rvd, NULL, NULL))
2685                        spa_async_request(spa, SPA_ASYNC_RESILVER);

2687                /*
2688                 * Log the fact that we booted up (so that we can detect if
2689                 * we rebooted in the middle of an operation).
2690                 */
2691                spa_history_log_version(spa, "open");

2693                /*
2694                 * Delete any inconsistent datasets.
2695                 */
2696                (void) dmu_objset_find(spa_name(spa),
2697                    dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2699                /*
2700                 * Clean up any stale temporary dataset userrefs.
2701                 */
2702                dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2703        }

2705        spa_async_request(spa, SPA_ASYNC_L2CACHE_REBUILD);

2707        return (0);
2708 }
______unchanged_portion_omitted_

5580 static void
5581 spa_async_thread(spa_t *spa)
5582 {
5583        int tasks;

5585        ASSERT(spa->spa_sync_on);

5587        mutex_enter(&spa->spa_async_lock);
5588        tasks = spa->spa_async_tasks;
5589        spa->spa_async_tasks = 0;
```

```
5590         mutex_exit(&spa->spa_async_lock);
5591
5592         /*
5593          * See if the config needs to be updated.
5594          */
5595         if (tasks & SPA_ASYNC_CONFIG_UPDATE) {
5596                 uint64_t old_space, new_space;
5597
5598                 mutex_enter(&spa_namespace_lock);
5599                 old_space = metaslab_class_get_space(spa_normal_class(spa));
5600                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
5601                 new_space = metaslab_class_get_space(spa_normal_class(spa));
5602                 mutex_exit(&spa_namespace_lock);
5603
5604                 /*
5605                  * If the pool grew as a result of the config update,
5606                  * then log an internal history event.
5607                  */
5608                 if (new_space != old_space) {
5609                         spa_history_log_internal(spa, "vdev online", NULL,
5610                             "pool '%s' size: %llu(+%llu)",
5611                             spa_name(spa), new_space, new_space - old_space);
5612                 }
5613         }
5614
5615         /*
5616          * See if any devices need to be marked REMOVED.
5617          */
5618         if (tasks & SPA_ASYNC_REMOVE) {
5619                 spa_vdev_state_enter(spa, SCL_NONE);
5620                 spa_async_remove(spa, spa->spa_root_vdev);
5621                 for (int i = 0; i < spa->spa_l2cache.sav_count; i++)
5622                         spa_async_remove(spa, spa->spa_l2cache.sav_vdevs[i]);
5623                 for (int i = 0; i < spa->spa_spares.sav_count; i++)
5624                         spa_async_remove(spa, spa->spa_spares.sav_vdevs[i]);
5625                 (void) spa_vdev_state_exit(spa, NULL, 0);
5626         }
5627
5628         if ((tasks & SPA_ASYNC_AUTOEXPAND) && !spa_suspended(spa)) {
5629                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
5630                 spa_async_autoexpand(spa, spa->spa_root_vdev);
5631                 spa_config_exit(spa, SCL_CONFIG, FTAG);
5632         }
5633
5634         /*
5635          * See if any devices need to be probed.
5636          */
5637         if (tasks & SPA_ASYNC_PROBE) {
5638                 spa_vdev_state_enter(spa, SCL_NONE);
5639                 spa_async_probe(spa, spa->spa_root_vdev);
5640                 (void) spa_vdev_state_exit(spa, NULL, 0);
5641         }
5642
5643         /*
5644          * If any devices are done replacing, detach them.
5645          */
5646         if (tasks & SPA_ASYNC_RESILVER_DONE)
5647                 spa_vdev_resilver_done(spa);
5648
5649         /*
5650          * Kick off a resilver.
5651          */
5652         if (tasks & SPA_ASYNC_RESILVER)
5653                 dsl_resilver_restart(spa->spa_dsl_pool, 0);
5654
5655         /*
```



```
5656          * Kick off L2 cache rebuilding.
5657          */
5658         if (tasks & SPA_ASYNC_L2CACHE_REBUILD)
5659                 l2arc_spa_rebuild_start(spa);
5660
5661         /*
5662          * Let the world know that we're done.
5663          */
5664         mutex_enter(&spa->spa_async_lock);
5665         spa->spa_async_thread = NULL;
5666         cv_broadcast(&spa->spa_async_cv);
5667         mutex_exit(&spa->spa_async_lock);
5668         thread_exit();
5669 }
```

*unchanged_portion_omitted_*

****
4611 Mon Dec 9 16:07:36 2013
new/usr/src/uts/common/fs/zfs/sys/arc.h
3525 Persistent L2ARC
****
_____unchanged_portion_omitted_

```
  84 void arc_space_consume(uint64_t space, arc_space_type_t type);
  85 void arc_space_return(uint64_t space, arc_space_type_t type);
  86 void *arc_data_buf_alloc(uint64_t space);
  87 void arc_data_buf_free(void *buf, uint64_t space);
  88 arc_buf_t *arc_buf_alloc(spa_t *spa, int size, void *tag,
  89     arc_buf_contents_t type);
  90 arc_buf_t *arc_loan_buf(spa_t *spa, int size);
  91 void arc_return_buf(arc_buf_t *buf, void *tag);
  92 void arc_loan_inuse_buf(arc_buf_t *buf, void *tag);
  93 void arc_buf_add_ref(arc_buf_t *buf, void *tag);
  94 boolean_t arc_buf_remove_ref(arc_buf_t *buf, void *tag);
  95 int arc_buf_size(arc_buf_t *buf);
  96 void arc_release(arc_buf_t *buf, void *tag);
  97 int arc_released(arc_buf_t *buf);
  98 int arc_has_callback(arc_buf_t *buf);
  99 void arc_buf_freeze(arc_buf_t *buf);
 100 void arc_buf_thaw(arc_buf_t *buf);
 101 boolean_t arc_buf_eviction_needed(arc_buf_t *buf);
 102 #ifdef ZFS_DEBUG
 103 int arc_referenced(arc_buf_t *buf);
 104 #endif

 106 int arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp,
 107     arc_done_func_t *done, void *private, zio_priority_t priority, int flags,
 108     uint32_t *arc_flags, const zbookmark_t *zb);
 109 zio_t *arc_write(zio_t *pio, spa_t *spa, uint64_t txg,
 110     blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, boolean_t l2arc_compress,
 111     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *physdone,
 112     arc_done_func_t *done, void *private, zio_priority_t priority,
 113     int zio_flags, const zbookmark_t *zb);
 114 void arc_freed(spa_t *spa, const blkptr_t *bp);

 116 void arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private);
 117 int arc_buf_evict(arc_buf_t *buf);

 119 void arc_flush(spa_t *spa);
 120 void arc_tempreserve_clear(uint64_t reserve);
 121 int arc_tempreserve_space(uint64_t reserve, uint64_t txg);

 123 void arc_init(void);
 124 void arc_fini(void);

 126 /*
 127  * Level 2 ARC
 128  */

 130 void l2arc_add_vdev(spa_t *spa, vdev_t *vd, boolean_t rebuild);
 130 void l2arc_add_vdev(spa_t *spa, vdev_t *vd);
 131 void l2arc_remove_vdev(vdev_t *vd);
 132 boolean_t l2arc_vdev_present(vdev_t *vd);
 133 void l2arc_init(void);
 134 void l2arc_fini(void);
 135 void l2arc_start(void);
 136 void l2arc_stop(void);
 137 void l2arc_spa_rebuild_start(spa_t *spa);

 139 #ifndef _KERNEL
 140 extern boolean_t arc_watch;
 141 extern int arc_procfd;
```



```
 142 #endif

 144 #ifdef  __cplusplus
 145 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   26017 Mon Dec  9 16:07:37 2013
new/usr/src/uts/common/fs/zfs/sys/spa.h
3525 Persistent L2ARC
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2013 Saso Kiselkov. All rights reserved.
 */

#ifndef _SYS_SPA_H
#define _SYS_SPA_H

#include <sys/avl.h>
#include <sys/zfs_context.h>
#include <sys/nvpair.h>
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/fs/zfs.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Forward references that lots of things need.
 */
typedef struct spa spa_t;
typedef struct vdev vdev_t;
typedef struct metaslab metaslab_t;
typedef struct metaslab_group metaslab_group_t;
typedef struct metaslab_class metaslab_class_t;
typedef struct zio zio_t;
typedef struct zilog zilog_t;
typedef struct spa_aux_vdev spa_aux_vdev_t;
typedef struct ddt ddt_t;
typedef struct ddt_entry ddt_entry_t;
struct dsl_pool;
struct dsl_dataset;

/*
 * General-purpose 32-bit and 64-bit bitfield encodings.
 */
#define BF32_DECODE(x, low, len)        P2PHASE((x) >> (low), 1U << (len))
```



```
#define BF64_DECODE(x, low, len)        P2PHASE((x) >> (low), 1ULL << (len))
#define BF32_ENCODE(x, low, len)        (P2PHASE((x), 1U << (len)) << (low))
#define BF64_ENCODE(x, low, len)        (P2PHASE((x), 1ULL << (len)) << (low))

#define BF32_GET(x, low, len)           BF32_DECODE(x, low, len)
#define BF64_GET(x, low, len)           BF64_DECODE(x, low, len)

#define BF32_SET(x, low, len, val)      \
        ((x) ^= BF32_ENCODE((x >> low) ^ (val), low, len))
#define BF64_SET(x, low, len, val)      \
        ((x) ^= BF64_ENCODE((x >> low) ^ (val), low, len))

#define BF32_GET_SB(x, low, len, shift, bias)   \
        ((BF32_GET(x, low, len) + (bias)) << (shift))
#define BF64_GET_SB(x, low, len, shift, bias)   \
        ((BF64_GET(x, low, len) + (bias)) << (shift))

#define BF32_SET_SB(x, low, len, shift, bias, val)      \
        BF32_SET(x, low, len, ((val) >> (shift)) - (bias))
#define BF64_SET_SB(x, low, len, shift, bias, val)      \
        BF64_SET(x, low, len, ((val) >> (shift)) - (bias))

/*
 * We currently support nine block sizes, from 512 bytes to 128K.
 * We could go higher, but the benefits are near-zero and the cost
 * of COWing a giant block to modify one byte would become excessive.
 */
#define SPA_MINBLOCKSHIFT       9
#define SPA_MAXBLOCKSHIFT       17
#define SPA_MINBLOCKSIZE        (1ULL << SPA_MINBLOCKSHIFT)
#define SPA_MAXBLOCKSIZE        (1ULL << SPA_MAXBLOCKSHIFT)

#define SPA_BLOCKSIZES          (SPA_MAXBLOCKSHIFT - SPA_MINBLOCKSHIFT + 1)

/*
 * Size of block to hold the configuration data (a packed nvlist)
 */
#define SPA_CONFIG_BLOCKSIZE    (1ULL << 14)

/*
 * The DVA size encodings for LSIZE and PSIZE support blocks up to 32MB.
 * The ASIZE encoding should be at least 64 times larger (6 more bits)
 * to support up to 4-way RAID-Z mirror mode with worst-case gang block
 * overhead, three DVAs per bp, plus one more bit in case we do anything
 * else that expands the ASIZE.
 */
#define SPA_LSIZEBITS           16      /* LSIZE up to 32M (2^16 * 512) */
#define SPA_PSIZEBITS           16      /* PSIZE up to 32M (2^16 * 512) */
#define SPA_ASIZEBITS           24      /* ASIZE up to 64 times larger  */

/*
 * All SPA data is represented by 128-bit data virtual addresses (DVAs).
 * The members of the dva_t should be considered opaque outside the SPA.
 */
typedef struct dva {
        uint64_t        dva_word[2];
} dva_t;
```

*unchanged_portion_omitted_*

```
#define BP_GET_ASIZE(bp)        \
        (DVA_GET_ASIZE(&(bp)->blk_dva[0]) + DVA_GET_ASIZE(&(bp)->blk_dva[1]) + \
                DVA_GET_ASIZE(&(bp)->blk_dva[2]))

#define BP_GET_UCSIZE(bp) \
        ((BP_GET_LEVEL(bp) > 0 || DMU_OT_IS_METADATA(BP_GET_TYPE(bp))) ? \
        BP_GET_PSIZE(bp) : BP_GET_LSIZE(bp))
```

```
270 #define BP_GET_NDVAS(bp)        \
271         (!!DVA_GET_ASIZE(&(bp)->blk_dva[0]) + \
272         !!DVA_GET_ASIZE(&(bp)->blk_dva[1]) + \
273         !!DVA_GET_ASIZE(&(bp)->blk_dva[2]))

275 #define BP_COUNT_GANG(bp)       \
276         (DVA_GET_GANG(&(bp)->blk_dva[0]) + \
277         DVA_GET_GANG(&(bp)->blk_dva[1]) + \
278         DVA_GET_GANG(&(bp)->blk_dva[2]))

280 #define DVA_EQUAL(dva1, dva2)   \
281         ((dva1)->dva_word[1] == (dva2)->dva_word[1] && \
282         (dva1)->dva_word[0] == (dva2)->dva_word[0])

284 #define BP_EQUAL(bp1, bp2)      \
285         (BP_PHYSICAL_BIRTH(bp1) == BP_PHYSICAL_BIRTH(bp2) &&    \
286         DVA_EQUAL(&(bp1)->blk_dva[0], &(bp2)->blk_dva[0]) &&    \
287         DVA_EQUAL(&(bp1)->blk_dva[1], &(bp2)->blk_dva[1]) &&    \
288         DVA_EQUAL(&(bp1)->blk_dva[2], &(bp2)->blk_dva[2]))

290 #define ZIO_CHECKSUM_EQUAL(zc1, zc2) \
291         (0 == (((zc1).zc_word[0] - (zc2).zc_word[0]) | \
292         ((zc1).zc_word[1] - (zc2).zc_word[1]) | \
293         ((zc1).zc_word[2] - (zc2).zc_word[2]) | \
294         ((zc1).zc_word[3] - (zc2).zc_word[3])))

296 #define ZIO_CHECKSUM_BSWAP(_zc) \
297         do { \
298                 zio_cksum_t *zc = (_zc); \
299                 zc->zc_word[0] = BSWAP_64(zc->zc_word[0]); \
300                 zc->zc_word[1] = BSWAP_64(zc->zc_word[1]); \
301                 zc->zc_word[2] = BSWAP_64(zc->zc_word[2]); \
302                 zc->zc_word[3] = BSWAP_64(zc->zc_word[3]); \
303                 _NOTE(NOTREACHED) \
304                 _NOTE(CONSTCOND) \
305         } while (0)

307 #define DVA_IS_VALID(dva)       (DVA_GET_ASIZE(dva) != 0)

309 #define ZIO_SET_CHECKSUM(zcp, w0, w1, w2, w3)   \
310 {                                               \
311         (zcp)->zc_word[0] = w0;                 \
312         (zcp)->zc_word[1] = w1;                 \
313         (zcp)->zc_word[2] = w2;                 \
314         (zcp)->zc_word[3] = w3;                 \
315 }
```

*unchanged_portion_omitted_*

```
427 /* state manipulation functions */
428 extern int spa_open(const char *pool, spa_t **, void *tag);
429 extern int spa_open_rewind(const char *pool, spa_t **, void *tag,
430     nvlist_t *policy, nvlist_t **config);
431 extern int spa_get_stats(const char *pool, nvlist_t **config, char *altroot,
432     size_t buflen);
433 extern int spa_create(const char *pool, nvlist_t *config, nvlist_t *props,
434     nvlist_t *zplprops);
435 extern int spa_import_rootpool(char *devpath, char *devid);
436 extern int spa_import(const char *pool, nvlist_t *config, nvlist_t *props,
437     uint64_t flags);
438 extern nvlist_t *spa_tryimport(nvlist_t *tryconfig);
439 extern int spa_destroy(char *pool);
440 extern int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
441     boolean_t hardforce);
442 extern int spa_reset(char *pool);
443 extern void spa_async_request(spa_t *spa, int flag);
```



```
444 extern void spa_async_unrequest(spa_t *spa, int flag);
445 extern void spa_async_suspend(spa_t *spa);
446 extern void spa_async_resume(spa_t *spa);
447 extern spa_t *spa_inject_addref(char *pool);
448 extern void spa_inject_delref(spa_t *spa);
449 extern void spa_scan_stat_init(spa_t *spa);
450 extern int spa_scan_get_stats(spa_t *spa, pool_scan_stat_t *ps);

452 #define SPA_ASYNC_CONFIG_UPDATE         0x01
453 #define SPA_ASYNC_REMOVE                0x02
454 #define SPA_ASYNC_PROBE                 0x04
455 #define SPA_ASYNC_RESILVER_DONE         0x08
456 #define SPA_ASYNC_RESILVER              0x10
457 #define SPA_ASYNC_AUTOEXPAND            0x20
458 #define SPA_ASYNC_REMOVE_DONE           0x40
459 #define SPA_ASYNC_REMOVE_STOP           0x80
460 #define SPA_ASYNC_L2CACHE_REBUILD       0x100

462 /*
463  * Controls the behavior of spa_vdev_remove().
464  */
465 #define SPA_REMOVE_UNSPARE      0x01
466 #define SPA_REMOVE_DONE         0x02

468 /* device manipulation */
469 extern int spa_vdev_add(spa_t *spa, nvlist_t *nvroot);
470 extern int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot,
471     int replacing);
472 extern int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid,
473     int replace_done);
474 extern int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare);
475 extern boolean_t spa_vdev_remove_active(spa_t *spa);
476 extern int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath);
477 extern int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru);
478 extern int spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
479     nvlist_t *props, boolean_t exp);

481 /* spare state (which is global across all pools) */
482 extern void spa_spare_add(vdev_t *vd);
483 extern void spa_spare_remove(vdev_t *vd);
484 extern boolean_t spa_spare_exists(uint64_t guid, uint64_t *pool, int *refcnt);
485 extern void spa_spare_activate(vdev_t *vd);

487 /* L2ARC state (which is global across all pools) */
488 extern void spa_l2cache_add(vdev_t *vd);
489 extern void spa_l2cache_remove(vdev_t *vd);
490 extern boolean_t spa_l2cache_exists(uint64_t guid, uint64_t *pool);
491 extern void spa_l2cache_activate(vdev_t *vd);
492 extern void spa_l2cache_drop(spa_t *spa);

494 /* scanning */
495 extern int spa_scan(spa_t *spa, pool_scan_func_t func);
496 extern int spa_scan_stop(spa_t *spa);

498 /* spa syncing */
499 extern void spa_sync(spa_t *spa, uint64_t txg); /* only for DMU use */
500 extern void spa_sync_allpools(void);

502 /* spa namespace global mutex */
503 extern kmutex_t spa_namespace_lock;

505 /*
506  * SPA configuration functions in spa_config.c
507  */

509 #define SPA_CONFIG_UPDATE_POOL  0
```

```
 510 #define SPA_CONFIG_UPDATE_VDEVS 1

 512 extern void spa_config_sync(spa_t *, boolean_t, boolean_t);
 513 extern void spa_config_load(void);
 514 extern nvlist_t *spa_all_configs(uint64_t *);
 515 extern void spa_config_set(spa_t *spa, nvlist_t *config);
 516 extern nvlist_t *spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg,
 517     int getstats);
 518 extern void spa_config_update(spa_t *spa, int what);

 520 /*
 521  * Miscellaneous SPA routines in spa_misc.c
 522  */

 524 /* Namespace manipulation */
 525 extern spa_t *spa_lookup(const char *name);
 526 extern spa_t *spa_add(const char *name, nvlist_t *config, const char *altroot);
 527 extern void spa_remove(spa_t *spa);
 528 extern spa_t *spa_next(spa_t *prev);

 530 /* Refcount functions */
 531 extern void spa_open_ref(spa_t *spa, void *tag);
 532 extern void spa_close(spa_t *spa, void *tag);
 533 extern boolean_t spa_refcount_zero(spa_t *spa);

 535 #define SCL_NONE        0x00
 536 #define SCL_CONFIG      0x01
 537 #define SCL_STATE       0x02
 538 #define SCL_L2ARC       0x04            /* hack until L2ARC 2.0 */
 539 #define SCL_ALLOC       0x08
 540 #define SCL_ZIO         0x10
 541 #define SCL_FREE        0x20
 542 #define SCL_VDEV        0x40
 543 #define SCL_LOCKS       7
 544 #define SCL_ALL         ((1 << SCL_LOCKS) - 1)
 545 #define SCL_STATE_ALL   (SCL_STATE | SCL_L2ARC | SCL_ZIO)

 547 /* Pool configuration locks */
 548 extern int spa_config_tryenter(spa_t *spa, int locks, void *tag, krw_t rw);
 549 extern void spa_config_enter(spa_t *spa, int locks, void *tag, krw_t rw);
 550 extern void spa_config_exit(spa_t *spa, int locks, void *tag);
 551 extern int spa_config_held(spa_t *spa, int locks, krw_t rw);

 553 /* Pool vdev add/remove lock */
 554 extern uint64_t spa_vdev_enter(spa_t *spa);
 555 extern uint64_t spa_vdev_config_enter(spa_t *spa);
 556 extern void spa_vdev_config_exit(spa_t *spa, vdev_t *vd, uint64_t txg,
 557     int error, char *tag);
 558 extern int spa_vdev_exit(spa_t *spa, vdev_t *vd, uint64_t txg, int error);

 560 /* Pool vdev state change lock */
 561 extern void spa_vdev_state_enter(spa_t *spa, int oplock);
 562 extern int spa_vdev_state_exit(spa_t *spa, vdev_t *vd, int error);

 564 /* Log state */
 565 typedef enum spa_log_state {
 566         SPA_LOG_UNKNOWN = 0,    /* unknown log state */
 567         SPA_LOG_MISSING,        /* missing log(s) */
 568         SPA_LOG_CLEAR,          /* clear the log(s) */
 569         SPA_LOG_GOOD,           /* log(s) are good */
 570 } spa_log_state_t;
_____unchanged_portion_omitted_
```

```
**********************************************************
   89277 Mon Dec  9 16:07:37 2013
new/usr/src/uts/common/fs/zfs/vdev.c
3525 Persistent L2ARC
**********************************************************
______unchanged_portion_omitted_

1481 /*
1482  * Reopen all interior vdevs and any unopened leaves.  We don't actually
1483  * reopen leaf vdevs which had previously been opened as they might deadlock
1484  * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
1485  * If the leaf has never been opened then open it, as usual.
1486  */
1487 void
1488 vdev_reopen(vdev_t *vd)
1489 {
1490         spa_t *spa = vd->vdev_spa;

1492         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1494         /* set the reopening flag unless we're taking the vdev offline */
1495         vd->vdev_reopening = !vd->vdev_offline;
1496         vdev_close(vd);
1497         (void) vdev_open(vd);

1499         /*
1500          * Call vdev_validate() here to make sure we have the same device.
1501          * Otherwise, a device with an invalid label could be successfully
1502          * opened in response to vdev_reopen().
1503          */
1504         if (vd->vdev_aux) {
1505                 (void) vdev_validate_aux(vd);
1506                 if (vdev_readable(vd) && vdev_writeable(vd) &&
1507                     vd->vdev_aux == &spa->spa_l2cache &&
1508                     !l2arc_vdev_present(vd)) {
1509                         /*
1510                          * When reopening we can assume persistent L2ARC is
1511                          * supported, since we've already opened the device
1512                          * in the past and prepended an L2ARC uberblock.
1513                          */
1514                         l2arc_add_vdev(spa, vd, B_TRUE);
1515                 }
1508                     !l2arc_vdev_present(vd))
1509                         l2arc_add_vdev(spa, vd);
1516         } else {
1517                 (void) vdev_validate(vd, B_TRUE);
1518         }

1520         /*
1521          * Reassess parent vdev's health.
1522          */
1523         vdev_propagate_state(vd);
1524 }
______unchanged_portion_omitted_
```

```
**********************************************************
   37670 Mon Dec  9 16:07:37 2013
new/usr/src/uts/common/fs/zfs/vdev_label.c
3525 Persistent L2ARC
**********************************************************
______unchanged_portion_omitted_

 210 /*
 211  * Generate the nvlist representing this vdev's config.
 212  */
 213 nvlist_t *
 214 vdev_config_generate(spa_t *spa, vdev_t *vd, boolean_t getstats,
 215     vdev_config_flag_t flags)
 216 {
 217         nvlist_t *nv = NULL;

 219         nv = fnvlist_alloc();

 221         fnvlist_add_string(nv, ZPOOL_CONFIG_TYPE, vd->vdev_ops->vdev_op_type);
 222         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)))
 223                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ID, vd->vdev_id);
 224         fnvlist_add_uint64(nv, ZPOOL_CONFIG_GUID, vd->vdev_guid);

 226         if (vd->vdev_path != NULL)
 227                 fnvlist_add_string(nv, ZPOOL_CONFIG_PATH, vd->vdev_path);

 229         if (vd->vdev_devid != NULL)
 230                 fnvlist_add_string(nv, ZPOOL_CONFIG_DEVID, vd->vdev_devid);

 232         if (vd->vdev_physpath != NULL)
 233                 fnvlist_add_string(nv, ZPOOL_CONFIG_PHYS_PATH,
 234                     vd->vdev_physpath);

 236         if (vd->vdev_fru != NULL)
 237                 fnvlist_add_string(nv, ZPOOL_CONFIG_FRU, vd->vdev_fru);

 239         if (vd->vdev_nparity != 0) {
 240                 ASSERT(strcmp(vd->vdev_ops->vdev_op_type,
 241                     VDEV_TYPE_RAIDZ) == 0);

 243                 /*
 244                  * Make sure someone hasn't managed to sneak a fancy new vdev
 245                  * into a crufty old storage pool.
 246                  */
 247                 ASSERT(vd->vdev_nparity == 1 ||
 248                     (vd->vdev_nparity <= 2 &&
 249                     spa_version(spa) >= SPA_VERSION_RAIDZ2) ||
 250                     (vd->vdev_nparity <= 3 &&
 251                     spa_version(spa) >= SPA_VERSION_RAIDZ3));

 253                 /*
 254                  * Note that we'll add the nparity tag even on storage pools
 255                  * that only support a single parity device -- older software
 256                  * will just ignore it.
 257                  */
 258                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_NPARITY, vd->vdev_nparity);
 259         }

 261         if (vd->vdev_wholedisk != -1ULL)
 262                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
 263                     vd->vdev_wholedisk);

 265         if (vd->vdev_not_present)
 266                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT, 1);

 268         if (vd->vdev_isspare)
```



```
 269                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_SPARE, 1);

 271         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)) &&
 272             vd == vd->vdev_top) {
 273                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
 274                     vd->vdev_ms_array);
 275                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
 276                     vd->vdev_ms_shift);
 277                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ASHIFT, vd->vdev_ashift);
 278                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_ASIZE,
 279                     vd->vdev_asize);
 280                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_LOG, vd->vdev_islog);
 281                 if (vd->vdev_removing)
 282                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVING,
 283                             vd->vdev_removing);
 284         }

 286         if (flags & VDEV_CONFIG_L2CACHE)
 287                 /* indicate that we support L2ARC persistency */
 288                 VERIFY(nvlist_add_boolean_value(nv,
 289                     ZPOOL_CONFIG_L2CACHE_PERSISTENT, B_TRUE) == 0);

 291         if (vd->vdev_dtl_sm != NULL) {
 292                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_DTL,
 293                     space_map_object(vd->vdev_dtl_sm));
 294         }

 296         if (vd->vdev_crtxg)
 297                 fnvlist_add_uint64(nv, ZPOOL_CONFIG_CREATE_TXG, vd->vdev_crtxg);

 299         if (getstats) {
 300                 vdev_stat_t vs;
 301                 pool_scan_stat_t ps;

 303                 vdev_get_stats(vd, &vs);
 304                 fnvlist_add_uint64_array(nv, ZPOOL_CONFIG_VDEV_STATS,
 305                     (uint64_t *)&vs, sizeof (vs) / sizeof (uint64_t));

 307                 /* provide either current or previous scan information */
 308                 if (spa_scan_get_stats(spa, &ps) == 0) {
 309                         fnvlist_add_uint64_array(nv,
 310                             ZPOOL_CONFIG_SCAN_STATS, (uint64_t *)&ps,
 311                             sizeof (pool_scan_stat_t) / sizeof (uint64_t));
 312                 }
 313         }

 315         if (!vd->vdev_ops->vdev_op_leaf) {
 316                 nvlist_t **child;
 317                 int c, idx;

 319                 ASSERT(!vd->vdev_ishole);

 321                 child = kmem_alloc(vd->vdev_children * sizeof (nvlist_t *),
 322                     KM_SLEEP);

 324                 for (c = 0, idx = 0; c < vd->vdev_children; c++) {
 325                         vdev_t *cvd = vd->vdev_child[c];

 327                         /*
 328                          * If we're generating an nvlist of removing
 329                          * vdevs then skip over any device which is
 330                          * not being removed.
 331                          */
 332                         if ((flags & VDEV_CONFIG_REMOVING) &&
 333                             !cvd->vdev_removing)
 334                                 continue;
```

```
336                         child[idx++] = vdev_config_generate(spa, cvd,
337                             getstats, flags);
338                 }
339
340                 if (idx) {
341                         fnvlist_add_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
342                             child, idx);
343                 }
344
345                 for (c = 0; c < idx; c++)
346                         nvlist_free(child[c]);
347
348                 kmem_free(child, vd->vdev_children * sizeof (nvlist_t *));
349
350         } else {
351                 const char *aux = NULL;
352
353                 if (vd->vdev_offline && !vd->vdev_tmpoffline)
354                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_OFFLINE, B_TRUE);
355                 if (vd->vdev_resilver_txg != 0)
356                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_RESILVER_TXG,
357                             vd->vdev_resilver_txg);
358                 if (vd->vdev_faulted)
359                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_FAULTED, B_TRUE);
360                 if (vd->vdev_degraded)
361                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_DEGRADED, B_TRUE);
362                 if (vd->vdev_removed)
363                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVED, B_TRUE);
364                 if (vd->vdev_unspare)
365                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_UNSPARE, B_TRUE);
366                 if (vd->vdev_ishole)
367                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_IS_HOLE, B_TRUE);
368
369                 switch (vd->vdev_stat.vs_aux) {
370                 case VDEV_AUX_ERR_EXCEEDED:
371                         aux = "err_exceeded";
372                         break;
373
374                 case VDEV_AUX_EXTERNAL:
375                         aux = "external";
376                         break;
377                 }
378
379                 if (aux != NULL)
380                         fnvlist_add_string(nv, ZPOOL_CONFIG_AUX_STATE, aux);
381
382                 if (vd->vdev_splitting && vd->vdev_orig_guid != 0LL) {
383                         fnvlist_add_uint64(nv, ZPOOL_CONFIG_ORIG_GUID,
384                             vd->vdev_orig_guid);
385                 }
386         }
387
388         return (nv);
389 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    4730 Mon Dec  9 16:07:37 2013
new/usr/src/uts/common/sys/debug.h
Reworking of L2ARC persistency to simplify design.
**********************************************************
```

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

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2013 Saso Kiselkov. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#ifndef _SYS_DEBUG_H
#define _SYS_DEBUG_H

#include <sys/isa_defs.h>
#include <sys/types.h>
#include <sys/note.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * ASSERT(ex) causes a panic or debugger entry if expression ex is not
 * true.  ASSERT() is included only for debugging, and is a no-op in
 * production kernels.  VERIFY(ex), on the other hand, behaves like
 * ASSERT and is evaluated on both debug and non-debug kernels.
 */

#if defined(__STDC__)
extern int assfail(const char *, const char *, int);
#define VERIFY(EX) ((void)((EX) || assfail(#EX, __FILE__, __LINE__)))
#if DEBUG
#define ASSERT(EX) ((void)((EX) || assfail(#EX, __FILE__, __LINE__)))
#else
#define ASSERT(x)  ((void)0)
#endif
#else   /* defined(__STDC__) */
extern int assfail();
```



```
#define VERIFY(EX) ((void)((EX) || assfail("EX", __FILE__, __LINE__)))
#if DEBUG
#define ASSERT(EX) ((void)((EX) || assfail("EX", __FILE__, __LINE__)))
#else
#define ASSERT(x)  ((void)0)
#endif
#endif  /* defined(__STDC__) */

/*
 * Assertion variants sensitive to the compilation data model
 */
#if defined(_LP64)
#define ASSERT64(x)     ASSERT(x)
#define ASSERT32(x)
#else
#define ASSERT64(x)
#define ASSERT32(x)     ASSERT(x)
#endif

/*
 * IMPLY and EQUIV are assertions of the form:
 *
 *      if (a) then (b)
 * and
 *      if (a) then (b) *AND* if (b) then (a)
 */
#if DEBUG
#define IMPLY(A, B) \
        ((void)(((!(A)) || (B)) || \
            assfail("(" #A ") implies (" #B ")", __FILE__, __LINE__)))
#define EQUIV(A, B) \
        ((void)((!!(A) == !!(B)) || \
            assfail("(" #A ") is equivalent to (" #B ")", __FILE__, __LINE__)))
#else
#define IMPLY(A, B) ((void)0)
#define EQUIV(A, B) ((void)0)
#endif

/*
 * ASSERT3() behaves like ASSERT() except that it is an explicit conditional,
 * and prints out the values of the left and right hand expressions as part of
 * the panic message to ease debugging.  The three variants imply the type
 * of their arguments.  ASSERT3S() is for signed data types, ASSERT3U() is
 * for unsigned, and ASSERT3P() is for pointers.  The VERIFY3*() macros
 * have the same relationship as above.
 */
extern void assfail3(const char *, uintmax_t, const char *, uintmax_t,
    const char *, int);
#define VERIFY3_IMPL(LEFT, OP, RIGHT, TYPE) do { \
        const TYPE __left = (TYPE)(LEFT); \
        const TYPE __right = (TYPE)(RIGHT); \
        if (!(__left OP __right)) \
                assfail3(#LEFT " " #OP " " #RIGHT, \
                        (uintmax_t)__left, #OP, (uintmax_t)__right, \
                        __FILE__, __LINE__); \
_NOTE(CONSTCOND) } while (0)

#define VERIFY3S(x, y, z)       VERIFY3_IMPL(x, y, z, int64_t)
#define VERIFY3U(x, y, z)       VERIFY3_IMPL(x, y, z, uint64_t)
#define VERIFY3P(x, y, z)       VERIFY3_IMPL(x, y, z, uintptr_t)
#define VERIFY0(x)              VERIFY3_IMPL(x, ==, 0, uintmax_t)

#if DEBUG
#define ASSERT3S(x, y, z)       VERIFY3_IMPL(x, y, z, int64_t)
#define ASSERT3U(x, y, z)       VERIFY3_IMPL(x, y, z, uint64_t)
#define ASSERT3P(x, y, z)       VERIFY3_IMPL(x, y, z, uintptr_t)
```

```
#define ASSERT0(x)              VERIFY3_IMPL(x, ==, 0, uintmax_t)
#else
#define ASSERT3S(x, y, z)       ((void)0)
#define ASSERT3U(x, y, z)       ((void)0)
#define ASSERT3P(x, y, z)       ((void)0)
#define ASSERT0(x)              ((void)0)
#endif

/*
 * Compile-time assertion. The condition 'x' must be constant.
 */
#define CTASSERT(x)             _CTASSERT(x, __LINE__)
#define _CTASSERT(x, y)         __CTASSERT(x, y)
#define __CTASSERT(x, y) \
        typedef char __compile_time_assertion__ ## y [(x) ? 1 : -1]

#ifdef  _KERNEL

extern void abort_sequence_enter(char *);
extern void debug_enter(char *);

#endif  /* _KERNEL */

#if defined(DEBUG) && !defined(__sun)
/* CSTYLED */
#define STATIC
#else
/* CSTYLED */
#define STATIC static
#endif

#ifdef  __cplusplus
}
```

*unchanged_portion_omitted_*

```
**********************************************************
   29033 Mon Dec  9 16:07:37 2013
new/usr/src/uts/common/sys/fs/zfs.h
3525 Persistent L2ARC
**********************************************************
```

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

  22 /*
  23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright (c) 2013 by Delphix. All rights reserved.
  25  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  26  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
  27  * Copyright (c) 2013, Saso Kiselkov. All rights reserved.
  28  */

  30 /* Portions Copyright 2010 Robert Milkowski */

  32 #ifndef _SYS_FS_ZFS_H
  33 #define _SYS_FS_ZFS_H

  35 #include <sys/time.h>

  37 #ifdef  __cplusplus
  38 extern "C" {
  39 #endif

  41 /*
  42  * Types and constants shared between userland and the kernel.
  43  */

  45 /*
  46  * Each dataset can be one of the following types.  These constants can be
  47  * combined into masks that can be passed to various functions.
  48  */
  49 typedef enum {
  50         ZFS_TYPE_FILESYSTEM     = 0x1,
  51         ZFS_TYPE_SNAPSHOT       = 0x2,
  52         ZFS_TYPE_VOLUME         = 0x4,
  53         ZFS_TYPE_POOL           = 0x8
  54 } zfs_type_t;
```

*unchanged_portion_omitted_*

```
 477 /*
 478  * The following are configuration names used in the nvlist describing a pool's
 479  * configuration.
 480  */
 481 #define ZPOOL_CONFIG_VERSION            "version"
```



```
 482 #define ZPOOL_CONFIG_POOL_NAME          "name"
 483 #define ZPOOL_CONFIG_POOL_STATE         "state"
 484 #define ZPOOL_CONFIG_POOL_TXG           "txg"
 485 #define ZPOOL_CONFIG_POOL_GUID          "pool_guid"
 486 #define ZPOOL_CONFIG_CREATE_TXG         "create_txg"
 487 #define ZPOOL_CONFIG_TOP_GUID           "top_guid"
 488 #define ZPOOL_CONFIG_VDEV_TREE          "vdev_tree"
 489 #define ZPOOL_CONFIG_TYPE               "type"
 490 #define ZPOOL_CONFIG_CHILDREN           "children"
 491 #define ZPOOL_CONFIG_ID                 "id"
 492 #define ZPOOL_CONFIG_GUID               "guid"
 493 #define ZPOOL_CONFIG_PATH               "path"
 494 #define ZPOOL_CONFIG_DEVID              "devid"
 495 #define ZPOOL_CONFIG_METASLAB_ARRAY     "metaslab_array"
 496 #define ZPOOL_CONFIG_METASLAB_SHIFT     "metaslab_shift"
 497 #define ZPOOL_CONFIG_ASHIFT             "ashift"
 498 #define ZPOOL_CONFIG_ASIZE              "asize"
 499 #define ZPOOL_CONFIG_DTL                "DTL"
 500 #define ZPOOL_CONFIG_SCAN_STATS         "scan_stats"    /* not stored on disk */
 501 #define ZPOOL_CONFIG_VDEV_STATS         "vdev_stats"    /* not stored on disk */
 502 #define ZPOOL_CONFIG_WHOLE_DISK         "whole_disk"
 503 #define ZPOOL_CONFIG_ERRCOUNT           "error_count"
 504 #define ZPOOL_CONFIG_NOT_PRESENT        "not_present"
 505 #define ZPOOL_CONFIG_SPARES             "spares"
 506 #define ZPOOL_CONFIG_IS_SPARE           "is_spare"
 507 #define ZPOOL_CONFIG_NPARITY            "nparity"
 508 #define ZPOOL_CONFIG_HOSTID             "hostid"
 509 #define ZPOOL_CONFIG_HOSTNAME           "hostname"
 510 #define ZPOOL_CONFIG_LOADED_TIME        "initial_load_time"
 511 #define ZPOOL_CONFIG_UNSPARE            "unspare"
 512 #define ZPOOL_CONFIG_PHYS_PATH          "phys_path"
 513 #define ZPOOL_CONFIG_IS_LOG             "is_log"
 514 #define ZPOOL_CONFIG_L2CACHE            "l2cache"
 515 #define ZPOOL_CONFIG_L2CACHE_PERSISTENT "l2cache_persistent"
 516 #define ZPOOL_CONFIG_HOLE_ARRAY         "hole_array"
 517 #define ZPOOL_CONFIG_VDEV_CHILDREN      "vdev_children"
 518 #define ZPOOL_CONFIG_IS_HOLE            "is_hole"
 519 #define ZPOOL_CONFIG_DDT_HISTOGRAM      "ddt_histogram"
 520 #define ZPOOL_CONFIG_DDT_OBJ_STATS      "ddt_object_stats"
 521 #define ZPOOL_CONFIG_DDT_STATS          "ddt_stats"
 522 #define ZPOOL_CONFIG_SPLIT              "splitcfg"
 523 #define ZPOOL_CONFIG_ORIG_GUID          "orig_guid"
 524 #define ZPOOL_CONFIG_SPLIT_GUID         "split_guid"
 525 #define ZPOOL_CONFIG_SPLIT_LIST         "guid_list"
 526 #define ZPOOL_CONFIG_REMOVING           "removing"
 527 #define ZPOOL_CONFIG_RESILVER_TXG       "resilver_txg"
 528 #define ZPOOL_CONFIG_COMMENT            "comment"
 529 #define ZPOOL_CONFIG_SUSPENDED          "suspended"     /* not stored on disk */
 530 #define ZPOOL_CONFIG_TIMESTAMP          "timestamp"     /* not stored on disk */
 531 #define ZPOOL_CONFIG_BOOTFS             "bootfs"        /* not stored on disk */
 532 #define ZPOOL_CONFIG_MISSING_DEVICES    "missing_vdevs" /* not stored on disk */
 533 #define ZPOOL_CONFIG_LOAD_INFO          "load_info"     /* not stored on disk */
 534 #define ZPOOL_CONFIG_REWIND_INFO        "rewind_info"   /* not stored on disk */
 535 #define ZPOOL_CONFIG_UNSUP_FEAT         "unsup_feat"    /* not stored on disk */
 536 #define ZPOOL_CONFIG_ENABLED_FEAT       "enabled_feat"  /* not stored on disk */
 537 #define ZPOOL_CONFIG_CAN_RDONLY         "can_rdonly"    /* not stored on disk */
 538 #define ZPOOL_CONFIG_FEATURES_FOR_READ  "features_for_read"
 539 #define ZPOOL_CONFIG_FEATURE_STATS      "feature_stats" /* not stored on disk */
 540 /*
 541  * The persistent vdev state is stored as separate values rather than a single
 542  * 'vdev_state' entry.  This is because a device can be in multiple states, such
 543  * as offline and degraded.
 544  */
 545 #define ZPOOL_CONFIG_OFFLINE            "offline"
 546 #define ZPOOL_CONFIG_FAULTED            "faulted"
 547 #define ZPOOL_CONFIG_DEGRADED           "degraded"
```

```
 548 #define ZPOOL_CONFIG_REMOVED            "removed"
 549 #define ZPOOL_CONFIG_FRU                "fru"
 550 #define ZPOOL_CONFIG_AUX_STATE          "aux_state"

 552 /* Rewind policy parameters */
 553 #define ZPOOL_REWIND_POLICY             "rewind-policy"
 554 #define ZPOOL_REWIND_REQUEST            "rewind-request"
 555 #define ZPOOL_REWIND_REQUEST_TXG        "rewind-request-txg"
 556 #define ZPOOL_REWIND_META_THRESH        "rewind-meta-thresh"
 557 #define ZPOOL_REWIND_DATA_THRESH        "rewind-data-thresh"

 559 /* Rewind data discovered */
 560 #define ZPOOL_CONFIG_LOAD_TIME          "rewind_txg_ts"
 561 #define ZPOOL_CONFIG_LOAD_DATA_ERRORS   "verify_data_errors"
 562 #define ZPOOL_CONFIG_REWIND_TIME        "seconds_of_rewind"

 564 #define VDEV_TYPE_ROOT                  "root"
 565 #define VDEV_TYPE_MIRROR                "mirror"
 566 #define VDEV_TYPE_REPLACING             "replacing"
 567 #define VDEV_TYPE_RAIDZ                 "raidz"
 568 #define VDEV_TYPE_DISK                  "disk"
 569 #define VDEV_TYPE_FILE                  "file"
 570 #define VDEV_TYPE_MISSING               "missing"
 571 #define VDEV_TYPE_HOLE                  "hole"
 572 #define VDEV_TYPE_SPARE                 "spare"
 573 #define VDEV_TYPE_LOG                   "log"
 574 #define VDEV_TYPE_L2CACHE               "l2cache"

 576 /*
 577  * This is needed in userland to report the minimum necessary device size.
 578  */
 579 #define SPA_MINDEVSIZE          (64ULL << 20)

 581 /*
 582  * The location of the pool configuration repository, shared between kernel and
 583  * userland.
 584  */
 585 #define ZPOOL_CACHE             "/etc/zfs/zpool.cache"

 587 /*
 588  * vdev states are ordered from least to most healthy.
 589  * A vdev that's CANT_OPEN or below is considered unusable.
 590  */
 591 typedef enum vdev_state {
 592         VDEV_STATE_UNKNOWN = 0, /* Uninitialized vdev                   */
 593         VDEV_STATE_CLOSED,      /* Not currently open                   */
 594         VDEV_STATE_OFFLINE,     /* Not allowed to open                  */
 595         VDEV_STATE_REMOVED,     /* Explicitly removed from system       */
 596         VDEV_STATE_CANT_OPEN,   /* Tried to open, but failed            */
 597         VDEV_STATE_FAULTED,     /* External request to fault device     */
 598         VDEV_STATE_DEGRADED,    /* Replicated vdev with unhealthy kids  */
 599         VDEV_STATE_HEALTHY      /* Presumed good                        */
 600 } vdev_state_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2456 Mon Dec  9 16:07:38 2013
new/usr/src/uts/common/sys/list.h
3525 Persistent L2ARC
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2013 Saso Kiselkov, All rights reserved.
 */

#ifndef _SYS_LIST_H
#define _SYS_LIST_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/list_impl.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Please note that a list_node_t contains pointers back to its parent list_t
 * so you cannot copy the list_t around once it has been initialized. In
 * particular, this kind of construct won't work:
 *
 * struct { list_t l; } a, b;
 * list_create(&a.l, ...);
 * b = a;    <= This will break the list in 'b', as the 'l' element in 'a'
 *              got copied to a different memory address.
 *
 * When copying structures with lists use list_move_tail() to move the list
 * from the src to dst (the source reference will then become invalid).
 */
typedef struct list_node list_node_t;
typedef struct list list_t;

void list_create(list_t *, size_t, size_t);
void list_destroy(list_t *);

void list_insert_after(list_t *, void *, void *);
void list_insert_before(list_t *, void *, void *);
void list_insert_head(list_t *, void *);
```



```
void list_insert_tail(list_t *, void *);
void list_remove(list_t *, void *);
void *list_remove_head(list_t *);
void *list_remove_tail(list_t *);
void list_move_tail(list_t *, list_t *);

void *list_head(list_t *);
void *list_tail(list_t *);
void *list_next(list_t *, void *);
void *list_prev(list_t *, void *);
int list_is_empty(list_t *);

void list_link_init(list_node_t *);
void list_link_replace(list_node_t *, list_node_t *);

int list_link_active(list_node_t *);

#ifdef  __cplusplus
}
_____unchanged_portion_omitted_
```