```
**********************************************************
  203650 Mon Aug  5 21:59:08 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 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

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



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

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

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

```
#include <sys/vdev.h>
#include <sys/vdev_impl.h>
#ifdef _KERNEL
#include <sys/vmsystm.h>
#include <vm/anon.h>
#include <sys/fs/swapnode.h>
#include <sys/dnlc.h>
#endif
#include <sys/callb.h>
#include <sys/kstat.h>
#include <zfs_fletcher.h>
#include <sys/byteorder.h>

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

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

extern int zfs_write_limit_shift;
extern uint64_t zfs_write_limit_max;
extern kmutex_t zfs_write_limit_lock;

#define ARC_REDUCE_DNLC_PERCENT 3
uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

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

/* The 6 states: */
static arc_state_t ARC_anon;
static arc_state_t ARC_mru;
static arc_state_t ARC_mru_ghost;
static arc_state_t ARC_mfu;
static arc_state_t ARC_mfu_ghost;
static arc_state_t ARC_l2c_only;

typedef struct arc_stats {
        kstat_named_t arcstat_hits;
        kstat_named_t arcstat_misses;
        kstat_named_t arcstat_demand_data_hits;
        kstat_named_t arcstat_demand_data_misses;
        kstat_named_t arcstat_demand_metadata_hits;
        kstat_named_t arcstat_demand_metadata_misses;
        kstat_named_t arcstat_prefetch_data_hits;
        kstat_named_t arcstat_prefetch_data_misses;
        kstat_named_t arcstat_prefetch_metadata_hits;
        kstat_named_t arcstat_prefetch_metadata_misses;
        kstat_named_t arcstat_mru_hits;
        kstat_named_t arcstat_mru_ghost_hits;
        kstat_named_t arcstat_mfu_hits;
        kstat_named_t arcstat_mfu_ghost_hits;
        kstat_named_t arcstat_deleted;
        kstat_named_t arcstat_recycle_miss;
        /*
         * Number of buffers that could not be evicted because the hash lock
         * was held by another thread.  The lock may not necessarily be held
         * by something using the same buffer, since hash locks are shared
         * by multiple buffers.
```



```
         */
        kstat_named_t arcstat_mutex_miss;
        /*
         * Number of buffers skipped because they have I/O in progress, are
         * indirect prefetch buffers that have not lived long enough, or are
         * not from the spa we're trying to evict from.
         */
        kstat_named_t arcstat_evict_skip;
        kstat_named_t arcstat_evict_l2_cached;
        kstat_named_t arcstat_evict_l2_eligible;
        kstat_named_t arcstat_evict_l2_ineligible;
        kstat_named_t arcstat_hash_elements;
        kstat_named_t arcstat_hash_elements_max;
        kstat_named_t arcstat_hash_collisions;
        kstat_named_t arcstat_hash_chains;
        kstat_named_t arcstat_hash_chain_max;
        kstat_named_t arcstat_p;
        kstat_named_t arcstat_c;
        kstat_named_t arcstat_c_min;
        kstat_named_t arcstat_c_max;
        kstat_named_t arcstat_size;
        kstat_named_t arcstat_hdr_size;
        kstat_named_t arcstat_data_size;
        kstat_named_t arcstat_other_size;
        kstat_named_t arcstat_l2_hits;
        kstat_named_t arcstat_l2_misses;
        kstat_named_t arcstat_l2_feeds;
        kstat_named_t arcstat_l2_rw_clash;
        kstat_named_t arcstat_l2_read_bytes;
        kstat_named_t arcstat_l2_write_bytes;
        kstat_named_t arcstat_l2_writes_sent;
        kstat_named_t arcstat_l2_writes_done;
        kstat_named_t arcstat_l2_writes_error;
        kstat_named_t arcstat_l2_writes_hdr_miss;
        kstat_named_t arcstat_l2_evict_lock_retry;
        kstat_named_t arcstat_l2_evict_reading;
        kstat_named_t arcstat_l2_free_on_write;
        kstat_named_t arcstat_l2_abort_lowmem;
        kstat_named_t arcstat_l2_cksum_bad;
        kstat_named_t arcstat_l2_io_error;
        kstat_named_t arcstat_l2_size;
        kstat_named_t arcstat_l2_asize;
        kstat_named_t arcstat_l2_hdr_size;
        kstat_named_t arcstat_l2_compress_successes;
        kstat_named_t arcstat_l2_compress_zeros;
        kstat_named_t arcstat_l2_compress_failures;
        kstat_named_t arcstat_l2_meta_writes;
        kstat_named_t arcstat_l2_meta_avg_size;
        kstat_named_t arcstat_l2_meta_avg_asize;
        kstat_named_t arcstat_l2_asize_to_meta_ratio;
        kstat_named_t arcstat_l2_rebuild_attempts;
        kstat_named_t arcstat_l2_rebuild_successes;
        kstat_named_t arcstat_l2_rebuild_unsupported;
        kstat_named_t arcstat_l2_rebuild_timeout;
        kstat_named_t arcstat_l2_rebuild_arc_bytes;
        kstat_named_t arcstat_l2_rebuild_l2arc_bytes;
        kstat_named_t arcstat_l2_rebuild_bufs;
        kstat_named_t arcstat_l2_rebuild_bufs_precached;
        kstat_named_t arcstat_l2_rebuild_metabufs;
        kstat_named_t arcstat_l2_rebuild_uberblk_errors;
        kstat_named_t arcstat_l2_rebuild_io_errors;
        kstat_named_t arcstat_l2_rebuild_cksum_errors;
        kstat_named_t arcstat_l2_rebuild_loop_errors;
        kstat_named_t arcstat_l2_rebuild_abort_lowmem;
        kstat_named_t arcstat_memory_throttle_count;
        kstat_named_t arcstat_duplicate_buffers;
```

```
331         kstat_named_t arcstat_duplicate_buffers_size;
332         kstat_named_t arcstat_duplicate_reads;
333         kstat_named_t arcstat_meta_used;
334         kstat_named_t arcstat_meta_limit;
335         kstat_named_t arcstat_meta_max;
336 } arc_stats_t;

338 static arc_stats_t arc_stats = {
339         { "hits",                       KSTAT_DATA_UINT64 },
340         { "misses",                     KSTAT_DATA_UINT64 },
341         { "demand_data_hits",           KSTAT_DATA_UINT64 },
342         { "demand_data_misses",         KSTAT_DATA_UINT64 },
343         { "demand_metadata_hits",       KSTAT_DATA_UINT64 },
344         { "demand_metadata_misses",     KSTAT_DATA_UINT64 },
345         { "prefetch_data_hits",         KSTAT_DATA_UINT64 },
346         { "prefetch_data_misses",       KSTAT_DATA_UINT64 },
347         { "prefetch_metadata_hits",     KSTAT_DATA_UINT64 },
348         { "prefetch_metadata_misses",   KSTAT_DATA_UINT64 },
349         { "mru_hits",                   KSTAT_DATA_UINT64 },
350         { "mru_ghost_hits",             KSTAT_DATA_UINT64 },
351         { "mfu_hits",                   KSTAT_DATA_UINT64 },
352         { "mfu_ghost_hits",             KSTAT_DATA_UINT64 },
353         { "deleted",                    KSTAT_DATA_UINT64 },
354         { "recycle_miss",               KSTAT_DATA_UINT64 },
355         { "mutex_miss",                 KSTAT_DATA_UINT64 },
356         { "evict_skip",                 KSTAT_DATA_UINT64 },
357         { "evict_l2_cached",            KSTAT_DATA_UINT64 },
358         { "evict_l2_eligible",          KSTAT_DATA_UINT64 },
359         { "evict_l2_ineligible",        KSTAT_DATA_UINT64 },
360         { "hash_elements",              KSTAT_DATA_UINT64 },
361         { "hash_elements_max",          KSTAT_DATA_UINT64 },
362         { "hash_collisions",            KSTAT_DATA_UINT64 },
363         { "hash_chains",                KSTAT_DATA_UINT64 },
364         { "hash_chain_max",             KSTAT_DATA_UINT64 },
365         { "p",                          KSTAT_DATA_UINT64 },
366         { "c",                          KSTAT_DATA_UINT64 },
367         { "c_min",                      KSTAT_DATA_UINT64 },
368         { "c_max",                      KSTAT_DATA_UINT64 },
369         { "size",                       KSTAT_DATA_UINT64 },
370         { "hdr_size",                   KSTAT_DATA_UINT64 },
371         { "data_size",                  KSTAT_DATA_UINT64 },
372         { "other_size",                 KSTAT_DATA_UINT64 },
373         { "l2_hits",                    KSTAT_DATA_UINT64 },
374         { "l2_misses",                  KSTAT_DATA_UINT64 },
375         { "l2_feeds",                   KSTAT_DATA_UINT64 },
376         { "l2_rw_clash",                KSTAT_DATA_UINT64 },
377         { "l2_read_bytes",              KSTAT_DATA_UINT64 },
378         { "l2_write_bytes",             KSTAT_DATA_UINT64 },
379         { "l2_writes_sent",             KSTAT_DATA_UINT64 },
380         { "l2_writes_done",             KSTAT_DATA_UINT64 },
381         { "l2_writes_error",            KSTAT_DATA_UINT64 },
382         { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
383         { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
384         { "l2_evict_reading",           KSTAT_DATA_UINT64 },
385         { "l2_free_on_write",           KSTAT_DATA_UINT64 },
386         { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
387         { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
388         { "l2_io_error",                KSTAT_DATA_UINT64 },
389         { "l2_size",                    KSTAT_DATA_UINT64 },
390         { "l2_asize",                   KSTAT_DATA_UINT64 },
391         { "l2_hdr_size",                KSTAT_DATA_UINT64 },
392         { "l2_compress_successes",      KSTAT_DATA_UINT64 },
393         { "l2_compress_zeros",          KSTAT_DATA_UINT64 },
394         { "l2_compress_failures",       KSTAT_DATA_UINT64 },
395         { "l2_meta_writes",             KSTAT_DATA_UINT64 },
396         { "l2_meta_avg_size",           KSTAT_DATA_UINT64 },
```



```
397         { "l2_meta_avg_asize",          KSTAT_DATA_UINT64 },
398         { "l2_asize_to_meta_ratio",     KSTAT_DATA_UINT64 },
399         { "l2_rebuild_attempts",        KSTAT_DATA_UINT64 },
400         { "l2_rebuild_successes",       KSTAT_DATA_UINT64 },
401         { "l2_rebuild_unsupported",     KSTAT_DATA_UINT64 },
402         { "l2_rebuild_timeout",         KSTAT_DATA_UINT64 },
403         { "l2_rebuild_arc_bytes",       KSTAT_DATA_UINT64 },
404         { "l2_rebuild_l2arc_bytes",     KSTAT_DATA_UINT64 },
405         { "l2_rebuild_bufs",            KSTAT_DATA_UINT64 },
406         { "l2_rebuild_precached",       KSTAT_DATA_UINT64 },
407         { "l2_rebuild_metabufs",        KSTAT_DATA_UINT64 },
408         { "l2_rebuild_uberblk_errors",  KSTAT_DATA_UINT64 },
409         { "l2_rebuild_io_errors",       KSTAT_DATA_UINT64 },
410         { "l2_rebuild_cksum_errors",    KSTAT_DATA_UINT64 },
411         { "l2_rebuild_loop_errors",     KSTAT_DATA_UINT64 },
412         { "l2_rebuild_abort_lowmem",    KSTAT_DATA_UINT64 },
413         { "memory_throttle_count",      KSTAT_DATA_UINT64 },
414         { "duplicate_buffers",          KSTAT_DATA_UINT64 },
415         { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
416         { "duplicate_reads",            KSTAT_DATA_UINT64 },
417         { "arc_meta_used",              KSTAT_DATA_UINT64 },
418         { "arc_meta_limit",             KSTAT_DATA_UINT64 },
419         { "arc_meta_max",               KSTAT_DATA_UINT64 }
420 };
```

*unchanged_portion_omitted_*

```
437 #define ARCSTAT_MAXSTAT(stat) \
438         ARCSTAT_MAX(stat##_max, arc_stats.stat.value.ui64)

440 /*
441  * We define a macro to allow ARC hits/misses to be easily broken down by
442  * two separate conditions, giving a total of four different subtypes for
443  * each of hits and misses (so eight statistics total).
444  */
445 #define ARCSTAT_CONDSTAT(cond1, stat1, notstat1, cond2, stat2, notstat2, stat) \
446         if (cond1) {                                                    \
447                 if (cond2) {                                            \
448                         ARCSTAT_BUMP(arcstat_##stat1##_##stat2##_##stat); \
449                 } else {                                                \
450                         ARCSTAT_BUMP(arcstat_##stat1##_##notstat2##_##stat); \
451                 }                                                       \
452         } else {                                                        \
453                 if (cond2) {                                            \
454                         ARCSTAT_BUMP(arcstat_##notstat1##_##stat2##_##stat); \
455                 } else {                                                \
456                         ARCSTAT_BUMP(arcstat_##notstat1##_##notstat2##_##stat);\
457                 }                                                       \
458         }

460 /*
461  * This macro allows us to use kstats as floating averages. Each time we
462  * update this kstat, we first factor it and the update value by
463  * ARCSTAT_AVG_FACTOR to shrink the new value's contribution to the overall
464  * average. This macro assumes that integer loads and stores are atomic, but
465  * is not safe for multiple writers updating the kstat in parallel (only the
466  * last writer's update will remain).
467  */
468 #define ARCSTAT_F_AVG_FACTOR    3
469 #define ARCSTAT_F_AVG(stat, value) \
470         do { \
471                 uint64_t x = ARCSTAT(stat); \
472                 x = x - x / ARCSTAT_F_AVG_FACTOR + \
473                     (value) / ARCSTAT_F_AVG_FACTOR; \
474                 ARCSTAT(stat) = x; \
475                 _NOTE(NOTREACHED) \
476                 _NOTE(CONSTCOND) \
```

```
477         } while (0)

479 kstat_t                 *arc_ksp;
480 static arc_state_t      *arc_anon;
481 static arc_state_t      *arc_mru;
482 static arc_state_t      *arc_mru_ghost;
483 static arc_state_t      *arc_mfu;
484 static arc_state_t      *arc_mfu_ghost;
485 static arc_state_t      *arc_l2c_only;

487 /*
488  * There are several ARC variables that are critical to export as kstats --
489  * but we don't want to have to grovel around in the kstat whenever we wish to
490  * manipulate them.  For these variables, we therefore define them to be in
491  * terms of the statistic variable.  This assures that we are not introducing
492  * the possibility of inconsistency by having shadow copies of the variables,
493  * while still allowing the code to be readable.
494  */
495 #define arc_size        ARCSTAT(arcstat_size)   /* actual total arc size */
496 #define arc_p           ARCSTAT(arcstat_p)      /* target size of MRU */
497 #define arc_c           ARCSTAT(arcstat_c)      /* target size of cache */
498 #define arc_c_min       ARCSTAT(arcstat_c_min)  /* min target cache size */
499 #define arc_c_max       ARCSTAT(arcstat_c_max)  /* max target cache size */
500 #define arc_meta_limit  ARCSTAT(arcstat_meta_limit) /* max size for metadata */
501 #define arc_meta_used   ARCSTAT(arcstat_meta_used) /* size of metadata */
502 #define arc_meta_max    ARCSTAT(arcstat_meta_max) /* max size of metadata */

504 #define L2ARC_IS_VALID_COMPRESS(_c_) \
505         ((_c_) == ZIO_COMPRESS_LZ4 || (_c_) == ZIO_COMPRESS_EMPTY)

507 static int              arc_no_grow;    /* Don't try to grow cache size */
508 static uint64_t         arc_tempreserve;
509 static uint64_t         arc_loaned_bytes;

511 typedef struct l2arc_buf_hdr l2arc_buf_hdr_t;

513 typedef struct arc_callback arc_callback_t;

515 struct arc_callback {
516         void                    *acb_private;
517         arc_done_func_t         *acb_done;
518         arc_buf_t               *acb_buf;
519         zio_t                   *acb_zio_dummy;
520         arc_callback_t          *acb_next;
521 };
```

*unchanged_portion_omitted_*

```
644 static buf_hash_table_t buf_hash_table;

646 #define BUF_HASH_INDEX(spa, dva, birth) \
647         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
648 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
649 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
650 #define HDR_LOCK(hdr) \
651         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))

653 uint64_t zfs_crc64_table[256];

655 /*
656  * Level 2 ARC
657  */

659 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
660 #define L2ARC_HEADROOM          2                       /* num of writes */
661 /*
662  * If we discover during ARC scan any buffers to be compressed, we boost
```



```
663  * our headroom for the next scanning cycle by this percentage multiple.
664  */
665 #define L2ARC_HEADROOM_BOOST    200
666 #define L2ARC_FEED_SECS         1               /* caching interval secs */
667 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

669 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
670 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

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

683 /*
684  * L2ARC Internals
685  */
686 typedef struct l2arc_dev l2arc_dev_t;
630 typedef struct l2arc_dev {
631         vdev_t                  *l2ad_vdev;     /* vdev */
632         spa_t                   *l2ad_spa;      /* spa */
633         uint64_t                l2ad_hand;      /* next write location */
634         uint64_t                l2ad_start;     /* first addr on device */
635         uint64_t                l2ad_end;       /* last addr on device */
636         uint64_t                l2ad_evict;     /* last addr eviction reached */
637         boolean_t               l2ad_first;     /* first sweep through */
638         boolean_t               l2ad_writing;   /* currently writing */
639         list_t                  *l2ad_buflist;  /* buffer list */
640         list_node_t             l2ad_node;      /* device list node */
641 } l2arc_dev_t;

687 static list_t L2ARC_dev_list;                   /* device list */
688 static list_t *l2arc_dev_list;                  /* device list pointer */
689 static kmutex_t l2arc_dev_mtx;                  /* device list mutex */
690 static l2arc_dev_t *l2arc_dev_last;             /* last device used */
691 static kmutex_t l2arc_buflist_mtx;              /* mutex for all buflists */
692 static list_t L2ARC_free_on_write;              /* free after write buf list */
693 static list_t *l2arc_free_on_write;             /* free after write list ptr */
694 static kmutex_t l2arc_free_on_write_mtx;        /* mutex for list */
695 static uint64_t l2arc_ndev;                     /* number of devices */

697 typedef struct l2arc_read_callback {
698         arc_buf_t               *l2rcb_buf;             /* read buffer */
699         spa_t                   *l2rcb_spa;             /* spa */
700         blkptr_t                l2rcb_bp;               /* original blkptr */
701         zbookmark_t             l2rcb_zb;               /* original bookmark */
702         int                     l2rcb_flags;            /* original flags */
703         enum zio_compress       l2rcb_compress;         /* applied compress */
704 } l2arc_read_callback_t;

706 typedef struct l2arc_write_callback {
707         l2arc_dev_t     *l2wcb_dev;             /* device info */
708         arc_buf_hdr_t   *l2wcb_head;            /* head of write buflist */
709         uint8_t         *l2wcb_pbuf;            /* pbuf sent in this write */
710         uint32_t        l2wcb_pbuf_size;        /* size of committed pbuf */
711         uint8_t         *l2wcb_ub_buf;          /* uberblock in this write */
712 } l2arc_write_callback_t;
```

*unchanged_portion_omitted_*

```
734 static kmutex_t l2arc_feed_thr_lock;
```

```
735 static kcondvar_t l2arc_feed_thr_cv;
736 static uint8_t l2arc_thread_exit;

738 static void l2arc_read_done(zio_t *zio);
739 static void l2arc_hdr_stat_add(boolean_t from_arc);
692 static void l2arc_hdr_stat_add(void);
740 static void l2arc_hdr_stat_remove(void);

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

747 typedef enum {
748         L2UBLK_BIG_ENDIAN = (1 << 0),   /* little endian assumed otherwise */
749         L2UBLK_EVICT_FIRST = (1 << 1)   /* mirror of l2ad_first in l2dev */
750 } l2uberblock_flags_t;

752 typedef struct l2uberblock {
753         uint32_t                ub_magic;
754         uint8_t                 ub_version;
755         l2uberblock_flags_t     ub_flags;

757         uint64_t                ub_spa_guid;
758         uint64_t                ub_birth;
759         uint64_t                ub_evict_tail;  /* current evict pointer */
760         uint64_t                ub_alloc_space; /* vdev space alloc status */
761         uint64_t                ub_pbuf_daddr;  /* address of newest pbuf */
762         uint32_t                ub_pbuf_asize;  /* size of newest pbuf */
763         zio_cksum_t             ub_pbuf_cksum;  /* fletcher4 of newest pbuf */

765         zio_cksum_t             ub_cksum;       /* cksum of uberblock */
766 } l2uberblock_t;

768 typedef enum {
769         L2PBUF_BIG_ENDIAN = (1 << 0),   /* little endian assumed otherwise */
770         L2PBUF_COMPRESSED = (1 << 1)    /* pbuf data items are compressed */
771 } l2pbuf_flags_t;

773 typedef struct l2pbuf {
774         uint32_t                pb_magic;
775         unsigned int            pb_version;
776         l2pbuf_flags_t          pb_flags;

778         uint64_t                pb_prev_daddr;  /* address of previous pbuf */
779         uint32_t                pb_prev_asize;  /* size of previous pbuf */
780         zio_cksum_t             pb_prev_cksum;  /* fletcher4 of prev. pbuf */

782         /*
783          * This is a set of item lists that are contained in this pbuf. Each
784          * L2ARC write appends a new l2pbuf_buflist_t array of l2pbuf_buf_t's.
785          * This serves as a soft timeout feature - once the limit of the
786          * number of item lists that a pbuf can hold is reached, the pbuf is
787          * flushed to stable storage, regardless of its total size.
788          */
789         list_t                  *pb_buflists_list;

791         /*
792          * Number of compressed bytes referenced by items in this pbuf and
793          * the number of lists present.
794          * This is not actually written to storage, it is only used by
795          * internal algorithms which check for when a pbuf reaches a
796          * certain size limit, after which it is flushed in a write.
797          */
798         uint64_t                pb_payload_asz;
799         /* Same thing for number of buflists */
```



```
800         int                     pb_nbuflists;

802         /*
803          * Filled in by l2arc_pbuf_read to hold this pbuf's alloc'd size.
804          * This is then used by l2arc_pbuf_restore to update used space
805          * on the L2ARC vdev.
806          */
807         size_t                  pb_asize;
808 } l2pbuf_t;

810 typedef struct l2pbuf_buf l2pbuf_buf_t;
811 typedef struct l2pbuf_buflist {
812         uint32_t                l2pbl_nbufs;
813         l2pbuf_buf_t            *l2pbl_bufs;
814         list_node_t             l2pbl_node;
815 } l2pbuf_buflist_t;

817 struct l2pbuf_buf {
818         dva_t                   b_dva;          /* dva of buffer */
819         uint64_t                b_birth;        /* birth txg of buffer */
820         uint64_t                b_cksum0;
821         zio_cksum_t             b_freeze_cksum;
822         uint32_t                b_size;         /* uncompressed buf size */
823         uint64_t                b_l2daddr;      /* buf location on l2dev */
824         uint32_t                b_l2asize;      /* actual buf data size */
825         enum zio_compress       b_l2compress;   /* compression applied */
826         uint16_t                b_contents_type;
827         uint32_t                b_flags;
828 };

830 struct l2arc_dev {
831         vdev_t                  *l2ad_vdev;     /* vdev */
832         spa_t                   *l2ad_spa;      /* spa */
833         uint64_t                l2ad_hand;      /* next write location */
834         uint64_t                l2ad_start;     /* first addr on device */
835         uint64_t                l2ad_end;       /* last addr on device */
836         uint64_t                l2ad_evict;     /* last addr eviction reached */
837         boolean_t               l2ad_first;     /* first sweep through */
838         boolean_t               l2ad_writing;   /* currently writing */
839         list_t                  *l2ad_buflist;  /* buffer list */
840         list_node_t             l2ad_node;      /* device list node */
841         l2pbuf_t                l2ad_pbuf;      /* currently open pbuf */
842         uint64_t                l2ad_pbuf_daddr;        /* prev pbuf daddr */
843         uint64_t                l2ad_pbuf_asize;        /* prev pbuf asize */
844         zio_cksum_t             l2ad_pbuf_cksum;        /* prev pbuf cksum */
845         /* uberblock birth counter - incremented for each committed uberblk */
846         uint64_t                l2ad_uberblock_birth;
847         /* flag indicating whether a rebuild is currently going on */
848         boolean_t               l2ad_rebuilding;
849 };

851 /* Stores information about an L2ARC prefetch zio */
852 typedef struct l2arc_prefetch_info {
853         uint8_t                 *pi_buf;        /* where the zio writes to */
854         uint64_t                pi_buflen;      /* length of 'buf' */
855         zio_t                   *pi_hdr_io;     /* see l2arc_pbuf_read below */
856 } l2arc_prefetch_info_t;

858 /* 256 x 4k of l2uberblocks */
859 #define L2UBERBLOCK_SIZE        4096
860 #define L2UBERBLOCK_MAGIC       0x12bab10c
861 #define L2UBERBLOCK_MAX_VERSION 1       /* our maximum uberblock version */
862 #define L2PBUF_MAGIC            0xdb0faba6
863 #define L2PBUF_MAX_VERSION      1       /* our maximum pbuf version */
864 #define L2PBUF_BUF_SIZE         88      /* size of one pbuf buf entry */
865 #define L2PBUF_HDR_SIZE         56      /* pbuf header excluding any payload */
```

```
#define L2PBUF_ENCODED_SIZE(_pb) \
        (L2PBUF_HDR_SIZE + l2arc_pbuf_items_encoded_size(_pb))
/*
 * Allocation limit for the payload of a pbuf. This also fundamentally
 * limits the number of bufs we can reference in a pbuf.
 */
#define L2PBUF_MAX_PAYLOAD_SIZE (24 * 1024 * 1024)
#define L2PBUF_MAX_BUFS         (L2PBUF_MAX_PAYLOAD_SIZE / L2PBUF_BUF_SIZE)
#define L2PBUF_COMPRESS_MINSZ   8192    /* minimum size to compress a pbuf */
#define L2PBUF_MAXSZ            100 * 1024 * 1024       /* maximum pbuf size */
#define L2PBUF_MAX_BUFLISTS     128     /* max number of buflists per pbuf */
#define L2ARC_REBUILD_TIMEOUT   60      /* a rebuild may take at most 60s */
#define L2PBUF_IS_FULL(_pb) \
        ((_pb)->pb_payload_asz > l2arc_pbuf_max_sz || \
        (_pb)->pb_nbuflists + 1 >= l2arc_pbuf_max_buflists)
/*
 * These are the flags we allow to persist in L2ARC pbufs. The other flags
 * of an ARC buffer pertain to the buffer's runtime behavior.
 */
#define L2ARC_PERSIST_FLAGS \
        (ARC_IN_HASH_TABLE | ARC_L2CACHE | ARC_L2COMPRESS | ARC_PREFETCH)

/*
 * Used during L2ARC rebuild after each read operation to check whether we
 * haven't exceeded the rebuild timeout value.
 */
#define L2ARC_CHK_REBUILD_TIMEOUT(_deadline_, ...) \
        do { \
                if ((_deadline_) != 0 && (_deadline_) < ddi_get_lbolt64()) { \
                        __VA_ARGS__; \
                        ARCSTAT_BUMP(arcstat_l2_rebuild_timeout); \
                        cmn_err(CE_WARN, "L2ARC rebuild is taking too long, " \
                            "dropping remaining L2ARC metadata."); \
                        return; \
                } \
                _NOTE(NOTREACHED) \
                _NOTE(CONSTCOND) \
        } while (0)

/*
 * Performance tuning of L2ARC persistency:
 *
 * l2arc_pbuf_compress_minsz : Minimum size of a pbuf in order to attempt
 *              compressing it.
 * l2arc_pbuf_max_sz : Upper bound on the physical size of L2ARC buffers
 *              referenced from a pbuf. Once a pbuf reaches this size, it is
 *              committed to stable storage. Ideally, there should be approx.
 *              l2arc_dev_size / l2arc_pbuf_max_sz pbufs on an L2ARC device.
 * l2arc_pbuf_max_buflists : Maximum number of L2ARC feed cycles that will
 *              be buffered in a pbuf before it is committed to L2ARC. This
 *              puts a soft temporal upper bound on pbuf commit intervals.
 * l2arc_rebuild_enabled : Controls whether L2ARC device adds (either at
 *              pool import or when adding one manually later) will attempt
 *              to rebuild L2ARC buffer contents. In special circumstances,
 *              the administrator may want to set this to B_FALSE, if they
 *              are having trouble importing a pool or attaching an L2ARC
 *              device (e.g. the L2ARC device is slow to read in stored pbuf
 *              metadata, or the metadata has become somehow
 *              fragmented/unusable).
 * l2arc_rebuild_timeout : A hard timeout value on L2ARC rebuilding to help
 *              avoid a slow L2ARC device from preventing pool import. If we
 *              are not done rebuilding an L2ARC device by this time, we
 *              stop the rebuild and return immediately.
 */
uint64_t l2arc_pbuf_compress_minsz = L2PBUF_COMPRESS_MINSZ;
uint64_t l2arc_pbuf_max_sz = L2PBUF_MAXSZ;
```



```
uint64_t l2arc_pbuf_max_buflists = L2PBUF_MAX_BUFLISTS;
boolean_t l2arc_rebuild_enabled = B_TRUE;
uint64_t l2arc_rebuild_timeout = L2ARC_REBUILD_TIMEOUT;

static void l2arc_rebuild_start(l2arc_dev_t *dev);
static void l2arc_rebuild(l2arc_dev_t *dev);
static void l2arc_pbuf_restore(l2arc_dev_t *dev, l2pbuf_t *pb);
static void l2arc_hdr_restore(const l2pbuf_buf_t *buf, l2arc_dev_t *dev,
    uint64_t guid);

static int l2arc_uberblock_find(l2arc_dev_t *dev, l2uberblock_t *ub);
static int l2arc_pbuf_read(l2arc_dev_t *dev, uint64_t daddr, uint32_t asize,
    zio_cksum_t cksum, l2pbuf_t *pb, zio_t *this_io, zio_t **next_io);
static int l2arc_pbuf_ptr_valid(l2arc_dev_t *dev, uint64_t daddr,
    uint32_t asize);
static zio_t *l2arc_pbuf_prefetch(vdev_t *vd, uint64_t daddr, uint32_t asize);
static void l2arc_pbuf_prefetch_abort(zio_t *zio);

static void l2arc_uberblock_encode(const l2uberblock_t *ub, uint8_t *buf);
static void l2arc_uberblock_decode(const uint8_t *buf, l2uberblock_t *ub);
static int l2arc_uberblock_verify(const uint8_t *buf, const l2uberblock_t *ub,
    uint64_t guid);
static void l2arc_uberblock_update(l2arc_dev_t *dev, zio_t *pio,
    l2arc_write_callback_t *cb);

static uint32_t l2arc_pbuf_encode(l2pbuf_t *pb, uint8_t *buf, uint32_t buflen);
static int l2arc_pbuf_decode(uint8_t *buf, uint32_t buflen,
    l2pbuf_t *pbuf);
static int l2arc_pbuf_decode_prev_ptr(const uint8_t *buf, size_t buflen,
    uint64_t *daddr, uint32_t *asize, zio_cksum_t *cksum);
static void l2arc_pbuf_init(l2pbuf_t *pb);
static void l2arc_pbuf_destroy(l2pbuf_t *pb);
static void l2arc_pbuf_commit(l2arc_dev_t *dev, zio_t *pio,
    l2arc_write_callback_t *cb);
static l2pbuf_buflist_t *l2arc_pbuf_buflist_alloc(l2pbuf_t *pb, int nbufs);
static void l2arc_pbuflist_insert(l2pbuf_t *pb, l2pbuf_buflist_t *pbl,
    const arc_buf_hdr_t *ab, int index);
static uint32_t l2arc_pbuf_items_encoded_size(l2pbuf_t *pb);

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);
```

```
1434         uint64_t from_delta, to_delta;

1436         ASSERT(MUTEX_HELD(hash_lock));
1437         ASSERT(new_state != old_state);
1438         ASSERT(refcnt == 0 || ab->b_datacnt > 0);
1439         ASSERT(ab->b_datacnt == 0 || !GHOST_STATE(new_state));
1440         ASSERT(ab->b_datacnt <= 1 || old_state != arc_anon);

1442         from_delta = to_delta = ab->b_datacnt * ab->b_size;

1444         /*
1445          * If this buffer is evictable, transfer it from the
1446          * old state list to the new state list.
1447          */
1448         if (refcnt == 0) {
1449                 if (old_state != arc_anon) {
1450                         int use_mutex = !MUTEX_HELD(&old_state->arcs_mtx);
1451                         uint64_t *size = &old_state->arcs_lsize[ab->b_type];

1453                         if (use_mutex)
1454                                 mutex_enter(&old_state->arcs_mtx);

1456                         ASSERT(list_link_active(&ab->b_arc_node));
1457                         list_remove(&old_state->arcs_list[ab->b_type], ab);

1459                         /*
1460                          * If prefetching out of the ghost cache,
1461                          * we will have a non-zero datacnt.
1462                          */
1463                         if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
1464                                 /* ghost elements have a ghost size */
1465                                 ASSERT(ab->b_buf == NULL);
1466                                 from_delta = ab->b_size;
1467                         }
1468                         ASSERT3U(*size, >=, from_delta);
1469                         atomic_add_64(size, -from_delta);

1471                         if (use_mutex)
1472                                 mutex_exit(&old_state->arcs_mtx);
1473                 }
1474                 if (new_state != arc_anon) {
1475                         int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
1476                         uint64_t *size = &new_state->arcs_lsize[ab->b_type];

1478                         if (use_mutex)
1479                                 mutex_enter(&new_state->arcs_mtx);

1481                         list_insert_head(&new_state->arcs_list[ab->b_type], ab);

1483                         /* ghost elements have a ghost size */
1484                         if (GHOST_STATE(new_state)) {
1485                                 ASSERT(ab->b_datacnt == 0);
1486                                 ASSERT(ab->b_buf == NULL);
1487                                 to_delta = ab->b_size;
1488                         }
1489                         atomic_add_64(size, to_delta);

1491                         if (use_mutex)
1492                                 mutex_exit(&new_state->arcs_mtx);
1493                 }
1494         }

1496         ASSERT(!BUF_EMPTY(ab));
1497         if (new_state == arc_anon && HDR_IN_HASH_TABLE(ab))
1498                 buf_hash_remove(ab);
```



```
1500         /* adjust state sizes */
1501         if (to_delta)
1502                 atomic_add_64(&new_state->arcs_size, to_delta);
1503         if (from_delta) {
1504                 ASSERT3U(old_state->arcs_size, >=, from_delta);
1505                 atomic_add_64(&old_state->arcs_size, -from_delta);
1506         }
1507         ab->b_state = new_state;

1509         /* adjust l2arc hdr stats */
1510         if (new_state == arc_l2c_only)
1511                 l2arc_hdr_stat_add(old_state != arc_anon);
1240                 l2arc_hdr_stat_add();
1512         else if (old_state == arc_l2c_only)
1513                 l2arc_hdr_stat_remove();
1514 }
```

*unchanged_portion_omitted_*

```
1615 /*
1616  * Allocates an empty arc_buf_hdr structure (lacking any data buffer).
1617  * This is used during l2arc reconstruction to make empty ARC buffers
1618  * which circumvent the regular disk->arc->l2arc path and instead come
1619  * into being in the reverse order, i.e. l2arc->arc->(disk).
1620  */
1621 arc_buf_hdr_t *
1622 arc_buf_hdr_alloc(uint64_t guid, int size, arc_buf_contents_t type)
1623 {
1624         arc_buf_hdr_t *hdr;

1626         ASSERT3U(size, >, 0);
1627         hdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
1628         ASSERT(BUF_EMPTY(hdr));
1629         hdr->b_size = size;
1630         hdr->b_type = type;
1631         hdr->b_spa = guid;
1632         hdr->b_state = arc_anon;
1633         hdr->b_arc_access = 0;
1634         hdr->b_buf = NULL;
1635         hdr->b_datacnt = 0;
1636         hdr->b_flags = 0;
1637         ASSERT(refcount_is_zero(&hdr->b_refcnt));

1639         return (hdr);
1640 }

1642 static char *arc_onloan_tag = "onloan";

1644 /*
1645  * Loan out an anonymous arc buffer. Loaned buffers are not counted as in
1646  * flight data by arc_tempreserve_space() until they are "returned". Loaned
1647  * buffers must be returned to the arc before they can be used by the DMU or
1648  * freed.
1649  */
1650 arc_buf_t *
1651 arc_loan_buf(spa_t *spa, int size)
1652 {
1653         arc_buf_t *buf;

1655         buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);

1657         atomic_add_64(&arc_loaned_bytes, size);
1658         return (buf);
1659 }
```

*unchanged_portion_omitted_*

```
4131 /*
```

```
4132  * Level 2 ARC
4133  *
4134  * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
4135  * It uses dedicated storage devices to hold cached data, which are populated
4136  * using large infrequent writes.  The main role of this cache is to boost
4137  * the performance of random read workloads.  The intended L2ARC devices
4138  * include short-stroked disks, solid state disks, and other media with
4139  * substantially faster read latency than disk.
4140  *
4141  *                 +-----------------------+
4142  *                 |         ARC           |
4143  *                 +-----------------------+
4144  *                    |         ^     ^
4145  *                    |         |     |
4146  *      l2arc_feed_thread()    arc_read()
4147  *                    |         |     |
4148  *                    |  l2arc read   |
4149  *                    V         |     |
4150  *               +---------------+    |
4151  *               |     L2ARC     |    |
4152  *               +---------------+    |
4153  *                   |    ^           |
4154  *          l2arc_write() |           |
4155  *                   |    |           |
4156  *                   V    |           |
4157  *                 +-------+      +-------+
4158  *                 | vdev  |      | vdev  |
4159  *                 | cache |      | cache |
4160  *                 +-------+      +-------+
4161  *                 +=========+     .-----.
4162  *                 :  L2ARC  :    |-_____-|
4163  *                 : devices :    | Disks |
4164  *                 +=========+    `-_____-'
4165  *
4166  * Read requests are satisfied from the following sources, in order:
4167  *
4168  *      1) ARC
4169  *      2) vdev cache of L2ARC devices
4170  *      3) L2ARC devices
4171  *      4) vdev cache of disks
4172  *      5) disks
4173  *
4174  * Some L2ARC device types exhibit extremely slow write performance.
4175  * To accommodate for this there are some significant differences between
4176  * the L2ARC and traditional cache design:
4177  *
4178  * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
4179  * the ARC behave as usual, freeing buffers and placing headers on ghost
4180  * lists.  The ARC does not send buffers to the L2ARC during eviction as
4181  * this would add inflated write latencies for all ARC memory pressure.
4182  *
4183  * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
4184  * It does this by periodically scanning buffers from the eviction-end of
4185  * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
4186  * not already there. It scans until a headroom of buffers is satisfied,
4187  * which itself is a buffer for ARC eviction. If a compressible buffer is
4188  * found during scanning and selected for writing to an L2ARC device, we
4189  * temporarily boost scanning headroom during the next scan cycle to make
4190  * sure we adapt to compression effects (which might significantly reduce
4191  * the data volume we write to L2ARC). The thread that does this is
4192  * l2arc_feed_thread(), illustrated below; example sizes are included to
4193  * provide a better sense of ratio than this diagram:
4194  *
4195  *             head -->                        tail
4196  *              +---------------------+----------+
4197  *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
```



```
4198  *              +---------------------+----------+   |   o L2ARC eligible
4199  *      ARC_mru |:#::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
4200  *              +---------------------+----------+   |
4201  *                   15.9 Gbytes      ^ 32 Mbytes    |
4202  *                                 headroom          |
4203  *                                           l2arc_feed_thread()
4204  *                                                   |
4205  *                        l2arc write hand <--[oooo]--'
4206  *                                |           8 Mbyte
4207  *                                |          write max
4208  *                                V
4209  *                 +==============================+
4210  *      L2ARC dev |####|#|###|###|    |####| ... |
4211  *                 +==============================+
4212  *                                32 Gbytes
4213  *
4214  * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
4215  * evicted, then the L2ARC has cached a buffer much sooner than it probably
4216  * needed to, potentially wasting L2ARC device bandwidth and storage.  It is
4217  * safe to say that this is an uncommon case, since buffers at the end of
4218  * the ARC lists have moved there due to inactivity.
4219  *
4220  * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
4221  * then the L2ARC simply misses copying some buffers.  This serves as a
4222  * pressure valve to prevent heavy read workloads from both stalling the ARC
4223  * with waits and clogging the L2ARC with writes.  This also helps prevent
4224  * the potential for the L2ARC to churn if it attempts to cache content too
4225  * quickly, such as during backups of the entire pool.
4226  *
4227  * 5. After system boot and before the ARC has filled main memory, there are
4228  * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
4229  * lists can remain mostly static.  Instead of searching from tail of these
4230  * lists as pictured, the l2arc_feed_thread() will search from the list heads
4231  * for eligible buffers, greatly increasing its chance of finding them.
4232  *
4233  * The L2ARC device write speed is also boosted during this time so that
4234  * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
4235  * there are no L2ARC reads, and no fear of degrading read performance
4236  * through increased writes.
4237  *
4238  * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
4239  * the vdev queue can aggregate them into larger and fewer writes.  Each
4240  * device is written to in a rotor fashion, sweeping writes through
4241  * available space then repeating.
4242  *
4243  * 7. The L2ARC does not store dirty content.  It never needs to flush
4244  * write buffers back to disk based storage.
4245  *
4246  * 8. If an ARC buffer is written (and dirtied) which also exists in the
4247  * L2ARC, the now stale L2ARC buffer is immediately dropped.
4248  *
4249  * The performance of the L2ARC can be tweaked by a number of tunables, which
4250  * may be necessary for different workloads:
4251  *
4252  *      l2arc_write_max         max write bytes per interval
4253  *      l2arc_write_boost       extra write bytes during device warmup
4254  *      l2arc_noprefetch        skip caching prefetched buffers
4255  *      l2arc_headroom          number of max device writes to precache
4256  *      l2arc_headroom_boost    when we find compressed buffers during ARC
4257  *                              scanning, we multiply headroom by this
4258  *                              percentage factor for the next scan cycle,
4259  *                              since more compressed buffers are likely to
4260  *                              be present
4261  *      l2arc_feed_secs         seconds between L2ARC writing
4262  *
4263  * Tunables may be removed or added as future performance improvements are
```

```
4264  * integrated, and also may become zpool properties.
4265  *
4266  * There are three key functions that control how the L2ARC warms up:
4267  *
4268  *      l2arc_write_eligible()  check if a buffer is eligible to cache
4269  *      l2arc_write_size()      calculate how much to write
4270  *      l2arc_write_interval()  calculate sleep delay between writes
4271  *
4272  * These three functions determine what to write, how much, and how quickly
4273  * to send writes.
4274  *
4275  * L2ARC persistency:
4276  *
4277  * When writing buffers to L2ARC, we periodically add some metadata to
4278  * make sure we can pick them up after reboot, thus dramatically reducing
4279  * the impact that any downtime has on the performance of storage systems
4280  * with large caches.
4281  *
4282  * The implementation works fairly simply by integrating the following two
4283  * modifications:
4284  *
4285  * *) Every now and then, at end of an L2ARC feed cycle, we append a piece
4286  *    of metadata (called a "pbuf", or "persistency buffer") to the L2ARC
4287  *    write. This allows us to understand what what's been written, so that
4288  *    we can rebuild the arc_buf_hdr_t structures of the main ARC buffers.
4289  *    The pbuf also includes a "back-reference" pointer to the previous
4290  *    pbuf, forming a linked list of pbufs on the L2ARC device.
4291  *
4292  * *) We reserve 4k of space at the start of each L2ARC device for our
4293  *    header bookkeeping purposes. This contains a single 4k uberblock, which
4294  *    contains our top-level reference structures. We update it on each pbuf
4295  *    write. If this write results in an inconsistent uberblock (e.g. due to
4296  *    power failure), we detect this by verifying the uberblock's checksum
4297  *    and simply drop the entries from L2ARC. Once an L2ARC pbuf update
4298  *    completes, we update the uberblock to point to it.
4299  *
4300  * Implementation diagram:
4301  *
4302  * +=== L2ARC device (not to scale) ======================================+
4303  * |       ___________newest pbuf pointer_____________                   |
4304  * |      /                                           \                  |
4305  * |     /                                             V                 |
4306  * ||l2uberblock|---|bufs|pbuf|bufs|pbuf|bufs|pbuf|bufs|pbuf|---(empty)---|
4307  * |                     ^       / ^       / ^       /                  |
4308  * |                     `-prev-'  `-prev-'  `-prev-'                   |
4309  * |                       pbuf      pbuf      pbuf                     |
4310  * +======================================================================+
4311  *
4312  * On-device data structures:
4313  *
4314  * (L2ARC persistent uberblock)
4315  * struct l2uberblock {
4316  *      (these fields are in network byte order)
4317  *      uint32_t magic = 0x12bab10c;    l2-ber-block
4318  *      uint8_t  version = 0x1;
4319  *      uint8_t  reserved = 0x0;
4320  *      uint16_t ublk_flags;            see l2uberblock_flags_t
4321  *
4322  *      (byte order of fields below determined by 'ublk_flags')
4323  *      uint64_t spa_guid;              what pool this l2arc dev belongs to
4324  *      uint64_t birth_txg;             ublk with highest birth_txg is newest
4325  *      uint64_t evict_tail;            current evict pointer on l2arc dev
4326  *      uint64_t alloc_space;           how much space is alloc'd on the dev
4327  *      uint64_t pbuf_daddr;            dev addr of the newest l2pbuf_t
4328  *      uint32_t pbuf_asize;            size of newest pbuf
4329  *      uint64_t pbuf_cksum[4];         fletcher4 of newest pbuf
```



```
4330  *
4331  *      uint8_t  reserved[3996] = {0x0, 0x0, ... 0x0};
4332  *
4333  *      uint64_t ublk_cksum[4] = fletcher4(of the 4064 bytes above);
4334  * } l2dev_uberblock;
4335  *
4336  * (L2ARC persistent buffer list)
4337  * typedef struct l2pbuf_t {
4338  *      (these fields are in network byte order)
4339  *      uint32_t magic = 0xdb0faba6;    the-buffer-bag
4340  *      uint8_t  version = 0x1;
4341  *      uint8_t  reserved = 0x0;
4342  *      uint16_t pbuf_flags;            see l2pbuf_flags_t
4343  *
4344  *      (byte order of fields below determined by 'pbuf_flags')
4345  *      uint64_t prev_pbuf_daddr;       previous pbuf dev addr
4346  *      uint32_t prev_pbuf_asize;       previous pbuf size
4347  *      uint64_t prev_pbuf_cksum[4];    fletcher4(of previous pbuf)
4348  *
4349  *      uint32_t items_size;            uncompressed size of 'items' below
4350  *      (if (pbuf_flags & compress) decompress 'items' prior to decoding)
4351  *      struct l2pbuf_buf_item {
4352  *              (these fields mirror [l2]arc_buf_hdr fields)
4353  *              uint64_t dva[2];                buffer's DVA
4354  *              uint64_t birth;                 buffer's birth TXG in ARC
4355  *              uint64_t cksum0;                lower 64-bits of buffer's cksum
4356  *              uint64_t freeze_cksum[4];       buffer's freeze cksum
4357  *              uint32_t size;                  uncompressed buffer data size
4358  *              uint64_t l2daddr;               device address (offset) of buf
4359  *              uint32_t l2asize;               actual space occupied by buf
4360  *              uint8_t  compress;              compress algo used on data
4361  *              uint8_t  contents_type;         buffer's contents type
4362  *              uint16_t reserved = 0x0;        for alignment and future use
4363  *              uint32_t flags;                 buffer's persistent flags
4364  *      } items[];                              continues for remainder of pbuf
4365  * } l2pbuf_t;
4366  *
4367  * L2ARC reconstruction:
4368  *
4369  * When writing data, we simply write in the standard rotary fashion,
4370  * evicting buffers as we go and simply writing new data over them (appending
4371  * an updated l2pbuf_t every now and then). This obviously means that once we
4372  * loop around the end of the device, we will start cutting into an already
4373  * committed l2pbuf (and its referenced data buffers), like so:
4374  *
4375  *    current write head__       __old tail
4376  *                       \     /
4377  *                        V    V
4378  * <--|bufs|pbuf|bufs|pbuf|    |bufs|pbuf|bufs|pbuf|-->
4379  *                         ^    ^^^^^^^^^_____________________________
4380  *                         |                                          \
4381  *                         <<nextwrite>> - will overwrite this pbuf --/
4382  *
4383  * When importing the pool, we detect this situation and use it to stop
4384  * our scanning process:
4385  * 1) Let 'this_pbuf' refer to the current l2pbuf_t and 'prev_pbuf' to the
4386  *      previous one.
4387  * 2) if (fletcher4(prev_pbuf) != this_pbuf->prev_pbuf_cksum)
4388  *      then the pbuf is invalid and stop scanning (goto step 3 below).
4389  * 3) if (this is the last valid pbuf)
4390  *      discard this pbuf as well (its ARC bufs may have been damaged by a
4391  *      partial overwrite).
4392  * (We could potentially salvage the remaining good arc bufs above in step 3,
4393  * buf the cost of doing so probably outweighs the value of the entire pbuf).
4394  *
4395  * There is one significant caveat to consider when rebuilding ARC contents
```

```
 * from an L2ARC device: what about invalidated buffers? Given the above
 * construction, we cannot update pbufs which we've already written to amend
 * them to remove buffers which were invalidated. Thus, during reconstruction,
 * we might be populating the cache with buffers for data that's not on the
 * main pool anymore, or may have been overwritten!
 *
 * As it turns out, this isn't a problem. Every arc_read request includes
 * both the DVA and, crucially, the birth TXG of the BP the caller is
 * looking for. So even if the cache were populated by completely rotten
 * blocks for data that had been long deleted and/or overwritten, we'll
 * never actually return bad data from the cache, since the DVA with the
 * birth TXG uniquely identify a block in space and time - once created,
 * a block is immutable on disk. The worst thing we have done is wasted
 * some time and memory at l2arc rebuild to reconstruct outdated ARC
 * entries that will get dropped from the l2arc as it is being updated
 * with new blocks.
 */

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

        return (B_TRUE);
}
______unchanged_portion_omitted_

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

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

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

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



```
        first = NULL;
        next = l2arc_dev_last;
        do {
                /*
                 * Loop around the list looking for a non-faulted vdev
                 * and one that isn't currently doing an L2ARC rebuild.
                 */
                /* loop around the list looking for a non-faulted vdev */
                if (next == NULL) {
                        next = list_head(l2arc_dev_list);
                } else {
                        next = list_next(l2arc_dev_list, next);
                        if (next == NULL)
                                next = list_head(l2arc_dev_list);
                }

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

        } while (vdev_is_dead(next->l2ad_vdev) || next->l2ad_rebuilding);
        } while (vdev_is_dead(next->l2ad_vdev));

        /* if we were unable to find any usable vdevs, return NULL */
        if (vdev_is_dead(next->l2ad_vdev) || next->l2ad_rebuilding)
        if (vdev_is_dead(next->l2ad_vdev))
                next = NULL;

        l2arc_dev_last = next;

out:
        mutex_exit(&l2arc_dev_mtx);

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

        return (next);
}
______unchanged_portion_omitted_

/*
 * A write to a cache device has completed.  Update all headers to allow
 * reads from these buffers to begin.
 */
static void
l2arc_write_done(zio_t *zio)
{
        l2arc_write_callback_t *cb;
        l2arc_dev_t *dev;
        list_t *buflist;
        arc_buf_hdr_t *head, *ab, *ab_prev;
        l2arc_buf_hdr_t *abl2;
        kmutex_t *hash_lock;

        cb = zio->io_private;
        ASSERT(cb != NULL);
        dev = cb->l2wcb_dev;
        ASSERT(dev != NULL);
```

```
4598         head = cb->l2wcb_head;
4599         ASSERT(head != NULL);
4600         buflist = dev->l2ad_buflist;
4601         ASSERT(buflist != NULL);
4602         DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
4603             l2arc_write_callback_t *, cb);

4605         if (zio->io_error != 0)
4606                 ARCSTAT_BUMP(arcstat_l2_writes_error);

4608         mutex_enter(&l2arc_buflist_mtx);

4610         /*
4611          * All writes completed, or an error was hit.
4612          */
4613         for (ab = list_prev(buflist, head); ab; ab = ab_prev) {
4614                 ab_prev = list_prev(buflist, ab);
4615                 abl2 = ab->b_l2hdr;

4617                 /*
4618                  * Release the temporary compressed buffer as soon as possible.
4619                  */
4620                 if (abl2->b_compress != ZIO_COMPRESS_OFF)
4621                         l2arc_release_cdata_buf(ab);

4623                 hash_lock = HDR_LOCK(ab);
4624                 if (!mutex_tryenter(hash_lock)) {
4625                         /*
4626                          * This buffer misses out.  It may be in a stage
4627                          * of eviction.  Its ARC_L2_WRITING flag will be
4628                          * left set, denying reads to this buffer.
4629                          */
4630                         ARCSTAT_BUMP(arcstat_l2_writes_hdr_miss);
4631                         continue;
4632                 }

4187                 abl2 = ab->b_l2hdr;

4189                 /*
4190                  * Release the temporary compressed buffer as soon as possible.
4191                  */
4192                 if (abl2->b_compress != ZIO_COMPRESS_OFF)
4193                         l2arc_release_cdata_buf(ab);

4634                 if (zio->io_error != 0) {
4635                         /*
4636                          * Error - drop L2ARC entry.
4637                          */
4638                         list_remove(buflist, ab);
4639                         ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
4640                         ab->b_l2hdr = NULL;
4641                         kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
4642                         ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
4643                 }

4645                 /*
4646                  * Allow ARC to begin reads to this L2ARC entry.
4647                  */
4648                 ab->b_flags &= ~ARC_L2_WRITING;

4650                 mutex_exit(hash_lock);
4651         }

4653         atomic_inc_64(&l2arc_writes_done);
4654         list_remove(buflist, head);
4655         kmem_cache_free(hdr_cache, head);
```



```
4656         mutex_exit(&l2arc_buflist_mtx);

4658         l2arc_do_free_on_write();

4660         if (cb->l2wcb_pbuf)
4661                 kmem_free(cb->l2wcb_pbuf, cb->l2wcb_pbuf_size);
4662         if (cb->l2wcb_ub_buf)
4663                 kmem_free(cb->l2wcb_ub_buf, L2UBERBLOCK_SIZE);
4664         kmem_free(cb, sizeof (l2arc_write_callback_t));
4665 }
_____unchanged_portion_omitted_

4918 /*
4919  * Find and write ARC buffers to the L2ARC device.
4920  *
4921  * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4922  * for reading until they have completed writing.
4923  * The headroom_boost is an in-out parameter used to maintain headroom boost
4924  * state between calls to this function.
4925  *
4926  * Returns the number of bytes actually written (which may be smaller than
4927  * the delta by which the device hand has changed due to alignment).
4928  */
4929 static uint64_t
4930 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
4931     boolean_t *headroom_boost)
4932 {
4933         arc_buf_hdr_t *ab, *ab_prev, *head;
4934         list_t *list;
4935         uint64_t write_asize, write_psize, write_sz, headroom,
4936             buf_compress_minsz;
4937         void *buf_data;
4938         kmutex_t *list_lock;
4939         boolean_t full;
4940         l2arc_write_callback_t *cb;
4941         zio_t *pio, *wzio;
4942         uint64_t guid = spa_load_guid(spa);
4943         const boolean_t do_headroom_boost = *headroom_boost;

4945         /* persistency-related */
4946         l2pbuf_t *pb;
4947         l2pbuf_buflist_t *pb_buflist;
4948         int num_bufs, buf_index;

4950         ASSERT(dev->l2ad_vdev != NULL);

4952         /* Lower the flag now, we might want to raise it again later. */
4953         *headroom_boost = B_FALSE;

4955         pio = NULL;
4956         cb = NULL;
4957         write_sz = write_asize = write_psize = 0;
4958         full = B_FALSE;
4959         head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
4960         head->b_flags |= ARC_L2_WRITE_HEAD;

4962         /*
4963          * We will want to try to compress buffers that are at least 2x the
4964          * device sector size.
4965          */
4966         buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

4968         pb = &dev->l2ad_pbuf;
4969         num_bufs = 0;

4971         /*
```

```
4972          * We will want to try to compress buffers that are at least 2x the
4973          * device sector size.
4974          */
4975         buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

4977         /*
4978          * Copy buffers for L2ARC writing.
4979          */
4980         mutex_enter(&l2arc_buflist_mtx);
4981         for (int try = 0; try <= 3; try++) {
4982                 uint64_t passed_sz = 0;

4984                 list = l2arc_list_locked(try, &list_lock);

4986                 /*
4987                  * L2ARC fast warmup.
4988                  *
4989                  * Until the ARC is warm and starts to evict, read from the
4990                  * head of the ARC lists rather than the tail.
4991                  */
4992                 if (arc_warm == B_FALSE)
4993                         ab = list_head(list);
4994                 else
4995                         ab = list_tail(list);

4997                 headroom = target_sz * l2arc_headroom;
4998                 if (do_headroom_boost)
4999                         headroom = (headroom * l2arc_headroom_boost) / 100;

5001                 for (; ab; ab = ab_prev) {
5002                         l2arc_buf_hdr_t *l2hdr;
5003                         kmutex_t *hash_lock;
5004                         uint64_t buf_sz;

5006                         if (arc_warm == B_FALSE)
5007                                 ab_prev = list_next(list, ab);
5008                         else
5009                                 ab_prev = list_prev(list, ab);

5011                         hash_lock = HDR_LOCK(ab);
5012                         if (!mutex_tryenter(hash_lock)) {
5013                                 /*
5014                                  * Skip this buffer rather than waiting.
5015                                  */
5016                                 continue;
5017                         }

5019                         passed_sz += ab->b_size;
5020                         if (passed_sz > headroom) {
5021                                 /*
5022                                  * Searched too far.
5023                                  */
5024                                 mutex_exit(hash_lock);
5025                                 break;
5026                         }

5028                         if (!l2arc_write_eligible(guid, ab)) {
5029                                 mutex_exit(hash_lock);
5030                                 continue;
5031                         }

5033                         if ((write_sz + ab->b_size) > target_sz) {
5034                                 full = B_TRUE;
5035                                 mutex_exit(hash_lock);
5036                                 break;
5037                         }
```



```
5039                         if (pio == NULL) {
5040                                 /*
5041                                  * Insert a dummy header on the buflist so
5042                                  * l2arc_write_done() can find where the
5043                                  * write buffers begin without searching.
5044                                  */
5045                                 list_insert_head(dev->l2ad_buflist, head);

5047                                 cb = kmem_zalloc(
4589                                 cb = kmem_alloc(
5048                                     sizeof (l2arc_write_callback_t), KM_SLEEP);
5049                                 cb->l2wcb_dev = dev;
5050                                 cb->l2wcb_head = head;
5051                                 pio = zio_root(spa, l2arc_write_done, cb,
5052                                     ZIO_FLAG_CANFAIL);
5053                         }

5055                         /*
5056                          * Create and add a new L2ARC header.
5057                          */
5058                         l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
5059                         l2hdr->b_dev = dev;
5060                         ab->b_flags |= ARC_L2_WRITING;

5062                         /*
5063                          * Temporarily stash the data buffer in b_tmp_cdata.
5064                          * The subsequent write step will pick it up from
5065                          * there. This is because can't access ab->b_buf
5066                          * without holding the hash_lock, which we in turn
5067                          * can't access without holding the ARC list locks
5068                          * (which we want to avoid during compression/writing).
5069                          */
5070                         l2hdr->b_compress = ZIO_COMPRESS_OFF;
5071                         l2hdr->b_asize = ab->b_size;
5072                         l2hdr->b_tmp_cdata = ab->b_buf->b_data;

5074                         buf_sz = ab->b_size;
5075                         ab->b_l2hdr = l2hdr;

5077                         list_insert_head(dev->l2ad_buflist, ab);

5079                         /*
5080                          * Compute and store the buffer cksum before
5081                          * writing.  On debug the cksum is verified first.
5082                          */
5083                         arc_cksum_verify(ab->b_buf);
5084                         arc_cksum_compute(ab->b_buf, B_TRUE);

5086                         mutex_exit(hash_lock);

5088                         write_sz += buf_sz;
5089                         num_bufs++;
5090                 }

5092                 mutex_exit(list_lock);

5094                 if (full == B_TRUE)
5095                         break;
5096         }

5098         /* No buffers selected for writing? */
5099         if (pio == NULL) {
5100                 ASSERT0(write_sz);
5101                 mutex_exit(&l2arc_buflist_mtx);
5102                 kmem_cache_free(hdr_cache, head);
```

```
5103                return (0);
5104        }

5106        /* expand the pbuf to include a new list */
5107        pb_buflist = l2arc_pbuf_buflist_alloc(pb, num_bufs);

5109        /*
5110         * Now start writing the buffers. We're starting at the write head
5111         * and work backwards, retracing the course of the buffer selector
5112         * loop above.
5113         */
5114        for (ab = list_prev(dev->l2ad_buflist, head), buf_index = 0; ab;
5115            ab = list_prev(dev->l2ad_buflist, ab), buf_index++) {
4652        for (ab = list_prev(dev->l2ad_buflist, head); ab;
4653            ab = list_prev(dev->l2ad_buflist, ab)) {
5116                l2arc_buf_hdr_t *l2hdr;
5117                uint64_t buf_sz;

5119                /*
5120                 * We shouldn't need to lock the buffer here, since we flagged
5121                 * it as ARC_L2_WRITING in the previous step, but we must take
5122                 * care to only access its L2 cache parameters. In particular,
5123                 * ab->b_buf may be invalid by now due to ARC eviction.
5124                 */
5125                l2hdr = ab->b_l2hdr;
5126                l2hdr->b_daddr = dev->l2ad_hand;

5128                if ((ab->b_flags & ARC_L2COMPRESS) &&
5129                    l2hdr->b_asize >= buf_compress_minsz) {
5130                        if (l2arc_compress_buf(l2hdr)) {
5131                                /*
5132                                 * If compression succeeded, enable headroom
5133                                 * boost on the next scan cycle.
5134                                 */
5135                                *headroom_boost = B_TRUE;
5136                        }
5137                }

5139                /*
5140                 * Pick up the buffer data we had previously stashed away
5141                 * (and now potentially also compressed).
5142                 */
5143                buf_data = l2hdr->b_tmp_cdata;
5144                buf_sz = l2hdr->b_asize;

5146                /* Compression may have squashed the buffer to zero length. */
5147                if (buf_sz != 0) {
5148                        uint64_t buf_p_sz;

5150                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
5151                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
5152                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
5153                            ZIO_FLAG_CANFAIL, B_FALSE);

5155                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
5156                            zio_t *, wzio);
5157                        (void) zio_nowait(wzio);

5159                        write_asize += buf_sz;
5160                        /*
5161                         * Keep the clock hand suitably device-aligned.
5162                         */
5163                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
5164                        write_psize += buf_p_sz;
5165                        dev->l2ad_hand += buf_p_sz;
5166                }
```



```
5168                        l2arc_pbuflist_insert(pb, pb_buflist, ab, buf_index);
5169                }
5170        ASSERT(buf_index == num_bufs);

5171        mutex_exit(&l2arc_buflist_mtx);

5173        ASSERT3U(write_asize, <=, target_sz);
5174        ARCSTAT_BUMP(arcstat_l2_writes_sent);
5175        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
5176        ARCSTAT_INCR(arcstat_l2_size, write_sz);
5177        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
5178        vdev_space_update(dev->l2ad_vdev, write_psize, 0, 0);

5180        /* Is it time to commit this pbuf? */
5181        if (L2PBUF_IS_FULL(pb) &&
5182            dev->l2ad_hand + L2PBUF_ENCODED_SIZE(pb) < dev->l2ad_end) {
5183                l2arc_pbuf_commit(dev, pio, cb);
5184                l2arc_pbuf_destroy(pb);
5185                l2arc_pbuf_init(pb);
5186        }

5188        /*
5189         * Bump device hand to the device start if it is approaching the end.
5190         * l2arc_evict() will already have evicted ahead for this case.
5191         */
5192        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
5193                vdev_space_update(dev->l2ad_vdev,
5194                    dev->l2ad_end - dev->l2ad_hand, 0, 0);
5195                dev->l2ad_hand = dev->l2ad_start;
5196                dev->l2ad_evict = dev->l2ad_start;
5197                dev->l2ad_first = B_FALSE;
5198        }

5200        dev->l2ad_writing = B_TRUE;
5201        (void) zio_wait(pio);
5202        dev->l2ad_writing = B_FALSE;

5204        return (write_asize);
5205 }
_____unchanged_portion_omitted_

5467 /*
5468  * Add a vdev for use by the L2ARC.  By this point the spa has already
5469  * validated the vdev and opened it. The 'rebuild' flag indicates whether
5470  * we should attempt an L2ARC persistency rebuild.
4997  * validated the vdev and opened it.
5471  */
5472 void
5473 l2arc_add_vdev(spa_t *spa, vdev_t *vd, boolean_t rebuild)
5000 l2arc_add_vdev(spa_t *spa, vdev_t *vd)
5474 {
5475        l2arc_dev_t *adddev;

5477        ASSERT(!l2arc_vdev_present(vd));

5479        /*
5480         * Create a new l2arc device entry.
5481         */
5482        adddev = kmem_zalloc(sizeof (l2arc_dev_t), KM_SLEEP);
5483        adddev->l2ad_spa = spa;
5484        adddev->l2ad_vdev = vd;
5485        adddev->l2ad_start = VDEV_LABEL_START_SIZE + L2UBERBLOCK_SIZE;
5012        adddev->l2ad_start = VDEV_LABEL_START_SIZE;
5486        adddev->l2ad_end = VDEV_LABEL_START_SIZE + vdev_get_min_asize(vd);
5487        adddev->l2ad_hand = adddev->l2ad_start;
```

```
5488         adddev->l2ad_evict = adddev->l2ad_start;
5489         adddev->l2ad_first = B_TRUE;
5490         adddev->l2ad_writing = B_FALSE;
5491         l2arc_pbuf_init(&adddev->l2ad_pbuf);

5493         /*
5494          * This is a list of all ARC buffers that are still valid on the
5495          * device.
5496          */
5497         adddev->l2ad_buflist = kmem_zalloc(sizeof (list_t), KM_SLEEP);
5498         list_create(adddev->l2ad_buflist, sizeof (arc_buf_hdr_t),
5499             offsetof(arc_buf_hdr_t, b_l2node));

5501         vdev_space_update(vd, 0, 0, adddev->l2ad_end - adddev->l2ad_hand);

5503         /*
5504          * Add device to global list
5505          */
5506         mutex_enter(&l2arc_dev_mtx);
5507         list_insert_head(l2arc_dev_list, adddev);
5508         atomic_inc_64(&l2arc_ndev);
5509         if (rebuild && l2arc_rebuild_enabled) {
5510                 adddev->l2ad_rebuilding = B_TRUE;
5511                 (void) thread_create(NULL, 0, l2arc_rebuild_start, adddev,
5512                     0, &p0, TS_RUN, minclsyspri);
5513         }
5514         mutex_exit(&l2arc_dev_mtx);
5515 }

5517 /*
5518  * Remove a vdev from the L2ARC.
5519  */
5520 void
5521 l2arc_remove_vdev(vdev_t *vd)
5522 {
5523         l2arc_dev_t *dev, *nextdev, *remdev = NULL;

5525         /*
5526          * Find the device by vdev
5527          */
5528         mutex_enter(&l2arc_dev_mtx);
5529         for (dev = list_head(l2arc_dev_list); dev; dev = nextdev) {
5530                 nextdev = list_next(l2arc_dev_list, dev);
5531                 if (vd == dev->l2ad_vdev) {
5532                         remdev = dev;
5533                         break;
5534                 }
5535         }
5536         ASSERT(remdev != NULL);

5538         /*
5539          * Remove device from global list
5540          */
5541         list_remove(l2arc_dev_list, remdev);
5542         l2arc_dev_last = NULL;          /* may have been invalidated */
5543         atomic_dec_64(&l2arc_ndev);
5544         mutex_exit(&l2arc_dev_mtx);

5546         /*
5547          * Clear all buflists and ARC references.  L2ARC device flush.
5548          */
5549         l2arc_pbuf_destroy(&remdev->l2ad_pbuf);
5550         l2arc_evict(remdev, 0, B_TRUE);
5551         list_destroy(remdev->l2ad_buflist);
5552         kmem_free(remdev->l2ad_buflist, sizeof (list_t));
5553         kmem_free(remdev, sizeof (l2arc_dev_t));
```



```
5554 }
______unchanged_portion_omitted_

5609 void
5610 l2arc_stop(void)
5611 {
5612         if (!(spa_mode_global & FWRITE))
5613                 return;

5615         mutex_enter(&l2arc_feed_thr_lock);
5616         cv_signal(&l2arc_feed_thr_cv);  /* kick thread out of startup */
5617         l2arc_thread_exit = 1;
5618         while (l2arc_thread_exit != 0)
5619                 cv_wait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock);
5620         mutex_exit(&l2arc_feed_thr_lock);
5621 }

5623 /*
5624  * Main entry point for L2ARC metadata rebuilding. This function must be
5625  * called via thread_create so that the L2ARC metadata rebuild doesn't block
5626  * pool import and may proceed in parallel on all available L2ARC devices.
5627  */
5628 static void
5629 l2arc_rebuild_start(l2arc_dev_t *dev)
5630 {
5631         vdev_t *vd = dev->l2ad_vdev;
5632         spa_t *spa = dev->l2ad_spa;

5634         /* Lock out device removal. */
5635         spa_config_enter(spa, SCL_L2ARC, vd, RW_READER);
5636         ASSERT(dev->l2ad_rebuilding == B_TRUE);
5637         l2arc_rebuild(dev);
5638         dev->l2ad_rebuilding = B_FALSE;
5639         spa_config_exit(spa, SCL_L2ARC, vd);
5640         thread_exit();
5641 }

5643 /*
5644  * This function implements the actual L2ARC metadata rebuild. It:
5645  *
5646  * 1) scans the device for valid l2uberblocks
5647  * 2) if it finds a good uberblock, starts reading the pbuf chain
5648  * 3) restores each pbuf's contents to memory
5649  *
5650  * Operation stops under any of the following conditions:
5651  *
5652  * 1) We reach the end of the pbuf chain (the previous-buffer reference
5653  *    in the pbuf is zero).
5654  * 2) We encounter *any* error condition (cksum errors, io errors, looped
5655  *    pbufs, etc.).
5656  * 3) The l2arc_rebuild_timeout is hit - this is a final resort to protect
5657  *    from making severely fragmented L2ARC pbufs or slow L2ARC devices
5658  *    prevent a machine from importing the pool (and letting the
5659  *    administrator take corrective action, e.g. by kicking the misbehaving
5660  *    L2ARC device out of the pool, or by reimporting the pool with L2ARC
5661  *    rebuilding disabled).
5662  */
5663 static void
5664 l2arc_rebuild(l2arc_dev_t *dev)
5665 {
5666         int err;
5667         l2uberblock_t ub;
5668         l2pbuf_t pb;
5669         zio_t *this_io = NULL, *next_io = NULL;
5670         int64_t deadline = ddi_get_lbolt64() + hz * l2arc_rebuild_timeout;
```

```
5672         if ((err = l2arc_uberblock_find(dev, &ub)) != 0)
5673                 return;
5674         L2ARC_CHK_REBUILD_TIMEOUT(deadline, /* nop */);

5676         /* set up uberblock update info */
5677         dev->l2ad_uberblock_birth = ub.ub_birth + 1;

5679         /* initial sanity checks */
5680         l2arc_pbuf_init(&pb);
5681         if ((err = l2arc_pbuf_read(dev, ub.ub_pbuf_daddr, ub.ub_pbuf_asize,
5682             ub.ub_pbuf_cksum, &pb, NULL, &this_io)) != 0) {
5683                 /* root pbuf is bad, we can't do anything about that */
5684                 if (err == EINVAL) {
5685                         ARCSTAT_BUMP(arcstat_l2_rebuild_cksum_errors);
5686                 } else {
5687                         ARCSTAT_BUMP(arcstat_l2_rebuild_io_errors);
5688                 }
5689                 l2arc_pbuf_destroy(&pb);
5690                 return;
5691         }
5692         L2ARC_CHK_REBUILD_TIMEOUT(deadline, l2arc_pbuf_destroy(&pb));

5694         dev->l2ad_evict = ub.ub_evict_tail;

5696         /* keep on chaining in new blocks */
5697         dev->l2ad_pbuf_daddr = ub.ub_pbuf_daddr;
5698         dev->l2ad_pbuf_asize = ub.ub_pbuf_asize;
5699         dev->l2ad_pbuf_cksum = ub.ub_pbuf_cksum;
5700         dev->l2ad_hand = vdev_psize_to_asize(dev->l2ad_vdev,
5701             ub.ub_pbuf_daddr + ub.ub_pbuf_asize);
5702         dev->l2ad_first = ((ub.ub_flags & L2UBLK_EVICT_FIRST) != 0);

5704         /* start the rebuild process */
5705         for (;;) {
5706                 l2pbuf_t pb_prev;

5708                 l2arc_pbuf_init(&pb_prev);
5709                 if ((err = l2arc_pbuf_read(dev, pb.pb_prev_daddr,
5710                     pb.pb_prev_asize, pb.pb_prev_cksum, &pb_prev, this_io,
5711                     &next_io)) != 0) {
5712                         /*
5713                          * We are done reading, discard the last good buffer.
5714                          */
5715                         if (pb.pb_prev_daddr > dev->l2ad_hand &&
5716                             pb.pb_prev_asize > L2PBUF_HDR_SIZE) {
5717                                 /* this is an error, we stopped too early */
5718                                 if (err == EINVAL) {
5719                                         ARCSTAT_BUMP(
5720                                             arcstat_l2_rebuild_cksum_errors);
5721                                 } else {
5722                                         ARCSTAT_BUMP(
5723                                             arcstat_l2_rebuild_io_errors);
5724                                 }
5725                         }
5726                         l2arc_pbuf_destroy(&pb_prev);
5727                         l2arc_pbuf_destroy(&pb);
5728                         break;
5729                 }

5731                 /*
5732                  * Protection against infinite loops of pbufs. This is also
5733                  * our primary termination mechanism - once the buffer list
5734                  * loops around our starting pbuf, we can stop.
5735                  */
5736                 if (pb.pb_prev_daddr >= ub.ub_pbuf_daddr &&
5737                     pb_prev.pb_prev_daddr <= ub.ub_pbuf_daddr) {
```



```
5738                         ARCSTAT_BUMP(arcstat_l2_rebuild_loop_errors);
5739                         l2arc_pbuf_destroy(&pb);
5740                         l2arc_pbuf_destroy(&pb_prev);
5741                         if (next_io)
5742                                 l2arc_pbuf_prefetch_abort(next_io);
5743                         return;
5744                 }

5746                 /*
5747                  * Our memory pressure valve. If the system is running low
5748                  * on memory, rather than swamping memory with new ARC buf
5749                  * hdrs, we opt not to reconstruct the L2ARC. At this point,
5750                  * however, we have already set up our L2ARC dev to chain in
5751                  * new metadata pbufs, so the user may choose to re-add the
5752                  * L2ARC dev at a later time to reconstruct it (when there's
5753                  * less memory pressure).
5754                  */
5755                 if (arc_reclaim_needed()) {
5756                         ARCSTAT_BUMP(arcstat_l2_rebuild_abort_lowmem);
5757                         cmn_err(CE_NOTE, "System running low on memory, "
5758                             "aborting L2ARC rebuild.");
5759                         l2arc_pbuf_destroy(&pb);
5760                         l2arc_pbuf_destroy(&pb_prev);
5761                         if (next_io)
5762                                 l2arc_pbuf_prefetch_abort(next_io);
5763                         break;
5764                 }

5766                 /*
5767                  * Now that we know that the prev_pbuf checks out alright, we
5768                  * can start reconstruction from this pbuf - we can be sure
5769                  * that the L2ARC write hand has not yet reached any of our
5770                  * buffers.
5771                  */
5772                 l2arc_pbuf_restore(dev, &pb);

5774                 /* pbuf restored, continue with next one in the list */
5775                 l2arc_pbuf_destroy(&pb);
5776                 pb = pb_prev;
5777                 this_io = next_io;
5778                 next_io = NULL;

5780                 L2ARC_CHK_REBUILD_TIMEOUT(deadline, l2arc_pbuf_destroy(&pb));
5781         }

5783         ARCSTAT_BUMP(arcstat_l2_rebuild_successes);
5784 }

5786 /*
5787  * Restores the payload of a pbuf to ARC. This creates empty ARC hdr entries
5788  * which only contain an l2arc hdr, essentially restoring the buffers to
5789  * their L2ARC evicted state. This function also updates space usage on the
5790  * L2ARC vdev to make sure it tracks restored buffers.
5791  */
5792 static void
5793 l2arc_pbuf_restore(l2arc_dev_t *dev, l2pbuf_t *pb)
5794 {
5795         spa_t *spa;
5796         uint64_t guid;
5797         list_t *buflists_list;
5798         l2pbuf_buflist_t *buflist;

5800         mutex_enter(&l2arc_buflist_mtx);
5801         spa = dev->l2ad_vdev->vdev_spa;
5802         guid = spa_load_guid(spa);
5803         buflists_list = pb->pb_buflists_list;
```

```
5804         for (buflist = list_head(buflists_list); buflist;
5805             buflist = list_next(buflists_list, buflist)) {
5806                 int i;
5807                 uint64_t size, asize, psize;
5808
5809                 size = asize = psize = 0;
5810                 for (i = 0; i < buflist->l2pbl_nbufs; i++) {
5811                         l2arc_hdr_restore(&buflist->l2pbl_bufs[i], dev,
5812                             guid);
5813                         size += buflist->l2pbl_bufs[i].b_size;
5814                         asize += buflist->l2pbl_bufs[i].b_l2asize;
5815                         psize += vdev_psize_to_asize(dev->l2ad_vdev,
5816                             buflist->l2pbl_bufs[i].b_l2asize);
5817                 }
5818                 ARCSTAT_INCR(arcstat_l2_rebuild_arc_bytes, size);
5819                 ARCSTAT_INCR(arcstat_l2_rebuild_l2arc_bytes, asize);
5820                 ARCSTAT_INCR(arcstat_l2_rebuild_bufs, buflist->l2pbl_nbufs);
5821                 vdev_space_update(dev->l2ad_vdev, psize, 0, 0);
5822         }
5823         mutex_exit(&l2arc_buflist_mtx);
5824         ARCSTAT_BUMP(arcstat_l2_rebuild_metabufs);
5825         vdev_space_update(dev->l2ad_vdev, vdev_psize_to_asize(dev->l2ad_vdev,
5826             pb->pb_asize), 0, 0);
5827 }
5828
5829 /*
5830  * Restores a single ARC buf hdr from a pbuf. The ARC buffer is put into
5831  * a state indicating that it has been evicted to L2ARC.
5832  * The ‘guid’ here is the ARC-load-guid from spa_load_guid.
5833  */
5834 static void
5835 l2arc_hdr_restore(const l2pbuf_buf_t *buf, l2arc_dev_t *dev, uint64_t guid)
5836 {
5837         arc_buf_hdr_t *hdr;
5838         kmutex_t *hash_lock;
5839         dva_t dva = {buf->b_dva.dva_word[0], buf->b_dva.dva_word[1]};
5840
5841         hdr = buf_hash_find(guid, &dva, buf->b_birth, &hash_lock);
5842         if (hdr == NULL) {
5843                 /* not in cache, try to insert */
5844                 arc_buf_hdr_t *exists;
5845                 arc_buf_contents_t type = buf->b_contents_type;
5846                 l2arc_buf_hdr_t *l2hdr;
5847
5848                 hdr = arc_buf_hdr_alloc(guid, buf->b_size, type);
5849                 hdr->b_dva = buf->b_dva;
5850                 hdr->b_birth = buf->b_birth;
5851                 hdr->b_cksum0 = buf->b_cksum0;
5852                 hdr->b_size = buf->b_size;
5853                 exists = buf_hash_insert(hdr, &hash_lock);
5854                 if (exists) {
5855                         /* somebody beat us to the hash insert */
5856                         mutex_exit(hash_lock);
5857                         arc_hdr_destroy(hdr);
5858                         ARCSTAT_BUMP(arcstat_l2_rebuild_bufs_precached);
5859                         return;
5860                 }
5861                 hdr->b_flags = buf->b_flags;
5862                 mutex_enter(&hdr->b_freeze_lock);
5863                 ASSERT(hdr->b_freeze_cksum == NULL);
5864                 hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t),
5865                     KM_SLEEP);
5866                 *hdr->b_freeze_cksum = buf->b_freeze_cksum;
5867                 mutex_exit(&hdr->b_freeze_lock);
5868
5869                 /* now rebuild the l2arc entry */
```



```
5870                 ASSERT(hdr->b_l2hdr == NULL);
5871                 l2hdr = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
5872                 l2hdr->b_dev = dev;
5873                 l2hdr->b_daddr = buf->b_l2daddr;
5874                 l2hdr->b_asize = buf->b_l2asize;
5875                 l2hdr->b_compress = buf->b_l2compress;
5876                 hdr->b_l2hdr = l2hdr;
5877                 list_insert_head(dev->l2ad_buflist, hdr);
5878                 ARCSTAT_INCR(arcstat_l2_size, hdr->b_size);
5879                 ARCSTAT_INCR(arcstat_l2_asize, l2hdr->b_asize);
5880
5881                 arc_change_state(arc_l2c_only, hdr, hash_lock);
5882         }
5883         mutex_exit(hash_lock);
5884 }
5885
5886 /*
5887  * Attempts to locate and read the newest valid uberblock on the provided
5888  * L2ARC device and writes it to ‘ub’. On success, this function returns 0,
5889  * otherwise the appropriate error code is returned.
5890  */
5891 static int
5892 l2arc_uberblock_find(l2arc_dev_t *dev, l2uberblock_t *ub)
5893 {
5894         int err = 0;
5895         uint8_t *ub_buf;
5896         uint64_t guid;
5897
5898         ARCSTAT_BUMP(arcstat_l2_rebuild_attempts);
5899         ub_buf = kmem_alloc(L2UBERBLOCK_SIZE, KM_SLEEP);
5900         guid = spa_guid(dev->l2ad_vdev->vdev_spa);
5901
5902         if ((err = zio_wait(zio_read_phys(NULL, dev->l2ad_vdev,
5903             VDEV_LABEL_START_SIZE, L2UBERBLOCK_SIZE, ub_buf,
5904             ZIO_CHECKSUM_OFF, NULL, NULL, ZIO_PRIORITY_ASYNC_READ,
5905             ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
5906             ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY, B_FALSE))) != 0) {
5907                 ARCSTAT_BUMP(arcstat_l2_rebuild_io_errors);
5908                 goto cleanup;
5909         }
5910
5911         /*
5912          * Initial peek - does the device even have any usable uberblocks?
5913          * If not, don’t bother continuing.
5914          */
5915         l2arc_uberblock_decode(ub_buf, ub);
5916         if (ub->ub_magic != L2UBERBLOCK_MAGIC || ub->ub_version == 0 ||
5917             ub->ub_version > L2UBERBLOCK_MAX_VERSION ||
5918             ub->ub_spa_guid != guid) {
5919                 err = ENOTSUP;
5920                 ARCSTAT_BUMP(arcstat_l2_rebuild_unsupported);
5921                 goto cleanup;
5922         }
5923
5924         /* now check to make sure that what we selected is okay */
5925         if ((err = l2arc_uberblock_verify(ub_buf, ub, guid)) != 0) {
5926                 if (err == EINVAL) {
5927                         ARCSTAT_BUMP(arcstat_l2_rebuild_cksum_errors);
5928                 } else {
5929                         ARCSTAT_BUMP(arcstat_l2_rebuild_uberblk_errors);
5930                 }
5931                 goto cleanup;
5932         }
5933
5934         /* this uberblock is valid */
```

```
5936 cleanup:
5937         kmem_free(ub_buf, L2UBERBLOCK_SIZE);
5938         return (err);
5939 }

5941 /*
5942  * Reads a pbuf from storage, decodes it and validates its contents against
5943  * the provided checksum. The result is placed in ‘pb’.
5944  *
5945  * The ‘this_io’ and ‘prefetch_io’ arguments are used for pbuf prefetching.
5946  * When issuing the first pbuf IO during rebuild, you should pass NULL for
5947  * ‘this_io’. This function will then issue a sync IO to read the pbuf and
5948  * also issue an async IO to fetch the next pbuf in the pbuf chain. The
5949  * prefetch IO is returned in ‘prefetch_io. On subsequent calls to this
5950  * function, pass the value returned in ‘prefetch_io’ from the previous
5951  * call as ‘this_io’ and a fresh ‘prefetch_io’ pointer to hold the next
5952  * prefetch IO. Prior to the call, you should initialize your ‘prefetch_io’
5953  * pointer to be NULL. If no prefetch IO was issued, the pointer is left
5954  * set at NULL.
5955  *
5956  * Actual prefetching takes place in two steps: a header IO (pi_hdr_io)
5957  * and the main pbuf payload IO (placed in prefetch_io). The pi_hdr_io
5958  * IO is used internally in this function to be able to ‘peek’ at the next
5959  * buffer’s header before the main IO to read it in completely has finished.
5960  * We can then begin to issue the IO for the next buffer in the chain before
5961  * we are done reading, keeping the L2ARC device’s pipeline saturated with
5962  * reads (rather than issuing an IO, waiting for it to complete, validating
5963  * the returned buffer and issuing the next one). This will make sure that
5964  * the rebuild proceeds at maximum read throughput.
5965  *
5966  * On success, this function returns 0, otherwise it returns an appropriate
5967  * error code. On error the prefetching IO is aborted and cleared before
5968  * returning from this function. Therefore, if we return ‘success’, the
5969  * caller can assume that we have taken care of cleanup of prefetch IOs.
5970  */
5971 static int
5972 l2arc_pbuf_read(l2arc_dev_t *dev, uint64_t daddr, uint32_t asize,
5973     zio_cksum_t cksum, l2pbuf_t *pb, zio_t *this_io, zio_t **prefetch_io)
5974 {
5975         int err = 0;
5976         uint64_t prev_pb_start;
5977         uint32_t prev_pb_asize;
5978         zio_cksum_t calc_cksum, prev_pb_cksum;
5979         l2arc_prefetch_info_t *pi = NULL;

5981         ASSERT(dev != NULL);
5982         ASSERT(pb != NULL);
5983         ASSERT(*prefetch_io == NULL);

5985         if (!l2arc_pbuf_ptr_valid(dev, daddr, asize)) {
5986                 /* We could not have issued a prefetch IO for this */
5987                 ASSERT(this_io == NULL);
5988                 return (EINVAL);
5989         }

5991         /*
5992          * Check to see if we have issued the IO for this pbuf in a previous
5993          * run. If not, issue it now.
5994          */
5995         if (this_io == NULL)
5996                 this_io = l2arc_pbuf_prefetch(dev->l2ad_vdev, daddr, asize);

5998         /* Pick up the prefetch info buffer and read its contents */
5999         pi = this_io->io_private;
6000         ASSERT(pi != NULL);
6001         ASSERT(asize <= pi->pi_buflen);
```



```
6003         /* Wait for the IO to read this pbuf’s header to complete */
6004         if ((err = zio_wait(pi->pi_hdr_io)) != 0) {
6005                 (void) zio_wait(this_io);
6006                 goto cleanup;
6007         }

6009         /*
6010          * Peek to see if we can start issuing the next pbuf IO immediately.
6011          * At this point, only the current pbuf’s header has been read.
6012          */
6013         if (l2arc_pbuf_decode_prev_ptr(pi->pi_buf, asize, &prev_pb_start,
6014             &prev_pb_asize, &prev_pb_cksum) == 0) {
6015                 uint64_t this_pb_start, this_pb_end, prev_pb_end;
6016                 /* Detect malformed pbuf references and loops */
6017                 this_pb_start = daddr;
6018                 this_pb_end = daddr + asize;
6019                 prev_pb_end = prev_pb_start + prev_pb_asize;
6020                 if ((prev_pb_start >= this_pb_start && prev_pb_start <
6021                     this_pb_end) ||
6022                     (prev_pb_end >= this_pb_start && prev_pb_end <
6023                     this_pb_end)) {
6024                         ARCSTAT_BUMP(arcstat_l2_rebuild_loop_errors);
6025                         cmn_err(CE_WARN, "Looping L2ARC metadata reference "
6026                             "detected, aborting rebuild.");
6027                         err = EINVAL;
6028                         goto cleanup;
6029                 }
6030                 /*
6031                  * Start issuing IO for the next pbuf early - this should
6032                  * help keep the L2ARC device busy while we read, decode
6033                  * and restore this pbuf.
6034                  */
6035                 if (l2arc_pbuf_ptr_valid(dev, prev_pb_start, prev_pb_asize))
6036                         *prefetch_io = l2arc_pbuf_prefetch(dev->l2ad_vdev,
6037                             prev_pb_start, prev_pb_asize);
6038         }

6040         /* Wait for the main pbuf IO to complete */
6041         if ((err = zio_wait(this_io)) != 0)
6042                 goto cleanup;

6044         /* Make sure the buffer checks out ok */
6045         fletcher_4_native(pi->pi_buf, asize, &calc_cksum);
6046         if (!ZIO_CHECKSUM_EQUAL(calc_cksum, cksum)) {
6047                 err = EINVAL;
6048                 goto cleanup;
6049         }

6051         /* Now we can take our time decoding this buffer */
6052         if ((err = l2arc_pbuf_decode(pi->pi_buf, asize, pb)) != 0)
6053                 goto cleanup;

6055         /* This will be used in l2arc_pbuf_restore for space accounting */
6056         pb->pb_asize = asize;

6058         ARCSTAT_F_AVG(arcstat_l2_meta_avg_size, L2PBUF_ENCODED_SIZE(pb));
6059         ARCSTAT_F_AVG(arcstat_l2_meta_avg_asize, asize);
6060         ARCSTAT_F_AVG(arcstat_l2_asize_to_meta_ratio,
6061             pb->pb_payload_asz / asize);

6063 cleanup:
6064         kmem_free(pi->pi_buf, pi->pi_buflen);
6065         pi->pi_buf = NULL;
6066         kmem_free(pi, sizeof (l2arc_prefetch_info_t));
6067         /* Abort an in-flight prefetch in case of error */
```

```
6068         if (err != 0 && *prefetch_io != NULL) {
6069                 l2arc_pbuf_prefetch_abort(*prefetch_io);
6070                 *prefetch_io = NULL;
6071         }
6072         return (err);
6073 }

6075 /*
6076  * Validates a pbuf device address to make sure that it can be read
6077  * from the provided L2ARC device. Returns 1 if the address is within
6078  * the device's bounds, or 0 if not.
6079  */
6080 static int
6081 l2arc_pbuf_ptr_valid(l2arc_dev_t *dev, uint64_t daddr, uint32_t asize)
6082 {
6083         uint32_t psize;
6084         uint64_t end;

6086         psize = vdev_psize_to_asize(dev->l2ad_vdev, asize);
6087         end = daddr + psize;

6089         if (end > dev->l2ad_end || asize < L2PBUF_HDR_SIZE ||
6090             asize > L2PBUF_MAX_PAYLOAD_SIZE || daddr < dev->l2ad_start ||
6091             /* check that the buffer address is correctly aligned */
6092             (daddr & (vdev_psize_to_asize(dev->l2ad_vdev,
6093             SPA_MINBLOCKSIZE) - 1)) != 0)
6094                 return (0);
6095         else
6096                 return (1);
6097 }

6099 /*
6100  * Starts an asynchronous read IO to read a pbuf. This is used in pbuf
6101  * reconstruction to start reading the next pbuf before we are done
6102  * decoding and reconstructing the current pbuf, to keep the l2arc device
6103  * nice and hot with read IO to process.
6104  * The returned zio will contain a newly allocated memory buffers for the IO
6105  * data which should then be freed by the caller once the zio is no longer
6106  * needed (i.e. due to it having completed). If you wish to abort this
6107  * zio, you should do so using l2arc_pbuf_prefetch_abort, which takes care
6108  * of disposing of the allocated buffers correctly.
6109  */
6110 static zio_t *
6111 l2arc_pbuf_prefetch(vdev_t *vd, uint64_t daddr, uint32_t asize)
6112 {
6113         uint32_t i, psize;
6114         zio_t *pio, *hdr_io;
6115         uint64_t hdr_rsize;
6116         uint8_t *buf;
6117         l2arc_prefetch_info_t *pinfo;

6119         psize = vdev_psize_to_asize(vd, asize);
6120         buf = kmem_alloc(psize, KM_SLEEP);
6121         pinfo = kmem_alloc(sizeof (l2arc_prefetch_info_t), KM_SLEEP);
6122         pinfo->pi_buf = buf;
6123         pinfo->pi_buflen = psize;

6125         /*
6126          * We start issuing the IO for the pbuf header early. This
6127          * allows l2arc_pbuf_read to start issuing IO for the next
6128          * buffer before the current pbuf is read in completely.
6129          */

6131         hdr_rsize = vdev_psize_to_asize(vd, SPA_MINBLOCKSIZE);
6132         ASSERT(hdr_rsize <= psize);
6133         pinfo->pi_hdr_io = zio_root(vd->vdev_spa, NULL, NULL,
```



```
6134             ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
6135             ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY);
6136         hdr_io = zio_read_phys(pinfo->pi_hdr_io, vd, daddr, hdr_rsize, buf,
6137             ZIO_CHECKSUM_OFF, NULL, NULL, ZIO_PRIORITY_SYNC_READ,
6138             ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE |
6139             ZIO_FLAG_DONT_RETRY, B_FALSE);
6140         (void) zio_nowait(hdr_io);

6142         /*
6143          * Read in the rest of the pbuf - this can take longer than just
6144          * having a peek at the header.
6145          */
6146         pio = zio_root(vd->vdev_spa, NULL, pinfo, ZIO_FLAG_DONT_CACHE |
6147             ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE |
6148             ZIO_FLAG_DONT_RETRY);
6149         for (i = hdr_rsize; i < psize; ) {
6150                 uint64_t rsize = psize - i;
6151                 zio_t *rzio;

6153                 if (psize - i > SPA_MAXBLOCKSIZE)
6154                         rsize = SPA_MAXBLOCKSIZE;
6155                 ASSERT(rsize >= SPA_MINBLOCKSIZE);
6156                 rzio = zio_read_phys(pio, vd, daddr + i,
6157                     rsize, buf + i, ZIO_CHECKSUM_OFF, NULL, NULL,
6158                     ZIO_PRIORITY_ASYNC_READ, ZIO_FLAG_DONT_CACHE |
6159                     ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE |
6160                     ZIO_FLAG_DONT_RETRY, B_FALSE);
6161                 (void) zio_nowait(rzio);
6162                 i += rsize;
6163         }

6165         return (pio);
6166 }

6168 /*
6169  * Aborts a zio returned from l2arc_pbuf_prefetch and frees the data
6170  * buffers allocated for it.
6171  */
6172 static void
6173 l2arc_pbuf_prefetch_abort(zio_t *zio)
6174 {
6175         l2arc_prefetch_info_t *pi;

6177         pi = zio->io_private;
6178         ASSERT(pi != NULL);
6179         if (pi->pi_hdr_io != NULL)
6180                 (void) zio_wait(pi->pi_hdr_io);
6181         (void) zio_wait(zio);
6182         kmem_free(pi->pi_buf, pi->pi_buflen);
6183         pi->pi_buf = NULL;
6184         kmem_free(pi, sizeof (l2arc_prefetch_info_t));
6185 }

6187 /*
6188  * Encodes an l2uberblock_t structure into a destination buffer. This
6189  * buffer must be at least L2UBERBLOCK_SIZE bytes long. The resulting
6190  * uberblock is always of this constant size.
6191  */
6192 static void
6193 l2arc_uberblock_encode(const l2uberblock_t *ub, uint8_t *buf)
6194 {
6195         zio_cksum_t cksum;

6197         bzero(buf, L2UBERBLOCK_SIZE);

6199 #if defined(_BIG_ENDIAN)
```

```
6200         *(uint32_t *)buf = L2UBERBLOCK_MAGIC;
6201         *(uint16_t *)(buf + 6) = L2UB_BIG_ENDIAN;
6202 #else   /* !defined(_BIG_ENDIAN) */
6203         *(uint32_t *)buf = BSWAP_32(L2UBERBLOCK_MAGIC);
6204         /* zero flags is ok */
6205 #endif  /* !defined(_BIG_ENDIAN) */
6206         buf[4] = L2UBERBLOCK_MAX_VERSION;

6208         /* rest in native byte order */
6209         *(uint64_t *)(buf + 8) = ub->ub_spa_guid;
6210         *(uint64_t *)(buf + 16) = ub->ub_birth;
6211         *(uint64_t *)(buf + 24) = ub->ub_evict_tail;
6212         *(uint64_t *)(buf + 32) = ub->ub_alloc_space;
6213         *(uint64_t *)(buf + 40) = ub->ub_pbuf_daddr;
6214         *(uint32_t *)(buf + 48) = ub->ub_pbuf_asize;
6215         bcopy(&ub->ub_pbuf_cksum, buf + 52, 32);

6217         fletcher_4_native(buf, L2UBERBLOCK_SIZE - 32, &cksum);
6218         bcopy(&cksum, buf + L2UBERBLOCK_SIZE - 32, 32);
6219 }

6221 /*
6222  * Decodes an l2uberblock_t from an on-disk representation. Please note
6223  * that this function does not perform any uberblock validation and
6224  * checksumming - call l2arc_uberblock_verify() for that.
6225  */
6226 static void
6227 l2arc_uberblock_decode(const uint8_t *buf, l2uberblock_t *ub)
6228 {
6229         boolean_t bswap_needed;

6231         /* these always come in big endian */
6232 #if defined(_BIG_ENDIAN)
6233         ub->ub_magic = *(uint32_t *)buf;
6234         ub->ub_flags = *(uint16_t *)(buf + 6);
6235         bswap_needed = ((ub->ub_flags & L2UBLK_BIG_ENDIAN) != 1);
6236 #else   /* !defined(_BIG_ENDIAN) */
6237         ub->ub_magic = BSWAP_32(*(uint32_t *)buf);
6238         ub->ub_flags = BSWAP_16(*(uint16_t *)(buf + 6));
6239         bswap_needed = ((ub->ub_flags & L2UBLK_BIG_ENDIAN) != 0);
6240 #endif  /* !defined(_BIG_ENDIAN) */
6241         ub->ub_version = buf[4];

6243         ub->ub_spa_guid = *(uint64_t *)(buf + 8);
6244         ub->ub_birth = *(uint64_t *)(buf + 16);
6245         ub->ub_evict_tail = *(uint64_t *)(buf + 24);
6246         ub->ub_alloc_space = *(uint64_t *)(buf + 32);
6247         ub->ub_pbuf_daddr = *(uint64_t *)(buf + 40);
6248         ub->ub_pbuf_asize = *(uint32_t *)(buf + 48);
6249         bcopy(buf + 52, &ub->ub_pbuf_cksum, 36);
6250         bcopy(buf + L2UBERBLOCK_SIZE - 32, &ub->ub_cksum, 32);

6252         /* swap the rest if endianness doesn't match us */
6253         if (bswap_needed) {
6254                 ub->ub_spa_guid = BSWAP_64(ub->ub_spa_guid);
6255                 ub->ub_birth = BSWAP_64(ub->ub_birth);
6256                 ub->ub_evict_tail = BSWAP_64(ub->ub_evict_tail);
6257                 ub->ub_alloc_space = BSWAP_64(ub->ub_alloc_space);
6258                 ub->ub_pbuf_daddr = BSWAP_64(ub->ub_pbuf_daddr);
6259                 ub->ub_pbuf_asize = BSWAP_32(ub->ub_pbuf_asize);
6260                 ZIO_CHECKSUM_BSWAP(&ub->ub_pbuf_cksum);
6261                 ZIO_CHECKSUM_BSWAP(&ub->ub_cksum);
6262         }
6263 }

6265 /*
```



```
6266  * Verifies whether a decoded uberblock (via l2arc_uberblock_decode()) is
6267  * valid and matches its checksum.
6268  */
6269 static int
6270 l2arc_uberblock_verify(const uint8_t *buf, const l2uberblock_t *ub,
6271     uint64_t guid)
6272 {
6273         zio_cksum_t cksum;

6275         if (ub->ub_magic != L2UBERBLOCK_MAGIC ||
6276             ub->ub_version == 0 || ub->ub_version > L2UBERBLOCK_MAX_VERSION)
6277                 /*
6278                  * bad magic or invalid version => persistent l2arc not
6279                  * supported
6280                  */
6281                 return (ENOTSUP);

6283         if (ub->ub_spa_guid != guid)
6284                 /* this l2arc dev isn't ours */
6285                 return (EINVAL);

6287         fletcher_4_native(buf, L2UBERBLOCK_SIZE - 32, &cksum);
6288         if (!ZIO_CHECKSUM_EQUAL(cksum, ub->ub_cksum))
6289                 /* bad checksum, corrupt uberblock */
6290                 return (EINVAL);

6292         return (0);
6293 }

6295 /*
6296  * Schedules a zio to update the uberblock on an l2arc device. The zio is
6297  * initiated as a child of 'pio' and 'cb' is filled with the information
6298  * needed to free the uberblock data buffer after writing.
6299  */
6300 static void
6301 l2arc_uberblock_update(l2arc_dev_t *dev, zio_t *pio, l2arc_write_callback_t *cb)
6302 {
6303         uint8_t *ub_buf;
6304         l2uberblock_t ub;
6305         zio_t *wzio;
6306         vdev_stat_t st;

6308         ASSERT(cb->l2wcb_ub_buf == NULL);
6309         vdev_get_stats(dev->l2ad_vdev, &st);

6311         bzero(&ub, sizeof (ub));
6312         ub.ub_spa_guid = spa_guid(dev->l2ad_vdev->vdev_spa);
6313         ub.ub_birth = dev->l2ad_uberblock_birth++;
6314         ub.ub_evict_tail = dev->l2ad_evict;
6315         ub.ub_alloc_space = st.vs_alloc;
6316         ub.ub_pbuf_daddr = dev->l2ad_pbuf_daddr;
6317         ub.ub_pbuf_asize = dev->l2ad_pbuf_asize;
6318         ub.ub_pbuf_cksum = dev->l2ad_pbuf_cksum;
6319         if (dev->l2ad_first)
6320                 ub.ub_flags |= L2UBLK_EVICT_FIRST;

6322         ub_buf = kmem_alloc(L2UBERBLOCK_SIZE, KM_SLEEP);
6323         cb->l2wcb_ub_buf = ub_buf;
6324         l2arc_uberblock_encode(&ub, ub_buf);
6325         wzio = zio_write_phys(pio, dev->l2ad_vdev, VDEV_LABEL_START_SIZE,
6326             L2UBERBLOCK_SIZE, ub_buf, ZIO_CHECKSUM_OFF, NULL, NULL,
6327             ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL, B_FALSE);
6328         DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
6329             zio_t *, wzio);
6330         (void) zio_nowait(wzio);
6331 }
```

```
/*
 * Encodes a l2pbuf_t structure into the portable on-disk format. The
 * 'buf' buffer must be suitably sized to hold the entire uncompressed
 * structure (use L2PBUF_ENCODED_SIZE()). If requested, this function
 * also compresses the buffer.
 *
 * The return value is the length of the resulting encoded pbuf structure.
 * This can be either equal to L2PBUF_ENCODED_SIZE(pb) if no compression
 * was applied, or smaller if compression was applied. In either case,
 * prior to writing to disk, the caller must suitably pad the output
 * buffer so that it is aligned on a multiple of the underlying storage
 * system's block size.
 */
static uint32_t
l2arc_pbuf_encode(l2pbuf_t *pb, uint8_t *buf, uint32_t buflen)
{
        uint16_t flags = 0;
        uint8_t *dst_buf;
        uint32_t enclen;
        l2pbuf_buflist_t *buflist;

        enclen = L2PBUF_ENCODED_SIZE(pb);
        ASSERT(buflen >= enclen);
        bzero(buf, enclen);

        /* non-header portions of pbufs are in native byte order */
        *(uint64_t *)(buf + 8) = pb->pb_prev_daddr;
        *(uint32_t *)(buf + 16) = pb->pb_prev_asize;
        bcopy(&pb->pb_prev_cksum, buf + 20, 32);
        *(uint32_t *)(buf + 52) = enclen - L2PBUF_HDR_SIZE;

        /* first we encode the buflists uncompressed */
        dst_buf = buf + L2PBUF_HDR_SIZE;
        for (buflist = list_head(pb->pb_buflists_list); buflist;
            buflist = list_next(pb->pb_buflists_list, buflist)) {
                int i;

                ASSERT(buflist->l2pbl_nbufs != 0);
                for (i = 0; i < buflist->l2pbl_nbufs; i++) {
                        l2pbuf_buf_t *pbl_buf = &buflist->l2pbl_bufs[i];

                        ASSERT(pbl_buf->b_size != 0);
                        *(uint64_t *)dst_buf = pbl_buf->b_dva.dva_word[0];
                        *(uint64_t *)(dst_buf + 8) = pbl_buf->b_dva.dva_word[1];
                        *(uint64_t *)(dst_buf + 16) = pbl_buf->b_birth;
                        *(uint64_t *)(dst_buf + 24) = pbl_buf->b_cksum0;
                        bcopy(&pbl_buf->b_freeze_cksum, dst_buf + 32, 32);
                        *(uint32_t *)(dst_buf + 64) = pbl_buf->b_size;
                        *(uint64_t *)(dst_buf + 68) = pbl_buf->b_l2daddr;
                        *(uint32_t *)(dst_buf + 76) = pbl_buf->b_l2asize;
                        dst_buf[80] = pbl_buf->b_l2compress;
                        dst_buf[81] = pbl_buf->b_contents_type;
                        *(uint32_t *)(dst_buf + 84) = pbl_buf->b_flags;
                        dst_buf += L2PBUF_BUF_SIZE;
                }
        }
        ASSERT((uint32_t)(dst_buf - buf) == enclen);

        /* and then compress them if necessary */
        if (enclen >= l2arc_pbuf_compress_minsz) {
                uint8_t *cbuf;
                size_t slen, clen;

                slen = l2arc_pbuf_items_encoded_size(pb);
                cbuf = kmem_alloc(slen, KM_SLEEP);
```



```
                clen = lz4_compress(buf + L2PBUF_HDR_SIZE, cbuf, slen, slen, 0);
                ASSERT(clen != 0);
                if (clen < slen) {
                        bcopy(cbuf, buf + L2PBUF_HDR_SIZE, clen);
                        flags |= L2PBUF_COMPRESSED;
                        /* zero out the rest of the input buffer */
                        bzero(buf + L2PBUF_HDR_SIZE + clen,
                            buflen - (L2PBUF_HDR_SIZE + clen));
                        /* adjust our buffer length now that it's shortened */
                        enclen = L2PBUF_HDR_SIZE + clen;
                }
                kmem_free(cbuf, slen);
        }

        /* the header goes last since 'flags' may change due to compression */
#if defined(_BIG_ENDIAN)
        *(uint32_t *)buf = L2PBUF_MAGIC;
        flags |= L2PBUF_BIG_ENDIAN;
        *(uint16_t *)(buf + 6) = flags;
#else   /* !defined(_BIG_ENDIAN) */
        *(uint32_t *)buf = BSWAP_32(L2PBUF_MAGIC);
        *(uint16_t *)(buf + 6) = BSWAP_16(flags);
#endif  /* !defined(_BIG_ENDIAN) */
        buf[4] = L2PBUF_MAX_VERSION;

        return (enclen);
}

/*
 * Decodes a stored l2pbuf_t structure previously encoded using
 * l2arc_pbuf_encode. The source buffer is not modified. The passed pbuf
 * must be initialized by l2arc_pbuf_init by the caller beforehand, but
 * must not have been used to store any buffers yet.
 *
 * Please note that we don't do checksum verification here, as we don't
 * know our own checksum (that's know by the previous block in the linked
 * list, or by the uberblock). This should be performed by the caller
 * prior to calling l2arc_pbuf_decode.
 */
static int
l2arc_pbuf_decode(uint8_t *input_buf, uint32_t buflen, l2pbuf_t *pb)
{
        boolean_t bswap_needed;
        uint32_t payload_sz, payload_asz;
        uint8_t *src_bufs;
        l2pbuf_buflist_t *buflist;
        int i, nbufs;

        ASSERT(input_buf != NULL);
        ASSERT(pb != NULL);
        ASSERT(pb->pb_version != 0);
        ASSERT(pb->pb_nbuflists == 0);

        /* no valid buffer can be this small */
        if (buflen < L2PBUF_HDR_SIZE)
                return (EINVAL);

        /* these always come in big endian */
#if defined(_BIG_ENDIAN)
        pb->pb_magic = *(uint32_t *)input_buf;
        pb->pb_flags = *(uint16_t *)(input_buf + 6);
        bswap_needed = ((pb->pb_flags & L2PBUF_BIG_ENDIAN) != 1);
#else   /* !defined(_BIG_ENDIAN) */
        pb->pb_magic = BSWAP_32(*(uint32_t *)input_buf);
        pb->pb_flags = BSWAP_16(*(uint16_t *)(input_buf + 6));
        bswap_needed = ((pb->pb_flags & L2PBUF_BIG_ENDIAN) != 0);
```

```
#endif  /* !defined(_BIG_ENDIAN) */
        pb->pb_version = input_buf[4];

        if (pb->pb_magic != L2PBUF_MAGIC || pb->pb_version == 0)
                return (EINVAL);
        if (pb->pb_version > L2PBUF_MAX_VERSION)
                return (ENOTSUP);

        /* remainder of pbuf may need bswap'ping */
        pb->pb_prev_daddr = *(uint64_t *)(input_buf + 8);
        pb->pb_prev_asize = *(uint64_t *)(input_buf + 16);
        bcopy(input_buf + 20, &pb->pb_prev_cksum, 32);
        payload_sz = *(uint32_t *)(input_buf + 52);
        payload_asz = buflen - L2PBUF_HDR_SIZE;

        if (bswap_needed) {
                pb->pb_prev_daddr = BSWAP_64(pb->pb_prev_daddr);
                pb->pb_prev_asize = BSWAP_64(pb->pb_prev_asize);
                ZIO_CHECKSUM_BSWAP(&pb->pb_prev_cksum);
                payload_sz = BSWAP_32(payload_sz);
        }

        /* check for sensible buffer allocation limits */
        if (((pb->pb_flags & L2PBUF_COMPRESSED) && payload_sz <= payload_asz) ||
            (payload_sz > L2PBUF_MAX_PAYLOAD_SIZE) ||
            (payload_sz % L2PBUF_BUF_SIZE) != 0 || payload_sz == 0)
                return (EINVAL);
        nbufs = payload_sz / L2PBUF_BUF_SIZE;

        /* decompression might be needed */
        if (pb->pb_flags & L2PBUF_COMPRESSED) {
                src_bufs = kmem_alloc(payload_sz, KM_SLEEP);
                if (lz4_decompress(input_buf + L2PBUF_HDR_SIZE, src_bufs,
                    payload_asz, payload_sz, 0) != 0) {
                        kmem_free(src_bufs, payload_sz);
                        return (EINVAL);
                }
        } else {
                src_bufs = input_buf + L2PBUF_HDR_SIZE;
        }

        /* Decode individual pbuf items from our source buffer. */
        buflist = l2arc_pbuf_buflist_alloc(pb, nbufs);
        for (i = 0; i < nbufs; i++) {
                l2pbuf_buf_t *pbl_buf = &buflist->l2pbl_bufs[i];
                const uint8_t *src = src_bufs + i * L2PBUF_BUF_SIZE;

                pbl_buf->b_dva.dva_word[0] = *(uint64_t *)src;
                pbl_buf->b_dva.dva_word[1] = *(uint64_t *)(src + 8);
                pbl_buf->b_birth = *(uint64_t *)(src + 16);
                pbl_buf->b_cksum0 = *(uint64_t *)(src + 24);
                bcopy(src + 32, &pbl_buf->b_freeze_cksum, 32);
                pbl_buf->b_size = *(uint32_t *)(src + 64);
                pbl_buf->b_l2daddr = *(uint64_t *)(src + 68);
                pbl_buf->b_l2asize = *(uint32_t *)(src + 76);
                pbl_buf->b_l2compress = src[80];
                pbl_buf->b_contents_type = src[81];
                pbl_buf->b_flags = *(uint32_t *)(src + 84);

                if (bswap_needed) {
                        pbl_buf->b_dva.dva_word[0] =
                            BSWAP_64(pbl_buf->b_dva.dva_word[0]);
                        pbl_buf->b_dva.dva_word[1] =
                            BSWAP_64(pbl_buf->b_dva.dva_word[1]);
                        pbl_buf->b_birth = BSWAP_64(pbl_buf->b_birth);
                        pbl_buf->b_cksum0 = BSWAP_64(pbl_buf->b_cksum0);
```



```
                        ZIO_CHECKSUM_BSWAP(&pbl_buf->b_freeze_cksum);
                        pbl_buf->b_size = BSWAP_32(pbl_buf->b_size);
                        pbl_buf->b_l2daddr = BSWAP_64(pbl_buf->b_l2daddr);
                        pbl_buf->b_l2asize = BSWAP_32(pbl_buf->b_l2asize);
                        pbl_buf->b_flags = BSWAP_32(pbl_buf->b_flags);
                }

                pb->pb_payload_asz += pbl_buf->b_l2asize;
        }

        if (pb->pb_flags & L2PBUF_COMPRESSED)
                kmem_free(src_bufs, payload_sz);

        return (0);
}

/*
 * Decodes the previous buffer pointer encoded in a pbuf. This is used
 * during L2ARC reconstruction to "peek" at the next buffer and start
 * issuing IO to fetch it early, before decoding of the current buffer
 * is done (which can take time due to decompression).
 * Returns 0 on success (and fills in the return parameters 'daddr',
 * 'asize' and 'cksum' with the info of the previous pbuf), and an errno
 * on error.
 */
static int
l2arc_pbuf_decode_prev_ptr(const uint8_t *buf, size_t buflen, uint64_t *daddr,
    uint32_t *asize, zio_cksum_t *cksum)
{
        boolean_t bswap_needed;
        uint16_t version, flags;
        uint32_t magic;

        ASSERT(buf != NULL);

        /* no valid buffer can be this small */
        if (buflen <= L2PBUF_HDR_SIZE)
                return (EINVAL);

        /* these always come in big endian */
#if defined(_BIG_ENDIAN)
        magic = *(uint32_t *)buf;
        flags = *(uint16_t *)(buf + 6);
        bswap_needed = ((flags & L2PBUF_BIG_ENDIAN) != 1);
#else   /* !defined(_BIG_ENDIAN) */
        magic = BSWAP_32(*(uint32_t *)buf);
        flags = BSWAP_16(*(uint16_t *)(buf + 6));
        bswap_needed = ((flags & L2PBUF_BIG_ENDIAN) != 0);
#endif  /* !defined(_BIG_ENDIAN) */
        version = buf[4];

        if (magic != L2PBUF_MAGIC || version == 0)
                return (EINVAL);
        if (version > L2PBUF_MAX_VERSION)
                return (ENOTSUP);

        *daddr = *(uint64_t *)(buf + 4);
        *asize = *(uint64_t *)(buf + 12);
        bcopy(buf + 16, cksum, 32);

        if (bswap_needed) {
                *daddr = BSWAP_64(*daddr);
                *asize = BSWAP_64(*asize);
                ZIO_CHECKSUM_BSWAP(cksum);
        }
```

```
6596         return (0);
6597 }

6599 /*
6600  * Initializes a pbuf structure into a clean state. All version and flags
6601  * fields are filled in as appropriate for this architecture.
6602  * If the structure was used before, first call l2arc_pbuf_destroy on it,
6603  * as this function assumes the structure is uninitialized.
6604  */
6605 static void
6606 l2arc_pbuf_init(l2pbuf_t *pb)
6607 {
6608         bzero(pb, sizeof (l2pbuf_t));
6609         pb->pb_version = L2PBUF_MAX_VERSION;
6610 #if defined(_BIG_ENDIAN)
6611         pb->pb_flags |= L2PB_BIG_ENDIAN;
6612 #endif
6613         pb->pb_buflists_list = kmem_zalloc(sizeof (list_t), KM_SLEEP);
6614         list_create(pb->pb_buflists_list, sizeof (l2pbuf_buflist_t),
6615             offsetof(l2pbuf_buflist_t, l2pbl_node));
6616 }

6618 /*
6619  * Destroys a pbuf structure and puts it into a clean state ready to be
6620  * initialized by l2arc_pbuf_init. All buflists created by
6621  * l2arc_pbuf_buflist_alloc are released as well.
6622  */
6623 static void
6624 l2arc_pbuf_destroy(l2pbuf_t *pb)
6625 {
6626         list_t *buflist_list = pb->pb_buflists_list;
6627         l2pbuf_buflist_t *buflist;

6629         while ((buflist = list_head(buflist_list)) != NULL) {
6630                 ASSERT(buflist->l2pbl_nbufs > 0);
6631                 kmem_free(buflist->l2pbl_bufs, sizeof (l2pbuf_buf_t) *
6632                     buflist->l2pbl_nbufs);
6633                 list_remove(buflist_list, buflist);
6634                 kmem_free(buflist, sizeof (l2pbuf_buflist_t));
6635         }
6636         pb->pb_nbuflists = 0;
6637         list_destroy(pb->pb_buflists_list);
6638         kmem_free(pb->pb_buflists_list, sizeof (list_t));
6639         bzero(pb, sizeof (l2pbuf_t));
6640 }

6642 /*
6643  * Allocates a new buflist inside of a pbuf, which can hold up to 'nbufs'
6644  * buffers. This is used during the buffer write cycle - each cycle allocates
6645  * a new buflist and fills it with buffers it writes. Then, when the pbuf
6646  * reaches its buflist limit, it is commited to stable storage.
6647  */
6648 static l2pbuf_buflist_t *
6649 l2arc_pbuf_buflist_alloc(l2pbuf_t *pb, int nbufs)
6650 {
6651         l2pbuf_buflist_t *buflist;

6653         ASSERT(pb->pb_buflists_list != NULL);
6654         buflist = kmem_zalloc(sizeof (l2pbuf_buflist_t), KM_SLEEP);
6655         buflist->l2pbl_nbufs = nbufs;
6656         buflist->l2pbl_bufs = kmem_zalloc(sizeof (l2pbuf_buf_t) * nbufs,
6657             KM_SLEEP);
6658         list_insert_tail(pb->pb_buflists_list, buflist);
6659         pb->pb_nbuflists++;

6661         return (buflist);
```



```
6662 }

6664 /*
6665  * Inserts ARC buffer 'ab' into the pbuf 'pb' buflist 'pbl' at index 'idx'.
6666  * The buffer being inserted must be present in L2ARC.
6667  */
6668 static void
6669 l2arc_pbuflist_insert(l2pbuf_t *pb, l2pbuf_buflist_t *pbl,
6670     const arc_buf_hdr_t *ab, int index)
6671 {
6672         l2pbuf_buf_t *pb_buf;
6673         const l2arc_buf_hdr_t *l2hdr;

6675         l2hdr = ab->b_l2hdr;
6676         ASSERT(l2hdr != NULL);
6677         ASSERT(pbl->l2pbl_nbufs > index);

6679         pb_buf = &pbl->l2pbl_bufs[index];
6680         pb_buf->b_dva = ab->b_dva;
6681         pb_buf->b_birth = ab->b_birth;
6682         pb_buf->b_cksum0 = ab->b_cksum0;
6683         pb_buf->b_freeze_cksum = *ab->b_freeze_cksum;
6684         pb_buf->b_size = ab->b_size;
6685         pb_buf->b_l2daddr = l2hdr->b_daddr;
6686         pb_buf->b_l2asize = l2hdr->b_asize;
6687         pb_buf->b_l2compress = l2hdr->b_compress;
6688         pb_buf->b_contents_type = ab->b_type;
6689         pb_buf->b_flags = ab->b_flags & L2ARC_PERSIST_FLAGS;
6690         pb->pb_payload_asz += l2hdr->b_asize;
6691 }

6693 /*
6694  * Commits a pbuf to stable storage. This routine is invoked when writing
6695  * ARC buffers to an L2ARC device. When the pbuf associated with the device
6696  * has reached its limits (either in size or in number of writes), it is
6697  * scheduled here for writing.
6698  * This function allocates some memory to temporarily hold the serialized
6699  * buffer to be written. This is then released in l2arc_write_done.
6700  */
6701 static void
6702 l2arc_pbuf_commit(l2arc_dev_t *dev, zio_t *pio, l2arc_write_callback_t *cb)
6703 {
6704         l2pbuf_t *pb = &dev->l2ad_pbuf;
6705         uint64_t i, est_encsize, bufsize, encsize, io_size;
6706         uint8_t *pb_buf;

6708         pb->pb_prev_daddr = dev->l2ad_pbuf_daddr;
6709         pb->pb_prev_asize = dev->l2ad_pbuf_asize;
6710         pb->pb_prev_cksum = dev->l2ad_pbuf_cksum;

6712         est_encsize = L2PBUF_ENCODED_SIZE(pb);
6713         bufsize = vdev_psize_to_asize(dev->l2ad_vdev, est_encsize);
6714         pb_buf = kmem_zalloc(bufsize, KM_SLEEP);
6715         encsize = l2arc_pbuf_encode(pb, pb_buf, bufsize);
6716         cb->l2wcb_pbuf = pb_buf;
6717         cb->l2wcb_pbuf_size = bufsize;

6719         dev->l2ad_pbuf_daddr = dev->l2ad_hand;
6720         dev->l2ad_pbuf_asize = encsize;
6721         fletcher_4_native(pb_buf, encsize, &dev->l2ad_pbuf_cksum);

6723         io_size = vdev_psize_to_asize(dev->l2ad_vdev, encsize);
6724         for (i = 0; i < io_size; ) {
6725                 zio_t *wzio;
6726                 uint64_t wsize = io_size - i;
```

```
6728                 if (wsize > SPA_MAXBLOCKSIZE)
6729                         wsize = SPA_MAXBLOCKSIZE;
6730                 ASSERT(wsize >= SPA_MINBLOCKSIZE);
6731                 wzio = zio_write_phys(pio, dev->l2ad_vdev, dev->l2ad_hand + i,
6732                     wsize, pb_buf + i, ZIO_CHECKSUM_OFF, NULL, NULL,
6733                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_CANFAIL, B_FALSE);
6734                 DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
6735                     zio_t *, wzio);
6736                 (void) zio_nowait(wzio);
6737                 i += wsize;
6738         }

6740         dev->l2ad_hand += io_size;
6741         vdev_space_update(dev->l2ad_vdev, io_size, 0, 0);
6742         l2arc_uberblock_update(dev, pio, cb);

6744         ARCSTAT_INCR(arcstat_l2_write_bytes, io_size);
6745         ARCSTAT_BUMP(arcstat_l2_meta_writes);
6746         ARCSTAT_F_AVG(arcstat_l2_meta_avg_size, est_encsize);
6747         ARCSTAT_F_AVG(arcstat_l2_meta_avg_asize, encsize);
6748         ARCSTAT_F_AVG(arcstat_l2_asize_to_meta_ratio,
6749             pb->pb_payload_asz / encsize);
6750 }

6752 /*
6753  * Returns the number of bytes occupied by the payload buffer items of
6754  * a pbuf in portable (on-disk) encoded form, i.e. the bytes following
6755  * L2PBUF_HDR_SIZE.
6756  */
6757 static uint32_t
6758 l2arc_pbuf_items_encoded_size(l2pbuf_t *pb)
6759 {
6760         uint32_t size = 0;
6761         l2pbuf_buflist_t *buflist;

6763         for (buflist = list_head(pb->pb_buflists_list); buflist != NULL;
6764             buflist = list_next(pb->pb_buflists_list, buflist))
6765                 size += L2PBUF_BUF_SIZE * buflist->l2pbl_nbufs;

6767         return (size);
6768 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  175566 Mon Aug  5 21:59:08 2013
new/usr/src/uts/common/fs/zfs/spa.c
3525 Persistent L2ARC
**********************************************************
______unchanged_portion_omitted_

1403 /*
1404  * Load (or re-load) the current list of vdevs describing the active l2cache for
1405  * this pool.  When this is called, we have some form of basic information in
1406  * 'spa_l2cache.sav_config'.  We parse this into vdevs, try to open them, and
1407  * then re-generate a more complete list including status information.
1408  * Devices which are already active have their details maintained, and are
1409  * not re-opened.
1410  */
1411 static void
1412 spa_load_l2cache(spa_t *spa)
1413 {
1414         nvlist_t **l2cache;
1415         uint_t nl2cache;
1416         int i, j, oldnvdevs;
1417         uint64_t guid;
1418         vdev_t *vd, **oldvdevs, **newvdevs;
1419         spa_aux_vdev_t *sav = &spa->spa_l2cache;

1421         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

1423         if (sav->sav_config != NULL) {
1424                 VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
1425                     ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
1426                 newvdevs = kmem_alloc(nl2cache * sizeof (void *), KM_SLEEP);
1427         } else {
1428                 nl2cache = 0;
1429                 newvdevs = NULL;
1430         }

1432         oldvdevs = sav->sav_vdevs;
1433         oldnvdevs = sav->sav_count;
1434         sav->sav_vdevs = NULL;
1435         sav->sav_count = 0;

1437         /*
1438          * Process new nvlist of vdevs.
1439          */
1440         for (i = 0; i < nl2cache; i++) {
1441                 VERIFY(nvlist_lookup_uint64(l2cache[i], ZPOOL_CONFIG_GUID,
1442                     &guid) == 0);

1444                 newvdevs[i] = NULL;
1445                 for (j = 0; j < oldnvdevs; j++) {
1446                         vd = oldvdevs[j];
1447                         if (vd != NULL && guid == vd->vdev_guid) {
1448                                 /*
1449                                  * Retain previous vdev for add/remove ops.
1450                                  */
1451                                 newvdevs[i] = vd;
1452                                 oldvdevs[j] = NULL;
1453                                 break;
1454                         }
1455                 }

1457                 if (newvdevs[i] == NULL) {
1458                         /*
1459                          * Create new vdev
1460                          */
1461                         VERIFY(spa_config_parse(spa, &vd, l2cache[i], NULL, 0,
```



```
1462                             VDEV_ALLOC_L2CACHE) == 0);
1463                         ASSERT(vd != NULL);
1464                         newvdevs[i] = vd;

1466                         /*
1467                          * Commit this vdev as an l2cache device,
1468                          * even if it fails to open.
1469                          */
1470                         spa_l2cache_add(vd);

1472                         vd->vdev_top = vd;
1473                         vd->vdev_aux = sav;

1475                         spa_l2cache_activate(vd);

1477                         if (vdev_open(vd) != 0)
1478                                 continue;

1480                         (void) vdev_validate_aux(vd);

1482                         if (!vdev_is_dead(vd)) {
1483                                 boolean_t persist = B_FALSE;

1485                                 if (spa->spa_load_state != SPA_LOAD_TRYIMPORT) {
1486                                         /*
1487                                          * Only allow to do the L2ARC rebuild
1488                                          * when not doing a spa try-load.
1489                                          */
1490                                         (void) nvlist_lookup_boolean_value(
1491                                             l2cache[i],
1492                                             ZPOOL_CONFIG_L2CACHE_PERSISTENT,
1493                                             &persist);
1482                         if (!vdev_is_dead(vd))
1483                                 l2arc_add_vdev(spa, vd);
1494                                 }
1495                                 l2arc_add_vdev(spa, vd, persist);
1496                         }
1497                 }
1498         }

1500         /*
1501          * Purge vdevs that were dropped
1502          */
1503         for (i = 0; i < oldnvdevs; i++) {
1504                 uint64_t pool;

1506                 vd = oldvdevs[i];
1507                 if (vd != NULL) {
1508                         ASSERT(vd->vdev_isl2cache);

1510                         if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
1511                             pool != 0ULL && l2arc_vdev_present(vd))
1512                                 l2arc_remove_vdev(vd);
1513                         vdev_clear_stats(vd);
1514                         vdev_free(vd);
1515                 }
1516         }

1518         if (oldvdevs)
1519                 kmem_free(oldvdevs, oldnvdevs * sizeof (void *));

1521         if (sav->sav_config == NULL)
1522                 goto out;

1524         sav->sav_vdevs = newvdevs;
1525         sav->sav_count = (int)nl2cache;
```

```
1527         /*
1528          * Recompute the stashed list of l2cache devices, with status
1529          * information this time.
1530          */
1531         VERIFY(nvlist_remove(sav->sav_config, ZPOOL_CONFIG_L2CACHE,
1532             DATA_TYPE_NVLIST_ARRAY) == 0);

1534         l2cache = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
1535         for (i = 0; i < sav->sav_count; i++)
1536                 l2cache[i] = vdev_config_generate(spa,
1537                     sav->sav_vdevs[i], B_TRUE, VDEV_CONFIG_L2CACHE);
1538         VERIFY(nvlist_add_nvlist_array(sav->sav_config,
1539             ZPOOL_CONFIG_L2CACHE, l2cache, sav->sav_count) == 0);
1540 out:
1541         for (i = 0; i < sav->sav_count; i++)
1542                 nvlist_free(l2cache[i]);
1543         if (sav->sav_count)
1544                 kmem_free(l2cache, sav->sav_count * sizeof (void *));
1545 }
_____unchanged_portion_omitted_

3801 #endif

3803 /*
3804  * Import a non-root pool into the system.
3805  */
3806 int
3807 spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)
3808 {
3809         spa_t *spa;
3810         char *altroot = NULL;
3811         spa_load_state_t state = SPA_LOAD_IMPORT;
3812         zpool_rewind_policy_t policy;
3813         uint64_t mode = spa_mode_global;
3814         uint64_t readonly = B_FALSE;
3815         int error;
3816         nvlist_t *nvroot;
3817         nvlist_t **spares, **l2cache;
3818         uint_t nspares, nl2cache;

3820         /*
3821          * If a pool with this name exists, return failure.
3822          */
3823         mutex_enter(&spa_namespace_lock);
3824         if (spa_lookup(pool) != NULL) {
3825                 mutex_exit(&spa_namespace_lock);
3826                 return (SET_ERROR(EEXIST));
3827         }

3829         /*
3830          * Create and initialize the spa structure.
3831          */
3832         (void) nvlist_lookup_string(props,
3833             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3834         (void) nvlist_lookup_uint64(props,
3835             zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);
3836         if (readonly)
3837                 mode = FREAD;
3838         spa = spa_add(pool, config, altroot);
3839         spa->spa_import_flags = flags;

3841         /*
3842          * Verbatim import - Take a pool and insert it into the namespace
3843          * as if it had been loaded at boot.
3844          */
```



```
3845         if (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) {
3846                 if (props != NULL)
3847                         spa_configfile_set(spa, props, B_FALSE);

3849                 spa_config_sync(spa, B_FALSE, B_TRUE);

3851                 mutex_exit(&spa_namespace_lock);
3852                 spa_history_log_version(spa, "import");

3854                 return (0);
3855         }

3857         spa_activate(spa, mode);

3859         /*
3860          * Don't start async tasks until we know everything is healthy.
3861          */
3862         spa_async_suspend(spa);

3864         zpool_get_rewind_policy(config, &policy);
3865         if (policy.zrp_request & ZPOOL_DO_REWIND)
3866                 state = SPA_LOAD_RECOVER;

3868         /*
3869          * Pass off the heavy lifting to spa_load().  Pass TRUE for mosconfig
3870          * because the user-supplied config is actually the one to trust when
3871          * doing an import.
3872          */
3873         if (state != SPA_LOAD_RECOVER)
3874                 spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

3876         error = spa_load_best(spa, state, B_TRUE, policy.zrp_txg,
3877             policy.zrp_request);

3879         /*
3880          * Propagate anything learned while loading the pool and pass it
3881          * back to caller (i.e. rewind info, missing devices, etc).
3882          */
3883         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
3884             spa->spa_load_info) == 0);

3886         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3887         /*
3888          * Toss any existing sparelist, as it doesn't have any validity
3889          * anymore, and conflicts with spa_has_spare().
3890          */
3891         if (spa->spa_spares.sav_config) {
3892                 nvlist_free(spa->spa_spares.sav_config);
3893                 spa->spa_spares.sav_config = NULL;
3894                 spa_load_spares(spa);
3895         }
3883         if (spa->spa_l2cache.sav_config) {
3884                 nvlist_free(spa->spa_l2cache.sav_config);
3885                 spa->spa_l2cache.sav_config = NULL;
3886                 spa_load_l2cache(spa);
3887         }

3897         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3898             &nvroot) == 0);
3899         if (error == 0)
3900                 error = spa_validate_aux(spa, nvroot, -1ULL,
3901                     VDEV_ALLOC_SPARE);
3902         if (error == 0)
3903                 error = spa_validate_aux(spa, nvroot, -1ULL,
3904                     VDEV_ALLOC_L2CACHE);
3905         spa_config_exit(spa, SCL_ALL, FTAG);
```

```
3907         if (props != NULL)
3908                 spa_configfile_set(spa, props, B_FALSE);

3910         if (error != 0 || (props && spa_writeable(spa) &&
3911             (error = spa_prop_set(spa, props)))) {
3912                 spa_unload(spa);
3913                 spa_deactivate(spa);
3914                 spa_remove(spa);
3915                 mutex_exit(&spa_namespace_lock);
3916                 return (error);
3917         }

3919         spa_async_resume(spa);

3921         /*
3922          * Override any spares and level 2 cache devices as specified by
3923          * the user, as these may have correct device names/devids, etc.
3924          */
3925         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3926             &spares, &nspares) == 0) {
3927                 if (spa->spa_spares.sav_config)
3928                         VERIFY(nvlist_remove(spa->spa_spares.sav_config,
3929                             ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
3930                 else
3931                         VERIFY(nvlist_alloc(&spa->spa_spares.sav_config,
3932                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3933                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3934                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3935                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3936                 spa_load_spares(spa);
3937                 spa_config_exit(spa, SCL_ALL, FTAG);
3938                 spa->spa_spares.sav_sync = B_TRUE;
3939         }
3940         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3941             &l2cache, &nl2cache) == 0) {
3942                 if (spa->spa_l2cache.sav_config)
3943                         VERIFY(nvlist_remove(spa->spa_l2cache.sav_config,
3944                             ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
3945                 else
3946                         VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3947                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3948                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3949                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3950                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3951                 spa_load_l2cache(spa);
3952                 spa_config_exit(spa, SCL_ALL, FTAG);
3953                 spa->spa_l2cache.sav_sync = B_TRUE;
3954         }

3956         /*
3957          * Check for any removed devices.
3958          */
3959         if (spa->spa_autoreplace) {
3960                 spa_aux_check_removed(&spa->spa_spares);
3961                 spa_aux_check_removed(&spa->spa_l2cache);
3962         }

3964         if (spa_writeable(spa)) {
3965                 /*
3966                  * Update the config cache to include the newly-imported pool.
3967                  */
3968                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
3969         }

3971         /*
```



```
3972          * It's possible that the pool was expanded while it was exported.
3973          * We kick off an async task to handle this for us.
3974          */
3975         spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);

3977         mutex_exit(&spa_namespace_lock);
3978         spa_history_log_version(spa, "import");

3980         return (0);
3981 }
```

_____***unchanged_portion_omitted_***

****
4516 Mon Aug 5 21:59:09 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, int 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 *done,
 112     void *private, int priority, int zio_flags, const zbookmark_t *zb);
 113 void arc_freed(spa_t *spa, const blkptr_t *bp);

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

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

 122 void arc_init(void);
 123 void arc_fini(void);

 125 /*
 126  * Level 2 ARC
 127  */

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

 137 #ifndef _KERNEL
 138 extern boolean_t arc_watch;
 139 extern int arc_procfd;
 140 #endif
```



```
 142 #ifdef  __cplusplus
 143 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   25969 Mon Aug  5 21:59:09 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_*

```
**********************************************************
   86544 Mon Aug  5 21:59:09 2013
new/usr/src/uts/common/fs/zfs/vdev.c
3525 Persistent L2ARC
**********************************************************
______unchanged_portion_omitted_

1487 /*
1488  * Reopen all interior vdevs and any unopened leaves.  We don't actually
1489  * reopen leaf vdevs which had previously been opened as they might deadlock
1490  * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
1491  * If the leaf has never been opened then open it, as usual.
1492  */
1493 void
1494 vdev_reopen(vdev_t *vd)
1495 {
1496         spa_t *spa = vd->vdev_spa;

1498         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1500         /* set the reopening flag unless we're taking the vdev offline */
1501         vd->vdev_reopening = !vd->vdev_offline;
1502         vdev_close(vd);
1503         (void) vdev_open(vd);

1505         /*
1506          * Call vdev_validate() here to make sure we have the same device.
1507          * Otherwise, a device with an invalid label could be successfully
1508          * opened in response to vdev_reopen().
1509          */
1510         if (vd->vdev_aux) {
1511                 (void) vdev_validate_aux(vd);
1512                 if (vdev_readable(vd) && vdev_writeable(vd) &&
1513                     vd->vdev_aux == &spa->spa_l2cache &&
1514                     !l2arc_vdev_present(vd)) {
1515                         /*
1516                          * When reopening we can assume persistent L2ARC is
1517                          * supported, since we've already opened the device
1518                          * in the past and prepended an L2 uberblock.
1519                          */
1520                         l2arc_add_vdev(spa, vd, B_TRUE);
1521                 }
1514                     !l2arc_vdev_present(vd))
1515                         l2arc_add_vdev(spa, vd);
1522         } else {
1523                 (void) vdev_validate(vd, B_TRUE);
1524         }

1526         /*
1527          * Reassess parent vdev's health.
1528          */
1529         vdev_propagate_state(vd);
1530 }
______unchanged_portion_omitted_
```

```
**********************************************************
   38163 Mon Aug  5 21:59:10 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         VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);

 221         VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_TYPE,
 222             vd->vdev_ops->vdev_op_type) == 0);
 223         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)))
 224                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ID, vd->vdev_id)
 225                     == 0);
 226         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_GUID, vd->vdev_guid) == 0);

 228         if (vd->vdev_path != NULL)
 229                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_PATH,
 230                     vd->vdev_path) == 0);

 232         if (vd->vdev_devid != NULL)
 233                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_DEVID,
 234                     vd->vdev_devid) == 0);

 236         if (vd->vdev_physpath != NULL)
 237                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_PHYS_PATH,
 238                     vd->vdev_physpath) == 0);

 240         if (vd->vdev_fru != NULL)
 241                 VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_FRU,
 242                     vd->vdev_fru) == 0);

 244         if (vd->vdev_nparity != 0) {
 245                 ASSERT(strcmp(vd->vdev_ops->vdev_op_type,
 246                     VDEV_TYPE_RAIDZ) == 0);

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

 258                 /*
 259                  * Note that we'll add the nparity tag even on storage pools
 260                  * that only support a single parity device -- older software
 261                  * will just ignore it.
 262                  */
 263                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_NPARITY,
 264                     vd->vdev_nparity) == 0);
 265         }

 267         if (vd->vdev_wholedisk != -1ULL)
 268                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
```



```
 269                     vd->vdev_wholedisk) == 0);

 271         if (vd->vdev_not_present)
 272                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT, 1) == 0);

 274         if (vd->vdev_isspare)
 275                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_SPARE, 1) == 0);

 277         if (!(flags & (VDEV_CONFIG_SPARE | VDEV_CONFIG_L2CACHE)) &&
 278             vd == vd->vdev_top) {
 279                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
 280                     vd->vdev_ms_array) == 0);
 281                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
 282                     vd->vdev_ms_shift) == 0);
 283                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ASHIFT,
 284                     vd->vdev_ashift) == 0);
 285                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ASIZE,
 286                     vd->vdev_asize) == 0);
 287                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_LOG,
 288                     vd->vdev_islog) == 0);
 289                 if (vd->vdev_removing)
 290                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVING,
 291                             vd->vdev_removing) == 0);
 292         }

 294         if (flags & VDEV_CONFIG_L2CACHE)
 295                 /* indicate that we support L2ARC persistency */
 296                 VERIFY(nvlist_add_boolean_value(nv,
 297                     ZPOOL_CONFIG_L2CACHE_PERSISTENT, B_TRUE) == 0);

 299         if (vd->vdev_dtl_smo.smo_object != 0)
 300                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_DTL,
 301                     vd->vdev_dtl_smo.smo_object) == 0);

 303         if (vd->vdev_crtxg)
 304                 VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
 305                     vd->vdev_crtxg) == 0);

 307         if (getstats) {
 308                 vdev_stat_t vs;
 309                 pool_scan_stat_t ps;

 311                 vdev_get_stats(vd, &vs);
 312                 VERIFY(nvlist_add_uint64_array(nv, ZPOOL_CONFIG_VDEV_STATS,
 313                     (uint64_t *)&vs, sizeof (vs) / sizeof (uint64_t)) == 0);

 315                 /* provide either current or previous scan information */
 316                 if (spa_scan_get_stats(spa, &ps) == 0) {
 317                         VERIFY(nvlist_add_uint64_array(nv,
 318                             ZPOOL_CONFIG_SCAN_STATS, (uint64_t *)&ps,
 319                             sizeof (pool_scan_stat_t) / sizeof (uint64_t))
 320                             == 0);
 321                 }
 322         }

 324         if (!vd->vdev_ops->vdev_op_leaf) {
 325                 nvlist_t **child;
 326                 int c, idx;

 328                 ASSERT(!vd->vdev_ishole);

 330                 child = kmem_alloc(vd->vdev_children * sizeof (nvlist_t *),
 331                     KM_SLEEP);

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

```
336                         /*
337                          * If we're generating an nvlist of removing
338                          * vdevs then skip over any device which is
339                          * not being removed.
340                          */
341                         if ((flags & VDEV_CONFIG_REMOVING) &&
342                             !cvd->vdev_removing)
343                                 continue;

345                         child[idx++] = vdev_config_generate(spa, cvd,
346                             getstats, flags);
347                 }

349                 if (idx) {
350                         VERIFY(nvlist_add_nvlist_array(nv,
351                             ZPOOL_CONFIG_CHILDREN, child, idx) == 0);
352                 }

354                 for (c = 0; c < idx; c++)
355                         nvlist_free(child[c]);

357                 kmem_free(child, vd->vdev_children * sizeof (nvlist_t *));

359         } else {
360                 const char *aux = NULL;

362                 if (vd->vdev_offline && !vd->vdev_tmpoffline)
363                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_OFFLINE,
364                             B_TRUE) == 0);
365                 if (vd->vdev_resilvering)
366                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_RESILVERING,
367                             B_TRUE) == 0);
368                 if (vd->vdev_faulted)
369                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_FAULTED,
370                             B_TRUE) == 0);
371                 if (vd->vdev_degraded)
372                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_DEGRADED,
373                             B_TRUE) == 0);
374                 if (vd->vdev_removed)
375                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_REMOVED,
376                             B_TRUE) == 0);
377                 if (vd->vdev_unspare)
378                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_UNSPARE,
379                             B_TRUE) == 0);
380                 if (vd->vdev_ishole)
381                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_IS_HOLE,
382                             B_TRUE) == 0);

384                 switch (vd->vdev_stat.vs_aux) {
385                 case VDEV_AUX_ERR_EXCEEDED:
386                         aux = "err_exceeded";
387                         break;

389                 case VDEV_AUX_EXTERNAL:
390                         aux = "external";
391                         break;
392                 }

394                 if (aux != NULL)
395                         VERIFY(nvlist_add_string(nv, ZPOOL_CONFIG_AUX_STATE,
396                             aux) == 0);

398                 if (vd->vdev_splitting && vd->vdev_orig_guid != 0LL) {
399                         VERIFY(nvlist_add_uint64(nv, ZPOOL_CONFIG_ORIG_GUID,
400                             vd->vdev_orig_guid) == 0);
```



```
401                 }
402         }

404         return (nv);
405 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   29031 Mon Aug  5 21:59:10 2013
new/usr/src/uts/common/sys/fs/zfs.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 (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013, Saso Kiselkov. All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_FS_ZFS_H
#define _SYS_FS_ZFS_H

#include <sys/time.h>

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Types and constants shared between userland and the kernel.
 */

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

*unchanged_portion_omitted_*

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



```
#define ZPOOL_CONFIG_POOL_NAME          "name"
#define ZPOOL_CONFIG_POOL_STATE         "state"
#define ZPOOL_CONFIG_POOL_TXG           "txg"
#define ZPOOL_CONFIG_POOL_GUID          "pool_guid"
#define ZPOOL_CONFIG_CREATE_TXG         "create_txg"
#define ZPOOL_CONFIG_TOP_GUID           "top_guid"
#define ZPOOL_CONFIG_VDEV_TREE          "vdev_tree"
#define ZPOOL_CONFIG_TYPE               "type"
#define ZPOOL_CONFIG_CHILDREN           "children"
#define ZPOOL_CONFIG_ID                 "id"
#define ZPOOL_CONFIG_GUID               "guid"
#define ZPOOL_CONFIG_PATH               "path"
#define ZPOOL_CONFIG_DEVID              "devid"
#define ZPOOL_CONFIG_METASLAB_ARRAY     "metaslab_array"
#define ZPOOL_CONFIG_METASLAB_SHIFT     "metaslab_shift"
#define ZPOOL_CONFIG_ASHIFT             "ashift"
#define ZPOOL_CONFIG_ASIZE              "asize"
#define ZPOOL_CONFIG_DTL                "DTL"
#define ZPOOL_CONFIG_SCAN_STATS         "scan_stats"    /* not stored on disk */
#define ZPOOL_CONFIG_VDEV_STATS         "vdev_stats"    /* not stored on disk */
#define ZPOOL_CONFIG_WHOLE_DISK         "whole_disk"
#define ZPOOL_CONFIG_ERRCOUNT           "error_count"
#define ZPOOL_CONFIG_NOT_PRESENT        "not_present"
#define ZPOOL_CONFIG_SPARES             "spares"
#define ZPOOL_CONFIG_IS_SPARE           "is_spare"
#define ZPOOL_CONFIG_NPARITY            "nparity"
#define ZPOOL_CONFIG_HOSTID             "hostid"
#define ZPOOL_CONFIG_HOSTNAME           "hostname"
#define ZPOOL_CONFIG_LOADED_TIME        "initial_load_time"
#define ZPOOL_CONFIG_UNSPARE            "unspare"
#define ZPOOL_CONFIG_PHYS_PATH          "phys_path"
#define ZPOOL_CONFIG_IS_LOG             "is_log"
#define ZPOOL_CONFIG_L2CACHE            "l2cache"
#define ZPOOL_CONFIG_L2CACHE_PERSISTENT "l2cache_persistent"
#define ZPOOL_CONFIG_HOLE_ARRAY         "hole_array"
#define ZPOOL_CONFIG_VDEV_CHILDREN      "vdev_children"
#define ZPOOL_CONFIG_IS_HOLE            "is_hole"
#define ZPOOL_CONFIG_DDT_HISTOGRAM      "ddt_histogram"
#define ZPOOL_CONFIG_DDT_OBJ_STATS      "ddt_object_stats"
#define ZPOOL_CONFIG_DDT_STATS          "ddt_stats"
#define ZPOOL_CONFIG_SPLIT              "splitcfg"
#define ZPOOL_CONFIG_ORIG_GUID          "orig_guid"
#define ZPOOL_CONFIG_SPLIT_GUID         "split_guid"
#define ZPOOL_CONFIG_SPLIT_LIST         "guid_list"
#define ZPOOL_CONFIG_REMOVING           "removing"
#define ZPOOL_CONFIG_RESILVERING        "resilvering"
#define ZPOOL_CONFIG_COMMENT            "comment"
#define ZPOOL_CONFIG_SUSPENDED          "suspended"     /* not stored on disk */
#define ZPOOL_CONFIG_TIMESTAMP          "timestamp"     /* not stored on disk */
#define ZPOOL_CONFIG_BOOTFS             "bootfs"        /* not stored on disk */
#define ZPOOL_CONFIG_MISSING_DEVICES    "missing_vdevs" /* not stored on disk */
#define ZPOOL_CONFIG_LOAD_INFO          "load_info"     /* not stored on disk */
#define ZPOOL_CONFIG_REWIND_INFO        "rewind_info"   /* not stored on disk */
#define ZPOOL_CONFIG_UNSUP_FEAT         "unsup_feat"    /* not stored on disk */
#define ZPOOL_CONFIG_ENABLED_FEAT       "enabled_feat"  /* not stored on disk */
#define ZPOOL_CONFIG_CAN_RDONLY         "can_rdonly"    /* not stored on disk */
#define ZPOOL_CONFIG_FEATURES_FOR_READ  "features_for_read"
#define ZPOOL_CONFIG_FEATURE_STATS      "feature_stats" /* not stored on disk */
/*
 * The persistent vdev state is stored as separate values rather than a single
 * 'vdev_state' entry.  This is because a device can be in multiple states, such
 * as offline and degraded.
 */
#define ZPOOL_CONFIG_OFFLINE            "offline"
#define ZPOOL_CONFIG_FAULTED            "faulted"
#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 Aug  5 21:59:10 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_
```