```
**********************************************************
    7219 Tue Apr 23 14:09:34 2013
new/usr/src/common/zfs/zfs_fletcher.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Fletcher Checksums
 * ------------------
 *
 * ZFS's 2nd and 4th order Fletcher checksums are defined by the following
 * recurrence relations:
 *
 *      a  = a    + f
 *       i    i-1    i-1
 *
 *      b  = b    + a
 *       i    i-1    i
 *
 *      c  = c    + b           (fletcher-4 only)
 *       i    i-1    i
 *
 *      d  = d    + c           (fletcher-4 only)
 *       i    i-1    i
 *
 * Where
 *      a_0 = b_0 = c_0 = d_0 = 0
 * and
 *      f_0 .. f_(n-1) are the input data.
 *
 * Using standard techniques, these translate into the following series:
 *
 *           __n_                          __n_
 *           \   |                         \   |
 *      a  =  >     f               b  =  >     i * f
 *       n   /___|   n - i            n   /___|       n - i
 *           i = 1                         i = 1
 *
```



```
 *
 *           __n_                          __n_
 *           \   |  i*(i+1)                 \   |  i*(i+1)*(i+2)
 *      c  =  >     ------- f           d  =  >     ------------- f
 *       n   /___|     2     n - i       n   /___|        6        n - i
 *           i = 1                         i = 1
 *
 * For fletcher-2, the f_is are 64-bit, and [ab]_i are 64-bit accumulators.
 * Since the additions are done mod (2^64), errors in the high bits may not
 * be noticed.  For this reason, fletcher-2 is deprecated.
 *
 * For fletcher-4, the f_is are 32-bit, and [abcd]_i are 64-bit accumulators.
 * A conservative estimate of how big the buffer can get before we overflow
 * can be estimated using f_i = 0xffffffff for all i:
 *
 * % bc
 *  f=2^32-1;d=0; for (i = 1; d<2^64; i++) { d += f*i*(i+1)*(i+2)/6 }; (i-1)*4
 * 2264
 *  quit
 * %
 *
 * So blocks of up to 2k will not overflow.  Our largest block size is
 * 128k, which has 32k 4-byte words, so we can compute the largest possible
 * accumulators, then divide by 2^64 to figure the max amount of overflow:
 *
 * % bc
 *  a=b=c=d=0; f=2^32-1; for (i=1; i<=32*1024; i++) { a+=f; b+=a; c+=b; d+=c }
 *  a/2^64;b/2^64;c/2^64;d/2^64
 * 0
 * 0
 * 1365
 * 11186858
 *  quit
 * %
 *
 * So a and b cannot overflow.  To make sure each bit of input has some
 * effect on the contents of c and d, we can look at what the factors of
 * the coefficients in the equations for c_n and d_n are.  The number of 2s
 * in the factors determines the lowest set bit in the multiplier.  Running
 * through the cases for n*(n+1)/2 reveals that the highest power of 2 is
 * 2^14, and for n*(n+1)*(n+2)/6 it is 2^15.  So while some data may overflow
 * the 64-bit accumulators, every bit of every f_i effects every accumulator,
 * even for 128k blocks.
 *
 * If we wanted to make a stronger version of fletcher4 (fletcher4c?),
 * we could do our calculations mod (2^32 - 1) by adding in the carries
 * periodically, and store the number of carries in the top 32-bits.
 *
 * --------------------
 * Checksum Performance
 * --------------------
 *
 * There are two interesting components to checksum performance: cached and
 * uncached performance.  With cached data, fletcher-2 is about four times
 * faster than fletcher-4.  With uncached data, the performance difference is
 * negligible, since the cost of a cache fill dominates the processing time.
 * Even though fletcher-4 is slower than fletcher-2, it is still a pretty
 * efficient pass over the data.
 *
 * In normal operation, the data which is being checksummed is in a buffer
 * which has been filled either by:
 *
 *      1. a compression step, which will be mostly cached, or
 *      2. a bcopy() or copyin(), which will be uncached (because the
 *         copy is cache-bypassing).
 *
```

```
 * For both cached and uncached data, both fletcher checksums are much faster
 * than sha-256, and slower than 'off', which doesn't touch the data at all.
 */

/*
 * TODO: vectorize these functions
 * All of these functions are written so that each iteration of the loop
 * depends on the value of the previous iteration.  Also, in the fletcher_4
 * functions, each statement of the loop body depends on the previous
 * statement.  These dependencies prevent the compiler from vectorizing the
 * code to take advantage of SIMD extensions (unless GCC is far smarter than I
 * think).  It would be easy to rewrite the loops to be amenable to
 * autovectorization.
 */

#endif /* ! codereview */
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/byteorder.h>
#include <sys/zio.h>
#include <sys/spa.h>

void
fletcher_2_native(const void *buf, uint64_t size, zio_cksum_t *zcp)
{
        const uint64_t *ip = buf;
        const uint64_t *ipend = ip + (size / sizeof (uint64_t));
        uint64_t a0, b0, a1, b1;

        for (a0 = b0 = a1 = b1 = 0; ip < ipend; ip += 2) {
                a0 += ip[0];
                a1 += ip[1];
                b0 += a0;
                b1 += a1;
        }

        ZIO_SET_CHECKSUM(zcp, a0, a1, b0, b1);
}

void
fletcher_2_byteswap(const void *buf, uint64_t size, zio_cksum_t *zcp)
{
        const uint64_t *ip = buf;
        const uint64_t *ipend = ip + (size / sizeof (uint64_t));
        uint64_t a0, b0, a1, b1;

        for (a0 = b0 = a1 = b1 = 0; ip < ipend; ip += 2) {
                a0 += BSWAP_64(ip[0]);
                a1 += BSWAP_64(ip[1]);
                b0 += a0;
                b1 += a1;
        }

        ZIO_SET_CHECKSUM(zcp, a0, a1, b0, b1);
}

void
fletcher_4_native(const void *buf, uint64_t size, zio_cksum_t *zcp)
{
        const uint32_t *ip = buf;
        const uint32_t *ipend = ip + (size / sizeof (uint32_t));
        uint64_t a, b, c, d;

        for (a = b = c = d = 0; ip < ipend; ip++) {
                a += ip[0];
                b += a;
```



```
                c += b;
                d += c;
        }

        ZIO_SET_CHECKSUM(zcp, a, b, c, d);
}

void
fletcher_4_byteswap(const void *buf, uint64_t size, zio_cksum_t *zcp)
{
        const uint32_t *ip = buf;
        const uint32_t *ipend = ip + (size / sizeof (uint32_t));
        uint64_t a, b, c, d;

        for (a = b = c = d = 0; ip < ipend; ip++) {
                a += BSWAP_32(ip[0]);
                b += a;
                c += b;
                d += c;
        }

        ZIO_SET_CHECKSUM(zcp, a, b, c, d);
}

void
fletcher_4_incremental_native(const void *buf, uint64_t size,
    zio_cksum_t *zcp)
{
        const uint32_t *ip = buf;
        const uint32_t *ipend = ip + (size / sizeof (uint32_t));
        uint64_t a, b, c, d;

        a = zcp->zc_word[0];
        b = zcp->zc_word[1];
        c = zcp->zc_word[2];
        d = zcp->zc_word[3];

        for (; ip < ipend; ip++) {
                a += ip[0];
                b += a;
                c += b;
                d += c;
        }

        ZIO_SET_CHECKSUM(zcp, a, b, c, d);
}

void
fletcher_4_incremental_byteswap(const void *buf, uint64_t size,
    zio_cksum_t *zcp)
{
        const uint32_t *ip = buf;
        const uint32_t *ipend = ip + (size / sizeof (uint32_t));
        uint64_t a, b, c, d;

        a = zcp->zc_word[0];
        b = zcp->zc_word[1];
        c = zcp->zc_word[2];
        d = zcp->zc_word[3];

        for (; ip < ipend; ip++) {
                a += BSWAP_32(ip[0]);
                b += a;
                c += b;
                d += c;
        }
```

```
257         ZIO_SET_CHECKSUM(zcp, a, b, c, d);
258 }
```

```
**********************************************************
  111007 Tue Apr 23 14:09:34 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
4431 /*
4432  * Convert the zvol's volume size to an appropriate reservation.
4433  * Note: If this routine is updated, it is necessary to update the ZFS test
4434  * suite's shell version in reservation.kshlib.
4435  */
4436 #endif /* ! codereview */
4437 uint64_t
4438 zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
4439 {
4440         uint64_t numdb;
4441         uint64_t nblocks, volblocksize;
4442         int ncopies;
4443         char *strval;

4445         if (nvlist_lookup_string(props,
4446             zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
4447                 ncopies = atoi(strval);
4448         else
4449                 ncopies = 1;
4450         if (nvlist_lookup_uint64(props,
4451             zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
4452             &volblocksize) != 0)
4453                 volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
4454         nblocks = volsize/volblocksize;
4455         /* start with metadnode L0-L6 */
4456         numdb = 7;
4457         /* calculate number of indirects */
4458         while (nblocks > 1) {
4459                 nblocks += DNODES_PER_LEVEL - 1;
4460                 nblocks /= DNODES_PER_LEVEL;
4461                 numdb += nblocks;
4462         }
4463         numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
4464         volsize *= ncopies;
4465         /*
4466          * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
4467          * compressed, but in practice they compress down to about
4468          * 1100 bytes
4469          */
4470         numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
4471         volsize += numdb;
4472         return (volsize);
4473 }
```

```
**********************************************************
  135193 Tue Apr 23 14:09:35 2013
new/usr/src/uts/common/fs/zfs/arc.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

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

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



```
 287         kstat_named_t arcstat_other_size;
 288         kstat_named_t arcstat_l2_hits;
 289         kstat_named_t arcstat_l2_misses;
 290         kstat_named_t arcstat_l2_feeds;
 291         kstat_named_t arcstat_l2_rw_clash;
 292         kstat_named_t arcstat_l2_read_bytes;
 293         kstat_named_t arcstat_l2_write_bytes;
 294         kstat_named_t arcstat_l2_writes_sent;
 295         kstat_named_t arcstat_l2_writes_done;
 296         kstat_named_t arcstat_l2_writes_error;
 297         kstat_named_t arcstat_l2_writes_hdr_miss;
 298         kstat_named_t arcstat_l2_evict_lock_retry;
 299         kstat_named_t arcstat_l2_evict_reading;
 300         kstat_named_t arcstat_l2_free_on_write;
 301         kstat_named_t arcstat_l2_abort_lowmem;
 302         kstat_named_t arcstat_l2_cksum_bad;
 303         kstat_named_t arcstat_l2_io_error;
 304         kstat_named_t arcstat_l2_size;
 305         kstat_named_t arcstat_l2_hdr_size;
 306         kstat_named_t arcstat_memory_throttle_count;
 307         kstat_named_t arcstat_duplicate_buffers;
 308         kstat_named_t arcstat_duplicate_buffers_size;
 309         kstat_named_t arcstat_duplicate_reads;
 310         kstat_named_t arcstat_meta_used;
 311         kstat_named_t arcstat_meta_limit;
 312         kstat_named_t arcstat_meta_max;
 313 } arc_stats_t;

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

```
353         { "l2_rw_clash",                KSTAT_DATA_UINT64 },
354         { "l2_read_bytes",              KSTAT_DATA_UINT64 },
355         { "l2_write_bytes",             KSTAT_DATA_UINT64 },
356         { "l2_writes_sent",             KSTAT_DATA_UINT64 },
357         { "l2_writes_done",             KSTAT_DATA_UINT64 },
358         { "l2_writes_error",            KSTAT_DATA_UINT64 },
359         { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
360         { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
361         { "l2_evict_reading",           KSTAT_DATA_UINT64 },
362         { "l2_free_on_write",           KSTAT_DATA_UINT64 },
363         { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
364         { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
365         { "l2_io_error",                KSTAT_DATA_UINT64 },
366         { "l2_size",                    KSTAT_DATA_UINT64 },
367         { "l2_hdr_size",                KSTAT_DATA_UINT64 },
368         { "memory_throttle_count",      KSTAT_DATA_UINT64 },
369         { "duplicate_buffers",          KSTAT_DATA_UINT64 },
370         { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
371         { "duplicate_reads",            KSTAT_DATA_UINT64 },
372         { "arc_meta_used",              KSTAT_DATA_UINT64 },
373         { "arc_meta_limit",             KSTAT_DATA_UINT64 },
374         { "arc_meta_max",               KSTAT_DATA_UINT64 }
375 };

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

379 #define ARCSTAT_INCR(stat, val) \
380         atomic_add_64(&arc_stats.stat.value.ui64, (val));

382 #define ARCSTAT_BUMP(stat)      ARCSTAT_INCR(stat, 1)
383 #define ARCSTAT_BUMPDOWN(stat)  ARCSTAT_INCR(stat, -1)

385 #define ARCSTAT_MAX(stat, val) {                                        \
386         uint64_t m;                                                    \
387         while ((val) > (m = arc_stats.stat.value.ui64) &&              \
388             (m != atomic_cas_64(&arc_stats.stat.value.ui64, m, (val)))) \
389                 continue;                                              \
390 }

392 #define ARCSTAT_MAXSTAT(stat) \
393         ARCSTAT_MAX(stat##_max, arc_stats.stat.value.ui64)

395 /*
396  * We define a macro to allow ARC hits/misses to be easily broken down by
397  * two separate conditions, giving a total of four different subtypes for
398  * each of hits and misses (so eight statistics total).
399  */
400 #define ARCSTAT_CONDSTAT(cond1, stat1, notstat1, cond2, stat2, notstat2, stat) \
401         if (cond1) {                                                   \
402                 if (cond2) {                                           \
403                         ARCSTAT_BUMP(arcstat_##stat1##_##stat2##_##stat); \
404                 } else {                                               \
405                         ARCSTAT_BUMP(arcstat_##stat1##_##notstat2##_##stat); \
406                 }                                                      \
407         } else {                                                       \
408                 if (cond2) {                                           \
409                         ARCSTAT_BUMP(arcstat_##notstat1##_##stat2##_##stat); \
410                 } else {                                               \
411                         ARCSTAT_BUMP(arcstat_##notstat1##_##notstat2##_##stat);\
412                 }                                                      \
413         }

415 kstat_t                 *arc_ksp;
416 static arc_state_t      *arc_anon;
417 static arc_state_t      *arc_mru;
418 static arc_state_t      *arc_mru_ghost;
```



```
419 static arc_state_t      *arc_mfu;
420 static arc_state_t      *arc_mfu_ghost;
421 static arc_state_t      *arc_l2c_only;

423 /*
424  * There are several ARC variables that are critical to export as kstats --
425  * but we don't want to have to grovel around in the kstat whenever we wish to
426  * manipulate them.  For these variables, we therefore define them to be in
427  * terms of the statistic variable.  This assures that we are not introducing
428  * the possibility of inconsistency by having shadow copies of the variables,
429  * while still allowing the code to be readable.
430  */
431 #define arc_size        ARCSTAT(arcstat_size)   /* actual total arc size */
432 #define arc_p           ARCSTAT(arcstat_p)      /* target size of MRU */
433 #define arc_c           ARCSTAT(arcstat_c)      /* target size of cache */
434 #define arc_c_min       ARCSTAT(arcstat_c_min)  /* min target cache size */
435 #define arc_c_max       ARCSTAT(arcstat_c_max)  /* max target cache size */
436 #define arc_meta_limit  ARCSTAT(arcstat_meta_limit) /* max size for metadata */
437 #define arc_meta_used   ARCSTAT(arcstat_meta_used) /* size of metadata */
438 #define arc_meta_max    ARCSTAT(arcstat_meta_max) /* max size of metadata */

440 static int              arc_no_grow;    /* Don't try to grow cache size */
441 static uint64_t         arc_tempreserve;
442 static uint64_t         arc_loaned_bytes;

444 typedef struct l2arc_buf_hdr l2arc_buf_hdr_t;

446 typedef struct arc_callback arc_callback_t;

448 struct arc_callback {
449         void                    *acb_private;
450         arc_done_func_t         *acb_done;
451         arc_buf_t               *acb_buf;
452         zio_t                   *acb_zio_dummy;
453         arc_callback_t          *acb_next;
454 };

456 typedef struct arc_write_callback arc_write_callback_t;

458 struct arc_write_callback {
459         void            *awcb_private;
460         arc_done_func_t *awcb_ready;
461         arc_done_func_t *awcb_done;
462         arc_buf_t       *awcb_buf;
463 };

465 struct arc_buf_hdr {
466         /* protected by hash lock */
467         dva_t                   b_dva;
468         uint64_t                b_birth;
469         uint64_t                b_cksum0;

471         kmutex_t                b_freeze_lock;
472         zio_cksum_t             *b_freeze_cksum;
473         void                    *b_thawed;

475         arc_buf_hdr_t           *b_hash_next;
476         arc_buf_t               *b_buf;
477         uint32_t                b_flags;
478         uint32_t                b_datacnt;

480         arc_callback_t          *b_acb;
481         kcondvar_t              b_cv;

483         /* immutable */
484         arc_buf_contents_t      b_type;
```

```
485         uint64_t                b_size;
486         uint64_t                b_spa;

488         /* protected by arc state mutex */
489         arc_state_t             *b_state;
490         list_node_t             b_arc_node;

492         /* updated atomically */
493         clock_t                 b_arc_access;

495         /* self protecting */
496         refcount_t              b_refcnt;

498         l2arc_buf_hdr_t         *b_l2hdr;
499         list_node_t             b_l2node;
500 };

502 static arc_buf_t *arc_eviction_list;
503 static kmutex_t arc_eviction_mtx;
504 static arc_buf_hdr_t arc_eviction_hdr;
505 static void arc_get_data_buf(arc_buf_t *buf);
506 static void arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock);
507 static int arc_evict_needed(arc_buf_contents_t type);
508 static void arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes);
509 static void arc_buf_watch(arc_buf_t *buf);

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

513 #define GHOST_STATE(state)      \
514         ((state) == arc_mru_ghost || (state) == arc_mfu_ghost ||        \
515         (state) == arc_l2c_only)

517 /*
518  * Private ARC flags.  These flags are private ARC only flags that will show up
519  * in b_flags in the arc_hdr_buf_t.  Some flags are publicly declared, and can
520  * be passed in as arc_flags in things like arc_read.  However, these flags
521  * should never be passed and should only be set by ARC code.  When adding new
522  * public flags, make sure not to smash the private ones.
523  */

525 #define ARC_IN_HASH_TABLE       (1 << 9)        /* this buffer is hashed */
526 #define ARC_IO_IN_PROGRESS      (1 << 10)       /* I/O in progress for buf */
527 #define ARC_IO_ERROR            (1 << 11)       /* I/O failed for buf */
528 #define ARC_FREED_IN_READ       (1 << 12)       /* buf freed while in read */
529 #define ARC_BUF_AVAILABLE       (1 << 13)       /* block not in active use */
530 #define ARC_INDIRECT            (1 << 14)       /* this is an indirect block */
531 #define ARC_FREE_IN_PROGRESS    (1 << 15)       /* hdr about to be freed */
532 #define ARC_L2_WRITING          (1 << 16)       /* L2ARC write in progress */
533 #define ARC_L2_EVICTED          (1 << 17)       /* evicted during I/O */
534 #define ARC_L2_WRITE_HEAD       (1 << 18)       /* head of write list */

536 #define HDR_IN_HASH_TABLE(hdr)  ((hdr)->b_flags & ARC_IN_HASH_TABLE)
537 #define HDR_IO_IN_PROGRESS(hdr) ((hdr)->b_flags & ARC_IO_IN_PROGRESS)
538 #define HDR_IO_ERROR(hdr)       ((hdr)->b_flags & ARC_IO_ERROR)
539 #define HDR_PREFETCH(hdr)       ((hdr)->b_flags & ARC_PREFETCH)
540 #define HDR_FREED_IN_READ(hdr)  ((hdr)->b_flags & ARC_FREED_IN_READ)
541 #define HDR_BUF_AVAILABLE(hdr)  ((hdr)->b_flags & ARC_BUF_AVAILABLE)
542 #define HDR_FREE_IN_PROGRESS(hdr)       ((hdr)->b_flags & ARC_FREE_IN_PROGRESS)
543 #define HDR_L2CACHE(hdr)        ((hdr)->b_flags & ARC_L2CACHE)
544 #define HDR_L2_READING(hdr)     ((hdr)->b_flags & ARC_IO_IN_PROGRESS && \
545                                     (hdr)->b_l2hdr != NULL)
546 #define HDR_L2_WRITING(hdr)     ((hdr)->b_flags & ARC_L2_WRITING)
547 #define HDR_L2_EVICTED(hdr)     ((hdr)->b_flags & ARC_L2_EVICTED)
548 #define HDR_L2_WRITE_HEAD(hdr)  ((hdr)->b_flags & ARC_L2_WRITE_HEAD)

550 /*
```



```
551  * Other sizes
552  */

554 #define HDR_SIZE ((int64_t)sizeof (arc_buf_hdr_t))
555 #define L2HDR_SIZE ((int64_t)sizeof (l2arc_buf_hdr_t))

557 /*
558  * Hash table routines
559  */

561 #define HT_LOCK_PAD     64

563 struct ht_lock {
564         kmutex_t        ht_lock;
565 #ifdef _KERNEL
566         unsigned char   pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
567 #endif
568 };

570 #define BUF_LOCKS 256
571 typedef struct buf_hash_table {
572         uint64_t ht_mask;
573         arc_buf_hdr_t **ht_table;
574         struct ht_lock ht_locks[BUF_LOCKS];
575 } buf_hash_table_t;

577 static buf_hash_table_t buf_hash_table;

579 #define BUF_HASH_INDEX(spa, dva, birth) \
580         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
581 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
582 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
583 #define HDR_LOCK(hdr) \
584         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))

586 uint64_t zfs_crc64_table[256];

588 /*
589  * Level 2 ARC
590  */

592 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
593 #define L2ARC_HEADROOM          2               /* num of writes */
594 #define L2ARC_FEED_SECS         1               /* caching interval secs */
595 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

597 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
598 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

600 /*
601  * L2ARC Performance Tunables
602  */
603 uint64_t l2arc_write_max = L2ARC_WRITE_SIZE;    /* default max write size */
604 uint64_t l2arc_write_boost = L2ARC_WRITE_SIZE;  /* extra write during warmup */
605 uint64_t l2arc_headroom = L2ARC_HEADROOM;       /* number of dev writes */
606 uint64_t l2arc_feed_secs = L2ARC_FEED_SECS;     /* interval seconds */
607 uint64_t l2arc_feed_min_ms = L2ARC_FEED_MIN_MS; /* min interval milliseconds */
608 boolean_t l2arc_noprefetch = B_TRUE;            /* don't cache prefetch bufs */
609 boolean_t l2arc_feed_again = B_TRUE;            /* turbo warmup */
610 boolean_t l2arc_norw = B_TRUE;                  /* no reads during writes */

612 /*
613  * L2ARC Internals
614  */
615 typedef struct l2arc_dev {
616         vdev_t                  *l2ad_vdev;     /* vdev */
```

```
617         spa_t                   *l2ad_spa;      /* spa */
618         uint64_t                l2ad_hand;      /* next write location */
619         uint64_t                l2ad_write;     /* desired write size, bytes */
620         uint64_t                l2ad_boost;     /* warmup write boost, bytes */
621         uint64_t                l2ad_start;     /* first addr on device */
622         uint64_t                l2ad_end;       /* last addr on device */
623         uint64_t                l2ad_evict;     /* last addr eviction reached */
624         boolean_t               l2ad_first;     /* first sweep through */
625         boolean_t               l2ad_writing;   /* currently writing */
626         list_t                  *l2ad_buflist;  /* buffer list */
627         list_node_t             l2ad_node;      /* device list node */
628 } l2arc_dev_t;

630 static list_t L2ARC_dev_list;                   /* device list */
631 static list_t *l2arc_dev_list;                  /* device list pointer */
632 static kmutex_t l2arc_dev_mtx;                  /* device list mutex */
633 static l2arc_dev_t *l2arc_dev_last;             /* last device used */
634 static kmutex_t l2arc_buflist_mtx;              /* mutex for all buflists */
635 static list_t L2ARC_free_on_write;              /* free after write buf list */
636 static list_t *l2arc_free_on_write;             /* free after write list ptr */
637 static kmutex_t l2arc_free_on_write_mtx;        /* mutex for list */
638 static uint64_t l2arc_ndev;                     /* number of devices */

640 typedef struct l2arc_read_callback {
641         arc_buf_t       *l2rcb_buf;             /* read buffer */
642         spa_t           *l2rcb_spa;             /* spa */
643         blkptr_t        l2rcb_bp;               /* original blkptr */
644         zbookmark_t     l2rcb_zb;               /* original bookmark */
645         int             l2rcb_flags;            /* original flags */
646 } l2arc_read_callback_t;

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

653 struct l2arc_buf_hdr {
654         /* protected by arc_buf_hdr  mutex */
655         l2arc_dev_t     *b_dev;                 /* L2ARC device */
656         uint64_t        b_daddr;                /* disk address, offset byte */
657 };

659 typedef struct l2arc_data_free {
660         /* protected by l2arc_free_on_write_mtx */
661         void            *l2df_data;
662         size_t          l2df_size;
663         void            (*l2df_func)(void *, size_t);
664         list_node_t     l2df_list_node;
665 } l2arc_data_free_t;

667 static kmutex_t l2arc_feed_thr_lock;
668 static kcondvar_t l2arc_feed_thr_cv;
669 static uint8_t l2arc_thread_exit;

671 static void l2arc_read_done(zio_t *zio);
672 static void l2arc_hdr_stat_add(void);
673 static void l2arc_hdr_stat_remove(void);

675 static uint64_t
676 buf_hash(uint64_t spa, const dva_t *dva, uint64_t birth)
677 {
678         uint8_t *vdva = (uint8_t *)dva;
679         uint64_t crc = -1ULL;
680         int i;

682         ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);
```



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

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

689         return (crc);
690 }

692 #define BUF_EMPTY(buf)                                          \
693         ((buf)->b_dva.dva_word[0] == 0 &&                       \
694         (buf)->b_dva.dva_word[1] == 0 &&                        \
695         (buf)->b_birth == 0)

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

702 static void
703 buf_discard_identity(arc_buf_hdr_t *hdr)
704 {
705         hdr->b_dva.dva_word[0] = 0;
706         hdr->b_dva.dva_word[1] = 0;
707         hdr->b_birth = 0;
708         hdr->b_cksum0 = 0;
709 }

711 static arc_buf_hdr_t *
712 buf_hash_find(uint64_t spa, const dva_t *dva, uint64_t birth, kmutex_t **lockp)
713 {
714         uint64_t idx = BUF_HASH_INDEX(spa, dva, birth);
715         kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
716         arc_buf_hdr_t *buf;

718         mutex_enter(hash_lock);
719         for (buf = buf_hash_table.ht_table[idx]; buf != NULL;
720             buf = buf->b_hash_next) {
721                 if (BUF_EQUAL(spa, dva, birth, buf)) {
722                         *lockp = hash_lock;
723                         return (buf);
724                 }
725         }
726         mutex_exit(hash_lock);
727         *lockp = NULL;
728         return (NULL);
729 }

731 /*
732  * Insert an entry into the hash table.  If there is already an element
733  * equal to elem in the hash table, then the already existing element
734  * will be returned and the new element will not be inserted.
735  * Otherwise returns NULL.
736  */
737 static arc_buf_hdr_t *
738 buf_hash_insert(arc_buf_hdr_t *buf, kmutex_t **lockp)
739 {
740         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
741         kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
742         arc_buf_hdr_t *fbuf;
743         uint32_t i;

745         ASSERT(!HDR_IN_HASH_TABLE(buf));
746         *lockp = hash_lock;
747         mutex_enter(hash_lock);
748         for (fbuf = buf_hash_table.ht_table[idx], i = 0; fbuf != NULL;
```

```
749             fbuf = fbuf->b_hash_next, i++) {
750                 if (BUF_EQUAL(buf->b_spa, &buf->b_dva, buf->b_birth, fbuf))
751                         return (fbuf);
752         }
753
754         buf->b_hash_next = buf_hash_table.ht_table[idx];
755         buf_hash_table.ht_table[idx] = buf;
756         buf->b_flags |= ARC_IN_HASH_TABLE;
757
758         /* collect some hash table performance data */
759         if (i > 0) {
760                 ARCSTAT_BUMP(arcstat_hash_collisions);
761                 if (i == 1)
762                         ARCSTAT_BUMP(arcstat_hash_chains);
763
764                 ARCSTAT_MAX(arcstat_hash_chain_max, i);
765         }
766
767         ARCSTAT_BUMP(arcstat_hash_elements);
768         ARCSTAT_MAXSTAT(arcstat_hash_elements);
769
770         return (NULL);
771 }
772
773 static void
774 buf_hash_remove(arc_buf_hdr_t *buf)
775 {
776         arc_buf_hdr_t *fbuf, **bufp;
777         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
778
779         ASSERT(MUTEX_HELD(BUF_HASH_LOCK(idx)));
780         ASSERT(HDR_IN_HASH_TABLE(buf));
781
782         bufp = &buf_hash_table.ht_table[idx];
783         while ((fbuf = *bufp) != buf) {
784                 ASSERT(fbuf != NULL);
785                 bufp = &fbuf->b_hash_next;
786         }
787         *bufp = buf->b_hash_next;
788         buf->b_hash_next = NULL;
789         buf->b_flags &= ~ARC_IN_HASH_TABLE;
790
791         /* collect some hash table performance data */
792         ARCSTAT_BUMPDOWN(arcstat_hash_elements);
793
794         if (buf_hash_table.ht_table[idx] &&
795             buf_hash_table.ht_table[idx]->b_hash_next == NULL)
796                 ARCSTAT_BUMPDOWN(arcstat_hash_chains);
797 }
798
799 /*
800  * Global data structures and functions for the buf kmem cache.
801  */
802 static kmem_cache_t *hdr_cache;
803 static kmem_cache_t *buf_cache;
804
805 static void
806 buf_fini(void)
807 {
808         int i;
809
810         kmem_free(buf_hash_table.ht_table,
811             (buf_hash_table.ht_mask + 1) * sizeof (void *));
812         for (i = 0; i < BUF_LOCKS; i++)
813                 mutex_destroy(&buf_hash_table.ht_locks[i].ht_lock);
814         kmem_cache_destroy(hdr_cache);
```



```
815         kmem_cache_destroy(buf_cache);
816 }
817
818 /*
819  * Constructor callback - called when the cache is empty
820  * and a new buf is requested.
821  */
822 /* ARGSUSED */
823 static int
824 hdr_cons(void *vbuf, void *unused, int kmflag)
825 {
826         arc_buf_hdr_t *buf = vbuf;
827
828         bzero(buf, sizeof (arc_buf_hdr_t));
829         refcount_create(&buf->b_refcnt);
830         cv_init(&buf->b_cv, NULL, CV_DEFAULT, NULL);
831         mutex_init(&buf->b_freeze_lock, NULL, MUTEX_DEFAULT, NULL);
832         arc_space_consume(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
833
834         return (0);
835 }
836
837 /* ARGSUSED */
838 static int
839 buf_cons(void *vbuf, void *unused, int kmflag)
840 {
841         arc_buf_t *buf = vbuf;
842
843         bzero(buf, sizeof (arc_buf_t));
844         mutex_init(&buf->b_evict_lock, NULL, MUTEX_DEFAULT, NULL);
845         arc_space_consume(sizeof (arc_buf_t), ARC_SPACE_HDRS);
846
847         return (0);
848 }
849
850 /*
851  * Destructor callback - called when a cached buf is
852  * no longer required.
853  */
854 /* ARGSUSED */
855 static void
856 hdr_dest(void *vbuf, void *unused)
857 {
858         arc_buf_hdr_t *buf = vbuf;
859
860         ASSERT(BUF_EMPTY(buf));
861         refcount_destroy(&buf->b_refcnt);
862         cv_destroy(&buf->b_cv);
863         mutex_destroy(&buf->b_freeze_lock);
864         arc_space_return(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
865 }
866
867 /* ARGSUSED */
868 static void
869 buf_dest(void *vbuf, void *unused)
870 {
871         arc_buf_t *buf = vbuf;
872
873         mutex_destroy(&buf->b_evict_lock);
874         arc_space_return(sizeof (arc_buf_t), ARC_SPACE_HDRS);
875 }
876
877 /*
878  * Reclaim callback -- invoked when memory is low.
879  */
880 /* ARGSUSED */
```

```
881 static void
882 hdr_recl(void *unused)
883 {
884         dprintf("hdr_recl called\n");
885         /*
886          * umem calls the reclaim func when we destroy the buf cache,
887          * which is after we do arc_fini().
888          */
889         if (!arc_dead)
890                 cv_signal(&arc_reclaim_thr_cv);
891 }

893 static void
894 buf_init(void)
895 {
896         uint64_t *ct;
897         uint64_t hsize = 1ULL << 12;
898         int i, j;

900         /*
901          * The hash table is big enough to fill all of physical memory
902          * with an average 64K block size.  The table will take up
903          * totalmem*sizeof(void*)/64K (eg. 128KB/GB with 8-byte pointers).
904          */
905         while (hsize * 65536 < physmem * PAGESIZE)
906                 hsize <<= 1;
907 retry:
908         buf_hash_table.ht_mask = hsize - 1;
909         buf_hash_table.ht_table =
910             kmem_zalloc(hsize * sizeof (void*), KM_NOSLEEP);
911         if (buf_hash_table.ht_table == NULL) {
912                 ASSERT(hsize > (1ULL << 8));
913                 hsize >>= 1;
914                 goto retry;
915         }

917         hdr_cache = kmem_cache_create("arc_buf_hdr_t", sizeof (arc_buf_hdr_t),
918             0, hdr_cons, hdr_dest, hdr_recl, NULL, NULL, 0);
919         buf_cache = kmem_cache_create("arc_buf_t", sizeof (arc_buf_t),
920             0, buf_cons, buf_dest, NULL, NULL, NULL, 0);

922         for (i = 0; i < 256; i++)
923                 for (ct = zfs_crc64_table + i, *ct = i, j = 8; j > 0; j--)
924                         *ct = (*ct >> 1) ^ (-(*ct & 1) & ZFS_CRC64_POLY);

926         for (i = 0; i < BUF_LOCKS; i++) {
927                 mutex_init(&buf_hash_table.ht_locks[i].ht_lock,
928                     NULL, MUTEX_DEFAULT, NULL);
929         }
930 }

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

934 static void
935 arc_cksum_verify(arc_buf_t *buf)
936 {
937         zio_cksum_t zc;

939         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
940                 return;

942         mutex_enter(&buf->b_hdr->b_freeze_lock);
943         if (buf->b_hdr->b_freeze_cksum == NULL ||
944             (buf->b_hdr->b_flags & ARC_IO_ERROR)) {
945                 mutex_exit(&buf->b_hdr->b_freeze_lock);
946                 return;
```



```
947         }
948         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
949         if (!ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc))
950                 panic("buffer modified while frozen!");
951         mutex_exit(&buf->b_hdr->b_freeze_lock);
952 }

954 static int
955 arc_cksum_equal(arc_buf_t *buf)
956 {
957         zio_cksum_t zc;
958         int equal;

960         mutex_enter(&buf->b_hdr->b_freeze_lock);
961         fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
962         equal = ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc);
963         mutex_exit(&buf->b_hdr->b_freeze_lock);

965         return (equal);
966 }

968 static void
969 arc_cksum_compute(arc_buf_t *buf, boolean_t force)
970 {
971         if (!force && !(zfs_flags & ZFS_DEBUG_MODIFY))
972                 return;

974         mutex_enter(&buf->b_hdr->b_freeze_lock);
975         if (buf->b_hdr->b_freeze_cksum != NULL) {
976                 mutex_exit(&buf->b_hdr->b_freeze_lock);
977                 return;
978         }
979         buf->b_hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
980         fletcher_2_native(buf->b_data, buf->b_hdr->b_size,
981             buf->b_hdr->b_freeze_cksum);
982         mutex_exit(&buf->b_hdr->b_freeze_lock);
983         arc_buf_watch(buf);
984 }

986 #ifndef _KERNEL
987 typedef struct procctl {
988         long cmd;
989         prwatch_t prwatch;
990 } procctl_t;
991 #endif

993 /* ARGSUSED */
994 static void
995 arc_buf_unwatch(arc_buf_t *buf)
996 {
997 #ifndef _KERNEL
998         if (arc_watch) {
999                 int result;
1000                procctl_t ctl;
1001                ctl.cmd = PCWATCH;
1002                ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1003                ctl.prwatch.pr_size = 0;
1004                ctl.prwatch.pr_wflags = 0;
1005                result = write(arc_procfd, &ctl, sizeof (ctl));
1006                ASSERT3U(result, ==, sizeof (ctl));
1007        }
1008 #endif
1009 }

1011 /* ARGSUSED */
1012 static void
```

```
1013 arc_buf_watch(arc_buf_t *buf)
1014 {
1015 #ifndef _KERNEL
1016         if (arc_watch) {
1017                 int result;
1018                 procctl_t ctl;
1019                 ctl.cmd = PCWATCH;
1020                 ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
1021                 ctl.prwatch.pr_size = buf->b_hdr->b_size;
1022                 ctl.prwatch.pr_wflags = WA_WRITE;
1023                 result = write(arc_procfd, &ctl, sizeof (ctl));
1024                 ASSERT3U(result, ==, sizeof (ctl));
1025         }
1026 #endif
1027 }

1029 void
1030 arc_buf_thaw(arc_buf_t *buf)
1031 {
1032         if (zfs_flags & ZFS_DEBUG_MODIFY) {
1033                 if (buf->b_hdr->b_state != arc_anon)
1034                         panic("modifying non-anon buffer!");
1035                 if (buf->b_hdr->b_flags & ARC_IO_IN_PROGRESS)
1036                         panic("modifying buffer while i/o in progress!");
1037                 arc_cksum_verify(buf);
1038         }

1040         mutex_enter(&buf->b_hdr->b_freeze_lock);
1041         if (buf->b_hdr->b_freeze_cksum != NULL) {
1042                 kmem_free(buf->b_hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1043                 buf->b_hdr->b_freeze_cksum = NULL;
1044         }

1046         if (zfs_flags & ZFS_DEBUG_MODIFY) {
1047                 if (buf->b_hdr->b_thawed)
1048                         kmem_free(buf->b_hdr->b_thawed, 1);
1049                 buf->b_hdr->b_thawed = kmem_alloc(1, KM_SLEEP);
1050         }

1052         mutex_exit(&buf->b_hdr->b_freeze_lock);

1054         arc_buf_unwatch(buf);
1055 }

1057 void
1058 arc_buf_freeze(arc_buf_t *buf)
1059 {
1060         kmutex_t *hash_lock;

1062         if (!(zfs_flags & ZFS_DEBUG_MODIFY))
1063                 return;

1065         hash_lock = HDR_LOCK(buf->b_hdr);
1066         mutex_enter(hash_lock);

1068         ASSERT(buf->b_hdr->b_freeze_cksum != NULL ||
1069             buf->b_hdr->b_state == arc_anon);
1070         arc_cksum_compute(buf, B_FALSE);
1071         mutex_exit(hash_lock);

1073 }

1075 static void
1076 add_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
1077 {
1078         ASSERT(MUTEX_HELD(hash_lock));
```



```
1080         if ((refcount_add(&ab->b_refcnt, tag) == 1) &&
1081             (ab->b_state != arc_anon)) {
1082                 uint64_t delta = ab->b_size * ab->b_datacnt;
1083                 list_t *list = &ab->b_state->arcs_list[ab->b_type];
1084                 uint64_t *size = &ab->b_state->arcs_lsize[ab->b_type];

1086                 ASSERT(!MUTEX_HELD(&ab->b_state->arcs_mtx));
1087                 mutex_enter(&ab->b_state->arcs_mtx);
1088                 ASSERT(list_link_active(&ab->b_arc_node));
1089                 list_remove(list, ab);
1090                 if (GHOST_STATE(ab->b_state)) {
1091                         ASSERT0(ab->b_datacnt);
1092                         ASSERT3P(ab->b_buf, ==, NULL);
1093                         delta = ab->b_size;
1094                 }
1095                 ASSERT(delta > 0);
1096                 ASSERT3U(*size, >=, delta);
1097                 atomic_add_64(size, -delta);
1098                 mutex_exit(&ab->b_state->arcs_mtx);
1099                 /* remove the prefetch flag if we get a reference */
1100                 if (ab->b_flags & ARC_PREFETCH)
1101                         ab->b_flags &= ~ARC_PREFETCH;
1102         }
1103 }

1105 static int
1106 remove_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
1107 {
1108         int cnt;
1109         arc_state_t *state = ab->b_state;

1111         ASSERT(state == arc_anon || MUTEX_HELD(hash_lock));
1112         ASSERT(!GHOST_STATE(state));

1114         if (((cnt = refcount_remove(&ab->b_refcnt, tag)) == 0) &&
1115             (state != arc_anon)) {
1116                 uint64_t *size = &state->arcs_lsize[ab->b_type];

1118                 ASSERT(!MUTEX_HELD(&state->arcs_mtx));
1119                 mutex_enter(&state->arcs_mtx);
1120                 ASSERT(!list_link_active(&ab->b_arc_node));
1121                 list_insert_head(&state->arcs_list[ab->b_type], ab);
1122                 ASSERT(ab->b_datacnt > 0);
1123                 atomic_add_64(size, ab->b_size * ab->b_datacnt);
1124                 mutex_exit(&state->arcs_mtx);
1125         }
1126         return (cnt);
1127 }

1129 /*
1130  * Move the supplied buffer to the indicated state.  The mutex
1131  * for the buffer must be held by the caller.
1132  */
1133 static void
1134 arc_change_state(arc_state_t *new_state, arc_buf_hdr_t *ab, kmutex_t *hash_lock)
1135 {
1136         arc_state_t *old_state = ab->b_state;
1137         int64_t refcnt = refcount_count(&ab->b_refcnt);
1138         uint64_t from_delta, to_delta;

1140         ASSERT(MUTEX_HELD(hash_lock));
1141         ASSERT(new_state != old_state);
1142         ASSERT(refcnt == 0 || ab->b_datacnt > 0);
1143         ASSERT(ab->b_datacnt == 0 || !GHOST_STATE(new_state));
1144         ASSERT(ab->b_datacnt <= 1 || old_state != arc_anon);
```

```
1146         from_delta = to_delta = ab->b_datacnt * ab->b_size;

1148         /*
1149          * If this buffer is evictable, transfer it from the
1150          * old state list to the new state list.
1151          */
1152         if (refcnt == 0) {
1153                 if (old_state != arc_anon) {
1154                         int use_mutex = !MUTEX_HELD(&old_state->arcs_mtx);
1155                         uint64_t *size = &old_state->arcs_lsize[ab->b_type];

1157                         if (use_mutex)
1158                                 mutex_enter(&old_state->arcs_mtx);

1160                         ASSERT(list_link_active(&ab->b_arc_node));
1161                         list_remove(&old_state->arcs_list[ab->b_type], ab);

1163                         /*
1164                          * If prefetching out of the ghost cache,
1165                          * we will have a non-zero datacnt.
1166                          */
1167                         if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
1168                                 /* ghost elements have a ghost size */
1169                                 ASSERT(ab->b_buf == NULL);
1170                                 from_delta = ab->b_size;
1171                         }
1172                         ASSERT3U(*size, >=, from_delta);
1173                         atomic_add_64(size, -from_delta);

1175                         if (use_mutex)
1176                                 mutex_exit(&old_state->arcs_mtx);
1177                 }
1178                 if (new_state != arc_anon) {
1179                         int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
1180                         uint64_t *size = &new_state->arcs_lsize[ab->b_type];

1182                         if (use_mutex)
1183                                 mutex_enter(&new_state->arcs_mtx);

1185                         list_insert_head(&new_state->arcs_list[ab->b_type], ab);

1187                         /* ghost elements have a ghost size */
1188                         if (GHOST_STATE(new_state)) {
1189                                 ASSERT(ab->b_datacnt == 0);
1190                                 ASSERT(ab->b_buf == NULL);
1191                                 to_delta = ab->b_size;
1192                         }
1193                         atomic_add_64(size, to_delta);

1195                         if (use_mutex)
1196                                 mutex_exit(&new_state->arcs_mtx);
1197                 }
1198         }

1200         ASSERT(!BUF_EMPTY(ab));
1201         if (new_state == arc_anon && HDR_IN_HASH_TABLE(ab))
1202                 buf_hash_remove(ab);

1204         /* adjust state sizes */
1205         if (to_delta)
1206                 atomic_add_64(&new_state->arcs_size, to_delta);
1207         if (from_delta) {
1208                 ASSERT3U(old_state->arcs_size, >=, from_delta);
1209                 atomic_add_64(&old_state->arcs_size, -from_delta);
1210         }
```



```
1211         ab->b_state = new_state;

1213         /* adjust l2arc hdr stats */
1214         if (new_state == arc_l2c_only)
1215                 l2arc_hdr_stat_add();
1216         else if (old_state == arc_l2c_only)
1217                 l2arc_hdr_stat_remove();
1218 }

1220 void
1221 arc_space_consume(uint64_t space, arc_space_type_t type)
1222 {
1223         ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);

1225         switch (type) {
1226         case ARC_SPACE_DATA:
1227                 ARCSTAT_INCR(arcstat_data_size, space);
1228                 break;
1229         case ARC_SPACE_OTHER:
1230                 ARCSTAT_INCR(arcstat_other_size, space);
1231                 break;
1232         case ARC_SPACE_HDRS:
1233                 ARCSTAT_INCR(arcstat_hdr_size, space);
1234                 break;
1235         case ARC_SPACE_L2HDRS:
1236                 ARCSTAT_INCR(arcstat_l2_hdr_size, space);
1237                 break;
1238         }

1240         ARCSTAT_INCR(arcstat_meta_used, space);
1241         atomic_add_64(&arc_size, space);
1242 }

1244 void
1245 arc_space_return(uint64_t space, arc_space_type_t type)
1246 {
1247         ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);

1249         switch (type) {
1250         case ARC_SPACE_DATA:
1251                 ARCSTAT_INCR(arcstat_data_size, -space);
1252                 break;
1253         case ARC_SPACE_OTHER:
1254                 ARCSTAT_INCR(arcstat_other_size, -space);
1255                 break;
1256         case ARC_SPACE_HDRS:
1257                 ARCSTAT_INCR(arcstat_hdr_size, -space);
1258                 break;
1259         case ARC_SPACE_L2HDRS:
1260                 ARCSTAT_INCR(arcstat_l2_hdr_size, -space);
1261                 break;
1262         }

1264         ASSERT(arc_meta_used >= space);
1265         if (arc_meta_max < arc_meta_used)
1266                 arc_meta_max = arc_meta_used;
1267         ARCSTAT_INCR(arcstat_meta_used, -space);
1268         ASSERT(arc_size >= space);
1269         atomic_add_64(&arc_size, -space);
1270 }

1272 void *
1273 arc_data_buf_alloc(uint64_t size)
1274 {
1275         if (arc_evict_needed(ARC_BUFC_DATA))
1276                 cv_signal(&arc_reclaim_thr_cv);
```

```
1277         atomic_add_64(&arc_size, size);
1278         return (zio_data_buf_alloc(size));
1279 }

1281 void
1282 arc_data_buf_free(void *buf, uint64_t size)
1283 {
1284         zio_data_buf_free(buf, size);
1285         ASSERT(arc_size >= size);
1286         atomic_add_64(&arc_size, -size);
1287 }

1289 arc_buf_t *
1290 arc_buf_alloc(spa_t *spa, int size, void *tag, arc_buf_contents_t type)
1291 {
1292         arc_buf_hdr_t *hdr;
1293         arc_buf_t *buf;

1295         ASSERT3U(size, >, 0);
1296         hdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
1297         ASSERT(BUF_EMPTY(hdr));
1298         hdr->b_size = size;
1299         hdr->b_type = type;
1300         hdr->b_spa = spa_load_guid(spa);
1301         hdr->b_state = arc_anon;
1302         hdr->b_arc_access = 0;
1303         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1304         buf->b_hdr = hdr;
1305         buf->b_data = NULL;
1306         buf->b_efunc = NULL;
1307         buf->b_private = NULL;
1308         buf->b_next = NULL;
1309         hdr->b_buf = buf;
1310         arc_get_data_buf(buf);
1311         hdr->b_datacnt = 1;
1312         hdr->b_flags = 0;
1313         ASSERT(refcount_is_zero(&hdr->b_refcnt));
1314         (void) refcount_add(&hdr->b_refcnt, tag);

1316         return (buf);
1317 }

1319 static char *arc_onloan_tag = "onloan";

1321 /*
1322  * Loan out an anonymous arc buffer. Loaned buffers are not counted as in
1323  * flight data by arc_tempreserve_space() until they are "returned". Loaned
1324  * buffers must be returned to the arc before they can be used by the DMU or
1325  * freed.
1326  */
1327 arc_buf_t *
1328 arc_loan_buf(spa_t *spa, int size)
1329 {
1330         arc_buf_t *buf;

1332         buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);

1334         atomic_add_64(&arc_loaned_bytes, size);
1335         return (buf);
1336 }

1338 /*
1339  * Return a loaned arc buffer to the arc.
1340  */
1341 void
1342 arc_return_buf(arc_buf_t *buf, void *tag)
```



```
1343 {
1344         arc_buf_hdr_t *hdr = buf->b_hdr;

1346         ASSERT(buf->b_data != NULL);
1347         (void) refcount_add(&hdr->b_refcnt, tag);
1348         (void) refcount_remove(&hdr->b_refcnt, arc_onloan_tag);

1350         atomic_add_64(&arc_loaned_bytes, -hdr->b_size);
1351 }

1353 /* Detach an arc_buf from a dbuf (tag) */
1354 void
1355 arc_loan_inuse_buf(arc_buf_t *buf, void *tag)
1356 {
1357         arc_buf_hdr_t *hdr;

1359         ASSERT(buf->b_data != NULL);
1360         hdr = buf->b_hdr;
1361         (void) refcount_add(&hdr->b_refcnt, arc_onloan_tag);
1362         (void) refcount_remove(&hdr->b_refcnt, tag);
1363         buf->b_efunc = NULL;
1364         buf->b_private = NULL;

1366         atomic_add_64(&arc_loaned_bytes, hdr->b_size);
1367 }

1369 static arc_buf_t *
1370 arc_buf_clone(arc_buf_t *from)
1371 {
1372         arc_buf_t *buf;
1373         arc_buf_hdr_t *hdr = from->b_hdr;
1374         uint64_t size = hdr->b_size;

1376         ASSERT(hdr->b_state != arc_anon);

1378         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
1379         buf->b_hdr = hdr;
1380         buf->b_data = NULL;
1381         buf->b_efunc = NULL;
1382         buf->b_private = NULL;
1383         buf->b_next = hdr->b_buf;
1384         hdr->b_buf = buf;
1385         arc_get_data_buf(buf);
1386         bcopy(from->b_data, buf->b_data, size);

1388         /*
1389          * This buffer already exists in the arc so create a duplicate
1390          * copy for the caller.  If the buffer is associated with user data
1391          * then track the size and number of duplicates.  These stats will be
1392          * updated as duplicate buffers are created and destroyed.
1393          */
1394         if (hdr->b_type == ARC_BUFC_DATA) {
1395                 ARCSTAT_BUMP(arcstat_duplicate_buffers);
1396                 ARCSTAT_INCR(arcstat_duplicate_buffers_size, size);
1397         }
1398         hdr->b_datacnt += 1;
1399         return (buf);
1400 }

1402 void
1403 arc_buf_add_ref(arc_buf_t *buf, void* tag)
1404 {
1405         arc_buf_hdr_t *hdr;
1406         kmutex_t *hash_lock;

1408         /*
```

```
1409          * Check to see if this buffer is evicted.  Callers
1410          * must verify b_data != NULL to know if the add_ref
1411          * was successful.
1412          */
1413         mutex_enter(&buf->b_evict_lock);
1414         if (buf->b_data == NULL) {
1415                 mutex_exit(&buf->b_evict_lock);
1416                 return;
1417         }
1418         hash_lock = HDR_LOCK(buf->b_hdr);
1419         mutex_enter(hash_lock);
1420         hdr = buf->b_hdr;
1421         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
1422         mutex_exit(&buf->b_evict_lock);

1424         ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);
1425         add_reference(hdr, hash_lock, tag);
1426         DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
1427         arc_access(hdr, hash_lock);
1428         mutex_exit(hash_lock);
1429         ARCSTAT_BUMP(arcstat_hits);
1430         ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
1431             demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
1432             data, metadata, hits);
1433 }

1435 /*
1436  * Free the arc data buffer.  If it is an l2arc write in progress,
1437  * the buffer is placed on l2arc_free_on_write to be freed later.
1438  */
1439 static void
1440 arc_buf_data_free(arc_buf_t *buf, void (*free_func)(void *, size_t))
1441 {
1442         arc_buf_hdr_t *hdr = buf->b_hdr;

1444         if (HDR_L2_WRITING(hdr)) {
1445                 l2arc_data_free_t *df;
1446                 df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
1447                 df->l2df_data = buf->b_data;
1448                 df->l2df_size = hdr->b_size;
1449                 df->l2df_func = free_func;
1450                 mutex_enter(&l2arc_free_on_write_mtx);
1451                 list_insert_head(l2arc_free_on_write, df);
1452                 mutex_exit(&l2arc_free_on_write_mtx);
1453                 ARCSTAT_BUMP(arcstat_l2_free_on_write);
1454         } else {
1455                 free_func(buf->b_data, hdr->b_size);
1456         }
1457 }

1459 static void
1460 arc_buf_destroy(arc_buf_t *buf, boolean_t recycle, boolean_t all)
1461 {
1462         arc_buf_t **bufp;

1464         /* free up data associated with the buf */
1465         if (buf->b_data) {
1466                 arc_state_t *state = buf->b_hdr->b_state;
1467                 uint64_t size = buf->b_hdr->b_size;
1468                 arc_buf_contents_t type = buf->b_hdr->b_type;

1470                 arc_cksum_verify(buf);
1471                 arc_buf_unwatch(buf);

1473                 if (!recycle) {
1474                         if (type == ARC_BUFC_METADATA) {
```



```
1475                                 arc_buf_data_free(buf, zio_buf_free);
1476                                 arc_space_return(size, ARC_SPACE_DATA);
1477                         } else {
1478                                 ASSERT(type == ARC_BUFC_DATA);
1479                                 arc_buf_data_free(buf, zio_data_buf_free);
1480                                 ARCSTAT_INCR(arcstat_data_size, -size);
1481                                 atomic_add_64(&arc_size, -size);
1482                         }
1483                 }
1484                 if (list_link_active(&buf->b_hdr->b_arc_node)) {
1485                         uint64_t *cnt = &state->arcs_lsize[type];

1487                         ASSERT(refcount_is_zero(&buf->b_hdr->b_refcnt));
1488                         ASSERT(state != arc_anon);

1490                         ASSERT3U(*cnt, >=, size);
1491                         atomic_add_64(cnt, -size);
1492                 }
1493                 ASSERT3U(state->arcs_size, >=, size);
1494                 atomic_add_64(&state->arcs_size, -size);
1495                 buf->b_data = NULL;

1497                 /*
1498                  * If we're destroying a duplicate buffer make sure
1499                  * that the appropriate statistics are updated.
1500                  */
1501                 if (buf->b_hdr->b_datacnt > 1 &&
1502                     buf->b_hdr->b_type == ARC_BUFC_DATA) {
1503                         ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
1504                         ARCSTAT_INCR(arcstat_duplicate_buffers_size, -size);
1505                 }
1506                 ASSERT(buf->b_hdr->b_datacnt > 0);
1507                 buf->b_hdr->b_datacnt -= 1;
1508         }

1510         /* only remove the buf if requested */
1511         if (!all)
1512                 return;

1514         /* remove the buf from the hdr list */
1515         for (bufp = &buf->b_hdr->b_buf; *bufp != buf; bufp = &(*bufp)->b_next)
1516                 continue;
1517         *bufp = buf->b_next;
1518         buf->b_next = NULL;

1520         ASSERT(buf->b_efunc == NULL);

1522         /* clean up the buf */
1523         buf->b_hdr = NULL;
1524         kmem_cache_free(buf_cache, buf);
1525 }

1527 static void
1528 arc_hdr_destroy(arc_buf_hdr_t *hdr)
1529 {
1530         ASSERT(refcount_is_zero(&hdr->b_refcnt));
1531         ASSERT3P(hdr->b_state, ==, arc_anon);
1532         ASSERT(!HDR_IO_IN_PROGRESS(hdr));
1533         l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;

1535         if (l2hdr != NULL) {
1536                 boolean_t buflist_held = MUTEX_HELD(&l2arc_buflist_mtx);
1537                 /*
1538                  * To prevent arc_free() and l2arc_evict() from
1539                  * attempting to free the same buffer at the same time,
1540                  * a FREE_IN_PROGRESS flag is given to arc_free() to
```

```
1541                 * give it priority.  l2arc_evict() can't destroy this
1542                 * header while we are waiting on l2arc_buflist_mtx.
1543                 *
1544                 * The hdr may be removed from l2ad_buflist before we
1545                 * grab l2arc_buflist_mtx, so b_l2hdr is rechecked.
1546                 */
1547                if (!buflist_held) {
1548                        mutex_enter(&l2arc_buflist_mtx);
1549                        l2hdr = hdr->b_l2hdr;
1550                }

1552                if (l2hdr != NULL) {
1553                        list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
1554                        ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);
1555                        kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
1556                        if (hdr->b_state == arc_l2c_only)
1557                                l2arc_hdr_stat_remove();
1558                        hdr->b_l2hdr = NULL;
1559                }

1561                if (!buflist_held)
1562                        mutex_exit(&l2arc_buflist_mtx);
1563        }

1565        if (!BUF_EMPTY(hdr)) {
1566                ASSERT(!HDR_IN_HASH_TABLE(hdr));
1567                buf_discard_identity(hdr);
1568        }
1569        while (hdr->b_buf) {
1570                arc_buf_t *buf = hdr->b_buf;

1572                if (buf->b_efunc) {
1573                        mutex_enter(&arc_eviction_mtx);
1574                        mutex_enter(&buf->b_evict_lock);
1575                        ASSERT(buf->b_hdr != NULL);
1576                        arc_buf_destroy(hdr->b_buf, FALSE, FALSE);
1577                        hdr->b_buf = buf->b_next;
1578                        buf->b_hdr = &arc_eviction_hdr;
1579                        buf->b_next = arc_eviction_list;
1580                        arc_eviction_list = buf;
1581                        mutex_exit(&buf->b_evict_lock);
1582                        mutex_exit(&arc_eviction_mtx);
1583                } else {
1584                        arc_buf_destroy(hdr->b_buf, FALSE, TRUE);
1585                }
1586        }
1587        if (hdr->b_freeze_cksum != NULL) {
1588                kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
1589                hdr->b_freeze_cksum = NULL;
1590        }
1591        if (hdr->b_thawed) {
1592                kmem_free(hdr->b_thawed, 1);
1593                hdr->b_thawed = NULL;
1594        }

1596        ASSERT(!list_link_active(&hdr->b_arc_node));
1597        ASSERT3P(hdr->b_hash_next, ==, NULL);
1598        ASSERT3P(hdr->b_acb, ==, NULL);
1599        kmem_cache_free(hdr_cache, hdr);
1600 }

1602 void
1603 arc_buf_free(arc_buf_t *buf, void *tag)
1604 {
1605        arc_buf_hdr_t *hdr = buf->b_hdr;
1606        int hashed = hdr->b_state != arc_anon;
```



```
1608        ASSERT(buf->b_efunc == NULL);
1609        ASSERT(buf->b_data != NULL);

1611        if (hashed) {
1612                kmutex_t *hash_lock = HDR_LOCK(hdr);

1614                mutex_enter(hash_lock);
1615                hdr = buf->b_hdr;
1616                ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

1618                (void) remove_reference(hdr, hash_lock, tag);
1619                if (hdr->b_datacnt > 1) {
1620                        arc_buf_destroy(buf, FALSE, TRUE);
1621                } else {
1622                        ASSERT(buf == hdr->b_buf);
1623                        ASSERT(buf->b_efunc == NULL);
1624                        hdr->b_flags |= ARC_BUF_AVAILABLE;
1625                }
1626                mutex_exit(hash_lock);
1627        } else if (HDR_IO_IN_PROGRESS(hdr)) {
1628                int destroy_hdr;
1629                /*
1630                 * We are in the middle of an async write.  Don't destroy
1631                 * this buffer unless the write completes before we finish
1632                 * decrementing the reference count.
1633                 */
1634                mutex_enter(&arc_eviction_mtx);
1635                (void) remove_reference(hdr, NULL, tag);
1636                ASSERT(refcount_is_zero(&hdr->b_refcnt));
1637                destroy_hdr = !HDR_IO_IN_PROGRESS(hdr);
1638                mutex_exit(&arc_eviction_mtx);
1639                if (destroy_hdr)
1640                        arc_hdr_destroy(hdr);
1641        } else {
1642                if (remove_reference(hdr, NULL, tag) > 0)
1643                        arc_buf_destroy(buf, FALSE, TRUE);
1644                else
1645                        arc_hdr_destroy(hdr);
1646        }
1647 }

1649 boolean_t
1650 arc_buf_remove_ref(arc_buf_t *buf, void* tag)
1651 {
1652        arc_buf_hdr_t *hdr = buf->b_hdr;
1653        kmutex_t *hash_lock = HDR_LOCK(hdr);
1654        boolean_t no_callback = (buf->b_efunc == NULL);

1656        if (hdr->b_state == arc_anon) {
1657                ASSERT(hdr->b_datacnt == 1);
1658                arc_buf_free(buf, tag);
1659                return (no_callback);
1660        }

1662        mutex_enter(hash_lock);
1663        hdr = buf->b_hdr;
1664        ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
1665        ASSERT(hdr->b_state != arc_anon);
1666        ASSERT(buf->b_data != NULL);

1668        (void) remove_reference(hdr, hash_lock, tag);
1669        if (hdr->b_datacnt > 1) {
1670                if (no_callback)
1671                        arc_buf_destroy(buf, FALSE, TRUE);
1672        } else if (no_callback) {
```

```
1673                 ASSERT(hdr->b_buf == buf && buf->b_next == NULL);
1674                 ASSERT(buf->b_efunc == NULL);
1675                 hdr->b_flags |= ARC_BUF_AVAILABLE;
1676         }
1677         ASSERT(no_callback || hdr->b_datacnt > 1 ||
1678             refcount_is_zero(&hdr->b_refcnt));
1679         mutex_exit(hash_lock);
1680         return (no_callback);
1681 }

1683 int
1684 arc_buf_size(arc_buf_t *buf)
1685 {
1686         return (buf->b_hdr->b_size);
1687 }

1689 /*
1690  * Called from the DMU to determine if the current buffer should be
1691  * evicted. In order to ensure proper locking, the eviction must be initiated
1692  * from the DMU. Return true if the buffer is associated with user data and
1693  * duplicate buffers still exist.
1694  */
1695 boolean_t
1696 arc_buf_eviction_needed(arc_buf_t *buf)
1697 {
1698         arc_buf_hdr_t *hdr;
1699         boolean_t evict_needed = B_FALSE;

1701         if (zfs_disable_dup_eviction)
1702                 return (B_FALSE);

1704         mutex_enter(&buf->b_evict_lock);
1705         hdr = buf->b_hdr;
1706         if (hdr == NULL) {
1707                 /*
1708                  * We are in arc_do_user_evicts(); let that function
1709                  * perform the eviction.
1710                  */
1711                 ASSERT(buf->b_data == NULL);
1712                 mutex_exit(&buf->b_evict_lock);
1713                 return (B_FALSE);
1714         } else if (buf->b_data == NULL) {
1715                 /*
1716                  * We have already been added to the arc eviction list;
1717                  * recommend eviction.
1718                  */
1719                 ASSERT3P(hdr, ==, &arc_eviction_hdr);
1720                 mutex_exit(&buf->b_evict_lock);
1721                 return (B_TRUE);
1722         }

1724         if (hdr->b_datacnt > 1 && hdr->b_type == ARC_BUFC_DATA)
1725                 evict_needed = B_TRUE;

1727         mutex_exit(&buf->b_evict_lock);
1728         return (evict_needed);
1729 }

1731 /*
1732  * Evict buffers from list until we've removed the specified number of
1733  * bytes.  Move the removed buffers to the appropriate evict state.
1734  * If the recycle flag is set, then attempt to "recycle" a buffer:
1735  * - look for a buffer to evict that is `bytes' long.
1736  * - return the data block from this buffer rather than freeing it.
1737  * This flag is used by callers that are trying to make space for a
1738  * new buffer in a full arc cache.
```



```
1739  *
1740  * This function makes a "best effort".  It skips over any buffers
1741  * it can't get a hash_lock on, and so may not catch all candidates.
1742  * It may also return without evicting as much space as requested.
1743  */
1744 static void *
1745 arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle,
1746     arc_buf_contents_t type)
1747 {
1748         arc_state_t *evicted_state;
1749         uint64_t bytes_evicted = 0, skipped = 0, missed = 0;
1750         arc_buf_hdr_t *ab, *ab_prev = NULL;
1751         list_t *list = &state->arcs_list[type];
1752         kmutex_t *hash_lock;
1753         boolean_t have_lock;
1754         void *stolen = NULL;

1756         ASSERT(state == arc_mru || state == arc_mfu);

1758         evicted_state = (state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;

1760         mutex_enter(&state->arcs_mtx);
1761         mutex_enter(&evicted_state->arcs_mtx);

1763         for (ab = list_tail(list); ab; ab = ab_prev) {
1764                 ab_prev = list_prev(list, ab);
1765                 /* prefetch buffers have a minimum lifespan */
1766                 if (HDR_IO_IN_PROGRESS(ab) ||
1767                     (spa && ab->b_spa != spa) ||
1768                     (ab->b_flags & (ARC_PREFETCH|ARC_INDIRECT) &&
1769                     ddi_get_lbolt() - ab->b_arc_access <
1770                     arc_min_prefetch_lifespan)) {
1771                         skipped++;
1772                         continue;
1773                 }
1774                 /* "lookahead" for better eviction candidate */
1775                 if (recycle && ab->b_size != bytes &&
1776                     ab_prev && ab_prev->b_size == bytes)
1777                         continue;
1778                 hash_lock = HDR_LOCK(ab);
1779                 have_lock = MUTEX_HELD(hash_lock);
1780                 if (have_lock || mutex_tryenter(hash_lock)) {
1781                         ASSERT0(refcount_count(&ab->b_refcnt));
1782                         ASSERT(ab->b_datacnt > 0);
1783                         while (ab->b_buf) {
1784                                 arc_buf_t *buf = ab->b_buf;
1785                                 if (!mutex_tryenter(&buf->b_evict_lock)) {
1786                                         missed += 1;
1787                                         break;
1788                                 }
1789                                 if (buf->b_data) {
1790                                         bytes_evicted += ab->b_size;
1791                                         if (recycle && ab->b_type == type &&
1792                                             ab->b_size == bytes &&
1793                                             !HDR_L2_WRITING(ab)) {
1794                                                 stolen = buf->b_data;
1795                                                 recycle = FALSE;
1796                                         }
1797                                 }
1798                                 if (buf->b_efunc) {
1799                                         mutex_enter(&arc_eviction_mtx);
1800                                         arc_buf_destroy(buf,
1801                                             buf->b_data == stolen, FALSE);
1802                                         ab->b_buf = buf->b_next;
1803                                         buf->b_hdr = &arc_eviction_hdr;
1804                                         buf->b_next = arc_eviction_list;
```

```
1805                                    arc_eviction_list = buf;
1806                                    mutex_exit(&arc_eviction_mtx);
1807                                    mutex_exit(&buf->b_evict_lock);
1808                            } else {
1809                                    mutex_exit(&buf->b_evict_lock);
1810                                    arc_buf_destroy(buf,
1811                                        buf->b_data == stolen, TRUE);
1812                            }
1813                    }

1815                    if (ab->b_l2hdr) {
1816                            ARCSTAT_INCR(arcstat_evict_l2_cached,
1817                                ab->b_size);
1818                    } else {
1819                            if (l2arc_write_eligible(ab->b_spa, ab)) {
1820                                    ARCSTAT_INCR(arcstat_evict_l2_eligible,
1821                                        ab->b_size);
1822                            } else {
1823                                    ARCSTAT_INCR(
1824                                        arcstat_evict_l2_ineligible,
1825                                        ab->b_size);
1826                            }
1827                    }

1829                    if (ab->b_datacnt == 0) {
1830                            arc_change_state(evicted_state, ab, hash_lock);
1831                            ASSERT(HDR_IN_HASH_TABLE(ab));
1832                            ab->b_flags |= ARC_IN_HASH_TABLE;
1833                            ab->b_flags &= ~ARC_BUF_AVAILABLE;
1834                            DTRACE_PROBE1(arc__evict, arc_buf_hdr_t *, ab);
1835                    }
1836                    if (!have_lock)
1837                            mutex_exit(hash_lock);
1838                    if (bytes >= 0 && bytes_evicted >= bytes)
1839                            break;
1840            } else {
1841                    missed += 1;
1842            }
1843    }

1845    mutex_exit(&evicted_state->arcs_mtx);
1846    mutex_exit(&state->arcs_mtx);

1848    if (bytes_evicted < bytes)
1849            dprintf("only evicted %lld bytes from %x",
1850                (longlong_t)bytes_evicted, state);

1852    if (skipped)
1853            ARCSTAT_INCR(arcstat_evict_skip, skipped);

1855    if (missed)
1856            ARCSTAT_INCR(arcstat_mutex_miss, missed);

1858    /*
1859     * We have just evicted some data into the ghost state, make
1860     * sure we also adjust the ghost state size if necessary.
1861     */
1862    if (arc_no_grow &&
1863        arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size > arc_c) {
1864            int64_t mru_over = arc_anon->arcs_size + arc_mru->arcs_size +
1865                arc_mru_ghost->arcs_size - arc_c;

1867            if (mru_over > 0 && arc_mru_ghost->arcs_lsize[type] > 0) {
1868                    int64_t todelete =
1869                        MIN(arc_mru_ghost->arcs_lsize[type], mru_over);
1870                    arc_evict_ghost(arc_mru_ghost, NULL, todelete);
```



```
1871            } else if (arc_mfu_ghost->arcs_lsize[type] > 0) {
1872                    int64_t todelete = MIN(arc_mfu_ghost->arcs_lsize[type],
1873                        arc_mru_ghost->arcs_size +
1874                        arc_mfu_ghost->arcs_size - arc_c);
1875                    arc_evict_ghost(arc_mfu_ghost, NULL, todelete);
1876            }
1877    }

1879    return (stolen);
1880 }

1882 /*
1883  * Remove buffers from list until we've removed the specified number of
1884  * bytes.  Destroy the buffers that are removed.
1885  */
1886 static void
1887 arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes)
1888 {
1889    arc_buf_hdr_t *ab, *ab_prev;
1890    arc_buf_hdr_t marker = { 0 };
1891    list_t *list = &state->arcs_list[ARC_BUFC_DATA];
1892    kmutex_t *hash_lock;
1893    uint64_t bytes_deleted = 0;
1894    uint64_t bufs_skipped = 0;

1896    ASSERT(GHOST_STATE(state));
1897 top:
1898    mutex_enter(&state->arcs_mtx);
1899    for (ab = list_tail(list); ab; ab = ab_prev) {
1900            ab_prev = list_prev(list, ab);
1901            if (spa && ab->b_spa != spa)
1902                    continue;

1904            /* ignore markers */
1905            if (ab->b_spa == 0)
1906                    continue;

1908            hash_lock = HDR_LOCK(ab);
1909            /* caller may be trying to modify this buffer, skip it */
1910            if (MUTEX_HELD(hash_lock))
1911                    continue;
1912            if (mutex_tryenter(hash_lock)) {
1913                    ASSERT(!HDR_IO_IN_PROGRESS(ab));
1914                    ASSERT(ab->b_buf == NULL);
1915                    ARCSTAT_BUMP(arcstat_deleted);
1916                    bytes_deleted += ab->b_size;

1918                    if (ab->b_l2hdr != NULL) {
1919                            /*
1920                             * This buffer is cached on the 2nd Level ARC;
1921                             * don't destroy the header.
1922                             */
1923                            arc_change_state(arc_l2c_only, ab, hash_lock);
1924                            mutex_exit(hash_lock);
1925                    } else {
1926                            arc_change_state(arc_anon, ab, hash_lock);
1927                            mutex_exit(hash_lock);
1928                            arc_hdr_destroy(ab);
1929                    }

1931                    DTRACE_PROBE1(arc__delete, arc_buf_hdr_t *, ab);
1932                    if (bytes >= 0 && bytes_deleted >= bytes)
1933                            break;
1934            } else if (bytes < 0) {
1935                    /*
1936                     * Insert a list marker and then wait for the
```

```
1937                         * hash lock to become available. Once its
1938                         * available, restart from where we left off.
1939                         */
1940                        list_insert_after(list, ab, &marker);
1941                        mutex_exit(&state->arcs_mtx);
1942                        mutex_enter(hash_lock);
1943                        mutex_exit(hash_lock);
1944                        mutex_enter(&state->arcs_mtx);
1945                        ab_prev = list_prev(list, &marker);
1946                        list_remove(list, &marker);
1947                } else
1948                        bufs_skipped += 1;
1949        }
1950        mutex_exit(&state->arcs_mtx);

1952        if (list == &state->arcs_list[ARC_BUFC_DATA] &&
1953            (bytes < 0 || bytes_deleted < bytes)) {
1954                list = &state->arcs_list[ARC_BUFC_METADATA];
1955                goto top;
1956        }

1958        if (bufs_skipped) {
1959                ARCSTAT_INCR(arcstat_mutex_miss, bufs_skipped);
1960                ASSERT(bytes >= 0);
1961        }

1963        if (bytes_deleted < bytes)
1964                dprintf("only deleted %lld bytes from %p",
1965                    (longlong_t)bytes_deleted, state);
1966 }

1968 static void
1969 arc_adjust(void)
1970 {
1971        int64_t adjustment, delta;

1973        /*
1974         * Adjust MRU size
1975         */

1977        adjustment = MIN((int64_t)(arc_size - arc_c),
1978            (int64_t)(arc_anon->arcs_size + arc_mru->arcs_size + arc_meta_used -
1979            arc_p));

1981        if (adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_DATA] > 0) {
1982                delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_DATA], adjustment);
1983                (void) arc_evict(arc_mru, NULL, delta, FALSE, ARC_BUFC_DATA);
1984                adjustment -= delta;
1985        }

1987        if (adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_METADATA] > 0) {
1988                delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_METADATA], adjustment);
1989                (void) arc_evict(arc_mru, NULL, delta, FALSE,
1990                    ARC_BUFC_METADATA);
1991        }

1993        /*
1994         * Adjust MFU size
1995         */

1997        adjustment = arc_size - arc_c;

1999        if (adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_DATA] > 0) {
2000                delta = MIN(adjustment, arc_mfu->arcs_lsize[ARC_BUFC_DATA]);
2001                (void) arc_evict(arc_mfu, NULL, delta, FALSE, ARC_BUFC_DATA);
2002                adjustment -= delta;
```



```
2003        }

2005        if (adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_METADATA] > 0) {
2006                int64_t delta = MIN(adjustment,
2007                    arc_mfu->arcs_lsize[ARC_BUFC_METADATA]);
2008                (void) arc_evict(arc_mfu, NULL, delta, FALSE,
2009                    ARC_BUFC_METADATA);
2010        }

2012        /*
2013         * Adjust ghost lists
2014         */

2016        adjustment = arc_mru->arcs_size + arc_mru_ghost->arcs_size - arc_c;

2018        if (adjustment > 0 && arc_mru_ghost->arcs_size > 0) {
2019                delta = MIN(arc_mru_ghost->arcs_size, adjustment);
2020                arc_evict_ghost(arc_mru_ghost, NULL, delta);
2021        }

2023        adjustment =
2024            arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size - arc_c;

2026        if (adjustment > 0 && arc_mfu_ghost->arcs_size > 0) {
2027                delta = MIN(arc_mfu_ghost->arcs_size, adjustment);
2028                arc_evict_ghost(arc_mfu_ghost, NULL, delta);
2029        }
2030 }

2032 static void
2033 arc_do_user_evicts(void)
2034 {
2035        mutex_enter(&arc_eviction_mtx);
2036        while (arc_eviction_list != NULL) {
2037                arc_buf_t *buf = arc_eviction_list;
2038                arc_eviction_list = buf->b_next;
2039                mutex_enter(&buf->b_evict_lock);
2040                buf->b_hdr = NULL;
2041                mutex_exit(&buf->b_evict_lock);
2042                mutex_exit(&arc_eviction_mtx);

2044                if (buf->b_efunc != NULL)
2045                        VERIFY(buf->b_efunc(buf) == 0);

2047                buf->b_efunc = NULL;
2048                buf->b_private = NULL;
2049                kmem_cache_free(buf_cache, buf);
2050                mutex_enter(&arc_eviction_mtx);
2051        }
2052        mutex_exit(&arc_eviction_mtx);
2053 }

2055 /*
2056  * Flush all *evictable* data from the cache for the given spa.
2057  * NOTE: this will not touch "active" (i.e. referenced) data.
2058  */
2059 void
2060 arc_flush(spa_t *spa)
2061 {
2062        uint64_t guid = 0;

2064        if (spa)
2065                guid = spa_load_guid(spa);

2067        while (list_head(&arc_mru->arcs_list[ARC_BUFC_DATA])) {
2068                (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_DATA);
```

```
2069                 if (spa)
2070                         break;
2071         }
2072         while (list_head(&arc_mru->arcs_list[ARC_BUFC_METADATA])) {
2073                 (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_METADATA);
2074                 if (spa)
2075                         break;
2076         }
2077         while (list_head(&arc_mfu->arcs_list[ARC_BUFC_DATA])) {
2078                 (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_DATA);
2079                 if (spa)
2080                         break;
2081         }
2082         while (list_head(&arc_mfu->arcs_list[ARC_BUFC_METADATA])) {
2083                 (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_METADATA);
2084                 if (spa)
2085                         break;
2086         }

2088         arc_evict_ghost(arc_mru_ghost, guid, -1);
2089         arc_evict_ghost(arc_mfu_ghost, guid, -1);

2091         mutex_enter(&arc_reclaim_thr_lock);
2092         arc_do_user_evicts();
2093         mutex_exit(&arc_reclaim_thr_lock);
2094         ASSERT(spa || arc_eviction_list == NULL);
2095 }

2097 void
2098 arc_shrink(void)
2099 {
2100         if (arc_c > arc_c_min) {
2101                 uint64_t to_free;

2103 #ifdef _KERNEL
2104                 to_free = MAX(arc_c >> arc_shrink_shift, ptob(needfree));
2105 #else
2106                 to_free = arc_c >> arc_shrink_shift;
2107 #endif
2108                 if (arc_c > arc_c_min + to_free)
2109                         atomic_add_64(&arc_c, -to_free);
2110                 else
2111                         arc_c = arc_c_min;

2113                 atomic_add_64(&arc_p, -(arc_p >> arc_shrink_shift));
2114                 if (arc_c > arc_size)
2115                         arc_c = MAX(arc_size, arc_c_min);
2116                 if (arc_p > arc_c)
2117                         arc_p = (arc_c >> 1);
2118                 ASSERT(arc_c >= arc_c_min);
2119                 ASSERT((int64_t)arc_p >= 0);
2120         }

2122         if (arc_size > arc_c)
2123                 arc_adjust();
2124 }

2126 /*
2127  * Determine if the system is under memory pressure and is asking
2128  * to reclaim memory. A return value of 1 indicates that the system
2129  * is under memory pressure and that the arc should adjust accordingly.
2130  */
2131 static int
2132 arc_reclaim_needed(void)
2133 {
2134         uint64_t extra;
```



```
2136 #ifdef _KERNEL

2138         if (needfree)
2139                 return (1);

2141         /*
2142          * take 'desfree' extra pages, so we reclaim sooner, rather than later
2143          */
2144         extra = desfree;

2146         /*
2147          * check that we're out of range of the pageout scanner.  It starts to
2148          * schedule paging if freemem is less than lotsfree and needfree.
2149          * lotsfree is the high-water mark for pageout, and needfree is the
2150          * number of needed free pages.  We add extra pages here to make sure
2151          * the scanner doesn't start up while we're freeing memory.
2152          */
2153         if (freemem < lotsfree + needfree + extra)
2154                 return (1);

2156         /*
2157          * check to make sure that swapfs has enough space so that anon
2158          * reservations can still succeed. anon_resvmem() checks that the
2159          * availrmem is greater than swapfs_minfree, and the number of reserved
2160          * swap pages.  We also add a bit of extra here just to prevent
2161          * circumstances from getting really dire.
2162          */
2163         if (availrmem < swapfs_minfree + swapfs_reserve + extra)
2164                 return (1);

2166 #if defined(__i386)
2167         /*
2168          * If we're on an i386 platform, it's possible that we'll exhaust the
2169          * kernel heap space before we ever run out of available physical
2170          * memory.  Most checks of the size of the heap_area compare against
2171          * tune.t_minarmem, which is the minimum available real memory that we
2172          * can have in the system.  However, this is generally fixed at 25 pages
2173          * which is so low that it's useless.  In this comparison, we seek to
2174          * calculate the total heap-size, and reclaim if more than 3/4ths of the
2175          * heap is allocated.  (Or, in the calculation, if less than 1/4th is
2176          * free)
2177          */
2178         if (vmem_size(heap_arena, VMEM_FREE) <
2179             (vmem_size(heap_arena, VMEM_FREE | VMEM_ALLOC) >> 2))
2180                 return (1);
2181 #endif

2183         /*
2184          * If zio data pages are being allocated out of a separate heap segment,
2185          * then enforce that the size of available vmem for this arena remains
2186          * above about 1/16th free.
2187          *
2188          * Note: The 1/16th arena free requirement was put in place
2189          * to aggressively evict memory from the arc in order to avoid
2190          * memory fragmentation issues.
2191          */
2192         if (zio_arena != NULL &&
2193             vmem_size(zio_arena, VMEM_FREE) <
2194             (vmem_size(zio_arena, VMEM_ALLOC) >> 4))
2195                 return (1);
2196 #else
2197         if (spa_get_random(100) == 0)
2198                 return (1);
2199 #endif
2200         return (0);
```

```
}

static void
arc_kmem_reap_now(arc_reclaim_strategy_t strat)
{
        size_t                  i;
        kmem_cache_t            *prev_cache = NULL;
        kmem_cache_t            *prev_data_cache = NULL;
        extern kmem_cache_t     *zio_buf_cache[];
        extern kmem_cache_t     *zio_data_buf_cache[];

#ifdef _KERNEL
        if (arc_meta_used >= arc_meta_limit) {
                /*
                 * We are exceeding our meta-data cache limit.
                 * Purge some DNLC entries to release holds on meta-data.
                 */
                dnlc_reduce_cache((void *)(uintptr_t)arc_reduce_dnlc_percent);
        }
#if defined(__i386)
        /*
         * Reclaim unused memory from all kmem caches.
         */
        kmem_reap();
#endif
#endif

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

        for (i = 0; i < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT; i++) {
                if (zio_buf_cache[i] != prev_cache) {
                        prev_cache = zio_buf_cache[i];
                        kmem_cache_reap_now(zio_buf_cache[i]);
                }
                if (zio_data_buf_cache[i] != prev_data_cache) {
                        prev_data_cache = zio_data_buf_cache[i];
                        kmem_cache_reap_now(zio_data_buf_cache[i]);
                }
        }
        kmem_cache_reap_now(buf_cache);
        kmem_cache_reap_now(hdr_cache);

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

static void
arc_reclaim_thread(void)
{
        clock_t                 growtime = 0;
        arc_reclaim_strategy_t  last_reclaim = ARC_RECLAIM_CONS;
        callb_cpr_t             cpr;

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

        mutex_enter(&arc_reclaim_thr_lock);
        while (arc_thread_exit == 0) {
```



```
                if (arc_reclaim_needed()) {

                        if (arc_no_grow) {
                                if (last_reclaim == ARC_RECLAIM_CONS) {
                                        last_reclaim = ARC_RECLAIM_AGGR;
                                } else {
                                        last_reclaim = ARC_RECLAIM_CONS;
                                }
                        } else {
                                arc_no_grow = TRUE;
                                last_reclaim = ARC_RECLAIM_AGGR;
                                membar_producer();
                        }

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

                        arc_kmem_reap_now(last_reclaim);
                        arc_warm = B_TRUE;

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

                arc_adjust();

                if (arc_eviction_list != NULL)
                        arc_do_user_evicts();

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

        arc_thread_exit = 0;
        cv_broadcast(&arc_reclaim_thr_cv);
        CALLB_CPR_EXIT(&cpr);           /* drops arc_reclaim_thr_lock */
        thread_exit();
}

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

        if (state == arc_l2c_only)
                return;

        ASSERT(bytes > 0);
        /*
         * Adapt the target size of the MRU list:
         *      - if we just hit in the MRU ghost list, then increase
         *        the target size of the MRU list.
         *      - if we just hit in the MFU ghost list, then increase
         *        the target size of the MFU list by decreasing the
         *        target size of the MRU list.
         */
        if (state == arc_mru_ghost) {
```

```
2333                 mult = ((arc_mru_ghost->arcs_size >= arc_mfu_ghost->arcs_size) ?
2334                     1 : (arc_mfu_ghost->arcs_size/arc_mru_ghost->arcs_size));
2335                 mult = MIN(mult, 10); /* avoid wild arc_p adjustment */
2336
2337                 arc_p = MIN(arc_c - arc_p_min, arc_p + bytes * mult);
2338         } else if (state == arc_mfu_ghost) {
2339                 uint64_t delta;
2340
2341                 mult = ((arc_mfu_ghost->arcs_size >= arc_mru_ghost->arcs_size) ?
2342                     1 : (arc_mru_ghost->arcs_size/arc_mfu_ghost->arcs_size));
2343                 mult = MIN(mult, 10);
2344
2345                 delta = MIN(bytes * mult, arc_p);
2346                 arc_p = MAX(arc_p_min, arc_p - delta);
2347         }
2348         ASSERT((int64_t)arc_p >= 0);
2349
2350         if (arc_reclaim_needed()) {
2351                 cv_signal(&arc_reclaim_thr_cv);
2352                 return;
2353         }
2354
2355         if (arc_no_grow)
2356                 return;
2357
2358         if (arc_c >= arc_c_max)
2359                 return;
2360
2361         /*
2362          * If we're within (2 * maxblocksize) bytes of the target
2363          * cache size, increment the target cache size
2364          */
2365         if (arc_size > arc_c - (2ULL << SPA_MAXBLOCKSHIFT)) {
2366                 atomic_add_64(&arc_c, (int64_t)bytes);
2367                 if (arc_c > arc_c_max)
2368                         arc_c = arc_c_max;
2369                 else if (state == arc_anon)
2370                         atomic_add_64(&arc_p, (int64_t)bytes);
2371                 if (arc_p > arc_c)
2372                         arc_p = arc_c;
2373         }
2374         ASSERT((int64_t)arc_p >= 0);
2375 }
2376
2377 /*
2378  * Check if the cache has reached its limits and eviction is required
2379  * prior to insert.
2380  */
2381 static int
2382 arc_evict_needed(arc_buf_contents_t type)
2383 {
2384         if (type == ARC_BUFC_METADATA && arc_meta_used >= arc_meta_limit)
2385                 return (1);
2386
2387         if (arc_reclaim_needed())
2388                 return (1);
2389
2390         return (arc_size > arc_c);
2391 }
2392
2393 /*
2394  * The buffer, supplied as the first argument, needs a data block.
2395  * So, if we are at cache max, determine which cache should be victimized.
2396  * We have the following cases:
2397  *
2398  * 1. Insert for MRU, p > sizeof(arc_anon + arc_mru) ->
```



```
2399  * In this situation if we're out of space, but the resident size of the MFU is
2400  * under the limit, victimize the MFU cache to satisfy this insertion request.
2401  *
2402  * 2. Insert for MRU, p <= sizeof(arc_anon + arc_mru) ->
2403  * Here, we've used up all of the available space for the MRU, so we need to
2404  * evict from our own cache instead.  Evict from the set of resident MRU
2405  * entries.
2406  *
2407  * 3. Insert for MFU (c - p) > sizeof(arc_mfu) ->
2408  * c minus p represents the MFU space in the cache, since p is the size of the
2409  * cache that is dedicated to the MRU.  In this situation there's still space on
2410  * the MFU side, so the MRU side needs to be victimized.
2411  *
2412  * 4. Insert for MFU (c - p) < sizeof(arc_mfu) ->
2413  * MFU's resident set is consuming more space than it has been allotted.  In
2414  * this situation, we must victimize our own cache, the MFU, for this insertion.
2415  */
2416 static void
2417 arc_get_data_buf(arc_buf_t *buf)
2418 {
2419         arc_state_t             *state = buf->b_hdr->b_state;
2420         uint64_t                size = buf->b_hdr->b_size;
2421         arc_buf_contents_t      type = buf->b_hdr->b_type;
2422
2423         arc_adapt(size, state);
2424
2425         /*
2426          * We have not yet reached cache maximum size,
2427          * just allocate a new buffer.
2428          */
2429         if (!arc_evict_needed(type)) {
2430                 if (type == ARC_BUFC_METADATA) {
2431                         buf->b_data = zio_buf_alloc(size);
2432                         arc_space_consume(size, ARC_SPACE_DATA);
2433                 } else {
2434                         ASSERT(type == ARC_BUFC_DATA);
2435                         buf->b_data = zio_data_buf_alloc(size);
2436                         ARCSTAT_INCR(arcstat_data_size, size);
2437                         atomic_add_64(&arc_size, size);
2438                 }
2439                 goto out;
2440         }
2441
2442         /*
2443          * If we are prefetching from the mfu ghost list, this buffer
2444          * will end up on the mru list; so steal space from there.
2445          */
2446         if (state == arc_mfu_ghost)
2447                 state = buf->b_hdr->b_flags & ARC_PREFETCH ? arc_mru : arc_mfu;
2448         else if (state == arc_mru_ghost)
2449                 state = arc_mru;
2450
2451         if (state == arc_mru || state == arc_anon) {
2452                 uint64_t mru_used = arc_anon->arcs_size + arc_mru->arcs_size;
2453                 state = (arc_mfu->arcs_lsize[type] >= size &&
2454                     arc_p > mru_used) ? arc_mfu : arc_mru;
2455         } else {
2456                 /* MFU cases */
2457                 uint64_t mfu_space = arc_c - arc_p;
2458                 state =  (arc_mru->arcs_lsize[type] >= size &&
2459                     mfu_space > arc_mfu->arcs_size) ? arc_mru : arc_mfu;
2460         }
2461         if ((buf->b_data = arc_evict(state, NULL, size, TRUE, type)) == NULL) {
2462                 if (type == ARC_BUFC_METADATA) {
2463                         buf->b_data = zio_buf_alloc(size);
2464                         arc_space_consume(size, ARC_SPACE_DATA);
```

```
2465                 } else {
2466                         ASSERT(type == ARC_BUFC_DATA);
2467                         buf->b_data = zio_data_buf_alloc(size);
2468                         ARCSTAT_INCR(arcstat_data_size, size);
2469                         atomic_add_64(&arc_size, size);
2470                 }
2471                 ARCSTAT_BUMP(arcstat_recycle_miss);
2472         }
2473         ASSERT(buf->b_data != NULL);
2474 out:
2475         /*
2476          * Update the state size.  Note that ghost states have a
2477          * "ghost size" and so don't need to be updated.
2478          */
2479         if (!GHOST_STATE(buf->b_hdr->b_state)) {
2480                 arc_buf_hdr_t *hdr = buf->b_hdr;

2482                 atomic_add_64(&hdr->b_state->arcs_size, size);
2483                 if (list_link_active(&hdr->b_arc_node)) {
2484                         ASSERT(refcount_is_zero(&hdr->b_refcnt));
2485                         atomic_add_64(&hdr->b_state->arcs_lsize[type], size);
2486                 }
2487                 /*
2488                  * If we are growing the cache, and we are adding anonymous
2489                  * data, and we have outgrown arc_p, update arc_p
2490                  */
2491                 if (arc_size < arc_c && hdr->b_state == arc_anon &&
2492                     arc_anon->arcs_size + arc_mru->arcs_size > arc_p)
2493                         arc_p = MIN(arc_c, arc_p + size);
2494         }
2495 }

2497 /*
2498  * This routine is called whenever a buffer is accessed.
2499  * NOTE: the hash lock is dropped in this function.
2500  */
2501 static void
2502 arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock)
2503 {
2504         clock_t now;

2506         ASSERT(MUTEX_HELD(hash_lock));

2508         if (buf->b_state == arc_anon) {
2509                 /*
2510                  * This buffer is not in the cache, and does not
2511                  * appear in our "ghost" list.  Add the new buffer
2512                  * to the MRU state.
2513                  */

2515                 ASSERT(buf->b_arc_access == 0);
2516                 buf->b_arc_access = ddi_get_lbolt();
2517                 DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2518                 arc_change_state(arc_mru, buf, hash_lock);

2520         } else if (buf->b_state == arc_mru) {
2521                 now = ddi_get_lbolt();

2523                 /*
2524                  * If this buffer is here because of a prefetch, then either:
2525                  * - clear the flag if this is a "referencing" read
2526                  *   (any subsequent access will bump this into the MFU state).
2527                  * or
2528                  * - move the buffer to the head of the list if this is
2529                  *   another prefetch (to make it less likely to be evicted).
2530                  */
```



```
2531                 if ((buf->b_flags & ARC_PREFETCH) != 0) {
2532                         if (refcount_count(&buf->b_refcnt) == 0) {
2533                                 ASSERT(list_link_active(&buf->b_arc_node));
2534                         } else {
2535                                 buf->b_flags &= ~ARC_PREFETCH;
2536                                 ARCSTAT_BUMP(arcstat_mru_hits);
2537                         }
2538                         buf->b_arc_access = now;
2539                         return;
2540                 }

2542                 /*
2543                  * This buffer has been "accessed" only once so far,
2544                  * but it is still in the cache. Move it to the MFU
2545                  * state.
2546                  */
2547                 if (now > buf->b_arc_access + ARC_MINTIME) {
2548                         /*
2549                          * More than 125ms have passed since we
2550                          * instantiated this buffer.  Move it to the
2551                          * most frequently used state.
2552                          */
2553                         buf->b_arc_access = now;
2554                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2555                         arc_change_state(arc_mfu, buf, hash_lock);
2556                 }
2557                 ARCSTAT_BUMP(arcstat_mru_hits);
2558         } else if (buf->b_state == arc_mru_ghost) {
2559                 arc_state_t     *new_state;
2560                 /*
2561                  * This buffer has been "accessed" recently, but
2562                  * was evicted from the cache.  Move it to the
2563                  * MFU state.
2564                  */

2566                 if (buf->b_flags & ARC_PREFETCH) {
2567                         new_state = arc_mru;
2568                         if (refcount_count(&buf->b_refcnt) > 0)
2569                                 buf->b_flags &= ~ARC_PREFETCH;
2570                         DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2571                 } else {
2572                         new_state = arc_mfu;
2573                         DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2574                 }

2576                 buf->b_arc_access = ddi_get_lbolt();
2577                 arc_change_state(new_state, buf, hash_lock);

2579                 ARCSTAT_BUMP(arcstat_mru_ghost_hits);
2580         } else if (buf->b_state == arc_mfu) {
2581                 /*
2582                  * This buffer has been accessed more than once and is
2583                  * still in the cache.  Keep it in the MFU state.
2584                  *
2585                  * NOTE: an add_reference() that occurred when we did
2586                  * the arc_read() will have kicked this off the list.
2587                  * If it was a prefetch, we will explicitly move it to
2588                  * the head of the list now.
2589                  */
2590                 if ((buf->b_flags & ARC_PREFETCH) != 0) {
2591                         ASSERT(refcount_count(&buf->b_refcnt) == 0);
2592                         ASSERT(list_link_active(&buf->b_arc_node));
2593                 }
2594                 ARCSTAT_BUMP(arcstat_mfu_hits);
2595                 buf->b_arc_access = ddi_get_lbolt();
2596         } else if (buf->b_state == arc_mfu_ghost) {
```

```
2597                arc_state_t     *new_state = arc_mfu;
2598                /*
2599                 * This buffer has been accessed more than once but has
2600                 * been evicted from the cache.  Move it back to the
2601                 * MFU state.
2602                 */

2604                if (buf->b_flags & ARC_PREFETCH) {
2605                        /*
2606                         * This is a prefetch access...
2607                         * move this block back to the MRU state.
2608                         */
2609                        ASSERT0(refcount_count(&buf->b_refcnt));
2610                        new_state = arc_mru;
2611                }

2613                buf->b_arc_access = ddi_get_lbolt();
2614                DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2615                arc_change_state(new_state, buf, hash_lock);

2617                ARCSTAT_BUMP(arcstat_mfu_ghost_hits);
2618        } else if (buf->b_state == arc_l2c_only) {
2619                /*
2620                 * This buffer is on the 2nd Level ARC.
2621                 */

2623                buf->b_arc_access = ddi_get_lbolt();
2624                DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2625                arc_change_state(arc_mfu, buf, hash_lock);
2626        } else {
2627                ASSERT(!"invalid arc state");
2628        }
2629 }

2631 /* a generic arc_done_func_t which you can use */
2632 /* ARGSUSED */
2633 void
2634 arc_bcopy_func(zio_t *zio, arc_buf_t *buf, void *arg)
2635 {
2636        if (zio == NULL || zio->io_error == 0)
2637                bcopy(buf->b_data, arg, buf->b_hdr->b_size);
2638        VERIFY(arc_buf_remove_ref(buf, arg));
2639 }

2641 /* a generic arc_done_func_t */
2642 void
2643 arc_getbuf_func(zio_t *zio, arc_buf_t *buf, void *arg)
2644 {
2645        arc_buf_t **bufp = arg;
2646        if (zio && zio->io_error) {
2647                VERIFY(arc_buf_remove_ref(buf, arg));
2648                *bufp = NULL;
2649        } else {
2650                *bufp = buf;
2651                ASSERT(buf->b_data);
2652        }
2653 }

2655 static void
2656 arc_read_done(zio_t *zio)
2657 {
2658        arc_buf_hdr_t   *hdr, *found;
2659        arc_buf_t       *buf;
2660        arc_buf_t       *abuf;  /* buffer we're assigning to callback */
2661        kmutex_t        *hash_lock;
2662        arc_callback_t  *callback_list, *acb;
```



```
2663        int             freeable = FALSE;

2665        buf = zio->io_private;
2666        hdr = buf->b_hdr;

2668        /*
2669         * The hdr was inserted into hash-table and removed from lists
2670         * prior to starting I/O.  We should find this header, since
2671         * it's in the hash table, and it should be legit since it's
2672         * not possible to evict it during the I/O.  The only possible
2673         * reason for it not to be found is if we were freed during the
2674         * read.
2675         */
2676        found = buf_hash_find(hdr->b_spa, &hdr->b_dva, hdr->b_birth,
2677            &hash_lock);

2679        ASSERT((found == NULL && HDR_FREED_IN_READ(hdr) && hash_lock == NULL) ||
2680            (found == hdr && DVA_EQUAL(&hdr->b_dva, BP_IDENTITY(zio->io_bp))) ||
2681            (found == hdr && HDR_L2_READING(hdr)));

2683        hdr->b_flags &= ~ARC_L2_EVICTED;
2684        if (l2arc_noprefetch && (hdr->b_flags & ARC_PREFETCH))
2685                hdr->b_flags &= ~ARC_L2CACHE;

2687        /* byteswap if necessary */
2688        callback_list = hdr->b_acb;
2689        ASSERT(callback_list != NULL);
2690        if (BP_SHOULD_BYTESWAP(zio->io_bp) && zio->io_error == 0) {
2691                dmu_object_byteswap_t bswap =
2692                    DMU_OT_BYTESWAP(BP_GET_TYPE(zio->io_bp));
2693                arc_byteswap_func_t *func = BP_GET_LEVEL(zio->io_bp) > 0 ?
2694                    byteswap_uint64_array :
2695                    dmu_ot_byteswap[bswap].ob_func;
2696                func(buf->b_data, hdr->b_size);
2697        }

2699        arc_cksum_compute(buf, B_FALSE);
2700        arc_buf_watch(buf);

2702        if (hash_lock && zio->io_error == 0 && hdr->b_state == arc_anon) {
2703                /*
2704                 * Only call arc_access on anonymous buffers.  This is because
2705                 * if we've issued an I/O for an evicted buffer, we've already
2706                 * called arc_access (to prevent any simultaneous readers from
2707                 * getting confused).
2708                 */
2709                arc_access(hdr, hash_lock);
2710        }

2712        /* create copies of the data buffer for the callers */
2713        abuf = buf;
2714        for (acb = callback_list; acb; acb = acb->acb_next) {
2715                if (acb->acb_done) {
2716                        if (abuf == NULL) {
2717                                ARCSTAT_BUMP(arcstat_duplicate_reads);
2718                                abuf = arc_buf_clone(buf);
2719                        }
2720                        acb->acb_buf = abuf;
2721                        abuf = NULL;
2722                }
2723        }
2724        hdr->b_acb = NULL;
2725        hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
2726        ASSERT(!HDR_BUF_AVAILABLE(hdr));
2727        if (abuf == buf) {
2728                ASSERT(buf->b_efunc == NULL);
```

```
2729                 ASSERT(hdr->b_datacnt == 1);
2730                 hdr->b_flags |= ARC_BUF_AVAILABLE;
2731         }

2733         ASSERT(refcount_is_zero(&hdr->b_refcnt) || callback_list != NULL);

2735         if (zio->io_error != 0) {
2736                 hdr->b_flags |= ARC_IO_ERROR;
2737                 if (hdr->b_state != arc_anon)
2738                         arc_change_state(arc_anon, hdr, hash_lock);
2739                 if (HDR_IN_HASH_TABLE(hdr))
2740                         buf_hash_remove(hdr);
2741                 freeable = refcount_is_zero(&hdr->b_refcnt);
2742         }

2744         /*
2745          * Broadcast before we drop the hash_lock to avoid the possibility
2746          * that the hdr (and hence the cv) might be freed before we get to
2747          * the cv_broadcast().
2748          */
2749         cv_broadcast(&hdr->b_cv);

2751         if (hash_lock) {
2752                 mutex_exit(hash_lock);
2753         } else {
2754                 /*
2755                  * This block was freed while we waited for the read to
2756                  * complete.  It has been removed from the hash table and
2757                  * moved to the anonymous state (so that it won't show up
2758                  * in the cache).
2759                  */
2760                 ASSERT3P(hdr->b_state, ==, arc_anon);
2761                 freeable = refcount_is_zero(&hdr->b_refcnt);
2762         }

2764         /* execute each callback and free its structure */
2765         while ((acb = callback_list) != NULL) {
2766                 if (acb->acb_done)
2767                         acb->acb_done(zio, acb->acb_buf, acb->acb_private);

2769                 if (acb->acb_zio_dummy != NULL) {
2770                         acb->acb_zio_dummy->io_error = zio->io_error;
2771                         zio_nowait(acb->acb_zio_dummy);
2772                 }

2774                 callback_list = acb->acb_next;
2775                 kmem_free(acb, sizeof (arc_callback_t));
2776         }

2778         if (freeable)
2779                 arc_hdr_destroy(hdr);
2780 }

2782 /*
2783  * "Read" the block at the specified DVA (in bp) via the
2784  * cache.  If the block is found in the cache, invoke the provided
2785  * callback immediately and return.  Note that the 'zio' parameter
2786  * in the callback will be NULL in this case, since no IO was
2787  * required.  If the block is not in the cache pass the read request
2788  * on to the spa with a substitute callback function, so that the
2789  * requested block will be added to the cache.
2790  *
2791  * If a read request arrives for a block that has a read in-progress,
2792  * either wait for the in-progress read to complete (and return the
2793  * results); or, if this is a read with a "done" func, add a record
2794  * to the read to invoke the "done" func when the read completes,
```



```
2795  * and return; or just return.
2796  *
2797  * arc_read_done() will invoke all the requested "done" functions
2798  * for readers of this block.
2799  */
2800 int
2801 arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
2802     void *private, int priority, int zio_flags, uint32_t *arc_flags,
2803     const zbookmark_t *zb)
2804 {
2805         arc_buf_hdr_t *hdr;
2806         arc_buf_t *buf = NULL;
2807         kmutex_t *hash_lock;
2808         zio_t *rzio;
2809         uint64_t guid = spa_load_guid(spa);

2811 top:
2812         hdr = buf_hash_find(guid, BP_IDENTITY(bp), BP_PHYSICAL_BIRTH(bp),
2813             &hash_lock);
2814         if (hdr && hdr->b_datacnt > 0) {

2816                 *arc_flags |= ARC_CACHED;

2818                 if (HDR_IO_IN_PROGRESS(hdr)) {

2820                         if (*arc_flags & ARC_WAIT) {
2821                                 cv_wait(&hdr->b_cv, hash_lock);
2822                                 mutex_exit(hash_lock);
2823                                 goto top;
2824                         }
2825                         ASSERT(*arc_flags & ARC_NOWAIT);

2827                         if (done) {
2828                                 arc_callback_t  *acb = NULL;

2830                                 acb = kmem_zalloc(sizeof (arc_callback_t),
2831                                     KM_SLEEP);
2832                                 acb->acb_done = done;
2833                                 acb->acb_private = private;
2834                                 if (pio != NULL)
2835                                         acb->acb_zio_dummy = zio_null(pio,
2836                                             spa, NULL, NULL, NULL, zio_flags);

2838                                 ASSERT(acb->acb_done != NULL);
2839                                 acb->acb_next = hdr->b_acb;
2840                                 hdr->b_acb = acb;
2841                                 add_reference(hdr, hash_lock, private);
2842                                 mutex_exit(hash_lock);
2843                                 return (0);
2844                         }
2845                         mutex_exit(hash_lock);
2846                         return (0);
2847                 }

2849                 ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);

2851                 if (done) {
2852                         add_reference(hdr, hash_lock, private);
2853                         /*
2854                          * If this block is already in use, create a new
2855                          * copy of the data so that we will be guaranteed
2856                          * that arc_release() will always succeed.
2857                          */
2858                         buf = hdr->b_buf;
2859                         ASSERT(buf);
2860                         ASSERT(buf->b_data);
```

```
2861                        if (HDR_BUF_AVAILABLE(hdr)) {
2862                                ASSERT(buf->b_efunc == NULL);
2863                                hdr->b_flags &= ~ARC_BUF_AVAILABLE;
2864                        } else {
2865                                buf = arc_buf_clone(buf);
2866                        }

2868                } else if (*arc_flags & ARC_PREFETCH &&
2869                    refcount_count(&hdr->b_refcnt) == 0) {
2870                        hdr->b_flags |= ARC_PREFETCH;
2871                }
2872                DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
2873                arc_access(hdr, hash_lock);
2874                if (*arc_flags & ARC_L2CACHE)
2875                        hdr->b_flags |= ARC_L2CACHE;
2876                mutex_exit(hash_lock);
2877                ARCSTAT_BUMP(arcstat_hits);
2878                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
2879                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
2880                    data, metadata, hits);

2882                if (done)
2883                        done(NULL, buf, private);
2884        } else {
2885                uint64_t size = BP_GET_LSIZE(bp);
2886                arc_callback_t  *acb;
2887                vdev_t *vd = NULL;
2888                uint64_t addr = 0;
2889                boolean_t devw = B_FALSE;

2891                if (hdr == NULL) {
2892                        /* this block is not in the cache */
2893                        arc_buf_hdr_t   *exists;
2894                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
2895                        buf = arc_buf_alloc(spa, size, private, type);
2896                        hdr = buf->b_hdr;
2897                        hdr->b_dva = *BP_IDENTITY(bp);
2898                        hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
2899                        hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
2900                        exists = buf_hash_insert(hdr, &hash_lock);
2901                        if (exists) {
2902                                /* somebody beat us to the hash insert */
2903                                mutex_exit(hash_lock);
2904                                buf_discard_identity(hdr);
2905                                (void) arc_buf_remove_ref(buf, private);
2906                                goto top; /* restart the IO request */
2907                        }
2908                        /* if this is a prefetch, we don't have a reference */
2909                        if (*arc_flags & ARC_PREFETCH) {
2910                                (void) remove_reference(hdr, hash_lock,
2911                                    private);
2912                                hdr->b_flags |= ARC_PREFETCH;
2913                        }
2914                        if (*arc_flags & ARC_L2CACHE)
2915                                hdr->b_flags |= ARC_L2CACHE;
2916                        if (BP_GET_LEVEL(bp) > 0)
2917                                hdr->b_flags |= ARC_INDIRECT;
2918                } else {
2919                        /* this block is in the ghost cache */
2920                        ASSERT(GHOST_STATE(hdr->b_state));
2921                        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
2922                        ASSERT0(refcount_count(&hdr->b_refcnt));
2923                        ASSERT(hdr->b_buf == NULL);

2925                        /* if this is a prefetch, we don't have a reference */
2926                        if (*arc_flags & ARC_PREFETCH)
```



```
2927                                hdr->b_flags |= ARC_PREFETCH;
2928                        else
2929                                add_reference(hdr, hash_lock, private);
2930                        if (*arc_flags & ARC_L2CACHE)
2931                                hdr->b_flags |= ARC_L2CACHE;
2932                        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
2933                        buf->b_hdr = hdr;
2934                        buf->b_data = NULL;
2935                        buf->b_efunc = NULL;
2936                        buf->b_private = NULL;
2937                        buf->b_next = NULL;
2938                        hdr->b_buf = buf;
2939                        ASSERT(hdr->b_datacnt == 0);
2940                        hdr->b_datacnt = 1;
2941                        arc_get_data_buf(buf);
2942                        arc_access(hdr, hash_lock);
2943                }

2945                ASSERT(!GHOST_STATE(hdr->b_state));

2947                acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
2948                acb->acb_done = done;
2949                acb->acb_private = private;

2951                ASSERT(hdr->b_acb == NULL);
2952                hdr->b_acb = acb;
2953                hdr->b_flags |= ARC_IO_IN_PROGRESS;

2955                if (HDR_L2CACHE(hdr) && hdr->b_l2hdr != NULL &&
2956                    (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
2957                        devw = hdr->b_l2hdr->b_dev->l2ad_writing;
2958                        addr = hdr->b_l2hdr->b_daddr;
2959                        /*
2960                         * Lock out device removal.
2961                         */
2962                        if (vdev_is_dead(vd) ||
2963                            !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
2964                                vd = NULL;
2965                }

2967                mutex_exit(hash_lock);

2969                /*
2970                 * At this point, we have a level 1 cache miss.  Try again in
2971                 * L2ARC if possible.
2972                 */
2973 #endif /* ! codereview */
2974                ASSERT3U(hdr->b_size, ==, size);
2975                DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
2976                    uint64_t, size, zbookmark_t *, zb);
2977                ARCSTAT_BUMP(arcstat_misses);
2978                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
2979                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
2980                    data, metadata, misses);

2982                if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
2983                        /*
2984                         * Read from the L2ARC if the following are true:
2985                         * 1. The L2ARC vdev was previously cached.
2986                         * 2. This buffer still has L2ARC metadata.
2987                         * 3. This buffer isn't currently writing to the L2ARC.
2988                         * 4. The L2ARC entry wasn't evicted, which may
2989                         *    also have invalidated the vdev.
2990                         * 5. This isn't prefetch and l2arc_noprefetch is set.
2991                         */
2992                        if (hdr->b_l2hdr != NULL &&
```

```
2993                            !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
2994                            !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
2995                                l2arc_read_callback_t *cb;

2997                                DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
2998                                ARCSTAT_BUMP(arcstat_l2_hits);

3000                                cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
3001                                    KM_SLEEP);
3002                                cb->l2rcb_buf = buf;
3003                                cb->l2rcb_spa = spa;
3004                                cb->l2rcb_bp = *bp;
3005                                cb->l2rcb_zb = *zb;
3006                                cb->l2rcb_flags = zio_flags;

3008                                ASSERT(addr >= VDEV_LABEL_START_SIZE &&
3009                                    addr + size < vd->vdev_psize -
3010                                    VDEV_LABEL_END_SIZE);

3012                                /*
3013                                 * l2arc read.  The SCL_L2ARC lock will be
3014                                 * released by l2arc_read_done().
3015                                 */
3016                                rzio = zio_read_phys(pio, vd, addr, size,
3017                                    buf->b_data, ZIO_CHECKSUM_OFF,
3018                                    l2arc_read_done, cb, priority, zio_flags |
3019                                    ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
3020                                    ZIO_FLAG_DONT_PROPAGATE |
3021                                    ZIO_FLAG_DONT_RETRY, B_FALSE);
3022                                DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
3023                                    zio_t *, rzio);
3024                                ARCSTAT_INCR(arcstat_l2_read_bytes, size);

3026                                if (*arc_flags & ARC_NOWAIT) {
3027                                        zio_nowait(rzio);
3028                                        return (0);
3029                                }

3031                                ASSERT(*arc_flags & ARC_WAIT);
3032                                if (zio_wait(rzio) == 0)
3033                                        return (0);

3035                                /* l2arc read error; goto zio_read() */
3036                        } else {
3037                                DTRACE_PROBE1(l2arc__miss,
3038                                    arc_buf_hdr_t *, hdr);
3039                                ARCSTAT_BUMP(arcstat_l2_misses);
3040                                if (HDR_L2_WRITING(hdr))
3041                                        ARCSTAT_BUMP(arcstat_l2_rw_clash);
3042                                spa_config_exit(spa, SCL_L2ARC, vd);
3043                        }
3044                } else {
3045                        if (vd != NULL)
3046                                spa_config_exit(spa, SCL_L2ARC, vd);
3047                        if (l2arc_ndev != 0) {
3048                                DTRACE_PROBE1(l2arc__miss,
3049                                    arc_buf_hdr_t *, hdr);
3050                                ARCSTAT_BUMP(arcstat_l2_misses);
3051                        }
3052                }

3054                rzio = zio_read(pio, spa, bp, buf->b_data, size,
3055                    arc_read_done, buf, priority, zio_flags, zb);

3057                if (*arc_flags & ARC_WAIT)
3058                        return (zio_wait(rzio));
```



```
3060                ASSERT(*arc_flags & ARC_NOWAIT);
3061                zio_nowait(rzio);
3062        }
3063        return (0);
3064 }

3066 void
3067 arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private)
3068 {
3069        ASSERT(buf->b_hdr != NULL);
3070        ASSERT(buf->b_hdr->b_state != arc_anon);
3071        ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt) || func == NULL);
3072        ASSERT(buf->b_efunc == NULL);
3073        ASSERT(!HDR_BUF_AVAILABLE(buf->b_hdr));

3075        buf->b_efunc = func;
3076        buf->b_private = private;
3077 }

3079 /*
3080  * This is used by the DMU to let the ARC know that a buffer is
3081  * being evicted, so the ARC should clean up.  If this arc buf
3082  * is not yet in the evicted state, it will be put there.
3083  */
3084 int
3085 arc_buf_evict(arc_buf_t *buf)
3086 {
3087        arc_buf_hdr_t *hdr;
3088        kmutex_t *hash_lock;
3089        arc_buf_t **bufp;

3091        mutex_enter(&buf->b_evict_lock);
3092        hdr = buf->b_hdr;
3093        if (hdr == NULL) {
3094                /*
3095                 * We are in arc_do_user_evicts().
3096                 */
3097                ASSERT(buf->b_data == NULL);
3098                mutex_exit(&buf->b_evict_lock);
3099                return (0);
3100        } else if (buf->b_data == NULL) {
3101                arc_buf_t copy = *buf; /* structure assignment */
3102                /*
3103                 * We are on the eviction list; process this buffer now
3104                 * but let arc_do_user_evicts() do the reaping.
3105                 */
3106                buf->b_efunc = NULL;
3107                mutex_exit(&buf->b_evict_lock);
3108                VERIFY(copy.b_efunc(&copy) == 0);
3109                return (1);
3110        }
3111        hash_lock = HDR_LOCK(hdr);
3112        mutex_enter(hash_lock);
3113        hdr = buf->b_hdr;
3114        ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

3116        ASSERT3U(refcount_count(&hdr->b_refcnt), <, hdr->b_datacnt);
3117        ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);

3119        /*
3120         * Pull this buffer off of the hdr
3121         */
3122        bufp = &hdr->b_buf;
3123        while (*bufp != buf)
3124                bufp = &(*bufp)->b_next;
```

```
3125         *bufp = buf->b_next;

3127         ASSERT(buf->b_data != NULL);
3128         arc_buf_destroy(buf, FALSE, FALSE);

3130         if (hdr->b_datacnt == 0) {
3131                 arc_state_t *old_state = hdr->b_state;
3132                 arc_state_t *evicted_state;

3134                 ASSERT(hdr->b_buf == NULL);
3135                 ASSERT(refcount_is_zero(&hdr->b_refcnt));

3137                 evicted_state =
3138                     (old_state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;

3140                 mutex_enter(&old_state->arcs_mtx);
3141                 mutex_enter(&evicted_state->arcs_mtx);

3143                 arc_change_state(evicted_state, hdr, hash_lock);
3144                 ASSERT(HDR_IN_HASH_TABLE(hdr));
3145                 hdr->b_flags |= ARC_IN_HASH_TABLE;
3146                 hdr->b_flags &= ~ARC_BUF_AVAILABLE;

3148                 mutex_exit(&evicted_state->arcs_mtx);
3149                 mutex_exit(&old_state->arcs_mtx);
3150         }
3151         mutex_exit(hash_lock);
3152         mutex_exit(&buf->b_evict_lock);

3154         VERIFY(buf->b_efunc(buf) == 0);
3155         buf->b_efunc = NULL;
3156         buf->b_private = NULL;
3157         buf->b_hdr = NULL;
3158         buf->b_next = NULL;
3159         kmem_cache_free(buf_cache, buf);
3160         return (1);
3161 }

3163 /*
3164  * Release this buffer from the cache, making it an anonymous buffer.  This
3165  * must be done after a read and prior to modifying the buffer contents.
 257  * Release this buffer from the cache.  This must be done
 258  * after a read and prior to modifying the buffer contents.
3166  * If the buffer has more than one reference, we must make
3167  * a new hdr for the buffer.
3168  */
3169 void
3170 arc_release(arc_buf_t *buf, void *tag)
3171 {
3172         arc_buf_hdr_t *hdr;
3173         kmutex_t *hash_lock = NULL;
3174         l2arc_buf_hdr_t *l2hdr;
3175         uint64_t buf_size;

3177         /*
3178          * It would be nice to assert that if it's DMU metadata (level >
3179          * 0 || it's the dnode file), then it must be syncing context.
3180          * But we don't know that information at this level.
3181          */

3183         mutex_enter(&buf->b_evict_lock);
3184         hdr = buf->b_hdr;

3186         /* this buffer is not on any list */
3187         ASSERT(refcount_count(&hdr->b_refcnt) > 0);
```



```
3189         if (hdr->b_state == arc_anon) {
3190                 /* this buffer is already released */
3191                 ASSERT(buf->b_efunc == NULL);
3192         } else {
3193                 hash_lock = HDR_LOCK(hdr);
3194                 mutex_enter(hash_lock);
3195                 hdr = buf->b_hdr;
3196                 ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
3197         }

3199         l2hdr = hdr->b_l2hdr;
3200         if (l2hdr) {
3201                 mutex_enter(&l2arc_buflist_mtx);
3202                 hdr->b_l2hdr = NULL;
3203         }
3204         buf_size = hdr->b_size;

3206         /*
3207          * Do we have more than one buf?
3208          */
3209         if (hdr->b_datacnt > 1) {
3210                 arc_buf_hdr_t *nhdr;
3211                 arc_buf_t **bufp;
3212                 uint64_t blksz = hdr->b_size;
3213                 uint64_t spa = hdr->b_spa;
3214                 arc_buf_contents_t type = hdr->b_type;
3215                 uint32_t flags = hdr->b_flags;

3217                 ASSERT(hdr->b_buf != buf || buf->b_next != NULL);
3218                 /*
3219                  * Pull the data off of this hdr and attach it to
3220                  * a new anonymous hdr.
3221                  */
3222                 (void) remove_reference(hdr, hash_lock, tag);
3223                 bufp = &hdr->b_buf;
3224                 while (*bufp != buf)
3225                         bufp = &(*bufp)->b_next;
3226                 *bufp = buf->b_next;
3227                 buf->b_next = NULL;

3229                 ASSERT3U(hdr->b_state->arcs_size, >=, hdr->b_size);
3230                 atomic_add_64(&hdr->b_state->arcs_size, -hdr->b_size);
3231                 if (refcount_is_zero(&hdr->b_refcnt)) {
3232                         uint64_t *size = &hdr->b_state->arcs_lsize[hdr->b_type];
3233                         ASSERT3U(*size, >=, hdr->b_size);
3234                         atomic_add_64(size, -hdr->b_size);
3235                 }

3237                 /*
3238                  * We're releasing a duplicate user data buffer, update
3239                  * our statistics accordingly.
3240                  */
3241                 if (hdr->b_type == ARC_BUFC_DATA) {
3242                         ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
3243                         ARCSTAT_INCR(arcstat_duplicate_buffers_size,
3244                             -hdr->b_size);
3245                 }
3246                 hdr->b_datacnt -= 1;
3247                 arc_cksum_verify(buf);
3248                 arc_buf_unwatch(buf);

3250                 mutex_exit(hash_lock);

3252                 nhdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
3253                 nhdr->b_size = blksz;
3254                 nhdr->b_spa = spa;
```

```
3255                 nhdr->b_type = type;
3256                 nhdr->b_buf = buf;
3257                 nhdr->b_state = arc_anon;
3258                 nhdr->b_arc_access = 0;
3259                 nhdr->b_flags = flags & ARC_L2_WRITING;
3260                 nhdr->b_l2hdr = NULL;
3261                 nhdr->b_datacnt = 1;
3262                 nhdr->b_freeze_cksum = NULL;
3263                 (void) refcount_add(&nhdr->b_refcnt, tag);
3264                 buf->b_hdr = nhdr;
3265                 mutex_exit(&buf->b_evict_lock);
3266                 atomic_add_64(&arc_anon->arcs_size, blksz);
3267         } else {
3268                 mutex_exit(&buf->b_evict_lock);
3269                 ASSERT(refcount_count(&hdr->b_refcnt) == 1);
3270                 ASSERT(!list_link_active(&hdr->b_arc_node));
3271                 ASSERT(!HDR_IO_IN_PROGRESS(hdr));
3272                 if (hdr->b_state != arc_anon)
3273                         arc_change_state(arc_anon, hdr, hash_lock);
3274                 hdr->b_arc_access = 0;
3275                 if (hash_lock)
3276                         mutex_exit(hash_lock);

3278                 buf_discard_identity(hdr);
3279                 arc_buf_thaw(buf);
3280         }
3281         buf->b_efunc = NULL;
3282         buf->b_private = NULL;

3284         if (l2hdr) {
3285                 list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
3286                 kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
3287                 ARCSTAT_INCR(arcstat_l2_size, -buf_size);
3288                 mutex_exit(&l2arc_buflist_mtx);
3289         }
3290 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   74441 Tue Apr 23 14:09:35 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
_____unchanged_portion_omitted_

 586 int
 587 dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags)
 588 {
 589         int err = 0;
 590         int havepzio = (zio != NULL);
 591         int prefetch;
 592         dnode_t *dn;

 594         /*
 595          * We don't have to hold the mutex to check db_state because it
 596          * can't be freed while we have a hold on the buffer.
 597          */
 598         ASSERT(!refcount_is_zero(&db->db_holds));

 600         if (db->db_state == DB_NOFILL)
 601                 return (SET_ERROR(EIO));

 603         DB_DNODE_ENTER(db);
 604         dn = DB_DNODE(db);
 605         if ((flags & DB_RF_HAVESTRUCT) == 0)
 606                 rw_enter(&dn->dn_struct_rwlock, RW_READER);

 608         prefetch = db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
 609             (flags & DB_RF_NOPREFETCH) == 0 && dn != NULL &&
 610             DBUF_IS_CACHEABLE(db);

 612         mutex_enter(&db->db_mtx);
 613         if (db->db_state == DB_CACHED) {
 614                 mutex_exit(&db->db_mtx);
 615                 if (prefetch)
 616                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
 617                             db->db.db_size, TRUE);
 618                 if ((flags & DB_RF_HAVESTRUCT) == 0)
 619                         rw_exit(&dn->dn_struct_rwlock);
 620                 DB_DNODE_EXIT(db);
 621         } else if (db->db_state == DB_UNCACHED) {
 622                 spa_t *spa = dn->dn_objset->os_spa;

 624                 if (zio == NULL)
 625                         zio = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL);
 626                 dbuf_read_impl(db, zio, &flags);

 628                 /* dbuf_read_impl has dropped db_mtx for us */

 630                 if (prefetch)
 631                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
 632                             db->db.db_size, flags & DB_RF_CACHED);

 634                 if ((flags & DB_RF_HAVESTRUCT) == 0)
 635                         rw_exit(&dn->dn_struct_rwlock);
 636                 DB_DNODE_EXIT(db);

 638                 if (!havepzio)
 639                         err = zio_wait(zio);
 640         } else {
```



```
 641                 /*
 642                  * Another reader came in while the dbuf was in flight
 643                  * between UNCACHED and CACHED.  Either a writer will finish
 644                  * writing the buffer (sending the dbuf to CACHED) or the
 645                  * first reader's request will reach the read_done callback
 646                  * and send the dbuf to CACHED.  Otherwise, a failure
 647                  * occurred and the dbuf went to UNCACHED.
 648                  */
 649 #endif /* ! codereview */
 650                 mutex_exit(&db->db_mtx);
 651                 if (prefetch)
 652                         dmu_zfetch(&dn->dn_zfetch, db->db.db_offset,
 653                             db->db.db_size, TRUE);
 654                 if ((flags & DB_RF_HAVESTRUCT) == 0)
 655                         rw_exit(&dn->dn_struct_rwlock);
 656                 DB_DNODE_EXIT(db);

 658                 /* Skip the wait per the caller's request. */
 659 #endif /* ! codereview */
 660                 mutex_enter(&db->db_mtx);
 661                 if ((flags & DB_RF_NEVERWAIT) == 0) {
 662                         while (db->db_state == DB_READ ||
 663                             db->db_state == DB_FILL) {
 664                                 ASSERT(db->db_state == DB_READ ||
 665                                     (flags & DB_RF_HAVESTRUCT) == 0);
 666                                 cv_wait(&db->db_changed, &db->db_mtx);
 667                         }
 668                         if (db->db_state == DB_UNCACHED)
 669                                 err = SET_ERROR(EIO);
 670                 }
 671                 mutex_exit(&db->db_mtx);
 672         }

 674         ASSERT(err || havepzio || db->db_state == DB_CACHED);
 675         return (err);
 676 }

 678 static void
 679 dbuf_noread(dmu_buf_impl_t *db)
 680 {
 681         ASSERT(!refcount_is_zero(&db->db_holds));
 682         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
 683         mutex_enter(&db->db_mtx);
 684         while (db->db_state == DB_READ || db->db_state == DB_FILL)
 685                 cv_wait(&db->db_changed, &db->db_mtx);
 686         if (db->db_state == DB_UNCACHED) {
 687                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
 688                 spa_t *spa;

 690                 ASSERT(db->db_buf == NULL);
 691                 ASSERT(db->db.db_data == NULL);
 692                 DB_GET_SPA(&spa, db);
 693                 dbuf_set_data(db, arc_buf_alloc(spa, db->db.db_size, db, type));
 694                 db->db_state = DB_FILL;
 695         } else if (db->db_state == DB_NOFILL) {
 696                 dbuf_set_data(db, NULL);
 697         } else {
 698                 ASSERT3U(db->db_state, ==, DB_CACHED);
 699         }
 700         mutex_exit(&db->db_mtx);
 701 }

 703 /*
 704  * This is our just-in-time copy function.  It makes a copy of
 705  * buffers, that have been modified in a previous transaction
 706  * group, before we modify them in the current active group.
```

```
 *
 * This function is used in two places: when we are dirtying a
 * buffer for the first time in a txg, and when we are freeing
 * a range in a dnode that includes this buffer.
 *
 * Note that when we are called from dbuf_free_range() we do
 * not put a hold on the buffer, we just traverse the active
 * dbuf list for the dnode.
 */
static void
dbuf_fix_old_data(dmu_buf_impl_t *db, uint64_t txg)
{
        dbuf_dirty_record_t *dr = db->db_last_dirty;

        ASSERT(MUTEX_HELD(&db->db_mtx));
        ASSERT(db->db.db_data != NULL);
        ASSERT(db->db_level == 0);
        ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT);

        if (dr == NULL ||
            (dr->dt.dl.dr_data !=
            ((db->db_blkid  == DMU_BONUS_BLKID) ? db->db.db_data : db->db_buf)))
                return;

        /*
         * If the last dirty record for this dbuf has not yet synced
         * and its referencing the dbuf data, either:
         *      reset the reference to point to a new copy,
         * or (if there a no active holders)
         *      just null out the current db_data pointer.
         */
        ASSERT(dr->dr_txg >= txg - 2);
        if (db->db_blkid == DMU_BONUS_BLKID) {
                /* Note that the data bufs here are zio_bufs */
                dr->dt.dl.dr_data = zio_buf_alloc(DN_MAX_BONUSLEN);
                arc_space_consume(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
                bcopy(db->db.db_data, dr->dt.dl.dr_data, DN_MAX_BONUSLEN);
        } else if (refcount_count(&db->db_holds) > db->db_dirtycnt) {
                int size = db->db.db_size;
                arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
                spa_t *spa;

                DB_GET_SPA(&spa, db);
                dr->dt.dl.dr_data = arc_buf_alloc(spa, size, db, type);
                bcopy(db->db.db_data, dr->dt.dl.dr_data->b_data, size);
        } else {
                dbuf_set_data(db, NULL);
        }
}

void
dbuf_unoverride(dbuf_dirty_record_t *dr)
{
        dmu_buf_impl_t *db = dr->dr_dbuf;
        blkptr_t *bp = &dr->dt.dl.dr_overridden_by;
        uint64_t txg = dr->dr_txg;

        ASSERT(MUTEX_HELD(&db->db_mtx));
        ASSERT(dr->dt.dl.dr_override_state != DR_IN_DMU_SYNC);
        ASSERT(db->db_level == 0);

        if (db->db_blkid == DMU_BONUS_BLKID ||
            dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN)
                return;

        ASSERT(db->db_data_pending != dr);
```



```
        /* free this block */
        if (!BP_IS_HOLE(bp) && !dr->dt.dl.dr_nopwrite) {
                spa_t *spa;

                DB_GET_SPA(&spa, db);
                zio_free(spa, txg, bp);
        }
        dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
        dr->dt.dl.dr_nopwrite = B_FALSE;

        /*
         * Release the already-written buffer, so we leave it in
         * a consistent dirty state.  Note that all callers are
         * modifying the buffer, so they will immediately do
         * another (redundant) arc_release().  Therefore, leave
         * the buf thawed to save the effort of freezing &
         * immediately re-thawing it.
         */
        arc_release(dr->dt.dl.dr_data, db);
}

/*
 * Evict (if its unreferenced) or clear (if its referenced) any level-0
 * data blocks in the free range, so that any future readers will find
 * empty blocks.  Also, if we happen accross any level-1 dbufs in the
 * range that have not already been marked dirty, mark them dirty so
 * they stay in memory.
 */
void
dbuf_free_range(dnode_t *dn, uint64_t start, uint64_t end, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db, *db_next;
        uint64_t txg = tx->tx_txg;
        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
        uint64_t first_l1 = start >> epbs;
        uint64_t last_l1 = end >> epbs;

        if (end > dn->dn_maxblkid && (end != DMU_SPILL_BLKID)) {
                end = dn->dn_maxblkid;
                last_l1 = end >> epbs;
        }
        dprintf_dnode(dn, "start=%llu end=%llu\n", start, end);
        mutex_enter(&dn->dn_dbufs_mtx);
        for (db = list_head(&dn->dn_dbufs); db; db = db_next) {
                db_next = list_next(&dn->dn_dbufs, db);
                ASSERT(db->db_blkid != DMU_BONUS_BLKID);

                if (db->db_level == 1 &&
                    db->db_blkid >= first_l1 && db->db_blkid <= last_l1) {
                        mutex_enter(&db->db_mtx);
                        if (db->db_last_dirty &&
                            db->db_last_dirty->dr_txg < txg) {
                                dbuf_add_ref(db, FTAG);
                                mutex_exit(&db->db_mtx);
                                dbuf_will_dirty(db, tx);
                                dbuf_rele(db, FTAG);
                        } else {
                                mutex_exit(&db->db_mtx);
                        }
                }

                if (db->db_level != 0)
                        continue;
                dprintf_dbuf(db, "found buf %s\n", "");
                if (db->db_blkid < start || db->db_blkid > end)
```

```
839                         continue;

841                 /* found a level 0 buffer in the range */
842                 mutex_enter(&db->db_mtx);
843                 if (dbuf_undirty(db, tx)) {
844                         /* mutex has been dropped and dbuf destroyed */
845                         continue;
846                 }

848                 if (db->db_state == DB_UNCACHED ||
849                     db->db_state == DB_NOFILL ||
850                     db->db_state == DB_EVICTING) {
851                         ASSERT(db->db.db_data == NULL);
852                         mutex_exit(&db->db_mtx);
853                         continue;
854                 }
855                 if (db->db_state == DB_READ || db->db_state == DB_FILL) {
856                         /* will be handled in dbuf_read_done or dbuf_rele */
857                         db->db_freed_in_flight = TRUE;
858                         mutex_exit(&db->db_mtx);
859                         continue;
860                 }
861                 if (refcount_count(&db->db_holds) == 0) {
862                         ASSERT(db->db_buf);
863                         dbuf_clear(db);
864                         continue;
865                 }
866                 /* The dbuf is referenced */

868                 if (db->db_last_dirty != NULL) {
869                         dbuf_dirty_record_t *dr = db->db_last_dirty;

871                         if (dr->dr_txg == txg) {
872                                 /*
873                                  * This buffer is "in-use", re-adjust the file
874                                  * size to reflect that this buffer may
875                                  * contain new data when we sync.
876                                  */
877                                 if (db->db_blkid != DMU_SPILL_BLKID &&
878                                     db->db_blkid > dn->dn_maxblkid)
879                                         dn->dn_maxblkid = db->db_blkid;
880                                 dbuf_unoverride(dr);
881                         } else {
882                                 /*
883                                  * This dbuf is not dirty in the open context.
884                                  * Either uncache it (if its not referenced in
885                                  * the open context) or reset its contents to
886                                  * empty.
887                                  */
888                                 dbuf_fix_old_data(db, txg);
889                         }
890                 }
891                 /* clear the contents if its cached */
892                 if (db->db_state == DB_CACHED) {
893                         ASSERT(db->db.db_data != NULL);
894                         arc_release(db->db_buf, db);
895                         bzero(db->db.db_data, db->db.db_size);
896                         arc_buf_freeze(db->db_buf);
897                 }

899                 mutex_exit(&db->db_mtx);
900         }
901         mutex_exit(&dn->dn_dbufs_mtx);
902 }

904 static int
```



```
905 dbuf_block_freeable(dmu_buf_impl_t *db)
906 {
907         dsl_dataset_t *ds = db->db_objset->os_dsl_dataset;
908         uint64_t birth_txg = 0;

910         /*
911          * We don't need any locking to protect db_blkptr:
912          * If it's syncing, then db_last_dirty will be set
913          * so we'll ignore db_blkptr.
914          */
915         ASSERT(MUTEX_HELD(&db->db_mtx));
916         if (db->db_last_dirty)
917                 birth_txg = db->db_last_dirty->dr_txg;
918         else if (db->db_blkptr)
919                 birth_txg = db->db_blkptr->blk_birth;

921         /*
922          * If we don't exist or are in a snapshot, we can't be freed.
923          * Don't pass the bp to dsl_dataset_block_freeable() since we
924          * are holding the db_mtx lock and might deadlock if we are
925          * prefetching a dedup-ed block.
926          */
927         if (birth_txg)
928                 return (ds == NULL ||
929                     dsl_dataset_block_freeable(ds, NULL, birth_txg));
930         else
931                 return (FALSE);
932 }

934 void
935 dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx)
936 {
937         arc_buf_t *buf, *obuf;
938         int osize = db->db.db_size;
939         arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
940         dnode_t *dn;

942         ASSERT(db->db_blkid != DMU_BONUS_BLKID);

944         DB_DNODE_ENTER(db);
945         dn = DB_DNODE(db);

947         /* XXX does *this* func really need the lock? */
948         ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));

950         /*
951          * This call to dbuf_will_dirty() with the dn_struct_rwlock held
952          * is OK, because there can be no other references to the db
953          * when we are changing its size, so no concurrent DB_FILL can
954          * be happening.
955          */
956         /*
957          * XXX we should be doing a dbuf_read, checking the return
958          * value and returning that up to our callers
959          */
960         dbuf_will_dirty(db, tx);

962         /* create the data buffer for the new block */
963         buf = arc_buf_alloc(dn->dn_objset->os_spa, size, db, type);

965         /* copy old block data to the new block */
966         obuf = db->db_buf;
967         bcopy(obuf->b_data, buf->b_data, MIN(osize, size));
968         /* zero the remainder */
969         if (size > osize)
970                 bzero((uint8_t *)buf->b_data + osize, size - osize);
```

```
 972         mutex_enter(&db->db_mtx);
 973         dbuf_set_data(db, buf);
 974         VERIFY(arc_buf_remove_ref(obuf, db));
 975         db->db.db_size = size;

 977         if (db->db_level == 0) {
 978                 ASSERT3U(db->db_last_dirty->dr_txg, ==, tx->tx_txg);
 979                 db->db_last_dirty->dt.dl.dr_data = buf;
 980         }
 981         mutex_exit(&db->db_mtx);

 983         dnode_willuse_space(dn, size-osize, tx);
 984         DB_DNODE_EXIT(db);
 985 }

 987 void
 988 dbuf_release_bp(dmu_buf_impl_t *db)
 989 {
 990         objset_t *os;

 992         DB_GET_OBJSET(&os, db);
 993         ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
 994         ASSERT(arc_released(os->os_phys_buf) ||
 995             list_link_active(&os->os_dsl_dataset->ds_synced_link));
 996         ASSERT(db->db_parent == NULL || arc_released(db->db_parent->db_buf));

 998         (void) arc_release(db->db_buf, db);
 999 }

1001 dbuf_dirty_record_t *
1002 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1003 {
1004         dnode_t *dn;
1005         objset_t *os;
1006         dbuf_dirty_record_t **drp, *dr;
1007         int drop_struct_lock = FALSE;
1008         boolean_t do_free_accounting = B_FALSE;
1009         int txgoff = tx->tx_txg & TXG_MASK;

1011         ASSERT(tx->tx_txg != 0);
1012         ASSERT(!refcount_is_zero(&db->db_holds));
1013         DMU_TX_DIRTY_BUF(tx, db);

1015         DB_DNODE_ENTER(db);
1016         dn = DB_DNODE(db);
1017         /*
1018          * Shouldn't dirty a regular buffer in syncing context.  Private
1019          * objects may be dirtied in syncing context, but only if they
1020          * were already pre-dirtied in open context.
1021          */
1022         ASSERT(!dmu_tx_is_syncing(tx) ||
1023             BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1024             DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1025             dn->dn_objset->os_dsl_dataset == NULL);
1026         /*
1027          * We make this assert for private objects as well, but after we
1028          * check if we're already dirty.  They are allowed to re-dirty
1029          * in syncing context.
1030          */
1031         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1032             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1033             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1035         mutex_enter(&db->db_mtx);
1036         /*
```



```
1037          * XXX make this true for indirects too?  The problem is that
1038          * transactions created with dmu_tx_create_assigned() from
1039          * syncing context don't bother holding ahead.
1040          */
1041         ASSERT(db->db_level != 0 ||
1042             db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1043             db->db_state == DB_NOFILL);

1045         mutex_enter(&dn->dn_mtx);
1046         /*
1047          * Don't set dirtyctx to SYNC if we're just modifying this as we
1048          * initialize the objset.
1049          */
1050         if (dn->dn_dirtyctx == DN_UNDIRTIED &&
1051             !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1052                 dn->dn_dirtyctx =
1053                     (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1054                 ASSERT(dn->dn_dirtyctx_firstset == NULL);
1055                 dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1056         }
1057         mutex_exit(&dn->dn_mtx);

1059         if (db->db_blkid == DMU_SPILL_BLKID)
1060                 dn->dn_have_spill = B_TRUE;

1062         /*
1063          * If this buffer is already dirty, we're done.
1064          */
1065         drp = &db->db_last_dirty;
1066         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1067             db->db.db_object == DMU_META_DNODE_OBJECT);
1068         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1069                 drp = &dr->dr_next;
1070         if (dr && dr->dr_txg == tx->tx_txg) {
1071                 DB_DNODE_EXIT(db);

1073                 if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1074                         /*
1075                          * If this buffer has already been written out,
1076                          * we now need to reset its state.
1077                          */
1078                         dbuf_unoverride(dr);
1079                         if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1080                             db->db_state != DB_NOFILL)
1081                                 arc_buf_thaw(db->db_buf);
1082                 }
1083                 mutex_exit(&db->db_mtx);
1084                 return (dr);
1085         }

1087         /*
1088          * Only valid if not already dirty.
1089          */
1090         ASSERT(dn->dn_object == 0 ||
1091             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1092             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1094         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1095         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1096             dn->dn_phys->dn_nlevels > db->db_level ||
1097             dn->dn_next_nlevels[txgoff] > db->db_level ||
1098             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1099             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);

1101         /*
1102          * We should only be dirtying in syncing context if it's the
```

```
1103          * mos or we're initializing the os or it's a special object.
1104          * However, we are allowed to dirty in syncing context provided
1105          * we already dirtied it in open context.  Hence we must make
1106          * this assertion only if we're not already dirty.
1107          */
1108         os = dn->dn_objset;
1109         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1110             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1111         ASSERT(db->db.db_size != 0);

1113         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1115         if (db->db_blkid != DMU_BONUS_BLKID) {
1116                 /*
1117                  * Update the accounting.
1118                  * Note: we delay "free accounting" until after we drop
1119                  * the db_mtx.  This keeps us from grabbing other locks
1120                  * (and possibly deadlocking) in bp_get_dsize() while
1121                  * also holding the db_mtx.
1122                  */
1123                 dnode_willuse_space(dn, db->db.db_size, tx);
1124                 do_free_accounting = dbuf_block_freeable(db);
1125         }

1127         /*
1128          * If this buffer is dirty in an old transaction group we need
1129          * to make a copy of it so that the changes we make in this
1130          * transaction group won't leak out when we sync the older txg.
1131          */
1132         dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1133         if (db->db_level == 0) {
1134                 void *data_old = db->db_buf;

1136                 if (db->db_state != DB_NOFILL) {
1137                         if (db->db_blkid == DMU_BONUS_BLKID) {
1138                                 dbuf_fix_old_data(db, tx->tx_txg);
1139                                 data_old = db->db.db_data;
1140                         } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1141                                 /*
1142                                  * Release the data buffer from the cache so
1143                                  * that we can modify it without impacting
1144                                  * possible other users of this cached data
1145                                  * block.  Note that indirect blocks and
1146                                  * private objects are not released until the
1147                                  * syncing state (since they are only modified
1148                                  * then).
1149                                  */
1150                                 arc_release(db->db_buf, db);
1151                                 dbuf_fix_old_data(db, tx->tx_txg);
1152                                 data_old = db->db_buf;
1153                         }
1154                         ASSERT(data_old != NULL);
1155                 }
1156                 dr->dt.dl.dr_data = data_old;
1157         } else {
1158                 mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1159                 list_create(&dr->dt.di.dr_children,
1160                     sizeof (dbuf_dirty_record_t),
1161                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
1162         }
1163         dr->dr_dbuf = db;
1164         dr->dr_txg = tx->tx_txg;
1165         dr->dr_next = *drp;
1166         *drp = dr;

1168         /*
```



```
1169          * We could have been freed_in_flight between the dbuf_noread
1170          * and dbuf_dirty.  We win, as though the dbuf_noread() had
1171          * happened after the free.
1172          */
1173         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1174             db->db_blkid != DMU_SPILL_BLKID) {
1175                 mutex_enter(&dn->dn_mtx);
1176                 dnode_clear_range(dn, db->db_blkid, 1, tx);
1177                 mutex_exit(&dn->dn_mtx);
1178                 db->db_freed_in_flight = FALSE;
1179         }

1181         /*
1182          * This buffer is now part of this txg
1183          */
1184         dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1185         db->db_dirtycnt += 1;
1186         ASSERT3U(db->db_dirtycnt, <=, 3);

1188         mutex_exit(&db->db_mtx);

1190         if (db->db_blkid == DMU_BONUS_BLKID ||
1191             db->db_blkid == DMU_SPILL_BLKID) {
1192                 mutex_enter(&dn->dn_mtx);
1193                 ASSERT(!list_link_active(&dr->dr_dirty_node));
1194                 list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1195                 mutex_exit(&dn->dn_mtx);
1196                 dnode_setdirty(dn, tx);
1197                 DB_DNODE_EXIT(db);
1198                 return (dr);
1199         } else if (do_free_accounting) {
1200                 blkptr_t *bp = db->db_blkptr;
1201                 int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1202                     bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1203                 /*
1204                  * This is only a guess -- if the dbuf is dirty
1205                  * in a previous txg, we don't know how much
1206                  * space it will use on disk yet.  We should
1207                  * really have the struct_rwlock to access
1208                  * db_blkptr, but since this is just a guess,
1209                  * it's OK if we get an odd answer.
1210                  */
1211                 ddt_prefetch(os->os_spa, bp);
1212                 dnode_willuse_space(dn, -willfree, tx);
1213         }

1215         if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1216                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
1217                 drop_struct_lock = TRUE;
1218         }

1220         if (db->db_level == 0) {
1221                 dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1222                 ASSERT(dn->dn_maxblkid >= db->db_blkid);
1223         }

1225         if (db->db_level+1 < dn->dn_nlevels) {
1226                 dmu_buf_impl_t *parent = db->db_parent;
1227                 dbuf_dirty_record_t *di;
1228                 int parent_held = FALSE;

1230                 if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1231                         int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1233                         parent = dbuf_hold_level(dn, db->db_level+1,
1234                             db->db_blkid >> epbs, FTAG);
```

```
1235                        ASSERT(parent != NULL);
1236                        parent_held = TRUE;
1237                }
1238                if (drop_struct_lock)
1239                        rw_exit(&dn->dn_struct_rwlock);
1240                ASSERT3U(db->db_level+1, ==, parent->db_level);
1241                di = dbuf_dirty(parent, tx);
1242                if (parent_held)
1243                        dbuf_rele(parent, FTAG);

1245                mutex_enter(&db->db_mtx);
1246                /*  possible race with dbuf_undirty() */
1247                if (db->db_last_dirty == dr ||
1248                    dn->dn_object == DMU_META_DNODE_OBJECT) {
1249                        mutex_enter(&di->dt.di.dr_mtx);
1250                        ASSERT3U(di->dr_txg, ==, tx->tx_txg);
1251                        ASSERT(!list_link_active(&dr->dr_dirty_node));
1252                        list_insert_tail(&di->dt.di.dr_children, dr);
1253                        mutex_exit(&di->dt.di.dr_mtx);
1254                        dr->dr_parent = di;
1255                }
1256                mutex_exit(&db->db_mtx);
1257        } else {
1258                ASSERT(db->db_level+1 == dn->dn_nlevels);
1259                ASSERT(db->db_blkid < dn->dn_nblkptr);
1260                ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1261                mutex_enter(&dn->dn_mtx);
1262                ASSERT(!list_link_active(&dr->dr_dirty_node));
1263                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1264                mutex_exit(&dn->dn_mtx);
1265                if (drop_struct_lock)
1266                        rw_exit(&dn->dn_struct_rwlock);
1267        }

1269        dnode_setdirty(dn, tx);
1270        DB_DNODE_EXIT(db);
1271        return (dr);
1272 }

1274 /*
1275  * Undirty a buffer in the transaction group referenced by the given
1276  * transaction.  Return whether this evicted the dbuf.
 641  * Return TRUE if this evicted the dbuf.
1277  */
1278 static boolean_t
1279 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1280 {
1281         dnode_t *dn;
1282         uint64_t txg = tx->tx_txg;
1283         dbuf_dirty_record_t *dr, **drp;

1285         ASSERT(txg != 0);
1286         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1287         ASSERT0(db->db_level);
1288         ASSERT(MUTEX_HELD(&db->db_mtx));

1290         /*
1291          * If this buffer is not dirty, we're done.
1292          */
1293         for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1294                 if (dr->dr_txg <= txg)
1295                         break;
1296         if (dr == NULL || dr->dr_txg < txg)
1297                 return (B_FALSE);
1298         ASSERT(dr->dr_txg == txg);
1299         ASSERT(dr->dr_dbuf == db);
```



```
1301         DB_DNODE_ENTER(db);
1302         dn = DB_DNODE(db);

1304         /*
1305          * Note:  This code will probably work even if there are concurrent
1306          * holders, but it is untested in that scenerio, as the ZPL and
1307          * ztest have additional locking (the range locks) that prevents
1308          * that type of concurrent access.
1309          */
1310         ASSERT3U(refcount_count(&db->db_holds), ==, db->db_dirtycnt);

1312         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1314         ASSERT(db->db.db_size != 0);

1316         /* XXX would be nice to fix up dn_towrite_space[] */

1318         *drp = dr->dr_next;

1320         /*
1321          * Note that there are three places in dbuf_dirty()
1322          * where this dirty record may be put on a list.
1323          * Make sure to do a list_remove corresponding to
1324          * every one of those list_insert calls.
1325          */
1326         if (dr->dr_parent) {
1327                 mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1328                 list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1329                 mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1330         } else if (db->db_blkid == DMU_SPILL_BLKID ||
1331             db->db_level+1 == dn->dn_nlevels) {
1332                 ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1333                 mutex_enter(&dn->dn_mtx);
1334                 list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1335                 mutex_exit(&dn->dn_mtx);
1336         }
1337         DB_DNODE_EXIT(db);

1339         if (db->db_state != DB_NOFILL) {
1340                 dbuf_unoverride(dr);

1342                 ASSERT(db->db_buf != NULL);
1343                 ASSERT(dr->dt.dl.dr_data != NULL);
1344                 if (dr->dt.dl.dr_data != db->db_buf)
1345                         VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1346         }
1347         kmem_free(dr, sizeof (dbuf_dirty_record_t));

1349         ASSERT(db->db_dirtycnt > 0);
1350         db->db_dirtycnt -= 1;

1352         if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1353                 arc_buf_t *buf = db->db_buf;

1355                 ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1356                 dbuf_set_data(db, NULL);
1357                 VERIFY(arc_buf_remove_ref(buf, db));
1358                 dbuf_evict(db);
1359                 return (B_TRUE);
1360         }

1362         return (B_FALSE);
1363 }
_____*unchanged_portion_omitted_*
```

```
2221 static void
2222 dbuf_sync_indirect(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2223 {
2224         dmu_buf_impl_t *db = dr->dr_dbuf;
2225         dnode_t *dn;
2226         zio_t *zio;

2228         ASSERT(dmu_tx_is_syncing(tx));

2230         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2232         mutex_enter(&db->db_mtx);

2234         ASSERT(db->db_level > 0);
2235         DBUF_VERIFY(db);

2237         /* Read the block if it hasn't been read yet. */
2238 #endif /* ! codereview */
2239         if (db->db_buf == NULL) {
2240                 mutex_exit(&db->db_mtx);
2241                 (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED);
2242                 mutex_enter(&db->db_mtx);
2243         }
2244         ASSERT3U(db->db_state, ==, DB_CACHED);
2245         ASSERT(db->db_buf != NULL);

2247         DB_DNODE_ENTER(db);
2248         dn = DB_DNODE(db);
2249         /* Indirect block size must match what the dnode thinks it is. */
2250 #endif /* ! codereview */
2251         ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2252         dbuf_check_blkptr(dn, db);
2253         DB_DNODE_EXIT(db);

2255         /* Provide the pending dirty record to child dbufs */
2256 #endif /* ! codereview */
2257         db->db_data_pending = dr;

2259         mutex_exit(&db->db_mtx);
2260         dbuf_write(dr, db->db_buf, tx);

2262         zio = dr->dr_zio;
2263         mutex_enter(&dr->dt.di.dr_mtx);
2264         dbuf_sync_list(&dr->dt.di.dr_children, tx);
2265         ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2266         mutex_exit(&dr->dt.di.dr_mtx);
2267         zio_nowait(zio);
2268 }

2270 static void
2271 dbuf_sync_leaf(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2272 {
2273         arc_buf_t **datap = &dr->dt.dl.dr_data;
2274         dmu_buf_impl_t *db = dr->dr_dbuf;
2275         dnode_t *dn;
2276         objset_t *os;
2277         uint64_t txg = tx->tx_txg;

2279         ASSERT(dmu_tx_is_syncing(tx));

2281         dprintf_dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);

2283         mutex_enter(&db->db_mtx);
2284         /*
2285          * To be synced, we must be dirtied.  But we
2286          * might have been freed after the dirty.
```



```
2287          */
2288         if (db->db_state == DB_UNCACHED) {
2289                 /* This buffer has been freed since it was dirtied */
2290                 ASSERT(db->db.db_data == NULL);
2291         } else if (db->db_state == DB_FILL) {
2292                 /* This buffer was freed and is now being re-filled */
2293                 ASSERT(db->db.db_data != dr->dt.dl.dr_data);
2294         } else {
2295                 ASSERT(db->db_state == DB_CACHED || db->db_state == DB_NOFILL);
2296         }
2297         DBUF_VERIFY(db);

2299         DB_DNODE_ENTER(db);
2300         dn = DB_DNODE(db);

2302         if (db->db_blkid == DMU_SPILL_BLKID) {
2303                 mutex_enter(&dn->dn_mtx);
2304                 dn->dn_phys->dn_flags |= DNODE_FLAG_SPILL_BLKPTR;
2305                 mutex_exit(&dn->dn_mtx);
2306         }

2308         /*
2309          * If this is a bonus buffer, simply copy the bonus data into the
2310          * dnode.  It will be written out when the dnode is synced (and it
2311          * will be synced, since it must have been dirty for dbuf_sync to
2312          * be called).
2313          */
2314         if (db->db_blkid == DMU_BONUS_BLKID) {
2315                 dbuf_dirty_record_t **drp;

2317                 ASSERT(*datap != NULL);
2318                 ASSERT0(db->db_level);
2319                 ASSERT3U(dn->dn_phys->dn_bonuslen, <=, DN_MAX_BONUSLEN);
2320                 bcopy(*datap, DN_BONUS(dn->dn_phys), dn->dn_phys->dn_bonuslen);
2321                 DB_DNODE_EXIT(db);

2323                 if (*datap != db->db.db_data) {
2324                         zio_buf_free(*datap, DN_MAX_BONUSLEN);
2325                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
2326                 }
2327                 db->db_data_pending = NULL;
2328                 drp = &db->db_last_dirty;
2329                 while (*drp != dr)
2330                         drp = &(*drp)->dr_next;
2331                 ASSERT(dr->dr_next == NULL);
2332                 ASSERT(dr->dr_dbuf == db);
2333                 *drp = dr->dr_next;
2334                 kmem_free(dr, sizeof (dbuf_dirty_record_t));
2335                 ASSERT(db->db_dirtycnt > 0);
2336                 db->db_dirtycnt -= 1;
2337                 dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2338                 return;
2339         }

2341         os = dn->dn_objset;

2343         /*
2344          * This function may have dropped the db_mtx lock allowing a dmu_sync
2345          * operation to sneak in. As a result, we need to ensure that we
2346          * don't check the dr_override_state until we have returned from
2347          * dbuf_check_blkptr.
2348          */
2349         dbuf_check_blkptr(dn, db);

2351         /*
2352          * If this buffer is in the middle of an immediate write,
```

```
2353          * wait for the synchronous IO to complete.
2354          */
2355         while (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC) {
2356                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
2357                 cv_wait(&db->db_changed, &db->db_mtx);
2358                 ASSERT(dr->dt.dl.dr_override_state != DR_NOT_OVERRIDDEN);
2359         }

2361         if (db->db_state != DB_NOFILL &&
2362             dn->dn_object != DMU_META_DNODE_OBJECT &&
2363             refcount_count(&db->db_holds) > 1 &&
2364             dr->dt.dl.dr_override_state != DR_OVERRIDDEN &&
2365             *datap == db->db_buf) {
2366                 /*
2367                  * If this buffer is currently "in use" (i.e., there
2368                  * are active holds and db_data still references it),
2369                  * then make a copy before we start the write so that
2370                  * any modifications from the open txg will not leak
2371                  * into this write.
2372                  *
2373                  * NOTE: this copy does not need to be made for
2374                  * objects only modified in the syncing context (e.g.
2375                  * DNONE_DNODE blocks).
2376                  */
2377                 int blksz = arc_buf_size(*datap);
2378                 arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
2379                 *datap = arc_buf_alloc(os->os_spa, blksz, db, type);
2380                 bcopy(db->db.db_data, (*datap)->b_data, blksz);
2381         }
2382         db->db_data_pending = dr;

2384         mutex_exit(&db->db_mtx);

2386         dbuf_write(dr, *datap, tx);

2388         ASSERT(!list_link_active(&dr->dr_dirty_node));
2389         if (dn->dn_object == DMU_META_DNODE_OBJECT) {
2390                 list_insert_tail(&dn->dn_dirty_records[txg&TXG_MASK], dr);
2391                 DB_DNODE_EXIT(db);
2392         } else {
2393                 /*
2394                  * Although zio_nowait() does not "wait for an IO", it does
2395                  * initiate the IO. If this is an empty write it seems plausible
2396                  * that the IO could actually be completed before the nowait
2397                  * returns. We need to DB_DNODE_EXIT() first in case
2398                  * zio_nowait() invalidates the dbuf.
2399                  */
2400                 DB_DNODE_EXIT(db);
2401                 zio_nowait(dr->dr_zio);
2402         }
2403 }

2405 void
2406 dbuf_sync_list(list_t *list, dmu_tx_t *tx)
2407 {
2408         dbuf_dirty_record_t *dr;

2410         while (dr = list_head(list)) {
2411                 if (dr->dr_zio != NULL) {
2412                         /*
2413                          * If we find an already initialized zio then we
2414                          * are processing the meta-dnode, and we have finished.
2415                          * The dbufs for all dnodes are put back on the list
2416                          * during processing, so that we can zio_wait()
2417                          * these IOs after initiating all child IOs.
2418                          */
```



```
2419                         ASSERT3U(dr->dr_dbuf->db.db_object, ==,
2420                             DMU_META_DNODE_OBJECT);
2421                         break;
2422                 }
2423                 list_remove(list, dr);
2424                 if (dr->dr_dbuf->db_level > 0)
2425                         dbuf_sync_indirect(dr, tx);
2426                 else
2427                         dbuf_sync_leaf(dr, tx);
2428         }
2429 }

2431 /* ARGSUSED */
2432 static void
2433 dbuf_write_ready(zio_t *zio, arc_buf_t *buf, void *vdb)
2434 {
2435         dmu_buf_impl_t *db = vdb;
2436         dnode_t *dn;
2437         blkptr_t *bp = zio->io_bp;
2438         blkptr_t *bp_orig = &zio->io_bp_orig;
2439         spa_t *spa = zio->io_spa;
2440         int64_t delta;
2441         uint64_t fill = 0;
2442         int i;

2444         ASSERT(db->db_blkptr == bp);

2446         DB_DNODE_ENTER(db);
2447         dn = DB_DNODE(db);
2448         delta = bp_get_dsize_sync(spa, bp) - bp_get_dsize_sync(spa, bp_orig);
2449         dnode_diduse_space(dn, delta - zio->io_prev_space_delta);
2450         zio->io_prev_space_delta = delta;

2452         if (BP_IS_HOLE(bp)) {
2453                 ASSERT(bp->blk_fill == 0);
2454                 DB_DNODE_EXIT(db);
2455                 return;
2456         }

2458         ASSERT((db->db_blkid != DMU_SPILL_BLKID &&
2459             BP_GET_TYPE(bp) == dn->dn_type) ||
2460             (db->db_blkid == DMU_SPILL_BLKID &&
2461             BP_GET_TYPE(bp) == dn->dn_bonustype));
2462         ASSERT(BP_GET_LEVEL(bp) == db->db_level);

2464         mutex_enter(&db->db_mtx);

2466 #ifdef ZFS_DEBUG
2467         if (db->db_blkid == DMU_SPILL_BLKID) {
2468                 ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2469                 ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2470                     db->db_blkptr == &dn->dn_phys->dn_spill);
2471         }
2472 #endif

2474         if (db->db_level == 0) {
2475                 mutex_enter(&dn->dn_mtx);
2476                 if (db->db_blkid > dn->dn_phys->dn_maxblkid &&
2477                     db->db_blkid != DMU_SPILL_BLKID)
2478                         dn->dn_phys->dn_maxblkid = db->db_blkid;
2479                 mutex_exit(&dn->dn_mtx);

2481                 if (dn->dn_type == DMU_OT_DNODE) {
2482                         dnode_phys_t *dnp = db->db.db_data;
2483                         for (i = db->db.db_size >> DNODE_SHIFT; i > 0;
2484                             i--, dnp++) {
```

```
2485                                if (dnp->dn_type != DMU_OT_NONE)
2486                                        fill++;
2487                        }
2488                } else {
2489                        fill = 1;
2490                }
2491        } else {
2492                blkptr_t *ibp = db->db.db_data;
2493                ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2494                for (i = db->db.db_size >> SPA_BLKPTRSHIFT; i > 0; i--, ibp++) {
2495                        if (BP_IS_HOLE(ibp))
2496                                continue;
2497                        fill += ibp->blk_fill;
2498                }
2499        }
2500        DB_DNODE_EXIT(db);

2502        bp->blk_fill = fill;

2504        mutex_exit(&db->db_mtx);
2505 }

2507 /* ARGSUSED */
2508 static void
2509 dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
2510 {
2511        dmu_buf_impl_t *db = vdb;
2512        blkptr_t *bp = zio->io_bp;
2513        blkptr_t *bp_orig = &zio->io_bp_orig;
2514        uint64_t txg = zio->io_txg;
2515        dbuf_dirty_record_t **drp, *dr;

2517        ASSERT0(zio->io_error);
2518        ASSERT(db->db_blkptr == bp);

2520        /*
2521         * For nopwrites and rewrites we ensure that the bp matches our
2522         * original and bypass all the accounting.
2523         */
2524        if (zio->io_flags & (ZIO_FLAG_IO_REWRITE | ZIO_FLAG_NOPWRITE)) {
2525                ASSERT(BP_EQUAL(bp, bp_orig));
2526        } else {
2527                objset_t *os;
2528                dsl_dataset_t *ds;
2529                dmu_tx_t *tx;

2531                DB_GET_OBJSET(&os, db);
2532                ds = os->os_dsl_dataset;
2533                tx = os->os_synctx;

2535                (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
2536                dsl_dataset_block_born(ds, bp, tx);
2537        }

2539        mutex_enter(&db->db_mtx);

2541        DBUF_VERIFY(db);

2543        drp = &db->db_last_dirty;
2544        while ((dr = *drp) != db->db_data_pending)
2545                drp = &dr->dr_next;
2546        ASSERT(!list_link_active(&dr->dr_dirty_node));
2547        ASSERT(dr->dr_txg == txg);
2548        ASSERT(dr->dr_dbuf == db);
2549        ASSERT(dr->dr_next == NULL);
2550        *drp = dr->dr_next;
```



```
2552 #ifdef ZFS_DEBUG
2553        if (db->db_blkid == DMU_SPILL_BLKID) {
2554                dnode_t *dn;

2556                DB_DNODE_ENTER(db);
2557                dn = DB_DNODE(db);
2558                ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2559                ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2560                    db->db_blkptr == &dn->dn_phys->dn_spill);
2561                DB_DNODE_EXIT(db);
2562        }
2563 #endif

2565        if (db->db_level == 0) {
2566                ASSERT(db->db_blkid != DMU_BONUS_BLKID);
2567                ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
2568                if (db->db_state != DB_NOFILL) {
2569                        if (dr->dt.dl.dr_data != db->db_buf)
2570                                VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
2571                                    db));
2572                        else if (!arc_released(db->db_buf))
2573                                arc_set_callback(db->db_buf, dbuf_do_evict, db);
2574                }
2575        } else {
2576                dnode_t *dn;

2578                DB_DNODE_ENTER(db);
2579                dn = DB_DNODE(db);
2580                ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2581                ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2582                if (!BP_IS_HOLE(db->db_blkptr)) {
2583                        int epbs =
2584                            dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
2585                        ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
2586                            db->db.db_size);
2587                        ASSERT3U(dn->dn_phys->dn_maxblkid
2588                            >> (db->db_level * epbs), >=, db->db_blkid);
2589                        arc_set_callback(db->db_buf, dbuf_do_evict, db);
2590                }
2591                DB_DNODE_EXIT(db);
2592                mutex_destroy(&dr->dt.di.dr_mtx);
2593                list_destroy(&dr->dt.di.dr_children);
2594        }
2595        kmem_free(dr, sizeof (dbuf_dirty_record_t));

2597        cv_broadcast(&db->db_changed);
2598        ASSERT(db->db_dirtycnt > 0);
2599        db->db_dirtycnt -= 1;
2600        db->db_data_pending = NULL;
2601        dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2602 }

2604 static void
2605 dbuf_write_nofill_ready(zio_t *zio)
2606 {
2607        dbuf_write_ready(zio, NULL, zio->io_private);
2608 }

2610 static void
2611 dbuf_write_nofill_done(zio_t *zio)
2612 {
2613        dbuf_write_done(zio, NULL, zio->io_private);
2614 }

2616 static void
```

```
dbuf_write_override_ready(zio_t *zio)
{
        dbuf_dirty_record_t *dr = zio->io_private;
        dmu_buf_impl_t *db = dr->dr_dbuf;

        dbuf_write_ready(zio, NULL, db);
}

static void
dbuf_write_override_done(zio_t *zio)
{
        dbuf_dirty_record_t *dr = zio->io_private;
        dmu_buf_impl_t *db = dr->dr_dbuf;
        blkptr_t *obp = &dr->dt.dl.dr_overridden_by;

        mutex_enter(&db->db_mtx);
        if (!BP_EQUAL(zio->io_bp, obp)) {
                if (!BP_IS_HOLE(obp))
                        dsl_free(spa_get_dsl(zio->io_spa), zio->io_txg, obp);
                arc_release(dr->dt.dl.dr_data, db);
        }
        mutex_exit(&db->db_mtx);

        dbuf_write_done(zio, NULL, db);
}

/* Issue I/O to commit a dirty buffer to disk. */
#endif /* ! codereview */
static void
dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db = dr->dr_dbuf;
        dnode_t *dn;
        objset_t *os;
        dmu_buf_impl_t *parent = db->db_parent;
        uint64_t txg = tx->tx_txg;
        zbookmark_t zb;
        zio_prop_t zp;
        zio_t *zio;
        int wp_flag = 0;

        DB_DNODE_ENTER(db);
        dn = DB_DNODE(db);
        os = dn->dn_objset;

        if (db->db_state != DB_NOFILL) {
                if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
                        /*
                         * Private object buffers are released here rather
                         * than in dbuf_dirty() since they are only modified
                         * in the syncing context and we don't want the
                         * overhead of making multiple copies of the data.
                         */
                        if (BP_IS_HOLE(db->db_blkptr)) {
                                arc_buf_thaw(data);
                        } else {
                                dbuf_release_bp(db);
                        }
                }
        }

        if (parent != dn->dn_dbuf) {
                /* Our parent is an indirect block. */
                /* We have a dirty parent that has been scheduled for write. */
#endif /* ! codereview */
                ASSERT(parent && parent->db_data_pending);
```



```
                /* Our parent's buffer is one level closer to the dnode. */
#endif /* ! codereview */
                ASSERT(db->db_level == parent->db_level-1);
                /*
                 * We're about to modify our parent's db_data by modifying
                 * our block pointer, so the parent must be released.
                 */
#endif /* ! codereview */
                ASSERT(arc_released(parent->db_buf));
                zio = parent->db_data_pending->dr_zio;
        } else {
                /* Our parent is the dnode itself. */
#endif /* ! codereview */
                ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
                    db->db_blkid != DMU_SPILL_BLKID) ||
                    (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
                if (db->db_blkid != DMU_SPILL_BLKID)
                        ASSERT3P(db->db_blkptr, ==,
                            &dn->dn_phys->dn_blkptr[db->db_blkid]);
                zio = dn->dn_zio;
        }

        ASSERT(db->db_level == 0 || data == db->db_buf);
        ASSERT3U(db->db_blkptr->blk_birth, <=, txg);
        ASSERT(zio);

        SET_BOOKMARK(&zb, os->os_dsl_dataset ?
            os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
            db->db.db_object, db->db_level, db->db_blkid);

        if (db->db_blkid == DMU_SPILL_BLKID)
                wp_flag = WP_SPILL;
        wp_flag |= (db->db_state == DB_NOFILL) ? WP_NOFILL : 0;

        dmu_write_policy(os, dn, db->db_level, wp_flag, &zp);
        DB_DNODE_EXIT(db);

        if (db->db_level == 0 && dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
                ASSERT(db->db_state != DB_NOFILL);
                dr->dr_zio = zio_write(zio, os->os_spa, txg,
                    db->db_blkptr, data->b_data, arc_buf_size(data), &zp,
                    dbuf_write_override_ready, dbuf_write_override_done, dr,
                    ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
                mutex_enter(&db->db_mtx);
                dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
                zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
                    dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
                mutex_exit(&db->db_mtx);
        } else if (db->db_state == DB_NOFILL) {
                ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF);
                dr->dr_zio = zio_write(zio, os->os_spa, txg,
                    db->db_blkptr, NULL, db->db.db_size, &zp,
                    dbuf_write_nofill_ready, dbuf_write_nofill_done, db,
                    ZIO_PRIORITY_ASYNC_WRITE,
                    ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
        } else {
                ASSERT(arc_released(data));
                dr->dr_zio = arc_write(zio, os->os_spa, txg,
                    db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db), &zp,
                    dbuf_write_ready, dbuf_write_done, db,
                    ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
        }
}
```

****
44142 Tue Apr 23 14:09:35 2013
new/usr/src/uts/common/fs/zfs/dmu.c
3741 zfs needs better comments
Submitted by: Will Andrews <willa@spectralogic.com>
Submitted by: Justin Gibbs <justing@spectralogic.com>
Submitted by: Alan Somers <alans@spectralogic.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
****
_____***unchanged_portion_omitted_***

```
1824 void
1825 dmu_fini(void)
1826 {
1827         arc_fini(); /* arc depends on l2arc, so arc must go first */
1827         arc_fini();
1828         l2arc_fini();
1829         zfetch_fini();
1830         dbuf_fini();
1831         dnode_fini();
1832         dmu_objset_fini();
1833         xuio_stat_fini();
1834         sa_cache_fini();
1835         zfs_dbgmsg_fini();
1836 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35492 Tue Apr 23 14:09:36 2013
new/usr/src/uts/common/fs/zfs/dmu_tx.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1007 static void
1008 dmu_tx_unassign(dmu_tx_t *tx)
1009 {
1010         dmu_tx_hold_t *txh;

1012         if (tx->tx_txg == 0)
1013                 return;

1015         txg_rele_to_quiesce(&tx->tx_txgh);

1017         /*
1018          * Walk the transaction's hold list, removing the hold on the
1019          * associated dnode, and notifying waiters if the refcount drops to 0.
1020          */
1021 #endif /* ! codereview */
1022         for (txh = list_head(&tx->tx_holds); txh != tx->tx_needassign_txh;
1023             txh = list_next(&tx->tx_holds, txh)) {
1024                 dnode_t *dn = txh->txh_dnode;

1026                 if (dn == NULL)
1027                         continue;
1028                 mutex_enter(&dn->dn_mtx);
1029                 ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);

1031                 if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
1032                         dn->dn_assigned_txg = 0;
1033                         cv_broadcast(&dn->dn_notxholds);
1034                 }
1035                 mutex_exit(&dn->dn_mtx);
1036         }

1038         txg_rele_to_sync(&tx->tx_txgh);

1040         tx->tx_lasttried_txg = tx->tx_txg;
1041         tx->tx_txg = 0;
1042 }

1044 /*
1045  * Assign tx to a transaction group.  txg_how can be one of:
1046  *
1047  * (1)  TXG_WAIT.  If the current open txg is full, waits until there's
1048  *      a new one.  This should be used when you're not holding locks.
1049  *      It will only fail if we're truly out of space (or over quota).
1050  *
1051  * (2)  TXG_NOWAIT.  If we can't assign into the current open txg without
1052  *      blocking, returns immediately with ERESTART.  This should be used
1053  *      whenever you're holding locks.  On an ERESTART error, the caller
1054  *      should drop locks, do a dmu_tx_wait(tx), and try again.
1055  */
1056 int
1057 dmu_tx_assign(dmu_tx_t *tx, txg_how_t txg_how)
1058 {
1059         int err;

1061         ASSERT(tx->tx_txg == 0);
```



```
1062         ASSERT(txg_how == TXG_WAIT || txg_how == TXG_NOWAIT);
1063         ASSERT(!dsl_pool_sync_context(tx->tx_pool));

1065         /* If we might wait, we must not hold the config lock. */
1066         ASSERT(txg_how != TXG_WAIT || !dsl_pool_config_held(tx->tx_pool));

1068         while ((err = dmu_tx_try_assign(tx, txg_how)) != 0) {
1069                 dmu_tx_unassign(tx);

1071                 if (err != ERESTART || txg_how != TXG_WAIT)
1072                         return (err);

1074                 dmu_tx_wait(tx);
1075         }

1077         txg_rele_to_quiesce(&tx->tx_txgh);

1079         return (0);
1080 }

1082 void
1083 dmu_tx_wait(dmu_tx_t *tx)
1084 {
1085         spa_t *spa = tx->tx_pool->dp_spa;

1087         ASSERT(tx->tx_txg == 0);
1088         ASSERT(!dsl_pool_config_held(tx->tx_pool));

1090         /*
1091          * It's possible that the pool has become active after this thread
1092          * has tried to obtain a tx. If that's the case then his
1093          * tx_lasttried_txg would not have been assigned.
1094          */
1095         if (spa_suspended(spa) || tx->tx_lasttried_txg == 0) {
1096                 txg_wait_synced(tx->tx_pool, spa_last_synced_txg(spa) + 1);
1097         } else if (tx->tx_needassign_txh) {
1098                 dnode_t *dn = tx->tx_needassign_txh->txh_dnode;

1100                 mutex_enter(&dn->dn_mtx);
1101                 while (dn->dn_assigned_txg == tx->tx_lasttried_txg - 1)
1102                         cv_wait(&dn->dn_notxholds, &dn->dn_mtx);
1103                 mutex_exit(&dn->dn_mtx);
1104                 tx->tx_needassign_txh = NULL;
1105         } else {
1106                 txg_wait_open(tx->tx_pool, tx->tx_lasttried_txg + 1);
1107         }
1108 }

1110 void
1111 dmu_tx_willuse_space(dmu_tx_t *tx, int64_t delta)
1112 {
1113 #ifdef ZFS_DEBUG
1114         if (tx->tx_dir == NULL || delta == 0)
1115                 return;

1117         if (delta > 0) {
1118                 ASSERT3U(refcount_count(&tx->tx_space_written) + delta, <=,
1119                     tx->tx_space_towrite);
1120                 (void) refcount_add_many(&tx->tx_space_written, delta, NULL);
1121         } else {
1122                 (void) refcount_add_many(&tx->tx_space_freed, -delta, NULL);
1123         }
1124 #endif
1125 }

1127 void
```

```
1128 dmu_tx_commit(dmu_tx_t *tx)
1129 {
1130         dmu_tx_hold_t *txh;

1132         ASSERT(tx->tx_txg != 0);

1134         /*
1135          * Go through the transaction's hold list and remove holds on
1136          * associated dnodes, notifying waiters if no holds remain.
1137          */
1138 #endif /* ! codereview */
1139         while (txh = list_head(&tx->tx_holds)) {
1140                 dnode_t *dn = txh->txh_dnode;

1142                 list_remove(&tx->tx_holds, txh);
1143                 kmem_free(txh, sizeof (dmu_tx_hold_t));
1144                 if (dn == NULL)
1145                         continue;
1146                 mutex_enter(&dn->dn_mtx);
1147                 ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);

1149                 if (refcount_remove(&dn->dn_tx_holds, tx) == 0) {
1150                         dn->dn_assigned_txg = 0;
1151                         cv_broadcast(&dn->dn_notxholds);
1152                 }
1153                 mutex_exit(&dn->dn_mtx);
1154                 dnode_rele(dn, tx);
1155         }

1157         if (tx->tx_tempreserve_cookie)
1158                 dsl_dir_tempreserve_clear(tx->tx_tempreserve_cookie, tx);

1160         if (!list_is_empty(&tx->tx_callbacks))
1161                 txg_register_callbacks(&tx->tx_txgh, &tx->tx_callbacks);

1163         if (tx->tx_anyobj == FALSE)
1164                 txg_rele_to_sync(&tx->tx_txgh);

1166         list_destroy(&tx->tx_callbacks);
1167         list_destroy(&tx->tx_holds);
1168 #ifdef ZFS_DEBUG
1169         dprintf("towrite=%llu written=%llu tofree=%llu freed=%llu\n",
1170             tx->tx_space_towrite, refcount_count(&tx->tx_space_written),
1171             tx->tx_space_tofree, refcount_count(&tx->tx_space_freed));
1172         refcount_destroy_many(&tx->tx_space_written,
1173             refcount_count(&tx->tx_space_written));
1174         refcount_destroy_many(&tx->tx_space_freed,
1175             refcount_count(&tx->tx_space_freed));
1176 #endif
1177         kmem_free(tx, sizeof (dmu_tx_t));
1178 }

1180 void
1181 dmu_tx_abort(dmu_tx_t *tx)
1182 {
1183         dmu_tx_hold_t *txh;

1185         ASSERT(tx->tx_txg == 0);

1187         while (txh = list_head(&tx->tx_holds)) {
1188                 dnode_t *dn = txh->txh_dnode;

1190                 list_remove(&tx->tx_holds, txh);
1191                 kmem_free(txh, sizeof (dmu_tx_hold_t));
1192                 if (dn != NULL)
1193                         dnode_rele(dn, tx);
```



```
1194         }

1196         /*
1197          * Call any registered callbacks with an error code.
1198          */
1199         if (!list_is_empty(&tx->tx_callbacks))
1200                 dmu_tx_do_callbacks(&tx->tx_callbacks, ECANCELED);

1202         list_destroy(&tx->tx_callbacks);
1203         list_destroy(&tx->tx_holds);
1204 #ifdef ZFS_DEBUG
1205         refcount_destroy_many(&tx->tx_space_written,
1206             refcount_count(&tx->tx_space_written));
1207         refcount_destroy_many(&tx->tx_space_freed,
1208             refcount_count(&tx->tx_space_freed));
1209 #endif
1210         kmem_free(tx, sizeof (dmu_tx_t));
1211 }

1213 uint64_t
1214 dmu_tx_get_txg(dmu_tx_t *tx)
1215 {
1216         ASSERT(tx->tx_txg != 0);
1217         return (tx->tx_txg);
1218 }

1220 dsl_pool_t *
1221 dmu_tx_pool(dmu_tx_t *tx)
1222 {
1223         ASSERT(tx->tx_pool != NULL);
1224         return (tx->tx_pool);
1225 }


1228 void
1229 dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *func, void *data)
1230 {
1231         dmu_tx_callback_t *dcb;

1233         dcb = kmem_alloc(sizeof (dmu_tx_callback_t), KM_SLEEP);

1235         dcb->dcb_func = func;
1236         dcb->dcb_data = data;

1238         list_insert_tail(&tx->tx_callbacks, dcb);
1239 }

1241 /*
1242  * Call all the commit callbacks on a list, with a given error code.
1243  */
1244 void
1245 dmu_tx_do_callbacks(list_t *cb_list, int error)
1246 {
1247         dmu_tx_callback_t *dcb;

1249         while (dcb = list_head(cb_list)) {
1250                 list_remove(cb_list, dcb);
1251                 dcb->dcb_func(dcb->dcb_data, error);
1252                 kmem_free(dcb, sizeof (dmu_tx_callback_t));
1253         }
1254 }

1256 /*
1257  * Interface to hold a bunch of attributes.
1258  * used for creating new files.
1259  * attrsize is the total size of all attributes
```

```
1260  * to be added during object creation
1261  *
1262  * For updating/adding a single attribute dmu_tx_hold_sa() should be used.
1263  */
1264
1265 /*
1266  * hold necessary attribute name for attribute registration.
1267  * should be a very rare case where this is needed.  If it does
1268  * happen it would only happen on the first write to the file system.
1269  */
1270 static void
1271 dmu_tx_sa_registration_hold(sa_os_t *sa, dmu_tx_t *tx)
1272 {
1273         int i;
1274
1275         if (!sa->sa_need_attr_registration)
1276                 return;
1277
1278         for (i = 0; i != sa->sa_num_attrs; i++) {
1279                 if (!sa->sa_attr_table[i].sa_registered) {
1280                         if (sa->sa_reg_attr_obj)
1281                                 dmu_tx_hold_zap(tx, sa->sa_reg_attr_obj,
1282                                     B_TRUE, sa->sa_attr_table[i].sa_name);
1283                         else
1284                                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT,
1285                                     B_TRUE, sa->sa_attr_table[i].sa_name);
1286                 }
1287         }
1288 }
1289
1290
1291 void
1292 dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object)
1293 {
1294         dnode_t *dn;
1295         dmu_tx_hold_t *txh;
1296
1297         txh = dmu_tx_hold_object_impl(tx, tx->tx_objset, object,
1298             THT_SPILL, 0, 0);
1299
1300         dn = txh->txh_dnode;
1301
1302         if (dn == NULL)
1303                 return;
1304
1305         /* If blkptr doesn't exist then add space to towrite */
1306         if (!(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR)) {
1307                 txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
1308         } else {
1309                 blkptr_t *bp;
1310
1311                 bp = &dn->dn_phys->dn_spill;
1312                 if (dsl_dataset_block_freeable(dn->dn_objset->os_dsl_dataset,
1313                     bp, bp->blk_birth))
1314                         txh->txh_space_tooverwrite += SPA_MAXBLOCKSIZE;
1315                 else
1316                         txh->txh_space_towrite += SPA_MAXBLOCKSIZE;
1317                 if (!BP_IS_HOLE(bp))
1318                         txh->txh_space_tounref += SPA_MAXBLOCKSIZE;
1319         }
1320 }
1321
1322 void
1323 dmu_tx_hold_sa_create(dmu_tx_t *tx, int attrsize)
1324 {
1325         sa_os_t *sa = tx->tx_objset->os_sa;
```



```
1326
1327         dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);
1328
1329         if (tx->tx_objset->os_sa->sa_master_obj == 0)
1330                 return;
1331
1332         if (tx->tx_objset->os_sa->sa_layout_attr_obj)
1333                 dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
1334         else {
1335                 dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
1336                 dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
1337                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1338                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1339         }
1340
1341         dmu_tx_sa_registration_hold(sa, tx);
1342
1343         if (attrsize <= DN_MAX_BONUSLEN && !sa->sa_force_spill)
1344                 return;
1345
1346         (void) dmu_tx_hold_object_impl(tx, tx->tx_objset, DMU_NEW_OBJECT,
1347             THT_SPILL, 0, 0);
1348 }
1349
1350 /*
1351  * Hold SA attribute
1352  *
1353  * dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *, attribute, add, size)
1354  *
1355  * variable_size is the total size of all variable sized attributes
1356  * passed to this function.  It is not the total size of all
1357  * variable size attributes that *may* exist on this object.
1358  */
1359 void
1360 dmu_tx_hold_sa(dmu_tx_t *tx, sa_handle_t *hdl, boolean_t may_grow)
1361 {
1362         uint64_t object;
1363         sa_os_t *sa = tx->tx_objset->os_sa;
1364
1365         ASSERT(hdl != NULL);
1366
1367         object = sa_handle_object(hdl);
1368
1369         dmu_tx_hold_bonus(tx, object);
1370
1371         if (tx->tx_objset->os_sa->sa_master_obj == 0)
1372                 return;
1373
1374         if (tx->tx_objset->os_sa->sa_reg_attr_obj == 0 ||
1375             tx->tx_objset->os_sa->sa_layout_attr_obj == 0) {
1376                 dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_LAYOUTS);
1377                 dmu_tx_hold_zap(tx, sa->sa_master_obj, B_TRUE, SA_REGISTRY);
1378                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1379                 dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1380         }
1381
1382         dmu_tx_sa_registration_hold(sa, tx);
1383
1384         if (may_grow && tx->tx_objset->os_sa->sa_layout_attr_obj)
1385                 dmu_tx_hold_zap(tx, sa->sa_layout_attr_obj, B_TRUE, NULL);
1386
1387         if (sa->sa_force_spill || may_grow || hdl->sa_spill) {
1388                 ASSERT(tx->tx_txg == 0);
1389                 dmu_tx_hold_spill(tx, object);
1390         } else {
1391                 dmu_buf_impl_t *db = (dmu_buf_impl_t *)hdl->sa_bonus;
```

```
1392                 dnode_t *dn;
1393
1394                 DB_DNODE_ENTER(db);
1395                 dn = DB_DNODE(db);
1396                 if (dn->dn_have_spill) {
1397                         ASSERT(tx->tx_txg == 0);
1398                         dmu_tx_hold_spill(tx, object);
1399                 }
1400                 DB_DNODE_EXIT(db);
1401         }
1402 }
```

```
**********************************************************
   19144 Tue Apr 23 14:09:36 2013
new/usr/src/uts/common/fs/zfs/dmu_zfetch.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/zfs_context.h>
#include <sys/dnode.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_zfetch.h>
#include <sys/dmu.h>
#include <sys/dbuf.h>
#include <sys/kstat.h>

/*
 * I'm against tune-ables, but these should probably exist as tweakable globals
 * until we can get this working the way we want it to.
 */

int zfs_prefetch_disable = 0;

/* max # of streams per zfetch */
uint32_t        zfetch_max_streams = 8;
/* min time before stream reclaim */
uint32_t        zfetch_min_sec_reap = 2;
/* max number of blocks to fetch at a time */
uint32_t        zfetch_block_cap = 256;
/* number of bytes in a array_read at which we stop prefetching (1Mb) */
uint64_t        zfetch_array_rd_sz = 1024 * 1024;

/* forward decls for static routines */
static boolean_t        dmu_zfetch_colinear(zfetch_t *, zstream_t *);
static int              dmu_zfetch_colinear(zfetch_t *, zstream_t *);
static void             dmu_zfetch_dofetch(zfetch_t *, zstream_t *);
static uint64_t         dmu_zfetch_fetch(dnode_t *, uint64_t, uint64_t);
static uint64_t         dmu_zfetch_fetchsz(dnode_t *, uint64_t, uint64_t);
static boolean_t        dmu_zfetch_find(zfetch_t *, zstream_t *, int);
static int              dmu_zfetch_find(zfetch_t *, zstream_t *, int);
```



```
static int              dmu_zfetch_stream_insert(zfetch_t *, zstream_t *);
static zstream_t        *dmu_zfetch_stream_reclaim(zfetch_t *);
static void             dmu_zfetch_stream_remove(zfetch_t *, zstream_t *);
static int              dmu_zfetch_streams_equal(zstream_t *, zstream_t *);

typedef struct zfetch_stats {
        kstat_named_t zfetchstat_hits;
        kstat_named_t zfetchstat_misses;
        kstat_named_t zfetchstat_colinear_hits;
        kstat_named_t zfetchstat_colinear_misses;
        kstat_named_t zfetchstat_stride_hits;
        kstat_named_t zfetchstat_stride_misses;
        kstat_named_t zfetchstat_reclaim_successes;
        kstat_named_t zfetchstat_reclaim_failures;
        kstat_named_t zfetchstat_stream_resets;
        kstat_named_t zfetchstat_stream_noresets;
        kstat_named_t zfetchstat_bogus_streams;
} zfetch_stats_t;
_____unchanged_portion_omitted_

#define ZFETCHSTAT_INCR(stat, val) \
        atomic_add_64(&zfetch_stats.stat.value.ui64, (val));

#define ZFETCHSTAT_BUMP(stat)           ZFETCHSTAT_INCR(stat, 1);

kstat_t         *zfetch_ksp;

/*
 * Given a zfetch structure and a zstream structure, determine whether the
 * blocks to be read are part of a co-linear pair of existing prefetch
 * streams.  If a set is found, coalesce the streams, removing one, and
 * configure the prefetch so it looks for a strided access pattern.
 *
 * In other words: if we find two sequential access streams that are
 * the same length and distance N appart, and this read is N from the
 * last stream, then we are probably in a strided access pattern.  So
 * combine the two sequential streams into a single strided stream.
 *
 * Returns whether co-linear streams were found.
 * If no co-linear streams are found, return NULL.
 */
static boolean_t
static int
dmu_zfetch_colinear(zfetch_t *zf, zstream_t *zh)
{
        zstream_t       *z_walk;
        zstream_t       *z_comp;

        if (! rw_tryenter(&zf->zf_rwlock, RW_WRITER))
                return (0);

        if (zh == NULL) {
                rw_exit(&zf->zf_rwlock);
                return (0);
        }

        for (z_walk = list_head(&zf->zf_stream); z_walk;
            z_walk = list_next(&zf->zf_stream, z_walk)) {
                for (z_comp = list_next(&zf->zf_stream, z_walk); z_comp;
                    z_comp = list_next(&zf->zf_stream, z_comp)) {
                        int64_t         diff;

                        if (z_walk->zst_len != z_walk->zst_stride ||
                            z_comp->zst_len != z_comp->zst_stride) {
                                continue;
                        }
```

```
134                         diff = z_comp->zst_offset - z_walk->zst_offset;
135                         if (z_comp->zst_offset + diff == zh->zst_offset) {
136                                 z_walk->zst_offset = zh->zst_offset;
137                                 z_walk->zst_direction = diff < 0 ? -1 : 1;
138                                 z_walk->zst_stride =
139                                     diff * z_walk->zst_direction;
140                                 z_walk->zst_ph_offset =
141                                     zh->zst_offset + z_walk->zst_stride;
142                                 dmu_zfetch_stream_remove(zf, z_comp);
143                                 mutex_destroy(&z_comp->zst_lock);
144                                 kmem_free(z_comp, sizeof (zstream_t));

146                                 dmu_zfetch_dofetch(zf, z_walk);

148                                 rw_exit(&zf->zf_rwlock);
149                                 return (1);
150                         }

152                         diff = z_walk->zst_offset - z_comp->zst_offset;
153                         if (z_walk->zst_offset + diff == zh->zst_offset) {
154                                 z_walk->zst_offset = zh->zst_offset;
155                                 z_walk->zst_direction = diff < 0 ? -1 : 1;
156                                 z_walk->zst_stride =
157                                     diff * z_walk->zst_direction;
158                                 z_walk->zst_ph_offset =
159                                     zh->zst_offset + z_walk->zst_stride;
160                                 dmu_zfetch_stream_remove(zf, z_comp);
161                                 mutex_destroy(&z_comp->zst_lock);
162                                 kmem_free(z_comp, sizeof (zstream_t));

164                                 dmu_zfetch_dofetch(zf, z_walk);

166                                 rw_exit(&zf->zf_rwlock);
167                                 return (1);
168                         }
169                 }
170         }

172         rw_exit(&zf->zf_rwlock);
173         return (0);
174 }
```

unchanged_portion_omitted_

```
324 /*
325  * given a zfetch and a zstream structure, see if there is an associated zstream
326  * for this block read.  If so, it starts a prefetch for the stream it
327  * located and returns true, otherwise it returns false
328  */
329 static boolean_t
329 static int
330 dmu_zfetch_find(zfetch_t *zf, zstream_t *zh, int prefetched)
331 {
332         zstream_t       *zs;
333         int64_t         diff;
334         int             reset = !prefetched;
335         int             rc = 0;

337         if (zh == NULL)
338                 return (0);

340         /*
341          * XXX: This locking strategy is a bit coarse; however, it's impact has
342          * yet to be tested.  If this turns out to be an issue, it can be
343          * modified in a number of different ways.
344          */
```



```
346         rw_enter(&zf->zf_rwlock, RW_READER);
347 top:

349         for (zs = list_head(&zf->zf_stream); zs;
350             zs = list_next(&zf->zf_stream, zs)) {

352                 /*
353                  * XXX - should this be an assert?
354                  */
355                 if (zs->zst_len == 0) {
356                         /* bogus stream */
357                         ZFETCHSTAT_BUMP(zfetchstat_bogus_streams);
358                         continue;
359                 }

361                 /*
362                  * We hit this case when we are in a strided prefetch stream:
363                  * we will read "len" blocks before "striding".
364                  */
365                 if (zh->zst_offset >= zs->zst_offset &&
366                     zh->zst_offset < zs->zst_offset + zs->zst_len) {
367                         if (prefetched) {
368                                 /* already fetched */
369                                 ZFETCHSTAT_BUMP(zfetchstat_stride_hits);
370                                 rc = 1;
371                                 goto out;
372                         } else {
373                                 ZFETCHSTAT_BUMP(zfetchstat_stride_misses);
374                         }
375                 }

377                 /*
378                  * This is the forward sequential read case: we increment
379                  * len by one each time we hit here, so we will enter this
380                  * case on every read.
381                  */
382                 if (zh->zst_offset == zs->zst_offset + zs->zst_len) {

384                         reset = !prefetched && zs->zst_len > 1;

386                         mutex_enter(&zs->zst_lock);

388                         if (zh->zst_offset != zs->zst_offset + zs->zst_len) {
389                                 mutex_exit(&zs->zst_lock);
390                                 goto top;
391                         }
392                         zs->zst_len += zh->zst_len;
393                         diff = zs->zst_len - zfetch_block_cap;
394                         if (diff > 0) {
395                                 zs->zst_offset += diff;
396                                 zs->zst_len = zs->zst_len > diff ?
397                                     zs->zst_len - diff : 0;
398                         }
399                         zs->zst_direction = ZFETCH_FORWARD;

401                         break;

403                 /*
404                  * Same as above, but reading backwards through the file.
405                  */
406                 } else if (zh->zst_offset == zs->zst_offset - zh->zst_len) {
407                         /* backwards sequential access */

409                         reset = !prefetched && zs->zst_len > 1;
```

```
411                         mutex_enter(&zs->zst_lock);
413                         if (zh->zst_offset != zs->zst_offset - zh->zst_len) {
414                                 mutex_exit(&zs->zst_lock);
415                                 goto top;
416                         }
418                         zs->zst_offset = zs->zst_offset > zh->zst_len ?
419                             zs->zst_offset - zh->zst_len : 0;
420                         zs->zst_ph_offset = zs->zst_ph_offset > zh->zst_len ?
421                             zs->zst_ph_offset - zh->zst_len : 0;
422                         zs->zst_len += zh->zst_len;
424                         diff = zs->zst_len - zfetch_block_cap;
425                         if (diff > 0) {
426                                 zs->zst_ph_offset = zs->zst_ph_offset > diff ?
427                                     zs->zst_ph_offset - diff : 0;
428                                 zs->zst_len = zs->zst_len > diff ?
429                                     zs->zst_len - diff : zs->zst_len;
430                         }
431                         zs->zst_direction = ZFETCH_BACKWARD;
433                         break;
435                 } else if ((zh->zst_offset - zs->zst_offset - zs->zst_stride <
436                     zs->zst_len) && (zs->zst_len != zs->zst_stride)) {
437                         /* strided forward access */
439                         mutex_enter(&zs->zst_lock);
441                         if ((zh->zst_offset - zs->zst_offset - zs->zst_stride >=
442                             zs->zst_len) || (zs->zst_len == zs->zst_stride)) {
443                                 mutex_exit(&zs->zst_lock);
444                                 goto top;
445                         }
447                         zs->zst_offset += zs->zst_stride;
448                         zs->zst_direction = ZFETCH_FORWARD;
450                         break;
452                 } else if ((zh->zst_offset - zs->zst_offset + zs->zst_stride <
453                     zs->zst_len) && (zs->zst_len != zs->zst_stride)) {
454                         /* strided reverse access */
456                         mutex_enter(&zs->zst_lock);
458                         if ((zh->zst_offset - zs->zst_offset + zs->zst_stride >=
459                             zs->zst_len) || (zs->zst_len == zs->zst_stride)) {
460                                 mutex_exit(&zs->zst_lock);
461                                 goto top;
462                         }
464                         zs->zst_offset = zs->zst_offset > zs->zst_stride ?
465                             zs->zst_offset - zs->zst_stride : 0;
466                         zs->zst_ph_offset = (zs->zst_ph_offset >
467                             (2 * zs->zst_stride)) ?
468                             (zs->zst_ph_offset - (2 * zs->zst_stride)) : 0;
469                         zs->zst_direction = ZFETCH_BACKWARD;
471                         break;
472                 }
473         }
475         if (zs) {
476                 if (reset) {
```



```
477                         zstream_t *remove = zs;
479                         ZFETCHSTAT_BUMP(zfetchstat_stream_resets);
480                         rc = 0;
481                         mutex_exit(&zs->zst_lock);
482                         rw_exit(&zf->zf_rwlock);
483                         rw_enter(&zf->zf_rwlock, RW_WRITER);
484                         /*
485                          * Relocate the stream, in case someone removes
486                          * it while we were acquiring the WRITER lock.
487                          */
488                         for (zs = list_head(&zf->zf_stream); zs;
489                             zs = list_next(&zf->zf_stream, zs)) {
490                                 if (zs == remove) {
491                                         dmu_zfetch_stream_remove(zf, zs);
492                                         mutex_destroy(&zs->zst_lock);
493                                         kmem_free(zs, sizeof (zstream_t));
494                                         break;
495                                 }
496                         }
497                 } else {
498                         ZFETCHSTAT_BUMP(zfetchstat_stream_noresets);
499                         rc = 1;
500                         dmu_zfetch_dofetch(zf, zs);
501                         mutex_exit(&zs->zst_lock);
502                 }
503         }
504 out:
505         rw_exit(&zf->zf_rwlock);
506         return (rc);
507 }
```

*unchanged_portion_omitted_*

```
633 /*
634  * This is the prefetch entry point.  It calls all of the other dmu_zfetch
635  * routines to create, delete, find, or operate upon prefetch streams.
636  */
637 void
638 dmu_zfetch(zfetch_t *zf, uint64_t offset, uint64_t size, int prefetched)
639 {
640         zstream_t       zst;
641         zstream_t       *newstream;
642         boolean_t       fetched;
642         int             fetched;
643         int             inserted;
644         unsigned int    blkshft;
645         uint64_t        blksz;
647         if (zfs_prefetch_disable)
648                 return;
650         /* files that aren't ln2 blocksz are only one block -- nothing to do */
651         if (!zf->zf_dnode->dn_datablkshift)
652                 return;
654         /* convert offset and size, into blockid and nblocks */
655         blkshft = zf->zf_dnode->dn_datablkshift;
656         blksz = (1 << blkshft);
658         bzero(&zst, sizeof (zstream_t));
659         zst.zst_offset = offset >> blkshft;
660         zst.zst_len = (P2ROUNDUP(offset + size, blksz) -
661             P2ALIGN(offset, blksz)) >> blkshft;
663         fetched = dmu_zfetch_find(zf, &zst, prefetched);
664         if (fetched) {
```

```
665                 ZFETCHSTAT_BUMP(zfetchstat_hits);
666         } else {
667                 ZFETCHSTAT_BUMP(zfetchstat_misses);
668                 fetched = dmu_zfetch_colinear(zf, &zst);
669                 if (fetched) {
668                 if (fetched = dmu_zfetch_colinear(zf, &zst)) {
670                         ZFETCHSTAT_BUMP(zfetchstat_colinear_hits);
671                 } else {
672                         ZFETCHSTAT_BUMP(zfetchstat_colinear_misses);
673                 }
674         }

676         if (!fetched) {
677                 newstream = dmu_zfetch_stream_reclaim(zf);

679                 /*
680                  * we still couldn't find a stream, drop the lock, and allocate
681                  * one if possible.  Otherwise, give up and go home.
682                  */
683                 if (newstream) {
684                         ZFETCHSTAT_BUMP(zfetchstat_reclaim_successes);
685                 } else {
686                         uint64_t        maxblocks;
687                         uint32_t        max_streams;
688                         uint32_t        cur_streams;

690                         ZFETCHSTAT_BUMP(zfetchstat_reclaim_failures);
691                         cur_streams = zf->zf_stream_cnt;
692                         maxblocks = zf->zf_dnode->dn_maxblkid;

694                         max_streams = MIN(zfetch_max_streams,
695                             (maxblocks / zfetch_block_cap));
696                         if (max_streams == 0) {
697                                 max_streams++;
698                         }

700                         if (cur_streams >= max_streams) {
701                                 return;
702                         }
703                         newstream = kmem_zalloc(sizeof (zstream_t), KM_SLEEP);
704                 }

706                 newstream->zst_offset = zst.zst_offset;
707                 newstream->zst_len = zst.zst_len;
708                 newstream->zst_stride = zst.zst_len;
709                 newstream->zst_ph_offset = zst.zst_len + zst.zst_offset;
710                 newstream->zst_cap = zst.zst_len;
711                 newstream->zst_direction = ZFETCH_FORWARD;
712                 newstream->zst_last = ddi_get_lbolt();

714                 mutex_init(&newstream->zst_lock, NULL, MUTEX_DEFAULT, NULL);

716                 rw_enter(&zf->zf_rwlock, RW_WRITER);
717                 inserted = dmu_zfetch_stream_insert(zf, newstream);
718                 rw_exit(&zf->zf_rwlock);

720                 if (!inserted) {
721                         mutex_destroy(&newstream->zst_lock);
722                         kmem_free(newstream, sizeof (zstream_t));
723                 }
724         }
725 }
```

unchanged_portion_omitted_

```
**********************************************************
  174736 Tue Apr 23 14:09:36 2013
new/usr/src/uts/common/fs/zfs/spa.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
```

```
/*
 * 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) 2013 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 * SPA: Storage Pool Allocator
 *
#endif /* ! codereview */
 * This file contains all the routines used when modifying on-disk SPA state.
 * This includes opening, importing, destroying, exporting a pool, and syncing a
 * pool.
 */

#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/ddt.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/metaslab_impl.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_objset.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
```



```
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/fs/zfs.h>
#include <sys/arc.h>
#include <sys/callb.h>
#include <sys/systeminfo.h>
#include <sys/spa_boot.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_scan.h>
#include <sys/zfeature.h>
#include <sys/dsl_destroy.h>

#ifdef  _KERNEL
#include <sys/bootprops.h>
#include <sys/callb.h>
#include <sys/cpupart.h>
#include <sys/pool.h>
#include <sys/sysdc.h>
#include <sys/zone.h>
#endif  /* _KERNEL */

#include "zfs_prop.h"
#include "zfs_comutil.h"

typedef enum zti_modes {
        ZTI_MODE_FIXED,                 /* value is # of threads (min 1) */
        ZTI_MODE_ONLINE_PERCENT,        /* value is % of online CPUs */
        ZTI_MODE_BATCH,                 /* cpu-intensive; value is ignored */
        ZTI_MODE_NULL,                  /* don't create a taskq */
        ZTI_NMODES
} zti_modes_t;

#define ZTI_P(n, q)     { ZTI_MODE_FIXED, (n), (q) }
#define ZTI_PCT(n)      { ZTI_MODE_ONLINE_PERCENT, (n), 1 }
#define ZTI_BATCH       { ZTI_MODE_BATCH, 0, 1 }
#define ZTI_NULL        { ZTI_MODE_NULL, 0, 0 }

#define ZTI_N(n)        ZTI_P(n, 1)
#define ZTI_ONE         ZTI_N(1)

typedef struct zio_taskq_info {
        zti_modes_t zti_mode;
        uint_t zti_value;
        uint_t zti_count;
} zio_taskq_info_t;

static const char *const zio_taskq_types[ZIO_TASKQ_TYPES] = {
        "issue", "issue_high", "intr", "intr_high"
};

/*
 * This table defines the taskq settings for each ZFS I/O type. When
 * initializing a pool, we use this table to create an appropriately sized
 * taskq. Some operations are low volume and therefore have a small, static
 * number of threads assigned to their taskqs using the ZTI_N(#) or ZTI_ONE
 * macros. Other operations process a large amount of data; the ZTI_BATCH
 * macro causes us to create a taskq oriented for throughput. Some operations
 * are so high frequency and short-lived that the taskq itself can become a a
 * point of lock contention. The ZTI_P(#, #) macro indicates that we need an
 * additional degree of parallelism specified by the number of threads per-
 * taskq and the number of taskqs; when dispatching an event in this case, the
 * particular taskq is chosen at random.
 *
 * The different taskq priorities are to handle the different contexts (issue
 * and interrupt) and then to reserve threads for ZIO_PRIORITY_NOW I/Os that
```

```
 * need to be handled with minimum delay.
 */
const zio_taskq_info_t zio_taskqs[ZIO_TYPES][ZIO_TASKQ_TYPES] = {
        /* ISSUE        ISSUE_HIGH      INTR            INTR_HIGH */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* NULL */
        { ZTI_N(8),     ZTI_NULL,       ZTI_BATCH,      ZTI_NULL }, /* READ */
        { ZTI_BATCH,    ZTI_N(5),       ZTI_N(8),       ZTI_N(5) }, /* WRITE */
        { ZTI_P(12, 8), ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* FREE */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* CLAIM */
        { ZTI_ONE,      ZTI_NULL,       ZTI_ONE,        ZTI_NULL }, /* IOCTL */
};

static void spa_sync_version(void *arg, dmu_tx_t *tx);
static void spa_sync_props(void *arg, dmu_tx_t *tx);
static boolean_t spa_has_active_shared_spare(spa_t *spa);
static int spa_load_impl(spa_t *spa, uint64_t, nvlist_t *config,
    spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
    char **ereport);
static void spa_vdev_resilver_done(spa_t *spa);

uint_t          zio_taskq_batch_pct = 100;      /* 1 thread per cpu in pset */
id_t            zio_taskq_psrset_bind = PS_NONE;
boolean_t       zio_taskq_sysdc = B_TRUE;       /* use SDC scheduling class */
uint_t          zio_taskq_basedc = 80;          /* base duty cycle */

boolean_t       spa_create_process = B_TRUE;    /* no process ==> no sysdc */
extern int      zfs_sync_pass_deferred_free;

/*
 * This (illegal) pool name is used when temporarily importing a spa_t in order
 * to get the vdev stats associated with the imported devices.
 */
#define TRYIMPORT_NAME  "$import"

/*
 * ==========================================================================
 * SPA properties routines
 * ==========================================================================
 */

/*
 * Add a (source=src, propname=propval) list to an nvlist.
 */
static void
spa_prop_add_list(nvlist_t *nvl, zpool_prop_t prop, char *strval,
    uint64_t intval, zprop_source_t src)
{
        const char *propname = zpool_prop_to_name(prop);
        nvlist_t *propval;

        VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
        VERIFY(nvlist_add_uint64(propval, ZPROP_SOURCE, src) == 0);

        if (strval != NULL)
                VERIFY(nvlist_add_string(propval, ZPROP_VALUE, strval) == 0);
        else
                VERIFY(nvlist_add_uint64(propval, ZPROP_VALUE, intval) == 0);

        VERIFY(nvlist_add_nvlist(nvl, propname, propval) == 0);
        nvlist_free(propval);
}

/*
 * Get property values from the spa configuration.
 */
static void
```



```
spa_prop_get_config(spa_t *spa, nvlist_t **nvp)
{
        vdev_t *rvd = spa->spa_root_vdev;
        dsl_pool_t *pool = spa->spa_dsl_pool;
        uint64_t size;
        uint64_t alloc;
        uint64_t space;
        uint64_t cap, version;
        zprop_source_t src = ZPROP_SRC_NONE;
        spa_config_dirent_t *dp;

        ASSERT(MUTEX_HELD(&spa->spa_props_lock));

        if (rvd != NULL) {
                alloc = metaslab_class_get_alloc(spa_normal_class(spa));
                size = metaslab_class_get_space(spa_normal_class(spa));
                spa_prop_add_list(*nvp, ZPOOL_PROP_NAME, spa_name(spa), 0, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_SIZE, NULL, size, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_ALLOCATED, NULL, alloc, src);
                spa_prop_add_list(*nvp, ZPOOL_PROP_FREE, NULL,
                    size - alloc, src);

                space = 0;
                for (int c = 0; c < rvd->vdev_children; c++) {
                        vdev_t *tvd = rvd->vdev_child[c];
                        space += tvd->vdev_max_asize - tvd->vdev_asize;
                }
                spa_prop_add_list(*nvp, ZPOOL_PROP_EXPANDSZ, NULL, space,
                    src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_READONLY, NULL,
                    (spa_mode(spa) == FREAD), src);

                cap = (size == 0) ? 0 : (alloc * 100 / size);
                spa_prop_add_list(*nvp, ZPOOL_PROP_CAPACITY, NULL, cap, src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_DEDUPRATIO, NULL,
                    ddt_get_pool_dedup_ratio(spa), src);

                spa_prop_add_list(*nvp, ZPOOL_PROP_HEALTH, NULL,
                    rvd->vdev_state, src);

                version = spa_version(spa);
                if (version == zpool_prop_default_numeric(ZPOOL_PROP_VERSION))
                        src = ZPROP_SRC_DEFAULT;
                else
                        src = ZPROP_SRC_LOCAL;
                spa_prop_add_list(*nvp, ZPOOL_PROP_VERSION, NULL, version, src);
        }

        if (pool != NULL) {
                dsl_dir_t *freedir = pool->dp_free_dir;

                /*
                 * The $FREE directory was introduced in SPA_VERSION_DEADLISTS,
                 * when opening pools before this version freedir will be NULL.
                 */
                if (freedir != NULL) {
                        spa_prop_add_list(*nvp, ZPOOL_PROP_FREEING, NULL,
                            freedir->dd_phys->dd_used_bytes, src);
                } else {
                        spa_prop_add_list(*nvp, ZPOOL_PROP_FREEING,
                            NULL, 0, src);
                }
        }
```

```
256         spa_prop_add_list(*nvp, ZPOOL_PROP_GUID, NULL, spa_guid(spa), src);

258         if (spa->spa_comment != NULL) {
259                 spa_prop_add_list(*nvp, ZPOOL_PROP_COMMENT, spa->spa_comment,
260                     0, ZPROP_SRC_LOCAL);
261         }

263         if (spa->spa_root != NULL)
264                 spa_prop_add_list(*nvp, ZPOOL_PROP_ALTROOT, spa->spa_root,
265                     0, ZPROP_SRC_LOCAL);

267         if ((dp = list_head(&spa->spa_config_list)) != NULL) {
268                 if (dp->scd_path == NULL) {
269                         spa_prop_add_list(*nvp, ZPOOL_PROP_CACHEFILE,
270                             "none", 0, ZPROP_SRC_LOCAL);
271                 } else if (strcmp(dp->scd_path, spa_config_path) != 0) {
272                         spa_prop_add_list(*nvp, ZPOOL_PROP_CACHEFILE,
273                             dp->scd_path, 0, ZPROP_SRC_LOCAL);
274                 }
275         }
276 }

278 /*
279  * Get zpool property values.
280  */
281 int
282 spa_prop_get(spa_t *spa, nvlist_t **nvp)
283 {
284         objset_t *mos = spa->spa_meta_objset;
285         zap_cursor_t zc;
286         zap_attribute_t za;
287         int err;

289         VERIFY(nvlist_alloc(nvp, NV_UNIQUE_NAME, KM_SLEEP) == 0);

291         mutex_enter(&spa->spa_props_lock);

293         /*
294          * Get properties from the spa config.
295          */
296         spa_prop_get_config(spa, nvp);

298         /* If no pool property object, no more prop to get. */
299         if (mos == NULL || spa->spa_pool_props_object == 0) {
300                 mutex_exit(&spa->spa_props_lock);
301                 return (0);
302         }

304         /*
305          * Get properties from the MOS pool property object.
306          */
307         for (zap_cursor_init(&zc, mos, spa->spa_pool_props_object);
308             (err = zap_cursor_retrieve(&zc, &za)) == 0;
309             zap_cursor_advance(&zc)) {
310                 uint64_t intval = 0;
311                 char *strval = NULL;
312                 zprop_source_t src = ZPROP_SRC_DEFAULT;
313                 zpool_prop_t prop;

315                 if ((prop = zpool_name_to_prop(za.za_name)) == ZPROP_INVAL)
316                         continue;

318                 switch (za.za_integer_length) {
319                 case 8:
320                         /* integer property */
321                         if (za.za_first_integer !=
```



```
322                             zpool_prop_default_numeric(prop))
323                                 src = ZPROP_SRC_LOCAL;

325                         if (prop == ZPOOL_PROP_BOOTFS) {
326                                 dsl_pool_t *dp;
327                                 dsl_dataset_t *ds = NULL;

329                                 dp = spa_get_dsl(spa);
330                                 dsl_pool_config_enter(dp, FTAG);
331                                 if (err = dsl_dataset_hold_obj(dp,
332                                     za.za_first_integer, FTAG, &ds)) {
333                                         dsl_pool_config_exit(dp, FTAG);
334                                         break;
335                                 }

337                                 strval = kmem_alloc(
338                                     MAXNAMELEN + strlen(MOS_DIR_NAME) + 1,
339                                     KM_SLEEP);
340                                 dsl_dataset_name(ds, strval);
341                                 dsl_dataset_rele(ds, FTAG);
342                                 dsl_pool_config_exit(dp, FTAG);
343                         } else {
344                                 strval = NULL;
345                                 intval = za.za_first_integer;
346                         }

348                         spa_prop_add_list(*nvp, prop, strval, intval, src);

350                         if (strval != NULL)
351                                 kmem_free(strval,
352                                     MAXNAMELEN + strlen(MOS_DIR_NAME) + 1);

354                         break;

356                 case 1:
357                         /* string property */
358                         strval = kmem_alloc(za.za_num_integers, KM_SLEEP);
359                         err = zap_lookup(mos, spa->spa_pool_props_object,
360                             za.za_name, 1, za.za_num_integers, strval);
361                         if (err) {
362                                 kmem_free(strval, za.za_num_integers);
363                                 break;
364                         }
365                         spa_prop_add_list(*nvp, prop, strval, 0, src);
366                         kmem_free(strval, za.za_num_integers);
367                         break;

369                 default:
370                         break;
371                 }
372         }
373         zap_cursor_fini(&zc);
374         mutex_exit(&spa->spa_props_lock);
375 out:
376         if (err && err != ENOENT) {
377                 nvlist_free(*nvp);
378                 *nvp = NULL;
379                 return (err);
380         }

382         return (0);
383 }

385 /*
386  * Validate the given pool properties nvlist and modify the list
387  * for the property values to be set.
```

```
 */
static int
spa_prop_validate(spa_t *spa, nvlist_t *props)
{
        nvpair_t *elem;
        int error = 0, reset_bootfs = 0;
        uint64_t objnum = 0;
        boolean_t has_feature = B_FALSE;

        elem = NULL;
        while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
                uint64_t intval;
                char *strval, *slash, *check, *fname;
                const char *propname = nvpair_name(elem);
                zpool_prop_t prop = zpool_name_to_prop(propname);

                switch (prop) {
                case ZPROP_INVAL:
                        if (!zpool_prop_feature(propname)) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }

                        /*
                         * Sanitize the input.
                         */
                        if (nvpair_type(elem) != DATA_TYPE_UINT64) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }

                        if (nvpair_value_uint64(elem, &intval) != 0) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }

                        if (intval != 0) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }

                        fname = strchr(propname, '@') + 1;
                        if (zfeature_lookup_name(fname, NULL) != 0) {
                                error = SET_ERROR(EINVAL);
                                break;
                        }

                        has_feature = B_TRUE;
                        break;

                case ZPOOL_PROP_VERSION:
                        error = nvpair_value_uint64(elem, &intval);
                        if (!error &&
                            (intval < spa_version(spa) ||
                            intval > SPA_VERSION_BEFORE_FEATURES ||
                            has_feature))
                                error = SET_ERROR(EINVAL);
                        break;

                case ZPOOL_PROP_DELEGATION:
                case ZPOOL_PROP_AUTOREPLACE:
                case ZPOOL_PROP_LISTSNAPS:
                case ZPOOL_PROP_AUTOEXPAND:
                        error = nvpair_value_uint64(elem, &intval);
                        if (!error && intval > 1)
                                error = SET_ERROR(EINVAL);
```



```
                        break;

                case ZPOOL_PROP_BOOTFS:
                        /*
                         * If the pool version is less than SPA_VERSION_BOOTFS,
                         * or the pool is still being created (version == 0),
                         * the bootfs property cannot be set.
                         */
                        if (spa_version(spa) < SPA_VERSION_BOOTFS) {
                                error = SET_ERROR(ENOTSUP);
                                break;
                        }

                        /*
                         * Make sure the vdev config is bootable
                         */
                        if (!vdev_is_bootable(spa->spa_root_vdev)) {
                                error = SET_ERROR(ENOTSUP);
                                break;
                        }

                        reset_bootfs = 1;

                        error = nvpair_value_string(elem, &strval);

                        if (!error) {
                                objset_t *os;
                                uint64_t compress;

                                if (strval == NULL || strval[0] == '\0') {
                                        objnum = zpool_prop_default_numeric(
                                            ZPOOL_PROP_BOOTFS);
                                        break;
                                }

                                if (error = dmu_objset_hold(strval, FTAG, &os))
                                        break;

                                /* Must be ZPL and not gzip compressed. */

                                if (dmu_objset_type(os) != DMU_OST_ZFS) {
                                        error = SET_ERROR(ENOTSUP);
                                } else if ((error =
                                    dsl_prop_get_int_ds(dmu_objset_ds(os),
                                    zfs_prop_to_name(ZFS_PROP_COMPRESSION),
                                    &compress)) == 0 &&
                                    !BOOTFS_COMPRESS_VALID(compress)) {
                                        error = SET_ERROR(ENOTSUP);
                                } else {
                                        objnum = dmu_objset_id(os);
                                }
                                dmu_objset_rele(os, FTAG);
                        }
                        break;

                case ZPOOL_PROP_FAILUREMODE:
                        error = nvpair_value_uint64(elem, &intval);
                        if (!error && (intval < ZIO_FAILURE_MODE_WAIT ||
                            intval > ZIO_FAILURE_MODE_PANIC))
                                error = SET_ERROR(EINVAL);

                        /*
                         * This is a special case which only occurs when
                         * the pool has completely failed. This allows
                         * the user to change the in-core failmode property
                         * without syncing it out to disk (I/Os might
```

```
520                          * currently be blocked). We do this by returning
521                          * EIO to the caller (spa_prop_set) to trick it
522                          * into thinking we encountered a property validation
523                          * error.
524                          */
525                         if (!error && spa_suspended(spa)) {
526                                 spa->spa_failmode = intval;
527                                 error = SET_ERROR(EIO);
528                         }
529                         break;
530
531                 case ZPOOL_PROP_CACHEFILE:
532                         if ((error = nvpair_value_string(elem, &strval)) != 0)
533                                 break;
534
535                         if (strval[0] == '\0')
536                                 break;
537
538                         if (strcmp(strval, "none") == 0)
539                                 break;
540
541                         if (strval[0] != '/') {
542                                 error = SET_ERROR(EINVAL);
543                                 break;
544                         }
545
546                         slash = strrchr(strval, '/');
547                         ASSERT(slash != NULL);
548
549                         if (slash[1] == '\0' || strcmp(slash, "/.") == 0 ||
550                             strcmp(slash, "/..") == 0)
551                                 error = SET_ERROR(EINVAL);
552                         break;
553
554                 case ZPOOL_PROP_COMMENT:
555                         if ((error = nvpair_value_string(elem, &strval)) != 0)
556                                 break;
557                         for (check = strval; *check != '\0'; check++) {
558                                 /*
559                                  * The kernel doesn't have an easy isprint()
560                                  * check.  For this kernel check, we merely
561                                  * check ASCII apart from DEL.  Fix this if
562                                  * there is an easy-to-use kernel isprint().
563                                  */
564                                 if (*check >= 0x7f) {
565                                         error = SET_ERROR(EINVAL);
566                                         break;
567                                 }
568                                 check++;
569                         }
570                         if (strlen(strval) > ZPROP_MAX_COMMENT)
571                                 error = E2BIG;
572                         break;
573
574                 case ZPOOL_PROP_DEDUPDITTO:
575                         if (spa_version(spa) < SPA_VERSION_DEDUP)
576                                 error = SET_ERROR(ENOTSUP);
577                         else
578                                 error = nvpair_value_uint64(elem, &intval);
579                         if (error == 0 &&
580                             intval != 0 && intval < ZIO_DEDUPDITTO_MIN)
581                                 error = SET_ERROR(EINVAL);
582                         break;
583                 }
584
585                 if (error)
```



```
586                         break;
587         }
588
589         if (!error && reset_bootfs) {
590                 error = nvlist_remove(props,
591                     zpool_prop_to_name(ZPOOL_PROP_BOOTFS), DATA_TYPE_STRING);
592
593                 if (!error) {
594                         error = nvlist_add_uint64(props,
595                             zpool_prop_to_name(ZPOOL_PROP_BOOTFS), objnum);
596                 }
597         }
598
599         return (error);
600 }
601
602 void
603 spa_configfile_set(spa_t *spa, nvlist_t *nvp, boolean_t need_sync)
604 {
605         char *cachefile;
606         spa_config_dirent_t *dp;
607
608         if (nvlist_lookup_string(nvp, zpool_prop_to_name(ZPOOL_PROP_CACHEFILE),
609             &cachefile) != 0)
610                 return;
611
612         dp = kmem_alloc(sizeof (spa_config_dirent_t),
613             KM_SLEEP);
614
615         if (cachefile[0] == '\0')
616                 dp->scd_path = spa_strdup(spa_config_path);
617         else if (strcmp(cachefile, "none") == 0)
618                 dp->scd_path = NULL;
619         else
620                 dp->scd_path = spa_strdup(cachefile);
621
622         list_insert_head(&spa->spa_config_list, dp);
623         if (need_sync)
624                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
625 }
626
627 int
628 spa_prop_set(spa_t *spa, nvlist_t *nvp)
629 {
630         int error;
631         nvpair_t *elem = NULL;
632         boolean_t need_sync = B_FALSE;
633
634         if ((error = spa_prop_validate(spa, nvp)) != 0)
635                 return (error);
636
637         while ((elem = nvlist_next_nvpair(nvp, elem)) != NULL) {
638                 zpool_prop_t prop = zpool_name_to_prop(nvpair_name(elem));
639
640                 if (prop == ZPOOL_PROP_CACHEFILE ||
641                     prop == ZPOOL_PROP_ALTROOT ||
642                     prop == ZPOOL_PROP_READONLY)
643                         continue;
644
645                 if (prop == ZPOOL_PROP_VERSION || prop == ZPROP_INVAL) {
646                         uint64_t ver;
647
648                         if (prop == ZPOOL_PROP_VERSION) {
649                                 VERIFY(nvpair_value_uint64(elem, &ver) == 0);
650                         } else {
651                                 ASSERT(zpool_prop_feature(nvpair_name(elem)));
```

```
652                                 ver = SPA_VERSION_FEATURES;
653                                 need_sync = B_TRUE;
654                         }
655
656                         /* Save time if the version is already set. */
657                         if (ver == spa_version(spa))
658                                 continue;
659
660                         /*
661                          * In addition to the pool directory object, we might
662                          * create the pool properties object, the features for
663                          * read object, the features for write object, or the
664                          * feature descriptions object.
665                          */
666                         error = dsl_sync_task(spa->spa_name, NULL,
667                             spa_sync_version, &ver, 6);
668                         if (error)
669                                 return (error);
670                         continue;
671                 }
672
673                 need_sync = B_TRUE;
674                 break;
675         }
676
677         if (need_sync) {
678                 return (dsl_sync_task(spa->spa_name, NULL, spa_sync_props,
679                     nvp, 6));
680         }
681
682         return (0);
683 }
684
685 /*
686  * If the bootfs property value is dsobj, clear it.
687  */
688 void
689 spa_prop_clear_bootfs(spa_t *spa, uint64_t dsobj, dmu_tx_t *tx)
690 {
691         if (spa->spa_bootfs == dsobj && spa->spa_pool_props_object != 0) {
692                 VERIFY(zap_remove(spa->spa_meta_objset,
693                     spa->spa_pool_props_object,
694                     zpool_prop_to_name(ZPOOL_PROP_BOOTFS), tx) == 0);
695                 spa->spa_bootfs = 0;
696         }
697 }
698
699 /*ARGSUSED*/
700 static int
701 spa_change_guid_check(void *arg, dmu_tx_t *tx)
702 {
703         uint64_t *newguid = arg;
704         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
705         vdev_t *rvd = spa->spa_root_vdev;
706         uint64_t vdev_state;
707
708         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
709         vdev_state = rvd->vdev_state;
710         spa_config_exit(spa, SCL_STATE, FTAG);
711
712         if (vdev_state != VDEV_STATE_HEALTHY)
713                 return (SET_ERROR(ENXIO));
714
715         ASSERT3U(spa_guid(spa), !=, *newguid);
716
717         return (0);
```



```
718 }
719
720 static void
721 spa_change_guid_sync(void *arg, dmu_tx_t *tx)
722 {
723         uint64_t *newguid = arg;
724         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
725         uint64_t oldguid;
726         vdev_t *rvd = spa->spa_root_vdev;
727
728         oldguid = spa_guid(spa);
729
730         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
731         rvd->vdev_guid = *newguid;
732         rvd->vdev_guid_sum += (*newguid - oldguid);
733         vdev_config_dirty(rvd);
734         spa_config_exit(spa, SCL_STATE, FTAG);
735
736         spa_history_log_internal(spa, "guid change", tx, "old=%llu new=%llu",
737             oldguid, *newguid);
738 }
739
740 /*
741  * Change the GUID for the pool.  This is done so that we can later
742  * re-import a pool built from a clone of our own vdevs.  We will modify
743  * the root vdev's guid, our own pool guid, and then mark all of our
744  * vdevs dirty.  Note that we must make sure that all our vdevs are
745  * online when we do this, or else any vdevs that weren't present
746  * would be orphaned from our pool.  We are also going to issue a
747  * sysevent to update any watchers.
748  */
749 int
750 spa_change_guid(spa_t *spa)
751 {
752         int error;
753         uint64_t guid;
754
755         mutex_enter(&spa_namespace_lock);
756         guid = spa_generate_guid(NULL);
757
758         error = dsl_sync_task(spa->spa_name, spa_change_guid_check,
759             spa_change_guid_sync, &guid, 5);
760
761         if (error == 0) {
762                 spa_config_sync(spa, B_FALSE, B_TRUE);
763                 spa_event_notify(spa, NULL, ESC_ZFS_POOL_REGUID);
764         }
765
766         mutex_exit(&spa_namespace_lock);
767
768         return (error);
769 }
770
771 /*
772  * ========================================================================
773  * SPA state manipulation (open/create/destroy/import/export)
774  * ========================================================================
775  */
776
777 static int
778 spa_error_entry_compare(const void *a, const void *b)
779 {
780         spa_error_entry_t *sa = (spa_error_entry_t *)a;
781         spa_error_entry_t *sb = (spa_error_entry_t *)b;
782         int ret;
```

```
784         ret = bcmp(&sa->se_bookmark, &sb->se_bookmark,
785             sizeof (zbookmark_t));
786
787         if (ret < 0)
788                 return (-1);
789         else if (ret > 0)
790                 return (1);
791         else
792                 return (0);
793 }
794
795 /*
796  * Utility function which retrieves copies of the current logs and
797  * re-initializes them in the process.
798  */
799 void
800 spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub)
801 {
802         ASSERT(MUTEX_HELD(&spa->spa_errlist_lock));
803
804         bcopy(&spa->spa_errlist_last, last, sizeof (avl_tree_t));
805         bcopy(&spa->spa_errlist_scrub, scrub, sizeof (avl_tree_t));
806
807         avl_create(&spa->spa_errlist_scrub,
808             spa_error_entry_compare, sizeof (spa_error_entry_t),
809             offsetof(spa_error_entry_t, se_avl));
810         avl_create(&spa->spa_errlist_last,
811             spa_error_entry_compare, sizeof (spa_error_entry_t),
812             offsetof(spa_error_entry_t, se_avl));
813 }
814
815 static void
816 spa_taskqs_init(spa_t *spa, zio_type_t t, zio_taskq_type_t q)
817 {
818         const zio_taskq_info_t *ztip = &zio_taskqs[t][q];
819         enum zti_modes mode = ztip->zti_mode;
820         uint_t value = ztip->zti_value;
821         uint_t count = ztip->zti_count;
822         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
823         char name[32];
824         uint_t flags = 0;
825         boolean_t batch = B_FALSE;
826
827         if (mode == ZTI_MODE_NULL) {
828                 tqs->stqs_count = 0;
829                 tqs->stqs_taskq = NULL;
830                 return;
831         }
832
833         ASSERT3U(count, >, 0);
834
835         tqs->stqs_count = count;
836         tqs->stqs_taskq = kmem_alloc(count * sizeof (taskq_t *), KM_SLEEP);
837
838         for (uint_t i = 0; i < count; i++) {
839                 taskq_t *tq;
840
841                 switch (mode) {
842                 case ZTI_MODE_FIXED:
843                         ASSERT3U(value, >=, 1);
844                         value = MAX(value, 1);
845                         break;
846
847                 case ZTI_MODE_BATCH:
848                         batch = B_TRUE;
849                         flags |= TASKQ_THREADS_CPU_PCT;
```



```
850                         value = zio_taskq_batch_pct;
851                         break;
852
853                 case ZTI_MODE_ONLINE_PERCENT:
854                         flags |= TASKQ_THREADS_CPU_PCT;
855                         break;
856
857                 default:
858                         panic("unrecognized mode for %s_%s taskq (%u:%u) in "
859                             "spa_activate()",
860                             zio_type_name[t], zio_taskq_types[q], mode, value);
861                         break;
862                 }
863
864                 if (count > 1) {
865                         (void) snprintf(name, sizeof (name), "%s_%s_%u",
866                             zio_type_name[t], zio_taskq_types[q], i);
867                 } else {
868                         (void) snprintf(name, sizeof (name), "%s_%s",
869                             zio_type_name[t], zio_taskq_types[q]);
870                 }
871
872                 if (zio_taskq_sysdc && spa->spa_proc != &p0) {
873                         if (batch)
874                                 flags |= TASKQ_DC_BATCH;
875
876                         tq = taskq_create_sysdc(name, value, 50, INT_MAX,
877                             spa->spa_proc, zio_taskq_basedc, flags);
878                 } else {
879                         tq = taskq_create_proc(name, value, maxclsyspri, 50,
880                             INT_MAX, spa->spa_proc, flags);
881                 }
882
883                 tqs->stqs_taskq[i] = tq;
884         }
885 }
886
887 static void
888 spa_taskqs_fini(spa_t *spa, zio_type_t t, zio_taskq_type_t q)
889 {
890         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
891
892         if (tqs->stqs_taskq == NULL) {
893                 ASSERT0(tqs->stqs_count);
894                 return;
895         }
896
897         for (uint_t i = 0; i < tqs->stqs_count; i++) {
898                 ASSERT3P(tqs->stqs_taskq[i], !=, NULL);
899                 taskq_destroy(tqs->stqs_taskq[i]);
900         }
901
902         kmem_free(tqs->stqs_taskq, tqs->stqs_count * sizeof (taskq_t *));
903         tqs->stqs_taskq = NULL;
904 }
905
906 /*
907  * Dispatch a task to the appropriate taskq for the ZFS I/O type and priority.
908  * Note that a type may have multiple discrete taskqs to avoid lock contention
909  * on the taskq itself. In that case we choose which taskq at random by using
910  * the low bits of gethrtime().
911  */
912 void
913 spa_taskq_dispatch_ent(spa_t *spa, zio_type_t t, zio_taskq_type_t q,
914     task_func_t *func, void *arg, uint_t flags, taskq_ent_t *ent)
915 {
```

```
916         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
917         taskq_t *tq;

919         ASSERT3P(tqs->stqs_taskq, !=, NULL);
920         ASSERT3U(tqs->stqs_count, !=, 0);

922         if (tqs->stqs_count == 1) {
923                 tq = tqs->stqs_taskq[0];
924         } else {
925                 tq = tqs->stqs_taskq[gethrtime() % tqs->stqs_count];
926         }

928         taskq_dispatch_ent(tq, func, arg, flags, ent);
929 }

931 static void
932 spa_create_zio_taskqs(spa_t *spa)
933 {
934         for (int t = 0; t < ZIO_TYPES; t++) {
935                 for (int q = 0; q < ZIO_TASKQ_TYPES; q++) {
936                         spa_taskqs_init(spa, t, q);
937                 }
938         }
939 }

941 #ifdef _KERNEL
942 static void
943 spa_thread(void *arg)
944 {
945         callb_cpr_t cprinfo;

947         spa_t *spa = arg;
948         user_t *pu = PTOU(curproc);

950         CALLB_CPR_INIT(&cprinfo, &spa->spa_proc_lock, callb_generic_cpr,
951             spa->spa_name);

953         ASSERT(curproc != &p0);
954         (void) snprintf(pu->u_psargs, sizeof (pu->u_psargs),
955             "zpool-%s", spa->spa_name);
956         (void) strlcpy(pu->u_comm, pu->u_psargs, sizeof (pu->u_comm));

958         /* bind this thread to the requested psrset */
959         if (zio_taskq_psrset_bind != PS_NONE) {
960                 pool_lock();
961                 mutex_enter(&cpu_lock);
962                 mutex_enter(&pidlock);
963                 mutex_enter(&curproc->p_lock);

965                 if (cpupart_bind_thread(curthread, zio_taskq_psrset_bind,
966                     0, NULL, NULL) == 0)  {
967                         curthread->t_bind_pset = zio_taskq_psrset_bind;
968                 } else {
969                         cmn_err(CE_WARN,
970                             "Couldn't bind process for zfs pool \"%s\" to "
971                             "pset %d\n", spa->spa_name, zio_taskq_psrset_bind);
972                 }

974                 mutex_exit(&curproc->p_lock);
975                 mutex_exit(&pidlock);
976                 mutex_exit(&cpu_lock);
977                 pool_unlock();
978         }

980         if (zio_taskq_sysdc) {
981                 sysdc_thread_enter(curthread, 100, 0);
```



```
982         }

984         spa->spa_proc = curproc;
985         spa->spa_did = curthread->t_did;

987         spa_create_zio_taskqs(spa);

989         mutex_enter(&spa->spa_proc_lock);
990         ASSERT(spa->spa_proc_state == SPA_PROC_CREATED);

992         spa->spa_proc_state = SPA_PROC_ACTIVE;
993         cv_broadcast(&spa->spa_proc_cv);

995         CALLB_CPR_SAFE_BEGIN(&cprinfo);
996         while (spa->spa_proc_state == SPA_PROC_ACTIVE)
997                 cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
998         CALLB_CPR_SAFE_END(&cprinfo, &spa->spa_proc_lock);

1000        ASSERT(spa->spa_proc_state == SPA_PROC_DEACTIVATE);
1001        spa->spa_proc_state = SPA_PROC_GONE;
1002        spa->spa_proc = &p0;
1003        cv_broadcast(&spa->spa_proc_cv);
1004        CALLB_CPR_EXIT(&cprinfo);       /* drops spa_proc_lock */

1006        mutex_enter(&curproc->p_lock);
1007        lwp_exit();
1008 }
1009 #endif

1011 /*
1012  * Activate an uninitialized pool.
1013  */
1014 static void
1015 spa_activate(spa_t *spa, int mode)
1016 {
1017         ASSERT(spa->spa_state == POOL_STATE_UNINITIALIZED);

1019         spa->spa_state = POOL_STATE_ACTIVE;
1020         spa->spa_mode = mode;

1022         spa->spa_normal_class = metaslab_class_create(spa, zfs_metaslab_ops);
1023         spa->spa_log_class = metaslab_class_create(spa, zfs_metaslab_ops);

1025         /* Try to create a covering process */
1026         mutex_enter(&spa->spa_proc_lock);
1027         ASSERT(spa->spa_proc_state == SPA_PROC_NONE);
1028         ASSERT(spa->spa_proc == &p0);
1029         spa->spa_did = 0;

1031         /* Only create a process if we're going to be around a while. */
1032         if (spa_create_process && strcmp(spa->spa_name, TRYIMPORT_NAME) != 0) {
1033                 if (newproc(spa_thread, (caddr_t)spa, syscid, maxclsyspri,
1034                     NULL, 0) == 0) {
1035                         spa->spa_proc_state = SPA_PROC_CREATED;
1036                         while (spa->spa_proc_state == SPA_PROC_CREATED) {
1037                                 cv_wait(&spa->spa_proc_cv,
1038                                     &spa->spa_proc_lock);
1039                         }
1040                         ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
1041                         ASSERT(spa->spa_proc != &p0);
1042                         ASSERT(spa->spa_did != 0);
1043                 } else {
1044 #ifdef _KERNEL
1045                         cmn_err(CE_WARN,
1046                             "Couldn't create process for zfs pool \"%s\"\n",
1047                             spa->spa_name);
```

```
#endif
                }
        }
        mutex_exit(&spa->spa_proc_lock);

        /* If we didn't create a process, we need to create our taskqs. */
        if (spa->spa_proc == &p0) {
                spa_create_zio_taskqs(spa);
        }

        list_create(&spa->spa_config_dirty_list, sizeof (vdev_t),
            offsetof(vdev_t, vdev_config_dirty_node));
        list_create(&spa->spa_state_dirty_list, sizeof (vdev_t),
            offsetof(vdev_t, vdev_state_dirty_node));

        txg_list_create(&spa->spa_vdev_txg_list,
            offsetof(struct vdev, vdev_txg_node));

        avl_create(&spa->spa_errlist_scrub,
            spa_error_entry_compare, sizeof (spa_error_entry_t),
            offsetof(spa_error_entry_t, se_avl));
        avl_create(&spa->spa_errlist_last,
            spa_error_entry_compare, sizeof (spa_error_entry_t),
            offsetof(spa_error_entry_t, se_avl));
}

/*
 * Opposite of spa_activate().
 */
static void
spa_deactivate(spa_t *spa)
{
        ASSERT(spa->spa_sync_on == B_FALSE);
        ASSERT(spa->spa_dsl_pool == NULL);
        ASSERT(spa->spa_root_vdev == NULL);
        ASSERT(spa->spa_async_zio_root == NULL);
        ASSERT(spa->spa_state != POOL_STATE_UNINITIALIZED);

        txg_list_destroy(&spa->spa_vdev_txg_list);

        list_destroy(&spa->spa_config_dirty_list);
        list_destroy(&spa->spa_state_dirty_list);

        for (int t = 0; t < ZIO_TYPES; t++) {
                for (int q = 0; q < ZIO_TASKQ_TYPES; q++) {
                        spa_taskqs_fini(spa, t, q);
                }
        }

        metaslab_class_destroy(spa->spa_normal_class);
        spa->spa_normal_class = NULL;

        metaslab_class_destroy(spa->spa_log_class);
        spa->spa_log_class = NULL;

        /*
         * If this was part of an import or the open otherwise failed, we may
         * still have errors left in the queues.  Empty them just in case.
         */
        spa_errlog_drain(spa);

        avl_destroy(&spa->spa_errlist_scrub);
        avl_destroy(&spa->spa_errlist_last);

        spa->spa_state = POOL_STATE_UNINITIALIZED;
```



```
        mutex_enter(&spa->spa_proc_lock);
        if (spa->spa_proc_state != SPA_PROC_NONE) {
                ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
                spa->spa_proc_state = SPA_PROC_DEACTIVATE;
                cv_broadcast(&spa->spa_proc_cv);
                while (spa->spa_proc_state == SPA_PROC_DEACTIVATE) {
                        ASSERT(spa->spa_proc != &p0);
                        cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
                }
                ASSERT(spa->spa_proc_state == SPA_PROC_GONE);
                spa->spa_proc_state = SPA_PROC_NONE;
        }
        ASSERT(spa->spa_proc == &p0);
        mutex_exit(&spa->spa_proc_lock);

        /*
         * We want to make sure spa_thread() has actually exited the ZFS
         * module, so that the module can't be unloaded out from underneath
         * it.
         */
        if (spa->spa_did != 0) {
                thread_join(spa->spa_did);
                spa->spa_did = 0;
        }
}

/*
 * Verify a pool configuration, and construct the vdev tree appropriately.  This
 * will create all the necessary vdevs in the appropriate layout, with each vdev
 * in the CLOSED state.  This will prep the pool before open/creation/import.
 * All vdev validation is done by the vdev_alloc() routine.
 */
static int
spa_config_parse(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent,
    uint_t id, int atype)
{
        nvlist_t **child;
        uint_t children;
        int error;

        if ((error = vdev_alloc(spa, vdp, nv, parent, id, atype)) != 0)
                return (error);

        if ((*vdp)->vdev_ops->vdev_op_leaf)
                return (0);

        error = nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
            &child, &children);

        if (error == ENOENT)
                return (0);

        if (error) {
                vdev_free(*vdp);
                *vdp = NULL;
                return (SET_ERROR(EINVAL));
        }

        for (int c = 0; c < children; c++) {
                vdev_t *vd;
                if ((error = spa_config_parse(spa, &vd, child[c], *vdp, c,
                    atype)) != 0) {
                        vdev_free(*vdp);
                        *vdp = NULL;
                        return (error);
                }
```

```
1180         }

1182         ASSERT(*vdp != NULL);

1184         return (0);
1185 }

1187 /*
1188  * Opposite of spa_load().
1189  */
1190 static void
1191 spa_unload(spa_t *spa)
1192 {
1193         int i;

1195         ASSERT(MUTEX_HELD(&spa_namespace_lock));

1197         /*
1198          * Stop async tasks.
1199          */
1200         spa_async_suspend(spa);

1202         /*
1203          * Stop syncing.
1204          */
1205         if (spa->spa_sync_on) {
1206                 txg_sync_stop(spa->spa_dsl_pool);
1207                 spa->spa_sync_on = B_FALSE;
1208         }

1210         /*
1211          * Wait for any outstanding async I/O to complete.
1212          */
1213         if (spa->spa_async_zio_root != NULL) {
1214                 (void) zio_wait(spa->spa_async_zio_root);
1215                 spa->spa_async_zio_root = NULL;
1216         }

1218         bpobj_close(&spa->spa_deferred_bpobj);

1220         /*
1221          * Close the dsl pool.
1222          */
1223         if (spa->spa_dsl_pool) {
1224                 dsl_pool_close(spa->spa_dsl_pool);
1225                 spa->spa_dsl_pool = NULL;
1226                 spa->spa_meta_objset = NULL;
1227         }

1229         ddt_unload(spa);

1231         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

1233         /*
1234          * Drop and purge level 2 cache
1235          */
1236         spa_l2cache_drop(spa);

1238         /*
1239          * Close all vdevs.
1240          */
1241         if (spa->spa_root_vdev)
1242                 vdev_free(spa->spa_root_vdev);
1243         ASSERT(spa->spa_root_vdev == NULL);

1245         for (i = 0; i < spa->spa_spares.sav_count; i++)
```



```
1246                 vdev_free(spa->spa_spares.sav_vdevs[i]);
1247         if (spa->spa_spares.sav_vdevs) {
1248                 kmem_free(spa->spa_spares.sav_vdevs,
1249                     spa->spa_spares.sav_count * sizeof (void *));
1250                 spa->spa_spares.sav_vdevs = NULL;
1251         }
1252         if (spa->spa_spares.sav_config) {
1253                 nvlist_free(spa->spa_spares.sav_config);
1254                 spa->spa_spares.sav_config = NULL;
1255         }
1256         spa->spa_spares.sav_count = 0;

1258         for (i = 0; i < spa->spa_l2cache.sav_count; i++) {
1259                 vdev_clear_stats(spa->spa_l2cache.sav_vdevs[i]);
1260                 vdev_free(spa->spa_l2cache.sav_vdevs[i]);
1261         }
1262         if (spa->spa_l2cache.sav_vdevs) {
1263                 kmem_free(spa->spa_l2cache.sav_vdevs,
1264                     spa->spa_l2cache.sav_count * sizeof (void *));
1265                 spa->spa_l2cache.sav_vdevs = NULL;
1266         }
1267         if (spa->spa_l2cache.sav_config) {
1268                 nvlist_free(spa->spa_l2cache.sav_config);
1269                 spa->spa_l2cache.sav_config = NULL;
1270         }
1271         spa->spa_l2cache.sav_count = 0;

1273         spa->spa_async_suspended = 0;

1275         if (spa->spa_comment != NULL) {
1276                 spa_strfree(spa->spa_comment);
1277                 spa->spa_comment = NULL;
1278         }

1280         spa_config_exit(spa, SCL_ALL, FTAG);
1281 }

1283 /*
1284  * Load (or re-load) the current list of vdevs describing the active spares for
1285  * this pool.  When this is called, we have some form of basic information in
1286  * 'spa_spares.sav_config'.  We parse this into vdevs, try to open them, and
1287  * then re-generate a more complete list including status information.
1288  */
1289 static void
1290 spa_load_spares(spa_t *spa)
1291 {
1292         nvlist_t **spares;
1293         uint_t nspares;
1294         int i;
1295         vdev_t *vd, *tvd;

1297         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

1299         /*
1300          * First, close and free any existing spare vdevs.
1301          */
1302         for (i = 0; i < spa->spa_spares.sav_count; i++) {
1303                 vd = spa->spa_spares.sav_vdevs[i];

1305                 /* Undo the call to spa_activate() below */
1306                 if ((tvd = spa_lookup_by_guid(spa, vd->vdev_guid,
1307                     B_FALSE)) != NULL && tvd->vdev_isspare)
1308                         spa_spare_remove(tvd);
1309                 vdev_close(vd);
1310                 vdev_free(vd);
1311         }
```

```
1313         if (spa->spa_spares.sav_vdevs)
1314                 kmem_free(spa->spa_spares.sav_vdevs,
1315                     spa->spa_spares.sav_count * sizeof (void *));
1316
1317         if (spa->spa_spares.sav_config == NULL)
1318                 nspares = 0;
1319         else
1320                 VERIFY(nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
1321                     ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
1322
1323         spa->spa_spares.sav_count = (int)nspares;
1324         spa->spa_spares.sav_vdevs = NULL;
1325
1326         if (nspares == 0)
1327                 return;
1328
1329         /*
1330          * Construct the array of vdevs, opening them to get status in the
1331          * process.   For each spare, there is potentially two different vdev_t
1332          * structures associated with it: one in the list of spares (used only
1333          * for basic validation purposes) and one in the active vdev
1334          * configuration (if it's spared in).  During this phase we open and
1335          * validate each vdev on the spare list.  If the vdev also exists in the
1336          * active configuration, then we also mark this vdev as an active spare.
1337          */
1338         spa->spa_spares.sav_vdevs = kmem_alloc(nspares * sizeof (void *),
1339             KM_SLEEP);
1340         for (i = 0; i < spa->spa_spares.sav_count; i++) {
1341                 VERIFY(spa_config_parse(spa, &vd, spares[i], NULL, 0,
1342                     VDEV_ALLOC_SPARE) == 0);
1343                 ASSERT(vd != NULL);
1344
1345                 spa->spa_spares.sav_vdevs[i] = vd;
1346
1347                 if ((tvd = spa_lookup_by_guid(spa, vd->vdev_guid,
1348                     B_FALSE)) != NULL) {
1349                         if (!tvd->vdev_isspare)
1350                                 spa_spare_add(tvd);
1351
1352                         /*
1353                          * We only mark the spare active if we were successfully
1354                          * able to load the vdev.  Otherwise, importing a pool
1355                          * with a bad active spare would result in strange
1356                          * behavior, because multiple pool would think the spare
1357                          * is actively in use.
1358                          *
1359                          * There is a vulnerability here to an equally bizarre
1360                          * circumstance, where a dead active spare is later
1361                          * brought back to life (onlined or otherwise).  Given
1362                          * the rarity of this scenario, and the extra complexity
1363                          * it adds, we ignore the possibility.
1364                          */
1365                         if (!vdev_is_dead(tvd))
1366                                 spa_spare_activate(tvd);
1367                 }
1368
1369                 vd->vdev_top = vd;
1370                 vd->vdev_aux = &spa->spa_spares;
1371
1372                 if (vdev_open(vd) != 0)
1373                         continue;
1374
1375                 if (vdev_validate_aux(vd) == 0)
1376                         spa_spare_add(vd);
1377         }
```



```
1379         /*
1380          * Recompute the stashed list of spares, with status information
1381          * this time.
1382          */
1383         VERIFY(nvlist_remove(spa->spa_spares.sav_config, ZPOOL_CONFIG_SPARES,
1384             DATA_TYPE_NVLIST_ARRAY) == 0);
1385
1386         spares = kmem_alloc(spa->spa_spares.sav_count * sizeof (void *),
1387             KM_SLEEP);
1388         for (i = 0; i < spa->spa_spares.sav_count; i++)
1389                 spares[i] = vdev_config_generate(spa,
1390                     spa->spa_spares.sav_vdevs[i], B_TRUE, VDEV_CONFIG_SPARE);
1391         VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
1392             ZPOOL_CONFIG_SPARES, spares, spa->spa_spares.sav_count) == 0);
1393         for (i = 0; i < spa->spa_spares.sav_count; i++)
1394                 nvlist_free(spares[i]);
1395         kmem_free(spares, spa->spa_spares.sav_count * sizeof (void *));
1396 }
1397
1398 /*
1399  * Load (or re-load) the current list of vdevs describing the active l2cache for
1400  * this pool.  When this is called, we have some form of basic information in
1401  * 'spa_l2cache.sav_config'.  We parse this into vdevs, try to open them, and
1402  * then re-generate a more complete list including status information.
1403  * Devices which are already active have their details maintained, and are
1404  * not re-opened.
1405  */
1406 static void
1407 spa_load_l2cache(spa_t *spa)
1408 {
1409         nvlist_t **l2cache;
1410         uint_t nl2cache;
1411         int i, j, oldnvdevs;
1412         uint64_t guid;
1413         vdev_t *vd, **oldvdevs, **newvdevs;
1414         spa_aux_vdev_t *sav = &spa->spa_l2cache;
1415
1416         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
1417
1418         if (sav->sav_config != NULL) {
1419                 VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
1420                     ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
1421                 newvdevs = kmem_alloc(nl2cache * sizeof (void *), KM_SLEEP);
1422         } else {
1423                 nl2cache = 0;
1424                 newvdevs = NULL;
1425         }
1426
1427         oldvdevs = sav->sav_vdevs;
1428         oldnvdevs = sav->sav_count;
1429         sav->sav_vdevs = NULL;
1430         sav->sav_count = 0;
1431
1432         /*
1433          * Process new nvlist of vdevs.
1434          */
1435         for (i = 0; i < nl2cache; i++) {
1436                 VERIFY(nvlist_lookup_uint64(l2cache[i], ZPOOL_CONFIG_GUID,
1437                     &guid) == 0);
1438
1439                 newvdevs[i] = NULL;
1440                 for (j = 0; j < oldnvdevs; j++) {
1441                         vd = oldvdevs[j];
1442                         if (vd != NULL && guid == vd->vdev_guid) {
1443                                 /*
```

```
1444                                 * Retain previous vdev for add/remove ops.
1445                                 */
1446                                newvdevs[i] = vd;
1447                                oldvdevs[j] = NULL;
1448                                break;
1449                        }
1450                }

1452                if (newvdevs[i] == NULL) {
1453                        /*
1454                         * Create new vdev
1455                         */
1456                        VERIFY(spa_config_parse(spa, &vd, l2cache[i], NULL, 0,
1457                            VDEV_ALLOC_L2CACHE) == 0);
1458                        ASSERT(vd != NULL);
1459                        newvdevs[i] = vd;

1461                        /*
1462                         * Commit this vdev as an l2cache device,
1463                         * even if it fails to open.
1464                         */
1465                        spa_l2cache_add(vd);

1467                        vd->vdev_top = vd;
1468                        vd->vdev_aux = sav;

1470                        spa_l2cache_activate(vd);

1472                        if (vdev_open(vd) != 0)
1473                                continue;

1475                        (void) vdev_validate_aux(vd);

1477                        if (!vdev_is_dead(vd))
1478                                l2arc_add_vdev(spa, vd);
1479                }
1480        }

1482        /*
1483         * Purge vdevs that were dropped
1484         */
1485        for (i = 0; i < oldnvdevs; i++) {
1486                uint64_t pool;

1488                vd = oldvdevs[i];
1489                if (vd != NULL) {
1490                        ASSERT(vd->vdev_isl2cache);

1492                        if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
1493                            pool != 0ULL && l2arc_vdev_present(vd))
1494                                l2arc_remove_vdev(vd);
1495                        vdev_clear_stats(vd);
1496                        vdev_free(vd);
1497                }
1498        }

1500        if (oldvdevs)
1501                kmem_free(oldvdevs, oldnvdevs * sizeof (void *));

1503        if (sav->sav_config == NULL)
1504                goto out;

1506        sav->sav_vdevs = newvdevs;
1507        sav->sav_count = (int)nl2cache;

1509        /*
```



```
1510         * Recompute the stashed list of l2cache devices, with status
1511         * information this time.
1512         */
1513        VERIFY(nvlist_remove(sav->sav_config, ZPOOL_CONFIG_L2CACHE,
1514            DATA_TYPE_NVLIST_ARRAY) == 0);

1516        l2cache = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
1517        for (i = 0; i < sav->sav_count; i++)
1518                l2cache[i] = vdev_config_generate(spa,
1519                    sav->sav_vdevs[i], B_TRUE, VDEV_CONFIG_L2CACHE);
1520        VERIFY(nvlist_add_nvlist_array(sav->sav_config,
1521            ZPOOL_CONFIG_L2CACHE, l2cache, sav->sav_count) == 0);
1522 out:
1523        for (i = 0; i < sav->sav_count; i++)
1524                nvlist_free(l2cache[i]);
1525        if (sav->sav_count)
1526                kmem_free(l2cache, sav->sav_count * sizeof (void *));
1527 }

1529 static int
1530 load_nvlist(spa_t *spa, uint64_t obj, nvlist_t **value)
1531 {
1532        dmu_buf_t *db;
1533        char *packed = NULL;
1534        size_t nvsize = 0;
1535        int error;
1536        *value = NULL;

1538        VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset, obj, FTAG, &db));
1539        nvsize = *(uint64_t *)db->db_data;
1540        dmu_buf_rele(db, FTAG);

1542        packed = kmem_alloc(nvsize, KM_SLEEP);
1543        error = dmu_read(spa->spa_meta_objset, obj, 0, nvsize, packed,
1544            DMU_READ_PREFETCH);
1545        if (error == 0)
1546                error = nvlist_unpack(packed, nvsize, value, 0);
1547        kmem_free(packed, nvsize);

1549        return (error);
1550 }

1552 /*
1553  * Checks to see if the given vdev could not be opened, in which case we post a
1554  * sysevent to notify the autoreplace code that the device has been removed.
1555  */
1556 static void
1557 spa_check_removed(vdev_t *vd)
1558 {
1559        for (int c = 0; c < vd->vdev_children; c++)
1560                spa_check_removed(vd->vdev_child[c]);

1562        if (vd->vdev_ops->vdev_op_leaf && vdev_is_dead(vd) &&
1563            !vd->vdev_ishole) {
1564                zfs_post_autoreplace(vd->vdev_spa, vd);
1565                spa_event_notify(vd->vdev_spa, vd, ESC_ZFS_VDEV_CHECK);
1566        }
1567 }

1569 /*
1570  * Validate the current config against the MOS config
1571  */
1572 static boolean_t
1573 spa_config_valid(spa_t *spa, nvlist_t *config)
1574 {
1575        vdev_t *mrvd, *rvd = spa->spa_root_vdev;
```

```
1576         nvlist_t *nv;
1578         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nv) == 0);
1580         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
1581         VERIFY(spa_config_parse(spa, &mrvd, nv, NULL, 0, VDEV_ALLOC_LOAD) == 0);
1583         ASSERT3U(rvd->vdev_children, ==, mrvd->vdev_children);
1585         /*
1586          * If we're doing a normal import, then build up any additional
1587          * diagnostic information about missing devices in this config.
1588          * We'll pass this up to the user for further processing.
1589          */
1590         if (!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG)) {
1591                 nvlist_t **child, *nv;
1592                 uint64_t idx = 0;
1594                 child = kmem_alloc(rvd->vdev_children * sizeof (nvlist_t **),
1595                     KM_SLEEP);
1596                 VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
1598                 for (int c = 0; c < rvd->vdev_children; c++) {
1599                         vdev_t *tvd = rvd->vdev_child[c];
1600                         vdev_t *mtvd  = mrvd->vdev_child[c];
1602                         if (tvd->vdev_ops == &vdev_missing_ops &&
1603                             mtvd->vdev_ops != &vdev_missing_ops &&
1604                             mtvd->vdev_islog)
1605                                 child[idx++] = vdev_config_generate(spa, mtvd,
1606                                     B_FALSE, 0);
1607                 }
1609                 if (idx) {
1610                         VERIFY(nvlist_add_nvlist_array(nv,
1611                             ZPOOL_CONFIG_CHILDREN, child, idx) == 0);
1612                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
1613                             ZPOOL_CONFIG_MISSING_DEVICES, nv) == 0);
1615                         for (int i = 0; i < idx; i++)
1616                                 nvlist_free(child[i]);
1617                 }
1618                 nvlist_free(nv);
1619                 kmem_free(child, rvd->vdev_children * sizeof (char **));
1620         }
1622         /*
1623          * Compare the root vdev tree with the information we have
1624          * from the MOS config (mrvd). Check each top-level vdev
1625          * with the corresponding MOS config top-level (mtvd).
1626          */
1627         for (int c = 0; c < rvd->vdev_children; c++) {
1628                 vdev_t *tvd = rvd->vdev_child[c];
1629                 vdev_t *mtvd  = mrvd->vdev_child[c];
1631                 /*
1632                  * Resolve any "missing" vdevs in the current configuration.
1633                  * If we find that the MOS config has more accurate information
1634                  * about the top-level vdev then use that vdev instead.
1635                  */
1636                 if (tvd->vdev_ops == &vdev_missing_ops &&
1637                     mtvd->vdev_ops != &vdev_missing_ops) {
1639                         if (!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG))
1640                                 continue;
```



```
1642                         /*
1643                          * Device specific actions.
1644                          */
1645                         if (mtvd->vdev_islog) {
1646                                 spa_set_log_state(spa, SPA_LOG_CLEAR);
1647                         } else {
1648                                 /*
1649                                  * XXX - once we have 'readonly' pool
1650                                  * support we should be able to handle
1651                                  * missing data devices by transitioning
1652                                  * the pool to readonly.
1653                                  */
1654                                 continue;
1655                         }
1657                         /*
1658                          * Swap the missing vdev with the data we were
1659                          * able to obtain from the MOS config.
1660                          */
1661                         vdev_remove_child(rvd, tvd);
1662                         vdev_remove_child(mrvd, mtvd);
1664                         vdev_add_child(rvd, mtvd);
1665                         vdev_add_child(mrvd, tvd);
1667                         spa_config_exit(spa, SCL_ALL, FTAG);
1668                         vdev_load(mtvd);
1669                         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
1671                         vdev_reopen(rvd);
1672                 } else if (mtvd->vdev_islog) {
1673                         /*
1674                          * Load the slog device's state from the MOS config
1675                          * since it's possible that the label does not
1676                          * contain the most up-to-date information.
1677                          */
1678                         vdev_load_log_state(tvd, mtvd);
1679                         vdev_reopen(tvd);
1680                 }
1681         }
1682         vdev_free(mrvd);
1683         spa_config_exit(spa, SCL_ALL, FTAG);
1685         /*
1686          * Ensure we were able to validate the config.
1687          */
1688         return (rvd->vdev_guid_sum == spa->spa_uberblock.ub_guid_sum);
1689 }
1691 /*
1692  * Check for missing log devices
1693  */
1694 static boolean_t
1695 spa_check_logs(spa_t *spa)
1696 {
1697         boolean_t rv = B_FALSE;
1699         switch (spa->spa_log_state) {
1700         case SPA_LOG_MISSING:
1701                 /* need to recheck in case slog has been restored */
1702         case SPA_LOG_UNKNOWN:
1703                 rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
1704                     NULL, DS_FIND_CHILDREN) != 0);
1705                 if (rv)
1706                         spa_set_log_state(spa, SPA_LOG_MISSING);
1707                 break;
```

```
1708         }
1709         return (rv);
1710 }

1712 static boolean_t
1713 spa_passivate_log(spa_t *spa)
1714 {
1715         vdev_t *rvd = spa->spa_root_vdev;
1716         boolean_t slog_found = B_FALSE;

1718         ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));

1720         if (!spa_has_slogs(spa))
1721                 return (B_FALSE);

1723         for (int c = 0; c < rvd->vdev_children; c++) {
1724                 vdev_t *tvd = rvd->vdev_child[c];
1725                 metaslab_group_t *mg = tvd->vdev_mg;

1727                 if (tvd->vdev_islog) {
1728                         metaslab_group_passivate(mg);
1729                         slog_found = B_TRUE;
1730                 }
1731         }

1733         return (slog_found);
1734 }

1736 static void
1737 spa_activate_log(spa_t *spa)
1738 {
1739         vdev_t *rvd = spa->spa_root_vdev;

1741         ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));

1743         for (int c = 0; c < rvd->vdev_children; c++) {
1744                 vdev_t *tvd = rvd->vdev_child[c];
1745                 metaslab_group_t *mg = tvd->vdev_mg;

1747                 if (tvd->vdev_islog)
1748                         metaslab_group_activate(mg);
1749         }
1750 }

1752 int
1753 spa_offline_log(spa_t *spa)
1754 {
1755         int error;

1757         error = dmu_objset_find(spa_name(spa), zil_vdev_offline,
1758             NULL, DS_FIND_CHILDREN);
1759         if (error == 0) {
1760                 /*
1761                  * We successfully offlined the log device, sync out the
1762                  * current txg so that the "stubby" block can be removed
1763                  * by zil_sync().
1764                  */
1765                 txg_wait_synced(spa->spa_dsl_pool, 0);
1766         }
1767         return (error);
1768 }

1770 static void
1771 spa_aux_check_removed(spa_aux_vdev_t *sav)
1772 {
1773         for (int i = 0; i < sav->sav_count; i++)
```



```
1774                 spa_check_removed(sav->sav_vdevs[i]);
1775 }

1777 void
1778 spa_claim_notify(zio_t *zio)
1779 {
1780         spa_t *spa = zio->io_spa;

1782         if (zio->io_error)
1783                 return;

1785         mutex_enter(&spa->spa_props_lock);      /* any mutex will do */
1786         if (spa->spa_claim_max_txg < zio->io_bp->blk_birth)
1787                 spa->spa_claim_max_txg = zio->io_bp->blk_birth;
1788         mutex_exit(&spa->spa_props_lock);
1789 }

1791 typedef struct spa_load_error {
1792         uint64_t        sle_meta_count;
1793         uint64_t        sle_data_count;
1794 } spa_load_error_t;

1796 static void
1797 spa_load_verify_done(zio_t *zio)
1798 {
1799         blkptr_t *bp = zio->io_bp;
1800         spa_load_error_t *sle = zio->io_private;
1801         dmu_object_type_t type = BP_GET_TYPE(bp);
1802         int error = zio->io_error;

1804         if (error) {
1805                 if ((BP_GET_LEVEL(bp) != 0 || DMU_OT_IS_METADATA(type)) &&
1806                     type != DMU_OT_INTENT_LOG)
1807                         atomic_add_64(&sle->sle_meta_count, 1);
1808                 else
1809                         atomic_add_64(&sle->sle_data_count, 1);
1810         }
1811         zio_data_buf_free(zio->io_data, zio->io_size);
1812 }

1814 /*ARGSUSED*/
1815 static int
1816 spa_load_verify_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp,
1817     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
1818 {
1819         if (bp != NULL) {
1820                 zio_t *rio = arg;
1821                 size_t size = BP_GET_PSIZE(bp);
1822                 void *data = zio_data_buf_alloc(size);

1824                 zio_nowait(zio_read(rio, spa, bp, data, size,
1825                     spa_load_verify_done, rio->io_private, ZIO_PRIORITY_SCRUB,
1826                     ZIO_FLAG_SPECULATIVE | ZIO_FLAG_CANFAIL |
1827                     ZIO_FLAG_SCRUB | ZIO_FLAG_RAW, zb));
1828         }
1829         return (0);
1830 }

1832 static int
1833 spa_load_verify(spa_t *spa)
1834 {
1835         zio_t *rio;
1836         spa_load_error_t sle = { 0 };
1837         zpool_rewind_policy_t policy;
1838         boolean_t verify_ok = B_FALSE;
1839         int error;
```

```
1841         zpool_get_rewind_policy(spa->spa_config, &policy);

1843         if (policy.zrp_request & ZPOOL_NEVER_REWIND)
1844                 return (0);

1846         rio = zio_root(spa, NULL, &sle,
1847             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE);

1849         error = traverse_pool(spa, spa->spa_verify_min_txg,
1850             TRAVERSE_PRE | TRAVERSE_PREFETCH, spa_load_verify_cb, rio);

1852         (void) zio_wait(rio);

1854         spa->spa_load_meta_errors = sle.sle_meta_count;
1855         spa->spa_load_data_errors = sle.sle_data_count;

1857         if (!error && sle.sle_meta_count <= policy.zrp_maxmeta &&
1858             sle.sle_data_count <= policy.zrp_maxdata) {
1859                 int64_t loss = 0;

1861                 verify_ok = B_TRUE;
1862                 spa->spa_load_txg = spa->spa_uberblock.ub_txg;
1863                 spa->spa_load_txg_ts = spa->spa_uberblock.ub_timestamp;

1865                 loss = spa->spa_last_ubsync_txg_ts - spa->spa_load_txg_ts;
1866                 VERIFY(nvlist_add_uint64(spa->spa_load_info,
1867                     ZPOOL_CONFIG_LOAD_TIME, spa->spa_load_txg_ts) == 0);
1868                 VERIFY(nvlist_add_int64(spa->spa_load_info,
1869                     ZPOOL_CONFIG_REWIND_TIME, loss) == 0);
1870                 VERIFY(nvlist_add_uint64(spa->spa_load_info,
1871                     ZPOOL_CONFIG_LOAD_DATA_ERRORS, sle.sle_data_count) == 0);
1872         } else {
1873                 spa->spa_load_max_txg = spa->spa_uberblock.ub_txg;
1874         }

1876         if (error) {
1877                 if (error != ENXIO && error != EIO)
1878                         error = SET_ERROR(EIO);
1879                 return (error);
1880         }

1882         return (verify_ok ? 0 : EIO);
1883 }

1885 /*
1886  * Find a value in the pool props object.
1887  */
1888 static void
1889 spa_prop_find(spa_t *spa, zpool_prop_t prop, uint64_t *val)
1890 {
1891         (void) zap_lookup(spa->spa_meta_objset, spa->spa_pool_props_object,
1892             zpool_prop_to_name(prop), sizeof (uint64_t), 1, val);
1893 }

1895 /*
1896  * Find a value in the pool directory object.
1897  */
1898 static int
1899 spa_dir_prop(spa_t *spa, const char *name, uint64_t *val)
1900 {
1901         return (zap_lookup(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
1902             name, sizeof (uint64_t), 1, val));
1903 }

1905 static int
```



```
1906 spa_vdev_err(vdev_t *vdev, vdev_aux_t aux, int err)
1907 {
1908         vdev_set_state(vdev, B_TRUE, VDEV_STATE_CANT_OPEN, aux);
1909         return (err);
1910 }

1912 /*
1913  * Fix up config after a partly-completed split.  This is done with the
1914  * ZPOOL_CONFIG_SPLIT nvlist.  Both the splitting pool and the split-off
1915  * pool have that entry in their config, but only the splitting one contains
1916  * a list of all the guids of the vdevs that are being split off.
1917  *
1918  * This function determines what to do with that list: either rejoin
1919  * all the disks to the pool, or complete the splitting process.  To attempt
1920  * the rejoin, each disk that is offlined is marked online again, and
1921  * we do a reopen() call.  If the vdev label for every disk that was
1922  * marked online indicates it was successfully split off (VDEV_AUX_SPLIT_POOL)
1923  * then we call vdev_split() on each disk, and complete the split.
1924  *
1925  * Otherwise we leave the config alone, with all the vdevs in place in
1926  * the original pool.
1927  */
1928 static void
1929 spa_try_repair(spa_t *spa, nvlist_t *config)
1930 {
1931         uint_t extracted;
1932         uint64_t *glist;
1933         uint_t i, gcount;
1934         nvlist_t *nvl;
1935         vdev_t **vd;
1936         boolean_t attempt_reopen;

1938         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_SPLIT, &nvl) != 0)
1939                 return;

1941         /* check that the config is complete */
1942         if (nvlist_lookup_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
1943             &glist, &gcount) != 0)
1944                 return;

1946         vd = kmem_zalloc(gcount * sizeof (vdev_t *), KM_SLEEP);

1948         /* attempt to online all the vdevs & validate */
1949         attempt_reopen = B_TRUE;
1950         for (i = 0; i < gcount; i++) {
1951                 if (glist[i] == 0)      /* vdev is hole */
1952                         continue;

1954                 vd[i] = spa_lookup_by_guid(spa, glist[i], B_FALSE);
1955                 if (vd[i] == NULL) {
1956                         /*
1957                          * Don't bother attempting to reopen the disks;
1958                          * just do the split.
1959                          */
1960                         attempt_reopen = B_FALSE;
1961                 } else {
1962                         /* attempt to re-online it */
1963                         vd[i]->vdev_offline = B_FALSE;
1964                 }
1965         }

1967         if (attempt_reopen) {
1968                 vdev_reopen(spa->spa_root_vdev);

1970                 /* check each device to see what state it's in */
1971                 for (extracted = 0, i = 0; i < gcount; i++) {
```

```
1972                        if (vd[i] != NULL &&
1973                            vd[i]->vdev_stat.vs_aux != VDEV_AUX_SPLIT_POOL)
1974                                break;
1975                        ++extracted;
1976                }
1977        }
1978
1979        /*
1980         * If every disk has been moved to the new pool, or if we never
1981         * even attempted to look at them, then we split them off for
1982         * good.
1983         */
1984        if (!attempt_reopen || gcount == extracted) {
1985                for (i = 0; i < gcount; i++)
1986                        if (vd[i] != NULL)
1987                                vdev_split(vd[i]);
1988                vdev_reopen(spa->spa_root_vdev);
1989        }
1990
1991        kmem_free(vd, gcount * sizeof (vdev_t *));
1992 }
1993
1994 static int
1995 spa_load(spa_t *spa, spa_load_state_t state, spa_import_type_t type,
1996     boolean_t mosconfig)
1997 {
1998        nvlist_t *config = spa->spa_config;
1999        char *ereport = FM_EREPORT_ZFS_POOL;
2000        char *comment;
2001        int error;
2002        uint64_t pool_guid;
2003        nvlist_t *nvl;
2004
2005        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &pool_guid))
2006                return (SET_ERROR(EINVAL));
2007
2008        ASSERT(spa->spa_comment == NULL);
2009        if (nvlist_lookup_string(config, ZPOOL_CONFIG_COMMENT, &comment) == 0)
2010                spa->spa_comment = spa_strdup(comment);
2011
2012        /*
2013         * Versioning wasn't explicitly added to the label until later, so if
2014         * it's not present treat it as the initial version.
2015         */
2016        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VERSION,
2017            &spa->spa_ubsync.ub_version) != 0)
2018                spa->spa_ubsync.ub_version = SPA_VERSION_INITIAL;
2019
2020        (void) nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
2021            &spa->spa_config_txg);
2022
2023        if ((state == SPA_LOAD_IMPORT || state == SPA_LOAD_TRYIMPORT) &&
2024            spa_guid_exists(pool_guid, 0)) {
2025                error = SET_ERROR(EEXIST);
2026        } else {
2027                spa->spa_config_guid = pool_guid;
2028
2029                if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_SPLIT,
2030                    &nvl) == 0) {
2031                        VERIFY(nvlist_dup(nvl, &spa->spa_config_splitting,
2032                            KM_SLEEP) == 0);
2033                }
2034
2035                nvlist_free(spa->spa_load_info);
2036                spa->spa_load_info = fnvlist_alloc();
```



```
2038                gethrestime(&spa->spa_loaded_ts);
2039                error = spa_load_impl(spa, pool_guid, config, state, type,
2040                    mosconfig, &ereport);
2041        }
2042
2043        spa->spa_minref = refcount_count(&spa->spa_refcount);
2044        if (error) {
2045                if (error != EEXIST) {
2046                        spa->spa_loaded_ts.tv_sec = 0;
2047                        spa->spa_loaded_ts.tv_nsec = 0;
2048                }
2049                if (error != EBADF) {
2050                        zfs_ereport_post(ereport, spa, NULL, NULL, 0, 0);
2051                }
2052        }
2053        spa->spa_load_state = error ? SPA_LOAD_ERROR : SPA_LOAD_NONE;
2054        spa->spa_ena = 0;
2055
2056        return (error);
2057 }
2058
2059 /*
2060  * Load an existing storage pool, using the pool's builtin spa_config as a
2061  * source of configuration information.
2062  */
2063 static int
2064 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2065     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2066     char **ereport)
2067 {
2068        int error = 0;
2069        nvlist_t *nvroot = NULL;
2070        nvlist_t *label;
2071        vdev_t *rvd;
2072        uberblock_t *ub = &spa->spa_uberblock;
2073        uint64_t children, config_cache_txg = spa->spa_config_txg;
2074        int orig_mode = spa->spa_mode;
2075        int parse;
2076        uint64_t obj;
2077        boolean_t missing_feat_write = B_FALSE;
2078
2079        /*
2080         * If this is an untrusted config, access the pool in read-only mode.
2081         * This prevents things like resilvering recently removed devices.
2082         */
2083        if (!mosconfig)
2084                spa->spa_mode = FREAD;
2085
2086        ASSERT(MUTEX_HELD(&spa_namespace_lock));
2087
2088        spa->spa_load_state = state;
2089
2090        if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2091                return (SET_ERROR(EINVAL));
2092
2093        parse = (type == SPA_IMPORT_EXISTING ?
2094            VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);
2095
2096        /*
2097         * Create "The Godfather" zio to hold all async IOs
2098         */
2099        spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2100            ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
2101
2102        /*
2103         * Parse the configuration into a vdev tree.  We explicitly set the
```

```
2104          * value that will be returned by spa_version() since parsing the
2105          * configuration requires knowing the version number.
2106          */
2107         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2108         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2109         spa_config_exit(spa, SCL_ALL, FTAG);
2111         if (error != 0)
2112                 return (error);
2114         ASSERT(spa->spa_root_vdev == rvd);
2116         if (type != SPA_IMPORT_ASSEMBLE) {
2117                 ASSERT(spa_guid(spa) == pool_guid);
2118         }
2120         /*
2121          * Try to open all vdevs, loading each label in the process.
2122          */
2123         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2124         error = vdev_open(rvd);
2125         spa_config_exit(spa, SCL_ALL, FTAG);
2126         if (error != 0)
2127                 return (error);
2129         /*
2130          * We need to validate the vdev labels against the configuration that
2131          * we have in hand, which is dependent on the setting of mosconfig. If
2132          * mosconfig is true then we're validating the vdev labels based on
2133          * that config.  Otherwise, we're validating against the cached config
2134          * (zpool.cache) that was read when we loaded the zfs module, and then
2135          * later we will recursively call spa_load() and validate against
2136          * the vdev config.
2137          *
2138          * If we're assembling a new pool that's been split off from an
2139          * existing pool, the labels haven't yet been updated so we skip
2140          * validation for now.
2141          */
2142         if (type != SPA_IMPORT_ASSEMBLE) {
2143                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2144                 error = vdev_validate(rvd, mosconfig);
2145                 spa_config_exit(spa, SCL_ALL, FTAG);
2147                 if (error != 0)
2148                         return (error);
2150                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2151                         return (SET_ERROR(ENXIO));
2152         }
2154         /*
2155          * Find the best uberblock.
2156          */
2157         vdev_uberblock_load(rvd, ub, &label);
2159         /*
2160          * If we weren't able to find a single valid uberblock, return failure.
2161          */
2162         if (ub->ub_txg == 0) {
2163                 nvlist_free(label);
2164                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2165         }
2167         /*
2168          * If the pool has an unsupported version we can't open it.
2169          */
```



```
2170         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2171                 nvlist_free(label);
2172                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2173         }
2175         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2176                 nvlist_t *features;
2178                 /*
2179                  * If we weren't able to find what's necessary for reading the
2180                  * MOS in the label, return failure.
2181                  */
2182                 if (label == NULL || nvlist_lookup_nvlist(label,
2183                     ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2184                         nvlist_free(label);
2185                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2186                             ENXIO));
2187                 }
2189                 /*
2190                  * Update our in-core representation with the definitive values
2191                  * from the label.
2192                  */
2193                 nvlist_free(spa->spa_label_features);
2194                 VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2195         }
2197         nvlist_free(label);
2199         /*
2200          * Look through entries in the label nvlist's features_for_read. If
2201          * there is a feature listed there which we don't understand then we
2202          * cannot open a pool.
2203          */
2204         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2205                 nvlist_t *unsup_feat;
2207                 VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2208                     0);
2210                 for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2211                     NULL); nvp != NULL;
2212                     nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2213                         if (!zfeature_is_supported(nvpair_name(nvp))) {
2214                                 VERIFY(nvlist_add_string(unsup_feat,
2215                                     nvpair_name(nvp), "") == 0);
2216                         }
2217                 }
2219                 if (!nvlist_empty(unsup_feat)) {
2220                         VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2221                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2222                         nvlist_free(unsup_feat);
2223                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2224                             ENOTSUP));
2225                 }
2227                 nvlist_free(unsup_feat);
2228         }
2230         /*
2231          * If the vdev guid sum doesn't match the uberblock, we have an
2232          * incomplete configuration.  We first check to see if the pool
2233          * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2234          * If it is, defer the vdev_guid_sum check till later so we
2235          * can handle missing vdevs.
```

```
2236          */
2237         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2238             &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2239             rvd->vdev_guid_sum != ub->ub_guid_sum)
2240                 return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

2242         if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2243                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2244                 spa_try_repair(spa, config);
2245                 spa_config_exit(spa, SCL_ALL, FTAG);
2246                 nvlist_free(spa->spa_config_splitting);
2247                 spa->spa_config_splitting = NULL;
2248         }

2250         /*
2251          * Initialize internal SPA structures.
2252          */
2253         spa->spa_state = POOL_STATE_ACTIVE;
2254         spa->spa_ubsync = spa->spa_uberblock;
2255         spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2256             TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2257         spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2258             spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2259         spa->spa_claim_max_txg = spa->spa_first_txg;
2260         spa->spa_prev_software_version = ub->ub_software_version;

2262         error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2263         if (error)
2264                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2265         spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2267         if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2268                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2270         if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2271                 boolean_t missing_feat_read = B_FALSE;
2272                 nvlist_t *unsup_feat, *enabled_feat;

2274                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2275                     &spa->spa_feat_for_read_obj) != 0) {
2276                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2277                 }

2279                 if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2280                     &spa->spa_feat_for_write_obj) != 0) {
2281                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2282                 }

2284                 if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2285                     &spa->spa_feat_desc_obj) != 0) {
2286                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2287                 }

2289                 enabled_feat = fnvlist_alloc();
2290                 unsup_feat = fnvlist_alloc();

2292                 if (!feature_is_supported(spa->spa_meta_objset,
2293                     spa->spa_feat_for_read_obj, spa->spa_feat_desc_obj,
2294                     unsup_feat, enabled_feat))
2295                         missing_feat_read = B_TRUE;

2297                 if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2298                         if (!feature_is_supported(spa->spa_meta_objset,
2299                             spa->spa_feat_for_write_obj, spa->spa_feat_desc_obj,
2300                             unsup_feat, enabled_feat)) {
2301                                 missing_feat_write = B_TRUE;
```



```
2302                         }
2303                 }

2305                 fnvlist_add_nvlist(spa->spa_load_info,
2306                     ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2308                 if (!nvlist_empty(unsup_feat)) {
2309                         fnvlist_add_nvlist(spa->spa_load_info,
2310                             ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2311                 }

2313                 fnvlist_free(enabled_feat);
2314                 fnvlist_free(unsup_feat);

2316                 if (!missing_feat_read) {
2317                         fnvlist_add_boolean(spa->spa_load_info,
2318                             ZPOOL_CONFIG_CAN_RDONLY);
2319                 }

2321                 /*
2322                  * If the state is SPA_LOAD_TRYIMPORT, our objective is
2323                  * twofold: to determine whether the pool is available for
2324                  * import in read-write mode and (if it is not) whether the
2325                  * pool is available for import in read-only mode. If the pool
2326                  * is available for import in read-write mode, it is displayed
2327                  * as available in userland; if it is not available for import
2328                  * in read-only mode, it is displayed as unavailable in
2329                  * userland. If the pool is available for import in read-only
2330                  * mode but not read-write mode, it is displayed as unavailable
2331                  * in userland with a special note that the pool is actually
2332                  * available for open in read-only mode.
2333                  *
2334                  * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2335                  * missing a feature for write, we must first determine whether
2336                  * the pool can be opened read-only before returning to
2337                  * userland in order to know whether to display the
2338                  * abovementioned note.
2339                  */
2340                 if (missing_feat_read || (missing_feat_write &&
2341                     spa_writeable(spa))) {
2342                         return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2343                             ENOTSUP));
2344                 }
2345         }

2347         spa->spa_is_initializing = B_TRUE;
2348         error = dsl_pool_open(spa->spa_dsl_pool);
2349         spa->spa_is_initializing = B_FALSE;
2350         if (error != 0)
2351                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2353         if (!mosconfig) {
2354                 uint64_t hostid;
2355                 nvlist_t *policy = NULL, *nvconfig;

2357                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2358                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2360                 if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2361                     ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2362                         char *hostname;
2363                         unsigned long myhostid = 0;

2365                         VERIFY(nvlist_lookup_string(nvconfig,
2366                             ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);
```

```
#ifdef  _KERNEL
                        myhostid = zone_get_hostid(NULL);
#else   /* _KERNEL */
                        /*
                         * We're emulating the system's hostid in userland, so
                         * we can't use zone_get_hostid().
                         */
                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
#endif  /* _KERNEL */
                        if (hostid != 0 && myhostid != 0 &&
                            hostid != myhostid) {
                                nvlist_free(nvconfig);
                                cmn_err(CE_WARN, "pool '%s' could not be "
                                    "loaded as it was last accessed by "
                                    "another system (host: %s hostid: 0x%lx). "
                                    "See: http://illumos.org/msg/ZFS-8000-EY",
                                    spa_name(spa), hostname,
                                    (unsigned long)hostid);
                                return (SET_ERROR(EBADF));
                        }
                }
                if (nvlist_lookup_nvlist(spa->spa_config,
                    ZPOOL_REWIND_POLICY, &policy) == 0)
                        VERIFY(nvlist_add_nvlist(nvconfig,
                            ZPOOL_REWIND_POLICY, policy) == 0);

                spa_config_set(spa, nvconfig);
                spa_unload(spa);
                spa_deactivate(spa);
                spa_activate(spa, orig_mode);

                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
        }

        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
        if (error != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * Load the bit that tells us to use the new accounting function
         * (raid-z deflation).  If we have an older pool, this will not
         * be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
            &spa->spa_creation_version);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * Load the persistent error log.  If we have an older pool, this will
         * not be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
            &spa->spa_errlog_scrub);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
```



```
        /*
         * Load the history object.  If we have an older pool, this
         * will not be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * If we're assembling the pool from the split-off vdevs of
         * an existing pool, we don't want to attach the spares & cache
         * devices.
         */

        /*
         * Load any hot spares for this pool.
         */
        error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
                ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
                if (load_nvlist(spa, spa->spa_spares.sav_object,
                    &spa->spa_spares.sav_config) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                spa_load_spares(spa);
                spa_config_exit(spa, SCL_ALL, FTAG);
        } else if (error == 0) {
                spa->spa_spares.sav_sync = B_TRUE;
        }

        /*
         * Load any level 2 ARC devices for this pool.
         */
        error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
            &spa->spa_l2cache.sav_object);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
                ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
                if (load_nvlist(spa, spa->spa_l2cache.sav_object,
                    &spa->spa_l2cache.sav_config) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                spa_load_l2cache(spa);
                spa_config_exit(spa, SCL_ALL, FTAG);
        } else if (error == 0) {
                spa->spa_l2cache.sav_sync = B_TRUE;
        }

        spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

        error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
        if (error && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        if (error == 0) {
                uint64_t autoreplace;

                spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
                spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
                spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
```

```
2500                spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2501                spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2502                spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2503                    &spa->spa_dedup_ditto);

2505                spa->spa_autoreplace = (autoreplace != 0);
2506        }

2508        /*
2509         * If the 'autoreplace' property is set, then post a resource notifying
2510         * the ZFS DE that it should not issue any faults for unopenable
2511         * devices.  We also iterate over the vdevs, and post a sysevent for any
2512         * unopenable vdevs so that the normal autoreplace handler can take
2513         * over.
2514         */
2515        if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2516                spa_check_removed(spa->spa_root_vdev);
2517                /*
2518                 * For the import case, this is done in spa_import(), because
2519                 * at this point we're using the spare definitions from
2520                 * the MOS config, not necessarily from the userland config.
2521                 */
2522                if (state != SPA_LOAD_IMPORT) {
2523                        spa_aux_check_removed(&spa->spa_spares);
2524                        spa_aux_check_removed(&spa->spa_l2cache);
2525                }
2526        }

2528        /*
2529         * Load the vdev state for all toplevel vdevs.
2530         */
2531        vdev_load(rvd);

2533        /*
2534         * Propagate the leaf DTLs we just loaded all the way up the tree.
2535         */
2536        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2537        vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2538        spa_config_exit(spa, SCL_ALL, FTAG);

2540        /*
2541         * Load the DDTs (dedup tables).
2542         */
2543        error = ddt_load(spa);
2544        if (error != 0)
2545                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2547        spa_update_dspace(spa);

2549        /*
2550         * Validate the config, using the MOS config to fill in any
2551         * information which might be missing.  If we fail to validate
2552         * the config then declare the pool unfit for use. If we're
2553         * assembling a pool from a split, the log is not transferred
2554         * over.
2555         */
2556        if (type != SPA_IMPORT_ASSEMBLE) {
2557                nvlist_t *nvconfig;

2559                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2560                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2562                if (!spa_config_valid(spa, nvconfig)) {
2563                        nvlist_free(nvconfig);
2564                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2565                            ENXIO));
```



```
2566                }
2567                nvlist_free(nvconfig);

2569                /*
2570                 * Now that we've validated the config, check the state of the
2571                 * root vdev.  If it can't be opened, it indicates one or
2572                 * more toplevel vdevs are faulted.
2573                 */
2574                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2575                        return (SET_ERROR(ENXIO));

2577                if (spa_check_logs(spa)) {
2578                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2579                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2580                }
2581        }

2583        if (missing_feat_write) {
2584                ASSERT(state == SPA_LOAD_TRYIMPORT);

2586                /*
2587                 * At this point, we know that we can open the pool in
2588                 * read-only mode but not read-write mode. We now have enough
2589                 * information and can return to userland.
2590                 */
2591                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2592        }

2594        /*
2595         * We've successfully opened the pool, verify that we're ready
2596         * to start pushing transactions.
2597         */
2598        if (state != SPA_LOAD_TRYIMPORT) {
2599                if (error = spa_load_verify(spa))
2600                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2601                            error));
2602        }

2604        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2605            spa->spa_load_max_txg == UINT64_MAX)) {
2606                dmu_tx_t *tx;
2607                int need_update = B_FALSE;

2609                ASSERT(state != SPA_LOAD_TRYIMPORT);

2611                /*
2612                 * Claim log blocks that haven't been committed yet.
2613                 * This must all happen in a single txg.
2614                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2615                 * invoked from zil_claim_log_block()'s i/o done callback.
2616                 * Price of rollback is that we abandon the log.
2617                 */
2618                spa->spa_claiming = B_TRUE;

2620                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2621                    spa_first_txg(spa));
2622                (void) dmu_objset_find(spa_name(spa),
2623                    zil_claim, tx, DS_FIND_CHILDREN);
2624                dmu_tx_commit(tx);

2626                spa->spa_claiming = B_FALSE;

2628                spa_set_log_state(spa, SPA_LOG_GOOD);
2629                spa->spa_sync_on = B_TRUE;
2630                txg_sync_start(spa->spa_dsl_pool);
```

```
2632                 /*
2633                  * Wait for all claims to sync.  We sync up to the highest
2634                  * claimed log block birth time so that claimed log blocks
2635                  * don't appear to be from the future.  spa_claim_max_txg
2636                  * will have been set for us by either zil_check_log_chain()
2637                  * (invoked from spa_check_logs()) or zil_claim() above.
2638                  */
2639                 txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

2641                 /*
2642                  * If the config cache is stale, or we have uninitialized
2643                  * metaslabs (see spa_vdev_add()), then update the config.
2644                  *
2645                  * If this is a verbatim import, trust the current
2646                  * in-core spa_config and update the disk labels.
2647                  */
2648                 if (config_cache_txg != spa->spa_config_txg ||
2649                     state == SPA_LOAD_IMPORT ||
2650                     state == SPA_LOAD_RECOVER ||
2651                     (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2652                         need_update = B_TRUE;

2654                 for (int c = 0; c < rvd->vdev_children; c++)
2655                         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2656                                 need_update = B_TRUE;

2658                 /*
2659                  * Update the config cache asychronously in case we're the
2660                  * root pool, in which case the config cache isn't writable yet.
2661                  */
2662                 if (need_update)
2663                         spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2665                 /*
2666                  * Check all DTLs to see if anything needs resilvering.
2667                  */
2668                 if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2669                     vdev_resilver_needed(rvd, NULL, NULL))
2670                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2672                 /*
2673                  * Log the fact that we booted up (so that we can detect if
2674                  * we rebooted in the middle of an operation).
2675                  */
2676                 spa_history_log_version(spa, "open");

2678                 /*
2679                  * Delete any inconsistent datasets.
2680                  */
2681                 (void) dmu_objset_find(spa_name(spa),
2682                     dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2684                 /*
2685                  * Clean up any stale temporary dataset userrefs.
2686                  */
2687                 dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2688         }

2690         return (0);
2691 }

2693 static int
2694 spa_load_retry(spa_t *spa, spa_load_state_t state, int mosconfig)
2695 {
2696         int mode = spa->spa_mode;
```



```
2698         spa_unload(spa);
2699         spa_deactivate(spa);

2701         spa->spa_load_max_txg--;

2703         spa_activate(spa, mode);
2704         spa_async_suspend(spa);

2706         return (spa_load(spa, state, SPA_IMPORT_EXISTING, mosconfig));
2707 }

2709 /*
2710  * If spa_load() fails this function will try loading prior txg's. If
2711  * 'state' is SPA_LOAD_RECOVER and one of these loads succeeds the pool
2712  * will be rewound to that txg. If 'state' is not SPA_LOAD_RECOVER this
2713  * function will not rewind the pool and will return the same error as
2714  * spa_load().
2715  */
2716 static int
2717 spa_load_best(spa_t *spa, spa_load_state_t state, int mosconfig,
2718     uint64_t max_request, int rewind_flags)
2719 {
2720         nvlist_t *loadinfo = NULL;
2721         nvlist_t *config = NULL;
2722         int load_error, rewind_error;
2723         uint64_t safe_rewind_txg;
2724         uint64_t min_txg;

2726         if (spa->spa_load_txg && state == SPA_LOAD_RECOVER) {
2727                 spa->spa_load_max_txg = spa->spa_load_txg;
2728                 spa_set_log_state(spa, SPA_LOG_CLEAR);
2729         } else {
2730                 spa->spa_load_max_txg = max_request;
2731         }

2733         load_error = rewind_error = spa_load(spa, state, SPA_IMPORT_EXISTING,
2734             mosconfig);
2735         if (load_error == 0)
2736                 return (0);

2738         if (spa->spa_root_vdev != NULL)
2739                 config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);

2741         spa->spa_last_ubsync_txg = spa->spa_uberblock.ub_txg;
2742         spa->spa_last_ubsync_txg_ts = spa->spa_uberblock.ub_timestamp;

2744         if (rewind_flags & ZPOOL_NEVER_REWIND) {
2745                 nvlist_free(config);
2746                 return (load_error);
2747         }

2749         if (state == SPA_LOAD_RECOVER) {
2750                 /* Price of rolling back is discarding txgs, including log */
2751                 spa_set_log_state(spa, SPA_LOG_CLEAR);
2752         } else {
2753                 /*
2754                  * If we aren't rolling back save the load info from our first
2755                  * import attempt so that we can restore it after attempting
2756                  * to rewind.
2757                  */
2758                 loadinfo = spa->spa_load_info;
2759                 spa->spa_load_info = fnvlist_alloc();
2760         }

2762         spa->spa_load_max_txg = spa->spa_last_ubsync_txg;
2763         safe_rewind_txg = spa->spa_last_ubsync_txg - TXG_DEFER_SIZE;
```

```
2764         min_txg = (rewind_flags & ZPOOL_EXTREME_REWIND) ?
2765             TXG_INITIAL : safe_rewind_txg;
2767         /*
2768          * Continue as long as we're finding errors, we're still within
2769          * the acceptable rewind range, and we're still finding uberblocks
2770          */
2771         while (rewind_error && spa->spa_uberblock.ub_txg >= min_txg &&
2772             spa->spa_uberblock.ub_txg <= spa->spa_load_max_txg) {
2773                 if (spa->spa_load_max_txg < safe_rewind_txg)
2774                         spa->spa_extreme_rewind = B_TRUE;
2775                 rewind_error = spa_load_retry(spa, state, mosconfig);
2776         }
2778         spa->spa_extreme_rewind = B_FALSE;
2779         spa->spa_load_max_txg = UINT64_MAX;
2781         if (config && (rewind_error || state != SPA_LOAD_RECOVER))
2782                 spa_config_set(spa, config);
2784         if (state == SPA_LOAD_RECOVER) {
2785                 ASSERT3P(loadinfo, ==, NULL);
2786                 return (rewind_error);
2787         } else {
2788                 /* Store the rewind info as part of the initial load info */
2789                 fnvlist_add_nvlist(loadinfo, ZPOOL_CONFIG_REWIND_INFO,
2790                     spa->spa_load_info);
2792                 /* Restore the initial load info */
2793                 fnvlist_free(spa->spa_load_info);
2794                 spa->spa_load_info = loadinfo;
2796                 return (load_error);
2797         }
2798 }
2800 /*
2801  * Pool Open/Import
2802  *
2803  * The import case is identical to an open except that the configuration is sent
2804  * down from userland, instead of grabbed from the configuration cache.  For the
2805  * case of an open, the pool configuration will exist in the
2806  * POOL_STATE_UNINITIALIZED state.
2807  *
2808  * The stats information (gen/count/ustats) is used to gather vdev statistics at
2809  * the same time open the pool, without having to keep around the spa_t in some
2810  * ambiguous state.
2811  */
2812 static int
2813 spa_open_common(const char *pool, spa_t **spapp, void *tag, nvlist_t *nvpolicy,
2814     nvlist_t **config)
2815 {
2816         spa_t *spa;
2817         spa_load_state_t state = SPA_LOAD_OPEN;
2818         int error;
2819         int locked = B_FALSE;
2821         *spapp = NULL;
2823         /*
2824          * As disgusting as this is, we need to support recursive calls to this
2825          * function because dsl_dir_open() is called during spa_load(), and ends
2826          * up calling spa_open() again.  The real fix is to figure out how to
2827          * avoid dsl_dir_open() calling this in the first place.
2828          */
2829         if (mutex_owner(&spa_namespace_lock) != curthread) {
```



```
2830                 mutex_enter(&spa_namespace_lock);
2831                 locked = B_TRUE;
2832         }
2834         if ((spa = spa_lookup(pool)) == NULL) {
2835                 if (locked)
2836                         mutex_exit(&spa_namespace_lock);
2837                 return (SET_ERROR(ENOENT));
2838         }
2840         if (spa->spa_state == POOL_STATE_UNINITIALIZED) {
2841                 zpool_rewind_policy_t policy;
2843                 zpool_get_rewind_policy(nvpolicy ? nvpolicy : spa->spa_config,
2844                     &policy);
2845                 if (policy.zrp_request & ZPOOL_DO_REWIND)
2846                         state = SPA_LOAD_RECOVER;
2848                 spa_activate(spa, spa_mode_global);
2850                 if (state != SPA_LOAD_RECOVER)
2851                         spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;
2853                 error = spa_load_best(spa, state, B_FALSE, policy.zrp_txg,
2854                     policy.zrp_request);
2856                 if (error == EBADF) {
2857                         /*
2858                          * If vdev_validate() returns failure (indicated by
2859                          * EBADF), it indicates that one of the vdevs indicates
2860                          * that the pool has been exported or destroyed.  If
2861                          * this is the case, the config cache is out of sync and
2862                          * we should remove the pool from the namespace.
2863                          */
2864                         spa_unload(spa);
2865                         spa_deactivate(spa);
2866                         spa_config_sync(spa, B_TRUE, B_TRUE);
2867                         spa_remove(spa);
2868                         if (locked)
2869                                 mutex_exit(&spa_namespace_lock);
2870                         return (SET_ERROR(ENOENT));
2871                 }
2873                 if (error) {
2874                         /*
2875                          * We can't open the pool, but we still have useful
2876                          * information: the state of each vdev after the
2877                          * attempted vdev_open().  Return this to the user.
2878                          */
2879                         if (config != NULL && spa->spa_config) {
2880                                 VERIFY(nvlist_dup(spa->spa_config, config,
2881                                     KM_SLEEP) == 0);
2882                                 VERIFY(nvlist_add_nvlist(*config,
2883                                     ZPOOL_CONFIG_LOAD_INFO,
2884                                     spa->spa_load_info) == 0);
2885                         }
2886                         spa_unload(spa);
2887                         spa_deactivate(spa);
2888                         spa->spa_last_open_failed = error;
2889                         if (locked)
2890                                 mutex_exit(&spa_namespace_lock);
2891                         *spapp = NULL;
2892                         return (error);
2893                 }
2894         }
```

```
2896         spa_open_ref(spa, tag);

2898         if (config != NULL)
2899                 *config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);

2901         /*
2902          * If we've recovered the pool, pass back any information we
2903          * gathered while doing the load.
2904          */
2905         if (state == SPA_LOAD_RECOVER) {
2906                 VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO,
2907                     spa->spa_load_info) == 0);
2908         }

2910         if (locked) {
2911                 spa->spa_last_open_failed = 0;
2912                 spa->spa_last_ubsync_txg = 0;
2913                 spa->spa_load_txg = 0;
2914                 mutex_exit(&spa_namespace_lock);
2915         }

2917         *spapp = spa;

2919         return (0);
2920 }

2922 int
2923 spa_open_rewind(const char *name, spa_t **spapp, void *tag, nvlist_t *policy,
2924     nvlist_t **config)
2925 {
2926         return (spa_open_common(name, spapp, tag, policy, config));
2927 }

2929 int
2930 spa_open(const char *name, spa_t **spapp, void *tag)
2931 {
2932         return (spa_open_common(name, spapp, tag, NULL, NULL));
2933 }

2935 /*
2936  * Lookup the given spa_t, incrementing the inject count in the process,
2937  * preventing it from being exported or destroyed.
2938  */
2939 spa_t *
2940 spa_inject_addref(char *name)
2941 {
2942         spa_t *spa;

2944         mutex_enter(&spa_namespace_lock);
2945         if ((spa = spa_lookup(name)) == NULL) {
2946                 mutex_exit(&spa_namespace_lock);
2947                 return (NULL);
2948         }
2949         spa->spa_inject_ref++;
2950         mutex_exit(&spa_namespace_lock);

2952         return (spa);
2953 }

2955 void
2956 spa_inject_delref(spa_t *spa)
2957 {
2958         mutex_enter(&spa_namespace_lock);
2959         spa->spa_inject_ref--;
2960         mutex_exit(&spa_namespace_lock);
2961 }
```



```
2963 /*
2964  * Add spares device information to the nvlist.
2965  */
2966 static void
2967 spa_add_spares(spa_t *spa, nvlist_t *config)
2968 {
2969         nvlist_t **spares;
2970         uint_t i, nspares;
2971         nvlist_t *nvroot;
2972         uint64_t guid;
2973         vdev_stat_t *vs;
2974         uint_t vsc;
2975         uint64_t pool;

2977         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));

2979         if (spa->spa_spares.sav_count == 0)
2980                 return;

2982         VERIFY(nvlist_lookup_nvlist(config,
2983             ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
2984         VERIFY(nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
2985             ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
2986         if (nspares != 0) {
2987                 VERIFY(nvlist_add_nvlist_array(nvroot,
2988                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
2989                 VERIFY(nvlist_lookup_nvlist_array(nvroot,
2990                     ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);

2992                 /*
2993                  * Go through and find any spares which have since been
2994                  * repurposed as an active spare.  If this is the case, update
2995                  * their status appropriately.
2996                  */
2997                 for (i = 0; i < nspares; i++) {
2998                         VERIFY(nvlist_lookup_uint64(spares[i],
2999                             ZPOOL_CONFIG_GUID, &guid) == 0);
3000                         if (spa_spare_exists(guid, &pool, NULL) &&
3001                             pool != 0ULL) {
3002                                 VERIFY(nvlist_lookup_uint64_array(
3003                                     spares[i], ZPOOL_CONFIG_VDEV_STATS,
3004                                     (uint64_t **)&vs, &vsc) == 0);
3005                                 vs->vs_state = VDEV_STATE_CANT_OPEN;
3006                                 vs->vs_aux = VDEV_AUX_SPARED;
3007                         }
3008                 }
3009         }
3010 }

3012 /*
3013  * Add l2cache device information to the nvlist, including vdev stats.
3014  */
3015 static void
3016 spa_add_l2cache(spa_t *spa, nvlist_t *config)
3017 {
3018         nvlist_t **l2cache;
3019         uint_t i, j, nl2cache;
3020         nvlist_t *nvroot;
3021         uint64_t guid;
3022         vdev_t *vd;
3023         vdev_stat_t *vs;
3024         uint_t vsc;

3026         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
```

```
3028         if (spa->spa_l2cache.sav_count == 0)
3029                 return;
3030
3031         VERIFY(nvlist_lookup_nvlist(config,
3032             ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
3033         VERIFY(nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
3034             ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
3035         if (nl2cache != 0) {
3036                 VERIFY(nvlist_add_nvlist_array(nvroot,
3037                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3038                 VERIFY(nvlist_lookup_nvlist_array(nvroot,
3039                     ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
3040
3041                 /*
3042                  * Update level 2 cache device stats.
3043                  */
3044
3045                 for (i = 0; i < nl2cache; i++) {
3046                         VERIFY(nvlist_lookup_uint64(l2cache[i],
3047                             ZPOOL_CONFIG_GUID, &guid) == 0);
3048
3049                         vd = NULL;
3050                         for (j = 0; j < spa->spa_l2cache.sav_count; j++) {
3051                                 if (guid ==
3052                                     spa->spa_l2cache.sav_vdevs[j]->vdev_guid) {
3053                                         vd = spa->spa_l2cache.sav_vdevs[j];
3054                                         break;
3055                                 }
3056                         }
3057                         ASSERT(vd != NULL);
3058
3059                         VERIFY(nvlist_lookup_uint64_array(l2cache[i],
3060                             ZPOOL_CONFIG_VDEV_STATS, (uint64_t **)&vs, &vsc)
3061                             == 0);
3062                         vdev_get_stats(vd, vs);
3063                 }
3064         }
3065 }
3066
3067 static void
3068 spa_add_feature_stats(spa_t *spa, nvlist_t *config)
3069 {
3070         nvlist_t *features;
3071         zap_cursor_t zc;
3072         zap_attribute_t za;
3073
3074         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
3075         VERIFY(nvlist_alloc(&features, NV_UNIQUE_NAME, KM_SLEEP) == 0);
3076
3077         if (spa->spa_feat_for_read_obj != 0) {
3078                 for (zap_cursor_init(&zc, spa->spa_meta_objset,
3079                     spa->spa_feat_for_read_obj);
3080                     zap_cursor_retrieve(&zc, &za) == 0;
3081                     zap_cursor_advance(&zc)) {
3082                         ASSERT(za.za_integer_length == sizeof (uint64_t) &&
3083                             za.za_num_integers == 1);
3084                         VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
3085                             za.za_first_integer));
3086                 }
3087                 zap_cursor_fini(&zc);
3088         }
3089
3090         if (spa->spa_feat_for_write_obj != 0) {
3091                 for (zap_cursor_init(&zc, spa->spa_meta_objset,
3092                     spa->spa_feat_for_write_obj);
3093                     zap_cursor_retrieve(&zc, &za) == 0;
```



```
3094                     zap_cursor_advance(&zc)) {
3095                         ASSERT(za.za_integer_length == sizeof (uint64_t) &&
3096                             za.za_num_integers == 1);
3097                         VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
3098                             za.za_first_integer));
3099                 }
3100                 zap_cursor_fini(&zc);
3101         }
3102
3103         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
3104             features) == 0);
3105         nvlist_free(features);
3106 }
3107
3108 int
3109 spa_get_stats(const char *name, nvlist_t **config,
3110     char *altroot, size_t buflen)
3111 {
3112         int error;
3113         spa_t *spa;
3114
3115         *config = NULL;
3116         error = spa_open_common(name, &spa, FTAG, NULL, config);
3117
3118         if (spa != NULL) {
3119                 /*
3120                  * This still leaves a window of inconsistency where the spares
3121                  * or l2cache devices could change and the config would be
3122                  * self-inconsistent.
3123                  */
3124                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
3125
3126                 if (*config != NULL) {
3127                         uint64_t loadtimes[2];
3128
3129                         loadtimes[0] = spa->spa_loaded_ts.tv_sec;
3130                         loadtimes[1] = spa->spa_loaded_ts.tv_nsec;
3131                         VERIFY(nvlist_add_uint64_array(*config,
3132                             ZPOOL_CONFIG_LOADED_TIME, loadtimes, 2) == 0);
3133
3134                         VERIFY(nvlist_add_uint64(*config,
3135                             ZPOOL_CONFIG_ERRCOUNT,
3136                             spa_get_errlog_size(spa)) == 0);
3137
3138                         if (spa_suspended(spa))
3139                                 VERIFY(nvlist_add_uint64(*config,
3140                                     ZPOOL_CONFIG_SUSPENDED,
3141                                     spa->spa_failmode) == 0);
3142
3143                         spa_add_spares(spa, *config);
3144                         spa_add_l2cache(spa, *config);
3145                         spa_add_feature_stats(spa, *config);
3146                 }
3147         }
3148
3149         /*
3150          * We want to get the alternate root even for faulted pools, so we cheat
3151          * and call spa_lookup() directly.
3152          */
3153         if (altroot) {
3154                 if (spa == NULL) {
3155                         mutex_enter(&spa_namespace_lock);
3156                         spa = spa_lookup(name);
3157                         if (spa)
3158                                 spa_altroot(spa, altroot, buflen);
3159                         else
```

```
3160                                altroot[0] = '\0';
3161                        spa = NULL;
3162                        mutex_exit(&spa_namespace_lock);
3163                } else {
3164                        spa_altroot(spa, altroot, buflen);
3165                }
3166        }

3168        if (spa != NULL) {
3169                spa_config_exit(spa, SCL_CONFIG, FTAG);
3170                spa_close(spa, FTAG);
3171        }

3173        return (error);
3174 }

3176 /*
3177  * Validate that the auxiliary device array is well formed.  We must have an
3178  * array of nvlists, each which describes a valid leaf vdev.  If this is an
3179  * import (mode is VDEV_ALLOC_SPARE), then we allow corrupted spares to be
3180  * specified, as long as they are well-formed.
3181  */
3182 static int
3183 spa_validate_aux_devs(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode,
3184     spa_aux_vdev_t *sav, const char *config, uint64_t version,
3185     vdev_labeltype_t label)
3186 {
3187        nvlist_t **dev;
3188        uint_t i, ndev;
3189        vdev_t *vd;
3190        int error;

3192        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3194        /*
3195         * It's acceptable to have no devs specified.
3196         */
3197        if (nvlist_lookup_nvlist_array(nvroot, config, &dev, &ndev) != 0)
3198                return (0);

3200        if (ndev == 0)
3201                return (SET_ERROR(EINVAL));

3203        /*
3204         * Make sure the pool is formatted with a version that supports this
3205         * device type.
3206         */
3207        if (spa_version(spa) < version)
3208                return (SET_ERROR(ENOTSUP));

3210        /*
3211         * Set the pending device list so we correctly handle device in-use
3212         * checking.
3213         */
3214        sav->sav_pending = dev;
3215        sav->sav_npending = ndev;

3217        for (i = 0; i < ndev; i++) {
3218                if ((error = spa_config_parse(spa, &vd, dev[i], NULL, 0,
3219                    mode)) != 0)
3220                        goto out;

3222                if (!vd->vdev_ops->vdev_op_leaf) {
3223                        vdev_free(vd);
3224                        error = SET_ERROR(EINVAL);
3225                        goto out;
```



```
3226                }

3228                /*
3229                 * The L2ARC currently only supports disk devices in
3230                 * kernel context.  For user-level testing, we allow it.
3231                 */
3232 #ifdef _KERNEL
3233                if ((strcmp(config, ZPOOL_CONFIG_L2CACHE) == 0) &&
3234                    strcmp(vd->vdev_ops->vdev_op_type, VDEV_TYPE_DISK) != 0) {
3235                        error = SET_ERROR(ENOTBLK);
3236                        vdev_free(vd);
3237                        goto out;
3238                }
3239 #endif
3240                vd->vdev_top = vd;

3242                if ((error = vdev_open(vd)) == 0 &&
3243                    (error = vdev_label_init(vd, crtxg, label)) == 0) {
3244                        VERIFY(nvlist_add_uint64(dev[i], ZPOOL_CONFIG_GUID,
3245                            vd->vdev_guid) == 0);
3246                }

3248                vdev_free(vd);

3250                if (error &&
3251                    (mode != VDEV_ALLOC_SPARE && mode != VDEV_ALLOC_L2CACHE))
3252                        goto out;
3253                else
3254                        error = 0;
3255        }

3257 out:
3258        sav->sav_pending = NULL;
3259        sav->sav_npending = 0;
3260        return (error);
3261 }

3263 static int
3264 spa_validate_aux(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode)
3265 {
3266        int error;

3268        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3270        if ((error = spa_validate_aux_devs(spa, nvroot, crtxg, mode,
3271            &spa->spa_spares, ZPOOL_CONFIG_SPARES, SPA_VERSION_SPARES,
3272            VDEV_LABEL_SPARE)) != 0) {
3273                return (error);
3274        }

3276        return (spa_validate_aux_devs(spa, nvroot, crtxg, mode,
3277            &spa->spa_l2cache, ZPOOL_CONFIG_L2CACHE, SPA_VERSION_L2CACHE,
3278            VDEV_LABEL_L2CACHE));
3279 }

3281 static void
3282 spa_set_aux_vdevs(spa_aux_vdev_t *sav, nvlist_t **devs, int ndevs,
3283     const char *config)
3284 {
3285        int i;

3287        if (sav->sav_config != NULL) {
3288                nvlist_t **olddevs;
3289                uint_t oldndevs;
3290                nvlist_t **newdevs;
```

```
3292                 /*
3293                  * Generate new dev list by concatentating with the
3294                  * current dev list.
3295                  */
3296                 VERIFY(nvlist_lookup_nvlist_array(sav->sav_config, config,
3297                     &olddevs, &oldndevs) == 0);

3299                 newdevs = kmem_alloc(sizeof (void *) *
3300                     (ndevs + oldndevs), KM_SLEEP);
3301                 for (i = 0; i < oldndevs; i++)
3302                         VERIFY(nvlist_dup(olddevs[i], &newdevs[i],
3303                             KM_SLEEP) == 0);
3304                 for (i = 0; i < ndevs; i++)
3305                         VERIFY(nvlist_dup(devs[i], &newdevs[i + oldndevs],
3306                             KM_SLEEP) == 0);

3308                 VERIFY(nvlist_remove(sav->sav_config, config,
3309                     DATA_TYPE_NVLIST_ARRAY) == 0);

3311                 VERIFY(nvlist_add_nvlist_array(sav->sav_config,
3312                     config, newdevs, ndevs + oldndevs) == 0);
3313                 for (i = 0; i < oldndevs + ndevs; i++)
3314                         nvlist_free(newdevs[i]);
3315                 kmem_free(newdevs, (oldndevs + ndevs) * sizeof (void *));
3316         } else {
3317                 /*
3318                  * Generate a new dev list.
3319                  */
3320                 VERIFY(nvlist_alloc(&sav->sav_config, NV_UNIQUE_NAME,
3321                     KM_SLEEP) == 0);
3322                 VERIFY(nvlist_add_nvlist_array(sav->sav_config, config,
3323                     devs, ndevs) == 0);
3324         }
3325 }

3327 /*
3328  * Stop and drop level 2 ARC devices
3329  */
3330 void
3331 spa_l2cache_drop(spa_t *spa)
3332 {
3333         vdev_t *vd;
3334         int i;
3335         spa_aux_vdev_t *sav = &spa->spa_l2cache;

3337         for (i = 0; i < sav->sav_count; i++) {
3338                 uint64_t pool;

3340                 vd = sav->sav_vdevs[i];
3341                 ASSERT(vd != NULL);

3343                 if (spa_l2cache_exists(vd->vdev_guid, &pool) &&
3344                     pool != 0ULL && l2arc_vdev_present(vd))
3345                         l2arc_remove_vdev(vd);
3346         }
3347 }

3349 /*
3350  * Pool Creation
3351  */
3352 int
3353 spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props,
3354     nvlist_t *zplprops)
3355 {
3356         spa_t *spa;
3357         char *altroot = NULL;
```



```
3358         vdev_t *rvd;
3359         dsl_pool_t *dp;
3360         dmu_tx_t *tx;
3361         int error = 0;
3362         uint64_t txg = TXG_INITIAL;
3363         nvlist_t **spares, **l2cache;
3364         uint_t nspares, nl2cache;
3365         uint64_t version, obj;
3366         boolean_t has_features;

3368         /*
3369          * If this pool already exists, return failure.
3370          */
3371         mutex_enter(&spa_namespace_lock);
3372         if (spa_lookup(pool) != NULL) {
3373                 mutex_exit(&spa_namespace_lock);
3374                 return (SET_ERROR(EEXIST));
3375         }

3377         /*
3378          * Allocate a new spa_t structure.
3379          */
3380         (void) nvlist_lookup_string(props,
3381             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3382         spa = spa_add(pool, NULL, altroot);
3383         spa_activate(spa, spa_mode_global);

3385         if (props && (error = spa_prop_validate(spa, props))) {
3386                 spa_deactivate(spa);
3387                 spa_remove(spa);
3388                 mutex_exit(&spa_namespace_lock);
3389                 return (error);
3390         }

3392         has_features = B_FALSE;
3393         for (nvpair_t *elem = nvlist_next_nvpair(props, NULL);
3394             elem != NULL; elem = nvlist_next_nvpair(props, elem)) {
3395                 if (zpool_prop_feature(nvpair_name(elem)))
3396                         has_features = B_TRUE;
3397         }

3399         if (has_features || nvlist_lookup_uint64(props,
3400             zpool_prop_to_name(ZPOOL_PROP_VERSION), &version) != 0) {
3401                 version = SPA_VERSION;
3402         }
3403         ASSERT(SPA_VERSION_IS_SUPPORTED(version));

3405         spa->spa_first_txg = txg;
3406         spa->spa_uberblock.ub_txg = txg - 1;
3407         spa->spa_uberblock.ub_version = version;
3408         spa->spa_ubsync = spa->spa_uberblock;

3410         /*
3411          * Create "The Godfather" zio to hold all async IOs
3412          */
3413         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
3414             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

3416         /*
3417          * Create the root vdev.
3418          */
3419         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

3421         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);

3423         ASSERT(error != 0 || rvd != NULL);
```

```
3424         ASSERT(error != 0 || spa->spa_root_vdev == rvd);

3426         if (error == 0 && !zfs_allocatable_devs(nvroot))
3427                 error = SET_ERROR(EINVAL);

3429         if (error == 0 &&
3430             (error = vdev_create(rvd, txg, B_FALSE)) == 0 &&
3431             (error = spa_validate_aux(spa, nvroot, txg,
3432             VDEV_ALLOC_ADD)) == 0) {
3433                 for (int c = 0; c < rvd->vdev_children; c++) {
3434                         vdev_metaslab_set_size(rvd->vdev_child[c]);
3435                         vdev_expand(rvd->vdev_child[c], txg);
3436                 }
3437         }

3439         spa_config_exit(spa, SCL_ALL, FTAG);

3441         if (error != 0) {
3442                 spa_unload(spa);
3443                 spa_deactivate(spa);
3444                 spa_remove(spa);
3445                 mutex_exit(&spa_namespace_lock);
3446                 return (error);
3447         }

3449         /*
3450          * Get the list of spares, if specified.
3451          */
3452         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3453             &spares, &nspares) == 0) {
3454                 VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME,
3455                     KM_SLEEP) == 0);
3456                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3457                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3458                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3459                 spa_load_spares(spa);
3460                 spa_config_exit(spa, SCL_ALL, FTAG);
3461                 spa->spa_spares.sav_sync = B_TRUE;
3462         }

3464         /*
3465          * Get the list of level 2 cache devices, if specified.
3466          */
3467         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3468             &l2cache, &nl2cache) == 0) {
3469                 VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3470                     NV_UNIQUE_NAME, KM_SLEEP) == 0);
3471                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3472                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3473                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3474                 spa_load_l2cache(spa);
3475                 spa_config_exit(spa, SCL_ALL, FTAG);
3476                 spa->spa_l2cache.sav_sync = B_TRUE;
3477         }

3479         spa->spa_is_initializing = B_TRUE;
3480         spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
3481         spa->spa_meta_objset = dp->dp_meta_objset;
3482         spa->spa_is_initializing = B_FALSE;

3484         /*
3485          * Create DDTs (dedup tables).
3486          */
3487         ddt_create(spa);

3489         spa_update_dspace(spa);
```



```
3491         tx = dmu_tx_create_assigned(dp, txg);

3493         /*
3494          * Create the pool config object.
3495          */
3496         spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset,
3497             DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE,
3498             DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);

3500         if (zap_add(spa->spa_meta_objset,
3501             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CONFIG,
3502             sizeof (uint64_t), 1, &spa->spa_config_object, tx) != 0) {
3503                 cmn_err(CE_PANIC, "failed to add pool config");
3504         }

3506         if (spa_version(spa) >= SPA_VERSION_FEATURES)
3507                 spa_feature_create_zap_objects(spa, tx);

3509         if (zap_add(spa->spa_meta_objset,
3510             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CREATION_VERSION,
3511             sizeof (uint64_t), 1, &version, tx) != 0) {
3512                 cmn_err(CE_PANIC, "failed to add pool version");
3513         }

3515         /* Newly created pools with the right version are always deflated. */
3516         if (version >= SPA_VERSION_RAIDZ_DEFLATE) {
3517                 spa->spa_deflate = TRUE;
3518                 if (zap_add(spa->spa_meta_objset,
3519                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
3520                     sizeof (uint64_t), 1, &spa->spa_deflate, tx) != 0) {
3521                         cmn_err(CE_PANIC, "failed to add deflate");
3522                 }
3523         }

3525         /*
3526          * Create the deferred-free bpobj.  Turn off compression
3527          * because sync-to-convergence takes longer if the blocksize
3528          * keeps changing.
3529          */
3530         obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
3531         dmu_object_set_compress(spa->spa_meta_objset, obj,
3532             ZIO_COMPRESS_OFF, tx);
3533         if (zap_add(spa->spa_meta_objset,
3534             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ,
3535             sizeof (uint64_t), 1, &obj, tx) != 0) {
3536                 cmn_err(CE_PANIC, "failed to add bpobj");
3537         }
3538         VERIFY3U(0, ==, bpobj_open(&spa->spa_deferred_bpobj,
3539             spa->spa_meta_objset, obj));

3541         /*
3542          * Create the pool's history object.
3543          */
3544         if (version >= SPA_VERSION_ZPOOL_HISTORY)
3545                 spa_history_create_obj(spa, tx);

3547         /*
3548          * Set pool properties.
3549          */
3550         spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
3551         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
3552         spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
3553         spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);

3555         if (props != NULL) {
```

```
3556                 spa_configfile_set(spa, props, B_FALSE);
3557                 spa_sync_props(props, tx);
3558         }

3560         dmu_tx_commit(tx);

3562         spa->spa_sync_on = B_TRUE;
3563         txg_sync_start(spa->spa_dsl_pool);

3565         /*
3566          * We explicitly wait for the first transaction to complete so that our
3567          * bean counters are appropriately updated.
3568          */
3569         txg_wait_synced(spa->spa_dsl_pool, txg);

3571         spa_config_sync(spa, B_FALSE, B_TRUE);

3573         spa_history_log_version(spa, "create");

3575         spa->spa_minref = refcount_count(&spa->spa_refcount);

3577         mutex_exit(&spa_namespace_lock);

3579         return (0);
3580 }

3582 #ifdef _KERNEL
3583 /*
3584  * Get the root pool information from the root disk, then import the root pool
3585  * during the system boot up time.
3586  */
3587 extern int vdev_disk_read_rootlabel(char *, char *, nvlist_t **);

3589 static nvlist_t *
3590 spa_generate_rootconf(char *devpath, char *devid, uint64_t *guid)
3591 {
3592         nvlist_t *config;
3593         nvlist_t *nvtop, *nvroot;
3594         uint64_t pgid;

3596         if (vdev_disk_read_rootlabel(devpath, devid, &config) != 0)
3597                 return (NULL);

3599         /*
3600          * Add this top-level vdev to the child array.
3601          */
3602         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3603             &nvtop) == 0);
3604         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
3605             &pgid) == 0);
3606         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, guid) == 0);

3608         /*
3609          * Put this pool's top-level vdevs into a root vdev.
3610          */
3611         VERIFY(nvlist_alloc(&nvroot, NV_UNIQUE_NAME, KM_SLEEP) == 0);
3612         VERIFY(nvlist_add_string(nvroot, ZPOOL_CONFIG_TYPE,
3613             VDEV_TYPE_ROOT) == 0);
3614         VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_ID, 0ULL) == 0);
3615         VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_GUID, pgid) == 0);
3616         VERIFY(nvlist_add_nvlist_array(nvroot, ZPOOL_CONFIG_CHILDREN,
3617             &nvtop, 1) == 0);

3619         /*
3620          * Replace the existing vdev_tree with the new root vdev in
3621          * this pool's configuration (remove the old, add the new).
```



```
3622          */
3623         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, nvroot) == 0);
3624         nvlist_free(nvroot);
3625         return (config);
3626 }

3628 /*
3629  * Walk the vdev tree and see if we can find a device with "better"
3630  * configuration. A configuration is "better" if the label on that
3631  * device has a more recent txg.
3632  */
3633 static void
3634 spa_alt_rootvdev(vdev_t *vd, vdev_t **avd, uint64_t *txg)
3635 {
3636         for (int c = 0; c < vd->vdev_children; c++)
3637                 spa_alt_rootvdev(vd->vdev_child[c], avd, txg);

3639         if (vd->vdev_ops->vdev_op_leaf) {
3640                 nvlist_t *label;
3641                 uint64_t label_txg;

3643                 if (vdev_disk_read_rootlabel(vd->vdev_physpath, vd->vdev_devid,
3644                     &label) != 0)
3645                         return;

3647                 VERIFY(nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_TXG,
3648                     &label_txg) == 0);

3650                 /*
3651                  * Do we have a better boot device?
3652                  */
3653                 if (label_txg > *txg) {
3654                         *txg = label_txg;
3655                         *avd = vd;
3656                 }
3657                 nvlist_free(label);
3658         }
3659 }

3661 /*
3662  * Import a root pool.
3663  *
3664  * For x86. devpath_list will consist of devid and/or physpath name of
3665  * the vdev (e.g. "id1,sd@SSEAGATE..." or "/pci@1f,0/ide@d/disk@0,0:a").
3666  * The GRUB "findroot" command will return the vdev we should boot.
3667  *
3668  * For Sparc, devpath_list consists the physpath name of the booting device
3669  * no matter the rootpool is a single device pool or a mirrored pool.
3670  * e.g.
3671  *      "/pci@1f,0/ide@d/disk@0,0:a"
3672  */
3673 int
3674 spa_import_rootpool(char *devpath, char *devid)
3675 {
3676         spa_t *spa;
3677         vdev_t *rvd, *bvd, *avd = NULL;
3678         nvlist_t *config, *nvtop;
3679         uint64_t guid, txg;
3680         char *pname;
3681         int error;

3683         /*
3684          * Read the label from the boot device and generate a configuration.
3685          */
3686         config = spa_generate_rootconf(devpath, devid, &guid);
3687 #if defined(_OBP) && defined(_KERNEL)
```

```
3688         if (config == NULL) {
3689                 if (strstr(devpath, "/iscsi/ssd") != NULL) {
3690                         /* iscsi boot */
3691                         get_iscsi_bootpath_phy(devpath);
3692                         config = spa_generate_rootconf(devpath, devid, &guid);
3693                 }
3694         }
3695 #endif
3696         if (config == NULL) {
3697                 cmn_err(CE_NOTE, "Cannot read the pool label from '%s'",
3698                     devpath);
3699                 return (SET_ERROR(EIO));
3700         }

3702         VERIFY(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
3703             &pname) == 0);
3704         VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG, &txg) == 0);

3706         mutex_enter(&spa_namespace_lock);
3707         if ((spa = spa_lookup(pname)) != NULL) {
3708                 /*
3709                  * Remove the existing root pool from the namespace so that we
3710                  * can replace it with the correct config we just read in.
3711                  */
3712                 spa_remove(spa);
3713         }

3715         spa = spa_add(pname, config, NULL);
3716         spa->spa_is_root = B_TRUE;
3717         spa->spa_import_flags = ZFS_IMPORT_VERBATIM;

3719         /*
3720          * Build up a vdev tree based on the boot device's label config.
3721          */
3722         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3723             &nvtop) == 0);
3724         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3725         error = spa_config_parse(spa, &rvd, nvtop, NULL, 0,
3726             VDEV_ALLOC_ROOTPOOL);
3727         spa_config_exit(spa, SCL_ALL, FTAG);
3728         if (error) {
3729                 mutex_exit(&spa_namespace_lock);
3730                 nvlist_free(config);
3731                 cmn_err(CE_NOTE, "Can not parse the config for pool '%s'",
3732                     pname);
3733                 return (error);
3734         }

3736         /*
3737          * Get the boot vdev.
3738          */
3739         if ((bvd = vdev_lookup_by_guid(rvd, guid)) == NULL) {
3740                 cmn_err(CE_NOTE, "Can not find the boot vdev for guid %llu",
3741                     (u_longlong_t)guid);
3742                 error = SET_ERROR(ENOENT);
3743                 goto out;
3744         }

3746         /*
3747          * Determine if there is a better boot device.
3748          */
3749         avd = bvd;
3750         spa_alt_rootvdev(rvd, &avd, &txg);
3751         if (avd != bvd) {
3752                 cmn_err(CE_NOTE, "The boot device is 'degraded'. Please "
3753                     "try booting from '%s'", avd->vdev_path);
```



```
3754                 error = SET_ERROR(EINVAL);
3755                 goto out;
3756         }

3758         /*
3759          * If the boot device is part of a spare vdev then ensure that
3760          * we're booting off the active spare.
3761          */
3762         if (bvd->vdev_parent->vdev_ops == &vdev_spare_ops &&
3763             !bvd->vdev_isspare) {
3764                 cmn_err(CE_NOTE, "The boot device is currently spared. Please "
3765                     "try booting from '%s'",
3766                     bvd->vdev_parent->
3767                     vdev_child[bvd->vdev_parent->vdev_children - 1]->vdev_path);
3768                 error = SET_ERROR(EINVAL);
3769                 goto out;
3770         }

3772         error = 0;
3773 out:
3774         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3775         vdev_free(rvd);
3776         spa_config_exit(spa, SCL_ALL, FTAG);
3777         mutex_exit(&spa_namespace_lock);

3779         nvlist_free(config);
3780         return (error);
3781 }

3783 #endif

3785 /*
3786  * Import a non-root pool into the system.
3787  */
3788 int
3789 spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)
3790 {
3791         spa_t *spa;
3792         char *altroot = NULL;
3793         spa_load_state_t state = SPA_LOAD_IMPORT;
3794         zpool_rewind_policy_t policy;
3795         uint64_t mode = spa_mode_global;
3796         uint64_t readonly = B_FALSE;
3797         int error;
3798         nvlist_t *nvroot;
3799         nvlist_t **spares, **l2cache;
3800         uint_t nspares, nl2cache;

3802         /*
3803          * If a pool with this name exists, return failure.
3804          */
3805         mutex_enter(&spa_namespace_lock);
3806         if (spa_lookup(pool) != NULL) {
3807                 mutex_exit(&spa_namespace_lock);
3808                 return (SET_ERROR(EEXIST));
3809         }

3811         /*
3812          * Create and initialize the spa structure.
3813          */
3814         (void) nvlist_lookup_string(props,
3815             zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3816         (void) nvlist_lookup_uint64(props,
3817             zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);
3818         if (readonly)
3819                 mode = FREAD;
```

```
3820         spa = spa_add(pool, config, altroot);
3821         spa->spa_import_flags = flags;

3823         /*
3824          * Verbatim import - Take a pool and insert it into the namespace
3825          * as if it had been loaded at boot.
3826          */
3827         if (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) {
3828                 if (props != NULL)
3829                         spa_configfile_set(spa, props, B_FALSE);

3831                 spa_config_sync(spa, B_FALSE, B_TRUE);

3833                 mutex_exit(&spa_namespace_lock);
3834                 spa_history_log_version(spa, "import");

3836                 return (0);
3837         }

3839         spa_activate(spa, mode);

3841         /*
3842          * Don't start async tasks until we know everything is healthy.
3843          */
3844         spa_async_suspend(spa);

3846         zpool_get_rewind_policy(config, &policy);
3847         if (policy.zrp_request & ZPOOL_DO_REWIND)
3848                 state = SPA_LOAD_RECOVER;

3850         /*
3851          * Pass off the heavy lifting to spa_load().  Pass TRUE for mosconfig
3852          * because the user-supplied config is actually the one to trust when
3853          * doing an import.
3854          */
3855         if (state != SPA_LOAD_RECOVER)
3856                 spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;

3858         error = spa_load_best(spa, state, B_TRUE, policy.zrp_txg,
3859             policy.zrp_request);

3861         /*
3862          * Propagate anything learned while loading the pool and pass it
3863          * back to caller (i.e. rewind info, missing devices, etc).
3864          */
3865         VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
3866             spa->spa_load_info) == 0);

3868         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3869         /*
3870          * Toss any existing sparelist, as it doesn't have any validity
3871          * anymore, and conflicts with spa_has_spare().
3872          */
3873         if (spa->spa_spares.sav_config) {
3874                 nvlist_free(spa->spa_spares.sav_config);
3875                 spa->spa_spares.sav_config = NULL;
3876                 spa_load_spares(spa);
3877         }
3878         if (spa->spa_l2cache.sav_config) {
3879                 nvlist_free(spa->spa_l2cache.sav_config);
3880                 spa->spa_l2cache.sav_config = NULL;
3881                 spa_load_l2cache(spa);
3882         }

3884         VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE,
3885             &nvroot) == 0);
```



```
3886         if (error == 0)
3887                 error = spa_validate_aux(spa, nvroot, -1ULL,
3888                     VDEV_ALLOC_SPARE);
3889         if (error == 0)
3890                 error = spa_validate_aux(spa, nvroot, -1ULL,
3891                     VDEV_ALLOC_L2CACHE);
3892         spa_config_exit(spa, SCL_ALL, FTAG);

3894         if (props != NULL)
3895                 spa_configfile_set(spa, props, B_FALSE);

3897         if (error != 0 || (props && spa_writeable(spa) &&
3898             (error = spa_prop_set(spa, props)))) {
3899                 spa_unload(spa);
3900                 spa_deactivate(spa);
3901                 spa_remove(spa);
3902                 mutex_exit(&spa_namespace_lock);
3903                 return (error);
3904         }

3906         spa_async_resume(spa);

3908         /*
3909          * Override any spares and level 2 cache devices as specified by
3910          * the user, as these may have correct device names/devids, etc.
3911          */
3912         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3913             &spares, &nspares) == 0) {
3914                 if (spa->spa_spares.sav_config)
3915                         VERIFY(nvlist_remove(spa->spa_spares.sav_config,
3916                             ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
3917                 else
3918                         VERIFY(nvlist_alloc(&spa->spa_spares.sav_config,
3919                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3920                 VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3921                     ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3922                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3923                 spa_load_spares(spa);
3924                 spa_config_exit(spa, SCL_ALL, FTAG);
3925                 spa->spa_spares.sav_sync = B_TRUE;
3926         }
3927         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3928             &l2cache, &nl2cache) == 0) {
3929                 if (spa->spa_l2cache.sav_config)
3930                         VERIFY(nvlist_remove(spa->spa_l2cache.sav_config,
3931                             ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
3932                 else
3933                         VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3934                             NV_UNIQUE_NAME, KM_SLEEP) == 0);
3935                 VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3936                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3937                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3938                 spa_load_l2cache(spa);
3939                 spa_config_exit(spa, SCL_ALL, FTAG);
3940                 spa->spa_l2cache.sav_sync = B_TRUE;
3941         }

3943         /*
3944          * Check for any removed devices.
3945          */
3946         if (spa->spa_autoreplace) {
3947                 spa_aux_check_removed(&spa->spa_spares);
3948                 spa_aux_check_removed(&spa->spa_l2cache);
3949         }

3951         if (spa_writeable(spa)) {
```

```
3952                 /*
3953                  * Update the config cache to include the newly-imported pool.
3954                  */
3955                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
3956         }
3957
3958         /*
3959          * It's possible that the pool was expanded while it was exported.
3960          * We kick off an async task to handle this for us.
3961          */
3962         spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);
3963
3964         mutex_exit(&spa_namespace_lock);
3965         spa_history_log_version(spa, "import");
3966
3967         return (0);
3968 }
3969
3970 nvlist_t *
3971 spa_tryimport(nvlist_t *tryconfig)
3972 {
3973         nvlist_t *config = NULL;
3974         char *poolname;
3975         spa_t *spa;
3976         uint64_t state;
3977         int error;
3978
3979         if (nvlist_lookup_string(tryconfig, ZPOOL_CONFIG_POOL_NAME, &poolname))
3980                 return (NULL);
3981
3982         if (nvlist_lookup_uint64(tryconfig, ZPOOL_CONFIG_POOL_STATE, &state))
3983                 return (NULL);
3984
3985         /*
3986          * Create and initialize the spa structure.
3987          */
3988         mutex_enter(&spa_namespace_lock);
3989         spa = spa_add(TRYIMPORT_NAME, tryconfig, NULL);
3990         spa_activate(spa, FREAD);
3991
3992         /*
3993          * Pass off the heavy lifting to spa_load().
3994          * Pass TRUE for mosconfig because the user-supplied config
3995          * is actually the one to trust when doing an import.
3996          */
3997         error = spa_load(spa, SPA_LOAD_TRYIMPORT, SPA_IMPORT_EXISTING, B_TRUE);
3998
3999         /*
4000          * If 'tryconfig' was at least parsable, return the current config.
4001          */
4002         if (spa->spa_root_vdev != NULL) {
4003                 config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
4004                 VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME,
4005                     poolname) == 0);
4006                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4007                     state) == 0);
4008                 VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_TIMESTAMP,
4009                     spa->spa_uberblock.ub_timestamp) == 0);
4010                 VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO,
4011                     spa->spa_load_info) == 0);
4012
4013                 /*
4014                  * If the bootfs property exists on this pool then we
4015                  * copy it out so that external consumers can tell which
4016                  * pools are bootable.
4017                  */
```



```
4018                 if ((!error || error == EEXIST) && spa->spa_bootfs) {
4019                         char *tmpname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
4020
4021                         /*
4022                          * We have to play games with the name since the
4023                          * pool was opened as TRYIMPORT_NAME.
4024                          */
4025                         if (dsl_dsobj_to_dsname(spa_name(spa),
4026                             spa->spa_bootfs, tmpname) == 0) {
4027                                 char *cp;
4028                                 char *dsname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
4029
4030                                 cp = strchr(tmpname, '/');
4031                                 if (cp == NULL) {
4032                                         (void) strlcpy(dsname, tmpname,
4033                                             MAXPATHLEN);
4034                                 } else {
4035                                         (void) snprintf(dsname, MAXPATHLEN,
4036                                             "%s/%s", poolname, ++cp);
4037                                 }
4038                                 VERIFY(nvlist_add_string(config,
4039                                     ZPOOL_CONFIG_BOOTFS, dsname) == 0);
4040                                 kmem_free(dsname, MAXPATHLEN);
4041                         }
4042                         kmem_free(tmpname, MAXPATHLEN);
4043                 }
4044
4045                 /*
4046                  * Add the list of hot spares and level 2 cache devices.
4047                  */
4048                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
4049                 spa_add_spares(spa, config);
4050                 spa_add_l2cache(spa, config);
4051                 spa_config_exit(spa, SCL_CONFIG, FTAG);
4052         }
4053
4054         spa_unload(spa);
4055         spa_deactivate(spa);
4056         spa_remove(spa);
4057         mutex_exit(&spa_namespace_lock);
4058
4059         return (config);
4060 }
4061
4062 /*
4063  * Pool export/destroy
4064  *
4065  * The act of destroying or exporting a pool is very simple.  We make sure there
4066  * is no more pending I/O and any references to the pool are gone.  Then, we
4067  * update the pool state and sync all the labels to disk, removing the
4068  * configuration from the cache afterwards. If the 'hardforce' flag is set, then
4069  * we don't sync the labels or remove the configuration cache.
4070  */
4071 static int
4072 spa_export_common(char *pool, int new_state, nvlist_t **oldconfig,
4073     boolean_t force, boolean_t hardforce)
4074 {
4075         spa_t *spa;
4076
4077         if (oldconfig)
4078                 *oldconfig = NULL;
4079
4080         if (!(spa_mode_global & FWRITE))
4081                 return (SET_ERROR(EROFS));
4082
4083         mutex_enter(&spa_namespace_lock);
```

```
4084         if ((spa = spa_lookup(pool)) == NULL) {
4085                 mutex_exit(&spa_namespace_lock);
4086                 return (SET_ERROR(ENOENT));
4087         }
4089         /*
4090          * Put a hold on the pool, drop the namespace lock, stop async tasks,
4091          * reacquire the namespace lock, and see if we can export.
4092          */
4093         spa_open_ref(spa, FTAG);
4094         mutex_exit(&spa_namespace_lock);
4095         spa_async_suspend(spa);
4096         mutex_enter(&spa_namespace_lock);
4097         spa_close(spa, FTAG);
4099         /*
4100          * The pool will be in core if it's openable,
4101          * in which case we can modify its state.
4102          */
4103         if (spa->spa_state != POOL_STATE_UNINITIALIZED && spa->spa_sync_on) {
4104                 /*
4105                  * Objsets may be open only because they're dirty, so we
4106                  * have to force it to sync before checking spa_refcnt.
4107                  */
4108                 txg_wait_synced(spa->spa_dsl_pool, 0);
4110                 /*
4111                  * A pool cannot be exported or destroyed if there are active
4112                  * references.  If we are resetting a pool, allow references by
4113                  * fault injection handlers.
4114                  */
4115                 if (!spa_refcount_zero(spa) ||
4116                     (spa->spa_inject_ref != 0 &&
4117                     new_state != POOL_STATE_UNINITIALIZED)) {
4118                         spa_async_resume(spa);
4119                         mutex_exit(&spa_namespace_lock);
4120                         return (SET_ERROR(EBUSY));
4121                 }
4123                 /*
4124                  * A pool cannot be exported if it has an active shared spare.
4125                  * This is to prevent other pools stealing the active spare
4126                  * from an exported pool. At user's own will, such pool can
4127                  * be forcedly exported.
4128                  */
4129                 if (!force && new_state == POOL_STATE_EXPORTED &&
4130                     spa_has_active_shared_spare(spa)) {
4131                         spa_async_resume(spa);
4132                         mutex_exit(&spa_namespace_lock);
4133                         return (SET_ERROR(EXDEV));
4134                 }
4136                 /*
4137                  * We want this to be reflected on every label,
4138                  * so mark them all dirty.  spa_unload() will do the
4139                  * final sync that pushes these changes out.
4140                  */
4141                 if (new_state != POOL_STATE_UNINITIALIZED && !hardforce) {
4142                         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
4143                         spa->spa_state = new_state;
4144                         spa->spa_final_txg = spa_last_synced_txg(spa) +
4145                             TXG_DEFER_SIZE + 1;
4146                         vdev_config_dirty(spa->spa_root_vdev);
4147                         spa_config_exit(spa, SCL_ALL, FTAG);
4148                 }
4149         }
```



```
4151         spa_event_notify(spa, NULL, ESC_ZFS_POOL_DESTROY);
4153         if (spa->spa_state != POOL_STATE_UNINITIALIZED) {
4154                 spa_unload(spa);
4155                 spa_deactivate(spa);
4156         }
4158         if (oldconfig && spa->spa_config)
4159                 VERIFY(nvlist_dup(spa->spa_config, oldconfig, 0) == 0);
4161         if (new_state != POOL_STATE_UNINITIALIZED) {
4162                 if (!hardforce)
4163                         spa_config_sync(spa, B_TRUE, B_TRUE);
4164                 spa_remove(spa);
4165         }
4166         mutex_exit(&spa_namespace_lock);
4168         return (0);
4169 }
4171 /*
4172  * Destroy a storage pool.
4173  */
4174 int
4175 spa_destroy(char *pool)
4176 {
4177         return (spa_export_common(pool, POOL_STATE_DESTROYED, NULL,
4178             B_FALSE, B_FALSE));
4179 }
4181 /*
4182  * Export a storage pool.
4183  */
4184 int
4185 spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
4186     boolean_t hardforce)
4187 {
4188         return (spa_export_common(pool, POOL_STATE_EXPORTED, oldconfig,
4189             force, hardforce));
4190 }
4192 /*
4193  * Similar to spa_export(), this unloads the spa_t without actually removing it
4194  * from the namespace in any way.
4195  */
4196 int
4197 spa_reset(char *pool)
4198 {
4199         return (spa_export_common(pool, POOL_STATE_UNINITIALIZED, NULL,
4200             B_FALSE, B_FALSE));
4201 }
4203 /*
4204  * ========================================================================
4205  * Device manipulation
4206  * ========================================================================
4207  */
4209 /*
4210  * Add a device to a storage pool.
4211  */
4212 int
4213 spa_vdev_add(spa_t *spa, nvlist_t *nvroot)
4214 {
4215         uint64_t txg, id;
```

```
4216         int error;
4217         vdev_t *rvd = spa->spa_root_vdev;
4218         vdev_t *vd, *tvd;
4219         nvlist_t **spares, **l2cache;
4220         uint_t nspares, nl2cache;

4222         ASSERT(spa_writeable(spa));

4224         txg = spa_vdev_enter(spa);

4226         if ((error = spa_config_parse(spa, &vd, nvroot, NULL, 0,
4227             VDEV_ALLOC_ADD)) != 0)
4228                 return (spa_vdev_exit(spa, NULL, txg, error));

4230         spa->spa_pending_vdev = vd;     /* spa_vdev_exit() will clear this */

4232         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES, &spares,
4233             &nspares) != 0)
4234                 nspares = 0;

4236         if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE, &l2cache,
4237             &nl2cache) != 0)
4238                 nl2cache = 0;

4240         if (vd->vdev_children == 0 && nspares == 0 && nl2cache == 0)
4241                 return (spa_vdev_exit(spa, vd, txg, EINVAL));

4243         if (vd->vdev_children != 0 &&
4244             (error = vdev_create(vd, txg, B_FALSE)) != 0)
4245                 return (spa_vdev_exit(spa, vd, txg, error));

4247         /*
4248          * We must validate the spares and l2cache devices after checking the
4249          * children.  Otherwise, vdev_inuse() will blindly overwrite the spare.
4250          */
4251         if ((error = spa_validate_aux(spa, nvroot, txg, VDEV_ALLOC_ADD)) != 0)
4252                 return (spa_vdev_exit(spa, vd, txg, error));

4254         /*
4255          * Transfer each new top-level vdev from vd to rvd.
4256          */
4257         for (int c = 0; c < vd->vdev_children; c++) {

4259                 /*
4260                  * Set the vdev id to the first hole, if one exists.
4261                  */
4262                 for (id = 0; id < rvd->vdev_children; id++) {
4263                         if (rvd->vdev_child[id]->vdev_ishole) {
4264                                 vdev_free(rvd->vdev_child[id]);
4265                                 break;
4266                         }
4267                 }
4268                 tvd = vd->vdev_child[c];
4269                 vdev_remove_child(vd, tvd);
4270                 tvd->vdev_id = id;
4271                 vdev_add_child(rvd, tvd);
4272                 vdev_config_dirty(tvd);
4273         }

4275         if (nspares != 0) {
4276                 spa_set_aux_vdevs(&spa->spa_spares, spares, nspares,
4277                     ZPOOL_CONFIG_SPARES);
4278                 spa_load_spares(spa);
4279                 spa->spa_spares.sav_sync = B_TRUE;
4280         }
```



```
4282         if (nl2cache != 0) {
4283                 spa_set_aux_vdevs(&spa->spa_l2cache, l2cache, nl2cache,
4284                     ZPOOL_CONFIG_L2CACHE);
4285                 spa_load_l2cache(spa);
4286                 spa->spa_l2cache.sav_sync = B_TRUE;
4287         }

4289         /*
4290          * We have to be careful when adding new vdevs to an existing pool.
4291          * If other threads start allocating from these vdevs before we
4292          * sync the config cache, and we lose power, then upon reboot we may
4293          * fail to open the pool because there are DVAs that the config cache
4294          * can't translate.  Therefore, we first add the vdevs without
4295          * initializing metaslabs; sync the config cache (via spa_vdev_exit());
4296          * and then let spa_config_update() initialize the new metaslabs.
4297          *
4298          * spa_load() checks for added-but-not-initialized vdevs, so that
4299          * if we lose power at any point in this sequence, the remaining
4300          * steps will be completed the next time we load the pool.
4301          */
4302         (void) spa_vdev_exit(spa, vd, txg, 0);

4304         mutex_enter(&spa_namespace_lock);
4305         spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
4306         mutex_exit(&spa_namespace_lock);

4308         return (0);
4309 }

4311 /*
4312  * Attach a device to a mirror.  The arguments are the path to any device
4313  * in the mirror, and the nvroot for the new device.  If the path specifies
4314  * a device that is not mirrored, we automatically insert the mirror vdev.
4315  *
4316  * If 'replacing' is specified, the new device is intended to replace the
4317  * existing device; in this case the two devices are made into their own
4318  * mirror using the 'replacing' vdev, which is functionally identical to
4319  * the mirror vdev (it actually reuses all the same ops) but has a few
4320  * extra rules: you can't attach to it after it's been created, and upon
4321  * completion of resilvering, the first disk (the one being replaced)
4322  * is automatically detached.
4323  */
4324 int
4325 spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot, int replacing)
4326 {
4327         uint64_t txg, dtl_max_txg;
4328         vdev_t *rvd = spa->spa_root_vdev;
4329         vdev_t *oldvd, *newvd, *newrootvd, *pvd, *tvd;
4330         vdev_ops_t *pvops;
4331         char *oldvdpath, *newvdpath;
4332         int newvd_isspare;
4333         int error;

4335         ASSERT(spa_writeable(spa));

4337         txg = spa_vdev_enter(spa);

4339         oldvd = spa_lookup_by_guid(spa, guid, B_FALSE);

4341         if (oldvd == NULL)
4342                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4344         if (!oldvd->vdev_ops->vdev_op_leaf)
4345                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4347         pvd = oldvd->vdev_parent;
```

```
4349         if ((error = spa_config_parse(spa, &newrootvd, nvroot, NULL, 0,
4350             VDEV_ALLOC_ATTACH)) != 0)
4351                 return (spa_vdev_exit(spa, NULL, txg, EINVAL));
4352
4353         if (newrootvd->vdev_children != 1)
4354                 return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));
4355
4356         newvd = newrootvd->vdev_child[0];
4357
4358         if (!newvd->vdev_ops->vdev_op_leaf)
4359                 return (spa_vdev_exit(spa, newrootvd, txg, EINVAL));
4360
4361         if ((error = vdev_create(newrootvd, txg, replacing)) != 0)
4362                 return (spa_vdev_exit(spa, newrootvd, txg, error));
4363
4364         /*
4365          * Spares can't replace logs
4366          */
4367         if (oldvd->vdev_top->vdev_islog && newvd->vdev_isspare)
4368                 return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4369
4370         if (!replacing) {
4371                 /*
4372                  * For attach, the only allowable parent is a mirror or the root
4373                  * vdev.
4374                  */
4375                 if (pvd->vdev_ops != &vdev_mirror_ops &&
4376                     pvd->vdev_ops != &vdev_root_ops)
4377                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4378
4379                 pvops = &vdev_mirror_ops;
4380         } else {
4381                 /*
4382                  * Active hot spares can only be replaced by inactive hot
4383                  * spares.
4384                  */
4385                 if (pvd->vdev_ops == &vdev_spare_ops &&
4386                     oldvd->vdev_isspare &&
4387                     !spa_has_spare(spa, newvd->vdev_guid))
4388                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4389
4390                 /*
4391                  * If the source is a hot spare, and the parent isn't already a
4392                  * spare, then we want to create a new hot spare.  Otherwise, we
4393                  * want to create a replacing vdev.  The user is not allowed to
4394                  * attach to a spared vdev child unless the 'isspare' state is
4395                  * the same (spare replaces spare, non-spare replaces
4396                  * non-spare).
4397                  */
4398                 if (pvd->vdev_ops == &vdev_replacing_ops &&
4399                     spa_version(spa) < SPA_VERSION_MULTI_REPLACE) {
4400                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4401                 } else if (pvd->vdev_ops == &vdev_spare_ops &&
4402                     newvd->vdev_isspare != oldvd->vdev_isspare) {
4403                         return (spa_vdev_exit(spa, newrootvd, txg, ENOTSUP));
4404                 }
4405
4406                 if (newvd->vdev_isspare)
4407                         pvops = &vdev_spare_ops;
4408                 else
4409                         pvops = &vdev_replacing_ops;
4410         }
4411
4412         /*
4413          * Make sure the new device is big enough.
```



```
4414          */
4415         if (newvd->vdev_asize < vdev_get_min_asize(oldvd))
4416                 return (spa_vdev_exit(spa, newrootvd, txg, EOVERFLOW));
4417
4418         /*
4419          * The new device cannot have a higher alignment requirement
4420          * than the top-level vdev.
4421          */
4422         if (newvd->vdev_ashift > oldvd->vdev_top->vdev_ashift)
4423                 return (spa_vdev_exit(spa, newrootvd, txg, EDOM));
4424
4425         /*
4426          * If this is an in-place replacement, update oldvd's path and devid
4427          * to make it distinguishable from newvd, and unopenable from now on.
4428          */
4429         if (strcmp(oldvd->vdev_path, newvd->vdev_path) == 0) {
4430                 spa_strfree(oldvd->vdev_path);
4431                 oldvd->vdev_path = kmem_alloc(strlen(newvd->vdev_path) + 5,
4432                     KM_SLEEP);
4433                 (void) sprintf(oldvd->vdev_path, "%s/%s",
4434                     newvd->vdev_path, "old");
4435                 if (oldvd->vdev_devid != NULL) {
4436                         spa_strfree(oldvd->vdev_devid);
4437                         oldvd->vdev_devid = NULL;
4438                 }
4439         }
4440
4441         /* mark the device being resilvered */
4442         newvd->vdev_resilvering = B_TRUE;
4443
4444         /*
4445          * If the parent is not a mirror, or if we're replacing, insert the new
4446          * mirror/replacing/spare vdev above oldvd.
4447          */
4448         if (pvd->vdev_ops != pvops)
4449                 pvd = vdev_add_parent(oldvd, pvops);
4450
4451         ASSERT(pvd->vdev_top->vdev_parent == rvd);
4452         ASSERT(pvd->vdev_ops == pvops);
4453         ASSERT(oldvd->vdev_parent == pvd);
4454
4455         /*
4456          * Extract the new device from its root and add it to pvd.
4457          */
4458         vdev_remove_child(newrootvd, newvd);
4459         newvd->vdev_id = pvd->vdev_children;
4460         newvd->vdev_crtxg = oldvd->vdev_crtxg;
4461         vdev_add_child(pvd, newvd);
4462
4463         tvd = newvd->vdev_top;
4464         ASSERT(pvd->vdev_top == tvd);
4465         ASSERT(tvd->vdev_parent == rvd);
4466
4467         vdev_config_dirty(tvd);
4468
4469         /*
4470          * Set newvd's DTL to [TXG_INITIAL, dtl_max_txg) so that we account
4471          * for any dmu_sync-ed blocks.  It will propagate upward when
4472          * spa_vdev_exit() calls vdev_dtl_reassess().
4473          */
4474         dtl_max_txg = txg + TXG_CONCURRENT_STATES;
4475
4476         vdev_dtl_dirty(newvd, DTL_MISSING, TXG_INITIAL,
4477             dtl_max_txg - TXG_INITIAL);
4478
4479         if (newvd->vdev_isspare) {
```

```
4480                 spa_spare_activate(newvd);
4481                 spa_event_notify(spa, newvd, ESC_ZFS_VDEV_SPARE);
4482         }

4484         oldvdpath = spa_strdup(oldvd->vdev_path);
4485         newvdpath = spa_strdup(newvd->vdev_path);
4486         newvd_isspare = newvd->vdev_isspare;

4488         /*
4489          * Mark newvd's DTL dirty in this txg.
4490          */
4491         vdev_dirty(tvd, VDD_DTL, newvd, txg);

4493         /*
4494          * Restart the resilver
4495          */
4496         dsl_resilver_restart(spa->spa_dsl_pool, dtl_max_txg);

4498         /*
4499          * Commit the config
4500          */
4501         (void) spa_vdev_exit(spa, newrootvd, dtl_max_txg, 0);

4503         spa_history_log_internal(spa, "vdev attach", NULL,
4504             "%s vdev=%s %s vdev=%s",
4505             replacing && newvd_isspare ? "spare in" :
4506             replacing ? "replace" : "attach", newvdpath,
4507             replacing ? "for" : "to", oldvdpath);

4509         spa_strfree(oldvdpath);
4510         spa_strfree(newvdpath);

4512         if (spa->spa_bootfs)
4513                 spa_event_notify(spa, newvd, ESC_ZFS_BOOTFS_VDEV_ATTACH);

4515         return (0);
4516 }

4518 /*
4519  * Detach a device from a mirror or replacing vdev.
4520  * If 'replace_done' is specified, only detach if the parent
4521  * is a replacing vdev.
4522  */
4523 int
4524 spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
4525 {
4526         uint64_t txg;
4527         int error;
4528         vdev_t *rvd = spa->spa_root_vdev;
4529         vdev_t *vd, *pvd, *cvd, *tvd;
4530         boolean_t unspare = B_FALSE;
4531         uint64_t unspare_guid = 0;
4532         char *vdpath;

4534         ASSERT(spa_writeable(spa));

4536         txg = spa_vdev_enter(spa);

4538         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

4540         if (vd == NULL)
4541                 return (spa_vdev_exit(spa, NULL, txg, ENODEV));

4543         if (!vd->vdev_ops->vdev_op_leaf)
4544                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));
```



```
4546         pvd = vd->vdev_parent;

4548         /*
4549          * If the parent/child relationship is not as expected, don't do it.
4550          * Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
4551          * vdev that's replacing B with C.  The user's intent in replacing
4552          * is to go from M(A,B) to M(A,C).  If the user decides to cancel
4553          * the replace by detaching C, the expected behavior is to end up
4554          * M(A,B).  But suppose that right after deciding to detach C,
4555          * the replacement of B completes.  We would have M(A,C), and then
4556          * ask to detach C, which would leave us with just A -- not what
4557          * the user wanted.  To prevent this, we make sure that the
4558          * parent/child relationship hasn't changed -- in this example,
4559          * that C's parent is still the replacing vdev R.
4560          */
4561         if (pvd->vdev_guid != pguid && pguid != 0)
4562                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4564         /*
4565          * Only 'replacing' or 'spare' vdevs can be replaced.
4566          */
4567         if (replace_done && pvd->vdev_ops != &vdev_replacing_ops &&
4568             pvd->vdev_ops != &vdev_spare_ops)
4569                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4571         ASSERT(pvd->vdev_ops != &vdev_spare_ops ||
4572             spa_version(spa) >= SPA_VERSION_SPARES);

4574         /*
4575          * Only mirror, replacing, and spare vdevs support detach.
4576          */
4577         if (pvd->vdev_ops != &vdev_replacing_ops &&
4578             pvd->vdev_ops != &vdev_mirror_ops &&
4579             pvd->vdev_ops != &vdev_spare_ops)
4580                 return (spa_vdev_exit(spa, NULL, txg, ENOTSUP));

4582         /*
4583          * If this device has the only valid copy of some data,
4584          * we cannot safely detach it.
4585          */
4586         if (vdev_dtl_required(vd))
4587                 return (spa_vdev_exit(spa, NULL, txg, EBUSY));

4589         ASSERT(pvd->vdev_children >= 2);

4591         /*
4592          * If we are detaching the second disk from a replacing vdev, then
4593          * check to see if we changed the original vdev's path to have "/old"
4594          * at the end in spa_vdev_attach().  If so, undo that change now.
4595          */
4596         if (pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 &&
4597             vd->vdev_path != NULL) {
4598                 size_t len = strlen(vd->vdev_path);

4600                 for (int c = 0; c < pvd->vdev_children; c++) {
4601                         cvd = pvd->vdev_child[c];

4603                         if (cvd == vd || cvd->vdev_path == NULL)
4604                                 continue;

4606                         if (strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 &&
4607                             strcmp(cvd->vdev_path + len, "/old") == 0) {
4608                                 spa_strfree(cvd->vdev_path);
4609                                 cvd->vdev_path = spa_strdup(vd->vdev_path);
4610                                 break;
4611                         }
```

```
4612                }
4613        }

4615        /*
4616         * If we are detaching the original disk from a spare, then it implies
4617         * that the spare should become a real disk, and be removed from the
4618         * active spare list for the pool.
4619         */
4620        if (pvd->vdev_ops == &vdev_spare_ops &&
4621            vd->vdev_id == 0 &&
4622            pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare)
4623                unspare = B_TRUE;

4625        /*
4626         * Erase the disk labels so the disk can be used for other things.
4627         * This must be done after all other error cases are handled,
4628         * but before we disembowel vd (so we can still do I/O to it).
4629         * But if we can't do it, don't treat the error as fatal --
4630         * it may be that the unwritability of the disk is the reason
4631         * it's being detached!
4632         */
4633        error = vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

4635        /*
4636         * Remove vd from its parent and compact the parent's children.
4637         */
4638        vdev_remove_child(pvd, vd);
4639        vdev_compact_children(pvd);

4641        /*
4642         * Remember one of the remaining children so we can get tvd below.
4643         */
4644        cvd = pvd->vdev_child[pvd->vdev_children - 1];

4646        /*
4647         * If we need to remove the remaining child from the list of hot spares,
4648         * do it now, marking the vdev as no longer a spare in the process.
4649         * We must do this before vdev_remove_parent(), because that can
4650         * change the GUID if it creates a new toplevel GUID.  For a similar
4651         * reason, we must remove the spare now, in the same txg as the detach;
4652         * otherwise someone could attach a new sibling, change the GUID, and
4653         * the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
4654         */
4655        if (unspare) {
4656                ASSERT(cvd->vdev_isspare);
4657                spa_spare_remove(cvd);
4658                unspare_guid = cvd->vdev_guid;
4659                (void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
4660                cvd->vdev_unspare = B_TRUE;
4661        }

4663        /*
4664         * If the parent mirror/replacing vdev only has one child,
4665         * the parent is no longer needed.  Remove it from the tree.
4666         */
4667        if (pvd->vdev_children == 1) {
4668                if (pvd->vdev_ops == &vdev_spare_ops)
4669                        cvd->vdev_unspare = B_FALSE;
4670                vdev_remove_parent(cvd);
4671                cvd->vdev_resilvering = B_FALSE;
4672        }


4675        /*
4676         * We don't set tvd until now because the parent we just removed
4677         * may have been the previous top-level vdev.
```



```
4678         */
4679        tvd = cvd->vdev_top;
4680        ASSERT(tvd->vdev_parent == rvd);

4682        /*
4683         * Reevaluate the parent vdev state.
4684         */
4685        vdev_propagate_state(cvd);

4687        /*
4688         * If the 'autoexpand' property is set on the pool then automatically
4689         * try to expand the size of the pool. For example if the device we
4690         * just detached was smaller than the others, it may be possible to
4691         * add metaslabs (i.e. grow the pool). We need to reopen the vdev
4692         * first so that we can obtain the updated sizes of the leaf vdevs.
4693         */
4694        if (spa->spa_autoexpand) {
4695                vdev_reopen(tvd);
4696                vdev_expand(tvd, txg);
4697        }

4699        vdev_config_dirty(tvd);

4701        /*
4702         * Mark vd's DTL as dirty in this txg.  vdev_dtl_sync() will see that
4703         * vd->vdev_detached is set and free vd's DTL object in syncing context.
4704         * But first make sure we're not on any *other* txg's DTL list, to
4705         * prevent vd from being accessed after it's freed.
4706         */
4707        vdpath = spa_strdup(vd->vdev_path);
4708        for (int t = 0; t < TXG_SIZE; t++)
4709                (void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
4710        vd->vdev_detached = B_TRUE;
4711        vdev_dirty(tvd, VDD_DTL, vd, txg);

4713        spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);

4715        /* hang on to the spa before we release the lock */
4716        spa_open_ref(spa, FTAG);

4718        error = spa_vdev_exit(spa, vd, txg, 0);

4720        spa_history_log_internal(spa, "detach", NULL,
4721            "vdev=%s", vdpath);
4722        spa_strfree(vdpath);

4724        /*
4725         * If this was the removal of the original device in a hot spare vdev,
4726         * then we want to go through and remove the device from the hot spare
4727         * list of every other pool.
4728         */
4729        if (unspare) {
4730                spa_t *altspa = NULL;

4732                mutex_enter(&spa_namespace_lock);
4733                while ((altspa = spa_next(altspa)) != NULL) {
4734                        if (altspa->spa_state != POOL_STATE_ACTIVE ||
4735                            altspa == spa)
4736                                continue;

4738                        spa_open_ref(altspa, FTAG);
4739                        mutex_exit(&spa_namespace_lock);
4740                        (void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
4741                        mutex_enter(&spa_namespace_lock);
4742                        spa_close(altspa, FTAG);
4743                }
```

```
4744                mutex_exit(&spa_namespace_lock);

4746                /* search the rest of the vdevs for spares to remove */
4747                spa_vdev_resilver_done(spa);
4748        }

4750        /* all done with the spa; OK to release */
4751        mutex_enter(&spa_namespace_lock);
4752        spa_close(spa, FTAG);
4753        mutex_exit(&spa_namespace_lock);

4755        return (error);
4756 }

4758 /*
4759  * Split a set of devices from their mirrors, and create a new pool from them.
4760  */
4761 int
4762 spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
4763     nvlist_t *props, boolean_t exp)
4764 {
4765        int error = 0;
4766        uint64_t txg, *glist;
4767        spa_t *newspa;
4768        uint_t c, children, lastlog;
4769        nvlist_t **child, *nvl, *tmp;
4770        dmu_tx_t *tx;
4771        char *altroot = NULL;
4772        vdev_t *rvd, **vml = NULL;                      /* vdev modify list */
4773        boolean_t activate_slog;

4775        ASSERT(spa_writeable(spa));

4777        txg = spa_vdev_enter(spa);

4779        /* clear the log and flush everything up to now */
4780        activate_slog = spa_passivate_log(spa);
4781        (void) spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4782        error = spa_offline_log(spa);
4783        txg = spa_vdev_config_enter(spa);

4785        if (activate_slog)
4786                spa_activate_log(spa);

4788        if (error != 0)
4789                return (spa_vdev_exit(spa, NULL, txg, error));

4791        /* check new spa name before going any further */
4792        if (spa_lookup(newname) != NULL)
4793                return (spa_vdev_exit(spa, NULL, txg, EEXIST));

4795        /*
4796         * scan through all the children to ensure they're all mirrors
4797         */
4798        if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvl) != 0 ||
4799            nvlist_lookup_nvlist_array(nvl, ZPOOL_CONFIG_CHILDREN, &child,
4800            &children) != 0)
4801                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4803        /* first, check to ensure we've got the right child count */
4804        rvd = spa->spa_root_vdev;
4805        lastlog = 0;
4806        for (c = 0; c < rvd->vdev_children; c++) {
4807                vdev_t *vd = rvd->vdev_child[c];

4809                /* don't count the holes & logs as children */
```



```
4810                if (vd->vdev_islog || vd->vdev_ishole) {
4811                        if (lastlog == 0)
4812                                lastlog = c;
4813                        continue;
4814                }

4816                lastlog = 0;
4817        }
4818        if (children != (lastlog != 0 ? lastlog : rvd->vdev_children))
4819                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4821        /* next, ensure no spare or cache devices are part of the split */
4822        if (nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_SPARES, &tmp) == 0 ||
4823            nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_L2CACHE, &tmp) == 0)
4824                return (spa_vdev_exit(spa, NULL, txg, EINVAL));

4826        vml = kmem_zalloc(children * sizeof (vdev_t *), KM_SLEEP);
4827        glist = kmem_zalloc(children * sizeof (uint64_t), KM_SLEEP);

4829        /* then, loop over each vdev and validate it */
4830        for (c = 0; c < children; c++) {
4831                uint64_t is_hole = 0;

4833                (void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE,
4834                    &is_hole);

4836                if (is_hole != 0) {
4837                        if (spa->spa_root_vdev->vdev_child[c]->vdev_ishole ||
4838                            spa->spa_root_vdev->vdev_child[c]->vdev_islog) {
4839                                continue;
4840                        } else {
4841                                error = SET_ERROR(EINVAL);
4842                                break;
4843                        }
4844                }

4846                /* which disk is going to be split? */
4847                if (nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_GUID,
4848                    &glist[c]) != 0) {
4849                        error = SET_ERROR(EINVAL);
4850                        break;
4851                }

4853                /* look it up in the spa */
4854                vml[c] = spa_lookup_by_guid(spa, glist[c], B_FALSE);
4855                if (vml[c] == NULL) {
4856                        error = SET_ERROR(ENODEV);
4857                        break;
4858                }

4860                /* make sure there's nothing stopping the split */
4861                if (vml[c]->vdev_parent->vdev_ops != &vdev_mirror_ops ||
4862                    vml[c]->vdev_islog ||
4863                    vml[c]->vdev_ishole ||
4864                    vml[c]->vdev_isspare ||
4865                    vml[c]->vdev_isl2cache ||
4866                    !vdev_writeable(vml[c]) ||
4867                    vml[c]->vdev_children != 0 ||
4868                    vml[c]->vdev_state != VDEV_STATE_HEALTHY ||
4869                    c != spa->spa_root_vdev->vdev_child[c]->vdev_id) {
4870                        error = SET_ERROR(EINVAL);
4871                        break;
4872                }

4874                if (vdev_dtl_required(vml[c])) {
4875                        error = SET_ERROR(EBUSY);
```

```
4876                        break;
4877                }
4878
4879                /* we need certain info from the top level */
4880                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_ARRAY,
4881                    vml[c]->vdev_top->vdev_ms_array) == 0);
4882                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_SHIFT,
4883                    vml[c]->vdev_top->vdev_ms_shift) == 0);
4884                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASIZE,
4885                    vml[c]->vdev_top->vdev_asize) == 0);
4886                VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASHIFT,
4887                    vml[c]->vdev_top->vdev_ashift) == 0);
4888        }
4889
4890        if (error != 0) {
4891                kmem_free(vml, children * sizeof (vdev_t *));
4892                kmem_free(glist, children * sizeof (uint64_t));
4893                return (spa_vdev_exit(spa, NULL, txg, error));
4894        }
4895
4896        /* stop writers from using the disks */
4897        for (c = 0; c < children; c++) {
4898                if (vml[c] != NULL)
4899                        vml[c]->vdev_offline = B_TRUE;
4900        }
4901        vdev_reopen(spa->spa_root_vdev);
4902
4903        /*
4904         * Temporarily record the splitting vdevs in the spa config.  This
4905         * will disappear once the config is regenerated.
4906         */
4907        VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
4908        VERIFY(nvlist_add_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST,
4909            glist, children) == 0);
4910        kmem_free(glist, children * sizeof (uint64_t));
4911
4912        mutex_enter(&spa->spa_props_lock);
4913        VERIFY(nvlist_add_nvlist(spa->spa_config, ZPOOL_CONFIG_SPLIT,
4914            nvl) == 0);
4915        mutex_exit(&spa->spa_props_lock);
4916        spa->spa_config_splitting = nvl;
4917        vdev_config_dirty(spa->spa_root_vdev);
4918
4919        /* configure and create the new pool */
4920        VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, newname) == 0);
4921        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
4922            exp ? POOL_STATE_EXPORTED : POOL_STATE_ACTIVE) == 0);
4923        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
4924            spa_version(spa)) == 0);
4925        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG,
4926            spa->spa_config_txg) == 0);
4927        VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID,
4928            spa_generate_guid(NULL)) == 0);
4929        (void) nvlist_lookup_string(props,
4930            zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
4931
4932        /* add the new pool to the namespace */
4933        newspa = spa_add(newname, config, altroot);
4934        newspa->spa_config_txg = spa->spa_config_txg;
4935        spa_set_log_state(newspa, SPA_LOG_CLEAR);
4936
4937        /* release the spa config lock, retaining the namespace lock */
4938        spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
4939
4940        if (zio_injection_enabled)
4941                zio_handle_panic_injection(spa, FTAG, 1);
```



```
4943        spa_activate(newspa, spa_mode_global);
4944        spa_async_suspend(newspa);
4945
4946        /* create the new pool from the disks of the original pool */
4947        error = spa_load(newspa, SPA_LOAD_IMPORT, SPA_IMPORT_ASSEMBLE, B_TRUE);
4948        if (error)
4949                goto out;
4950
4951        /* if that worked, generate a real config for the new pool */
4952        if (newspa->spa_root_vdev != NULL) {
4953                VERIFY(nvlist_alloc(&newspa->spa_config_splitting,
4954                    NV_UNIQUE_NAME, KM_SLEEP) == 0);
4955                VERIFY(nvlist_add_uint64(newspa->spa_config_splitting,
4956                    ZPOOL_CONFIG_SPLIT_GUID, spa_guid(spa)) == 0);
4957                spa_config_set(newspa, spa_config_generate(newspa, NULL, -1ULL,
4958                    B_TRUE));
4959        }
4960
4961        /* set the props */
4962        if (props != NULL) {
4963                spa_configfile_set(newspa, props, B_FALSE);
4964                error = spa_prop_set(newspa, props);
4965                if (error)
4966                        goto out;
4967        }
4968
4969        /* flush everything */
4970        txg = spa_vdev_config_enter(newspa);
4971        vdev_config_dirty(newspa->spa_root_vdev);
4972        (void) spa_vdev_config_exit(newspa, NULL, txg, 0, FTAG);
4973
4974        if (zio_injection_enabled)
4975                zio_handle_panic_injection(spa, FTAG, 2);
4976
4977        spa_async_resume(newspa);
4978
4979        /* finally, update the original pool's config */
4980        txg = spa_vdev_config_enter(spa);
4981        tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
4982        error = dmu_tx_assign(tx, TXG_WAIT);
4983        if (error != 0)
4984                dmu_tx_abort(tx);
4985        for (c = 0; c < children; c++) {
4986                if (vml[c] != NULL) {
4987                        vdev_split(vml[c]);
4988                        if (error == 0)
4989                                spa_history_log_internal(spa, "detach", tx,
4990                                    "vdev=%s", vml[c]->vdev_path);
4991                        vdev_free(vml[c]);
4992                }
4993        }
4994        vdev_config_dirty(spa->spa_root_vdev);
4995        spa->spa_config_splitting = NULL;
4996        nvlist_free(nvl);
4997        if (error == 0)
4998                dmu_tx_commit(tx);
4999        (void) spa_vdev_exit(spa, NULL, txg, 0);
5000
5001        if (zio_injection_enabled)
5002                zio_handle_panic_injection(spa, FTAG, 3);
5003
5004        /* split is complete; log a history record */
5005        spa_history_log_internal(newspa, "split", NULL,
5006            "from pool %s", spa_name(spa));
```

```
5008         kmem_free(vml, children * sizeof (vdev_t *));

5010         /* if we're not going to mount the filesystems in userland, export */
5011         if (exp)
5012                 error = spa_export_common(newname, POOL_STATE_EXPORTED, NULL,
5013                     B_FALSE, B_FALSE);

5015         return (error);

5017 out:
5018         spa_unload(newspa);
5019         spa_deactivate(newspa);
5020         spa_remove(newspa);

5022         txg = spa_vdev_config_enter(spa);

5024         /* re-online all offlined disks */
5025         for (c = 0; c < children; c++) {
5026                 if (vml[c] != NULL)
5027                         vml[c]->vdev_offline = B_FALSE;
5028         }
5029         vdev_reopen(spa->spa_root_vdev);

5031         nvlist_free(spa->spa_config_splitting);
5032         spa->spa_config_splitting = NULL;
5033         (void) spa_vdev_exit(spa, NULL, txg, error);

5035         kmem_free(vml, children * sizeof (vdev_t *));
5036         return (error);
5037 }

5039 static nvlist_t *
5040 spa_nvlist_lookup_by_guid(nvlist_t **nvpp, int count, uint64_t target_guid)
5041 {
5042         for (int i = 0; i < count; i++) {
5043                 uint64_t guid;

5045                 VERIFY(nvlist_lookup_uint64(nvpp[i], ZPOOL_CONFIG_GUID,
5046                     &guid) == 0);

5048                 if (guid == target_guid)
5049                         return (nvpp[i]);
5050         }

5052         return (NULL);
5053 }

5055 static void
5056 spa_vdev_remove_aux(nvlist_t *config, char *name, nvlist_t **dev, int count,
5057         nvlist_t *dev_to_remove)
5058 {
5059         nvlist_t **newdev = NULL;

5061         if (count > 1)
5062                 newdev = kmem_alloc((count - 1) * sizeof (void *), KM_SLEEP);

5064         for (int i = 0, j = 0; i < count; i++) {
5065                 if (dev[i] == dev_to_remove)
5066                         continue;
5067                 VERIFY(nvlist_dup(dev[i], &newdev[j++], KM_SLEEP) == 0);
5068         }

5070         VERIFY(nvlist_remove(config, name, DATA_TYPE_NVLIST_ARRAY) == 0);
5071         VERIFY(nvlist_add_nvlist_array(config, name, newdev, count - 1) == 0);

5073         for (int i = 0; i < count - 1; i++)
```



```
5074                 nvlist_free(newdev[i]);

5076         if (count > 1)
5077                 kmem_free(newdev, (count - 1) * sizeof (void *));
5078 }

5080 /*
5081  * Evacuate the device.
5082  */
5083 static int
5084 spa_vdev_remove_evacuate(spa_t *spa, vdev_t *vd)
5085 {
5086         uint64_t txg;
5087         int error = 0;

5089         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5090         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
5091         ASSERT(vd == vd->vdev_top);

5093         /*
5094          * Evacuate the device.  We don't hold the config lock as writer
5095          * since we need to do I/O but we do keep the
5096          * spa_namespace_lock held.  Once this completes the device
5097          * should no longer have any blocks allocated on it.
5098          */
5099         if (vd->vdev_islog) {
5100                 if (vd->vdev_stat.vs_alloc != 0)
5101                         error = spa_offline_log(spa);
5102         } else {
5103                 error = SET_ERROR(ENOTSUP);
5104         }

5106         if (error)
5107                 return (error);

5109         /*
5110          * The evacuation succeeded.  Remove any remaining MOS metadata
5111          * associated with this vdev, and wait for these changes to sync.
5112          */
5113         ASSERT0(vd->vdev_stat.vs_alloc);
5114         txg = spa_vdev_config_enter(spa);
5115         vd->vdev_removing = B_TRUE;
5116         vdev_dirty(vd, 0, NULL, txg);
5117         vdev_config_dirty(vd);
5118         spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);

5120         return (0);
5121 }

5123 /*
5124  * Complete the removal by cleaning up the namespace.
5125  */
5126 static void
5127 spa_vdev_remove_from_namespace(spa_t *spa, vdev_t *vd)
5128 {
5129         vdev_t *rvd = spa->spa_root_vdev;
5130         uint64_t id = vd->vdev_id;
5131         boolean_t last_vdev = (id == (rvd->vdev_children - 1));

5133         ASSERT(MUTEX_HELD(&spa_namespace_lock));
5134         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
5135         ASSERT(vd == vd->vdev_top);

5137         /*
5138          * Only remove any devices which are empty.
5139          */
```

```
5140         if (vd->vdev_stat.vs_alloc != 0)
5141                 return;

5143         (void) vdev_label_init(vd, 0, VDEV_LABEL_REMOVE);

5145         if (list_link_active(&vd->vdev_state_dirty_node))
5146                 vdev_state_clean(vd);
5147         if (list_link_active(&vd->vdev_config_dirty_node))
5148                 vdev_config_clean(vd);

5150         vdev_free(vd);

5152         if (last_vdev) {
5153                 vdev_compact_children(rvd);
5154         } else {
5155                 vd = vdev_alloc_common(spa, id, 0, &vdev_hole_ops);
5156                 vdev_add_child(rvd, vd);
5157         }
5158         vdev_config_dirty(rvd);

5160         /*
5161          * Reassess the health of our root vdev.
5162          */
5163         vdev_reopen(rvd);
5164 }

5166 /*
5167  * Remove a device from the pool -
5168  *
5169  * Removing a device from the vdev namespace requires several steps
5170  * and can take a significant amount of time.  As a result we use
5171  * the spa_vdev_config_[enter/exit] functions which allow us to
5172  * grab and release the spa_config_lock while still holding the namespace
5173  * lock.  During each step the configuration is synced out.
5174  */

5176 /*
5177  * Remove a device from the pool.  Currently, this supports removing only hot
5178  * spares, slogs, and level 2 ARC devices.
5179  */
5180 int
5181 spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare)
5182 {
5183         vdev_t *vd;
5184         metaslab_group_t *mg;
5185         nvlist_t **spares, **l2cache, *nv;
5186         uint64_t txg = 0;
5187         uint_t nspares, nl2cache;
5188         int error = 0;
5189         boolean_t locked = MUTEX_HELD(&spa_namespace_lock);

5191         ASSERT(spa_writeable(spa));

5193         if (!locked)
5194                 txg = spa_vdev_enter(spa);

5196         vd = spa_lookup_by_guid(spa, guid, B_FALSE);

5198         if (spa->spa_spares.sav_vdevs != NULL &&
5199             nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
5200             ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0 &&
5201             (nv = spa_nvlist_lookup_by_guid(spares, nspares, guid)) != NULL) {
5202                 /*
5203                  * Only remove the hot spare if it's not currently in use
5204                  * in this pool.
5205                  */
```



```
5206                 if (vd == NULL || unspare) {
5207                         spa_vdev_remove_aux(spa->spa_spares.sav_config,
5208                             ZPOOL_CONFIG_SPARES, spares, nspares, nv);
5209                         spa_load_spares(spa);
5210                         spa->spa_spares.sav_sync = B_TRUE;
5211                 } else {
5212                         error = SET_ERROR(EBUSY);
5213                 }
5214         } else if (spa->spa_l2cache.sav_vdevs != NULL &&
5215             nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config,
5216             ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0 &&
5217             (nv = spa_nvlist_lookup_by_guid(l2cache, nl2cache, guid)) != NULL) {
5218                 /*
5219                  * Cache devices can always be removed.
5220                  */
5221                 spa_vdev_remove_aux(spa->spa_l2cache.sav_config,
5222                     ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache, nv);
5223                 spa_load_l2cache(spa);
5224                 spa->spa_l2cache.sav_sync = B_TRUE;
5225         } else if (vd != NULL && vd->vdev_islog) {
5226                 ASSERT(!locked);
5227                 ASSERT(vd == vd->vdev_top);

5229                 /*
5230                  * XXX - Once we have bp-rewrite this should
5231                  * become the common case.
5232                  */

5234                 mg = vd->vdev_mg;

5236                 /*
5237                  * Stop allocating from this vdev.
5238                  */
5239                 metaslab_group_passivate(mg);

5241                 /*
5242                  * Wait for the youngest allocations and frees to sync,
5243                  * and then wait for the deferral of those frees to finish.
5244                  */
5245                 spa_vdev_config_exit(spa, NULL,
5246                     txg + TXG_CONCURRENT_STATES + TXG_DEFER_SIZE, 0, FTAG);

5248                 /*
5249                  * Attempt to evacuate the vdev.
5250                  */
5251                 error = spa_vdev_remove_evacuate(spa, vd);

5253                 txg = spa_vdev_config_enter(spa);

5255                 /*
5256                  * If we couldn't evacuate the vdev, unwind.
5257                  */
5258                 if (error) {
5259                         metaslab_group_activate(mg);
5260                         return (spa_vdev_exit(spa, NULL, txg, error));
5261                 }

5263                 /*
5264                  * Clean up the vdev namespace.
5265                  */
5266                 spa_vdev_remove_from_namespace(spa, vd);

5268         } else if (vd != NULL) {
5269                 /*
5270                  * Normal vdevs cannot be removed (yet).
5271                  */
```

```
5272                 error = SET_ERROR(ENOTSUP);
5273         } else {
5274                 /*
5275                  * There is no vdev of any kind with the specified guid.
5276                  */
5277                 error = SET_ERROR(ENOENT);
5278         }

5280         if (!locked)
5281                 return (spa_vdev_exit(spa, NULL, txg, error));

5283         return (error);
5284 }

5286 /*
5287  * Find any device that's done replacing, or a vdev marked 'unspare' that's
5288  * current spared, so we can detach it.
5289  */
5290 static vdev_t *
5291 spa_vdev_resilver_done_hunt(vdev_t *vd)
5292 {
5293         vdev_t *newvd, *oldvd;

5295         for (int c = 0; c < vd->vdev_children; c++) {
5296                 oldvd = spa_vdev_resilver_done_hunt(vd->vdev_child[c]);
5297                 if (oldvd != NULL)
5298                         return (oldvd);
5299         }

5301         /*
5302          * Check for a completed replacement.  We always consider the first
5303          * vdev in the list to be the oldest vdev, and the last one to be
5304          * the newest (see spa_vdev_attach() for how that works).  In
5305          * the case where the newest vdev is faulted, we will not automatically
5306          * remove it after a resilver completes.  This is OK as it will require
5307          * user intervention to determine which disk the admin wishes to keep.
5308          */
5309         if (vd->vdev_ops == &vdev_replacing_ops) {
5310                 ASSERT(vd->vdev_children > 1);

5312                 newvd = vd->vdev_child[vd->vdev_children - 1];
5313                 oldvd = vd->vdev_child[0];

5315                 if (vdev_dtl_empty(newvd, DTL_MISSING) &&
5316                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5317                     !vdev_dtl_required(oldvd))
5318                         return (oldvd);
5319         }

5321         /*
5322          * Check for a completed resilver with the 'unspare' flag set.
5323          */
5324         if (vd->vdev_ops == &vdev_spare_ops) {
5325                 vdev_t *first = vd->vdev_child[0];
5326                 vdev_t *last = vd->vdev_child[vd->vdev_children - 1];

5328                 if (last->vdev_unspare) {
5329                         oldvd = first;
5330                         newvd = last;
5331                 } else if (first->vdev_unspare) {
5332                         oldvd = last;
5333                         newvd = first;
5334                 } else {
5335                         oldvd = NULL;
5336                 }
```



```
5338                 if (oldvd != NULL &&
5339                     vdev_dtl_empty(newvd, DTL_MISSING) &&
5340                     vdev_dtl_empty(newvd, DTL_OUTAGE) &&
5341                     !vdev_dtl_required(oldvd))
5342                         return (oldvd);

5344                 /*
5345                  * If there are more than two spares attached to a disk,
5346                  * and those spares are not required, then we want to
5347                  * attempt to free them up now so that they can be used
5348                  * by other pools.  Once we're back down to a single
5349                  * disk+spare, we stop removing them.
5350                  */
5351                 if (vd->vdev_children > 2) {
5352                         newvd = vd->vdev_child[1];

5354                         if (newvd->vdev_isspare && last->vdev_isspare &&
5355                             vdev_dtl_empty(last, DTL_MISSING) &&
5356                             vdev_dtl_empty(last, DTL_OUTAGE) &&
5357                             !vdev_dtl_required(newvd))
5358                                 return (newvd);
5359                 }
5360         }

5362         return (NULL);
5363 }

5365 static void
5366 spa_vdev_resilver_done(spa_t *spa)
5367 {
5368         vdev_t *vd, *pvd, *ppvd;
5369         uint64_t guid, sguid, pguid, ppguid;

5371         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

5373         while ((vd = spa_vdev_resilver_done_hunt(spa->spa_root_vdev)) != NULL) {
5374                 pvd = vd->vdev_parent;
5375                 ppvd = pvd->vdev_parent;
5376                 guid = vd->vdev_guid;
5377                 pguid = pvd->vdev_guid;
5378                 ppguid = ppvd->vdev_guid;
5379                 sguid = 0;
5380                 /*
5381                  * If we have just finished replacing a hot spared device, then
5382                  * we need to detach the parent's first child (the original hot
5383                  * spare) as well.
5384                  */
5385                 if (ppvd->vdev_ops == &vdev_spare_ops && pvd->vdev_id == 0 &&
5386                     ppvd->vdev_children == 2) {
5387                         ASSERT(pvd->vdev_ops == &vdev_replacing_ops);
5388                         sguid = ppvd->vdev_child[1]->vdev_guid;
5389                 }
5390                 spa_config_exit(spa, SCL_ALL, FTAG);
5391                 if (spa_vdev_detach(spa, guid, pguid, B_TRUE) != 0)
5392                         return;
5393                 if (sguid && spa_vdev_detach(spa, sguid, ppguid, B_TRUE) != 0)
5394                         return;
5395                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
5396         }

5398         spa_config_exit(spa, SCL_ALL, FTAG);
5399 }

5401 /*
5402  * Update the stored path or FRU for this vdev.
5403  */
```

```
5404 int
5405 spa_vdev_set_common(spa_t *spa, uint64_t guid, const char *value,
5406     boolean_t ispath)
5407 {
5408         vdev_t *vd;
5409         boolean_t sync = B_FALSE;

5411         ASSERT(spa_writeable(spa));

5413         spa_vdev_state_enter(spa, SCL_ALL);

5415         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
5416                 return (spa_vdev_state_exit(spa, NULL, ENOENT));

5418         if (!vd->vdev_ops->vdev_op_leaf)
5419                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

5421         if (ispath) {
5422                 if (strcmp(value, vd->vdev_path) != 0) {
5423                         spa_strfree(vd->vdev_path);
5424                         vd->vdev_path = spa_strdup(value);
5425                         sync = B_TRUE;
5426                 }
5427         } else {
5428                 if (vd->vdev_fru == NULL) {
5429                         vd->vdev_fru = spa_strdup(value);
5430                         sync = B_TRUE;
5431                 } else if (strcmp(value, vd->vdev_fru) != 0) {
5432                         spa_strfree(vd->vdev_fru);
5433                         vd->vdev_fru = spa_strdup(value);
5434                         sync = B_TRUE;
5435                 }
5436         }

5438         return (spa_vdev_state_exit(spa, sync ? vd : NULL, 0));
5439 }

5441 int
5442 spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath)
5443 {
5444         return (spa_vdev_set_common(spa, guid, newpath, B_TRUE));
5445 }

5447 int
5448 spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru)
5449 {
5450         return (spa_vdev_set_common(spa, guid, newfru, B_FALSE));
5451 }

5453 /*
5454  * ========================================================================
5455  * SPA Scanning
5456  * ========================================================================
5457  */

5459 int
5460 spa_scan_stop(spa_t *spa)
5461 {
5462         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
5463         if (dsl_scan_resilvering(spa->spa_dsl_pool))
5464                 return (SET_ERROR(EBUSY));
5465         return (dsl_scan_cancel(spa->spa_dsl_pool));
5466 }

5468 int
5469 spa_scan(spa_t *spa, pool_scan_func_t func)
```



```
5470 {
5471         ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);

5473         if (func >= POOL_SCAN_FUNCS || func == POOL_SCAN_NONE)
5474                 return (SET_ERROR(ENOTSUP));

5476         /*
5477          * If a resilver was requested, but there is no DTL on a
5478          * writeable leaf device, we have nothing to do.
5479          */
5480         if (func == POOL_SCAN_RESILVER &&
5481             !vdev_resilver_needed(spa->spa_root_vdev, NULL, NULL)) {
5482                 spa_async_request(spa, SPA_ASYNC_RESILVER_DONE);
5483                 return (0);
5484         }

5486         return (dsl_scan(spa->spa_dsl_pool, func));
5487 }

5489 /*
5490  * ========================================================================
5491  * SPA async task processing
5492  * ========================================================================
5493  */

5495 static void
5496 spa_async_remove(spa_t *spa, vdev_t *vd)
5497 {
5498         if (vd->vdev_remove_wanted) {
5499                 vd->vdev_remove_wanted = B_FALSE;
5500                 vd->vdev_delayed_close = B_FALSE;
5501                 vdev_set_state(vd, B_FALSE, VDEV_STATE_REMOVED, VDEV_AUX_NONE);

5503                 /*
5504                  * We want to clear the stats, but we don't want to do a full
5505                  * vdev_clear() as that will cause us to throw away
5506                  * degraded/faulted state as well as attempt to reopen the
5507                  * device, all of which is a waste.
5508                  */
5509                 vd->vdev_stat.vs_read_errors = 0;
5510                 vd->vdev_stat.vs_write_errors = 0;
5511                 vd->vdev_stat.vs_checksum_errors = 0;

5513                 vdev_state_dirty(vd->vdev_top);
5514         }

5516         for (int c = 0; c < vd->vdev_children; c++)
5517                 spa_async_remove(spa, vd->vdev_child[c]);
5518 }

5520 static void
5521 spa_async_probe(spa_t *spa, vdev_t *vd)
5522 {
5523         if (vd->vdev_probe_wanted) {
5524                 vd->vdev_probe_wanted = B_FALSE;
5525                 vdev_reopen(vd);        /* vdev_open() does the actual probe */
5526         }

5528         for (int c = 0; c < vd->vdev_children; c++)
5529                 spa_async_probe(spa, vd->vdev_child[c]);
5530 }

5532 static void
5533 spa_async_autoexpand(spa_t *spa, vdev_t *vd)
5534 {
5535         sysevent_id_t eid;
```

```
5536         nvlist_t *attr;
5537         char *physpath;

5539         if (!spa->spa_autoexpand)
5540                 return;

5542         for (int c = 0; c < vd->vdev_children; c++) {
5543                 vdev_t *cvd = vd->vdev_child[c];
5544                 spa_async_autoexpand(spa, cvd);
5545         }

5547         if (!vd->vdev_ops->vdev_op_leaf || vd->vdev_physpath == NULL)
5548                 return;

5550         physpath = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
5551         (void) snprintf(physpath, MAXPATHLEN, "/devices%s", vd->vdev_physpath);

5553         VERIFY(nvlist_alloc(&attr, NV_UNIQUE_NAME, KM_SLEEP) == 0);
5554         VERIFY(nvlist_add_string(attr, DEV_PHYS_PATH, physpath) == 0);

5556         (void) ddi_log_sysevent(zfs_dip, SUNW_VENDOR, EC_DEV_STATUS,
5557             ESC_DEV_DLE, attr, &eid, DDI_SLEEP);

5559         nvlist_free(attr);
5560         kmem_free(physpath, MAXPATHLEN);
5561 }

5563 static void
5564 spa_async_thread(spa_t *spa)
5565 {
5566         int tasks;

5568         ASSERT(spa->spa_sync_on);

5570         mutex_enter(&spa->spa_async_lock);
5571         tasks = spa->spa_async_tasks;
5572         spa->spa_async_tasks = 0;
5573         mutex_exit(&spa->spa_async_lock);

5575         /*
5576          * See if the config needs to be updated.
5577          */
5578         if (tasks & SPA_ASYNC_CONFIG_UPDATE) {
5579                 uint64_t old_space, new_space;

5581                 mutex_enter(&spa_namespace_lock);
5582                 old_space = metaslab_class_get_space(spa_normal_class(spa));
5583                 spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
5584                 new_space = metaslab_class_get_space(spa_normal_class(spa));
5585                 mutex_exit(&spa_namespace_lock);

5587                 /*
5588                  * If the pool grew as a result of the config update,
5589                  * then log an internal history event.
5590                  */
5591                 if (new_space != old_space) {
5592                         spa_history_log_internal(spa, "vdev online", NULL,
5593                             "pool '%s' size: %llu(+%llu)",
5594                             spa_name(spa), new_space, new_space - old_space);
5595                 }
5596         }

5598         /*
5599          * See if any devices need to be marked REMOVED.
5600          */
5601         if (tasks & SPA_ASYNC_REMOVE) {
```



```
5602                 spa_vdev_state_enter(spa, SCL_NONE);
5603                 spa_async_remove(spa, spa->spa_root_vdev);
5604                 for (int i = 0; i < spa->spa_l2cache.sav_count; i++)
5605                         spa_async_remove(spa, spa->spa_l2cache.sav_vdevs[i]);
5606                 for (int i = 0; i < spa->spa_spares.sav_count; i++)
5607                         spa_async_remove(spa, spa->spa_spares.sav_vdevs[i]);
5608                 (void) spa_vdev_state_exit(spa, NULL, 0);
5609         }

5611         if ((tasks & SPA_ASYNC_AUTOEXPAND) && !spa_suspended(spa)) {
5612                 spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
5613                 spa_async_autoexpand(spa, spa->spa_root_vdev);
5614                 spa_config_exit(spa, SCL_CONFIG, FTAG);
5615         }

5617         /*
5618          * See if any devices need to be probed.
5619          */
5620         if (tasks & SPA_ASYNC_PROBE) {
5621                 spa_vdev_state_enter(spa, SCL_NONE);
5622                 spa_async_probe(spa, spa->spa_root_vdev);
5623                 (void) spa_vdev_state_exit(spa, NULL, 0);
5624         }

5626         /*
5627          * If any devices are done replacing, detach them.
5628          */
5629         if (tasks & SPA_ASYNC_RESILVER_DONE)
5630                 spa_vdev_resilver_done(spa);

5632         /*
5633          * Kick off a resilver.
5634          */
5635         if (tasks & SPA_ASYNC_RESILVER)
5636                 dsl_resilver_restart(spa->spa_dsl_pool, 0);

5638         /*
5639          * Let the world know that we're done.
5640          */
5641         mutex_enter(&spa->spa_async_lock);
5642         spa->spa_async_thread = NULL;
5643         cv_broadcast(&spa->spa_async_cv);
5644         mutex_exit(&spa->spa_async_lock);
5645         thread_exit();
5646 }

5648 void
5649 spa_async_suspend(spa_t *spa)
5650 {
5651         mutex_enter(&spa->spa_async_lock);
5652         spa->spa_async_suspended++;
5653         while (spa->spa_async_thread != NULL)
5654                 cv_wait(&spa->spa_async_cv, &spa->spa_async_lock);
5655         mutex_exit(&spa->spa_async_lock);
5656 }

5658 void
5659 spa_async_resume(spa_t *spa)
5660 {
5661         mutex_enter(&spa->spa_async_lock);
5662         ASSERT(spa->spa_async_suspended != 0);
5663         spa->spa_async_suspended--;
5664         mutex_exit(&spa->spa_async_lock);
5665 }

5667 static void
```

```
5668 spa_async_dispatch(spa_t *spa)
5669 {
5670         mutex_enter(&spa->spa_async_lock);
5671         if (spa->spa_async_tasks && !spa->spa_async_suspended &&
5672             spa->spa_async_thread == NULL &&
5673             rootdir != NULL && !vn_is_readonly(rootdir))
5674                 spa->spa_async_thread = thread_create(NULL, 0,
5675                     spa_async_thread, spa, 0, &p0, TS_RUN, maxclsyspri);
5676         mutex_exit(&spa->spa_async_lock);
5677 }

5679 void
5680 spa_async_request(spa_t *spa, int task)
5681 {
5682         zfs_dbgmsg("spa=%s async request task=%u", spa->spa_name, task);
5683         mutex_enter(&spa->spa_async_lock);
5684         spa->spa_async_tasks |= task;
5685         mutex_exit(&spa->spa_async_lock);
5686 }

5688 /*
5689  * ==========================================================================
5690  * SPA syncing routines
5691  * ==========================================================================
5692  */

5694 static int
5695 bpobj_enqueue_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
5696 {
5697         bpobj_t *bpo = arg;
5698         bpobj_enqueue(bpo, bp, tx);
5699         return (0);
5700 }

5702 static int
5703 spa_free_sync_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
5704 {
5705         zio_t *zio = arg;

5707         zio_nowait(zio_free_sync(zio, zio->io_spa, dmu_tx_get_txg(tx), bp,
5708             zio->io_flags));
5709         return (0);
5710 }

5712 static void
5713 spa_sync_nvlist(spa_t *spa, uint64_t obj, nvlist_t *nv, dmu_tx_t *tx)
5714 {
5715         char *packed = NULL;
5716         size_t bufsize;
5717         size_t nvsize = 0;
5718         dmu_buf_t *db;

5720         VERIFY(nvlist_size(nv, &nvsize, NV_ENCODE_XDR) == 0);

5722         /*
5723          * Write full (SPA_CONFIG_BLOCKSIZE) blocks of configuration
5724          * information.  This avoids the dbuf_will_dirty() path and
5725          * saves us a pre-read to get data we don't actually care about.
5726          */
5727         bufsize = P2ROUNDUP((uint64_t)nvsize, SPA_CONFIG_BLOCKSIZE);
5728         packed = kmem_alloc(bufsize, KM_SLEEP);

5730         VERIFY(nvlist_pack(nv, &packed, &nvsize, NV_ENCODE_XDR,
5731             KM_SLEEP) == 0);
5732         bzero(packed + nvsize, bufsize - nvsize);
```



```
5734         dmu_write(spa->spa_meta_objset, obj, 0, bufsize, packed, tx);

5736         kmem_free(packed, bufsize);

5738         VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset, obj, FTAG, &db));
5739         dmu_buf_will_dirty(db, tx);
5740         *(uint64_t *)db->db_data = nvsize;
5741         dmu_buf_rele(db, FTAG);
5742 }

5744 static void
5745 spa_sync_aux_dev(spa_t *spa, spa_aux_vdev_t *sav, dmu_tx_t *tx,
5746     const char *config, const char *entry)
5747 {
5748         nvlist_t *nvroot;
5749         nvlist_t **list;
5750         int i;

5752         if (!sav->sav_sync)
5753                 return;

5755         /*
5756          * Update the MOS nvlist describing the list of available devices.
5757          * spa_validate_aux() will have already made sure this nvlist is
5758          * valid and the vdevs are labeled appropriately.
5759          */
5760         if (sav->sav_object == 0) {
5761                 sav->sav_object = dmu_object_alloc(spa->spa_meta_objset,
5762                     DMU_OT_PACKED_NVLIST, 1 << 14, DMU_OT_PACKED_NVLIST_SIZE,
5763                     sizeof (uint64_t), tx);
5764                 VERIFY(zap_update(spa->spa_meta_objset,
5765                     DMU_POOL_DIRECTORY_OBJECT, entry, sizeof (uint64_t), 1,
5766                     &sav->sav_object, tx) == 0);
5767         }

5769         VERIFY(nvlist_alloc(&nvroot, NV_UNIQUE_NAME, KM_SLEEP) == 0);
5770         if (sav->sav_count == 0) {
5771                 VERIFY(nvlist_add_nvlist_array(nvroot, config, NULL, 0) == 0);
5772         } else {
5773                 list = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
5774                 for (i = 0; i < sav->sav_count; i++)
5775                         list[i] = vdev_config_generate(spa, sav->sav_vdevs[i],
5776                             B_FALSE, VDEV_CONFIG_L2CACHE);
5777                 VERIFY(nvlist_add_nvlist_array(nvroot, config, list,
5778                     sav->sav_count) == 0);
5779                 for (i = 0; i < sav->sav_count; i++)
5780                         nvlist_free(list[i]);
5781                 kmem_free(list, sav->sav_count * sizeof (void *));
5782         }

5784         spa_sync_nvlist(spa, sav->sav_object, nvroot, tx);
5785         nvlist_free(nvroot);

5787         sav->sav_sync = B_FALSE;
5788 }

5790 static void
5791 spa_sync_config_object(spa_t *spa, dmu_tx_t *tx)
5792 {
5793         nvlist_t *config;

5795         if (list_is_empty(&spa->spa_config_dirty_list))
5796                 return;

5798         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
```

```
5800         config = spa_config_generate(spa, spa->spa_root_vdev,
5801             dmu_tx_get_txg(tx), B_FALSE);
5802
5803         /*
5804          * If we're upgrading the spa version then make sure that
5805          * the config object gets updated with the correct version.
5806          */
5807         if (spa->spa_ubsync.ub_version < spa->spa_uberblock.ub_version)
5808                 fnvlist_add_uint64(config, ZPOOL_CONFIG_VERSION,
5809                     spa->spa_uberblock.ub_version);
5810
5811         spa_config_exit(spa, SCL_STATE, FTAG);
5812
5813         if (spa->spa_config_syncing)
5814                 nvlist_free(spa->spa_config_syncing);
5815         spa->spa_config_syncing = config;
5816
5817         spa_sync_nvlist(spa, spa->spa_config_object, config, tx);
5818 }
5819
5820 static void
5821 spa_sync_version(void *arg, dmu_tx_t *tx)
5822 {
5823         uint64_t *versionp = arg;
5824         uint64_t version = *versionp;
5825         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
5826
5827         /*
5828          * Setting the version is special cased when first creating the pool.
5829          */
5830         ASSERT(tx->tx_txg != TXG_INITIAL);
5831
5832         ASSERT(SPA_VERSION_IS_SUPPORTED(version));
5833         ASSERT(version >= spa_version(spa));
5834
5835         spa->spa_uberblock.ub_version = version;
5836         vdev_config_dirty(spa->spa_root_vdev);
5837         spa_history_log_internal(spa, "set", tx, "version=%lld", version);
5838 }
5839
5840 /*
5841  * Set zpool properties.
5842  */
5843 static void
5844 spa_sync_props(void *arg, dmu_tx_t *tx)
5845 {
5846         nvlist_t *nvp = arg;
5847         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
5848         objset_t *mos = spa->spa_meta_objset;
5849         nvpair_t *elem = NULL;
5850
5851         mutex_enter(&spa->spa_props_lock);
5852
5853         while ((elem = nvlist_next_nvpair(nvp, elem))) {
5854                 uint64_t intval;
5855                 char *strval, *fname;
5856                 zpool_prop_t prop;
5857                 const char *propname;
5858                 zprop_type_t proptype;
5859                 zfeature_info_t *feature;
5860
5861                 switch (prop = zpool_name_to_prop(nvpair_name(elem))) {
5862                 case ZPROP_INVAL:
5863                         /*
5864                          * We checked this earlier in spa_prop_validate().
5865                          */
```



```
5866                         ASSERT(zpool_prop_feature(nvpair_name(elem)));
5867
5868                         fname = strchr(nvpair_name(elem), '@') + 1;
5869                         VERIFY3U(0, ==, zfeature_lookup_name(fname, &feature));
5870
5871                         spa_feature_enable(spa, feature, tx);
5872                         spa_history_log_internal(spa, "set", tx,
5873                             "%s=enabled", nvpair_name(elem));
5874                         break;
5875
5876                 case ZPOOL_PROP_VERSION:
5877                         VERIFY(nvpair_value_uint64(elem, &intval) == 0);
5878                         /*
5879                          * The version is synced seperatly before other
5880                          * properties and should be correct by now.
5881                          */
5882                         ASSERT3U(spa_version(spa), >=, intval);
5883                         break;
5884
5885                 case ZPOOL_PROP_ALTROOT:
5886                         /*
5887                          * 'altroot' is a non-persistent property. It should
5888                          * have been set temporarily at creation or import time.
5889                          */
5890                         ASSERT(spa->spa_root != NULL);
5891                         break;
5892
5893                 case ZPOOL_PROP_READONLY:
5894                 case ZPOOL_PROP_CACHEFILE:
5895                         /*
5896                          * 'readonly' and 'cachefile' are also non-persisitent
5897                          * properties.
5898                          */
5899                         break;
5900                 case ZPOOL_PROP_COMMENT:
5901                         VERIFY(nvpair_value_string(elem, &strval) == 0);
5902                         if (spa->spa_comment != NULL)
5903                                 spa_strfree(spa->spa_comment);
5904                         spa->spa_comment = spa_strdup(strval);
5905                         /*
5906                          * We need to dirty the configuration on all the vdevs
5907                          * so that their labels get updated.  It's unnecessary
5908                          * to do this for pool creation since the vdev's
5909                          * configuratoin has already been dirtied.
5910                          */
5911                         if (tx->tx_txg != TXG_INITIAL)
5912                                 vdev_config_dirty(spa->spa_root_vdev);
5913                         spa_history_log_internal(spa, "set", tx,
5914                             "%s=%s", nvpair_name(elem), strval);
5915                         break;
5916                 default:
5917                         /*
5918                          * Set pool property values in the poolprops mos object.
5919                          */
5920                         if (spa->spa_pool_props_object == 0) {
5921                                 spa->spa_pool_props_object =
5922                                     zap_create_link(mos, DMU_OT_POOL_PROPS,
5923                                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_PROPS,
5924                                     tx);
5925                         }
5926
5927                         /* normalize the property name */
5928                         propname = zpool_prop_to_name(prop);
5929                         proptype = zpool_prop_get_type(prop);
5930
5931                         if (nvpair_type(elem) == DATA_TYPE_STRING) {
```

```
5932                                ASSERT(proptype == PROP_TYPE_STRING);
5933                                VERIFY(nvpair_value_string(elem, &strval) == 0);
5934                                VERIFY(zap_update(mos,
5935                                    spa->spa_pool_props_object, propname,
5936                                    1, strlen(strval) + 1, strval, tx) == 0);
5937                                spa_history_log_internal(spa, "set", tx,
5938                                    "%s=%s", nvpair_name(elem), strval);
5939                        } else if (nvpair_type(elem) == DATA_TYPE_UINT64) {
5940                                VERIFY(nvpair_value_uint64(elem, &intval) == 0);

5942                                if (proptype == PROP_TYPE_INDEX) {
5943                                        const char *unused;
5944                                        VERIFY(zpool_prop_index_to_string(
5945                                            prop, intval, &unused) == 0);
5946                                }
5947                                VERIFY(zap_update(mos,
5948                                    spa->spa_pool_props_object, propname,
5949                                    8, 1, &intval, tx) == 0);
5950                                spa_history_log_internal(spa, "set", tx,
5951                                    "%s=%lld", nvpair_name(elem), intval);
5952                        } else {
5953                                ASSERT(0); /* not allowed */
5954                        }

5956                        switch (prop) {
5957                        case ZPOOL_PROP_DELEGATION:
5958                                spa->spa_delegation = intval;
5959                                break;
5960                        case ZPOOL_PROP_BOOTFS:
5961                                spa->spa_bootfs = intval;
5962                                break;
5963                        case ZPOOL_PROP_FAILUREMODE:
5964                                spa->spa_failmode = intval;
5965                                break;
5966                        case ZPOOL_PROP_AUTOEXPAND:
5967                                spa->spa_autoexpand = intval;
5968                                if (tx->tx_txg != TXG_INITIAL)
5969                                        spa_async_request(spa,
5970                                            SPA_ASYNC_AUTOEXPAND);
5971                                break;
5972                        case ZPOOL_PROP_DEDUPDITTO:
5973                                spa->spa_dedup_ditto = intval;
5974                                break;
5975                        default:
5976                                break;
5977                        }
5978                }

5980        }

5982        mutex_exit(&spa->spa_props_lock);
5983 }

5985 /*
5986  * Perform one-time upgrade on-disk changes.  spa_version() does not
5987  * reflect the new version this txg, so there must be no changes this
5988  * txg to anything that the upgrade code depends on after it executes.
5989  * Therefore this must be called after dsl_pool_sync() does the sync
5990  * tasks.
5991  */
5992 static void
5993 spa_sync_upgrades(spa_t *spa, dmu_tx_t *tx)
5994 {
5995        dsl_pool_t *dp = spa->spa_dsl_pool;

5997        ASSERT(spa->spa_sync_pass == 1);
```



```
5999        rrw_enter(&dp->dp_config_rwlock, RW_WRITER, FTAG);

6001        if (spa->spa_ubsync.ub_version < SPA_VERSION_ORIGIN &&
6002            spa->spa_uberblock.ub_version >= SPA_VERSION_ORIGIN) {
6003                dsl_pool_create_origin(dp, tx);

6005                /* Keeping the origin open increases spa_minref */
6006                spa->spa_minref += 3;
6007        }

6009        if (spa->spa_ubsync.ub_version < SPA_VERSION_NEXT_CLONES &&
6010            spa->spa_uberblock.ub_version >= SPA_VERSION_NEXT_CLONES) {
6011                dsl_pool_upgrade_clones(dp, tx);
6012        }

6014        if (spa->spa_ubsync.ub_version < SPA_VERSION_DIR_CLONES &&
6015            spa->spa_uberblock.ub_version >= SPA_VERSION_DIR_CLONES) {
6016                dsl_pool_upgrade_dir_clones(dp, tx);

6018                /* Keeping the freedir open increases spa_minref */
6019                spa->spa_minref += 3;
6020        }

6022        if (spa->spa_ubsync.ub_version < SPA_VERSION_FEATURES &&
6023            spa->spa_uberblock.ub_version >= SPA_VERSION_FEATURES) {
6024                spa_feature_create_zap_objects(spa, tx);
6025        }
6026        rrw_exit(&dp->dp_config_rwlock, FTAG);
6027 }

6029 /*
6030  * Sync the specified transaction group.  New blocks may be dirtied as
6031  * part of the process, so we iterate until it converges.
6032  */
6033 void
6034 spa_sync(spa_t *spa, uint64_t txg)
6035 {
6036        dsl_pool_t *dp = spa->spa_dsl_pool;
6037        objset_t *mos = spa->spa_meta_objset;
6038        bpobj_t *defer_bpo = &spa->spa_deferred_bpobj;
6039        bplist_t *free_bpl = &spa->spa_free_bplist[txg & TXG_MASK];
6040        vdev_t *rvd = spa->spa_root_vdev;
6041        vdev_t *vd;
6042        dmu_tx_t *tx;
6043        int error;

6045        VERIFY(spa_writeable(spa));

6047        /*
6048         * Lock out configuration changes.
6049         */
6050        spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);

6052        spa->spa_syncing_txg = txg;
6053        spa->spa_sync_pass = 0;

6055        /*
6056         * If there are any pending vdev state changes, convert them
6057         * into config changes that go out with this transaction group.
6058         */
6059        spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
6060        while (list_head(&spa->spa_state_dirty_list) != NULL) {
6061                /*
6062                 * We need the write lock here because, for aux vdevs,
6063                 * calling vdev_config_dirty() modifies sav_config.
```

```
6064                 * This is ugly and will become unnecessary when we
6065                 * eliminate the aux vdev wart by integrating all vdevs
6066                 * into the root vdev tree.
6067                 */
6068                spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
6069                spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_WRITER);
6070                while ((vd = list_head(&spa->spa_state_dirty_list)) != NULL) {
6071                        vdev_state_clean(vd);
6072                        vdev_config_dirty(vd);
6073                }
6074                spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
6075                spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
6076        }
6077        spa_config_exit(spa, SCL_STATE, FTAG);

6079        tx = dmu_tx_create_assigned(dp, txg);

6081        spa->spa_sync_starttime = gethrtime();
6082        VERIFY(cyclic_reprogram(spa->spa_deadman_cycid,
6083            spa->spa_sync_starttime + spa->spa_deadman_synctime));

6085        /*
6086         * If we are upgrading to SPA_VERSION_RAIDZ_DEFLATE this txg,
6087         * set spa_deflate if we have no raid-z vdevs.
6088         */
6089        if (spa->spa_ubsync.ub_version < SPA_VERSION_RAIDZ_DEFLATE &&
6090            spa->spa_uberblock.ub_version >= SPA_VERSION_RAIDZ_DEFLATE) {
6091                int i;

6093                for (i = 0; i < rvd->vdev_children; i++) {
6094                        vd = rvd->vdev_child[i];
6095                        if (vd->vdev_deflate_ratio != SPA_MINBLOCKSIZE)
6096                                break;
6097                }
6098                if (i == rvd->vdev_children) {
6099                        spa->spa_deflate = TRUE;
6100                        VERIFY(0 == zap_add(spa->spa_meta_objset,
6101                            DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
6102                            sizeof (uint64_t), 1, &spa->spa_deflate, tx));
6103                }
6104        }

6106        /*
6107         * If anything has changed in this txg, or if someone is waiting
6108         * for this txg to sync (eg, spa_vdev_remove()), push the
6109         * deferred frees from the previous txg.  If not, leave them
6110         * alone so that we don't generate work on an otherwise idle
6111         * system.
6112         */
6113        if (!txg_list_empty(&dp->dp_dirty_datasets, txg) ||
6114            !txg_list_empty(&dp->dp_dirty_dirs, txg) ||
6115            !txg_list_empty(&dp->dp_sync_tasks, txg) ||
6116            ((dsl_scan_active(dp->dp_scan) ||
6117            txg_sync_waiting(dp)) && !spa_shutting_down(spa))) {
6118                zio_t *zio = zio_root(spa, NULL, NULL, 0);
6119                VERIFY3U(bpobj_iterate(defer_bpo,
6120                    spa_free_sync_cb, zio, tx), ==, 0);
6121                VERIFY0(zio_wait(zio));
6122        }

6124        /*
6125         * Iterate to convergence.
6126         */
6127        do {
6128                int pass = ++spa->spa_sync_pass;
```



```
6130                spa_sync_config_object(spa, tx);
6131                spa_sync_aux_dev(spa, &spa->spa_spares, tx,
6132                    ZPOOL_CONFIG_SPARES, DMU_POOL_SPARES);
6133                spa_sync_aux_dev(spa, &spa->spa_l2cache, tx,
6134                    ZPOOL_CONFIG_L2CACHE, DMU_POOL_L2CACHE);
6135                spa_errlog_sync(spa, txg);
6136                dsl_pool_sync(dp, txg);

6138                if (pass < zfs_sync_pass_deferred_free) {
6139                        zio_t *zio = zio_root(spa, NULL, NULL, 0);
6140                        bplist_iterate(free_bpl, spa_free_sync_cb,
6141                            zio, tx);
6142                        VERIFY(zio_wait(zio) == 0);
6143                } else {
6144                        bplist_iterate(free_bpl, bpobj_enqueue_cb,
6145                            defer_bpo, tx);
6146                }

6148                ddt_sync(spa, txg);
6149                dsl_scan_sync(dp, tx);

6151                while (vd = txg_list_remove(&spa->spa_vdev_txg_list, txg))
6152                        vdev_sync(vd, txg);

6154                if (pass == 1)
6155                        spa_sync_upgrades(spa, tx);

6157        } while (dmu_objset_is_dirty(mos, txg));

6159        /*
6160         * Rewrite the vdev configuration (which includes the uberblock)
6161         * to commit the transaction group.
6162         *
6163         * If there are no dirty vdevs, we sync the uberblock to a few
6164         * random top-level vdevs that are known to be visible in the
6165         * config cache (see spa_vdev_add() for a complete description).
6166         * If there *are* dirty vdevs, sync the uberblock to all vdevs.
6167         */
6168        for (;;) {
6169                /*
6170                 * We hold SCL_STATE to prevent vdev open/close/etc.
6171                 * while we're attempting to write the vdev labels.
6172                 */
6173                spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);

6175                if (list_is_empty(&spa->spa_config_dirty_list)) {
6176                        vdev_t *svd[SPA_DVAS_PER_BP];
6177                        int svdcount = 0;
6178                        int children = rvd->vdev_children;
6179                        int c0 = spa_get_random(children);

6181                        for (int c = 0; c < children; c++) {
6182                                vd = rvd->vdev_child[(c0 + c) % children];
6183                                if (vd->vdev_ms_array == 0 || vd->vdev_islog)
6184                                        continue;
6185                                svd[svdcount++] = vd;
6186                                if (svdcount == SPA_DVAS_PER_BP)
6187                                        break;
6188                        }
6189                        error = vdev_config_sync(svd, svdcount, txg, B_FALSE);
6190                        if (error != 0)
6191                                error = vdev_config_sync(svd, svdcount, txg,
6192                                    B_TRUE);
6193                } else {
6194                        error = vdev_config_sync(rvd->vdev_child,
6195                            rvd->vdev_children, txg, B_FALSE);
```

```
6196                        if (error != 0)
6197                                error = vdev_config_sync(rvd->vdev_child,
6198                                    rvd->vdev_children, txg, B_TRUE);
6199                }

6201                if (error == 0)
6202                        spa->spa_last_synced_guid = rvd->vdev_guid;

6204                spa_config_exit(spa, SCL_STATE, FTAG);

6206                if (error == 0)
6207                        break;
6208                zio_suspend(spa, NULL);
6209                zio_resume_wait(spa);
6210        }
6211        dmu_tx_commit(tx);

6213        VERIFY(cyclic_reprogram(spa->spa_deadman_cycid, CY_INFINITY));

6215        /*
6216         * Clear the dirty config list.
6217         */
6218        while ((vd = list_head(&spa->spa_config_dirty_list)) != NULL)
6219                vdev_config_clean(vd);

6221        /*
6222         * Now that the new config has synced transactionally,
6223         * let it become visible to the config cache.
6224         */
6225        if (spa->spa_config_syncing != NULL) {
6226                spa_config_set(spa, spa->spa_config_syncing);
6227                spa->spa_config_txg = txg;
6228                spa->spa_config_syncing = NULL;
6229        }

6231        spa->spa_ubsync = spa->spa_uberblock;

6233        dsl_pool_sync_done(dp, txg);

6235        /*
6236         * Update usable space statistics.
6237         */
6238        while (vd = txg_list_remove(&spa->spa_vdev_txg_list, TXG_CLEAN(txg)))
6239                vdev_sync_done(vd, txg);

6241        spa_update_dspace(spa);

6243        /*
6244         * It had better be the case that we didn't dirty anything
6245         * since vdev_config_sync().
6246         */
6247        ASSERT(txg_list_empty(&dp->dp_dirty_datasets, txg));
6248        ASSERT(txg_list_empty(&dp->dp_dirty_dirs, txg));
6249        ASSERT(txg_list_empty(&spa->spa_vdev_txg_list, txg));

6251        spa->spa_sync_pass = 0;

6253        spa_config_exit(spa, SCL_CONFIG, FTAG);

6255        spa_handle_ignored_writes(spa);

6257        /*
6258         * If any async tasks have been requested, kick them off.
6259         */
6260        spa_async_dispatch(spa);
6261 }
```



```
6263 /*
6264  * Sync all pools.  We don't want to hold the namespace lock across these
6265  * operations, so we take a reference on the spa_t and drop the lock during the
6266  * sync.
6267  */
6268 void
6269 spa_sync_allpools(void)
6270 {
6271        spa_t *spa = NULL;
6272        mutex_enter(&spa_namespace_lock);
6273        while ((spa = spa_next(spa)) != NULL) {
6274                if (spa_state(spa) != POOL_STATE_ACTIVE ||
6275                    !spa_writeable(spa) || spa_suspended(spa))
6276                        continue;
6277                spa_open_ref(spa, FTAG);
6278                mutex_exit(&spa_namespace_lock);
6279                txg_wait_synced(spa_get_dsl(spa), 0);
6280                mutex_enter(&spa_namespace_lock);
6281                spa_close(spa, FTAG);
6282        }
6283        mutex_exit(&spa_namespace_lock);
6284 }

6286 /*
6287  * ==========================================================================
6288  * Miscellaneous routines
6289  * ==========================================================================
6290  */

6292 /*
6293  * Remove all pools in the system.
6294  */
6295 void
6296 spa_evict_all(void)
6297 {
6298        spa_t *spa;

6300        /*
6301         * Remove all cached state.  All pools should be closed now,
6302         * so every spa in the AVL tree should be unreferenced.
6303         */
6304        mutex_enter(&spa_namespace_lock);
6305        while ((spa = spa_next(NULL)) != NULL) {
6306                /*
6307                 * Stop async tasks.  The async thread may need to detach
6308                 * a device that's been replaced, which requires grabbing
6309                 * spa_namespace_lock, so we must drop it here.
6310                 */
6311                spa_open_ref(spa, FTAG);
6312                mutex_exit(&spa_namespace_lock);
6313                spa_async_suspend(spa);
6314                mutex_enter(&spa_namespace_lock);
6315                spa_close(spa, FTAG);

6317                if (spa->spa_state != POOL_STATE_UNINITIALIZED) {
6318                        spa_unload(spa);
6319                        spa_deactivate(spa);
6320                }
6321                spa_remove(spa);
6322        }
6323        mutex_exit(&spa_namespace_lock);
6324 }

6326 vdev_t *
6327 spa_lookup_by_guid(spa_t *spa, uint64_t guid, boolean_t aux)
```

```
6328 {
6329         vdev_t *vd;
6330         int i;

6332         if ((vd = vdev_lookup_by_guid(spa->spa_root_vdev, guid)) != NULL)
6333                 return (vd);

6335         if (aux) {
6336                 for (i = 0; i < spa->spa_l2cache.sav_count; i++) {
6337                         vd = spa->spa_l2cache.sav_vdevs[i];
6338                         if (vd->vdev_guid == guid)
6339                                 return (vd);
6340                 }

6342                 for (i = 0; i < spa->spa_spares.sav_count; i++) {
6343                         vd = spa->spa_spares.sav_vdevs[i];
6344                         if (vd->vdev_guid == guid)
6345                                 return (vd);
6346                 }
6347         }

6349         return (NULL);
6350 }

6352 void
6353 spa_upgrade(spa_t *spa, uint64_t version)
6354 {
6355         ASSERT(spa_writeable(spa));

6357         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

6359         /*
6360          * This should only be called for a non-faulted pool, and since a
6361          * future version would result in an unopenable pool, this shouldn't be
6362          * possible.
6363          */
6364         ASSERT(SPA_VERSION_IS_SUPPORTED(spa->spa_uberblock.ub_version));
6365         ASSERT(version >= spa->spa_uberblock.ub_version);

6367         spa->spa_uberblock.ub_version = version;
6368         vdev_config_dirty(spa->spa_root_vdev);

6370         spa_config_exit(spa, SCL_ALL, FTAG);

6372         txg_wait_synced(spa_get_dsl(spa), 0);
6373 }

6375 boolean_t
6376 spa_has_spare(spa_t *spa, uint64_t guid)
6377 {
6378         int i;
6379         uint64_t spareguid;
6380         spa_aux_vdev_t *sav = &spa->spa_spares;

6382         for (i = 0; i < sav->sav_count; i++)
6383                 if (sav->sav_vdevs[i]->vdev_guid == guid)
6384                         return (B_TRUE);

6386         for (i = 0; i < sav->sav_npending; i++) {
6387                 if (nvlist_lookup_uint64(sav->sav_pending[i], ZPOOL_CONFIG_GUID,
6388                     &spareguid) == 0 && spareguid == guid)
6389                         return (B_TRUE);
6390         }

6392         return (B_FALSE);
6393 }
```



```
6395 /*
6396  * Check if a pool has an active shared spare device.
6397  * Note: reference count of an active spare is 2, as a spare and as a replace
6398  */
6399 static boolean_t
6400 spa_has_active_shared_spare(spa_t *spa)
6401 {
6402         int i, refcnt;
6403         uint64_t pool;
6404         spa_aux_vdev_t *sav = &spa->spa_spares;

6406         for (i = 0; i < sav->sav_count; i++) {
6407                 if (spa_spare_exists(sav->sav_vdevs[i]->vdev_guid, &pool,
6408                     &refcnt) && pool != 0ULL && pool == spa_guid(spa) &&
6409                     refcnt > 2)
6410                         return (B_TRUE);
6411         }

6413         return (B_FALSE);
6414 }

6416 /*
6417  * Post a sysevent corresponding to the given event.  The 'name' must be one of
6418  * the event definitions in sys/sysevent/eventdefs.h.  The payload will be
6419  * filled in from the spa and (optionally) the vdev.  This doesn't do anything
6420  * in the userland libzpool, as we don't want consumers to misinterpret ztest
6421  * or zdb as real changes.
6422  */
6423 void
6424 spa_event_notify(spa_t *spa, vdev_t *vd, const char *name)
6425 {
6426 #ifdef _KERNEL
6427         sysevent_t              *ev;
6428         sysevent_attr_list_t    *attr = NULL;
6429         sysevent_value_t        value;
6430         sysevent_id_t           eid;

6432         ev = sysevent_alloc(EC_ZFS, (char *)name, SUNW_KERN_PUB "zfs",
6433             SE_SLEEP);

6435         value.value_type = SE_DATA_TYPE_STRING;
6436         value.value.sv_string = spa_name(spa);
6437         if (sysevent_add_attr(&attr, ZFS_EV_POOL_NAME, &value, SE_SLEEP) != 0)
6438                 goto done;

6440         value.value_type = SE_DATA_TYPE_UINT64;
6441         value.value.sv_uint64 = spa_guid(spa);
6442         if (sysevent_add_attr(&attr, ZFS_EV_POOL_GUID, &value, SE_SLEEP) != 0)
6443                 goto done;

6445         if (vd) {
6446                 value.value_type = SE_DATA_TYPE_UINT64;
6447                 value.value.sv_uint64 = vd->vdev_guid;
6448                 if (sysevent_add_attr(&attr, ZFS_EV_VDEV_GUID, &value,
6449                     SE_SLEEP) != 0)
6450                         goto done;

6452                 if (vd->vdev_path) {
6453                         value.value_type = SE_DATA_TYPE_STRING;
6454                         value.value.sv_string = vd->vdev_path;
6455                         if (sysevent_add_attr(&attr, ZFS_EV_VDEV_PATH,
6456                             &value, SE_SLEEP) != 0)
6457                                 goto done;
6458                 }
6459         }
```

```
6461         if (sysevent_attach_attributes(ev, attr) != 0)
6462                 goto done;
6463         attr = NULL;

6465         (void) log_sysevent(ev, SE_SLEEP, &eid);

6467 done:
6468         if (attr)
6469                 sysevent_free_attr(attr);
6470         sysevent_free(ev);
6471 #endif
6472 }
```

```
**********************************************************
   28892 Tue Apr 23 14:09:37 2013
new/usr/src/uts/common/fs/zfs/sys/dmu.h
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 284 typedef void dmu_buf_evict_func_t(struct dmu_buf *db, void *user_ptr);

 286 /*
 287  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 288  */
 289 #define DMU_POOL_DIRECTORY_OBJECT       1
 290 #define DMU_POOL_CONFIG                 "config"
 291 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
 292 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
 293 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
 294 #define DMU_POOL_ROOT_DATASET           "root_dataset"
 295 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
 296 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
 297 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
 298 #define DMU_POOL_SPARES                 "spares"
 299 #define DMU_POOL_DEFLATE                "deflate"
 300 #define DMU_POOL_HISTORY                "history"
 301 #define DMU_POOL_PROPS                  "pool_props"
 302 #define DMU_POOL_L2CACHE                "l2cache"
 303 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
 304 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
 305 #define DMU_POOL_DDT_STATS              "DDT-statistics"
 306 #define DMU_POOL_CREATION_VERSION       "creation_version"
 307 #define DMU_POOL_SCAN                   "scan"
 308 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
 309 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
 310 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

 312 /*
 313  * Allocate an object from this objset.  The range of object numbers
 314  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 315  *
 316  * The transaction must be assigned to a txg.  The newly allocated
 317  * object will be "held" in the transaction (ie. you can modify the
 318  * newly allocated object in this transaction).
 319  *
 320  * dmu_object_alloc() chooses an object and returns it in *objectp.
 321  *
 322  * dmu_object_claim() allocates a specific object number.  If that
 323  * number is already allocated, it fails and returns EEXIST.
 324  *
 325  * Return 0 on success, or ENOSPC or EEXIST as specified above.
 326  */
 327 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
 328     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 329 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 330     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 331 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 332     int blocksize, dmu_object_type_t bonustype, int bonuslen);

 334 /*
 335  * Free an object from this objset.
 336  *
 337  * The object's data will be freed as well (ie. you don't need to call
 338  * dmu_free(object, 0, -1, tx)).
```



```
 339  *
 340  * The object need not be held in the transaction.
 341  *
 342  * If there are any holds on this object's buffers (via dmu_buf_hold()),
 343  * or tx holds on the object (via dmu_tx_hold_object()), you can not
 344  * free it; it fails and returns EBUSY.
 345  *
 346  * If the object is not allocated, it fails and returns ENOENT.
 347  *
 348  * Return 0 on success, or EBUSY or ENOENT as specified above.
 349  */
 350 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

 352 /*
 353  * Find the next allocated or free object.
 354  *
 355  * The objectp parameter is in-out.  It will be updated to be the next
 356  * object which is allocated.  Ignore objects which have not been
 357  * modified since txg.
 358  *
 359  * XXX Can only be called on a objset with no dirty data.
 360  *
 361  * Returns 0 on success, or ENOENT if there are no more objects.
 362  */
 363 int dmu_object_next(objset_t *os, uint64_t *objectp,
 364     boolean_t hole, uint64_t txg);

 366 /*
 367  * Set the data blocksize for an object.
 368  *
 369  * The object cannot have any blocks allcated beyond the first.  If
 370  * the first block is allocated already, the new size must be greater
 371  * than the current block size.  If these conditions are not met,
 372  * ENOTSUP will be returned.
 373  *
 374  * Returns 0 on success, or EBUSY if there are any holds on the object
 375  * contents, or ENOTSUP as described above.
 376  */
 377 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
 378     int ibs, dmu_tx_t *tx);

 380 /*
 381  * Set the checksum property on a dnode.  The new checksum algorithm will
 382  * apply to all newly written blocks; existing blocks will not be affected.
 383  */
 384 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
 385     dmu_tx_t *tx);

 387 /*
 388  * Set the compress property on a dnode.  The new compression algorithm will
 389  * apply to all newly written blocks; existing blocks will not be affected.
 390  */
 391 void dmu_object_set_compress(objset_t *os, uint64_t object, uint8_t compress,
 392     dmu_tx_t *tx);

 394 /*
 395  * Decide how to write a block: checksum, compression, number of copies, etc.
 396  */
 397 #define WP_NOFILL       0x1
 398 #define WP_DMU_SYNC     0x2
 399 #define WP_SPILL        0x4

 401 void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
 402     struct zio_prop *zp);
 403 /*
 404  * The bonus data is accessed more or less like a regular buffer.
```

```
 * You must dmu_bonus_hold() to get the buffer, which will give you a
 * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
 * data.  As with any normal buffer, you must call dmu_buf_read() to
 * read db_data, dmu_buf_will_dirty() before modifying it, and the
 * object must be held in an assigned transaction before calling
 * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
 * buffer as well.  You must release your hold with dmu_buf_rele().
 *
 * Returns ENOENT, EIO, or 0.
#endif /* ! codereview */
 */
int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
int dmu_bonus_max(void);
int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

/*
 * Special spill buffer support used by "SA" framework
 */

int dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
int dmu_spill_hold_by_dnode(struct dnode *dn, uint32_t flags,
    void *tag, dmu_buf_t **dbp);
int dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);

/*
 * Obtain the DMU buffer from the specified object which contains the
 * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
 * that it will remain in memory.  You must release the hold with
 * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
 * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
 *
 * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
 * on the returned buffer before reading or writing the buffer's
 * db_data.  The comments for those routines describe what particular
 * operations are valid after calling them.
 *
 * The object number must be a valid, allocated object number.
 */
int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
    void *tag, dmu_buf_t **, int flags);
void dmu_buf_add_ref(dmu_buf_t *db, void* tag);
void dmu_buf_rele(dmu_buf_t *db, void *tag);
uint64_t dmu_buf_refcount(dmu_buf_t *db);

/*
 * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 * range of an object.  A pointer to an array of dmu_buf_t*'s is
 * returned (in *dbpp).
 *
 * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
 * frees the array.  The hold on the array of buffers MUST be released
 * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 * individually with dmu_buf_rele.
 */
int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
    uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

/*
 * Returns NULL on success, or the existing user ptr if it's already
 * been set.
 *
 * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
```



```
 *
 * user_data_ptr_ptr should be NULL, or a pointer to a pointer which
 * will be set to db->db_data when you are allowed to access it.  Note
 * that db->db_data (the pointer) can change when you do dmu_buf_read(),
 * dmu_buf_tryupgrade(), dmu_buf_will_dirty(), or dmu_buf_will_fill().
 * *user_data_ptr_ptr will be set to the new value when it changes.
 *
 * If non-NULL, pageout func will be called when this buffer is being
 * excised from the cache, so that you can clean up the data structure
 * pointed to by user_ptr.
 *
 * dmu_evict_user() will call the pageout func for all buffers in a
 * objset with a given pageout func.
 */
void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
/*
 * set_user_ie is the same as set_user, but request immediate eviction
 * when hold count goes to zero.
 */
void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
    void *user_data_ptr_ptr, dmu_buf_evict_func_t *pageout_func);
void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
    void *user_ptr, void *user_data_ptr_ptr,
    dmu_buf_evict_func_t *pageout_func);
void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

/*
 * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/*
 * Returns the blkptr associated with this dbuf, or NULL if not set.
 */
struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);

/*
 * Indicate that you are going to modify the buffer's data (db_data).
 *
 * The transaction (tx) must be assigned to a txg (ie. you've called
 * dmu_tx_assign()).  The buffer's object must be held in the tx
 * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
 */
void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

/*
 * Tells if the given dbuf is freeable.
 */
boolean_t dmu_buf_freeable(dmu_buf_t *);

/*
 * You must create a transaction, then hold the objects which you will
 * (or might) modify as part of this transaction.  Then you must assign
 * the transaction to a transaction group.  Once the transaction has
 * been assigned, you can modify buffers which belong to held objects as
 * part of this transaction.  You can't modify buffers before the
 * transaction has been assigned; you can't modify buffers which don't
 * belong to objects which this transaction holds; you can't hold
 * objects once the transaction has been assigned.  You may hold an
 * object which you are going to free (with dmu_object_free()), but you
 * don't have to.
 *
 * You can abort the transaction before it has been assigned.
 *
 * Note that you may hold buffers (with dmu_buf_hold) at any time,
```

```
 * regardless of transaction state.
 */

#define DMU_NEW_OBJECT  (-1ULL)
#define DMU_OBJECT_END  (-1ULL)

dmu_tx_t *dmu_tx_create(objset_t *os);
void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
    uint64_t len);
void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
void dmu_tx_abort(dmu_tx_t *tx);
int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
void dmu_tx_wait(dmu_tx_t *tx);
void dmu_tx_commit(dmu_tx_t *tx);

/*
 * To register a commit callback, dmu_tx_callback_register() must be called.
 *
 * dcb_data is a pointer to caller private data that is passed on as a
 * callback parameter. The caller is responsible for properly allocating and
 * freeing it.
 *
 * When registering a callback, the transaction must be already created, but
 * it cannot be committed or aborted. It can be assigned to a txg or not.
 *
 * The callback will be called after the transaction has been safely written
 * to stable storage and will also be called if the dmu_tx is aborted.
 * If there is any error which prevents the transaction from being committed to
 * disk, the callback will be called with a value of error != 0.
 */
typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
    void *dcb_data);

/*
 * Free up the data blocks for a defined range of a file.  If size is
 * -1, the range from offset to end-of-file is freed.
 */
int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size, dmu_tx_t *tx);
int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
        uint64_t size);
int dmu_free_object(objset_t *os, uint64_t object);

/*
 * Convenience functions.
 *
 * Canfail routines will return 0 on success, or an errno if there is a
 * nonrecoverable I/O error.
 */
#define DMU_READ_PREFETCH       0 /* prefetch */
#define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        void *buf, uint32_t flags);
void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        const void *buf, dmu_tx_t *tx);
void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        dmu_tx_t *tx);
int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
```



```
    dmu_tx_t *tx);
int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
    dmu_tx_t *tx);
int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
    uint64_t size, struct page *pp, dmu_tx_t *tx);
struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
void dmu_return_arcbuf(struct arc_buf *buf);
void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
    dmu_tx_t *tx);
int dmu_xuio_init(struct xuio *uio, int niov);
void dmu_xuio_fini(struct xuio *uio);
int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
    size_t n);
int dmu_xuio_cnt(struct xuio *uio);
struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
void dmu_xuio_clear(struct xuio *uio, int i);
void xuio_stat_wbuf_copied();
void xuio_stat_wbuf_nocopy();

extern int zfs_prefetch_disable;

/*
 * Asynchronously try to read in the data.
 */
void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
    uint64_t len);

typedef struct dmu_object_info {
        /* All sizes are in bytes unless otherwise indicated. */
        uint32_t doi_data_block_size;
        uint32_t doi_metadata_block_size;
        dmu_object_type_t doi_type;
        dmu_object_type_t doi_bonus_type;
        uint64_t doi_bonus_size;
        uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
        uint8_t doi_checksum;
        uint8_t doi_compress;
        uint8_t doi_pad[5];
        uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
        uint64_t doi_max_offset;
        uint64_t doi_fill_count;                /* number of non-empty blocks */
} dmu_object_info_t;

typedef void arc_byteswap_func_t(void *buf, size_t size);

typedef struct dmu_object_type_info {
        dmu_object_byteswap_t   ot_byteswap;
        boolean_t               ot_metadata;
        char                    *ot_name;
} dmu_object_type_info_t;

typedef struct dmu_object_byteswap_info {
        arc_byteswap_func_t     *ob_func;
        char                    *ob_name;
} dmu_object_byteswap_info_t;

extern const dmu_object_type_info_t dmu_ot[DMU_OT_NUMTYPES];
extern const dmu_object_byteswap_info_t dmu_ot_byteswap[DMU_BSWAP_NUMFUNCS];

/*
 * Get information on a DMU object.
 *
 * Return 0 on success or ENOENT if object is not allocated.
 *
 * If doi is NULL, just indicates whether the object exists.
 */
```

```
669 int dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi);
670 /* Like dmu_object_info, but faster if you have a held dnode in hand. */
671 #endif /* ! codereview */
672 void dmu_object_info_from_dnode(struct dnode *dn, dmu_object_info_t *doi);
673 /* Like dmu_object_info, but faster if you have a held dbuf in hand. */
674 #endif /* ! codereview */
675 void dmu_object_info_from_db(dmu_buf_t *db, dmu_object_info_t *doi);
676 /*
677  * Like dmu_object_info_from_db, but faster still when you only care about
678  * the size.  This is specifically optimized for zfs_getattr().
679  */
680 #endif /* ! codereview */
681 void dmu_object_size_from_db(dmu_buf_t *db, uint32_t *blksize,
682     u_longlong_t *nblk512);

684 typedef struct dmu_objset_stats {
685         uint64_t dds_num_clones; /* number of clones of this */
686         uint64_t dds_creation_txg;
687         uint64_t dds_guid;
688         dmu_objset_type_t dds_type;
689         uint8_t dds_is_snapshot;
690         uint8_t dds_inconsistent;
691         char dds_origin[MAXNAMELEN];
692 } dmu_objset_stats_t;

694 /*
695  * Get stats on a dataset.
696  */
697 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);

699 /*
700  * Add entries to the nvlist for all the objset's properties.  See
701  * zfs_prop_table[] and zfs(1m) for details on the properties.
702  */
703 void dmu_objset_stats(objset_t *os, struct nvlist *nv);

705 /*
706  * Get the space usage statistics for statvfs().
707  *
708  * refdbytes is the amount of space "referenced" by this objset.
709  * availbytes is the amount of space available to this objset, taking
710  * into account quotas & reservations, assuming that no other objsets
711  * use the space first.  These values correspond to the 'referenced' and
712  * 'available' properties, described in the zfs(1m) manpage.
713  *
714  * usedobjs and availobjs are the number of objects currently allocated,
715  * and available.
716  */
717 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
718     uint64_t *usedobjsp, uint64_t *availobjsp);

720 /*
721  * The fsid_guid is a 56-bit ID that can change to avoid collisions.
722  * (Contrast with the ds_guid which is a 64-bit ID that will never
723  * change, so there is a small probability that it will collide.)
724  */
725 uint64_t dmu_objset_fsid_guid(objset_t *os);

727 /*
728  * Get the [cm]time for an objset's snapshot dir
729  */
730 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

732 int dmu_objset_is_snapshot(objset_t *os);

734 extern struct spa *dmu_objset_spa(objset_t *os);
```



```
735 extern struct zilog *dmu_objset_zil(objset_t *os);
736 extern struct dsl_pool *dmu_objset_pool(objset_t *os);
737 extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
738 extern void dmu_objset_name(objset_t *os, char *buf);
739 extern dmu_objset_type_t dmu_objset_type(objset_t *os);
740 extern uint64_t dmu_objset_id(objset_t *os);
741 extern uint64_t dmu_objset_syncprop(objset_t *os);
742 extern uint64_t dmu_objset_logbias(objset_t *os);
743 extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
744     uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
745 extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
746     int maxlen, boolean_t *conflict);
747 extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
748     uint64_t *idp, uint64_t *offp);

750 typedef int objset_used_cb_t(dmu_object_type_t bonustype,
751     void *bonus, uint64_t *userp, uint64_t *groupp);
752 extern void dmu_objset_register_type(dmu_objset_type_t ost,
753     objset_used_cb_t *cb);
754 extern void dmu_objset_set_user(objset_t *os, void *user_ptr);
755 extern void *dmu_objset_get_user(objset_t *os);

757 /*
758  * Return the txg number for the given assigned transaction.
759  */
760 uint64_t dmu_tx_get_txg(dmu_tx_t *tx);

762 /*
763  * Synchronous write.
764  * If a parent zio is provided this function initiates a write on the
765  * provided buffer as a child of the parent zio.
766  * In the absence of a parent zio, the write is completed synchronously.
767  * At write completion, blk is filled with the bp of the written block.
768  * Note that while the data covered by this function will be on stable
769  * storage when the write completes this new data does not become a
770  * permanent part of the file until the associated transaction commits.
771  */

773 /*
774  * {zfs,zvol,ztest}_get_done() args
775  */
776 typedef struct zgd {
777         struct zilog    *zgd_zilog;
778         struct blkptr   *zgd_bp;
779         dmu_buf_t       *zgd_db;
780         struct rl       *zgd_rl;
781         void            *zgd_private;
782 } zgd_t;

784 typedef void dmu_sync_cb_t(zgd_t *arg, int error);
785 int dmu_sync(struct zio *zio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd);

787 /*
788  * Find the next hole or data block in file starting at *off
789  * Return found offset in *off. Return ESRCH for end of file.
790  */
791 int dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole,
792     uint64_t *off);

794 /*
795  * Initial setup and final teardown.
796  */
797 extern void dmu_init(void);
798 extern void dmu_fini(void);

800 typedef void (*dmu_traverse_cb_t)(objset_t *os, void *arg, struct blkptr *bp,
```

```
801     uint64_t object, uint64_t offset, int len);
802 void dmu_traverse_objset(objset_t *os, uint64_t txg_start,
803     dmu_traverse_cb_t cb, void *arg);

805 int dmu_diff(const char *tosnap_name, const char *fromsnap_name,
806     struct vnode *vp, offset_t *offp);

808 /* CRC64 table */
809 #define ZFS_CRC64_POLY  0xC96C5795D7870F42ULL   /* ECMA-182, reflected form */
810 extern uint64_t zfs_crc64_table[256];

812 #ifdef  __cplusplus
813 }
814 #endif

816 #endif  /* _SYS_DMU_H */
```

```
**********************************************************
   21958 Tue Apr 23 14:09:37 2013
new/usr/src/uts/common/fs/zfs/txg.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
340 /*
341  * Quiesce, v.: to render temporarily inactive or disabled
342  *
343  * Blocks until all transactions in the group are committed.
344  *
345  * On return, the transaction group has reached a stable state in which it can
346  * then be passed off to the syncing context.
347  */
348 #endif /* ! codereview */
349 static void
350 txg_quiesce(dsl_pool_t *dp, uint64_t txg)
351 {
352         tx_state_t *tx = &dp->dp_tx;
353         int g = txg & TXG_MASK;
354         int c;
355
356         /*
357          * Grab all tx_cpu locks so nobody else can get into this txg.
358          */
359         for (c = 0; c < max_ncpus; c++)
360                 mutex_enter(&tx->tx_cpu[c].tc_lock);
361
362         ASSERT(txg == tx->tx_open_txg);
363         tx->tx_open_txg++;
364
365         DTRACE_PROBE2(txg__quiescing, dsl_pool_t *, dp, uint64_t, txg);
366         DTRACE_PROBE2(txg__opened, dsl_pool_t *, dp, uint64_t, tx->tx_open_txg);
367
368         /*
369          * Now that we've incremented tx_open_txg, we can let threads
370          * enter the next transaction group.
371          */
372         for (c = 0; c < max_ncpus; c++)
373                 mutex_exit(&tx->tx_cpu[c].tc_lock);
374
375         /*
376          * Quiesce the transaction group by waiting for everyone to txg_exit().
377          */
378         for (c = 0; c < max_ncpus; c++) {
379                 tx_cpu_t *tc = &tx->tx_cpu[c];
380                 mutex_enter(&tc->tc_lock);
381                 while (tc->tc_count[g] != 0)
382                         cv_wait(&tc->tc_cv[g], &tc->tc_lock);
383                 mutex_exit(&tc->tc_lock);
384         }
385 }
386
387 static void
388 txg_do_callbacks(list_t *cb_list)
389 {
390         dmu_tx_do_callbacks(cb_list, 0);
391
392         list_destroy(cb_list);
393
394         kmem_free(cb_list, sizeof (list_t));
```



```
395 }
396
397 /*
398  * Dispatch the commit callbacks registered on this txg to worker threads.
399  *
400  * If no callbacks are registered for a given TXG, nothing happens.
401  * This function creates a taskq for the associated pool, if needed.
402 #endif /* ! codereview */
403  */
404 static void
405 txg_dispatch_callbacks(dsl_pool_t *dp, uint64_t txg)
406 {
407         int c;
408         tx_state_t *tx = &dp->dp_tx;
409         list_t *cb_list;
410
411         for (c = 0; c < max_ncpus; c++) {
412                 tx_cpu_t *tc = &tx->tx_cpu[c];
413                 /*
414                  * No need to lock tx_cpu_t at this point, since this can
415                  * only be called once a txg has been synced.
416                  */
340                 /* No need to lock tx_cpu_t at this point */
417
418                 int g = txg & TXG_MASK;
419
420                 if (list_is_empty(&tc->tc_callbacks[g]))
421                         continue;
422
423                 if (tx->tx_commit_cb_taskq == NULL) {
424                         /*
425                          * Commit callback taskq hasn't been created yet.
426                          */
427                         tx->tx_commit_cb_taskq = taskq_create("tx_commit_cb",
428                             max_ncpus, minclsyspri, max_ncpus, max_ncpus * 2,
429                             TASKQ_PREPOPULATE);
430                 }
431
432                 cb_list = kmem_alloc(sizeof (list_t), KM_SLEEP);
433                 list_create(cb_list, sizeof (dmu_tx_callback_t),
434                     offsetof(dmu_tx_callback_t, dcb_node));
435
436                 list_move_tail(&tc->tc_callbacks[g], cb_list);
437
438                 (void) taskq_dispatch(tx->tx_commit_cb_taskq, (task_func_t *)
439                     txg_do_callbacks, cb_list, TQ_SLEEP);
440         }
441 }
```

```
______unchanged_portion_omitted_
```

```
**********************************************************
   37985 Tue Apr 23 14:09:38 2013
new/usr/src/uts/common/fs/zfs/vdev_label.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
1031 /* Sync the uberblocks to all vdevs in svd[] */
1032 #endif /* ! codereview */
1033 int
1034 vdev_uberblock_sync_list(vdev_t **svd, int svdcount, uberblock_t *ub, int flags)
1035 {
1036         spa_t *spa = svd[0]->vdev_spa;
1037         zio_t *zio;
1038         uint64_t good_writes = 0;

1040         zio = zio_root(spa, NULL, &good_writes, flags);

1042         for (int v = 0; v < svdcount; v++)
1043                 vdev_uberblock_sync(zio, ub, svd[v], flags);

1045         (void) zio_wait(zio);

1047         /*
1048          * Flush the uberblocks to disk.  This ensures that the odd labels
1049          * are no longer needed (because the new uberblocks and the even
1050          * labels are safely on disk), so it is safe to overwrite them.
1051          */
1052         zio = zio_root(spa, NULL, NULL, flags);

1054         for (int v = 0; v < svdcount; v++)
1055                 zio_flush(zio, svd[v]);

1057         (void) zio_wait(zio);

1059         return (good_writes >= 1 ? 0 : EIO);
1060 }

1062 /*
1063  * On success, increment the count of good writes for our top-level vdev.
1064  */
1065 static void
1066 vdev_label_sync_done(zio_t *zio)
1067 {
1068         uint64_t *good_writes = zio->io_private;

1070         if (zio->io_error == 0)
1071                 atomic_add_64(good_writes, 1);
1072 }

1074 /*
1075  * If there weren't enough good writes, indicate failure to the parent.
1076  */
1077 static void
1078 vdev_label_sync_top_done(zio_t *zio)
1079 {
1080         uint64_t *good_writes = zio->io_private;

1082         if (*good_writes == 0)
1083                 zio->io_error = SET_ERROR(EIO);

1085         kmem_free(good_writes, sizeof (uint64_t));
```



```
1086 }

1088 /*
1089  * We ignore errors for log and cache devices, simply free the private data.
1090  */
1091 static void
1092 vdev_label_sync_ignore_done(zio_t *zio)
1093 {
1094         kmem_free(zio->io_private, sizeof (uint64_t));
1095 }

1097 /*
1098  * Write all even or odd labels to all leaves of the specified vdev.
1099  */
1100 static void
1101 vdev_label_sync(zio_t *zio, vdev_t *vd, int l, uint64_t txg, int flags)
1102 {
1103         nvlist_t *label;
1104         vdev_phys_t *vp;
1105         char *buf;
1106         size_t buflen;

1108         for (int c = 0; c < vd->vdev_children; c++)
1109                 vdev_label_sync(zio, vd->vdev_child[c], l, txg, flags);

1111         if (!vd->vdev_ops->vdev_op_leaf)
1112                 return;

1114         if (!vdev_writeable(vd))
1115                 return;

1117         /*
1118          * Generate a label describing the top-level config to which we belong.
1119          */
1120         label = spa_config_generate(vd->vdev_spa, vd, txg, B_FALSE);

1122         vp = zio_buf_alloc(sizeof (vdev_phys_t));
1123         bzero(vp, sizeof (vdev_phys_t));

1125         buf = vp->vp_nvlist;
1126         buflen = sizeof (vp->vp_nvlist);

1128         if (nvlist_pack(label, &buf, &buflen, NV_ENCODE_XDR, KM_SLEEP) == 0) {
1129                 for (; l < VDEV_LABELS; l += 2) {
1130                         vdev_label_write(zio, vd, l, vp,
1131                             offsetof(vdev_label_t, vl_vdev_phys),
1132                             sizeof (vdev_phys_t),
1133                             vdev_label_sync_done, zio->io_private,
1134                             flags | ZIO_FLAG_DONT_PROPAGATE);
1135                 }
1136         }

1138         zio_buf_free(vp, sizeof (vdev_phys_t));
1139         nvlist_free(label);
1140 }

1142 int
1143 vdev_label_sync_list(spa_t *spa, int l, uint64_t txg, int flags)
1144 {
1145         list_t *dl = &spa->spa_config_dirty_list;
1146         vdev_t *vd;
1147         zio_t *zio;
1148         int error;

1150         /*
1151          * Write the new labels to disk.
```

```
1152          */
1153         zio = zio_root(spa, NULL, NULL, flags);
1154
1155         for (vd = list_head(dl); vd != NULL; vd = list_next(dl, vd)) {
1156                 uint64_t *good_writes = kmem_zalloc(sizeof (uint64_t),
1157                     KM_SLEEP);
1158
1159                 ASSERT(!vd->vdev_ishole);
1160
1161                 zio_t *vio = zio_null(zio, spa, NULL,
1162                     (vd->vdev_islog || vd->vdev_aux != NULL) ?
1163                     vdev_label_sync_ignore_done : vdev_label_sync_top_done,
1164                     good_writes, flags);
1165                 vdev_label_sync(vio, vd, l, txg, flags);
1166                 zio_nowait(vio);
1167         }
1168
1169         error = zio_wait(zio);
1170
1171         /*
1172          * Flush the new labels to disk.
1173          */
1174         zio = zio_root(spa, NULL, NULL, flags);
1175
1176         for (vd = list_head(dl); vd != NULL; vd = list_next(dl, vd))
1177                 zio_flush(zio, vd);
1178
1179         (void) zio_wait(zio);
1180
1181         return (error);
1182 }
1183
1184 /*
1185  * Sync the uberblock and any changes to the vdev configuration.
1186  *
1187  * The order of operations is carefully crafted to ensure that
1188  * if the system panics or loses power at any time, the state on disk
1189  * is still transactionally consistent.  The in-line comments below
1190  * describe the failure semantics at each stage.
1191  *
1192  * Moreover, vdev_config_sync() is designed to be idempotent: if it fails
1193  * at any time, you can just call it again, and it will resume its work.
1194  */
1195 int
1196 vdev_config_sync(vdev_t **svd, int svdcount, uint64_t txg, boolean_t tryhard)
1197 {
1198         spa_t *spa = svd[0]->vdev_spa;
1199         uberblock_t *ub = &spa->spa_uberblock;
1200         vdev_t *vd;
1201         zio_t *zio;
1202         int error;
1203         int flags = ZIO_FLAG_CONFIG_WRITER | ZIO_FLAG_CANFAIL;
1204
1205         /*
1206          * Normally, we don't want to try too hard to write every label and
1207          * uberblock.  If there is a flaky disk, we don't want the rest of the
1208          * sync process to block while we retry.  But if we can't write a
1209          * single label out, we should retry with ZIO_FLAG_TRYHARD before
1210          * bailing out and declaring the pool faulted.
1211          */
1212         if (tryhard)
1213                 flags |= ZIO_FLAG_TRYHARD;
1214
1215         ASSERT(ub->ub_txg <= txg);
1216
1217         /*
```



```
1218          * If this isn't a resync due to I/O errors,
1219          * and nothing changed in this transaction group,
1220          * and the vdev configuration hasn't changed,
1221          * then there's nothing to do.
1222          */
1223         if (ub->ub_txg < txg &&
1224             uberblock_update(ub, spa->spa_root_vdev, txg) == B_FALSE &&
1225             list_is_empty(&spa->spa_config_dirty_list))
1226                 return (0);
1227
1228         if (txg > spa_freeze_txg(spa))
1229                 return (0);
1230
1231         ASSERT(txg <= spa->spa_final_txg);
1232
1233         /*
1234          * Flush the write cache of every disk that's been written to
1235          * in this transaction group.  This ensures that all blocks
1236          * written in this txg will be committed to stable storage
1237          * before any uberblock that references them.
1238          */
1239         zio = zio_root(spa, NULL, NULL, flags);
1240
1241         for (vd = txg_list_head(&spa->spa_vdev_txg_list, TXG_CLEAN(txg)); vd;
1242             vd = txg_list_next(&spa->spa_vdev_txg_list, vd, TXG_CLEAN(txg)))
1243                 zio_flush(zio, vd);
1244
1245         (void) zio_wait(zio);
1246
1247         /*
1248          * Sync out the even labels (L0, L2) for every dirty vdev.  If the
1249          * system dies in the middle of this process, that's OK: all of the
1250          * even labels that made it to disk will be newer than any uberblock,
1251          * and will therefore be considered invalid.  The odd labels (L1, L3),
1252          * which have not yet been touched, will still be valid.  We flush
1253          * the new labels to disk to ensure that all even-label updates
1254          * are committed to stable storage before the uberblock update.
1255          */
1256         if ((error = vdev_label_sync_list(spa, 0, txg, flags)) != 0)
1257                 return (error);
1258
1259         /*
1260          * Sync the uberblocks to all vdevs in svd[].
1261          * If the system dies in the middle of this step, there are two cases
1262          * to consider, and the on-disk state is consistent either way:
1263          *
1264          * (1)  If none of the new uberblocks made it to disk, then the
1265          *      previous uberblock will be the newest, and the odd labels
1266          *      (which had not yet been touched) will be valid with respect
1267          *      to that uberblock.
1268          *
1269          * (2)  If one or more new uberblocks made it to disk, then they
1270          *      will be the newest, and the even labels (which had all
1271          *      been successfully committed) will be valid with respect
1272          *      to the new uberblocks.
1273          */
1274         if ((error = vdev_uberblock_sync_list(svd, svdcount, ub, flags)) != 0)
1275                 return (error);
1276
1277         /*
1278          * Sync out odd labels for every dirty vdev.  If the system dies
1279          * in the middle of this process, the even labels and the new
1280          * uberblocks will suffice to open the pool.  The next time
1281          * the pool is opened, the first thing we'll do -- before any
1282          * user data is modified -- is mark every vdev dirty so that
1283          * all labels will be brought up to date.  We flush the new labels
```

```
1284          * to disk to ensure that all odd-label updates are committed to
1285          * stable storage before the next transaction group begins.
1286          */
1287         return (vdev_label_sync_list(spa, 1, txg, flags));
1288 }
```

```
**********************************************************
   64446 Tue Apr 23 14:09:38 2013
new/usr/src/uts/common/fs/zfs/vdev_raidz.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
 434 /*
 435  * Divides the IO evenly across all child vdevs; usually, dcols is
 436  * the number of children in the target vdev.
 437  */
 438 #endif /* ! codereview */
 439 static raidz_map_t *
 440 vdev_raidz_map_alloc(zio_t *zio, uint64_t unit_shift, uint64_t dcols,
 441     uint64_t nparity)
 442 {
 443         raidz_map_t *rm;
 444         /* The starting RAIDZ (parent) vdev sector of the block. */
 445 #endif /* ! codereview */
 446         uint64_t b = zio->io_offset >> unit_shift;
 447         /* The zio's size in units of the vdev's minimum sector size */
 448 #endif /* ! codereview */
 449         uint64_t s = zio->io_size >> unit_shift;
 450         /* The first column for this stripe. */
 451 #endif /* ! codereview */
 452         uint64_t f = b % dcols;
 453         /* The starting byte offset on each child vdev. */
 454 #endif /* ! codereview */
 455         uint64_t o = (b / dcols) << unit_shift;
 456         uint64_t q, r, c, bc, col, acols, scols, coff, devidx, asize, tot;

 458         /*
 459          * "Quotient": The number of data sectors for this stripe on all but
 460          * the "big column" child vdevs that also contain "remainder" data.
 461          */
 462 #endif /* ! codereview */
 463         q = s / (dcols - nparity);

 465         /*
 466          * "Remainder": The number of partial stripe data sectors in this I/O.
 467          * This will add a sector to some, but not all, child vdevs.
 468          */
 469 #endif /* ! codereview */
 470         r = s - q * (dcols - nparity);

 472         /* The number of "big columns" - those which contain remainder data. */
 473 #endif /* ! codereview */
 474         bc = (r == 0 ? 0 : r + nparity);

 476         /*
 477          * The total number of data and parity sectors associated with
 478          * this I/O.
 479          */
 480 #endif /* ! codereview */
 481         tot = s + nparity * (q + (r == 0 ? 0 : 1));

 483         /* acols: The columns that will be accessed. */
 484         /* scols: The columns that will be accessed or skipped. */
 485 #endif /* ! codereview */
 486         if (q == 0) {
 487                 /* Our I/O request doesn't span all child vdevs. */
 488 #endif /* ! codereview */
```



```
 489                 acols = bc;
 490                 scols = MIN(dcols, roundup(bc, nparity + 1));
 491         } else {
 492                 acols = dcols;
 493                 scols = dcols;
 494         }

 496         ASSERT3U(acols, <=, scols);

 498         rm = kmem_alloc(offsetof(raidz_map_t, rm_col[scols]), KM_SLEEP);

 500         rm->rm_cols = acols;
 501         rm->rm_scols = scols;
 502         rm->rm_bigcols = bc;
 503         rm->rm_skipstart = bc;
 504         rm->rm_missingdata = 0;
 505         rm->rm_missingparity = 0;
 506         rm->rm_firstdatacol = nparity;
 507         rm->rm_datacopy = NULL;
 508         rm->rm_reports = 0;
 509         rm->rm_freed = 0;
 510         rm->rm_ecksuminjected = 0;

 512         asize = 0;

 514         for (c = 0; c < scols; c++) {
 515                 col = f + c;
 516                 coff = o;
 517                 if (col >= dcols) {
 518                         col -= dcols;
 519                         coff += 1ULL << unit_shift;
 520                 }
 521                 rm->rm_col[c].rc_devidx = col;
 522                 rm->rm_col[c].rc_offset = coff;
 523                 rm->rm_col[c].rc_data = NULL;
 524                 rm->rm_col[c].rc_gdata = NULL;
 525                 rm->rm_col[c].rc_error = 0;
 526                 rm->rm_col[c].rc_tried = 0;
 527                 rm->rm_col[c].rc_skipped = 0;

 529                 if (c >= acols)
 530                         rm->rm_col[c].rc_size = 0;
 531                 else if (c < bc)
 532                         rm->rm_col[c].rc_size = (q + 1) << unit_shift;
 533                 else
 534                         rm->rm_col[c].rc_size = q << unit_shift;

 536                 asize += rm->rm_col[c].rc_size;
 537         }

 539         ASSERT3U(asize, ==, tot << unit_shift);
 540         rm->rm_asize = roundup(asize, (nparity + 1) << unit_shift);
 541         rm->rm_nskip = roundup(tot, nparity + 1) - tot;
 542         ASSERT3U(rm->rm_asize - asize, ==, rm->rm_nskip << unit_shift);
 543         ASSERT3U(rm->rm_nskip, <=, nparity);

 545         for (c = 0; c < rm->rm_firstdatacol; c++)
 546                 rm->rm_col[c].rc_data = zio_buf_alloc(rm->rm_col[c].rc_size);

 548         rm->rm_col[c].rc_data = zio->io_data;

 550         for (c = c + 1; c < acols; c++)
 551                 rm->rm_col[c].rc_data = (char *)rm->rm_col[c - 1].rc_data +
 552                     rm->rm_col[c - 1].rc_size;

 554         /*
```

```
555          * If all data stored spans all columns, there's a danger that parity
556          * will always be on the same device and, since parity isn't read
557          * during normal operation, that that device's I/O bandwidth won't be
558          * used effectively. We therefore switch the parity every 1MB.
559          *
560          * ... at least that was, ostensibly, the theory. As a practical
561          * matter unless we juggle the parity between all devices evenly, we
562          * won't see any benefit. Further, occasional writes that aren't a
563          * multiple of the LCM of the number of children and the minimum
564          * stripe width are sufficient to avoid pessimal behavior.
565          * Unfortunately, this decision created an implicit on-disk format
566          * requirement that we need to support for all eternity, but only
567          * for single-parity RAID-Z.
568          *
569          * If we intend to skip a sector in the zeroth column for padding
570          * we must make sure to note this swap. We will never intend to
571          * skip the first column since at least one data and one parity
572          * column must appear in each row.
573          */
574         ASSERT(rm->rm_cols >= 2);
575         ASSERT(rm->rm_col[0].rc_size == rm->rm_col[1].rc_size);

577         if (rm->rm_firstdatacol == 1 && (zio->io_offset & (1ULL << 20))) {
578                 devidx = rm->rm_col[0].rc_devidx;
579                 o = rm->rm_col[0].rc_offset;
580                 rm->rm_col[0].rc_devidx = rm->rm_col[1].rc_devidx;
581                 rm->rm_col[0].rc_offset = rm->rm_col[1].rc_offset;
582                 rm->rm_col[1].rc_devidx = devidx;
583                 rm->rm_col[1].rc_offset = o;

585                 if (rm->rm_skipstart == 0)
586                         rm->rm_skipstart = 1;
587         }

589         zio->io_vsd = rm;
590         zio->io_vsd_ops = &vdev_raidz_vsd_ops;
591         return (rm);
592 }

594 static void
595 vdev_raidz_generate_parity_p(raidz_map_t *rm)
596 {
597         uint64_t *p, *src, pcount, ccount, i;
598         int c;

600         pcount = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);

602         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
603                 src = rm->rm_col[c].rc_data;
604                 p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
605                 ccount = rm->rm_col[c].rc_size / sizeof (src[0]);

607                 if (c == rm->rm_firstdatacol) {
608                         ASSERT(ccount == pcount);
609                         for (i = 0; i < ccount; i++, src++, p++) {
610                                 *p = *src;
611                         }
612                 } else {
613                         ASSERT(ccount <= pcount);
614                         for (i = 0; i < ccount; i++, src++, p++) {
615                                 *p ^= *src;
616                         }
617                 }
618         }
619 }
```



```
621 static void
622 vdev_raidz_generate_parity_pq(raidz_map_t *rm)
623 {
624         uint64_t *p, *q, *src, pcnt, ccnt, mask, i;
625         int c;

627         pcnt = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);
628         ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
629             rm->rm_col[VDEV_RAIDZ_Q].rc_size);

631         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
632                 src = rm->rm_col[c].rc_data;
633                 p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
634                 q = rm->rm_col[VDEV_RAIDZ_Q].rc_data;

636                 ccnt = rm->rm_col[c].rc_size / sizeof (src[0]);

638                 if (c == rm->rm_firstdatacol) {
639                         ASSERT(ccnt == pcnt || ccnt == 0);
640                         for (i = 0; i < ccnt; i++, src++, p++, q++) {
641                                 *p = *src;
642                                 *q = *src;
643                         }
644                         for (; i < pcnt; i++, src++, p++, q++) {
645                                 *p = 0;
646                                 *q = 0;
647                         }
648                 } else {
649                         ASSERT(ccnt <= pcnt);

651                         /*
652                          * Apply the algorithm described above by multiplying
653                          * the previous result and adding in the new value.
654                          */
655                         for (i = 0; i < ccnt; i++, src++, p++, q++) {
656                                 *p ^= *src;

658                                 VDEV_RAIDZ_64MUL_2(*q, mask);
659                                 *q ^= *src;
660                         }

662                         /*
663                          * Treat short columns as though they are full of 0s.
664                          * Note that there's therefore nothing needed for P.
665                          */
666                         for (; i < pcnt; i++, q++) {
667                                 VDEV_RAIDZ_64MUL_2(*q, mask);
668                         }
669                 }
670         }
671 }

673 static void
674 vdev_raidz_generate_parity_pqr(raidz_map_t *rm)
675 {
676         uint64_t *p, *q, *r, *src, pcnt, ccnt, mask, i;
677         int c;

679         pcnt = rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]);
680         ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
681             rm->rm_col[VDEV_RAIDZ_Q].rc_size);
682         ASSERT(rm->rm_col[VDEV_RAIDZ_P].rc_size ==
683             rm->rm_col[VDEV_RAIDZ_R].rc_size);

685         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
686                 src = rm->rm_col[c].rc_data;
```

```
687                 p = rm->rm_col[VDEV_RAIDZ_P].rc_data;
688                 q = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
689                 r = rm->rm_col[VDEV_RAIDZ_R].rc_data;

691                 ccnt = rm->rm_col[c].rc_size / sizeof (src[0]);

693                 if (c == rm->rm_firstdatacol) {
694                         ASSERT(ccnt == pcnt || ccnt == 0);
695                         for (i = 0; i < ccnt; i++, src++, p++, q++, r++) {
696                                 *p = *src;
697                                 *q = *src;
698                                 *r = *src;
699                         }
700                         for (; i < pcnt; i++, src++, p++, q++, r++) {
701                                 *p = 0;
702                                 *q = 0;
703                                 *r = 0;
704                         }
705                 } else {
706                         ASSERT(ccnt <= pcnt);

708                         /*
709                          * Apply the algorithm described above by multiplying
710                          * the previous result and adding in the new value.
711                          */
712                         for (i = 0; i < ccnt; i++, src++, p++, q++, r++) {
713                                 *p ^= *src;

715                                 VDEV_RAIDZ_64MUL_2(*q, mask);
716                                 *q ^= *src;

718                                 VDEV_RAIDZ_64MUL_4(*r, mask);
719                                 *r ^= *src;
720                         }

722                         /*
723                          * Treat short columns as though they are full of 0s.
724                          * Note that there's therefore nothing needed for P.
725                          */
726                         for (; i < pcnt; i++, q++, r++) {
727                                 VDEV_RAIDZ_64MUL_2(*q, mask);
728                                 VDEV_RAIDZ_64MUL_4(*r, mask);
729                         }
730                 }
731         }
732 }

734 /*
735  * Generate RAID parity in the first virtual columns according to the number of
736  * parity columns available.
737  */
738 static void
739 vdev_raidz_generate_parity(raidz_map_t *rm)
740 {
741         switch (rm->rm_firstdatacol) {
742         case 1:
743                 vdev_raidz_generate_parity_p(rm);
744                 break;
745         case 2:
746                 vdev_raidz_generate_parity_pq(rm);
747                 break;
748         case 3:
749                 vdev_raidz_generate_parity_pqr(rm);
750                 break;
751         default:
752                 cmn_err(CE_PANIC, "invalid RAID-Z configuration");
```



```
753         }
754 }

756 static int
757 vdev_raidz_reconstruct_p(raidz_map_t *rm, int *tgts, int ntgts)
758 {
759         uint64_t *dst, *src, xcount, ccount, count, i;
760         int x = tgts[0];
761         int c;

763         ASSERT(ntgts == 1);
764         ASSERT(x >= rm->rm_firstdatacol);
765         ASSERT(x < rm->rm_cols);

767         xcount = rm->rm_col[x].rc_size / sizeof (src[0]);
768         ASSERT(xcount <= rm->rm_col[VDEV_RAIDZ_P].rc_size / sizeof (src[0]));
769         ASSERT(xcount > 0);

771         src = rm->rm_col[VDEV_RAIDZ_P].rc_data;
772         dst = rm->rm_col[x].rc_data;
773         for (i = 0; i < xcount; i++, dst++, src++) {
774                 *dst = *src;
775         }

777         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
778                 src = rm->rm_col[c].rc_data;
779                 dst = rm->rm_col[x].rc_data;

781                 if (c == x)
782                         continue;

784                 ccount = rm->rm_col[c].rc_size / sizeof (src[0]);
785                 count = MIN(ccount, xcount);

787                 for (i = 0; i < count; i++, dst++, src++) {
788                         *dst ^= *src;
789                 }
790         }

792         return (1 << VDEV_RAIDZ_P);
793 }

795 static int
796 vdev_raidz_reconstruct_q(raidz_map_t *rm, int *tgts, int ntgts)
797 {
798         uint64_t *dst, *src, xcount, ccount, count, mask, i;
799         uint8_t *b;
800         int x = tgts[0];
801         int c, j, exp;

803         ASSERT(ntgts == 1);

805         xcount = rm->rm_col[x].rc_size / sizeof (src[0]);
806         ASSERT(xcount <= rm->rm_col[VDEV_RAIDZ_Q].rc_size / sizeof (src[0]));

808         for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
809                 src = rm->rm_col[c].rc_data;
810                 dst = rm->rm_col[x].rc_data;

812                 if (c == x)
813                         ccount = 0;
814                 else
815                         ccount = rm->rm_col[c].rc_size / sizeof (src[0]);

817                 count = MIN(ccount, xcount);
```

```
819                 if (c == rm->rm_firstdatacol) {
820                         for (i = 0; i < count; i++, dst++, src++) {
821                                 *dst = *src;
822                         }
823                         for (; i < xcount; i++, dst++) {
824                                 *dst = 0;
825                         }
826
827                 } else {
828                         for (i = 0; i < count; i++, dst++, src++) {
829                                 VDEV_RAIDZ_64MUL_2(*dst, mask);
830                                 *dst ^= *src;
831                         }
832
833                         for (; i < xcount; i++, dst++) {
834                                 VDEV_RAIDZ_64MUL_2(*dst, mask);
835                         }
836                 }
837         }
838
839         src = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
840         dst = rm->rm_col[x].rc_data;
841         exp = 255 - (rm->rm_cols - 1 - x);
842
843         for (i = 0; i < xcount; i++, dst++, src++) {
844                 *dst ^= *src;
845                 for (j = 0, b = (uint8_t *)dst; j < 8; j++, b++) {
846                         *b = vdev_raidz_exp2(*b, exp);
847                 }
848         }
849
850         return (1 << VDEV_RAIDZ_Q);
851 }
852
853 static int
854 vdev_raidz_reconstruct_pq(raidz_map_t *rm, int *tgts, int ntgts)
855 {
856         uint8_t *p, *q, *pxy, *qxy, *xd, *yd, tmp, a, b, aexp, bexp;
857         void *pdata, *qdata;
858         uint64_t xsize, ysize, i;
859         int x = tgts[0];
860         int y = tgts[1];
861
862         ASSERT(ntgts == 2);
863         ASSERT(x < y);
864         ASSERT(x >= rm->rm_firstdatacol);
865         ASSERT(y < rm->rm_cols);
866
867         ASSERT(rm->rm_col[x].rc_size >= rm->rm_col[y].rc_size);
868
869         /*
870          * Move the parity data aside -- we're going to compute parity as
871          * though columns x and y were full of zeros -- Pxy and Qxy. We want to
872          * reuse the parity generation mechanism without trashing the actual
873          * parity so we make those columns appear to be full of zeros by
874          * setting their lengths to zero.
875          */
876         pdata = rm->rm_col[VDEV_RAIDZ_P].rc_data;
877         qdata = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
878         xsize = rm->rm_col[x].rc_size;
879         ysize = rm->rm_col[y].rc_size;
880
881         rm->rm_col[VDEV_RAIDZ_P].rc_data =
882             zio_buf_alloc(rm->rm_col[VDEV_RAIDZ_P].rc_size);
883         rm->rm_col[VDEV_RAIDZ_Q].rc_data =
884             zio_buf_alloc(rm->rm_col[VDEV_RAIDZ_Q].rc_size);
```



```
885         rm->rm_col[x].rc_size = 0;
886         rm->rm_col[y].rc_size = 0;
887
888         vdev_raidz_generate_parity_pq(rm);
889
890         rm->rm_col[x].rc_size = xsize;
891         rm->rm_col[y].rc_size = ysize;
892
893         p = pdata;
894         q = qdata;
895         pxy = rm->rm_col[VDEV_RAIDZ_P].rc_data;
896         qxy = rm->rm_col[VDEV_RAIDZ_Q].rc_data;
897         xd = rm->rm_col[x].rc_data;
898         yd = rm->rm_col[y].rc_data;
899
900         /*
901          * We now have:
902          *      Pxy = P + D_x + D_y
903          *      Qxy = Q + 2^(ndevs - 1 - x) * D_x + 2^(ndevs - 1 - y) * D_y
904          *
905          * We can then solve for D_x:
906          *      D_x = A * (P + Pxy) + B * (Q + Qxy)
907          * where
908          *      A = 2^(x - y) * (2^(x - y) + 1)^-1
909          *      B = 2^(ndevs - 1 - x) * (2^(x - y) + 1)^-1
910          *
911          * With D_x in hand, we can easily solve for D_y:
912          *      D_y = P + Pxy + D_x
913          */
914
915         a = vdev_raidz_pow2[255 + x - y];
916         b = vdev_raidz_pow2[255 - (rm->rm_cols - 1 - x)];
917         tmp = 255 - vdev_raidz_log2[a ^ 1];
918
919         aexp = vdev_raidz_log2[vdev_raidz_exp2(a, tmp)];
920         bexp = vdev_raidz_log2[vdev_raidz_exp2(b, tmp)];
921
922         for (i = 0; i < xsize; i++, p++, q++, pxy++, qxy++, xd++, yd++) {
923                 *xd = vdev_raidz_exp2(*p ^ *pxy, aexp) ^
924                     vdev_raidz_exp2(*q ^ *qxy, bexp);
925
926                 if (i < ysize)
927                         *yd = *p ^ *pxy ^ *xd;
928         }
929
930         zio_buf_free(rm->rm_col[VDEV_RAIDZ_P].rc_data,
931             rm->rm_col[VDEV_RAIDZ_P].rc_size);
932         zio_buf_free(rm->rm_col[VDEV_RAIDZ_Q].rc_data,
933             rm->rm_col[VDEV_RAIDZ_Q].rc_size);
934
935         /*
936          * Restore the saved parity data.
937          */
938         rm->rm_col[VDEV_RAIDZ_P].rc_data = pdata;
939         rm->rm_col[VDEV_RAIDZ_Q].rc_data = qdata;
940
941         return ((1 << VDEV_RAIDZ_P) | (1 << VDEV_RAIDZ_Q));
942 }
943
944 /* BEGIN CSTYLED */
945 /*
946  * In the general case of reconstruction, we must solve the system of linear
947  * equations defined by the coeffecients used to generate parity as well as
948  * the contents of the data and parity disks. This can be expressed with
949  * vectors for the original data (D) and the actual data (d) and parity (p)
950  * and a matrix composed of the identity matrix (I) and a dispersal matrix (V):
```

```
 *
 *            __   __                     __     __
 *            |     |            __   __  |  p_0  |
 *            |  V  |            |  D_0  | |  p_m-1 |
 *            |     |     x      |   :   | =  d_0   |
 *            |  I  |            | D_n-1 |    :     |
 *            |     |            ~~     ~~  | d_n-1 |
 *            ~~   ~~                      ~~     ~~
 *
 * I is simply a square identity matrix of size n, and V is a vandermonde
 * matrix defined by the coeffecients we chose for the various parity columns
 * (1, 2, 4). Note that these values were chosen both for simplicity, speedy
 * computation as well as linear separability.
 *
 *      __               __               __     __
 *      |   1   ..  1 1 1 |            __   __  |  p_0  |
 *      | 2^n-1 ..  4 2 1 |            |  D_0  | |  p_m-1 |
 *      | 4^n-1 .. 16 4 1 |     x      |  D_1  | =  d_0   |
 *      |   1   ..  0 0 0 |            |  D_2  | |  d_1   |
 *      |   :       : : : |            |   :   | |  d_2   |
 *      |   0   ..  1 0 0 |            | D_n-1 | |   :    |
 *      |   0   ..  0 1 0 |            ~~     ~~ |   :    |
 *      |   0   ..  0 0 1 |                      | d_n-1 |
 *      ~~               ~~                     ~~     ~~
 *
 * Note that I, V, d, and p are known. To compute D, we must invert the
 * matrix and use the known data and parity values to reconstruct the unknown
 * data values. We begin by removing the rows in V|I and d|p that correspond
 * to failed or missing columns; we then make V|I square (n x n) and d|p
 * sized n by removing rows corresponding to unused parity from the bottom up
 * to generate (V|I)' and (d|p)'. We can then generate the inverse of (V|I)'
 * using Gauss-Jordan elimination. In the example below we use m=3 parity
 * columns, n=8 data columns, with errors in d_1, d_2, and p_1:
 *           __                               __
 *           |  1   1   1   1   1   1   1   1  |
 *           | 128  64  32  16  8   4   2   1  | <-----+-+-- missing disks
 *           |  19 205 116  29  64  16  4   1  |      / /
 *           |  1   0   0   0   0   0   0   0  |     / /
 *           |  0   1   0   0   0   0   0   0  | <--' /
 *  (V|I)  = |  0   0   1   0   0   0   0   0  | <---'
 *           |  0   0   0   1   0   0   0   0  |
 *           |  0   0   0   0   1   0   0   0  |
 *           |  0   0   0   0   0   1   0   0  |
 *           |  0   0   0   0   0   0   1   0  |
 *           |  0   0   0   0   0   0   0   1  |
 *           ~~                               ~~
 *           __                               __
 *           |  1   1   1   1   1   1   1   1  |
 *           | 128  64  32  16  8   4   2   1  |
 *           |  19 205 116  29  64  16  4   1  |
 *           |  1   0   0   0   0   0   0   0  |
 *           |  0   1   0   0   0   0   0   0  |
 *  (V|I)' = |  0   0   1   0   0   0   0   0  |
 *           |  0   0   0   1   0   0   0   0  |
 *           |  0   0   0   0   1   0   0   0  |
 *           |  0   0   0   0   0   1   0   0  |
 *           |  0   0   0   0   0   0   1   0  |
 *           |  0   0   0   0   0   0   0   1  |
 *           ~~                               ~~
 *
 * Here we employ Gauss-Jordan elimination to find the inverse of (V|I)'. We
 * have carefully chosen the seed values 1, 2, and 4 to ensure that this
 * matrix is not singular.
 * __                                                               __
 * |  1   1   1   1   1   1   1   1     1   0   0   0   0   0   0   0 |
```



```
 * |  19 205 116  29  64  16  4   1     0   1   0   0   0   0   0   0 |
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 * __                                                               __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  1   1   1   1   1   1   1   1     1   0   0   0   0   0   0   0 |
 * |  19 205 116  29  64  16  4   1     0   1   0   0   0   0   0   0 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 * __                                                               __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1 |
 * |  0  205 116  0   0   0   0   0     0   1   19  29  64  16  4   1 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 * __                                                               __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1 |
 * |  0   0  185  0   0   0   0   0    205  1  222 208 141 221 201 204 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 * __                                                               __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  0   1   1   0   0   0   0   0     1   0   1   1   1   1   1   1 |
 * |  0   0   1   0   0   0   0   0   166 100  4   40 158 168 216 209 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 * __                                                               __
 * |  1   0   0   0   0   0   0   0     0   0   1   0   0   0   0   0 |
 * |  0   1   0   0   0   0   0   0   167 100  5   41 159 169 217 208 |
 * |  0   0   1   0   0   0   0   0   166 100  4   40 158 168 216 209 |
 * |  0   0   0   1   0   0   0   0     0   0   0   1   0   0   0   0 |
 * |  0   0   0   0   1   0   0   0     0   0   0   0   1   0   0   0 |
 * |  0   0   0   0   0   1   0   0     0   0   0   0   0   1   0   0 |
 * |  0   0   0   0   0   0   1   0     0   0   0   0   0   0   1   0 |
 * |  0   0   0   0   0   0   0   1     0   0   0   0   0   0   0   1 |
 * ~~                                                               ~~
 *
 *                                 __                           __
 *                                 |  0   0   1   0   0   0   0   0 |
 *                                 | 167 100  5   41 159 169 217 208 |
 *                                 | 166 100  4   40 158 168 216 209 |
 *                      (V|I)'^-1 = |  0   0   0   1   0   0   0   0 |
 *                                 |  0   0   0   0   1   0   0   0 |
 *                                 |  0   0   0   0   0   1   0   0 |
 *                                 |  0   0   0   0   0   0   1   0 |
```

```
 *                   |  0   0   0   0   0   0   0   1  |
 *                   ~~                                ~~
 *
 * We can then simply compute D = (V|I)'^-1 x (d|p)' to discover the values
 * of the missing data.
 *
 * As is apparent from the example above, the only non-trivial rows in the
 * inverse matrix correspond to the data disks that we're trying to
 * reconstruct. Indeed, those are the only rows we need as the others would
 * only be useful for reconstructing data known or assumed to be valid. For
 * that reason, we only build the coefficients in the rows that correspond to
 * targeted columns.
 */
/* END CSTYLED */

static void
vdev_raidz_matrix_init(raidz_map_t *rm, int n, int nmap, int *map,
    uint8_t **rows)
{
        int i, j;
        int pow;

        ASSERT(n == rm->rm_cols - rm->rm_firstdatacol);

        /*
         * Fill in the missing rows of interest.
         */
        for (i = 0; i < nmap; i++) {
                ASSERT3S(0, <=, map[i]);
                ASSERT3S(map[i], <=, 2);

                pow = map[i] * n;
                if (pow > 255)
                        pow -= 255;
                ASSERT(pow <= 255);

                for (j = 0; j < n; j++) {
                        pow -= map[i];
                        if (pow < 0)
                                pow += 255;
                        rows[i][j] = vdev_raidz_pow2[pow];
                }
        }
}

static void
vdev_raidz_matrix_invert(raidz_map_t *rm, int n, int nmissing, int *missing,
    uint8_t **rows, uint8_t **invrows, const uint8_t *used)
{
        int i, j, ii, jj;
        uint8_t log;

        /*
         * Assert that the first nmissing entries from the array of used
         * columns correspond to parity columns and that subsequent entries
         * correspond to data columns.
         */
        for (i = 0; i < nmissing; i++) {
                ASSERT3S(used[i], <, rm->rm_firstdatacol);
        }
        for (; i < n; i++) {
                ASSERT3S(used[i], >=, rm->rm_firstdatacol);
        }

        /*
         * First initialize the storage where we'll compute the inverse rows.
```



```
         */
        for (i = 0; i < nmissing; i++) {
                for (j = 0; j < n; j++) {
                        invrows[i][j] = (i == j) ? 1 : 0;
                }
        }

        /*
         * Subtract all trivial rows from the rows of consequence.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = nmissing; j < n; j++) {
                        ASSERT3U(used[j], >=, rm->rm_firstdatacol);
                        jj = used[j] - rm->rm_firstdatacol;
                        ASSERT3S(jj, <, n);
                        invrows[i][j] = rows[i][jj];
                        rows[i][jj] = 0;
                }
        }

        /*
         * For each of the rows of interest, we must normalize it and subtract
         * a multiple of it from the other rows.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = 0; j < missing[i]; j++) {
                        ASSERT0(rows[i][j]);
                }
                ASSERT3U(rows[i][missing[i]], !=, 0);

                /*
                 * Compute the inverse of the first element and multiply each
                 * element in the row by that value.
                 */
                log = 255 - vdev_raidz_log2[rows[i][missing[i]]];

                for (j = 0; j < n; j++) {
                        rows[i][j] = vdev_raidz_exp2(rows[i][j], log);
                        invrows[i][j] = vdev_raidz_exp2(invrows[i][j], log);
                }

                for (ii = 0; ii < nmissing; ii++) {
                        if (i == ii)
                                continue;

                        ASSERT3U(rows[ii][missing[i]], !=, 0);

                        log = vdev_raidz_log2[rows[ii][missing[i]]];

                        for (j = 0; j < n; j++) {
                                rows[ii][j] ^=
                                    vdev_raidz_exp2(rows[i][j], log);
                                invrows[ii][j] ^=
                                    vdev_raidz_exp2(invrows[i][j], log);
                        }
                }
        }

        /*
         * Verify that the data that is left in the rows are properly part of
         * an identity matrix.
         */
        for (i = 0; i < nmissing; i++) {
                for (j = 0; j < n; j++) {
                        if (j == missing[i]) {
                                ASSERT3U(rows[i][j], ==, 1);
```

```
1215                        } else {
1216                                ASSERT0(rows[i][j]);
1217                        }
1218                }
1219        }
1220 }

1222 static void
1223 vdev_raidz_matrix_reconstruct(raidz_map_t *rm, int n, int nmissing,
1224     int *missing, uint8_t **invrows, const uint8_t *used)
1225 {
1226         int i, j, x, cc, c;
1227         uint8_t *src;
1228         uint64_t ccount;
1229         uint8_t *dst[VDEV_RAIDZ_MAXPARITY];
1230         uint64_t dcount[VDEV_RAIDZ_MAXPARITY];
1231         uint8_t log = 0;
1232         uint8_t val;
1233         int ll;
1234         uint8_t *invlog[VDEV_RAIDZ_MAXPARITY];
1235         uint8_t *p, *pp;
1236         size_t psize;

1238         psize = sizeof (invlog[0][0]) * n * nmissing;
1239         p = kmem_alloc(psize, KM_SLEEP);

1241         for (pp = p, i = 0; i < nmissing; i++) {
1242                 invlog[i] = pp;
1243                 pp += n;
1244         }

1246         for (i = 0; i < nmissing; i++) {
1247                 for (j = 0; j < n; j++) {
1248                         ASSERT3U(invrows[i][j], !=, 0);
1249                         invlog[i][j] = vdev_raidz_log2[invrows[i][j]];
1250                 }
1251         }

1253         for (i = 0; i < n; i++) {
1254                 c = used[i];
1255                 ASSERT3U(c, <, rm->rm_cols);

1257                 src = rm->rm_col[c].rc_data;
1258                 ccount = rm->rm_col[c].rc_size;
1259                 for (j = 0; j < nmissing; j++) {
1260                         cc = missing[j] + rm->rm_firstdatacol;
1261                         ASSERT3U(cc, >=, rm->rm_firstdatacol);
1262                         ASSERT3U(cc, <, rm->rm_cols);
1263                         ASSERT3U(cc, !=, c);

1265                         dst[j] = rm->rm_col[cc].rc_data;
1266                         dcount[j] = rm->rm_col[cc].rc_size;
1267                 }

1269                 ASSERT(ccount >= rm->rm_col[missing[0]].rc_size || i > 0);

1271                 for (x = 0; x < ccount; x++, src++) {
1272                         if (*src != 0)
1273                                 log = vdev_raidz_log2[*src];

1275                         for (cc = 0; cc < nmissing; cc++) {
1276                                 if (x >= dcount[cc])
1277                                         continue;

1279                                 if (*src == 0) {
1280                                         val = 0;
```



```
1281                                 } else {
1282                                         if ((ll = log + invlog[cc][i]) >= 255)
1283                                                 ll -= 255;
1284                                         val = vdev_raidz_pow2[ll];
1285                                 }

1287                                 if (i == 0)
1288                                         dst[cc][x] = val;
1289                                 else
1290                                         dst[cc][x] ^= val;
1291                         }
1292                 }
1293         }

1295         kmem_free(p, psize);
1296 }

1298 static int
1299 vdev_raidz_reconstruct_general(raidz_map_t *rm, int *tgts, int ntgts)
1300 {
1301         int n, i, c, t, tt;
1302         int nmissing_rows;
1303         int missing_rows[VDEV_RAIDZ_MAXPARITY];
1304         int parity_map[VDEV_RAIDZ_MAXPARITY];

1306         uint8_t *p, *pp;
1307         size_t psize;

1309         uint8_t *rows[VDEV_RAIDZ_MAXPARITY];
1310         uint8_t *invrows[VDEV_RAIDZ_MAXPARITY];
1311         uint8_t *used;

1313         int code = 0;


1316         n = rm->rm_cols - rm->rm_firstdatacol;

1318         /*
1319          * Figure out which data columns are missing.
1320          */
1321         nmissing_rows = 0;
1322         for (t = 0; t < ntgts; t++) {
1323                 if (tgts[t] >= rm->rm_firstdatacol) {
1324                         missing_rows[nmissing_rows++] =
1325                             tgts[t] - rm->rm_firstdatacol;
1326                 }
1327         }

1329         /*
1330          * Figure out which parity columns to use to help generate the missing
1331          * data columns.
1332          */
1333         for (tt = 0, c = 0, i = 0; i < nmissing_rows; c++) {
1334                 ASSERT(tt < ntgts);
1335                 ASSERT(c < rm->rm_firstdatacol);

1337                 /*
1338                  * Skip any targeted parity columns.
1339                  */
1340                 if (c == tgts[tt]) {
1341                         tt++;
1342                         continue;
1343                 }

1345                 code |= 1 << c;
```

```
1347                parity_map[i] = c;
1348                i++;
1349        }

1351        ASSERT(code != 0);
1352        ASSERT3U(code, <, 1 << VDEV_RAIDZ_MAXPARITY);

1354        psize = (sizeof (rows[0][0]) + sizeof (invrows[0][0])) *
1355            nmissing_rows * n + sizeof (used[0]) * n;
1356        p = kmem_alloc(psize, KM_SLEEP);

1358        for (pp = p, i = 0; i < nmissing_rows; i++) {
1359                rows[i] = pp;
1360                pp += n;
1361                invrows[i] = pp;
1362                pp += n;
1363        }
1364        used = pp;

1366        for (i = 0; i < nmissing_rows; i++) {
1367                used[i] = parity_map[i];
1368        }

1370        for (tt = 0, c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
1371                if (tt < nmissing_rows &&
1372                    c == missing_rows[tt] + rm->rm_firstdatacol) {
1373                        tt++;
1374                        continue;
1375                }

1377                ASSERT3S(i, <, n);
1378                used[i] = c;
1379                i++;
1380        }

1382        /*
1383         * Initialize the interesting rows of the matrix.
1384         */
1385        vdev_raidz_matrix_init(rm, n, nmissing_rows, parity_map, rows);

1387        /*
1388         * Invert the matrix.
1389         */
1390        vdev_raidz_matrix_invert(rm, n, nmissing_rows, missing_rows, rows,
1391            invrows, used);

1393        /*
1394         * Reconstruct the missing data using the generated matrix.
1395         */
1396        vdev_raidz_matrix_reconstruct(rm, n, nmissing_rows, missing_rows,
1397            invrows, used);

1399        kmem_free(p, psize);

1401        return (code);
1402 }

1404 static int
1405 vdev_raidz_reconstruct(raidz_map_t *rm, int *t, int nt)
1406 {
1407        int tgts[VDEV_RAIDZ_MAXPARITY], *dt;
1408        int ntgts;
1409        int i, c;
1410        int code;
1411        int nbadparity, nbaddata;
1412        int parity_valid[VDEV_RAIDZ_MAXPARITY];
```



```
1414        /*
1415         * The tgts list must already be sorted.
1416         */
1417        for (i = 1; i < nt; i++) {
1418                ASSERT(t[i] > t[i - 1]);
1419        }

1421        nbadparity = rm->rm_firstdatacol;
1422        nbaddata = rm->rm_cols - nbadparity;
1423        ntgts = 0;
1424        for (i = 0, c = 0; c < rm->rm_cols; c++) {
1425                if (c < rm->rm_firstdatacol)
1426                        parity_valid[c] = B_FALSE;

1428                if (i < nt && c == t[i]) {
1429                        tgts[ntgts++] = c;
1430                        i++;
1431                } else if (rm->rm_col[c].rc_error != 0) {
1432                        tgts[ntgts++] = c;
1433                } else if (c >= rm->rm_firstdatacol) {
1434                        nbaddata--;
1435                } else {
1436                        parity_valid[c] = B_TRUE;
1437                        nbadparity--;
1438                }
1439        }

1441        ASSERT(ntgts >= nt);
1442        ASSERT(nbaddata >= 0);
1443        ASSERT(nbaddata + nbadparity == ntgts);

1445        dt = &tgts[nbadparity];

1447        /*
1448         * See if we can use any of our optimized reconstruction routines.
1449         */
1450        if (!vdev_raidz_default_to_general) {
1451                switch (nbaddata) {
1452                case 1:
1453                        if (parity_valid[VDEV_RAIDZ_P])
1454                                return (vdev_raidz_reconstruct_p(rm, dt, 1));

1456                        ASSERT(rm->rm_firstdatacol > 1);

1458                        if (parity_valid[VDEV_RAIDZ_Q])
1459                                return (vdev_raidz_reconstruct_q(rm, dt, 1));

1461                        ASSERT(rm->rm_firstdatacol > 2);
1462                        break;

1464                case 2:
1465                        ASSERT(rm->rm_firstdatacol > 1);

1467                        if (parity_valid[VDEV_RAIDZ_P] &&
1468                            parity_valid[VDEV_RAIDZ_Q])
1469                                return (vdev_raidz_reconstruct_pq(rm, dt, 2));

1471                        ASSERT(rm->rm_firstdatacol > 2);

1473                        break;
1474                }
1475        }

1477        code = vdev_raidz_reconstruct_general(rm, tgts, ntgts);
1478        ASSERT(code < (1 << VDEV_RAIDZ_MAXPARITY));
```

```
1479         ASSERT(code > 0);
1480         return (code);
1481 }

1483 static int
1484 vdev_raidz_open(vdev_t *vd, uint64_t *asize, uint64_t *max_asize,
1485     uint64_t *ashift)
1486 {
1487         vdev_t *cvd;
1488         uint64_t nparity = vd->vdev_nparity;
1489         int c;
1490         int lasterror = 0;
1491         int numerrors = 0;

1493         ASSERT(nparity > 0);

1495         if (nparity > VDEV_RAIDZ_MAXPARITY ||
1496             vd->vdev_children < nparity + 1) {
1497                 vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
1498                 return (SET_ERROR(EINVAL));
1499         }

1501         vdev_open_children(vd);

1503         for (c = 0; c < vd->vdev_children; c++) {
1504                 cvd = vd->vdev_child[c];

1506                 if (cvd->vdev_open_error != 0) {
1507                         lasterror = cvd->vdev_open_error;
1508                         numerrors++;
1509                         continue;
1510                 }

1512                 *asize = MIN(*asize - 1, cvd->vdev_asize - 1) + 1;
1513                 *max_asize = MIN(*max_asize - 1, cvd->vdev_max_asize - 1) + 1;
1514                 *ashift = MAX(*ashift, cvd->vdev_ashift);
1515         }

1517         *asize *= vd->vdev_children;
1518         *max_asize *= vd->vdev_children;

1520         if (numerrors > nparity) {
1521                 vd->vdev_stat.vs_aux = VDEV_AUX_NO_REPLICAS;
1522                 return (lasterror);
1523         }

1525         return (0);
1526 }

1528 static void
1529 vdev_raidz_close(vdev_t *vd)
1530 {
1531         int c;

1533         for (c = 0; c < vd->vdev_children; c++)
1534                 vdev_close(vd->vdev_child[c]);
1535 }

1537 static uint64_t
1538 vdev_raidz_asize(vdev_t *vd, uint64_t psize)
1539 {
1540         uint64_t asize;
1541         uint64_t ashift = vd->vdev_top->vdev_ashift;
1542         uint64_t cols = vd->vdev_children;
1543         uint64_t nparity = vd->vdev_nparity;
```



```
1545         asize = ((psize - 1) >> ashift) + 1;
1546         asize += nparity * ((asize + cols - nparity - 1) / (cols - nparity));
1547         asize = roundup(asize, nparity + 1) << ashift;

1549         return (asize);
1550 }

1552 static void
1553 vdev_raidz_child_done(zio_t *zio)
1554 {
1555         raidz_col_t *rc = zio->io_private;

1557         rc->rc_error = zio->io_error;
1558         rc->rc_tried = 1;
1559         rc->rc_skipped = 0;
1560 }

1562 /*
1563  * Start an IO operation on a RAIDZ VDev
1564  *
1565  * Outline:
1566  * - For write operations:
1567  *   1. Generate the parity data
1568  *   2. Create child zio write operations to each column's vdev, for both
1569  *      data and parity.
1570  *   3. If the column skips any sectors for padding, create optional dummy
1571  *      write zio children for those areas to improve aggregation continuity.
1572  * - For read operations:
1573  *   1. Create child zio read operations to each data column's vdev to read
1574  *      the range of data required for zio.
1575  *   2. If this is a scrub or resilver operation, or if any of the data
1576  *      vdevs have had errors, then create zio read operations to the parity
1577  *      columns' VDevs as well.
1578  */
1579 #endif /* ! codereview */
1580 static int
1581 vdev_raidz_io_start(zio_t *zio)
1582 {
1583         vdev_t *vd = zio->io_vd;
1584         vdev_t *tvd = vd->vdev_top;
1585         vdev_t *cvd;
1586         raidz_map_t *rm;
1587         raidz_col_t *rc;
1588         int c, i;

1590         rm = vdev_raidz_map_alloc(zio, tvd->vdev_ashift, vd->vdev_children,
1591             vd->vdev_nparity);

1593         ASSERT3U(rm->rm_asize, ==, vdev_psize_to_asize(vd, zio->io_size));

1595         if (zio->io_type == ZIO_TYPE_WRITE) {
1596                 vdev_raidz_generate_parity(rm);

1598                 for (c = 0; c < rm->rm_cols; c++) {
1599                         rc = &rm->rm_col[c];
1600                         cvd = vd->vdev_child[rc->rc_devidx];
1601                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1602                             rc->rc_offset, rc->rc_data, rc->rc_size,
1603                             zio->io_type, zio->io_priority, 0,
1604                             vdev_raidz_child_done, rc));
1605                 }

1607                 /*
1608                  * Generate optional I/Os for any skipped sectors to improve
1609                  * aggregation contiguity.
1610                  */
```

```
1611                 for (c = rm->rm_skipstart, i = 0; i < rm->rm_nskip; c++, i++) {
1612                         ASSERT(c <= rm->rm_scols);
1613                         if (c == rm->rm_scols)
1614                                 c = 0;
1615                         rc = &rm->rm_col[c];
1616                         cvd = vd->vdev_child[rc->rc_devidx];
1617                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1618                             rc->rc_offset + rc->rc_size, NULL,
1619                             1 << tvd->vdev_ashift,
1620                             zio->io_type, zio->io_priority,
1621                             ZIO_FLAG_NODATA | ZIO_FLAG_OPTIONAL, NULL, NULL));
1622                 }

1624                 return (ZIO_PIPELINE_CONTINUE);
1625         }

1627         ASSERT(zio->io_type == ZIO_TYPE_READ);

1629         /*
1630          * Iterate over the columns in reverse order so that we hit the parity
1631          * last -- any errors along the way will force us to read the parity.
1632          */
1633         for (c = rm->rm_cols - 1; c >= 0; c--) {
1634                 rc = &rm->rm_col[c];
1635                 cvd = vd->vdev_child[rc->rc_devidx];
1636                 if (!vdev_readable(cvd)) {
1637                         if (c >= rm->rm_firstdatacol)
1638                                 rm->rm_missingdata++;
1639                         else
1640                                 rm->rm_missingparity++;
1641                         rc->rc_error = SET_ERROR(ENXIO);
1642                         rc->rc_tried = 1;       /* don't even try */
1643                         rc->rc_skipped = 1;
1644                         continue;
1645                 }
1646                 if (vdev_dtl_contains(cvd, DTL_MISSING, zio->io_txg, 1)) {
1647                         if (c >= rm->rm_firstdatacol)
1648                                 rm->rm_missingdata++;
1649                         else
1650                                 rm->rm_missingparity++;
1651                         rc->rc_error = SET_ERROR(ESTALE);
1652                         rc->rc_skipped = 1;
1653                         continue;
1654                 }
1655                 if (c >= rm->rm_firstdatacol || rm->rm_missingdata > 0 ||
1656                     (zio->io_flags & (ZIO_FLAG_SCRUB | ZIO_FLAG_RESILVER))) {
1657                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
1658                             rc->rc_offset, rc->rc_data, rc->rc_size,
1659                             zio->io_type, zio->io_priority, 0,
1660                             vdev_raidz_child_done, rc));
1661                 }
1662         }

1664         return (ZIO_PIPELINE_CONTINUE);
1665 }


1668 /*
1669  * Report a checksum error for a child of a RAID-Z device.
1670  */
1671 static void
1672 raidz_checksum_error(zio_t *zio, raidz_col_t *rc, void *bad_data)
1673 {
1674         vdev_t *vd = zio->io_vd->vdev_child[rc->rc_devidx];

1676         if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
```



```
1677                 zio_bad_cksum_t zbc;
1678                 raidz_map_t *rm = zio->io_vsd;

1680                 mutex_enter(&vd->vdev_stat_lock);
1681                 vd->vdev_stat.vs_checksum_errors++;
1682                 mutex_exit(&vd->vdev_stat_lock);

1684                 zbc.zbc_has_cksum = 0;
1685                 zbc.zbc_injected = rm->rm_ecksuminjected;

1687                 zfs_ereport_post_checksum(zio->io_spa, vd, zio,
1688                     rc->rc_offset, rc->rc_size, rc->rc_data, bad_data,
1689                     &zbc);
1690         }
1691 }

1693 /*
1694  * We keep track of whether or not there were any injected errors, so that
1695  * any ereports we generate can note it.
1696  */
1697 static int
1698 raidz_checksum_verify(zio_t *zio)
1699 {
1700         zio_bad_cksum_t zbc;
1701         raidz_map_t *rm = zio->io_vsd;

1703         int ret = zio_checksum_error(zio, &zbc);
1704         if (ret != 0 && zbc.zbc_injected != 0)
1705                 rm->rm_ecksuminjected = 1;

1707         return (ret);
1708 }

1710 /*
1711  * Generate the parity from the data columns. If we tried and were able to
1712  * read the parity without error, verify that the generated parity matches the
1713  * data we read. If it doesn't, we fire off a checksum error. Return the
1714  * number such failures.
1715  */
1716 static int
1717 raidz_parity_verify(zio_t *zio, raidz_map_t *rm)
1718 {
1719         void *orig[VDEV_RAIDZ_MAXPARITY];
1720         int c, ret = 0;
1721         raidz_col_t *rc;

1723         for (c = 0; c < rm->rm_firstdatacol; c++) {
1724                 rc = &rm->rm_col[c];
1725                 if (!rc->rc_tried || rc->rc_error != 0)
1726                         continue;
1727                 orig[c] = zio_buf_alloc(rc->rc_size);
1728                 bcopy(rc->rc_data, orig[c], rc->rc_size);
1729         }

1731         vdev_raidz_generate_parity(rm);

1733         for (c = 0; c < rm->rm_firstdatacol; c++) {
1734                 rc = &rm->rm_col[c];
1735                 if (!rc->rc_tried || rc->rc_error != 0)
1736                         continue;
1737                 if (bcmp(orig[c], rc->rc_data, rc->rc_size) != 0) {
1738                         raidz_checksum_error(zio, rc, orig[c]);
1739                         rc->rc_error = SET_ERROR(ECKSUM);
1740                         ret++;
1741                 }
1742                 zio_buf_free(orig[c], rc->rc_size);
```

```
1743         }

1745         return (ret);
1746 }

1748 /*
1749  * Keep statistics on all the ways that we used parity to correct data.
1750  */
1751 static uint64_t raidz_corrected[1 << VDEV_RAIDZ_MAXPARITY];

1753 static int
1754 vdev_raidz_worst_error(raidz_map_t *rm)
1755 {
1756         int error = 0;

1758         for (int c = 0; c < rm->rm_cols; c++)
1759                 error = zio_worst_error(error, rm->rm_col[c].rc_error);

1761         return (error);
1762 }

1764 /*
1765  * Iterate over all combinations of bad data and attempt a reconstruction.
1766  * Note that the algorithm below is non-optimal because it doesn't take into
1767  * account how reconstruction is actually performed. For example, with
1768  * triple-parity RAID-Z the reconstruction procedure is the same if column 4
1769  * is targeted as invalid as if columns 1 and 4 are targeted since in both
1770  * cases we'd only use parity information in column 0.
1771  */
1772 static int
1773 vdev_raidz_combrec(zio_t *zio, int total_errors, int data_errors)
1774 {
1775         raidz_map_t *rm = zio->io_vsd;
1776         raidz_col_t *rc;
1777         void *orig[VDEV_RAIDZ_MAXPARITY];
1778         int tstore[VDEV_RAIDZ_MAXPARITY + 2];
1779         int *tgts = &tstore[1];
1780         int current, next, i, c, n;
1781         int code, ret = 0;

1783         ASSERT(total_errors < rm->rm_firstdatacol);

1785         /*
1786          * This simplifies one edge condition.
1787          */
1788         tgts[-1] = -1;

1790         for (n = 1; n <= rm->rm_firstdatacol - total_errors; n++) {
1791                 /*
1792                  * Initialize the targets array by finding the first n columns
1793                  * that contain no error.
1794                  *
1795                  * If there were no data errors, we need to ensure that we're
1796                  * always explicitly attempting to reconstruct at least one
1797                  * data column. To do this, we simply push the highest target
1798                  * up into the data columns.
1799                  */
1800                 for (c = 0, i = 0; i < n; i++) {
1801                         if (i == n - 1 && data_errors == 0 &&
1802                             c < rm->rm_firstdatacol) {
1803                                 c = rm->rm_firstdatacol;
1804                         }

1806                         while (rm->rm_col[c].rc_error != 0) {
1807                                 c++;
1808                                 ASSERT3S(c, <, rm->rm_cols);
```



```
1809                         }

1811                         tgts[i] = c++;
1812                 }

1814                 /*
1815                  * Setting tgts[n] simplifies the other edge condition.
1816                  */
1817                 tgts[n] = rm->rm_cols;

1819                 /*
1820                  * These buffers were allocated in previous iterations.
1821                  */
1822                 for (i = 0; i < n - 1; i++) {
1823                         ASSERT(orig[i] != NULL);
1824                 }

1826                 orig[n - 1] = zio_buf_alloc(rm->rm_col[0].rc_size);

1828                 current = 0;
1829                 next = tgts[current];

1831                 while (current != n) {
1832                         tgts[current] = next;
1833                         current = 0;

1835                         /*
1836                          * Save off the original data that we're going to
1837                          * attempt to reconstruct.
1838                          */
1839                         for (i = 0; i < n; i++) {
1840                                 ASSERT(orig[i] != NULL);
1841                                 c = tgts[i];
1842                                 ASSERT3S(c, >=, 0);
1843                                 ASSERT3S(c, <, rm->rm_cols);
1844                                 rc = &rm->rm_col[c];
1845                                 bcopy(rc->rc_data, orig[i], rc->rc_size);
1846                         }

1848                         /*
1849                          * Attempt a reconstruction and exit the outer loop on
1850                          * success.
1851                          */
1852                         code = vdev_raidz_reconstruct(rm, tgts, n);
1853                         if (raidz_checksum_verify(zio) == 0) {
1854                                 atomic_inc_64(&raidz_corrected[code]);

1856                                 for (i = 0; i < n; i++) {
1857                                         c = tgts[i];
1858                                         rc = &rm->rm_col[c];
1859                                         ASSERT(rc->rc_error == 0);
1860                                         if (rc->rc_tried)
1861                                                 raidz_checksum_error(zio, rc,
1862                                                     orig[i]);
1863                                         rc->rc_error = SET_ERROR(ECKSUM);
1864                                 }

1866                                 ret = code;
1867                                 goto done;
1868                         }

1870                         /*
1871                          * Restore the original data.
1872                          */
1873                         for (i = 0; i < n; i++) {
1874                                 c = tgts[i];
```

```
1875                                rc = &rm->rm_col[c];
1876                                bcopy(orig[i], rc->rc_data, rc->rc_size);
1877                        }

1879                        do {
1880                                /*
1881                                 * Find the next valid column after the current
1882                                 * position..
1883                                 */
1884                                for (next = tgts[current] + 1;
1885                                    next < rm->rm_cols &&
1886                                    rm->rm_col[next].rc_error != 0; next++)
1887                                        continue;

1889                                ASSERT(next <= tgts[current + 1]);

1891                                /*
1892                                 * If that spot is available, we're done here.
1893                                 */
1894                                if (next != tgts[current + 1])
1895                                        break;

1897                                /*
1898                                 * Otherwise, find the next valid column after
1899                                 * the previous position.
1900                                 */
1901                                for (c = tgts[current - 1] + 1;
1902                                    rm->rm_col[c].rc_error != 0; c++)
1903                                        continue;

1905                                tgts[current] = c;
1906                                current++;

1908                        } while (current != n);
1909                }
1910        }
1911        n--;
1912 done:
1913        for (i = 0; i < n; i++) {
1914                zio_buf_free(orig[i], rm->rm_col[0].rc_size);
1915        }

1917        return (ret);
1918 }

1920 /*
1921  * Complete an IO operation on a RAIDZ VDev
1922  *
1923  * Outline:
1924  * - For write operations:
1925  *   1. Check for errors on the child IOs.
1926  *   2. Return, setting an error code if too few child VDevs were written
1927  *      to reconstruct the data later.  Note that partial writes are
1928  *      considered successful if they can be reconstructed at all.
1929  * - For read operations:
1930  *   1. Check for errors on the child IOs.
1931  *   2. If data errors occurred:
1932  *      a. Try to reassemble the data from the parity available.
1933  *      b. If we haven't yet read the parity drives, read them now.
1934  *      c. If all parity drives have been read but the data still doesn't
1935  *         reassemble with a correct checksum, then try combinatorial
1936  *         reconstruction.
1937  *      d. If that doesn't work, return an error.
1938  *   3. If there were unexpected errors or this is a resilver operation,
1939  *      rewrite the vdevs that had errors.
1940  */
```



```
1941 #endif /* ! codereview */
1942 static void
1943 vdev_raidz_io_done(zio_t *zio)
1944 {
1945        vdev_t *vd = zio->io_vd;
1946        vdev_t *cvd;
1947        raidz_map_t *rm = zio->io_vsd;
1948        raidz_col_t *rc;
1949        int unexpected_errors = 0;
1950        int parity_errors = 0;
1951        int parity_untried = 0;
1952        int data_errors = 0;
1953        int total_errors = 0;
1954        int n, c;
1955        int tgts[VDEV_RAIDZ_MAXPARITY];
1956        int code;

1958        ASSERT(zio->io_bp != NULL);  /* XXX need to add code to enforce this */

1960        ASSERT(rm->rm_missingparity <= rm->rm_firstdatacol);
1961        ASSERT(rm->rm_missingdata <= rm->rm_cols - rm->rm_firstdatacol);

1963        for (c = 0; c < rm->rm_cols; c++) {
1964                rc = &rm->rm_col[c];

1966                if (rc->rc_error) {
1967                        ASSERT(rc->rc_error != ECKSUM); /* child has no bp */

1969                        if (c < rm->rm_firstdatacol)
1970                                parity_errors++;
1971                        else
1972                                data_errors++;

1974                        if (!rc->rc_skipped)
1975                                unexpected_errors++;

1977                        total_errors++;
1978                } else if (c < rm->rm_firstdatacol && !rc->rc_tried) {
1979                        parity_untried++;
1980                }
1981        }

1983        if (zio->io_type == ZIO_TYPE_WRITE) {
1984                /*
1985                 * XXX -- for now, treat partial writes as a success.
1986                 * (If we couldn't write enough columns to reconstruct
1987                 * the data, the I/O failed.  Otherwise, good enough.)
1988                 *
1989                 * Now that we support write reallocation, it would be better
1990                 * to treat partial failure as real failure unless there are
1991                 * no non-degraded top-level vdevs left, and not update DTLs
1992                 * if we intend to reallocate.
1993                 */
1994                /* XXPOLICY */
1995                if (total_errors > rm->rm_firstdatacol)
1996                        zio->io_error = vdev_raidz_worst_error(rm);

1998                return;
1999        }

2001        ASSERT(zio->io_type == ZIO_TYPE_READ);
2002        /*
2003         * There are three potential phases for a read:
2004         *      1. produce valid data from the columns read
2005         *      2. read all disks and try again
2006         *      3. perform combinatorial reconstruction
```

```
2007          *
2008          * Each phase is progressively both more expensive and less likely to
2009          * occur. If we encounter more errors than we can repair or all phases
2010          * fail, we have no choice but to return an error.
2011          */
2013         /*
2014          * If the number of errors we saw was correctable -- less than or equal
2015          * to the number of parity disks read -- attempt to produce data that
2016          * has a valid checksum. Naturally, this case applies in the absence of
2017          * any errors.
2018          */
2019         if (total_errors <= rm->rm_firstdatacol - parity_untried) {
2020               if (data_errors == 0) {
2021                       if (raidz_checksum_verify(zio) == 0) {
2022                               /*
2023                                * If we read parity information (unnecessarily
2024                                * as it happens since no reconstruction was
2025                                * needed) regenerate and verify the parity.
2026                                * We also regenerate parity when resilvering
2027                                * so we can write it out to the failed device
2028                                * later.
2029                                */
2030                               if (parity_errors + parity_untried <
2031                                   rm->rm_firstdatacol ||
2032                                   (zio->io_flags & ZIO_FLAG_RESILVER)) {
2033                                       n = raidz_parity_verify(zio, rm);
2034                                       unexpected_errors += n;
2035                                       ASSERT(parity_errors + n <=
2036                                           rm->rm_firstdatacol);
2037                               }
2038                               goto done;
2039                       }
2040               } else {
2041                       /*
2042                        * We either attempt to read all the parity columns or
2043                        * none of them. If we didn't try to read parity, we
2044                        * wouldn't be here in the correctable case. There must
2045                        * also have been fewer parity errors than parity
2046                        * columns or, again, we wouldn't be in this code path.
2047                        */
2048                       ASSERT(parity_untried == 0);
2049                       ASSERT(parity_errors < rm->rm_firstdatacol);
2051                       /*
2052                        * Identify the data columns that reported an error.
2053                        */
2054                       n = 0;
2055                       for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
2056                               rc = &rm->rm_col[c];
2057                               if (rc->rc_error != 0) {
2058                                       ASSERT(n < VDEV_RAIDZ_MAXPARITY);
2059                                       tgts[n++] = c;
2060                               }
2061                       }
2063                       ASSERT(rm->rm_firstdatacol >= n);
2065                       code = vdev_raidz_reconstruct(rm, tgts, n);
2067                       if (raidz_checksum_verify(zio) == 0) {
2068                               atomic_inc_64(&raidz_corrected[code]);
2070                               /*
2071                                * If we read more parity disks than were used
2072                                * for reconstruction, confirm that the other
```



```
2073                                * parity disks produced correct data. This
2074                                * routine is suboptimal in that it regenerates
2075                                * the parity that we already used in addition
2076                                * to the parity that we're attempting to
2077                                * verify, but this should be a relatively
2078                                * uncommon case, and can be optimized if it
2079                                * becomes a problem. Note that we regenerate
2080                                * parity when resilvering so we can write it
2081                                * out to failed devices later.
2082                                */
2083                               if (parity_errors < rm->rm_firstdatacol - n ||
2084                                   (zio->io_flags & ZIO_FLAG_RESILVER)) {
2085                                       n = raidz_parity_verify(zio, rm);
2086                                       unexpected_errors += n;
2087                                       ASSERT(parity_errors + n <=
2088                                           rm->rm_firstdatacol);
2089                               }
2091                               goto done;
2092                       }
2093               }
2094       }
2096       /*
2097        * This isn't a typical situation -- either we got a read error or
2098        * a child silently returned bad data. Read every block so we can
2099        * try again with as much data and parity as we can track down. If
2100        * we've already been through once before, all children will be marked
2101        * as tried so we'll proceed to combinatorial reconstruction.
2102        */
2103       unexpected_errors = 1;
2104       rm->rm_missingdata = 0;
2105       rm->rm_missingparity = 0;
2107       for (c = 0; c < rm->rm_cols; c++) {
2108               if (rm->rm_col[c].rc_tried)
2109                       continue;
2111               zio_vdev_io_redone(zio);
2112               do {
2113                       rc = &rm->rm_col[c];
2114                       if (rc->rc_tried)
2115                               continue;
2116                       zio_nowait(zio_vdev_child_io(zio, NULL,
2117                           vd->vdev_child[rc->rc_devidx],
2118                           rc->rc_offset, rc->rc_data, rc->rc_size,
2119                           zio->io_type, zio->io_priority, 0,
2120                           vdev_raidz_child_done, rc));
2121               } while (++c < rm->rm_cols);
2123               return;
2124       }
2126       /*
2127        * At this point we've attempted to reconstruct the data given the
2128        * errors we detected, and we've attempted to read all columns. There
2129        * must, therefore, be one or more additional problems -- silent errors
2130        * resulting in invalid data rather than explicit I/O errors resulting
2131        * in absent data. We check if there is enough additional data to
2132        * possibly reconstruct the data and then perform combinatorial
2133        * reconstruction over all possible combinations. If that fails,
2134        * we're cooked.
2135        */
2136       if (total_errors > rm->rm_firstdatacol) {
2137               zio->io_error = vdev_raidz_worst_error(rm);
```

```
2139         } else if (total_errors < rm->rm_firstdatacol &&
2140             (code = vdev_raidz_combrec(zio, total_errors, data_errors)) != 0) {
2141                 /*
2142                  * If we didn't use all the available parity for the
2143                  * combinatorial reconstruction, verify that the remaining
2144                  * parity is correct.
2145                  */
2146                 if (code != (1 << rm->rm_firstdatacol) - 1)
2147                         (void) raidz_parity_verify(zio, rm);
2148         } else {
2149                 /*
2150                  * We're here because either:
2151                  *
2152                  *      total_errors == rm_first_datacol, or
2153                  *      vdev_raidz_combrec() failed
2154                  *
2155                  * In either case, there is enough bad data to prevent
2156                  * reconstruction.
2157                  *
2158                  * Start checksum ereports for all children which haven't
2159                  * failed, and the IO wasn't speculative.
2160                  */
2161                 zio->io_error = SET_ERROR(ECKSUM);

2163                 if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
2164                         for (c = 0; c < rm->rm_cols; c++) {
2165                                 rc = &rm->rm_col[c];
2166                                 if (rc->rc_error == 0) {
2167                                         zio_bad_cksum_t zbc;
2168                                         zbc.zbc_has_cksum = 0;
2169                                         zbc.zbc_injected =
2170                                             rm->rm_ecksuminjected;

2172                                         zfs_ereport_start_checksum(
2173                                             zio->io_spa,
2174                                             vd->vdev_child[rc->rc_devidx],
2175                                             zio, rc->rc_offset, rc->rc_size,
2176                                             (void *)(uintptr_t)c, &zbc);
2177                                 }
2178                         }
2179                 }
2180         }

2182 done:
2183         zio_checksum_verified(zio);

2185         if (zio->io_error == 0 && spa_writeable(zio->io_spa) &&
2186             (unexpected_errors || (zio->io_flags & ZIO_FLAG_RESILVER))) {
2187                 /*
2188                  * Use the good data we have in hand to repair damaged children.
2189                  */
2190                 for (c = 0; c < rm->rm_cols; c++) {
2191                         rc = &rm->rm_col[c];
2192                         cvd = vd->vdev_child[rc->rc_devidx];

2194                         if (rc->rc_error == 0)
2195                                 continue;

2197                         zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
2198                             rc->rc_offset, rc->rc_data, rc->rc_size,
2199                             ZIO_TYPE_WRITE, zio->io_priority,
2200                             ZIO_FLAG_IO_REPAIR | (unexpected_errors ?
2201                             ZIO_FLAG_SELF_HEAL : 0), NULL, NULL));
2202                 }
2203         }
2204 }
```



```
2206 static void
2207 vdev_raidz_state_change(vdev_t *vd, int faulted, int degraded)
2208 {
2209         if (faulted > vd->vdev_nparity)
2210                 vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2211                     VDEV_AUX_NO_REPLICAS);
2212         else if (degraded + faulted != 0)
2213                 vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, VDEV_AUX_NONE);
2214         else
2215                 vdev_set_state(vd, B_FALSE, VDEV_STATE_HEALTHY, VDEV_AUX_NONE);
2216 }

2218 vdev_ops_t vdev_raidz_ops = {
2219         vdev_raidz_open,
2220         vdev_raidz_close,
2221         vdev_raidz_asize,
2222         vdev_raidz_io_start,
2223         vdev_raidz_io_done,
2224         vdev_raidz_state_change,
2225         NULL,
2226         NULL,
2227         VDEV_TYPE_RAIDZ,        /* name of this vdev type */
2228         B_FALSE                 /* not a leaf vdev */
2229 };
```

```
**********************************************************
   34536 Tue Apr 23 14:09:38 2013
new/usr/src/uts/common/fs/zfs/zfs_ctldir.c
3741 zfs needs better comments
Submitted by:   Will Andrews <willa@spectralogic.com>
Submitted by:   Justin Gibbs <justing@spectralogic.com>
Submitted by:   Alan Somers <alans@spectralogic.com>
Reviewed by:    Matthew Ahrens <mahrens@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
 508 /*
 509  * Gets the full dataset name that corresponds to the given snapshot name
 510  * Example:
 511  *      zfsctl_snapshot_zname("snap1") -> "mypool/myfs@snap1"
 512  */
 513 #endif /* ! codereview */
 514 static int
 515 zfsctl_snapshot_zname(vnode_t *vp, const char *name, int len, char *zname)
 516 {
 517         objset_t *os = ((zfsvfs_t *)((vp)->v_vfsp->vfs_data))->z_os;

 519         if (snapshot_namecheck(name, NULL, NULL) != 0)
 520                 return (SET_ERROR(EILSEQ));
 521         dmu_objset_name(os, zname);
 522         if (strlen(zname) + 1 + strlen(name) >= len)
 523                 return (SET_ERROR(ENAMETOOLONG));
 524         (void) strcat(zname, "@");
 525         (void) strcat(zname, name);
 526         return (0);
 527 }

 529 static int
 530 zfsctl_unmount_snap(zfs_snapentry_t *sep, int fflags, cred_t *cr)
 531 {
 532         vnode_t *svp = sep->se_root;
 533         int error;

 535         ASSERT(vn_ismntpt(svp));

 537         /* this will be dropped by dounmount() */
 538         if ((error = vn_vfswlock(svp)) != 0)
 539                 return (error);

 541         VN_HOLD(svp);
 542         error = dounmount(vn_mountedvfs(svp), fflags, cr);
 543         if (error) {
 544                 VN_RELE(svp);
 545                 return (error);
 546         }

 548         /*
 549          * We can't use VN_RELE(), as that will try to invoke
 550          * zfsctl_snapdir_inactive(), which would cause us to destroy
 551          * the sd_lock mutex held by our caller.
 552          */
 553         ASSERT(svp->v_count == 1);
 554         gfs_vop_inactive(svp, cr, NULL);

 556         kmem_free(sep->se_name, strlen(sep->se_name) + 1);
 557         kmem_free(sep, sizeof (zfs_snapentry_t));

 559         return (0);
 560 }

 562 static void
```



```
 563 zfsctl_rename_snap(zfsctl_snapdir_t *sdp, zfs_snapentry_t *sep, const char *nm)
 564 {
 565         avl_index_t where;
 566         vfs_t *vfsp;
 567         refstr_t *pathref;
 568         char newpath[MAXNAMELEN];
 569         char *tail;

 571         ASSERT(MUTEX_HELD(&sdp->sd_lock));
 572         ASSERT(sep != NULL);

 574         vfsp = vn_mountedvfs(sep->se_root);
 575         ASSERT(vfsp != NULL);

 577         vfs_lock_wait(vfsp);

 579         /*
 580          * Change the name in the AVL tree.
 581          */
 582         avl_remove(&sdp->sd_snaps, sep);
 583         kmem_free(sep->se_name, strlen(sep->se_name) + 1);
 584         sep->se_name = kmem_alloc(strlen(nm) + 1, KM_SLEEP);
 585         (void) strcpy(sep->se_name, nm);
 586         VERIFY(avl_find(&sdp->sd_snaps, sep, &where) == NULL);
 587         avl_insert(&sdp->sd_snaps, sep, where);

 589         /*
 590          * Change the current mountpoint info:
 591          *      - update the tail of the mntpoint path
 592          *      - update the tail of the resource path
 593          */
 594         pathref = vfs_getmntpoint(vfsp);
 595         (void) strncpy(newpath, refstr_value(pathref), sizeof (newpath));
 596         VERIFY((tail = strrchr(newpath, '/')) != NULL);
 597         *(tail+1) = '\0';
 598         ASSERT3U(strlen(newpath) + strlen(nm), <, sizeof (newpath));
 599         (void) strcat(newpath, nm);
 600         refstr_rele(pathref);
 601         vfs_setmntpoint(vfsp, newpath, 0);

 603         pathref = vfs_getresource(vfsp);
 604         (void) strncpy(newpath, refstr_value(pathref), sizeof (newpath));
 605         VERIFY((tail = strrchr(newpath, '@')) != NULL);
 606         *(tail+1) = '\0';
 607         ASSERT3U(strlen(newpath) + strlen(nm), <, sizeof (newpath));
 608         (void) strcat(newpath, nm);
 609         refstr_rele(pathref);
 610         vfs_setresource(vfsp, newpath, 0);

 612         vfs_unlock(vfsp);
 613 }

 615 /*ARGSUSED*/
 616 static int
 617 zfsctl_snapdir_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm,
 618     cred_t *cr, caller_context_t *ct, int flags)
 619 {
 620         zfsctl_snapdir_t *sdp = sdvp->v_data;
 621         zfs_snapentry_t search, *sep;
 622         zfsvfs_t *zfsvfs;
 623         avl_index_t where;
 624         char from[MAXNAMELEN], to[MAXNAMELEN];
 625         char real[MAXNAMELEN], fsname[MAXNAMELEN];
 626         int err;

 628         zfsvfs = sdvp->v_vfsp->vfs_data;
```

```
629         ZFS_ENTER(zfsvfs);

631         if ((flags & FIGNORECASE) || zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {
632                 err = dmu_snapshot_realname(zfsvfs->z_os, snm, real,
633                     MAXNAMELEN, NULL);
634                 if (err == 0) {
635                         snm = real;
636                 } else if (err != ENOTSUP) {
637                         ZFS_EXIT(zfsvfs);
638                         return (err);
639                 }
640         }

642         ZFS_EXIT(zfsvfs);

644         dmu_objset_name(zfsvfs->z_os, fsname);

646         err = zfsctl_snapshot_zname(sdvp, snm, MAXNAMELEN, from);
647         if (err == 0)
648                 err = zfsctl_snapshot_zname(tdvp, tnm, MAXNAMELEN, to);
649         if (err == 0)
650                 err = zfs_secpolicy_rename_perms(from, to, cr);
651         if (err != 0)
652                 return (err);

654         /*
655          * Cannot move snapshots out of the snapdir.
656          */
657         if (sdvp != tdvp)
658                 return (SET_ERROR(EINVAL));

660         if (strcmp(snm, tnm) == 0)
661                 return (0);

663         mutex_enter(&sdp->sd_lock);

665         search.se_name = (char *)snm;
666         if ((sep = avl_find(&sdp->sd_snaps, &search, &where)) == NULL) {
667                 mutex_exit(&sdp->sd_lock);
668                 return (SET_ERROR(ENOENT));
669         }

671         err = dsl_dataset_rename_snapshot(fsname, snm, tnm, B_FALSE);
672         if (err == 0)
673                 zfsctl_rename_snap(sdp, sep, tnm);

675         mutex_exit(&sdp->sd_lock);

677         return (err);
678 }

680 /* ARGSUSED */
681 static int
682 zfsctl_snapdir_remove(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
683     caller_context_t *ct, int flags)
684 {
685         zfsctl_snapdir_t *sdp = dvp->v_data;
686         zfs_snapentry_t *sep;
687         zfs_snapentry_t search;
688         zfsvfs_t *zfsvfs;
689         char snapname[MAXNAMELEN];
690         char real[MAXNAMELEN];
691         int err;

693         zfsvfs = dvp->v_vfsp->vfs_data;
694         ZFS_ENTER(zfsvfs);
```



```
696         if ((flags & FIGNORECASE) || zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {

698                 err = dmu_snapshot_realname(zfsvfs->z_os, name, real,
699                     MAXNAMELEN, NULL);
700                 if (err == 0) {
701                         name = real;
702                 } else if (err != ENOTSUP) {
703                         ZFS_EXIT(zfsvfs);
704                         return (err);
705                 }
706         }

708         ZFS_EXIT(zfsvfs);

710         err = zfsctl_snapshot_zname(dvp, name, MAXNAMELEN, snapname);
711         if (err == 0)
712                 err = zfs_secpolicy_destroy_perms(snapname, cr);
713         if (err != 0)
714                 return (err);

716         mutex_enter(&sdp->sd_lock);

718         search.se_name = name;
719         sep = avl_find(&sdp->sd_snaps, &search, NULL);
720         if (sep) {
721                 avl_remove(&sdp->sd_snaps, sep);
722                 err = zfsctl_unmount_snap(sep, MS_FORCE, cr);
723                 if (err != 0)
724                         avl_add(&sdp->sd_snaps, sep);
725                 else
726                         err = dsl_destroy_snapshot(snapname, B_FALSE);
727         } else {
728                 err = SET_ERROR(ENOENT);
729         }

731         mutex_exit(&sdp->sd_lock);

733         return (err);
734 }

736 /*
737  * This creates a snapshot under '.zfs/snapshot'.
738  */
739 /* ARGSUSED */
740 static int
741 zfsctl_snapdir_mkdir(vnode_t *dvp, char *dirname, vattr_t *vap, vnode_t  **vpp,
742     cred_t *cr, caller_context_t *cc, int flags, vsecattr_t *vsecp)
743 {
744         zfsvfs_t *zfsvfs = dvp->v_vfsp->vfs_data;
745         char name[MAXNAMELEN];
746         int err;
747         static enum symfollow follow = NO_FOLLOW;
748         static enum uio_seg seg = UIO_SYSSPACE;

750         if (snapshot_namecheck(dirname, NULL, NULL) != 0)
751                 return (SET_ERROR(EILSEQ));

753         dmu_objset_name(zfsvfs->z_os, name);

755         *vpp = NULL;

757         err = zfs_secpolicy_snapshot_perms(name, cr);
758         if (err != 0)
759                 return (err);
```

```
761         if (err == 0) {
762                 err = dmu_objset_snapshot_one(name, dirname);
763                 if (err != 0)
764                         return (err);
765                 err = lookupnameat(dirname, seg, follow, NULL, vpp, dvp);
766         }
767
768         return (err);
769 }
770
771 /*
772  * Lookup entry point for the 'snapshot' directory.  Try to open the
773  * snapshot if it exist, creating the pseudo filesystem vnode as necessary.
774  * Perform a mount of the associated dataset on top of the vnode.
775  */
776 /* ARGSUSED */
777 static int
778 zfsctl_snapdir_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, pathname_t *pnp,
779     int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
780     int *direntflags, pathname_t *realpnp)
781 {
782         zfsctl_snapdir_t *sdp = dvp->v_data;
783         objset_t *snap;
784         char snapname[MAXNAMELEN];
785         char real[MAXNAMELEN];
786         char *mountpoint;
787         zfs_snapentry_t *sep, search;
788         struct mounta margs;
789         vfs_t *vfsp;
790         size_t mountpoint_len;
791         avl_index_t where;
792         zfsvfs_t *zfsvfs = dvp->v_vfsp->vfs_data;
793         int err;
794
795         /*
796          * No extended attributes allowed under .zfs
797          */
798         if (flags & LOOKUP_XATTR)
799                 return (SET_ERROR(EINVAL));
800
801         ASSERT(dvp->v_type == VDIR);
802
803         /*
804          * If we get a recursive call, that means we got called
805          * from the domount() code while it was trying to look up the
806          * spec (which looks like a local path for zfs).  We need to
807          * add some flag to domount() to tell it not to do this lookup.
808          */
809         if (MUTEX_HELD(&sdp->sd_lock))
810                 return (SET_ERROR(ENOENT));
811
812         ZFS_ENTER(zfsvfs);
813
814         if (gfs_lookup_dot(vpp, dvp, zfsvfs->z_ctldir, nm) == 0) {
815                 ZFS_EXIT(zfsvfs);
816                 return (0);
817         }
818
819         if (flags & FIGNORECASE) {
820                 boolean_t conflict = B_FALSE;
821
822                 err = dmu_snapshot_realname(zfsvfs->z_os, nm, real,
823                     MAXNAMELEN, &conflict);
824                 if (err == 0) {
825                         nm = real;
826                 } else if (err != ENOTSUP) {
```



```
827                         ZFS_EXIT(zfsvfs);
828                         return (err);
829                 }
830                 if (realpnp)
831                         (void) strlcpy(realpnp->pn_buf, nm,
832                             realpnp->pn_bufsize);
833                 if (conflict && direntflags)
834                         *direntflags = ED_CASE_CONFLICT;
835         }
836
837         mutex_enter(&sdp->sd_lock);
838         search.se_name = (char *)nm;
839         if ((sep = avl_find(&sdp->sd_snaps, &search, &where)) != NULL) {
840                 *vpp = sep->se_root;
841                 VN_HOLD(*vpp);
842                 err = traverse(vpp);
843                 if (err != 0) {
844                         VN_RELE(*vpp);
845                         *vpp = NULL;
846                 } else if (*vpp == sep->se_root) {
847                         /*
848                          * The snapshot was unmounted behind our backs,
849                          * try to remount it.
850                          */
851                         goto domount;
852                 } else {
853                         /*
854                          * VROOT was set during the traverse call.  We need
855                          * to clear it since we're pretending to be part
856                          * of our parent's vfs.
857                          */
858                         (*vpp)->v_flag &= ~VROOT;
859                 }
860                 mutex_exit(&sdp->sd_lock);
861                 ZFS_EXIT(zfsvfs);
862                 return (err);
863         }
864
865         /*
866          * The requested snapshot is not currently mounted, look it up.
867          */
868         err = zfsctl_snapshot_zname(dvp, nm, MAXNAMELEN, snapname);
869         if (err != 0) {
870                 mutex_exit(&sdp->sd_lock);
871                 ZFS_EXIT(zfsvfs);
872                 /*
873                  * handle "ls *" or "?" in a graceful manner,
874                  * forcing EILSEQ to ENOENT.
875                  * Since shell ultimately passes "*" or "?" as name to lookup
876                  */
877                 return (err == EILSEQ ? ENOENT : err);
878         }
879         if (dmu_objset_hold(snapname, FTAG, &snap) != 0) {
880                 mutex_exit(&sdp->sd_lock);
881                 ZFS_EXIT(zfsvfs);
882                 return (SET_ERROR(ENOENT));
883         }
884
885         sep = kmem_alloc(sizeof (zfs_snapentry_t), KM_SLEEP);
886         sep->se_name = kmem_alloc(strlen(nm) + 1, KM_SLEEP);
887         (void) strcpy(sep->se_name, nm);
888         *vpp = sep->se_root = zfsctl_snapshot_mknode(dvp, dmu_objset_id(snap));
889         avl_insert(&sdp->sd_snaps, sep, where);
890
891         dmu_objset_rele(snap, FTAG);
892 domount:
```

```
893         mountpoint_len = strlen(refstr_value(dvp->v_vfsp->vfs_mntpt)) +
894             strlen("/.zfs/snapshot/") + strlen(nm) + 1;
895         mountpoint = kmem_alloc(mountpoint_len, KM_SLEEP);
896         (void) snprintf(mountpoint, mountpoint_len, "%s/.zfs/snapshot/%s",
897             refstr_value(dvp->v_vfsp->vfs_mntpt), nm);
898
899         margs.spec = snapname;
900         margs.dir = mountpoint;
901         margs.flags = MS_SYSSPACE | MS_NOMNTTAB;
902         margs.fstype = "zfs";
903         margs.dataptr = NULL;
904         margs.datalen = 0;
905         margs.optptr = NULL;
906         margs.optlen = 0;
907
908         err = domount("zfs", &margs, *vpp, kcred, &vfsp);
909         kmem_free(mountpoint, mountpoint_len);
910
911         if (err == 0) {
912                 /*
913                  * Return the mounted root rather than the covered mount point.
914                  * Takes the GFS vnode at .zfs/snapshot/<snapname> and returns
915                  * the ZFS vnode mounted on top of the GFS node.  This ZFS
916                  * vnode is the root of the newly created vfsp.
917                  */
918                 VFS_RELE(vfsp);
919                 err = traverse(vpp);
920         }
921
922         if (err == 0) {
923                 /*
924                  * Fix up the root vnode mounted on .zfs/snapshot/<snapname>.
925                  *
926                  * This is where we lie about our v_vfsp in order to
927                  * make .zfs/snapshot/<snapname> accessible over NFS
928                  * without requiring manual mounts of <snapname>.
929                  */
930                 ASSERT(VTOZ(*vpp)->z_zfsvfs != zfsvfs);
931                 VTOZ(*vpp)->z_zfsvfs->z_parent = zfsvfs;
932                 (*vpp)->v_vfsp = zfsvfs->z_vfs;
933                 (*vpp)->v_flag &= ~VROOT;
934         }
935         mutex_exit(&sdp->sd_lock);
936         ZFS_EXIT(zfsvfs);
937
938         /*
939          * If we had an error, drop our hold on the vnode and
940          * zfsctl_snapshot_inactive() will clean up.
941          */
942         if (err != 0) {
943                 VN_RELE(*vpp);
944                 *vpp = NULL;
945         }
946         return (err);
947 }
948
949 /* ARGSUSED */
950 static int
951 zfsctl_shares_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, pathname_t *pnp,
952     int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
953     int *direntflags, pathname_t *realpnp)
954 {
955         zfsvfs_t *zfsvfs = dvp->v_vfsp->vfs_data;
956         znode_t *dzp;
957         int error;
```



```
959         ZFS_ENTER(zfsvfs);
960
961         if (gfs_lookup_dot(vpp, dvp, zfsvfs->z_ctldir, nm) == 0) {
962                 ZFS_EXIT(zfsvfs);
963                 return (0);
964         }
965
966         if (zfsvfs->z_shares_dir == 0) {
967                 ZFS_EXIT(zfsvfs);
968                 return (SET_ERROR(ENOTSUP));
969         }
970         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &dzp)) == 0)
971                 error = VOP_LOOKUP(ZTOV(dzp), nm, vpp, pnp,
972                     flags, rdir, cr, ct, direntflags, realpnp);
973
974         VN_RELE(ZTOV(dzp));
975         ZFS_EXIT(zfsvfs);
976
977         return (error);
978 }
979
980 /* ARGSUSED */
981 static int
982 zfsctl_snapdir_readdir_cb(vnode_t *vp, void *dp, int *eofp,
983     offset_t *offp, offset_t *nextp, void *data, int flags)
984 {
985         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
986         char snapname[MAXNAMELEN];
987         uint64_t id, cookie;
988         boolean_t case_conflict;
989         int error;
990
991         ZFS_ENTER(zfsvfs);
992
993         cookie = *offp;
994         dsl_pool_config_enter(dmu_objset_pool(zfsvfs->z_os), FTAG);
995         error = dmu_snapshot_list_next(zfsvfs->z_os, MAXNAMELEN, snapname, &id,
996             &cookie, &case_conflict);
997         dsl_pool_config_exit(dmu_objset_pool(zfsvfs->z_os), FTAG);
998         if (error) {
999                 ZFS_EXIT(zfsvfs);
1000                if (error == ENOENT) {
1001                        *eofp = 1;
1002                        return (0);
1003                }
1004                return (error);
1005        }
1006
1007        if (flags & V_RDDIR_ENTFLAGS) {
1008                edirent_t *eodp = dp;
1009
1010                (void) strcpy(eodp->ed_name, snapname);
1011                eodp->ed_ino = ZFSCTL_INO_SNAP(id);
1012                eodp->ed_eflags = case_conflict ? ED_CASE_CONFLICT : 0;
1013        } else {
1014                struct dirent64 *odp = dp;
1015
1016                (void) strcpy(odp->d_name, snapname);
1017                odp->d_ino = ZFSCTL_INO_SNAP(id);
1018        }
1019        *nextp = cookie;
1020
1021        ZFS_EXIT(zfsvfs);
1022
1023        return (0);
1024 }
```

```
1026 /* ARGSUSED */
1027 static int
1028 zfsctl_shares_readdir(vnode_t *vp, uio_t *uiop, cred_t *cr, int *eofp,
1029     caller_context_t *ct, int flags)
1030 {
1031         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
1032         znode_t *dzp;
1033         int error;

1035         ZFS_ENTER(zfsvfs);

1037         if (zfsvfs->z_shares_dir == 0) {
1038                 ZFS_EXIT(zfsvfs);
1039                 return (SET_ERROR(ENOTSUP));
1040         }
1041         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &dzp)) == 0) {
1042                 error = VOP_READDIR(ZTOV(dzp), uiop, cr, eofp, ct, flags);
1043                 VN_RELE(ZTOV(dzp));
1044         } else {
1045                 *eofp = 1;
1046                 error = SET_ERROR(ENOENT);
1047         }

1049         ZFS_EXIT(zfsvfs);
1050         return (error);
1051 }

1053 /*
1054  * pvp is the '.zfs' directory (zfsctl_node_t).
1055  * Creates vp, which is '.zfs/snapshot' (zfsctl_snapdir_t).
1056  *
1057  * This function is the callback to create a GFS vnode for '.zfs/snapshot'
1058  * when a lookup is performed on .zfs for "snapshot".
1059  */
1060 vnode_t *
1061 zfsctl_mknode_snapdir(vnode_t *pvp)
1062 {
1063         vnode_t *vp;
1064         zfsctl_snapdir_t *sdp;

1066         vp = gfs_dir_create(sizeof (zfsctl_snapdir_t), pvp,
1067             zfsctl_ops_snapdir, NULL, NULL, MAXNAMELEN,
1068             zfsctl_snapdir_readdir_cb, NULL);
1069         sdp = vp->v_data;
1070         sdp->sd_node.zc_id = ZFSCTL_INO_SNAPDIR;
1071         sdp->sd_node.zc_cmtime = ((zfsctl_node_t *)pvp->v_data)->zc_cmtime;
1072         mutex_init(&sdp->sd_lock, NULL, MUTEX_DEFAULT, NULL);
1073         avl_create(&sdp->sd_snaps, snapentry_compare,
1074             sizeof (zfs_snapentry_t), offsetof(zfs_snapentry_t, se_node));
1075         return (vp);
1076 }

1078 vnode_t *
1079 zfsctl_mknode_shares(vnode_t *pvp)
1080 {
1081         vnode_t *vp;
1082         zfsctl_node_t *sdp;

1084         vp = gfs_dir_create(sizeof (zfsctl_node_t), pvp,
1085             zfsctl_ops_shares, NULL, NULL, MAXNAMELEN,
1086             NULL, NULL);
1087         sdp = vp->v_data;
1088         sdp->zc_cmtime = ((zfsctl_node_t *)pvp->v_data)->zc_cmtime;
1089         return (vp);
```



```
1091 }

1093 /* ARGSUSED */
1094 static int
1095 zfsctl_shares_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
1096     caller_context_t *ct)
1097 {
1098         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
1099         znode_t *dzp;
1100         int error;

1102         ZFS_ENTER(zfsvfs);
1103         if (zfsvfs->z_shares_dir == 0) {
1104                 ZFS_EXIT(zfsvfs);
1105                 return (SET_ERROR(ENOTSUP));
1106         }
1107         if ((error = zfs_zget(zfsvfs, zfsvfs->z_shares_dir, &dzp)) == 0) {
1108                 error = VOP_GETATTR(ZTOV(dzp), vap, flags, cr, ct);
1109                 VN_RELE(ZTOV(dzp));
1110         }
1111         ZFS_EXIT(zfsvfs);
1112         return (error);


1115 }

1117 /* ARGSUSED */
1118 static int
1119 zfsctl_snapdir_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
1120     caller_context_t *ct)
1121 {
1122         zfsvfs_t *zfsvfs = vp->v_vfsp->vfs_data;
1123         zfsctl_snapdir_t *sdp = vp->v_data;

1125         ZFS_ENTER(zfsvfs);
1126         zfsctl_common_getattr(vp, vap);
1127         vap->va_nodeid = gfs_file_inode(vp);
1128         vap->va_nlink = vap->va_size = avl_numnodes(&sdp->sd_snaps) + 2;
1129         vap->va_ctime = vap->va_mtime = dmu_objset_snap_cmtime(zfsvfs->z_os);
1130         ZFS_EXIT(zfsvfs);

1132         return (0);
1133 }

1135 /* ARGSUSED */
1136 static void
1137 zfsctl_snapdir_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1138 {
1139         zfsctl_snapdir_t *sdp = vp->v_data;
1140         void *private;

1142         private = gfs_dir_inactive(vp);
1143         if (private != NULL) {
1144                 ASSERT(avl_numnodes(&sdp->sd_snaps) == 0);
1145                 mutex_destroy(&sdp->sd_lock);
1146                 avl_destroy(&sdp->sd_snaps);
1147                 kmem_free(private, sizeof (zfsctl_snapdir_t));
1148         }
1149 }

1151 static const fs_operation_def_t zfsctl_tops_snapdir[] = {
1152         { VOPNAME_OPEN,         { .vop_open = zfsctl_common_open }      },
1153         { VOPNAME_CLOSE,        { .vop_close = zfsctl_common_close }    },
1154         { VOPNAME_IOCTL,        { .error = fs_inval }                   },
1155         { VOPNAME_GETATTR,      { .vop_getattr = zfsctl_snapdir_getattr } },
1156         { VOPNAME_ACCESS,       { .vop_access = zfsctl_common_access }  },
```

```
1157         { VOPNAME_RENAME,       { .vop_rename = zfsctl_snapdir_rename } },
1158         { VOPNAME_RMDIR,        { .vop_rmdir = zfsctl_snapdir_remove }  },
1159         { VOPNAME_MKDIR,        { .vop_mkdir = zfsctl_snapdir_mkdir }   },
1160         { VOPNAME_READDIR,      { .vop_readdir = gfs_vop_readdir }      },
1161         { VOPNAME_LOOKUP,       { .vop_lookup = zfsctl_snapdir_lookup } },
1162         { VOPNAME_SEEK,         { .vop_seek = fs_seek }                 },
1163         { VOPNAME_INACTIVE,     { .vop_inactive = zfsctl_snapdir_inactive } },
1164         { VOPNAME_FID,          { .vop_fid = zfsctl_common_fid }        },
1165         { NULL }
1166 };

1168 static const fs_operation_def_t zfsctl_tops_shares[] = {
1169         { VOPNAME_OPEN,         { .vop_open = zfsctl_common_open }      },
1170         { VOPNAME_CLOSE,        { .vop_close = zfsctl_common_close }    },
1171         { VOPNAME_IOCTL,        { .error = fs_inval }                   },
1172         { VOPNAME_GETATTR,      { .vop_getattr = zfsctl_shares_getattr } },
1173         { VOPNAME_ACCESS,       { .vop_access = zfsctl_common_access }  },
1174         { VOPNAME_READDIR,      { .vop_readdir = zfsctl_shares_readdir } },
1175         { VOPNAME_LOOKUP,       { .vop_lookup = zfsctl_shares_lookup }  },
1176         { VOPNAME_SEEK,         { .vop_seek = fs_seek }                 },
1177         { VOPNAME_INACTIVE,     { .vop_inactive = gfs_vop_inactive } },
1178         { VOPNAME_FID,          { .vop_fid = zfsctl_shares_fid } },
1179         { NULL }
1180 };

1182 /*
1183  * pvp is the GFS vnode '.zfs/snapshot'.
1184  *
1185  * This creates a GFS node under '.zfs/snapshot' representing each
1186  * snapshot.  This newly created GFS node is what we mount snapshot
1187  * vfs_t's ontop of.
1188  */
1189 static vnode_t *
1190 zfsctl_snapshot_mknode(vnode_t *pvp, uint64_t objset)
1191 {
1192         vnode_t *vp;
1193         zfsctl_node_t *zcp;

1195         vp = gfs_dir_create(sizeof (zfsctl_node_t), pvp,
1196             zfsctl_ops_snapshot, NULL, NULL, MAXNAMELEN, NULL, NULL);
1197         zcp = vp->v_data;
1198         zcp->zc_id = objset;

1200         return (vp);
1201 }

1203 static void
1204 zfsctl_snapshot_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1205 {
1206         zfsctl_snapdir_t *sdp;
1207         zfs_snapentry_t *sep, *next;
1208         vnode_t *dvp;

1210         VERIFY(gfs_dir_lookup(vp, "..", &dvp, cr, 0, NULL, NULL) == 0);
1211         sdp = dvp->v_data;

1213         mutex_enter(&sdp->sd_lock);

1215         if (vp->v_count > 1) {
1216                 mutex_exit(&sdp->sd_lock);
1217                 return;
1218         }
1219         ASSERT(!vn_ismntpt(vp));

1221         sep = avl_first(&sdp->sd_snaps);
1222         while (sep != NULL) {
```



```
1223                 next = AVL_NEXT(&sdp->sd_snaps, sep);

1225                 if (sep->se_root == vp) {
1226                         avl_remove(&sdp->sd_snaps, sep);
1227                         kmem_free(sep->se_name, strlen(sep->se_name) + 1);
1228                         kmem_free(sep, sizeof (zfs_snapentry_t));
1229                         break;
1230                 }
1231                 sep = next;
1232         }
1233         ASSERT(sep != NULL);

1235         mutex_exit(&sdp->sd_lock);
1236         VN_RELE(dvp);

1238         /*
1239          * Dispose of the vnode for the snapshot mount point.
1240          * This is safe to do because once this entry has been removed
1241          * from the AVL tree, it can't be found again, so cannot become
1242          * "active".  If we lookup the same name again we will end up
1243          * creating a new vnode.
1244          */
1245         gfs_vop_inactive(vp, cr, ct);
1246 }


1249 /*
1250  * These VP's should never see the light of day.  They should always
1251  * be covered.
1252  */
1253 static const fs_operation_def_t zfsctl_tops_snapshot[] = {
1254         VOPNAME_INACTIVE, { .vop_inactive =  zfsctl_snapshot_inactive },
1255         NULL, NULL
1256 };

1258 int
1259 zfsctl_lookup_objset(vfs_t *vfsp, uint64_t objsetid, zfsvfs_t **zfsvfsp)
1260 {
1261         zfsvfs_t *zfsvfs = vfsp->vfs_data;
1262         vnode_t *dvp, *vp;
1263         zfsctl_snapdir_t *sdp;
1264         zfsctl_node_t *zcp;
1265         zfs_snapentry_t *sep;
1266         int error;

1268         ASSERT(zfsvfs->z_ctldir != NULL);
1269         error = zfsctl_root_lookup(zfsvfs->z_ctldir, "snapshot", &dvp,
1270             NULL, 0, NULL, kcred, NULL, NULL, NULL);
1271         if (error != 0)
1272                 return (error);
1273         sdp = dvp->v_data;

1275         mutex_enter(&sdp->sd_lock);
1276         sep = avl_first(&sdp->sd_snaps);
1277         while (sep != NULL) {
1278                 vp = sep->se_root;
1279                 zcp = vp->v_data;
1280                 if (zcp->zc_id == objsetid)
1281                         break;

1283                 sep = AVL_NEXT(&sdp->sd_snaps, sep);
1284         }

1286         if (sep != NULL) {
1287                 VN_HOLD(vp);
1288                 /*
```

```
1289                  * Return the mounted root rather than the covered mount point.
1290                  * Takes the GFS vnode at .zfs/snapshot/<snapshot objsetid>
1291                  * and returns the ZFS vnode mounted on top of the GFS node.
1292                  * This ZFS vnode is the root of the vfs for objset 'objsetid'.
1293                  */
1294                 error = traverse(&vp);
1295                 if (error == 0) {
1296                         if (vp == sep->se_root)
1297                                 error = SET_ERROR(EINVAL);
1298                         else
1299                                 *zfsvfsp = VTOZ(vp)->z_zfsvfs;
1300                 }
1301                 mutex_exit(&sdp->sd_lock);
1302                 VN_RELE(vp);
1303         } else {
1304                 error = SET_ERROR(EINVAL);
1305                 mutex_exit(&sdp->sd_lock);
1306         }

1308         VN_RELE(dvp);

1310         return (error);
1311 }

1313 /*
1314  * Unmount any snapshots for the given filesystem.  This is called from
1315  * zfs_umount() - if we have a ctldir, then go through and unmount all the
1316  * snapshots.
1317  */
1318 int
1319 zfsctl_umount_snapshots(vfs_t *vfsp, int fflags, cred_t *cr)
1320 {
1321         zfsvfs_t *zfsvfs = vfsp->vfs_data;
1322         vnode_t *dvp;
1323         zfsctl_snapdir_t *sdp;
1324         zfs_snapentry_t *sep, *next;
1325         int error;

1327         ASSERT(zfsvfs->z_ctldir != NULL);
1328         error = zfsctl_root_lookup(zfsvfs->z_ctldir, "snapshot", &dvp,
1329             NULL, 0, NULL, cr, NULL, NULL, NULL);
1330         if (error != 0)
1331                 return (error);
1332         sdp = dvp->v_data;

1334         mutex_enter(&sdp->sd_lock);

1336         sep = avl_first(&sdp->sd_snaps);
1337         while (sep != NULL) {
1338                 next = AVL_NEXT(&sdp->sd_snaps, sep);

1340                 /*
1341                  * If this snapshot is not mounted, then it must
1342                  * have just been unmounted by somebody else, and
1343                  * will be cleaned up by zfsctl_snapdir_inactive().
1344                  */
1345                 if (vn_ismntpt(sep->se_root)) {
1346                         avl_remove(&sdp->sd_snaps, sep);
1347                         error = zfsctl_unmount_snap(sep, fflags, cr);
1348                         if (error) {
1349                                 avl_add(&sdp->sd_snaps, sep);
1350                                 break;
1351                         }
1352                 }
1353                 sep = next;
1354         }
```



```
1356         mutex_exit(&sdp->sd_lock);
1357         VN_RELE(dvp);

1359         return (error);
1360 }
```