```
**********************************************************
  194384 Wed Sep  9 19:47:42 2015
new/usr/src/uts/common/fs/zfs/arc.c
6214 zpools going south
**********************************************************
______unchanged_portion_omitted_

 739 typedef struct l2arc_dev l2arc_dev_t;

 741 typedef struct l2arc_buf_hdr {
 742         /* protected by arc_buf_hdr mutex */
 743         l2arc_dev_t             *b_dev;         /* L2ARC device */
 744         uint64_t                b_daddr;        /* disk address, offset byte */
 745         /* real alloc'd buffer size depending on b_compress applied */
 746         int32_t                 b_asize;
 747         uint8_t                 b_compress;
 748 #endif /* ! codereview */

 750         list_node_t             b_l2node;
 751 } l2arc_buf_hdr_t;

 753 struct arc_buf_hdr {
 754         /* protected by hash lock */
 755         dva_t                   b_dva;
 756         uint64_t                b_birth;
 757         /*
 758          * Even though this checksum is only set/verified when a buffer is in
 759          * the L1 cache, it needs to be in the set of common fields because it
 760          * must be preserved from the time before a buffer is written out to
 761          * L2ARC until after it is read back in.
 762          */
 763         zio_cksum_t             *b_freeze_cksum;

 765         arc_buf_hdr_t           *b_hash_next;
 766         arc_flags_t             b_flags;

 768         /* immutable */
 769         int32_t                 b_size;
 770         uint64_t                b_spa;

 772         /* L2ARC fields. Undefined when not in L2ARC. */
 773         l2arc_buf_hdr_t         b_l2hdr;
 774         /* L1ARC fields. Undefined when in l2arc_only state */
 775         l1arc_buf_hdr_t         b_l1hdr;
 776 };

 778 static arc_buf_t *arc_eviction_list;
 779 static arc_buf_hdr_t arc_eviction_hdr;

 781 #define GHOST_STATE(state)      \
 782         ((state) == arc_mru_ghost || (state) == arc_mfu_ghost ||        \
 783         (state) == arc_l2c_only)

 785 #define HDR_IN_HASH_TABLE(hdr)  ((hdr)->b_flags & ARC_FLAG_IN_HASH_TABLE)
 786 #define HDR_IO_IN_PROGRESS(hdr) ((hdr)->b_flags & ARC_FLAG_IO_IN_PROGRESS)
 787 #define HDR_IO_ERROR(hdr)       ((hdr)->b_flags & ARC_FLAG_IO_ERROR)
 788 #define HDR_PREFETCH(hdr)       ((hdr)->b_flags & ARC_FLAG_PREFETCH)
 789 #define HDR_FREED_IN_READ(hdr)  ((hdr)->b_flags & ARC_FLAG_FREED_IN_READ)
 790 #define HDR_BUF_AVAILABLE(hdr)  ((hdr)->b_flags & ARC_FLAG_BUF_AVAILABLE)

 792 #define HDR_L2CACHE(hdr)        ((hdr)->b_flags & ARC_FLAG_L2CACHE)
 793 #define HDR_L2COMPRESS(hdr)     ((hdr)->b_flags & ARC_FLAG_L2COMPRESS)
 794 #define HDR_L2_READING(hdr)     \
 795             (((hdr)->b_flags & ARC_FLAG_IO_IN_PROGRESS) &&      \
 796             ((hdr)->b_flags & ARC_FLAG_HAS_L2HDR))
 797 #define HDR_L2_WRITING(hdr)     ((hdr)->b_flags & ARC_FLAG_L2_WRITING)
```



```
 798 #define HDR_L2_EVICTED(hdr)     ((hdr)->b_flags & ARC_FLAG_L2_EVICTED)
 799 #define HDR_L2_WRITE_HEAD(hdr)  ((hdr)->b_flags & ARC_FLAG_L2_WRITE_HEAD)

 801 #define HDR_ISTYPE_METADATA(hdr)        \
 802             ((hdr)->b_flags & ARC_FLAG_BUFC_METADATA)
 803 #define HDR_ISTYPE_DATA(hdr)    (!HDR_ISTYPE_METADATA(hdr))

 805 #define HDR_HAS_L1HDR(hdr)      ((hdr)->b_flags & ARC_FLAG_HAS_L1HDR)
 806 #define HDR_HAS_L2HDR(hdr)      ((hdr)->b_flags & ARC_FLAG_HAS_L2HDR)

 747 /* For storing compression mode in b_flags */
 748 #define HDR_COMPRESS_OFFSET     24
 749 #define HDR_COMPRESS_NBITS      7

 751 #define HDR_GET_COMPRESS(hdr)   ((enum zio_compress)BF32_GET(hdr->b_flags, \
 752             HDR_COMPRESS_OFFSET, HDR_COMPRESS_NBITS))
 753 #define HDR_SET_COMPRESS(hdr, cmp) BF32_SET(hdr->b_flags, \
 754             HDR_COMPRESS_OFFSET, HDR_COMPRESS_NBITS, (cmp))

 808 /*
 809  * Other sizes
 810  */

 812 #define HDR_FULL_SIZE ((int64_t)sizeof (arc_buf_hdr_t))
 813 #define HDR_L2ONLY_SIZE ((int64_t)offsetof(arc_buf_hdr_t, b_l1hdr))

 815 /*
 816  * Hash table routines
 817  */

 819 #define HT_LOCK_PAD     64

 821 struct ht_lock {
 822         kmutex_t        ht_lock;
 823 #ifdef _KERNEL
 824         unsigned char   pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
 825 #endif
 826 };
______unchanged_portion_omitted_

2006 static void
2007 arc_buf_l2_cdata_free(arc_buf_hdr_t *hdr)
2008 {
2009         ASSERT(HDR_HAS_L2HDR(hdr));
2010         ASSERT(MUTEX_HELD(&hdr->b_l2hdr.b_dev->l2ad_mtx));

2012         /*
2013          * The b_tmp_cdata field is linked off of the b_l1hdr, so if
2014          * that doesn't exist, the header is in the arc_l2c_only state,
2015          * and there isn't anything to free (it's already been freed).
2016          */
2017         if (!HDR_HAS_L1HDR(hdr))
2018                 return;

2020         /*
2021          * The header isn't being written to the l2arc device, thus it
2022          * shouldn't have a b_tmp_cdata to free.
2023          */
2024         if (!HDR_L2_WRITING(hdr)) {
2025                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
2026                 return;
2027         }

2029         /*
2030          * The header does not have compression enabled. This can be due
2031          * to the buffer not being compressible, or because we're
```

```
2032          * freeing the buffer before the second phase of
2033          * l2arc_write_buffer() has started (which does the compression
2034          * step). In either case, b_tmp_cdata does not point to a
2035          * separately compressed buffer, so there's nothing to free (it
2036          * points to the same buffer as the arc_buf_t's b_data field).
2037          */
2038         if (hdr->b_l2hdr.b_compress == ZIO_COMPRESS_OFF) {
1986         if (HDR_GET_COMPRESS(hdr) == ZIO_COMPRESS_OFF) {
2039                 hdr->b_l1hdr.b_tmp_cdata = NULL;
2040                 return;
2041         }

2043         /*
2044          * There's nothing to free since the buffer was all zero's and
2045          * compressed to a zero length buffer.
2046          */
2047         if (hdr->b_l2hdr.b_compress == ZIO_COMPRESS_EMPTY) {
1995         if (HDR_GET_COMPRESS(hdr) == ZIO_COMPRESS_EMPTY) {
2048                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
2049                 return;
2050         }

2052         ASSERT(L2ARC_IS_VALID_COMPRESS(hdr->b_l2hdr.b_compress));
2000         ASSERT(L2ARC_IS_VALID_COMPRESS(HDR_GET_COMPRESS(hdr)));

2054         arc_buf_free_on_write(hdr->b_l1hdr.b_tmp_cdata,
2055             hdr->b_size, zio_data_buf_free);

2057         ARCSTAT_BUMP(arcstat_l2_cdata_free_on_write);
2058         hdr->b_l1hdr.b_tmp_cdata = NULL;
2059 }
_____unchanged_portion_omitted_

3959 /*
3960  * "Read" the block at the specified DVA (in bp) via the
3961  * cache.  If the block is found in the cache, invoke the provided
3962  * callback immediately and return.  Note that the 'zio' parameter
3963  * in the callback will be NULL in this case, since no IO was
3964  * required.  If the block is not in the cache pass the read request
3965  * on to the spa with a substitute callback function, so that the
3966  * requested block will be added to the cache.
3967  *
3968  * If a read request arrives for a block that has a read in-progress,
3969  * either wait for the in-progress read to complete (and return the
3970  * results); or, if this is a read with a "done" func, add a record
3971  * to the read to invoke the "done" func when the read completes,
3972  * and return; or just return.
3973  *
3974  * arc_read_done() will invoke all the requested "done" functions
3975  * for readers of this block.
3976  */
3977 int
3978 arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
3979     void *private, zio_priority_t priority, int zio_flags,
3980     arc_flags_t *arc_flags, const zbookmark_phys_t *zb)
3981 {
3982         arc_buf_hdr_t *hdr = NULL;
3983         arc_buf_t *buf = NULL;
3984         kmutex_t *hash_lock = NULL;
3985         zio_t *rzio;
3986         uint64_t guid = spa_load_guid(spa);

3988         ASSERT(!BP_IS_EMBEDDED(bp) ||
3989             BPE_GET_ETYPE(bp) == BP_EMBEDDED_TYPE_DATA);

3991 top:
```



```
3992         if (!BP_IS_EMBEDDED(bp)) {
3993                 /*
3994                  * Embedded BP's have no DVA and require no I/O to "read".
3995                  * Create an anonymous arc buf to back it.
3996                  */
3997                 hdr = buf_hash_find(guid, bp, &hash_lock);
3998         }

4000         if (hdr != NULL && HDR_HAS_L1HDR(hdr) && hdr->b_l1hdr.b_datacnt > 0) {

4002                 *arc_flags |= ARC_FLAG_CACHED;

4004                 if (HDR_IO_IN_PROGRESS(hdr)) {

4006                         if (*arc_flags & ARC_FLAG_WAIT) {
4007                                 cv_wait(&hdr->b_l1hdr.b_cv, hash_lock);
4008                                 mutex_exit(hash_lock);
4009                                 goto top;
4010                         }
4011                         ASSERT(*arc_flags & ARC_FLAG_NOWAIT);

4013                         if (done) {
4014                                 arc_callback_t  *acb = NULL;

4016                                 acb = kmem_zalloc(sizeof (arc_callback_t),
4017                                     KM_SLEEP);
4018                                 acb->acb_done = done;
4019                                 acb->acb_private = private;
4020                                 if (pio != NULL)
4021                                         acb->acb_zio_dummy = zio_null(pio,
4022                                             spa, NULL, NULL, NULL, zio_flags);

4024                                 ASSERT(acb->acb_done != NULL);
4025                                 acb->acb_next = hdr->b_l1hdr.b_acb;
4026                                 hdr->b_l1hdr.b_acb = acb;
4027                                 add_reference(hdr, hash_lock, private);
4028                                 mutex_exit(hash_lock);
4029                                 return (0);
4030                         }
4031                         mutex_exit(hash_lock);
4032                         return (0);
4033                 }

4035                 ASSERT(hdr->b_l1hdr.b_state == arc_mru ||
4036                     hdr->b_l1hdr.b_state == arc_mfu);

4038                 if (done) {
4039                         add_reference(hdr, hash_lock, private);
4040                         /*
4041                          * If this block is already in use, create a new
4042                          * copy of the data so that we will be guaranteed
4043                          * that arc_release() will always succeed.
4044                          */
4045                         buf = hdr->b_l1hdr.b_buf;
4046                         ASSERT(buf);
4047                         ASSERT(buf->b_data);
4048                         if (HDR_BUF_AVAILABLE(hdr)) {
4049                                 ASSERT(buf->b_efunc == NULL);
4050                                 hdr->b_flags &= ~ARC_FLAG_BUF_AVAILABLE;
4051                         } else {
4052                                 buf = arc_buf_clone(buf);
4053                         }

4055                 } else if (*arc_flags & ARC_FLAG_PREFETCH &&
4056                     refcount_count(&hdr->b_l1hdr.b_refcnt) == 0) {
4057                         hdr->b_flags |= ARC_FLAG_PREFETCH;
```

```
4058                 }
4059                 DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
4060                 arc_access(hdr, hash_lock);
4061                 if (*arc_flags & ARC_FLAG_L2CACHE)
4062                         hdr->b_flags |= ARC_FLAG_L2CACHE;
4063                 if (*arc_flags & ARC_FLAG_L2COMPRESS)
4064                         hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4065                 mutex_exit(hash_lock);
4066                 ARCSTAT_BUMP(arcstat_hits);
4067                 ARCSTAT_CONDSTAT(!HDR_PREFETCH(hdr),
4068                     demand, prefetch, !HDR_ISTYPE_METADATA(hdr),
4069                     data, metadata, hits);

4071                 if (done)
4072                         done(NULL, buf, private);
4073         } else {
4074                 uint64_t size = BP_GET_LSIZE(bp);
4075                 arc_callback_t *acb;
4076                 vdev_t *vd = NULL;
4077                 uint64_t addr = 0;
4078                 boolean_t devw = B_FALSE;
4079                 enum zio_compress b_compress = ZIO_COMPRESS_OFF;
4080                 int32_t b_asize = 0;

4082                 if (hdr == NULL) {
4083                         /* this block is not in the cache */
4084                         arc_buf_hdr_t *exists = NULL;
4085                         arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
4086                         buf = arc_buf_alloc(spa, size, private, type);
4087                         hdr = buf->b_hdr;
4088                         if (!BP_IS_EMBEDDED(bp)) {
4089                                 hdr->b_dva = *BP_IDENTITY(bp);
4090                                 hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
4091                                 exists = buf_hash_insert(hdr, &hash_lock);
4092                         }
4093                         if (exists != NULL) {
4094                                 /* somebody beat us to the hash insert */
4095                                 mutex_exit(hash_lock);
4096                                 buf_discard_identity(hdr);
4097                                 (void) arc_buf_remove_ref(buf, private);
4098                                 goto top; /* restart the IO request */
4099                         }

4101                         /* if this is a prefetch, we don't have a reference */
4102                         if (*arc_flags & ARC_FLAG_PREFETCH) {
4103                                 (void) remove_reference(hdr, hash_lock,
4104                                     private);
4105                                 hdr->b_flags |= ARC_FLAG_PREFETCH;
4106                         }
4107                         if (*arc_flags & ARC_FLAG_L2CACHE)
4108                                 hdr->b_flags |= ARC_FLAG_L2CACHE;
4109                         if (*arc_flags & ARC_FLAG_L2COMPRESS)
4110                                 hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4111                         if (BP_GET_LEVEL(bp) > 0)
4112                                 hdr->b_flags |= ARC_FLAG_INDIRECT;
4113                 } else {
4114                         /*
4115                          * This block is in the ghost cache. If it was L2-only
4116                          * (and thus didn't have an L1 hdr), we realloc the
4117                          * header to add an L1 hdr.
4118                          */
4119                         if (!HDR_HAS_L1HDR(hdr)) {
4120                                 hdr = arc_hdr_realloc(hdr, hdr_l2only_cache,
4121                                     hdr_full_cache);
4122                         }
```



```
4124                         ASSERT(GHOST_STATE(hdr->b_l1hdr.b_state));
4125                         ASSERT(!HDR_IO_IN_PROGRESS(hdr));
4126                         ASSERT(refcount_is_zero(&hdr->b_l1hdr.b_refcnt));
4127                         ASSERT3P(hdr->b_l1hdr.b_buf, ==, NULL);

4129                         /* if this is a prefetch, we don't have a reference */
4130                         if (*arc_flags & ARC_FLAG_PREFETCH)
4131                                 hdr->b_flags |= ARC_FLAG_PREFETCH;
4132                         else
4133                                 add_reference(hdr, hash_lock, private);
4134                         if (*arc_flags & ARC_FLAG_L2CACHE)
4135                                 hdr->b_flags |= ARC_FLAG_L2CACHE;
4136                         if (*arc_flags & ARC_FLAG_L2COMPRESS)
4137                                 hdr->b_flags |= ARC_FLAG_L2COMPRESS;
4138                         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
4139                         buf->b_hdr = hdr;
4140                         buf->b_data = NULL;
4141                         buf->b_efunc = NULL;
4142                         buf->b_private = NULL;
4143                         buf->b_next = NULL;
4144                         hdr->b_l1hdr.b_buf = buf;
4145                         ASSERT0(hdr->b_l1hdr.b_datacnt);
4146                         hdr->b_l1hdr.b_datacnt = 1;
4147                         arc_get_data_buf(buf);
4148                         arc_access(hdr, hash_lock);
4149                 }

4151                 ASSERT(!GHOST_STATE(hdr->b_l1hdr.b_state));

4153                 acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
4154                 acb->acb_done = done;
4155                 acb->acb_private = private;

4157                 ASSERT(hdr->b_l1hdr.b_acb == NULL);
4158                 hdr->b_l1hdr.b_acb = acb;
4159                 hdr->b_flags |= ARC_FLAG_IO_IN_PROGRESS;

4161                 if (HDR_HAS_L2HDR(hdr) &&
4162                     (vd = hdr->b_l2hdr.b_dev->l2ad_vdev) != NULL) {
4163                         devw = hdr->b_l2hdr.b_dev->l2ad_writing;
4164                         addr = hdr->b_l2hdr.b_daddr;
4165                         b_compress = hdr->b_l2hdr.b_compress;
4113                         b_compress = HDR_GET_COMPRESS(hdr);
4166                         b_asize = hdr->b_l2hdr.b_asize;
4167                         /*
4168                          * Lock out device removal.
4169                          */
4170                         if (vdev_is_dead(vd) ||
4171                             !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
4172                                 vd = NULL;
4173                 }

4175                 if (hash_lock != NULL)
4176                         mutex_exit(hash_lock);

4178                 /*
4179                  * At this point, we have a level 1 cache miss.  Try again in
4180                  * L2ARC if possible.
4181                  */
4182                 ASSERT3U(hdr->b_size, ==, size);
4183                 DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
4184                     uint64_t, size, zbookmark_phys_t *, zb);
4185                 ARCSTAT_BUMP(arcstat_misses);
4186                 ARCSTAT_CONDSTAT(!HDR_PREFETCH(hdr),
4187                     demand, prefetch, !HDR_ISTYPE_METADATA(hdr),
4188                     data, metadata, misses);
```

```
4190                if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
4191                        /*
4192                         * Read from the L2ARC if the following are true:
4193                         * 1. The L2ARC vdev was previously cached.
4194                         * 2. This buffer still has L2ARC metadata.
4195                         * 3. This buffer isn't currently writing to the L2ARC.
4196                         * 4. The L2ARC entry wasn't evicted, which may
4197                         *    also have invalidated the vdev.
4198                         * 5. This isn't prefetch and l2arc_noprefetch is set.
4199                         */
4200                        if (HDR_HAS_L2HDR(hdr) &&
4201                            !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
4202                            !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
4203                                l2arc_read_callback_t *cb;

4205                                DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
4206                                ARCSTAT_BUMP(arcstat_l2_hits);

4208                                cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
4209                                    KM_SLEEP);
4210                                cb->l2rcb_buf = buf;
4211                                cb->l2rcb_spa = spa;
4212                                cb->l2rcb_bp = *bp;
4213                                cb->l2rcb_zb = *zb;
4214                                cb->l2rcb_flags = zio_flags;
4215                                cb->l2rcb_compress = b_compress;

4217                                ASSERT(addr >= VDEV_LABEL_START_SIZE &&
4218                                    addr + size < vd->vdev_psize -
4219                                    VDEV_LABEL_END_SIZE);

4221                                /*
4222                                 * l2arc read.  The SCL_L2ARC lock will be
4223                                 * released by l2arc_read_done().
4224                                 * Issue a null zio if the underlying buffer
4225                                 * was squashed to zero size by compression.
4226                                 */
4227                                if (b_compress == ZIO_COMPRESS_EMPTY) {
4228                                        rzio = zio_null(pio, spa, vd,
4229                                            l2arc_read_done, cb,
4230                                            zio_flags | ZIO_FLAG_DONT_CACHE |
4231                                            ZIO_FLAG_CANFAIL |
4232                                            ZIO_FLAG_DONT_PROPAGATE |
4233                                            ZIO_FLAG_DONT_RETRY);
4234                                } else {
4235                                        rzio = zio_read_phys(pio, vd, addr,
4236                                            b_asize, buf->b_data,
4237                                            ZIO_CHECKSUM_OFF,
4238                                            l2arc_read_done, cb, priority,
4239                                            zio_flags | ZIO_FLAG_DONT_CACHE |
4240                                            ZIO_FLAG_CANFAIL |
4241                                            ZIO_FLAG_DONT_PROPAGATE |
4242                                            ZIO_FLAG_DONT_RETRY, B_FALSE);
4243                                }
4244                                DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
4245                                    zio_t *, rzio);
4246                                ARCSTAT_INCR(arcstat_l2_read_bytes, b_asize);

4248                                if (*arc_flags & ARC_FLAG_NOWAIT) {
4249                                        zio_nowait(rzio);
4250                                        return (0);
4251                                }

4253                                ASSERT(*arc_flags & ARC_FLAG_WAIT);
4254                                if (zio_wait(rzio) == 0)
```



```
4255                                        return (0);

4257                                /* l2arc read error; goto zio_read() */
4258                        } else {
4259                                DTRACE_PROBE1(l2arc__miss,
4260                                    arc_buf_hdr_t *, hdr);
4261                                ARCSTAT_BUMP(arcstat_l2_misses);
4262                                if (HDR_L2_WRITING(hdr))
4263                                        ARCSTAT_BUMP(arcstat_l2_rw_clash);
4264                                spa_config_exit(spa, SCL_L2ARC, vd);
4265                        }
4266                } else {
4267                        if (vd != NULL)
4268                                spa_config_exit(spa, SCL_L2ARC, vd);
4269                        if (l2arc_ndev != 0) {
4270                                DTRACE_PROBE1(l2arc__miss,
4271                                    arc_buf_hdr_t *, hdr);
4272                                ARCSTAT_BUMP(arcstat_l2_misses);
4273                        }
4274                }

4276                rzio = zio_read(pio, spa, bp, buf->b_data, size,
4277                    arc_read_done, buf, priority, zio_flags, zb);

4279                if (*arc_flags & ARC_FLAG_WAIT)
4280                        return (zio_wait(rzio));

4282                ASSERT(*arc_flags & ARC_FLAG_NOWAIT);
4283                zio_nowait(rzio);
4284        }
4285        return (0);
4286 }
```

unchanged_portion_omitted_

```
5592 /*
5593  * A read to a cache device completed.  Validate buffer contents before
5594  * handing over to the regular ARC routines.
5595  */
5596 static void
5597 l2arc_read_done(zio_t *zio)
5598 {
5599         l2arc_read_callback_t *cb;
5600         arc_buf_hdr_t *hdr;
5601         arc_buf_t *buf;
5602         kmutex_t *hash_lock;
5603         int equal;

5605         ASSERT(zio->io_vd != NULL);
5606         ASSERT(zio->io_flags & ZIO_FLAG_DONT_PROPAGATE);

5608         spa_config_exit(zio->io_spa, SCL_L2ARC, zio->io_vd);

5610         cb = zio->io_private;
5611         ASSERT(cb != NULL);
5612         buf = cb->l2rcb_buf;
5613         ASSERT(buf != NULL);

5615         hash_lock = HDR_LOCK(buf->b_hdr);
5616         mutex_enter(hash_lock);
5617         hdr = buf->b_hdr;
5618         ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));

5620         /*
5621          * If the buffer was compressed, decompress it first.
5622          */
5623         if (cb->l2rcb_compress != ZIO_COMPRESS_OFF)
```

```
5624                l2arc_decompress_zio(zio, hdr, cb->l2rcb_compress);
5625        ASSERT(zio->io_data != NULL);
5626        ASSERT3U(zio->io_size, ==, hdr->b_size);
5627        ASSERT3U(BP_GET_LSIZE(&cb->l2rcb_bp), ==, hdr->b_size);
5628 #endif /* ! codereview */

5630        /*
5631         * Check this survived the L2ARC journey.
5632         */
5633        equal = arc_cksum_equal(buf);
5634        if (equal && zio->io_error == 0 && !HDR_L2_EVICTED(hdr)) {
5635                mutex_exit(hash_lock);
5636                zio->io_private = buf;
5637                zio->io_bp_copy = cb->l2rcb_bp; /* XXX fix in L2ARC 2.0 */
5638                zio->io_bp = &zio->io_bp_copy;  /* XXX fix in L2ARC 2.0 */
5639                arc_read_done(zio);
5640        } else {
5641                mutex_exit(hash_lock);
5642                /*
5643                 * Buffer didn't survive caching.  Increment stats and
5644                 * reissue to the original storage device.
5645                 */
5646                if (zio->io_error != 0) {
5647                        ARCSTAT_BUMP(arcstat_l2_io_error);
5648                } else {
5649                        zio->io_error = SET_ERROR(EIO);
5650                }
5651                if (!equal)
5652                        ARCSTAT_BUMP(arcstat_l2_cksum_bad);

5654                /*
5655                 * If there's no waiter, issue an async i/o to the primary
5656                 * storage now.  If there *is* a waiter, the caller must
5657                 * issue the i/o in a context where it's OK to block.
5658                 */
5659                if (zio->io_waiter == NULL) {
5660                        zio_t *pio = zio_unique_parent(zio);

5662                        ASSERT(!pio || pio->io_child_type == ZIO_CHILD_LOGICAL);

5664                        zio_nowait(zio_read(pio, cb->l2rcb_spa, &cb->l2rcb_bp,
5665                            buf->b_data, hdr->b_size, arc_read_done, buf,
5574                            buf->b_data, zio->io_size, arc_read_done, buf,
5666                            zio->io_priority, cb->l2rcb_flags, &cb->l2rcb_zb));
5667                }
5668        }

5670        kmem_free(cb, sizeof (l2arc_read_callback_t));
5671 }
______unchanged_portion_omitted_

5830 /*
5831  * Find and write ARC buffers to the L2ARC device.
5832  *
5833  * An ARC_FLAG_L2_WRITING flag is set so that the L2ARC buffers are not valid
5834  * for reading until they have completed writing.
5835  * The headroom_boost is an in-out parameter used to maintain headroom boost
5836  * state between calls to this function.
5837  *
5838  * Returns the number of bytes actually written (which may be smaller than
5839  * the delta by which the device hand has changed due to alignment).
5840  */
5841 static uint64_t
5842 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
5843     boolean_t *headroom_boost)
5844 {
```



```
5845        arc_buf_hdr_t *hdr, *hdr_prev, *head;
5846        uint64_t write_asize, write_psize, write_sz, headroom,
5847            buf_compress_minsz;
5848        void *buf_data;
5849        boolean_t full;
5850        l2arc_write_callback_t *cb;
5851        zio_t *pio, *wzio;
5852        uint64_t guid = spa_load_guid(spa);
5853        const boolean_t do_headroom_boost = *headroom_boost;

5855        ASSERT(dev->l2ad_vdev != NULL);

5857        /* Lower the flag now, we might want to raise it again later. */
5858        *headroom_boost = B_FALSE;

5860        pio = NULL;
5861        write_sz = write_asize = write_psize = 0;
5862        full = B_FALSE;
5863        head = kmem_cache_alloc(hdr_l2only_cache, KM_PUSHPAGE);
5864        head->b_flags |= ARC_FLAG_L2_WRITE_HEAD;
5865        head->b_flags |= ARC_FLAG_HAS_L2HDR;

5867        /*
5868         * We will want to try to compress buffers that are at least 2x the
5869         * device sector size.
5870         */
5871        buf_compress_minsz = 2 << dev->l2ad_vdev->vdev_ashift;

5873        /*
5874         * Copy buffers for L2ARC writing.
5875         */
5876        for (int try = 0; try <= 3; try++) {
5877                multilist_sublist_t *mls = l2arc_sublist_lock(try);
5878                uint64_t passed_sz = 0;

5880                /*
5881                 * L2ARC fast warmup.
5882                 *
5883                 * Until the ARC is warm and starts to evict, read from the
5884                 * head of the ARC lists rather than the tail.
5885                 */
5886                if (arc_warm == B_FALSE)
5887                        hdr = multilist_sublist_head(mls);
5888                else
5889                        hdr = multilist_sublist_tail(mls);

5891                headroom = target_sz * l2arc_headroom;
5892                if (do_headroom_boost)
5893                        headroom = (headroom * l2arc_headroom_boost) / 100;

5895                for (; hdr; hdr = hdr_prev) {
5896                        kmutex_t *hash_lock;
5897                        uint64_t buf_sz;

5899                        if (arc_warm == B_FALSE)
5900                                hdr_prev = multilist_sublist_next(mls, hdr);
5901                        else
5902                                hdr_prev = multilist_sublist_prev(mls, hdr);

5904                        hash_lock = HDR_LOCK(hdr);
5905                        if (!mutex_tryenter(hash_lock)) {
5906                                /*
5907                                 * Skip this buffer rather than waiting.
5908                                 */
5909                                continue;
5910                        }
```

```
5912                        passed_sz += hdr->b_size;
5913                        if (passed_sz > headroom) {
5914                                /*
5915                                 * Searched too far.
5916                                 */
5917                                mutex_exit(hash_lock);
5918                                break;
5919                        }

5921                        if (!l2arc_write_eligible(guid, hdr)) {
5922                                mutex_exit(hash_lock);
5923                                continue;
5924                        }

5926                        if ((write_sz + hdr->b_size) > target_sz) {
5927                                full = B_TRUE;
5928                                mutex_exit(hash_lock);
5929                                break;
5930                        }

5932                        if (pio == NULL) {
5933                                /*
5934                                 * Insert a dummy header on the buflist so
5935                                 * l2arc_write_done() can find where the
5936                                 * write buffers begin without searching.
5937                                 */
5938                                mutex_enter(&dev->l2ad_mtx);
5939                                list_insert_head(&dev->l2ad_buflist, head);
5940                                mutex_exit(&dev->l2ad_mtx);

5942                                cb = kmem_alloc(
5943                                    sizeof (l2arc_write_callback_t), KM_SLEEP);
5944                                cb->l2wcb_dev = dev;
5945                                cb->l2wcb_head = head;
5946                                pio = zio_root(spa, l2arc_write_done, cb,
5947                                    ZIO_FLAG_CANFAIL);
5948                        }

5950                        /*
5951                         * Create and add a new L2ARC header.
5952                         */
5953                        hdr->b_l2hdr.b_dev = dev;
5954                        hdr->b_flags |= ARC_FLAG_L2_WRITING;
5955                        /*
5956                         * Temporarily stash the data buffer in b_tmp_cdata.
5957                         * The subsequent write step will pick it up from
5958                         * there. This is because can't access b_l1hdr.b_buf
5959                         * without holding the hash_lock, which we in turn
5960                         * can't access without holding the ARC list locks
5961                         * (which we want to avoid during compression/writing).
5962                         */
5963                        hdr->b_l2hdr.b_compress = ZIO_COMPRESS_OFF;
5872                        HDR_SET_COMPRESS(hdr, ZIO_COMPRESS_OFF);
5964                        hdr->b_l2hdr.b_asize = hdr->b_size;
5965                        hdr->b_l1hdr.b_tmp_cdata = hdr->b_l1hdr.b_buf->b_data;

5967                        /*
5968                         * Explicitly set the b_daddr field to a known
5969                         * value which means "invalid address". This
5970                         * enables us to differentiate which stage of
5971                         * l2arc_write_buffers() the particular header
5972                         * is in (e.g. this loop, or the one below).
5973                         * ARC_FLAG_L2_WRITING is not enough to make
5974                         * this distinction, and we need to know in
5975                         * order to do proper l2arc vdev accounting in
```



```
5976                         * arc_release() and arc_hdr_destroy().
5977                         *
5978                         * Note, we can't use a new flag to distinguish
5979                         * the two stages because we don't hold the
5980                         * header's hash_lock below, in the second stage
5981                         * of this function. Thus, we can't simply
5982                         * change the b_flags field to denote that the
5983                         * IO has been sent. We can change the b_daddr
5984                         * field of the L2 portion, though, since we'll
5985                         * be holding the l2ad_mtx; which is why we're
5986                         * using it to denote the header's state change.
5987                         */
5988                        hdr->b_l2hdr.b_daddr = L2ARC_ADDR_UNSET;

5990                        buf_sz = hdr->b_size;
5991                        hdr->b_flags |= ARC_FLAG_HAS_L2HDR;

5993                        mutex_enter(&dev->l2ad_mtx);
5994                        list_insert_head(&dev->l2ad_buflist, hdr);
5995                        mutex_exit(&dev->l2ad_mtx);

5997                        /*
5998                         * Compute and store the buffer cksum before
5999                         * writing.  On debug the cksum is verified first.
6000                         */
6001                        arc_cksum_verify(hdr->b_l1hdr.b_buf);
6002                        arc_cksum_compute(hdr->b_l1hdr.b_buf, B_TRUE);

6004                        mutex_exit(hash_lock);

6006                        write_sz += buf_sz;
6007                }

6009                multilist_sublist_unlock(mls);

6011                if (full == B_TRUE)
6012                        break;
6013        }

6015        /* No buffers selected for writing? */
6016        if (pio == NULL) {
6017                ASSERT0(write_sz);
6018                ASSERT(!HDR_HAS_L1HDR(head));
6019                kmem_cache_free(hdr_l2only_cache, head);
6020                return (0);
6021        }

6023        mutex_enter(&dev->l2ad_mtx);

6025        /*
6026         * Now start writing the buffers. We're starting at the write head
6027         * and work backwards, retracing the course of the buffer selector
6028         * loop above.
6029         */
6030        for (hdr = list_prev(&dev->l2ad_buflist, head); hdr;
6031            hdr = list_prev(&dev->l2ad_buflist, hdr)) {
6032                uint64_t buf_sz;

6034                /*
6035                 * We rely on the L1 portion of the header below, so
6036                 * it's invalid for this header to have been evicted out
6037                 * of the ghost cache, prior to being written out. The
6038                 * ARC_FLAG_L2_WRITING bit ensures this won't happen.
6039                 */
6040                ASSERT(HDR_HAS_L1HDR(hdr));
```

```
6042                /*
6043                 * We shouldn't need to lock the buffer here, since we flagged
6044                 * it as ARC_FLAG_L2_WRITING in the previous step, but we must
6045                 * take care to only access its L2 cache parameters. In
6046                 * particular, hdr->l1hdr.b_buf may be invalid by now due to
6047                 * ARC eviction.
6048                 */
6049                hdr->b_l2hdr.b_daddr = dev->l2ad_hand;

6051                if ((HDR_L2COMPRESS(hdr)) &&
6052                    hdr->b_l2hdr.b_asize >= buf_compress_minsz) {
6053                        if (l2arc_compress_buf(hdr)) {
6054                                /*
6055                                 * If compression succeeded, enable headroom
6056                                 * boost on the next scan cycle.
6057                                 */
6058                                *headroom_boost = B_TRUE;
6059                        }
6060                }

6062                /*
6063                 * Pick up the buffer data we had previously stashed away
6064                 * (and now potentially also compressed).
6065                 */
6066                buf_data = hdr->b_l1hdr.b_tmp_cdata;
6067                buf_sz = hdr->b_l2hdr.b_asize;

6069                /*
6070                 * We need to do this regardless if buf_sz is zero or
6071                 * not, otherwise, when this l2hdr is evicted we'll
6072                 * remove a reference that was never added.
6073                 */
6074                (void) refcount_add_many(&dev->l2ad_alloc, buf_sz, hdr);

6076                /* Compression may have squashed the buffer to zero length. */
6077                if (buf_sz != 0) {
6078                        uint64_t buf_p_sz;

6080                        wzio = zio_write_phys(pio, dev->l2ad_vdev,
6081                            dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
6082                            NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
6083                            ZIO_FLAG_CANFAIL, B_FALSE);

6085                        DTRACE_PROBE2(l2arc__write, vdev_t *, dev->l2ad_vdev,
6086                            zio_t *, wzio);
6087                        (void) zio_nowait(wzio);

6089                        write_asize += buf_sz;

6091                        /*
6092                         * Keep the clock hand suitably device-aligned.
6093                         */
6094                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
6095                        write_psize += buf_p_sz;
6096                        dev->l2ad_hand += buf_p_sz;
6097                }
6098        }

6100        mutex_exit(&dev->l2ad_mtx);

6102        ASSERT3U(write_asize, <=, target_sz);
6103        ARCSTAT_BUMP(arcstat_l2_writes_sent);
6104        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
6105        ARCSTAT_INCR(arcstat_l2_size, write_sz);
6106        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
6107        vdev_space_update(dev->l2ad_vdev, write_asize, 0, 0);
```



```
6109        /*
6110         * Bump device hand to the device start if it is approaching the end.
6111         * l2arc_evict() will already have evicted ahead for this case.
6112         */
6113        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
6114                dev->l2ad_hand = dev->l2ad_start;
6115                dev->l2ad_first = B_FALSE;
6116        }

6118        dev->l2ad_writing = B_TRUE;
6119        (void) zio_wait(pio);
6120        dev->l2ad_writing = B_FALSE;

6122        return (write_asize);
6123 }

6125 /*
6126  * Compresses an L2ARC buffer.
6127  * The data to be compressed must be prefilled in l1hdr.b_tmp_cdata and its
6128  * size in l2hdr->b_asize. This routine tries to compress the data and
6129  * depending on the compression result there are three possible outcomes:
6130  * *) The buffer was incompressible. The original l2hdr contents were left
6131  *    untouched and are ready for writing to an L2 device.
6132  * *) The buffer was all-zeros, so there is no need to write it to an L2
6133  *    device. To indicate this situation b_tmp_cdata is NULL'ed, b_asize is
6134  *    set to zero and b_compress is set to ZIO_COMPRESS_EMPTY.
6135  * *) Compression succeeded and b_tmp_cdata was replaced with a temporary
6136  *    data buffer which holds the compressed data to be written, and b_asize
6137  *    tells us how much data there is. b_compress is set to the appropriate
6138  *    compression algorithm. Once writing is done, invoke
6139  *    l2arc_release_cdata_buf on this l2hdr to free this temporary buffer.
6140  *
6141  * Returns B_TRUE if compression succeeded, or B_FALSE if it didn't (the
6142  * buffer was incompressible).
6143  */
6144 static boolean_t
6145 l2arc_compress_buf(arc_buf_hdr_t *hdr)
6146 {
6147        void *cdata;
6148        size_t csize, len, rounded;
6149        ASSERT(HDR_HAS_L2HDR(hdr));
6150        l2arc_buf_hdr_t *l2hdr = &hdr->b_l2hdr;

6152        ASSERT(HDR_HAS_L1HDR(hdr));
6153        ASSERT(l2hdr->b_compress == ZIO_COMPRESS_OFF);
6062        ASSERT(HDR_GET_COMPRESS(hdr) == ZIO_COMPRESS_OFF);
6154        ASSERT(hdr->b_l1hdr.b_tmp_cdata != NULL);

6156        len = l2hdr->b_asize;
6157        cdata = zio_data_buf_alloc(len);
6158        ASSERT3P(cdata, !=, NULL);
6159        csize = zio_compress_data(ZIO_COMPRESS_LZ4, hdr->b_l1hdr.b_tmp_cdata,
6160            cdata, l2hdr->b_asize);

6162        rounded = P2ROUNDUP(csize, (size_t)SPA_MINBLOCKSIZE);
6163        if (rounded > csize) {
6164                bzero((char *)cdata + csize, rounded - csize);
6165                csize = rounded;
6166        }

6168        if (csize == 0) {
6169                /* zero block, indicate that there's nothing to write */
6170                zio_data_buf_free(cdata, len);
6171                l2hdr->b_compress = ZIO_COMPRESS_EMPTY;
6080                HDR_SET_COMPRESS(hdr, ZIO_COMPRESS_EMPTY);
```

```
6172                 l2hdr->b_asize = 0;
6173                 hdr->b_l1hdr.b_tmp_cdata = NULL;
6174                 ARCSTAT_BUMP(arcstat_l2_compress_zeros);
6175                 return (B_TRUE);
6176         } else if (csize > 0 && csize < len) {
6177                 /*
6178                  * Compression succeeded, we'll keep the cdata around for
6179                  * writing and release it afterwards.
6180                  */
6181                 l2hdr->b_compress = ZIO_COMPRESS_LZ4;
6090                 HDR_SET_COMPRESS(hdr, ZIO_COMPRESS_LZ4);
6182                 l2hdr->b_asize = csize;
6183                 hdr->b_l1hdr.b_tmp_cdata = cdata;
6184                 ARCSTAT_BUMP(arcstat_l2_compress_successes);
6185                 return (B_TRUE);
6186         } else {
6187                 /*
6188                  * Compression failed, release the compressed buffer.
6189                  * l2hdr will be left unmodified.
6190                  */
6191                 zio_data_buf_free(cdata, len);
6192                 ARCSTAT_BUMP(arcstat_l2_compress_failures);
6193                 return (B_FALSE);
6194         }
6195 }
```

_____***unchanged_portion_omitted_***

```
6259 /*
6260  * Releases the temporary b_tmp_cdata buffer in an l2arc header structure.
6261  * This buffer serves as a temporary holder of compressed data while
6262  * the buffer entry is being written to an l2arc device. Once that is
6263  * done, we can dispose of it.
6264  */
6265 static void
6266 l2arc_release_cdata_buf(arc_buf_hdr_t *hdr)
6267 {
6268         ASSERT(HDR_HAS_L2HDR(hdr));
6269         enum zio_compress comp = hdr->b_l2hdr.b_compress;
6177         enum zio_compress comp = HDR_GET_COMPRESS(hdr);

6271         ASSERT(HDR_HAS_L1HDR(hdr));
6272         ASSERT(comp == ZIO_COMPRESS_OFF || L2ARC_IS_VALID_COMPRESS(comp));

6274         if (comp == ZIO_COMPRESS_OFF) {
6275                 /*
6276                  * In this case, b_tmp_cdata points to the same buffer
6277                  * as the arc_buf_t's b_data field. We don't want to
6278                  * free it, since the arc_buf_t will handle that.
6279                  */
6280                 hdr->b_l1hdr.b_tmp_cdata = NULL;
6281         } else if (comp == ZIO_COMPRESS_EMPTY) {
6282                 /*
6283                  * In this case, b_tmp_cdata was compressed to an empty
6284                  * buffer, thus there's nothing to free and b_tmp_cdata
6285                  * should have been set to NULL in l2arc_write_buffers().
6286                  */
6287                 ASSERT3P(hdr->b_l1hdr.b_tmp_cdata, ==, NULL);
6288         } else {
6289                 /*
6290                  * If the data was compressed, then we've allocated a
6291                  * temporary buffer for it, so now we need to release it.
6292                  */
6293                 ASSERT(hdr->b_l1hdr.b_tmp_cdata != NULL);
6294                 zio_data_buf_free(hdr->b_l1hdr.b_tmp_cdata,
6295                     hdr->b_size);
6296                 hdr->b_l1hdr.b_tmp_cdata = NULL;
```



```
6297         }

6299 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5672 Wed Sep  9 19:47:42 2015
new/usr/src/uts/common/fs/zfs/sys/arc.h
6214 zpools going south
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

#ifndef _SYS_ARC_H
#define _SYS_ARC_H

#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/zio.h>
#include <sys/dmu.h>
#include <sys/spa.h>

/*
 * Used by arc_flush() to inform arc_evict_state() that it should evict
 * all available buffers from the arc state being passed in.
 */
#define ARC_EVICT_ALL   -1ULL

typedef struct arc_buf_hdr arc_buf_hdr_t;
typedef struct arc_buf arc_buf_t;
typedef void arc_done_func_t(zio_t *zio, arc_buf_t *buf, void *private);
typedef int arc_evict_func_t(void *private);

/* generic arc_done_func_t's which you can use */
arc_done_func_t arc_bcopy_func;
arc_done_func_t arc_getbuf_func;

typedef enum arc_flags
{
        /*
         * Public flags that can be passed into the ARC by external consumers.
         */
        ARC_FLAG_NONE                   = 1 << 0,       /* No flags set */
        ARC_FLAG_WAIT                   = 1 << 1,       /* perform sync I/O */
```



```
        ARC_FLAG_NOWAIT                 = 1 << 2,       /* perform async I/O */
        ARC_FLAG_PREFETCH               = 1 << 3,       /* I/O is a prefetch */
        ARC_FLAG_CACHED                 = 1 << 4,       /* I/O was in cache */
        ARC_FLAG_L2CACHE                = 1 << 5,       /* cache in L2ARC */
        ARC_FLAG_L2COMPRESS             = 1 << 6,       /* compress in L2ARC */

        /*
         * Private ARC flags.  These flags are private ARC only flags that
         * will show up in b_flags in the arc_hdr_buf_t. These flags should
         * only be set by ARC code.
         */
        ARC_FLAG_IN_HASH_TABLE          = 1 << 7,       /* buffer is hashed */
        ARC_FLAG_IO_IN_PROGRESS         = 1 << 8,       /* I/O in progress */
        ARC_FLAG_IO_ERROR               = 1 << 9,       /* I/O failed for buf */
        ARC_FLAG_FREED_IN_READ          = 1 << 10,      /* freed during read */
        ARC_FLAG_BUF_AVAILABLE          = 1 << 11,      /* block not in use */
        ARC_FLAG_INDIRECT               = 1 << 12,      /* indirect block */
        ARC_FLAG_L2_WRITING             = 1 << 13,      /* write in progress */
        ARC_FLAG_L2_EVICTED             = 1 << 14,      /* evicted during I/O */
        ARC_FLAG_L2_WRITE_HEAD          = 1 << 15,      /* head of write list */
        /* indicates that the buffer contains metadata (otherwise, data) */
        ARC_FLAG_BUFC_METADATA          = 1 << 16,

        /* Flags specifying whether optional hdr struct fields are defined */
        ARC_FLAG_HAS_L1HDR              = 1 << 17,
        ARC_FLAG_HAS_L2HDR              = 1 << 18,

        /*
         * The arc buffer's compression mode is stored in the top 7 bits of the
         * flags field, so these dummy flags are included so that MDB can
         * interpret the enum properly.
         */
        ARC_FLAG_COMPRESS_0             = 1 << 24,
        ARC_FLAG_COMPRESS_1             = 1 << 25,
        ARC_FLAG_COMPRESS_2             = 1 << 26,
        ARC_FLAG_COMPRESS_3             = 1 << 27,
        ARC_FLAG_COMPRESS_4             = 1 << 28,
        ARC_FLAG_COMPRESS_5             = 1 << 29,
        ARC_FLAG_COMPRESS_6             = 1 << 30

} arc_flags_t;
```

_____unchanged_portion_omitted_