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

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

typedef struct arc_stats {
        kstat_named_t arcstat_hits;
        kstat_named_t arcstat_misses;
        kstat_named_t arcstat_demand_data_hits;
        kstat_named_t arcstat_demand_data_misses;
        kstat_named_t arcstat_demand_metadata_hits;
        kstat_named_t arcstat_demand_metadata_misses;
        kstat_named_t arcstat_prefetch_data_hits;
        kstat_named_t arcstat_prefetch_data_misses;
        kstat_named_t arcstat_prefetch_metadata_hits;
        kstat_named_t arcstat_prefetch_metadata_misses;
        kstat_named_t arcstat_mru_hits;
        kstat_named_t arcstat_mru_ghost_hits;
        kstat_named_t arcstat_mfu_hits;
        kstat_named_t arcstat_mfu_ghost_hits;
        kstat_named_t arcstat_deleted;
        kstat_named_t arcstat_recycle_miss;
        /*
         * Number of buffers that could not be evicted because the hash lock
         * was held by another thread.  The lock may not necessarily be held
         * by something using the same buffer, since hash locks are shared
         * by multiple buffers.
         */
        kstat_named_t arcstat_mutex_miss;
        /*
         * Number of buffers skipped because they have I/O in progress, are
         * indrect prefetch buffers that have not lived long enough, or are
         * not from the spa we’re trying to evict from.
         */
        kstat_named_t arcstat_evict_skip;
        kstat_named_t arcstat_evict_l2_cached;
        kstat_named_t arcstat_evict_l2_eligible;
        kstat_named_t arcstat_evict_l2_ineligible;
        kstat_named_t arcstat_hash_elements;
        kstat_named_t arcstat_hash_elements_max;
        kstat_named_t arcstat_hash_collisions;
        kstat_named_t arcstat_hash_chains;
        kstat_named_t arcstat_hash_chain_max;
        kstat_named_t arcstat_p;
        kstat_named_t arcstat_c;
        kstat_named_t arcstat_c_min;
        kstat_named_t arcstat_c_max;
        kstat_named_t arcstat_size;
        kstat_named_t arcstat_hdr_size;
        kstat_named_t arcstat_data_size;
        kstat_named_t arcstat_other_size;
        kstat_named_t arcstat_l2_hits;
        kstat_named_t arcstat_l2_misses;
```

```
        kstat_named_t arcstat_l2_feeds;
        kstat_named_t arcstat_l2_rw_clash;
        kstat_named_t arcstat_l2_read_bytes;
        kstat_named_t arcstat_l2_write_bytes;
        kstat_named_t arcstat_l2_writes_sent;
        kstat_named_t arcstat_l2_writes_done;
        kstat_named_t arcstat_l2_writes_error;
        kstat_named_t arcstat_l2_writes_hdr_miss;
        kstat_named_t arcstat_l2_evict_lock_retry;
        kstat_named_t arcstat_l2_evict_reading;
        kstat_named_t arcstat_l2_free_on_write;
        kstat_named_t arcstat_l2_cdata_free_on_write;
#endif /* ! codereview */
        kstat_named_t arcstat_l2_abort_lowmem;
        kstat_named_t arcstat_l2_cksum_bad;
        kstat_named_t arcstat_l2_io_error;
        kstat_named_t arcstat_l2_size;
        kstat_named_t arcstat_l2_asize;
        kstat_named_t arcstat_l2_hdr_size;
        kstat_named_t arcstat_l2_compress_successes;
        kstat_named_t arcstat_l2_compress_zeros;
        kstat_named_t arcstat_l2_compress_failures;
        kstat_named_t arcstat_memory_throttle_count;
        kstat_named_t arcstat_duplicate_buffers;
        kstat_named_t arcstat_duplicate_buffers_size;
        kstat_named_t arcstat_duplicate_reads;
        kstat_named_t arcstat_meta_used;
        kstat_named_t arcstat_meta_limit;
        kstat_named_t arcstat_meta_max;
} arc_stats_t;

static arc_stats_t arc_stats = {
        { "hits",                       KSTAT_DATA_UINT64 },
        { "misses",                     KSTAT_DATA_UINT64 },
        { "demand_data_hits",           KSTAT_DATA_UINT64 },
        { "demand_data_misses",         KSTAT_DATA_UINT64 },
        { "demand_metadata_hits",       KSTAT_DATA_UINT64 },
        { "demand_metadata_misses",     KSTAT_DATA_UINT64 },
        { "prefetch_data_hits",         KSTAT_DATA_UINT64 },
        { "prefetch_data_misses",       KSTAT_DATA_UINT64 },
        { "prefetch_metadata_hits",     KSTAT_DATA_UINT64 },
        { "prefetch_metadata_misses",   KSTAT_DATA_UINT64 },
        { "mru_hits",                   KSTAT_DATA_UINT64 },
        { "mru_ghost_hits",             KSTAT_DATA_UINT64 },
        { "mfu_hits",                   KSTAT_DATA_UINT64 },
        { "mfu_ghost_hits",             KSTAT_DATA_UINT64 },
        { "deleted",                    KSTAT_DATA_UINT64 },
        { "recycle_miss",               KSTAT_DATA_UINT64 },
        { "mutex_miss",                 KSTAT_DATA_UINT64 },
        { "evict_skip",                 KSTAT_DATA_UINT64 },
        { "evict_l2_cached",            KSTAT_DATA_UINT64 },
        { "evict_l2_eligible",          KSTAT_DATA_UINT64 },
        { "evict_l2_ineligible",        KSTAT_DATA_UINT64 },
        { "hash_elements",              KSTAT_DATA_UINT64 },
        { "hash_elements_max",          KSTAT_DATA_UINT64 },
        { "hash_collisions",            KSTAT_DATA_UINT64 },
        { "hash_chains",                KSTAT_DATA_UINT64 },
        { "hash_chain_max",             KSTAT_DATA_UINT64 },
        { "p",                          KSTAT_DATA_UINT64 },
        { "c",                          KSTAT_DATA_UINT64 },
        { "c_min",                      KSTAT_DATA_UINT64 },
        { "c_max",                      KSTAT_DATA_UINT64 },
        { "size",                       KSTAT_DATA_UINT64 },
        { "hdr_size",                   KSTAT_DATA_UINT64 },
        { "data_size",                  KSTAT_DATA_UINT64 },
        { "other_size",                 KSTAT_DATA_UINT64 },
        { "l2_hits",                    KSTAT_DATA_UINT64 },
        { "l2_misses",                  KSTAT_DATA_UINT64 },
        { "l2_feeds",                   KSTAT_DATA_UINT64 },
        { "l2_rw_clash",                KSTAT_DATA_UINT64 },
        { "l2_read_bytes",              KSTAT_DATA_UINT64 },
        { "l2_write_bytes",             KSTAT_DATA_UINT64 },
        { "l2_writes_sent",             KSTAT_DATA_UINT64 },
        { "l2_writes_done",             KSTAT_DATA_UINT64 },
        { "l2_writes_error",            KSTAT_DATA_UINT64 },
        { "l2_writes_hdr_miss",         KSTAT_DATA_UINT64 },
        { "l2_evict_lock_retry",        KSTAT_DATA_UINT64 },
        { "l2_evict_reading",           KSTAT_DATA_UINT64 },
        { "l2_free_on_write",           KSTAT_DATA_UINT64 },
        { "l2_cdata_free_on_write",     KSTAT_DATA_UINT64 },
#endif /* ! codereview */
        { "l2_abort_lowmem",            KSTAT_DATA_UINT64 },
        { "l2_cksum_bad",               KSTAT_DATA_UINT64 },
        { "l2_io_error",                KSTAT_DATA_UINT64 },
        { "l2_size",                    KSTAT_DATA_UINT64 },
        { "l2_asize",                   KSTAT_DATA_UINT64 },
        { "l2_hdr_size",                KSTAT_DATA_UINT64 },
        { "l2_compress_successes",      KSTAT_DATA_UINT64 },
        { "l2_compress_zeros",          KSTAT_DATA_UINT64 },
        { "l2_compress_failures",       KSTAT_DATA_UINT64 },
        { "memory_throttle_count",      KSTAT_DATA_UINT64 },
        { "duplicate_buffers",          KSTAT_DATA_UINT64 },
        { "duplicate_buffers_size",     KSTAT_DATA_UINT64 },
        { "duplicate_reads",            KSTAT_DATA_UINT64 },
        { "arc_meta_used",              KSTAT_DATA_UINT64 },
        { "arc_meta_limit",             KSTAT_DATA_UINT64 },
        { "arc_meta_max",               KSTAT_DATA_UINT64 }
};

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

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

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

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

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

/*
 * We define a macro to allow ARC hits/misses to be easily broken down by
 * two separate conditions, giving a total of four different subtypes for
 * each of hits and misses (so eight statistics total).
 */
#define ARCSTAT_CONDSTAT(cond1, stat1, notstat1, cond2, stat2, notstat2, stat) \
        if (cond1) {                                                    \
                if (cond2) {                                            \
                        ARCSTAT_BUMP(arcstat_##stat1##_##stat2##_##stat); \
                } else {                                                \
                        ARCSTAT_BUMP(arcstat_##stat1##_##notstat2##_##stat); \
                }                                                       \
        } else {                                                        \
                if (cond2) {                                            \
                        ARCSTAT_BUMP(arcstat_##notstat1##_##stat2##_##stat); \
```

```
                } else {                                                \
                        ARCSTAT_BUMP(arcstat_##notstat1##_##notstat2##_##stat);\
                }                                                       \
        }

kstat_t                 *arc_ksp;
static arc_state_t      *arc_anon;
static arc_state_t      *arc_mru;
static arc_state_t      *arc_mru_ghost;
static arc_state_t      *arc_mfu;
static arc_state_t      *arc_mfu_ghost;
static arc_state_t      *arc_l2c_only;

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

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

static int              arc_no_grow;    /* Don’t try to grow cache size */
static uint64_t         arc_tempreserve;
static uint64_t         arc_loaned_bytes;

typedef struct l2arc_buf_hdr l2arc_buf_hdr_t;

typedef struct arc_callback arc_callback_t;

struct arc_callback {
        void                    *acb_private;
        arc_done_func_t         *acb_done;
        arc_buf_t               *acb_buf;
        zio_t                   *acb_zio_dummy;
        arc_callback_t          *acb_next;
};

typedef struct arc_write_callback arc_write_callback_t;

struct arc_write_callback {
        void            *awcb_private;
        arc_done_func_t *awcb_ready;
        arc_done_func_t *awcb_physdone;
        arc_done_func_t *awcb_done;
        arc_buf_t       *awcb_buf;
};

struct arc_buf_hdr {
        /* protected by hash lock */
        dva_t                   b_dva;
        uint64_t                b_birth;
        uint64_t                b_cksum0;

        kmutex_t                b_freeze_lock;
        zio_cksum_t             *b_freeze_cksum;
        void                    *b_thawed;

        arc_buf_hdr_t           *b_hash_next;
        arc_buf_t               *b_buf;
        uint32_t                b_flags;
        uint32_t                b_datacnt;

        arc_callback_t          *b_acb;
        kcondvar_t              b_cv;

        /* immutable */
        arc_buf_contents_t      b_type;
        uint64_t                b_size;
        uint64_t                b_spa;

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

        /* updated atomically */
        clock_t                 b_arc_access;

        /* self protecting */
        refcount_t              b_refcnt;

        l2arc_buf_hdr_t         *b_l2hdr;
        list_node_t             b_l2node;
};

static arc_buf_t *arc_eviction_list;
static kmutex_t arc_eviction_mtx;
static arc_buf_hdr_t arc_eviction_hdr;
static void arc_get_data_buf(arc_buf_t *buf);
static void arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock);
static int arc_evict_needed(arc_buf_contents_t type);
static void arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes);
static void arc_buf_watch(arc_buf_t *buf);

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

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

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

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

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

```
#define HDR_IO_ERROR(hdr)       ((hdr)->b_flags & ARC_IO_ERROR)
#define HDR_PREFETCH(hdr)       ((hdr)->b_flags & ARC_PREFETCH)
#define HDR_FREED_IN_READ(hdr)  ((hdr)->b_flags & ARC_FREED_IN_READ)
#define HDR_BUF_AVAILABLE(hdr)  ((hdr)->b_flags & ARC_BUF_AVAILABLE)
#define HDR_FREE_IN_PROGRESS(hdr)       ((hdr)->b_flags & ARC_FREE_IN_PROGRESS)
#define HDR_L2CACHE(hdr)        ((hdr)->b_flags & ARC_L2CACHE)
#define HDR_L2_READING(hdr)     ((hdr)->b_flags & ARC_IO_IN_PROGRESS && \
                                    (hdr)->b_l2hdr != NULL)
#define HDR_L2_WRITING(hdr)     ((hdr)->b_flags & ARC_L2_WRITING)
#define HDR_L2_EVICTED(hdr)     ((hdr)->b_flags & ARC_L2_EVICTED)
#define HDR_L2_WRITE_HEAD(hdr)  ((hdr)->b_flags & ARC_L2_WRITE_HEAD)

/*
 * Other sizes
 */

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

/*
 * Hash table routines
 */

#define HT_LOCK_PAD     64

struct ht_lock {
        kmutex_t        ht_lock;
#ifdef _KERNEL
        unsigned char   pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
#endif
};

#define BUF_LOCKS 256
typedef struct buf_hash_table {
        uint64_t ht_mask;
        arc_buf_hdr_t **ht_table;
        struct ht_lock ht_locks[BUF_LOCKS];
} buf_hash_table_t;

static buf_hash_table_t buf_hash_table;

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

uint64_t zfs_crc64_table[256];

/*
 * Level 2 ARC
 */

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);

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

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

        return (crc);
}

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

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

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

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

        mutex_enter(hash_lock);
        for (buf = buf_hash_table.ht_table[idx]; buf != NULL;
            buf = buf->b_hash_next) {
                if (BUF_EQUAL(spa, dva, birth, buf)) {
                        *lockp = hash_lock;
                        return (buf);
                }
        }
        mutex_exit(hash_lock);
        *lockp = NULL;
        return (NULL);
}

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

        ASSERT(!DVA_IS_EMPTY(&buf->b_dva));
        ASSERT(buf->b_birth != 0);
        ASSERT(!HDR_IN_HASH_TABLE(buf));
        *lockp = hash_lock;
        mutex_enter(hash_lock);
        for (fbuf = buf_hash_table.ht_table[idx], i = 0; fbuf != NULL;
            fbuf = fbuf->b_hash_next, i++) {
                if (BUF_EQUAL(buf->b_spa, &buf->b_dva, buf->b_birth, fbuf))
                        return (fbuf);
        }

        buf->b_hash_next = buf_hash_table.ht_table[idx];
        buf_hash_table.ht_table[idx] = buf;
        buf->b_flags |= ARC_IN_HASH_TABLE;

        /* collect some hash table performance data */
        if (i > 0) {
                ARCSTAT_BUMP(arcstat_hash_collisions);
                if (i == 1)
                        ARCSTAT_BUMP(arcstat_hash_chains);

                ARCSTAT_MAX(arcstat_hash_chain_max, i);
        }

        ARCSTAT_BUMP(arcstat_hash_elements);
        ARCSTAT_MAXSTAT(arcstat_hash_elements);

        return (NULL);
}

static void
buf_hash_remove(arc_buf_hdr_t *buf)
{
        arc_buf_hdr_t *fbuf, **bufp;
        uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);

        ASSERT(MUTEX_HELD(BUF_HASH_LOCK(idx)));
        ASSERT(HDR_IN_HASH_TABLE(buf));

        bufp = &buf_hash_table.ht_table[idx];
        while ((fbuf = *bufp) != buf) {
```

```
                ASSERT(fbuf != NULL);
                bufp = &fbuf->b_hash_next;
        }
        *bufp = buf->b_hash_next;
        buf->b_hash_next = NULL;
        buf->b_flags &= ~ARC_IN_HASH_TABLE;

        /* collect some hash table performance data */
        ARCSTAT_BUMPDOWN(arcstat_hash_elements);

        if (buf_hash_table.ht_table[idx] &&
            buf_hash_table.ht_table[idx]->b_hash_next == NULL)
                ARCSTAT_BUMPDOWN(arcstat_hash_chains);
}

/*
 * Global data structures and functions for the buf kmem cache.
 */
static kmem_cache_t *hdr_cache;
static kmem_cache_t *buf_cache;

static void
buf_fini(void)
{
        int i;

        kmem_free(buf_hash_table.ht_table,
            (buf_hash_table.ht_mask + 1) * sizeof (void *));
        for (i = 0; i < BUF_LOCKS; i++)
                mutex_destroy(&buf_hash_table.ht_locks[i].ht_lock);
        kmem_cache_destroy(hdr_cache);
        kmem_cache_destroy(buf_cache);
}

/*
 * Constructor callback - called when the cache is empty
 * and a new buf is requested.
 */
/* ARGSUSED */
static int
hdr_cons(void *vbuf, void *unused, int kmflag)
{
        arc_buf_hdr_t *buf = vbuf;

        bzero(buf, sizeof (arc_buf_hdr_t));
        refcount_create(&buf->b_refcnt);
        cv_init(&buf->b_cv, NULL, CV_DEFAULT, NULL);
        mutex_init(&buf->b_freeze_lock, NULL, MUTEX_DEFAULT, NULL);
        arc_space_consume(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);

        return (0);
}

/* ARGSUSED */
static int
buf_cons(void *vbuf, void *unused, int kmflag)
{
        arc_buf_t *buf = vbuf;

        bzero(buf, sizeof (arc_buf_t));
        mutex_init(&buf->b_evict_lock, NULL, MUTEX_DEFAULT, NULL);
        arc_space_consume(sizeof (arc_buf_t), ARC_SPACE_HDRS);

        return (0);
}
```

```
/*
 * Destructor callback - called when a cached buf is
 * no longer required.
 */
/* ARGSUSED */
static void
hdr_dest(void *vbuf, void *unused)
{
        arc_buf_hdr_t *buf = vbuf;

        ASSERT(BUF_EMPTY(buf));
        refcount_destroy(&buf->b_refcnt);
        cv_destroy(&buf->b_cv);
        mutex_destroy(&buf->b_freeze_lock);
        arc_space_return(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
}

/* ARGSUSED */
static void
buf_dest(void *vbuf, void *unused)
{
        arc_buf_t *buf = vbuf;

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

/*
 * Reclaim callback -- invoked when memory is low.
 */
/* ARGSUSED */
static void
hdr_recl(void *unused)
{
        dprintf("hdr_recl called\n");
        /*
         * umem calls the reclaim func when we destroy the buf cache,
         * which is after we do arc_fini().
         */
        if (!arc_dead)
                cv_signal(&arc_reclaim_thr_cv);
}

static void
buf_init(void)
{
        uint64_t *ct;
        uint64_t hsize = 1ULL << 12;
        int i, j;

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

```
        }

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

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

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

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

static void
arc_cksum_verify(arc_buf_t *buf)
{
        zio_cksum_t zc;

        if (!(zfs_flags & ZFS_DEBUG_MODIFY))
                return;

        mutex_enter(&buf->b_hdr->b_freeze_lock);
        if (buf->b_hdr->b_freeze_cksum == NULL ||
            (buf->b_hdr->b_flags & ARC_IO_ERROR)) {
                mutex_exit(&buf->b_hdr->b_freeze_lock);
                return;
        }
        fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
        if (!ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc))
                panic("buffer modified while frozen!");
        mutex_exit(&buf->b_hdr->b_freeze_lock);
}

static int
arc_cksum_equal(arc_buf_t *buf)
{
        zio_cksum_t zc;
        int equal;

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

        return (equal);
}

static void
arc_cksum_compute(arc_buf_t *buf, boolean_t force)
{
        if (!force && !(zfs_flags & ZFS_DEBUG_MODIFY))
                return;

        mutex_enter(&buf->b_hdr->b_freeze_lock);
        if (buf->b_hdr->b_freeze_cksum != NULL) {
                mutex_exit(&buf->b_hdr->b_freeze_lock);
                return;
        }
        buf->b_hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
        fletcher_2_native(buf->b_data, buf->b_hdr->b_size,
            buf->b_hdr->b_freeze_cksum);
        mutex_exit(&buf->b_hdr->b_freeze_lock);
        arc_buf_watch(buf);
}

#ifndef _KERNEL
typedef struct procctl {
        long cmd;
        prwatch_t prwatch;
} procctl_t;
#endif

/* ARGSUSED */
static void
arc_buf_unwatch(arc_buf_t *buf)
{
#ifndef _KERNEL
        if (arc_watch) {
                int result;
                procctl_t ctl;
                ctl.cmd = PCWATCH;
                ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
                ctl.prwatch.pr_size = 0;
                ctl.prwatch.pr_wflags = 0;
                result = write(arc_procfd, &ctl, sizeof (ctl));
                ASSERT3U(result, ==, sizeof (ctl));
        }
#endif
}

/* ARGSUSED */
static void
arc_buf_watch(arc_buf_t *buf)
{
#ifndef _KERNEL
        if (arc_watch) {
                int result;
                procctl_t ctl;
                ctl.cmd = PCWATCH;
                ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
                ctl.prwatch.pr_size = buf->b_hdr->b_size;
                ctl.prwatch.pr_wflags = WA_WRITE;
                result = write(arc_procfd, &ctl, sizeof (ctl));
                ASSERT3U(result, ==, sizeof (ctl));
        }
#endif
}

void
arc_buf_thaw(arc_buf_t *buf)
{
        if (zfs_flags & ZFS_DEBUG_MODIFY) {
                if (buf->b_hdr->b_state != arc_anon)
                        panic("modifying non-anon buffer!");
                if (buf->b_hdr->b_flags & ARC_IO_IN_PROGRESS)
                        panic("modifying buffer while i/o in progress!");
                arc_cksum_verify(buf);
        }

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

        if (zfs_flags & ZFS_DEBUG_MODIFY) {
```

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

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

        arc_buf_unwatch(buf);
}

void
arc_buf_freeze(arc_buf_t *buf)
{
        kmutex_t *hash_lock;

        if (!(zfs_flags & ZFS_DEBUG_MODIFY))
                return;

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

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

}

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

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

                ASSERT(!MUTEX_HELD(&ab->b_state->arcs_mtx));
                mutex_enter(&ab->b_state->arcs_mtx);
                ASSERT(list_link_active(&ab->b_arc_node));
                list_remove(list, ab);
                if (GHOST_STATE(ab->b_state)) {
                        ASSERT0(ab->b_datacnt);
                        ASSERT3P(ab->b_buf, ==, NULL);
                        delta = ab->b_size;
                }
                ASSERT(delta > 0);
                ASSERT3U(*size, >=, delta);
                atomic_add_64(size, -delta);
                mutex_exit(&ab->b_state->arcs_mtx);
                /* remove the prefetch flag if we get a reference */
                if (ab->b_flags & ARC_PREFETCH)
                        ab->b_flags &= ~ARC_PREFETCH;
        }
}

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

        ASSERT(state == arc_anon || MUTEX_HELD(hash_lock));
        ASSERT(!GHOST_STATE(state));
        if (((cnt = refcount_remove(&ab->b_refcnt, tag)) == 0) &&
            (state != arc_anon)) {
                uint64_t *size = &state->arcs_lsize[ab->b_type];

                ASSERT(!MUTEX_HELD(&state->arcs_mtx));
                mutex_enter(&state->arcs_mtx);
                ASSERT(!list_link_active(&ab->b_arc_node));
                list_insert_head(&state->arcs_list[ab->b_type], ab);
                ASSERT(ab->b_datacnt > 0);
                atomic_add_64(size, ab->b_size * ab->b_datacnt);
                mutex_exit(&state->arcs_mtx);
        }
        return (cnt);
}

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

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

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

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

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

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

                        /*
                         * If prefetching out of the ghost cache,
                         * we will have a non-zero datacnt.
                         */
                        if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
                                /* ghost elements have a ghost size */
                                ASSERT(ab->b_buf == NULL);
                                from_delta = ab->b_size;
                        }
                        ASSERT3U(*size, >=, from_delta);
                        atomic_add_64(size, -from_delta);

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

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

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

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

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

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

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

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

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

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

        switch (type) {
        case ARC_SPACE_DATA:
                ARCSTAT_INCR(arcstat_data_size, space);
                break;
        case ARC_SPACE_OTHER:
                ARCSTAT_INCR(arcstat_other_size, space);
                break;
        case ARC_SPACE_HDRS:
                ARCSTAT_INCR(arcstat_hdr_size, space);
                break;
        case ARC_SPACE_L2HDRS:
                ARCSTAT_INCR(arcstat_l2_hdr_size, space);
                break;
        }

        ARCSTAT_INCR(arcstat_meta_used, space);
        atomic_add_64(&arc_size, space);
}

void
```

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

        switch (type) {
        case ARC_SPACE_DATA:
                ARCSTAT_INCR(arcstat_data_size, -space);
                break;
        case ARC_SPACE_OTHER:
                ARCSTAT_INCR(arcstat_other_size, -space);
                break;
        case ARC_SPACE_HDRS:
                ARCSTAT_INCR(arcstat_hdr_size, -space);
                break;
        case ARC_SPACE_L2HDRS:
                ARCSTAT_INCR(arcstat_l2_hdr_size, -space);
                break;
        }

        ASSERT(arc_meta_used >= space);
        if (arc_meta_max < arc_meta_used)
                arc_meta_max = arc_meta_used;
        ARCSTAT_INCR(arcstat_meta_used, -space);
        ASSERT(arc_size >= space);
        atomic_add_64(&arc_size, -space);
}

void *
arc_data_buf_alloc(uint64_t size)
{
        if (arc_evict_needed(ARC_BUFC_DATA))
                cv_signal(&arc_reclaim_thr_cv);
        atomic_add_64(&arc_size, size);
        return (zio_data_buf_alloc(size));
}

void
arc_data_buf_free(void *buf, uint64_t size)
{
        zio_data_buf_free(buf, size);
        ASSERT(arc_size >= size);
        atomic_add_64(&arc_size, -size);
}

arc_buf_t *
arc_buf_alloc(spa_t *spa, int size, void *tag, arc_buf_contents_t type)
{
        arc_buf_hdr_t *hdr;
        arc_buf_t *buf;

        ASSERT3U(size, >, 0);
        hdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
        ASSERT(BUF_EMPTY(hdr));
        hdr->b_size = size;
        hdr->b_type = type;
        hdr->b_spa = spa_load_guid(spa);
        hdr->b_state = arc_anon;
        hdr->b_arc_access = 0;
        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
        buf->b_hdr = hdr;
        buf->b_data = NULL;
        buf->b_efunc = NULL;
        buf->b_private = NULL;
        buf->b_next = NULL;
        hdr->b_buf = buf;
        arc_get_data_buf(buf);
```

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

        return (buf);
}

static char *arc_onloan_tag = "onloan";

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

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

        atomic_add_64(&arc_loaned_bytes, size);
        return (buf);
}

/*
 * Return a loaned arc buffer to the arc.
 */
void
arc_return_buf(arc_buf_t *buf, void *tag)
{
        arc_buf_hdr_t *hdr = buf->b_hdr;

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

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

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

        ASSERT(buf->b_data != NULL);
        hdr = buf->b_hdr;
        (void) refcount_add(&hdr->b_refcnt, arc_onloan_tag);
        (void) refcount_remove(&hdr->b_refcnt, tag);
        buf->b_efunc = NULL;
        buf->b_private = NULL;

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

static arc_buf_t *
arc_buf_clone(arc_buf_t *from)
{
        arc_buf_t *buf;
        arc_buf_hdr_t *hdr = from->b_hdr;
        uint64_t size = hdr->b_size;

        ASSERT(hdr->b_state != arc_anon);
        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
        buf->b_hdr = hdr;
        buf->b_data = NULL;
        buf->b_efunc = NULL;
        buf->b_private = NULL;
        buf->b_next = hdr->b_buf;
        hdr->b_buf = buf;
        arc_get_data_buf(buf);
        bcopy(from->b_data, buf->b_data, size);

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

void
arc_buf_add_ref(arc_buf_t *buf, void* tag)
{
        arc_buf_hdr_t *hdr;
        kmutex_t *hash_lock;

        /*
         * Check to see if this buffer is evicted.  Callers
         * must verify b_data != NULL to know if the add_ref
         * was successful.
         */
        mutex_enter(&buf->b_evict_lock);
        if (buf->b_data == NULL) {
                mutex_exit(&buf->b_evict_lock);
                return;
        }
        hash_lock = HDR_LOCK(buf->b_hdr);
        mutex_enter(hash_lock);
        hdr = buf->b_hdr;
        ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
        mutex_exit(&buf->b_evict_lock);

        ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);
        add_reference(hdr, hash_lock, tag);
        DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
        arc_access(hdr, hash_lock);
        mutex_exit(hash_lock);
        ARCSTAT_BUMP(arcstat_hits);
        ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
            demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
            data, metadata, hits);
}

static void
arc_buf_free_on_write(void *data, size_t size,
    void (*free_func)(void *, size_t))
{
        l2arc_data_free_t *df;

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

```
        df->l2df_size = size;
        df->l2df_func = free_func;
        mutex_enter(&l2arc_free_on_write_mtx);
        list_insert_head(l2arc_free_on_write, df);
        mutex_exit(&l2arc_free_on_write_mtx);
}

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

        if (HDR_L2_WRITING(hdr)) {
                arc_buf_free_on_write(buf->b_data, hdr->b_size, free_func);
                l2arc_data_free_t *df;
                df = kmem_alloc(sizeof (l2arc_data_free_t), KM_SLEEP);
                df->l2df_data = buf->b_data;
                df->l2df_size = hdr->b_size;
                df->l2df_func = free_func;
                mutex_enter(&l2arc_free_on_write_mtx);
                list_insert_head(l2arc_free_on_write, df);
                mutex_exit(&l2arc_free_on_write_mtx);
                ARCSTAT_BUMP(arcstat_l2_free_on_write);
        } else {
                free_func(buf->b_data, hdr->b_size);
        }
}

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

        ASSERT(MUTEX_HELD(&l2arc_buflist_mtx));

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

        ASSERT(HDR_L2_WRITING(hdr));
        arc_buf_free_on_write(l2hdr->b_tmp_cdata, hdr->b_size,
            zio_data_buf_free);
        ARCSTAT_BUMP(arcstat_l2_cdata_free_on_write);
        l2hdr->b_tmp_cdata = NULL;
}

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

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

                if (!recycle) {
                        if (type == ARC_BUFC_METADATA) {
                                arc_buf_data_free(buf, zio_buf_free);
                                arc_space_return(size, ARC_SPACE_DATA);
                        } else {
                                ASSERT(type == ARC_BUFC_DATA);
                                arc_buf_data_free(buf, zio_data_buf_free);
                                ARCSTAT_INCR(arcstat_data_size, -size);
                                atomic_add_64(&arc_size, -size);
                        }
                }
                if (list_link_active(&buf->b_hdr->b_arc_node)) {
                        uint64_t *cnt = &state->arcs_lsize[type];

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

                        ASSERT3U(*cnt, >=, size);
                        atomic_add_64(cnt, -size);
                }
                ASSERT3U(state->arcs_size, >=, size);
                atomic_add_64(&state->arcs_size, -size);
                buf->b_data = NULL;

                /*
                 * If we’re destroying a duplicate buffer make sure
                 * that the appropriate statistics are updated.
                 */
                if (buf->b_hdr->b_datacnt > 1 &&
                    buf->b_hdr->b_type == ARC_BUFC_DATA) {
                        ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
                        ARCSTAT_INCR(arcstat_duplicate_buffers_size, -size);
                }
                ASSERT(buf->b_hdr->b_datacnt > 0);
                buf->b_hdr->b_datacnt -= 1;
        }

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

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

        ASSERT(buf->b_efunc == NULL);

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

static void
arc_hdr_destroy(arc_buf_hdr_t *hdr)
{
        ASSERT(refcount_is_zero(&hdr->b_refcnt));
        ASSERT3P(hdr->b_state, ==, arc_anon);
        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
        l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;

        if (l2hdr != NULL) {
```

```
                boolean_t buflist_held = MUTEX_HELD(&l2arc_buflist_mtx);
                /*
                 * To prevent arc_free() and l2arc_evict() from
                 * attempting to free the same buffer at the same time,
                 * a FREE_IN_PROGRESS flag is given to arc_free() to
                 * give it priority.  l2arc_evict() can’t destroy this
                 * header while we are waiting on l2arc_buflist_mtx.
                 *
                 * The hdr may be removed from l2ad_buflist before we
                 * grab l2arc_buflist_mtx, so b_l2hdr is rechecked.
                 */
                if (!buflist_held) {
                        mutex_enter(&l2arc_buflist_mtx);
                        l2hdr = hdr->b_l2hdr;
                }

                if (l2hdr != NULL) {
                        list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
                        arc_buf_l2_cdata_free(hdr);
#endif /* ! codereview */
                        ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);
                        ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
                        vdev_space_update(l2hdr->b_dev->l2ad_vdev,
                            -l2hdr->b_asize, 0, 0);
                        kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
                        if (hdr->b_state == arc_l2c_only)
                                l2arc_hdr_stat_remove();
                        hdr->b_l2hdr = NULL;
                }

                if (!buflist_held)
                        mutex_exit(&l2arc_buflist_mtx);
        }

        if (!BUF_EMPTY(hdr)) {
                ASSERT(!HDR_IN_HASH_TABLE(hdr));
                buf_discard_identity(hdr);
        }
        while (hdr->b_buf) {
                arc_buf_t *buf = hdr->b_buf;

                if (buf->b_efunc) {
                        mutex_enter(&arc_eviction_mtx);
                        mutex_enter(&buf->b_evict_lock);
                        ASSERT(buf->b_hdr != NULL);
                        arc_buf_destroy(hdr->b_buf, FALSE, FALSE);
                        hdr->b_buf = buf->b_next;
                        buf->b_hdr = &arc_eviction_hdr;
                        buf->b_next = arc_eviction_list;
                        arc_eviction_list = buf;
                        mutex_exit(&buf->b_evict_lock);
                        mutex_exit(&arc_eviction_mtx);
                } else {
                        arc_buf_destroy(hdr->b_buf, FALSE, TRUE);
                }
        }
        if (hdr->b_freeze_cksum != NULL) {
                kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
                hdr->b_freeze_cksum = NULL;
        }
        if (hdr->b_thawed) {
                kmem_free(hdr->b_thawed, 1);
                hdr->b_thawed = NULL;
        }

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

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

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

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

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

                (void) remove_reference(hdr, hash_lock, tag);
                if (hdr->b_datacnt > 1) {
                        arc_buf_destroy(buf, FALSE, TRUE);
                } else {
                        ASSERT(buf == hdr->b_buf);
                        ASSERT(buf->b_efunc == NULL);
                        hdr->b_flags |= ARC_BUF_AVAILABLE;
                }
                mutex_exit(hash_lock);
        } else if (HDR_IO_IN_PROGRESS(hdr)) {
                int destroy_hdr;
                /*
                 * We are in the middle of an async write.  Don’t destroy
                 * this buffer unless the write completes before we finish
                 * decrementing the reference count.
                 */
                mutex_enter(&arc_eviction_mtx);
                (void) remove_reference(hdr, NULL, tag);
                ASSERT(refcount_is_zero(&hdr->b_refcnt));
                destroy_hdr = !HDR_IO_IN_PROGRESS(hdr);
                mutex_exit(&arc_eviction_mtx);
                if (destroy_hdr)
                        arc_hdr_destroy(hdr);
        } else {
                if (remove_reference(hdr, NULL, tag) > 0)
                        arc_buf_destroy(buf, FALSE, TRUE);
                else
                        arc_hdr_destroy(hdr);
        }
}

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

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

        mutex_enter(hash_lock);
```

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

        (void) remove_reference(hdr, hash_lock, tag);
        if (hdr->b_datacnt > 1) {
                if (no_callback)
                        arc_buf_destroy(buf, FALSE, TRUE);
        } else if (no_callback) {
                ASSERT(hdr->b_buf == buf && buf->b_next == NULL);
                ASSERT(buf->b_efunc == NULL);
                hdr->b_flags |= ARC_BUF_AVAILABLE;
        }
        ASSERT(no_callback || hdr->b_datacnt > 1 ||
            refcount_is_zero(&hdr->b_refcnt));
        mutex_exit(hash_lock);
        return (no_callback);
}

int
arc_buf_size(arc_buf_t *buf)
{
        return (buf->b_hdr->b_size);
}

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

        if (zfs_disable_dup_eviction)
                return (B_FALSE);

        mutex_enter(&buf->b_evict_lock);
        hdr = buf->b_hdr;
        if (hdr == NULL) {
                /*
                 * We are in arc_do_user_evicts(); let that function
                 * perform the eviction.
                 */
                ASSERT(buf->b_data == NULL);
                mutex_exit(&buf->b_evict_lock);
                return (B_FALSE);
        } else if (buf->b_data == NULL) {
                /*
                 * We have already been added to the arc eviction list;
                 * recommend eviction.
                 */
                ASSERT3P(hdr, ==, &arc_eviction_hdr);
                mutex_exit(&buf->b_evict_lock);
                return (B_TRUE);
        }

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

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

```
}

/*
 * Evict buffers from list until we’ve removed the specified number of
 * bytes.  Move the removed buffers to the appropriate evict state.
 * If the recycle flag is set, then attempt to "recycle" a buffer:
 * - look for a buffer to evict that is ‘bytes’ long.
 * - return the data block from this buffer rather than freeing it.
 * This flag is used by callers that are trying to make space for a
 * new buffer in a full arc cache.
 *
 * This function makes a "best effort".  It skips over any buffers
 * it can’t get a hash_lock on, and so may not catch all candidates.
 * It may also return without evicting as much space as requested.
 */
static void *
arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle,
    arc_buf_contents_t type)
{
        arc_state_t *evicted_state;
        uint64_t bytes_evicted = 0, skipped = 0, missed = 0;
        arc_buf_hdr_t *ab, *ab_prev = NULL;
        list_t *list = &state->arcs_list[type];
        kmutex_t *hash_lock;
        boolean_t have_lock;
        void *stolen = NULL;
        arc_buf_hdr_t marker = { 0 };
        int count = 0;

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

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

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

        for (ab = list_tail(list); ab; ab = ab_prev) {
                ab_prev = list_prev(list, ab);
                /* prefetch buffers have a minimum lifespan */
                if (HDR_IO_IN_PROGRESS(ab) ||
                    (spa && ab->b_spa != spa) ||
                    (ab->b_flags & (ARC_PREFETCH|ARC_INDIRECT) &&
                    ddi_get_lbolt() - ab->b_arc_access <
                    arc_min_prefetch_lifespan)) {
                        skipped++;
                        continue;
                }
                /* "lookahead" for better eviction candidate */
                if (recycle && ab->b_size != bytes &&
                    ab_prev && ab_prev->b_size == bytes)
                        continue;

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

                /*
                 * It may take a long time to evict all the bufs requested.
                 * To avoid blocking all arc activity, periodically drop
                 * the arcs_mtx and give other threads a chance to run
                 * before reacquiring the lock.
                 *
                 * If we are looking for a buffer to recycle, we are in
                 * the hot code path, so don’t sleep.
                 */
                if (!recycle && count++ > arc_evict_iterations) {
```

```
                        list_insert_after(list, ab, &marker);
                        mutex_exit(&evicted_state->arcs_mtx);
                        mutex_exit(&state->arcs_mtx);
                        kpreempt(KPREEMPT_SYNC);
                        mutex_enter(&state->arcs_mtx);
                        mutex_enter(&evicted_state->arcs_mtx);
                        ab_prev = list_prev(list, &marker);
                        list_remove(list, &marker);
                        count = 0;
                        continue;
                }

                hash_lock = HDR_LOCK(ab);
                have_lock = MUTEX_HELD(hash_lock);
                if (have_lock || mutex_tryenter(hash_lock)) {
                        ASSERT0(refcount_count(&ab->b_refcnt));
                        ASSERT(ab->b_datacnt > 0);
                        while (ab->b_buf) {
                                arc_buf_t *buf = ab->b_buf;
                                if (!mutex_tryenter(&buf->b_evict_lock)) {
                                        missed += 1;
                                        break;
                                }
                                if (buf->b_data) {
                                        bytes_evicted += ab->b_size;
                                        if (recycle && ab->b_type == type &&
                                            ab->b_size == bytes &&
                                            !HDR_L2_WRITING(ab)) {
                                                stolen = buf->b_data;
                                                recycle = FALSE;
                                        }
                                }
                                if (buf->b_efunc) {
                                        mutex_enter(&arc_eviction_mtx);
                                        arc_buf_destroy(buf,
                                            buf->b_data == stolen, FALSE);
                                        ab->b_buf = buf->b_next;
                                        buf->b_hdr = &arc_eviction_hdr;
                                        buf->b_next = arc_eviction_list;
                                        arc_eviction_list = buf;
                                        mutex_exit(&arc_eviction_mtx);
                                        mutex_exit(&buf->b_evict_lock);
                                } else {
                                        mutex_exit(&buf->b_evict_lock);
                                        arc_buf_destroy(buf,
                                            buf->b_data == stolen, TRUE);
                                }
                        }

                        if (ab->b_l2hdr) {
                                ARCSTAT_INCR(arcstat_evict_l2_cached,
                                    ab->b_size);
                        } else {
                                if (l2arc_write_eligible(ab->b_spa, ab)) {
                                        ARCSTAT_INCR(arcstat_evict_l2_eligible,
                                            ab->b_size);
                                } else {
                                        ARCSTAT_INCR(
                                            arcstat_evict_l2_ineligible,
                                            ab->b_size);
                                }
                        }

                        if (ab->b_datacnt == 0) {
                                arc_change_state(evicted_state, ab, hash_lock);
                                ASSERT(HDR_IN_HASH_TABLE(ab));
                                ab->b_flags |= ARC_IN_HASH_TABLE;
                                ab->b_flags &= ~ARC_BUF_AVAILABLE;
                                DTRACE_PROBE1(arc__evict, arc_buf_hdr_t *, ab);
                        }
                        if (!have_lock)
                                mutex_exit(hash_lock);
                        if (bytes >= 0 && bytes_evicted >= bytes)
                                break;
                } else {
                        missed += 1;
                }
        }

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

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

        if (skipped)
                ARCSTAT_INCR(arcstat_evict_skip, skipped);

        if (missed)
                ARCSTAT_INCR(arcstat_mutex_miss, missed);

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

        return (stolen);
}

/*
 * Remove buffers from list until we've removed the specified number of
 * bytes.  Destroy the buffers that are removed.
 */
static void
arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes)
{
        arc_buf_hdr_t *ab, *ab_prev;
        arc_buf_hdr_t marker = { 0 };
        list_t *list = &state->arcs_list[ARC_BUFC_DATA];
        kmutex_t *hash_lock;
        uint64_t bytes_deleted = 0;
        uint64_t bufs_skipped = 0;
        int count = 0;

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

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

                hash_lock = HDR_LOCK(ab);
```

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

                /*
                 * It may take a long time to evict all the bufs requested.
                 * To avoid blocking all arc activity, periodically drop
                 * the arcs_mtx and give other threads a chance to run
                 * before reacquiring the lock.
                 */
                if (count++ > arc_evict_iterations) {
                        list_insert_after(list, ab, &marker);
                        mutex_exit(&state->arcs_mtx);
                        kpreempt(KPREEMPT_SYNC);
                        mutex_enter(&state->arcs_mtx);
                        ab_prev = list_prev(list, &marker);
                        list_remove(list, &marker);
                        count = 0;
                        continue;
                }
                if (mutex_tryenter(hash_lock)) {
                        ASSERT(!HDR_IO_IN_PROGRESS(ab));
                        ASSERT(ab->b_buf == NULL);
                        ARCSTAT_BUMP(arcstat_deleted);
                        bytes_deleted += ab->b_size;

                        if (ab->b_l2hdr != NULL) {
                                /*
                                 * This buffer is cached on the 2nd Level ARC;
                                 * don't destroy the header.
                                 */
                                arc_change_state(arc_l2c_only, ab, hash_lock);
                                mutex_exit(hash_lock);
                        } else {
                                arc_change_state(arc_anon, ab, hash_lock);
                                mutex_exit(hash_lock);
                                arc_hdr_destroy(ab);
                        }

                        DTRACE_PROBE1(arc__delete, arc_buf_hdr_t *, ab);
                        if (bytes >= 0 && bytes_deleted >= bytes)
                                break;
                } else if (bytes < 0) {
                        /*
                         * Insert a list marker and then wait for the
                         * hash lock to become available. Once its
                         * available, restart from where we left off.
                         */
                        list_insert_after(list, ab, &marker);
                        mutex_exit(&state->arcs_mtx);
                        mutex_enter(hash_lock);
                        mutex_exit(hash_lock);
                        mutex_enter(&state->arcs_mtx);
                        ab_prev = list_prev(list, &marker);
                        list_remove(list, &marker);
                } else {
                        bufs_skipped += 1;
                }

        }
        mutex_exit(&state->arcs_mtx);

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

        if (bufs_skipped) {
                ARCSTAT_INCR(arcstat_mutex_miss, bufs_skipped);
                ASSERT(bytes >= 0);
        }

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

static void
arc_adjust(void)
{
        int64_t adjustment, delta;

        /*
         * Adjust MRU size
         */

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

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

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

        /*
         * Adjust MFU size
         */

        adjustment = arc_size - arc_c;

        if (adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_DATA] > 0) {
                delta = MIN(adjustment, arc_mfu->arcs_lsize[ARC_BUFC_DATA]);
                (void) arc_evict(arc_mfu, NULL, delta, FALSE, ARC_BUFC_DATA);
                adjustment -= delta;
        }

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

        /*
         * Adjust ghost lists
         */

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

        if (adjustment > 0 && arc_mru_ghost->arcs_size > 0) {
                delta = MIN(arc_mru_ghost->arcs_size, adjustment);
                arc_evict_ghost(arc_mru_ghost, NULL, delta);
        }
```

```

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

        if (adjustment > 0 && arc_mfu_ghost->arcs_size > 0) {
                delta = MIN(arc_mfu_ghost->arcs_size, adjustment);
                arc_evict_ghost(arc_mfu_ghost, NULL, delta);
        }
}

static void
arc_do_user_evicts(void)
{
        mutex_enter(&arc_eviction_mtx);
        while (arc_eviction_list != NULL) {
                arc_buf_t *buf = arc_eviction_list;
                arc_eviction_list = buf->b_next;
                mutex_enter(&buf->b_evict_lock);
                buf->b_hdr = NULL;
                mutex_exit(&buf->b_evict_lock);
                mutex_exit(&arc_eviction_mtx);

                if (buf->b_efunc != NULL)
                        VERIFY0(buf->b_efunc(buf->b_private));

                buf->b_efunc = NULL;
                buf->b_private = NULL;
                kmem_cache_free(buf_cache, buf);
                mutex_enter(&arc_eviction_mtx);
        }
        mutex_exit(&arc_eviction_mtx);
}

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

        if (spa)
                guid = spa_load_guid(spa);

        while (list_head(&arc_mru->arcs_list[ARC_BUFC_DATA])) {
                (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_DATA);
                if (spa)
                        break;
        }
        while (list_head(&arc_mru->arcs_list[ARC_BUFC_METADATA])) {
                (void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_METADATA);
                if (spa)
                        break;
        }
        while (list_head(&arc_mfu->arcs_list[ARC_BUFC_DATA])) {
                (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_DATA);
                if (spa)
                        break;
        }
        while (list_head(&arc_mfu->arcs_list[ARC_BUFC_METADATA])) {
                (void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_METADATA);
                if (spa)
                        break;
        }
        arc_evict_ghost(arc_mru_ghost, guid, -1);
        arc_evict_ghost(arc_mfu_ghost, guid, -1);

        mutex_enter(&arc_reclaim_thr_lock);
        arc_do_user_evicts();
        mutex_exit(&arc_reclaim_thr_lock);
        ASSERT(spa || arc_eviction_list == NULL);
}

void
arc_shrink(void)
{
        if (arc_c > arc_c_min) {
                uint64_t to_free;

#ifdef _KERNEL
                to_free = MAX(arc_c >> arc_shrink_shift, ptob(needfree));
#else
                to_free = arc_c >> arc_shrink_shift;
#endif
                if (arc_c > arc_c_min + to_free)
                        atomic_add_64(&arc_c, -to_free);
                else
                        arc_c = arc_c_min;

                atomic_add_64(&arc_p, -(arc_p >> arc_shrink_shift));
                if (arc_c > arc_size)
                        arc_c = MAX(arc_size, arc_c_min);
                if (arc_p > arc_c)
                        arc_p = (arc_c >> 1);
                ASSERT(arc_c >= arc_c_min);
                ASSERT((int64_t)arc_p >= 0);
        }

        if (arc_size > arc_c)
                arc_adjust();
}

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

#ifdef _KERNEL

        if (needfree)
                return (1);

        /*
         * take ’desfree’ extra pages, so we reclaim sooner, rather than later
         */
        extra = desfree;

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

```
                return (1);

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

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

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

        /*
         * If zio data pages are being allocated out of a separate heap segment,
         * then enforce that the size of available vmem for this arena remains
         * above about 1/16th free.
         *
         * Note: The 1/16th arena free requirement was put in place
         * to aggressively evict memory from the arc in order to avoid
         * memory fragmentation issues.
         */
        if (zio_arena != NULL &&
            vmem_size(zio_arena, VMEM_FREE) <
            (vmem_size(zio_arena, VMEM_ALLOC) >> 4))
                return (1);
#else
        if (spa_get_random(100) == 0)
                return (1);
#endif
        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[];
        extern kmem_cache_t     *range_seg_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);
        kmem_cache_reap_now(range_seg_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) {
                mult = ((arc_mru_ghost->arcs_size >= arc_mfu_ghost->arcs_size) ?
                    1 : (arc_mfu_ghost->arcs_size/arc_mru_ghost->arcs_size));
                mult = MIN(mult, 10); /* avoid wild arc_p adjustment */

                arc_p = MIN(arc_c - arc_p_min, arc_p + bytes * mult);
        } else if (state == arc_mfu_ghost) {
                uint64_t delta;
                mult = ((arc_mfu_ghost->arcs_size >= arc_mru_ghost->arcs_size) ?
                    1 : (arc_mru_ghost->arcs_size/arc_mfu_ghost->arcs_size));
                mult = MIN(mult, 10);

                delta = MIN(bytes * mult, arc_p);
                arc_p = MAX(arc_p_min, arc_p - delta);
        }
        ASSERT((int64_t)arc_p >= 0);

        if (arc_reclaim_needed()) {
                cv_signal(&arc_reclaim_thr_cv);
                return;
        }

        if (arc_no_grow)
                return;

        if (arc_c >= arc_c_max)
                return;

        /*
         * If we're within (2 * maxblocksize) bytes of the target
         * cache size, increment the target cache size
         */
        if (arc_size > arc_c - (2ULL << SPA_MAXBLOCKSHIFT)) {
                atomic_add_64(&arc_c, (int64_t)bytes);
                if (arc_c > arc_c_max)
                        arc_c = arc_c_max;
                else if (state == arc_anon)
                        atomic_add_64(&arc_p, (int64_t)bytes);
                if (arc_p > arc_c)
                        arc_p = arc_c;
        }
        ASSERT((int64_t)arc_p >= 0);
}

/*
 * Check if the cache has reached its limits and eviction is required
 * prior to insert.
 */
static int
arc_evict_needed(arc_buf_contents_t type)
{
        if (type == ARC_BUFC_METADATA && arc_meta_used >= arc_meta_limit)
                return (1);

        if (arc_reclaim_needed())
                return (1);

        return (arc_size > arc_c);
}

/*
 * The buffer, supplied as the first argument, needs a data block.
 * So, if we are at cache max, determine which cache should be victimized.
 * We have the following cases:
 *
 * 1. Insert for MRU, p > sizeof(arc_anon + arc_mru) ->
 * In this situation if we're out of space, but the resident size of the MFU is
 * under the limit, victimize the MFU cache to satisfy this insertion request.
 *
 * 2. Insert for MRU, p <= sizeof(arc_anon + arc_mru) ->
 * Here, we've used up all of the available space for the MRU, so we need to
 * evict from our own cache instead.  Evict from the set of resident MRU
 * entries.
```

```
 *
 * 3. Insert for MFU (c - p) > sizeof(arc_mfu) ->
 * c minus p represents the MFU space in the cache, since p is the size of the
 * cache that is dedicated to the MRU.  In this situation there's still space on
 * the MFU side, so the MRU side needs to be victimized.
 *
 * 4. Insert for MFU (c - p) < sizeof(arc_mfu) ->
 * MFU's resident set is consuming more space than it has been allotted.  In
 * this situation, we must victimize our own cache, the MFU, for this insertion.
 */
static void
arc_get_data_buf(arc_buf_t *buf)
{
        arc_state_t             *state = buf->b_hdr->b_state;
        uint64_t                size = buf->b_hdr->b_size;
        arc_buf_contents_t      type = buf->b_hdr->b_type;

        arc_adapt(size, state);

        /*
         * We have not yet reached cache maximum size,
         * just allocate a new buffer.
         */
        if (!arc_evict_needed(type)) {
                if (type == ARC_BUFC_METADATA) {
                        buf->b_data = zio_buf_alloc(size);
                        arc_space_consume(size, ARC_SPACE_DATA);
                } else {
                        ASSERT(type == ARC_BUFC_DATA);
                        buf->b_data = zio_data_buf_alloc(size);
                        ARCSTAT_INCR(arcstat_data_size, size);
                        atomic_add_64(&arc_size, size);
                }
                goto out;
        }

        /*
         * If we are prefetching from the mfu ghost list, this buffer
         * will end up on the mru list; so steal space from there.
         */
        if (state == arc_mfu_ghost)
                state = buf->b_hdr->b_flags & ARC_PREFETCH ? arc_mru : arc_mfu;
        else if (state == arc_mru_ghost)
                state = arc_mru;

        if (state == arc_mru || state == arc_anon) {
                uint64_t mru_used = arc_anon->arcs_size + arc_mru->arcs_size;
                state = (arc_mfu->arcs_lsize[type] >= size &&
                    arc_p > mru_used) ? arc_mfu : arc_mru;
        } else {
                /* MFU cases */
                uint64_t mfu_space = arc_c - arc_p;
                state =  (arc_mru->arcs_lsize[type] >= size &&
                    mfu_space > arc_mfu->arcs_size) ? arc_mru : arc_mfu;
        }
        if ((buf->b_data = arc_evict(state, NULL, size, TRUE, type)) == NULL) {
                if (type == ARC_BUFC_METADATA) {
                        buf->b_data = zio_buf_alloc(size);
                        arc_space_consume(size, ARC_SPACE_DATA);
                } else {
                        ASSERT(type == ARC_BUFC_DATA);
                        buf->b_data = zio_data_buf_alloc(size);
                        ARCSTAT_INCR(arcstat_data_size, size);
                        atomic_add_64(&arc_size, size);
                }
                ARCSTAT_BUMP(arcstat_recycle_miss);
        }
        ASSERT(buf->b_data != NULL);
out:
        /*
         * Update the state size.  Note that ghost states have a
         * "ghost size" and so don’t need to be updated.
         */
        if (!GHOST_STATE(buf->b_hdr->b_state)) {
                arc_buf_hdr_t *hdr = buf->b_hdr;

                atomic_add_64(&hdr->b_state->arcs_size, size);
                if (list_link_active(&hdr->b_arc_node)) {
                        ASSERT(refcount_is_zero(&hdr->b_refcnt));
                        atomic_add_64(&hdr->b_state->arcs_lsize[type], size);
                }
                /*
                 * If we are growing the cache, and we are adding anonymous
                 * data, and we have outgrown arc_p, update arc_p
                 */
                if (arc_size < arc_c && hdr->b_state == arc_anon &&
                    arc_anon->arcs_size + arc_mru->arcs_size > arc_p)
                        arc_p = MIN(arc_c, arc_p + size);
        }
}

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

        ASSERT(MUTEX_HELD(hash_lock));

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

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

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

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

```
                                buf->b_arc_access = now;
                                return;
                }

                /*
                 * This buffer has been "accessed" only once so far,
                 * but it is still in the cache. Move it to the MFU
                 * state.
                 */
                if (now > buf->b_arc_access + ARC_MINTIME) {
                        /*
                         * More than 125ms have passed since we
                         * instantiated this buffer.  Move it to the
                         * most frequently used state.
                         */
                        buf->b_arc_access = now;
                        DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
                        arc_change_state(arc_mfu, buf, hash_lock);
                }
                ARCSTAT_BUMP(arcstat_mru_hits);
        } else if (buf->b_state == arc_mru_ghost) {
                arc_state_t     *new_state;
                /*
                 * This buffer has been "accessed" recently, but
                 * was evicted from the cache.  Move it to the
                 * MFU state.
                 */

                if (buf->b_flags & ARC_PREFETCH) {
                        new_state = arc_mru;
                        if (refcount_count(&buf->b_refcnt) > 0)
                                buf->b_flags &= ~ARC_PREFETCH;
                        DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
                } else {
                        new_state = arc_mfu;
                        DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
                }

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

                ARCSTAT_BUMP(arcstat_mru_ghost_hits);
        } else if (buf->b_state == arc_mfu) {
                /*
                 * This buffer has been accessed more than once and is
                 * still in the cache.  Keep it in the MFU state.
                 *
                 * NOTE: an add_reference() that occurred when we did
                 * the arc_read() will have kicked this off the list.
                 * If it was a prefetch, we will explicitly move it to
                 * the head of the list now.
                 */
                if ((buf->b_flags & ARC_PREFETCH) != 0) {
                        ASSERT(refcount_count(&buf->b_refcnt) == 0);
                        ASSERT(list_link_active(&buf->b_arc_node));
                }
                ARCSTAT_BUMP(arcstat_mfu_hits);
                buf->b_arc_access = ddi_get_lbolt();
        } else if (buf->b_state == arc_mfu_ghost) {
                arc_state_t     *new_state = arc_mfu;
                /*
                 * This buffer has been accessed more than once but has
                 * been evicted from the cache.  Move it back to the
                 * MFU state.
                 */
                if (buf->b_flags & ARC_PREFETCH) {
                        /*
                         * This is a prefetch access...
                         * move this block back to the MRU state.
                         */
                        ASSERT0(refcount_count(&buf->b_refcnt));
                        new_state = arc_mru;
                }

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

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

                buf->b_arc_access = ddi_get_lbolt();
                DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
                arc_change_state(arc_mfu, buf, hash_lock);
        } else {
                ASSERT(!"invalid arc state");
        }
}

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

/* a generic arc_done_func_t */
void
arc_getbuf_func(zio_t *zio, arc_buf_t *buf, void *arg)
{
        arc_buf_t **bufp = arg;
        if (zio && zio->io_error) {
                VERIFY(arc_buf_remove_ref(buf, arg));
                *bufp = NULL;
        } else {
                *bufp = buf;
                ASSERT(buf->b_data);
        }
}

static void
arc_read_done(zio_t *zio)
{
        arc_buf_hdr_t   *hdr;
        arc_buf_t       *buf;
        arc_buf_t       *abuf;  /* buffer we’re assigning to callback */
        kmutex_t        *hash_lock = NULL;
        arc_callback_t  *callback_list, *acb;
        int             freeable = FALSE;

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

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

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

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

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

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

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

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

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

        /* create copies of the data buffer for the callers */
        abuf = buf;
        for (acb = callback_list; acb; acb = acb->acb_next) {
                if (acb->acb_done) {
                        if (abuf == NULL) {
                                ARCSTAT_BUMP(arcstat_duplicate_reads);
                                abuf = arc_buf_clone(buf);
                        }
                        acb->acb_buf = abuf;
                        abuf = NULL;
                }
        }
        hdr->b_acb = NULL;
        hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
        ASSERT(!HDR_BUF_AVAILABLE(hdr));
        if (abuf == buf) {
                ASSERT(buf->b_efunc == NULL);
                ASSERT(hdr->b_datacnt == 1);
                hdr->b_flags |= ARC_BUF_AVAILABLE;
        }

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

        if (zio->io_error != 0) {
                hdr->b_flags |= ARC_IO_ERROR;
                if (hdr->b_state != arc_anon)
                        arc_change_state(arc_anon, hdr, hash_lock);
                if (HDR_IN_HASH_TABLE(hdr))
                        buf_hash_remove(hdr);
                freeable = refcount_is_zero(&hdr->b_refcnt);
        }

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

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

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

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

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

        if (freeable)
                arc_hdr_destroy(hdr);
}

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

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

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

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

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

                *arc_flags |= ARC_CACHED;

                if (HDR_IO_IN_PROGRESS(hdr)) {

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

                        if (done) {
                                arc_callback_t  *acb = NULL;

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

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

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

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

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

                if (hdr == NULL) {
                        /* this block is not in the cache */
                        arc_buf_hdr_t *exists = NULL;
                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
                        buf = arc_buf_alloc(spa, size, private, type);
                        hdr = buf->b_hdr;
                        if (!BP_IS_EMBEDDED(bp)) {
                                hdr->b_dva = *BP_IDENTITY(bp);
                                hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
                                hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
                                exists = buf_hash_insert(hdr, &hash_lock);
                        }
                        if (exists != NULL) {
                                /* somebody beat us to the hash insert */
                                mutex_exit(hash_lock);
                                buf_discard_identity(hdr);
                                (void) arc_buf_remove_ref(buf, private);
                                goto top; /* restart the IO request */
                        }
                        /* if this is a prefetch, we don't have a reference */
```

```
                        if (*arc_flags & ARC_PREFETCH) {
                                (void) remove_reference(hdr, hash_lock,
                                    private);
                                hdr->b_flags |= ARC_PREFETCH;
                        }
                        if (*arc_flags & ARC_L2CACHE)
                                hdr->b_flags |= ARC_L2CACHE;
                        if (*arc_flags & ARC_L2COMPRESS)
                                hdr->b_flags |= ARC_L2COMPRESS;
                        if (BP_GET_LEVEL(bp) > 0)
                                hdr->b_flags |= ARC_INDIRECT;
                } else {
                        /* this block is in the ghost cache */
                        ASSERT(GHOST_STATE(hdr->b_state));
                        ASSERT(!HDR_IO_IN_PROGRESS(hdr));
                        ASSERT0(refcount_count(&hdr->b_refcnt));
                        ASSERT(hdr->b_buf == NULL);

                        /* if this is a prefetch, we don't have a reference */
                        if (*arc_flags & ARC_PREFETCH)
                                hdr->b_flags |= ARC_PREFETCH;
                        else
                                add_reference(hdr, hash_lock, private);
                        if (*arc_flags & ARC_L2CACHE)
                                hdr->b_flags |= ARC_L2CACHE;
                        if (*arc_flags & ARC_L2COMPRESS)
                                hdr->b_flags |= ARC_L2COMPRESS;
                        buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
                        buf->b_hdr = hdr;
                        buf->b_data = NULL;
                        buf->b_efunc = NULL;
                        buf->b_private = NULL;
                        buf->b_next = NULL;
                        hdr->b_buf = buf;
                        ASSERT(hdr->b_datacnt == 0);
                        hdr->b_datacnt = 1;
                        arc_get_data_buf(buf);
                        arc_access(hdr, hash_lock);
                }

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

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

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

                if (hdr->b_l2hdr != NULL &&
                    (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
                        devw = hdr->b_l2hdr->b_dev->l2ad_writing;
                        addr = hdr->b_l2hdr->b_daddr;
                        b_compress = hdr->b_l2hdr->b_compress;
                        b_asize = hdr->b_l2hdr->b_asize;
                        /*
                         * Lock out device removal.
                         */
                        if (vdev_is_dead(vd) ||
                            !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
                                vd = NULL;
                }

                if (hash_lock != NULL)
                        mutex_exit(hash_lock);
                /*
                 * At this point, we have a level 1 cache miss.  Try again in
                 * L2ARC if possible.
                 */
                ASSERT3U(hdr->b_size, ==, size);
                DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
                    uint64_t, size, zbookmark_phys_t *, zb);
                ARCSTAT_BUMP(arcstat_misses);
                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
                    data, metadata, misses);

                if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
                        /*
                         * Read from the L2ARC if the following are true:
                         * 1. The L2ARC vdev was previously cached.
                         * 2. This buffer still has L2ARC metadata.
                         * 3. This buffer isn’t currently writing to the L2ARC.
                         * 4. The L2ARC entry wasn’t evicted, which may
                         *    also have invalidated the vdev.
                         * 5. This isn’t prefetch and l2arc_noprefetch is set.
                         */
                        if (hdr->b_l2hdr != NULL &&
                            !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
                            !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
                                l2arc_read_callback_t *cb;

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

                                cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
                                    KM_SLEEP);
                                cb->l2rcb_buf = buf;
                                cb->l2rcb_spa = spa;
                                cb->l2rcb_bp = *bp;
                                cb->l2rcb_zb = *zb;
                                cb->l2rcb_flags = zio_flags;
                                cb->l2rcb_compress = b_compress;

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

                                /*
                                 * l2arc read.  The SCL_L2ARC lock will be
                                 * released by l2arc_read_done().
                                 * Issue a null zio if the underlying buffer
                                 * was squashed to zero size by compression.
                                 */
                                if (b_compress == ZIO_COMPRESS_EMPTY) {
                                        rzio = zio_null(pio, spa, vd,
                                            l2arc_read_done, cb,
                                            zio_flags | ZIO_FLAG_DONT_CACHE |
                                            ZIO_FLAG_CANFAIL |
                                            ZIO_FLAG_DONT_PROPAGATE |
                                            ZIO_FLAG_DONT_RETRY);
                                } else {
                                        rzio = zio_read_phys(pio, vd, addr,
                                            b_asize, buf->b_data,
                                            ZIO_CHECKSUM_OFF,
                                            l2arc_read_done, cb, priority,
                                            zio_flags | ZIO_FLAG_DONT_CACHE |
                                            ZIO_FLAG_CANFAIL |
                                            ZIO_FLAG_DONT_PROPAGATE |
                                            ZIO_FLAG_DONT_RETRY, B_FALSE);
```

```
                                }
                                DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
                                    zio_t *, rzio);
                                ARCSTAT_INCR(arcstat_l2_read_bytes, b_asize);

                                if (*arc_flags & ARC_NOWAIT) {
                                        zio_nowait(rzio);
                                        return (0);
                                }

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

                                /* l2arc read error; goto zio_read() */
                        } else {
                                DTRACE_PROBE1(l2arc__miss,
                                    arc_buf_hdr_t *, hdr);
                                ARCSTAT_BUMP(arcstat_l2_misses);
                                if (HDR_L2_WRITING(hdr))
                                        ARCSTAT_BUMP(arcstat_l2_rw_clash);
                                spa_config_exit(spa, SCL_L2ARC, vd);
                        }
                } else {
                        if (vd != NULL)
                                spa_config_exit(spa, SCL_L2ARC, vd);
                        if (l2arc_ndev != 0) {
                                DTRACE_PROBE1(l2arc__miss,
                                    arc_buf_hdr_t *, hdr);
                                ARCSTAT_BUMP(arcstat_l2_misses);
                        }
                }

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

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

                ASSERT(*arc_flags & ARC_NOWAIT);
                zio_nowait(rzio);
        }
        return (0);
}

void
arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private)
{
        ASSERT(buf->b_hdr != NULL);
        ASSERT(buf->b_hdr->b_state != arc_anon);
        ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt) || func == NULL);
        ASSERT(buf->b_efunc == NULL);
        ASSERT(!HDR_BUF_AVAILABLE(buf->b_hdr));

        buf->b_efunc = func;
        buf->b_private = private;
}

/*
 * Notify the arc that a block was freed, and thus will never be used again.
 */
void
arc_freed(spa_t *spa, const blkptr_t *bp)
{
        arc_buf_hdr_t *hdr;
        kmutex_t *hash_lock;
        uint64_t guid = spa_load_guid(spa);

        ASSERT(!BP_IS_EMBEDDED(bp));

        hdr = buf_hash_find(guid, bp, &hash_lock);
        if (hdr == NULL)
                return;
        if (HDR_BUF_AVAILABLE(hdr)) {
                arc_buf_t *buf = hdr->b_buf;
                add_reference(hdr, hash_lock, FTAG);
                hdr->b_flags &= ~ARC_BUF_AVAILABLE;
                mutex_exit(hash_lock);

                arc_release(buf, FTAG);
                (void) arc_buf_remove_ref(buf, FTAG);
        } else {
                mutex_exit(hash_lock);
        }

}

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

        mutex_enter(&buf->b_evict_lock);
        hdr = buf->b_hdr;
        if (hdr == NULL) {
                /*
                 * We are in arc_do_user_evicts().
                 */
                ASSERT(buf->b_data == NULL);
                mutex_exit(&buf->b_evict_lock);
                return (B_FALSE);
        } else if (buf->b_data == NULL) {
                /*
                 * We are on the eviction list; process this buffer now
                 * but let arc_do_user_evicts() do the reaping.
                 */
                buf->b_efunc = NULL;
                mutex_exit(&buf->b_evict_lock);
                VERIFY0(efunc(private));
                return (B_TRUE);
        }
        hash_lock = HDR_LOCK(hdr);
        mutex_enter(hash_lock);
        hdr = buf->b_hdr;
        ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
```

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

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

        if (hdr->b_datacnt > 1) {
                mutex_exit(&buf->b_evict_lock);
                arc_buf_destroy(buf, FALSE, TRUE);
        } else {
                ASSERT(buf == hdr->b_buf);
                hdr->b_flags |= ARC_BUF_AVAILABLE;
                mutex_exit(&buf->b_evict_lock);
        }

        mutex_exit(hash_lock);
        VERIFY0(efunc(private));
        return (B_TRUE);
}

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

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

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

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

        if (hdr->b_state == arc_anon) {
                /* this buffer is already released */
                ASSERT(buf->b_efunc == NULL);
        } else {
                hash_lock = HDR_LOCK(hdr);
                mutex_enter(hash_lock);
                hdr = buf->b_hdr;
                ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
        }

        l2hdr = hdr->b_l2hdr;
        if (l2hdr) {
                mutex_enter(&l2arc_buflist_mtx);
                arc_buf_l2_cdata_free(hdr);
#endif /* ! codereview */
                hdr->b_l2hdr = NULL;
                list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
        }
        buf_size = hdr->b_size;
        /*
         * Do we have more than one buf?
         */
        if (hdr->b_datacnt > 1) {
                arc_buf_hdr_t *nhdr;
                arc_buf_t **bufp;
                uint64_t blksz = hdr->b_size;
                uint64_t spa = hdr->b_spa;
                arc_buf_contents_t type = hdr->b_type;
                uint32_t flags = hdr->b_flags;

                ASSERT(hdr->b_buf != buf || buf->b_next != NULL);
                /*
                 * Pull the data off of this hdr and attach it to
                 * a new anonymous hdr.
                 */
                (void) remove_reference(hdr, hash_lock, tag);
                bufp = &hdr->b_buf;
                while (*bufp != buf)
                        bufp = &(*bufp)->b_next;
                *bufp = buf->b_next;
                buf->b_next = NULL;

                ASSERT3U(hdr->b_state->arcs_size, >=, hdr->b_size);
                atomic_add_64(&hdr->b_state->arcs_size, -hdr->b_size);
                if (refcount_is_zero(&hdr->b_refcnt)) {
                        uint64_t *size = &hdr->b_state->arcs_lsize[hdr->b_type];
                        ASSERT3U(*size, >=, hdr->b_size);
                        atomic_add_64(size, -hdr->b_size);
                }

                /*
                 * We’re releasing a duplicate user data buffer, update
                 * our statistics accordingly.
                 */
                if (hdr->b_type == ARC_BUFC_DATA) {
                        ARCSTAT_BUMPDOWN(arcstat_duplicate_buffers);
                        ARCSTAT_INCR(arcstat_duplicate_buffers_size,
                            -hdr->b_size);
                }
                hdr->b_datacnt -= 1;
                arc_cksum_verify(buf);
                arc_buf_unwatch(buf);

                mutex_exit(hash_lock);

                nhdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
                nhdr->b_size = blksz;
                nhdr->b_spa = spa;
                nhdr->b_type = type;
                nhdr->b_buf = buf;
                nhdr->b_state = arc_anon;
                nhdr->b_arc_access = 0;
                nhdr->b_flags = flags & ARC_L2_WRITING;
                nhdr->b_l2hdr = NULL;
                nhdr->b_datacnt = 1;
                nhdr->b_freeze_cksum = NULL;
                (void) refcount_add(&nhdr->b_refcnt, tag);
                buf->b_hdr = nhdr;
                mutex_exit(&buf->b_evict_lock);
                atomic_add_64(&arc_anon->arcs_size, blksz);
        } else {
                mutex_exit(&buf->b_evict_lock);
                ASSERT(refcount_count(&hdr->b_refcnt) == 1);
                ASSERT(!list_link_active(&hdr->b_arc_node));
                ASSERT(!HDR_IO_IN_PROGRESS(hdr));
```

```
                if (hdr->b_state != arc_anon)
                        arc_change_state(arc_anon, hdr, hash_lock);
                hdr->b_arc_access = 0;
                if (hash_lock)
                        mutex_exit(hash_lock);

                buf_discard_identity(hdr);
                arc_buf_thaw(buf);
        }
        buf->b_efunc = NULL;
        buf->b_private = NULL;

        if (l2hdr) {
                ARCSTAT_INCR(arcstat_l2_asize, -l2hdr->b_asize);
                vdev_space_update(l2hdr->b_dev->l2ad_vdev,
                    -l2hdr->b_asize, 0, 0);
                kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
                ARCSTAT_INCR(arcstat_l2_size, -buf_size);
                mutex_exit(&l2arc_buflist_mtx);
        }
}

int
arc_released(arc_buf_t *buf)
{
        int released;

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

#ifdef ZFS_DEBUG
int
arc_referenced(arc_buf_t *buf)
{
        int referenced;

        mutex_enter(&buf->b_evict_lock);
        referenced = (refcount_count(&buf->b_hdr->b_refcnt));
        mutex_exit(&buf->b_evict_lock);
        return (referenced);
}
#endif

static void
arc_write_ready(zio_t *zio)
{
        arc_write_callback_t *callback = zio->io_private;
        arc_buf_t *buf = callback->awcb_buf;
        arc_buf_hdr_t *hdr = buf->b_hdr;

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

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

```
			hdr->b_freeze_cksum = NULL;
		}
		mutex_exit(&hdr->b_freeze_lock);
	}
	arc_cksum_compute(buf, B_FALSE);
	hdr->b_flags |= ARC_IO_IN_PROGRESS;
}

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

static void
arc_write_done(zio_t *zio)
{
	arc_write_callback_t *callback = zio->io_private;
	arc_buf_t *buf = callback->awcb_buf;
	arc_buf_hdr_t *hdr = buf->b_hdr;

	ASSERT(hdr->b_acb == NULL);

	if (zio->io_error == 0) {
		if (BP_IS_HOLE(zio->io_bp) || BP_IS_EMBEDDED(zio->io_bp)) {
			buf_discard_identity(hdr);
		} else {
			hdr->b_dva = *BP_IDENTITY(zio->io_bp);
			hdr->b_birth = BP_PHYSICAL_BIRTH(zio->io_bp);
			hdr->b_cksum0 = zio->io_bp->blk_cksum.zc_word[0];
		}
	} else {
		ASSERT(BUF_EMPTY(hdr));
	}

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

		ASSERT(zio->io_error == 0);

		arc_cksum_verify(buf);

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

```
				ASSERT(refcount_is_zero(&exists->b_refcnt));
				arc_change_state(arc_anon, exists, hash_lock);
				mutex_exit(hash_lock);
				arc_hdr_destroy(exists);
				exists = buf_hash_insert(hdr, &hash_lock);
				ASSERT3P(exists, ==, NULL);
			} else if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
				/* nopwrite */
				ASSERT(zio->io_prop.zp_nopwrite);
				if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
					panic("bad nopwrite, hdr=%p exists=%p",
					    (void *)hdr, (void *)exists);
			} else {
				/* Dedup */
				ASSERT(hdr->b_datacnt == 1);
				ASSERT(hdr->b_state == arc_anon);
				ASSERT(BP_GET_DEDUP(zio->io_bp));
				ASSERT(BP_GET_LEVEL(zio->io_bp) == 0);
			}
		}
		hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
		/* if it's not anon, we are doing a scrub */
		if (!exists && hdr->b_state == arc_anon)
			arc_access(hdr, hash_lock);
		mutex_exit(hash_lock);
	} else {
		hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
	}

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

	kmem_free(callback, sizeof (arc_write_callback_t));
}

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

	ASSERT(ready != NULL);
	ASSERT(done != NULL);
	ASSERT(!HDR_IO_ERROR(hdr));
	ASSERT((hdr->b_flags & ARC_IO_IN_PROGRESS) == 0);
	ASSERT(hdr->b_acb == NULL);
	if (l2arc)
		hdr->b_flags |= ARC_L2CACHE;
	if (l2arc_compress)
		hdr->b_flags |= ARC_L2COMPRESS;
	callback = kmem_zalloc(sizeof (arc_write_callback_t), KM_SLEEP);
	callback->awcb_ready = ready;
	callback->awcb_physdone = physdone;
	callback->awcb_done = done;
	callback->awcb_private = private;
	callback->awcb_buf = buf;

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

```
        return (zio);
}

static int
arc_memory_throttle(uint64_t reserve, uint64_t txg)
{
#ifdef _KERNEL
        uint64_t available_memory = ptob(freemem);
        static uint64_t page_load = 0;
        static uint64_t last_txg = 0;

#if defined(__i386)
        available_memory =
            MIN(available_memory, vmem_size(heap_arena, VMEM_FREE));
#endif

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

        if (txg > last_txg) {
                last_txg = txg;
                page_load = 0;
        }
        /*
         * If we are in pageout, we know that memory is already tight,
         * the arc is already going to be evicting, so we just want to
         * continue to let page writes occur as quickly as possible.
         */
        if (curproc == proc_pageout) {
                if (page_load > MAX(ptob(minfree), available_memory) / 4)
                        return (SET_ERROR(ERESTART));
                /* Note: reserve is inflated, so we deflate */
                page_load += reserve / 8;
                return (0);
        } else if (page_load > 0 && arc_reclaim_needed()) {
                /* memory is low, delay before restarting */
                ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
                return (SET_ERROR(EAGAIN));
        }
        page_load = 0;
#endif
        return (0);
}

void
arc_tempreserve_clear(uint64_t reserve)
{
        atomic_add_64(&arc_tempreserve, -reserve);
        ASSERT((int64_t)arc_tempreserve >= 0);
}

int
arc_tempreserve_space(uint64_t reserve, uint64_t txg)
{
        int error;
        uint64_t anon_size;

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

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

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

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

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

        if (reserve + arc_tempreserve + anon_size > arc_c / 2 &&
            anon_size > arc_c / 4) {
                dprintf("failing, arc_tempreserve=%lluK anon_meta=%lluK "
                    "anon_data=%lluK tempreserve=%lluK arc_c=%lluK\n",
                    arc_tempreserve>>10,
                    arc_anon->arcs_lsize[ARC_BUFC_METADATA]>>10,
                    arc_anon->arcs_lsize[ARC_BUFC_DATA]>>10,
                    reserve>>10, arc_c>>10);
                return (SET_ERROR(ERESTART));
        }
        atomic_add_64(&arc_tempreserve, reserve);
        return (0);
}

void
arc_init(void)
{
        mutex_init(&arc_reclaim_thr_lock, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&arc_reclaim_thr_cv, NULL, CV_DEFAULT, NULL);

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

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

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

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

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

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

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

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

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

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

	if (zfs_arc_grow_retry > 0)
		arc_grow_retry = zfs_arc_grow_retry;

	if (zfs_arc_shrink_shift > 0)
		arc_shrink_shift = zfs_arc_shrink_shift;

	if (zfs_arc_p_min_shift > 0)
		arc_p_min_shift = zfs_arc_p_min_shift;

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

	arc_anon = &ARC_anon;
	arc_mru = &ARC_mru;
	arc_mru_ghost = &ARC_mru_ghost;
	arc_mfu = &ARC_mfu;
	arc_mfu_ghost = &ARC_mfu_ghost;
	arc_l2c_only = &ARC_l2c_only;
	arc_size = 0;

	mutex_init(&arc_anon->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
	mutex_init(&arc_mru->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
	mutex_init(&arc_mru_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
	mutex_init(&arc_mfu->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
	mutex_init(&arc_mfu_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
	mutex_init(&arc_l2c_only->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);

	list_create(&arc_mru->arcs_list[ARC_BUFC_METADATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mru->arcs_list[ARC_BUFC_DATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mfu->arcs_list[ARC_BUFC_METADATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mfu->arcs_list[ARC_BUFC_DATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
	list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA],
	    sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
```

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

	buf_init();

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

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

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

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

	arc_dead = FALSE;
	arc_warm = B_FALSE;

	/*
	 * Calculate maximum amount of dirty data per pool.
	 *
	 * If it has been set by /etc/system, take that.
	 * Otherwise, use a percentage of physical memory defined by
	 * zfs_dirty_data_max_percent (default 10%) with a cap at
	 * zfs_dirty_data_max_max (default 4GB).
	 */
	if (zfs_dirty_data_max == 0) {
		zfs_dirty_data_max = physmem * PAGESIZE *
		    zfs_dirty_data_max_percent / 100;
		zfs_dirty_data_max = MIN(zfs_dirty_data_max,
		    zfs_dirty_data_max_max);
	}
}

void
arc_fini(void)
{
	mutex_enter(&arc_reclaim_thr_lock);
	arc_thread_exit = 1;
	while (arc_thread_exit != 0)
		cv_wait(&arc_reclaim_thr_cv, &arc_reclaim_thr_lock);
	mutex_exit(&arc_reclaim_thr_lock);

	arc_flush(NULL);

	arc_dead = TRUE;

	if (arc_ksp != NULL) {
		kstat_delete(arc_ksp);
		arc_ksp = NULL;
	}

	mutex_destroy(&arc_eviction_mtx);
	mutex_destroy(&arc_reclaim_thr_lock);
	cv_destroy(&arc_reclaim_thr_cv);

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

```
        list_destroy(&arc_mfu->arcs_list[ARC_BUFC_METADATA]);
        list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA]);
        list_destroy(&arc_mru->arcs_list[ARC_BUFC_DATA]);
        list_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA]);
        list_destroy(&arc_mfu->arcs_list[ARC_BUFC_DATA]);
        list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA]);

        mutex_destroy(&arc_anon->arcs_mtx);
        mutex_destroy(&arc_mru->arcs_mtx);
        mutex_destroy(&arc_mru_ghost->arcs_mtx);
        mutex_destroy(&arc_mfu->arcs_mtx);
        mutex_destroy(&arc_mfu_ghost->arcs_mtx);
        mutex_destroy(&arc_l2c_only->arcs_mtx);

        buf_fini();

        ASSERT(arc_loaned_bytes == 0);
}

/*
 * Level 2 ARC
 *
 * The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
 * It uses dedicated storage devices to hold cached data, which are populated
 * using large infrequent writes.  The main role of this cache is to boost
 * the performance of random read workloads.  The intended L2ARC devices
 * include short-stroked disks, solid state disks, and other media with
 * substantially faster read latency than disk.
 *
 *                 +-----------------------+
 *                 |         ARC           |
 *                 +-----------------------+
 *                    |         ^     ^
 *                    |         |     |
 *      l2arc_feed_thread()    arc_read()
 *                    |         |     |
 *                    |  l2arc read   |
 *                    V         |     |
 *               +---------------+    |
 *               |     L2ARC     |    |
 *               +---------------+    |
 *                   |    ^           |
 *          l2arc_write() |           |
 *                   |    |           |
 *                   V    |           |
 *                 +-------+      +-------+
 *                 | vdev  |      | vdev  |
 *                 | cache |      | cache |
 *                 +-------+      +-------+
 *                 +=========+     .-----.
 *                 :  L2ARC  :    |-_____-|
 *                 : devices :    | Disks |
 *                 +=========+    '-_____-'
 *
 * Read requests are satisfied from the following sources, in order:
 *
 *      1) ARC
 *      2) vdev cache of L2ARC devices
 *      3) L2ARC devices
 *      4) vdev cache of disks
 *      5) disks
 *
 * Some L2ARC device types exhibit extremely slow write performance.
 * To accommodate for this there are some significant differences between
 * the L2ARC and traditional cache design:
 *
```

```
 * 1. There is no eviction path from the ARC to the L2ARC.  Evictions from
 * the ARC behave as usual, freeing buffers and placing headers on ghost
 * lists.  The ARC does not send buffers to the L2ARC during eviction as
 * this would add inflated write latencies for all ARC memory pressure.
 *
 * 2. The L2ARC attempts to cache data from the ARC before it is evicted.
 * It does this by periodically scanning buffers from the eviction-end of
 * the MFU and MRU ARC lists, copying them to the L2ARC devices if they are
 * not already there. It scans until a headroom of buffers is satisfied,
 * which itself is a buffer for ARC eviction. If a compressible buffer is
 * found during scanning and selected for writing to an L2ARC device, we
 * temporarily boost scanning headroom during the next scan cycle to make
 * sure we adapt to compression effects (which might significantly reduce
 * the data volume we write to L2ARC). The thread that does this is
 * l2arc_feed_thread(), illustrated below; example sizes are included to
 * provide a better sense of ratio than this diagram:
 *
 *             head -->                        tail
 *              +---------------------+----------+
 *      ARC_mfu |:::::#:::::::::::::::|o#o###o###|-->.   # already on L2ARC
 *              +---------------------+----------+   |   o L2ARC eligible
 *      ARC_mru |:#:::::::::::::::::::|#o#ooo####|-->|   : ARC buffer
 *              +---------------------+----------+   |
 *                   15.9 Gbytes      ^ 32 Mbytes    |
 *                                 headroom          |
 *                                            l2arc_feed_thread()
 *                                                   |
 *                       l2arc write hand <--[oooo]--'
 *                               |           8 Mbyte
 *                               |          write max
 *                               V
 *                +==============================+
 *      L2ARC dev |####|#|###|###|    |####| ... |
 *                +==============================+
 *                           32 Gbytes
 *
 * 3. If an ARC buffer is copied to the L2ARC but then hit instead of
 * evicted, then the L2ARC has cached a buffer much sooner than it probably
 * needed to, potentially wasting L2ARC device bandwidth and storage.  It is
 * safe to say that this is an uncommon case, since buffers at the end of
 * the ARC lists have moved there due to inactivity.
 *
 * 4. If the ARC evicts faster than the L2ARC can maintain a headroom,
 * then the L2ARC simply misses copying some buffers.  This serves as a
 * pressure valve to prevent heavy read workloads from both stalling the ARC
 * with waits and clogging the L2ARC with writes.  This also helps prevent
 * the potential for the L2ARC to churn if it attempts to cache content too
 * quickly, such as during backups of the entire pool.
 *
 * 5. After system boot and before the ARC has filled main memory, there are
 * no evictions from the ARC and so the tails of the ARC_mfu and ARC_mru
 * lists can remain mostly static.  Instead of searching from tail of these
 * lists as pictured, the l2arc_feed_thread() will search from the list heads
 * for eligible buffers, greatly increasing its chance of finding them.
 *
 * The L2ARC device write speed is also boosted during this time so that
 * the L2ARC warms up faster.  Since there have been no ARC evictions yet,
 * there are no L2ARC reads, and no fear of degrading read performance
 * through increased writes.
 *
 * 6. Writes to the L2ARC devices are grouped and sent in-sequence, so that
 * the vdev queue can aggregate them into larger and fewer writes.  Each
 * device is written to in a rotor fashion, sweeping writes through
 * available space then repeating.
 *
 * 7. The L2ARC does not store dirty content.  It never needs to flush
 * write buffers back to disk based storage.
 *
 * 8. If an ARC buffer is written (and dirtied) which also exists in the
 * L2ARC, the now stale L2ARC buffer is immediately dropped.
 *
 * The performance of the L2ARC can be tweaked by a number of tunables, which
 * may be necessary for different workloads:
 *
 *      l2arc_write_max         max write bytes per interval
 *      l2arc_write_boost       extra write bytes during device warmup
 *      l2arc_noprefetch        skip caching prefetched buffers
 *      l2arc_headroom          number of max device writes to precache
 *      l2arc_headroom_boost    when we find compressed buffers during ARC
 *                              scanning, we multiply headroom by this
 *                              percentage factor for the next scan cycle,
 *                              since more compressed buffers are likely to
 *                              be present
 *      l2arc_feed_secs         seconds between L2ARC writing
 *
 * Tunables may be removed or added as future performance improvements are
 * integrated, and also may become zpool properties.
 *
 * There are three key functions that control how the L2ARC warms up:
 *
 *      l2arc_write_eligible()  check if a buffer is eligible to cache
 *      l2arc_write_size()      calculate how much to write
 *      l2arc_write_interval()  calculate sleep delay between writes
 *
 * These three functions determine what to write, how much, and how quickly
 * to send writes.
 */

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

        return (B_TRUE);
}

static uint64_t
l2arc_write_size(void)
{
        uint64_t size;

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

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

        return (size);

}

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

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

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

        return (next);
}

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

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

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

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

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

        first = NULL;
        next = l2arc_dev_last;
        do {
                /* loop around the list looking for a non-faulted vdev */
                if (next == NULL) {
                        next = list_head(l2arc_dev_list);
                } else {
                        next = list_next(l2arc_dev_list, next);
                        if (next == NULL)
                                next = list_head(l2arc_dev_list);
                }

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

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

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

        l2arc_dev_last = next;

out:
        mutex_exit(&l2arc_dev_mtx);

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

        return (next);
}

/*
 * Free buffers that were tagged for destruction.
 */
static void
l2arc_do_free_on_write()
{
        list_t *buflist;
        l2arc_data_free_t *df, *df_prev;

        mutex_enter(&l2arc_free_on_write_mtx);
        buflist = l2arc_free_on_write;

        for (df = list_tail(buflist); df; df = df_prev) {
                df_prev = list_prev(buflist, df);
                ASSERT(df->l2df_data != NULL);
                ASSERT(df->l2df_func != NULL);
                df->l2df_func(df->l2df_data, df->l2df_size);
                list_remove(buflist, df);
                kmem_free(df, sizeof (l2arc_data_free_t));
        }

        mutex_exit(&l2arc_free_on_write_mtx);
}

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

```
static void
l2arc_write_done(zio_t *zio)
{
        l2arc_write_callback_t *cb;
        l2arc_dev_t *dev;
        list_t *buflist;
        arc_buf_hdr_t *head, *ab, *ab_prev;
        l2arc_buf_hdr_t *abl2;
        kmutex_t *hash_lock;
        int64_t bytes_dropped = 0;

        cb = zio->io_private;
        ASSERT(cb != NULL);
        dev = cb->l2wcb_dev;
        ASSERT(dev != NULL);
        head = cb->l2wcb_head;
        ASSERT(head != NULL);
        buflist = dev->l2ad_buflist;
        ASSERT(buflist != NULL);
        DTRACE_PROBE2(l2arc__iodone, zio_t *, zio,
            l2arc_write_callback_t *, cb);

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

        mutex_enter(&l2arc_buflist_mtx);

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

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

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

                if (zio->io_error != 0) {
                        /*
                         * Error - drop L2ARC entry.
                         */
                        list_remove(buflist, ab);
                        ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
                        bytes_dropped += abl2->b_asize;
                        ab->b_l2hdr = NULL;
                        kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
                        ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
                }

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

                mutex_exit(hash_lock);
        }

        atomic_inc_64(&l2arc_writes_done);
        list_remove(buflist, head);
        kmem_cache_free(hdr_cache, head);
        mutex_exit(&l2arc_buflist_mtx);

        vdev_space_update(dev->l2ad_vdev, -bytes_dropped, 0, 0);

        l2arc_do_free_on_write();

        kmem_free(cb, sizeof (l2arc_write_callback_t));
}

/*
 * A read to a cache device completed.  Validate buffer contents before
 * handing over to the regular ARC routines.
 */
static void
l2arc_read_done(zio_t *zio)
{
        l2arc_read_callback_t *cb;
        arc_buf_hdr_t *hdr;
        arc_buf_t *buf;
        kmutex_t *hash_lock;
        int equal;

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

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

        cb = zio->io_private;
        ASSERT(cb != NULL);
        buf = cb->l2rcb_buf;
        ASSERT(buf != NULL);

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

        /*
         * If the buffer was compressed, decompress it first.
         */
        if (cb->l2rcb_compress != ZIO_COMPRESS_OFF)
                l2arc_decompress_zio(zio, hdr, cb->l2rcb_compress);
        ASSERT(zio->io_data != NULL);

        /*
         * Check this survived the L2ARC journey.
         */
        equal = arc_cksum_equal(buf);
        if (equal && zio->io_error == 0 && !HDR_L2_EVICTED(hdr)) {
                mutex_exit(hash_lock);
                zio->io_private = buf;
                zio->io_bp_copy = cb->l2rcb_bp; /* XXX fix in L2ARC 2.0 */
                zio->io_bp = &zio->io_bp_copy;  /* XXX fix in L2ARC 2.0 */
                arc_read_done(zio);
        } else {
                mutex_exit(hash_lock);
                /*
                 * Buffer didn’t survive caching.  Increment stats and
```

```
                 * reissue to the original storage device.
                 */
                if (zio->io_error != 0) {
                        ARCSTAT_BUMP(arcstat_l2_io_error);
                } else {
                        zio->io_error = SET_ERROR(EIO);
                }
                if (!equal)
                        ARCSTAT_BUMP(arcstat_l2_cksum_bad);

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

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

                        zio_nowait(zio_read(pio, cb->l2rcb_spa, &cb->l2rcb_bp,
                            buf->b_data, zio->io_size, arc_read_done, buf,
                            zio->io_priority, cb->l2rcb_flags, &cb->l2rcb_zb));
                }
        }

        kmem_free(cb, sizeof (l2arc_read_callback_t));
}

/*
 * This is the list priority from which the L2ARC will search for pages to
 * cache.  This is used within loops (0..3) to cycle through lists in the
 * desired order.  This order can have a significant effect on cache
 * performance.
 *
 * Currently the metadata lists are hit first, MFU then MRU, followed by
 * the data lists.  This function returns a locked list, and also returns
 * the lock pointer.
 */
static list_t *
l2arc_list_locked(int list_num, kmutex_t **lock)
{
        list_t *list = NULL;

        ASSERT(list_num >= 0 && list_num <= 3);

        switch (list_num) {
        case 0:
                list = &arc_mfu->arcs_list[ARC_BUFC_METADATA];
                *lock = &arc_mfu->arcs_mtx;
                break;
        case 1:
                list = &arc_mru->arcs_list[ARC_BUFC_METADATA];
                *lock = &arc_mru->arcs_mtx;
                break;
        case 2:
                list = &arc_mfu->arcs_list[ARC_BUFC_DATA];
                *lock = &arc_mfu->arcs_mtx;
                break;
        case 3:
                list = &arc_mru->arcs_list[ARC_BUFC_DATA];
                *lock = &arc_mru->arcs_mtx;
                break;
        }

        ASSERT(!(MUTEX_HELD(*lock)));
        mutex_enter(*lock);
        return (list);
}

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

        buflist = dev->l2ad_buflist;

        if (buflist == NULL)
                return;

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

        if (dev->l2ad_hand >= (dev->l2ad_end - (2 * distance))) {
                /*
                 * When nearing the end of the device, evict to the end
                 * before the device write hand jumps to the start.
                 */
                taddr = dev->l2ad_end;
        } else {
                taddr = dev->l2ad_hand + distance;
        }
        DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist,
            uint64_t, taddr, boolean_t, all);

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

                hash_lock = HDR_LOCK(ab);
                if (!mutex_tryenter(hash_lock)) {
                        /*
                         * Missed the hash lock.  Retry.
                         */
                        ARCSTAT_BUMP(arcstat_l2_evict_lock_retry);
                        mutex_exit(&l2arc_buflist_mtx);
                        mutex_enter(hash_lock);
                        mutex_exit(hash_lock);
                        goto top;
                }

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

```
                         */
                        list_remove(buflist, ab);
                        mutex_exit(hash_lock);
                        continue;
                }

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

                if (HDR_FREE_IN_PROGRESS(ab)) {
                        /*
                         * Already on the path to destruction.
                         */
                        mutex_exit(hash_lock);
                        continue;
                }

                if (ab->b_state == arc_l2c_only) {
                        ASSERT(!HDR_L2_READING(ab));
                        /*
                         * This doesn't exist in the ARC.  Destroy.
                         * arc_hdr_destroy() will call list_remove()
                         * and decrement arcstat_l2_size.
                         */
                        arc_change_state(arc_anon, ab, hash_lock);
                        arc_hdr_destroy(ab);
                } else {
                        /*
                         * Invalidate issued or about to be issued
                         * reads, since we may be about to write
                         * over this location.
                         */
                        if (HDR_L2_READING(ab)) {
                                ARCSTAT_BUMP(arcstat_l2_evict_reading);
                                ab->b_flags |= ARC_L2_EVICTED;
                        }

                        /*
                         * Tell ARC this no longer exists in L2ARC.
                         */
                        if (ab->b_l2hdr != NULL) {
                                abl2 = ab->b_l2hdr;
                                ARCSTAT_INCR(arcstat_l2_asize, -abl2->b_asize);
                                bytes_evicted += abl2->b_asize;
                                ab->b_l2hdr = NULL;
                                /*
                                 * We are destroying l2hdr, so ensure that
                                 * its compressed buffer, if any, is not leaked.
                                 */
                                ASSERT(abl2->b_tmp_cdata == NULL);
#endif /* ! codereview */
                                kmem_free(abl2, sizeof (l2arc_buf_hdr_t));
                                ARCSTAT_INCR(arcstat_l2_size, -ab->b_size);
                        }
                        list_remove(buflist, ab);

                        /*
                         * This may have been leftover after a
                         * failed write.
                         */
                        ab->b_flags &= ~ARC_L2_WRITING;
                }
                mutex_exit(hash_lock);
        }
        mutex_exit(&l2arc_buflist_mtx);

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

/*
 * Find and write ARC buffers to the L2ARC device.
 *
 * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
 * for reading until they have completed writing.
 * The headroom_boost is an in-out parameter used to maintain headroom boost
 * state between calls to this function.
 *
 * Returns the number of bytes actually written (which may be smaller than
 * the delta by which the device hand has changed due to alignment).
 */
static uint64_t
l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz,
    boolean_t *headroom_boost)
{
        arc_buf_hdr_t *ab, *ab_prev, *head;
        list_t *list;
        uint64_t write_asize, write_psize, write_sz, headroom,
            buf_compress_minsz;
        void *buf_data;
        kmutex_t *list_lock;
        boolean_t full;
        l2arc_write_callback_t *cb;
        zio_t *pio, *wzio;
        uint64_t guid = spa_load_guid(spa);
        const boolean_t do_headroom_boost = *headroom_boost;

        ASSERT(dev->l2ad_vdev != NULL);

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

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

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

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

                list = l2arc_list_locked(try, &list_lock);

                /*
```

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

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

                for (; ab; ab = ab_prev) {
                        l2arc_buf_hdr_t *l2hdr;
                        kmutex_t *hash_lock;
                        uint64_t buf_sz;

                        if (arc_warm == B_FALSE)
                                ab_prev = list_next(list, ab);
                        else
                                ab_prev = list_prev(list, ab);

                        hash_lock = HDR_LOCK(ab);
                        if (!mutex_tryenter(hash_lock)) {
                                /*
                                 * Skip this buffer rather than waiting.
                                 */
                                continue;
                        }

                        passed_sz += ab->b_size;
                        if (passed_sz > headroom) {
                                /*
                                 * Searched too far.
                                 */
                                mutex_exit(hash_lock);
                                break;
                        }

                        if (!l2arc_write_eligible(guid, ab)) {
                                mutex_exit(hash_lock);
                                continue;
                        }

                        if ((write_sz + ab->b_size) > target_sz) {
                                full = B_TRUE;
                                mutex_exit(hash_lock);
                                break;
                        }

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

                                cb = kmem_alloc(
                                    sizeof (l2arc_write_callback_t), KM_SLEEP);
                                cb->l2wcb_dev = dev;
                                cb->l2wcb_head = head;
                                pio = zio_root(spa, l2arc_write_done, cb,
                                    ZIO_FLAG_CANFAIL);
                        }

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

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

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

                        list_insert_head(dev->l2ad_buflist, ab);

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

                        mutex_exit(hash_lock);

                        write_sz += buf_sz;
                }

                mutex_exit(list_lock);

                if (full == B_TRUE)
                        break;
        }

        /* No buffers selected for writing? */
        if (pio == NULL) {
                ASSERT0(write_sz);
                mutex_exit(&l2arc_buflist_mtx);
                kmem_cache_free(hdr_cache, head);
                return (0);
        }

        /*
         * Now start writing the buffers. We’re starting at the write head
         * and work backwards, retracing the course of the buffer selector
         * loop above.
         */
        for (ab = list_prev(dev->l2ad_buflist, head); ab;
            ab = list_prev(dev->l2ad_buflist, ab)) {
                l2arc_buf_hdr_t *l2hdr;
                uint64_t buf_sz;

                /*
                 * We shouldn’t need to lock the buffer here, since we flagged
                 * it as ARC_L2_WRITING in the previous step, but we must take
                 * care to only access its L2 cache parameters. In particular,
```

```
                 * ab->b_buf may be invalid by now due to ARC eviction.
                 */
                l2hdr = ab->b_l2hdr;
                l2hdr->b_daddr = dev->l2ad_hand;

                if ((ab->b_flags & ARC_L2COMPRESS) &&
                    l2hdr->b_asize >= buf_compress_minsz) {
                        if (l2arc_compress_buf(l2hdr)) {
                                /*
                                 * If compression succeeded, enable headroom
                                 * boost on the next scan cycle.
                                 */
                                *headroom_boost = B_TRUE;
                        }
                }

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

                /*
                 * If the data has not been compressed, then clear b_tmp_cdata
                 * to make sure that it points only to a temporary compression
                 * buffer.
                 */
                if (!L2ARC_IS_VALID_COMPRESS(l2hdr->b_compress))
                        l2hdr->b_tmp_cdata = NULL;

#endif /* ! codereview */
                /* Compression may have squashed the buffer to zero length. */
                if (buf_sz != 0) {
                        uint64_t buf_p_sz;

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

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

                        write_asize += buf_sz;
                        /*
                         * Keep the clock hand suitably device-aligned.
                         */
                        buf_p_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
                        write_psize += buf_p_sz;
                        dev->l2ad_hand += buf_p_sz;
                }
        }

        mutex_exit(&l2arc_buflist_mtx);

        ASSERT3U(write_asize, <=, target_sz);
        ARCSTAT_BUMP(arcstat_l2_writes_sent);
        ARCSTAT_INCR(arcstat_l2_write_bytes, write_asize);
        ARCSTAT_INCR(arcstat_l2_size, write_sz);
        ARCSTAT_INCR(arcstat_l2_asize, write_asize);
        vdev_space_update(dev->l2ad_vdev, write_asize, 0, 0);

        /*
         * Bump device hand to the device start if it is approaching the end.
         * l2arc_evict() will already have evicted ahead for this case.
         */
        if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
                dev->l2ad_hand = dev->l2ad_start;
                dev->l2ad_evict = dev->l2ad_start;
                dev->l2ad_first = B_FALSE;
        }

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

        return (write_asize);
}

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

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

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

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

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

```
                 */
                l2hdr->b_compress = ZIO_COMPRESS_LZ4;
                l2hdr->b_asize = csize;
                l2hdr->b_tmp_cdata = cdata;
                ARCSTAT_BUMP(arcstat_l2_compress_successes);
                return (B_TRUE);
        } else {
                /*
                 * Compression failed, release the compressed buffer.
                 * l2hdr will be left unmodified.
                 */
                zio_data_buf_free(cdata, len);
                ARCSTAT_BUMP(arcstat_l2_compress_failures);
                return (B_FALSE);
        }
}

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

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

        if (c == ZIO_COMPRESS_EMPTY) {
                /*
                 * An empty buffer results in a null zio, which means we
                 * need to fill its io_data after we're done restoring the
                 * buffer's contents.
                 */
                ASSERT(hdr->b_buf != NULL);
                bzero(hdr->b_buf->b_data, hdr->b_size);
                zio->io_data = zio->io_orig_data = hdr->b_buf->b_data;
        } else {
                ASSERT(zio->io_data != NULL);
                /*
                 * We copy the compressed data from the start of the arc buffer
                 * (the zio_read will have pulled in only what we need, the
                 * rest is garbage which we will overwrite at decompression)
                 * and then decompress back to the ARC data buffer. This way we
                 * can minimize copying by simply decompressing back over the
                 * original compressed data (rather than decompressing to an
                 * aux buffer and then copying back the uncompressed buffer,
                 * which is likely to be much larger).
                 */
                uint64_t csize;
                void *cdata;
                csize = zio->io_size;
                cdata = zio_data_buf_alloc(csize);
                bcopy(zio->io_data, cdata, csize);
                if (zio_decompress_data(c, cdata, zio->io_data, csize,
                    hdr->b_size) != 0)
                        zio->io_error = EIO;
                zio_data_buf_free(cdata, csize);
        }

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

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

        ASSERT(L2ARC_IS_VALID_COMPRESS(l2hdr->b_compress));
        if (l2hdr->b_compress != ZIO_COMPRESS_EMPTY) {
        if (l2hdr->b_compress == ZIO_COMPRESS_LZ4) {
                /*
                 * If the data was compressed, then we’ve allocated a
                 * temporary buffer for it, so now we need to release it.
                 */
                ASSERT(l2hdr->b_tmp_cdata != NULL);
                zio_data_buf_free(l2hdr->b_tmp_cdata, ab->b_size);
                l2hdr->b_tmp_cdata = NULL;
        } else {
                ASSERT(l2hdr->b_tmp_cdata == NULL);
#endif /* ! codereview */
        }
        l2hdr->b_tmp_cdata = NULL;
}
______unchanged_portion_omitted_
```