```
**********************************************************
  149962 Thu Oct 17 00:34:49 2013
new/usr/src/uts/common/fs/zfs/arc.c
New ARC buf_hash architecture
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

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



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

/*
 * External users typically access ARC buffers via a hash table
 * lookup, using the DVA, spa_t pointer value and the birth TXG
 * number as the key. The hash value is derived by buf_hash(),
 * which spits out a 64-bit hash index. This index is then masked
 * with ht_mask to obtain the final index into the hash table:
 *
 *                     ,---------------- & ht_mask ----------------,
 * 64-bit hash value   |             (hash table index)            |
 * |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX|
 *
 * Sizing of the hash table is done at boot from the amount of
 * physical memory. We start with a base value of 2^12 hash
 * buckets and then evaluate whether this number, multiplied by
 * 2^zfs_arc_ht_base_masklen (the minimum mask length), is
 * greater than or equal to the amount of physical memory. If not,
 * we double the number of hash buckets and repeat. Using the
 * default settings these values translate to ~1 MB of hash tables
 * for each 1 GB of physical memory.
 *
 * The locking model:
 *
 * A new reference to a cache buffer can be obtained in two
 * ways: 1) via a hash table lookup using the DVA as a key,
 * or 2) via one of the ARC lists.  The arc_read() interface
 * uses method 1, while the internal arc algorithms for
 * adjusting the cache use method 2.  We therefore provide two
 * types of locks: 1) the hash table lock array, and 2) the
 * arc list locks.
 *
 * Buffers do not have their own mutexes, rather they rely on the
 * hash table mutexes for the bulk of their protection (i.e. most
 * fields in the arc_buf_hdr_t are protected by these mutexes). The
 * specific mutex is selected by taking its hash value and masking
 * it by ht_lock_mask, which then produces an index into the mutex
 * table. The size of the lock table is derived from the amount of
 * physical memory, which is simply divided by
 * 2^zfs_arc_ht_lock_shift, giving the number of locks, with a
 * minimum of MIN_BUF_LOCKS.
 * fields in the arc_buf_hdr_t are protected by these mutexes).
 *
 * buf_hash_find() returns the appropriate mutex (held) when it
 * locates the requested buffer in the hash table.  It returns
 * NULL for the mutex if the buffer was not in the table.
 *
 * buf_hash_remove() expects the appropriate hash mutex to be
 * already held before it is invoked.
 *
 * Each arc state also has a mutex which is used to protect the
 * buffer list associated with the state.  When attempting to
 * obtain a hash table lock while holding an arc list lock you
 * must use: mutex_tryenter() to avoid deadlock.  Also note that
 * the active state mutex must be held before the ghost state mutex.
 *
```

```
 * Arc buffers may have an associated eviction callback function.
 * This function will be invoked prior to removing the buffer (e.g.
 * in arc_do_user_evicts()).  Note however that the data associated
 * with the buffer may be evicted prior to the callback.  The callback
 * must be made with *no locks held* (to prevent deadlock).  Additionally,
 * the users of callbacks must ensure that their private data is
 * protected from simultaneous callbacks from arc_buf_evict()
 * and arc_do_user_evicts().
 *
 * Note that the majority of the performance stats are manipulated
 * with atomic operations.
 *
 * The L2ARC uses the l2arc_buflist_mtx global mutex for the following:
 *
 *      - L2ARC buflist creation
 *      - L2ARC buflist eviction
 *      - L2ARC write completion, which walks L2ARC buflists
 *      - ARC header destruction, as it removes from L2ARC buflists
 *      - ARC header release, as it removes from L2ARC buflists
 */

#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/zio_compress.h>
#include <sys/zfs_context.h>
#include <sys/arc.h>
#include <sys/refcount.h>
#include <sys/vdev.h>
#include <sys/vdev_impl.h>
#include <sys/dsl_pool.h>
#ifdef _KERNEL
#include <sys/vmsystm.h>
#include <vm/anon.h>
#include <sys/fs/swapnode.h>
#include <sys/dnlc.h>
#endif
#include <sys/callb.h>
#include <sys/kstat.h>
#include <zfs_fletcher.h>

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

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

#define ARC_REDUCE_DNLC_PERCENT 3
uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

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

/*
 * The number of iterations through arc_evict_*() before we
 * drop & reacquire the lock.
 */
int arc_evict_iterations = 100;

/* number of seconds before growing cache again */
static int              arc_grow_retry = 60;
```



```
/* shift of arc_c for calculating both min and max arc_p */
static int              arc_p_min_shift = 4;

/* log2(fraction of arc to reclaim) */
static int              arc_shrink_shift = 5;

/*
 * minimum lifespan of a prefetch block in clock ticks
 * (initialized in arc_init())
 */
static int              arc_min_prefetch_lifespan;

/*
 * If this percent of memory is free, don’t throttle.
 */
int arc_lotsfree_percent = 10;

static int arc_dead;

/*
 * The arc has filled available memory and has now warmed up.
 */
static boolean_t arc_warm;

/*
 * These tunables are for performance analysis.
 */
uint64_t zfs_arc_max;
uint64_t zfs_arc_min;
uint64_t zfs_arc_meta_limit = 0;
int zfs_arc_grow_retry = 0;
int zfs_arc_shrink_shift = 0;
int zfs_arc_p_min_shift = 0;
int zfs_disable_dup_eviction = 0;

/*
 * Used to calculate the size of ARC hash tables and number of hash locks.
 * See big theory block comment at the start of this file.
 */
uint64_t zfs_arc_ht_base_masklen = 13;
/*
 * We want to allocate one hash lock for every 4GB of memory with a minimum
 * of MIN_BUF_LOCKS.
 */
uint64_t zfs_arc_ht_lock_shift = 32;
#define MIN_BUF_LOCKS   256

/*
 * Note that buffers can be in one of 6 states:
 *      ARC_anon        - anonymous (discussed below)
 *      ARC_mru         - recently used, currently cached
 *      ARC_mru_ghost   - recentely used, no longer in cache
 *      ARC_mfu         - frequently used, currently cached
 *      ARC_mfu_ghost   - frequently used, no longer in cache
 *      ARC_l2c_only    - exists in L2ARC but not other states
 * When there are no active references to the buffer, they are
 * are linked onto a list in one of these arc states.  These are
 * the only buffers that can be evicted or deleted.  Within each
 * state there are multiple lists, one for meta-data and one for
 * non-meta-data.  Meta-data (indirect blocks, blocks of dnodes,
 * etc.) is tracked separately so that it can be managed more
 * explicitly: favored over data, limited explicitly.
 *
 * Anonymous buffers are buffers that are not associated with
 * a DVA.  These are buffers that hold dirty block copies
```

```
 * before they are written to stable storage.  By definition,
 * they are "ref'd" and are considered part of arc_mru
 * that cannot be freed.  Generally, they will aquire a DVA
 * as they are written and migrate onto the arc_mru list.
 *
 * The ARC_l2c_only state is for buffers that are in the second
 * level ARC but no longer in any of the ARC_m* lists.  The second
 * level ARC itself may also contain buffers that are in any of
 * the ARC_m* states - meaning that a buffer can exist in two
 * places.  The reason for the ARC_l2c_only state is to keep the
 * buffer header in the hash table, so that reads that hit the
 * second level ARC benefit from these fast lookups.
 */

typedef struct arc_state {
        list_t  arcs_list[ARC_BUFC_NUMTYPES];   /* list of evictable buffers */
        uint64_t arcs_lsize[ARC_BUFC_NUMTYPES]; /* amount of evictable data */
        uint64_t arcs_size;     /* total amount of data in this state */
        kmutex_t arcs_mtx;
} arc_state_t;
```

______*unchanged_portion_omitted_*

```
#define BUF_LOCKS 256
typedef struct buf_hash_table {
        uint64_t        ht_mask;
        arc_buf_hdr_t   **ht_table;
        struct ht_lock  *ht_locks;
        uint64_t        ht_num_locks, ht_lock_mask;
        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_hash_table.ht_lock_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)

/* 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;
```

______*unchanged_portion_omitted_*

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

        ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);

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

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

        return (crc);
}
```

______*unchanged_portion_omitted_*

```
/*
 * 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_hash_table.ht_num_locks; i++)
        for (i = 0; i < BUF_LOCKS; i++)
                mutex_destroy(&buf_hash_table.ht_locks[i].ht_lock);
        kmem_free(buf_hash_table.ht_locks, sizeof (struct ht_lock) *
            buf_hash_table.ht_num_locks);
        kmem_cache_destroy(hdr_cache);
        kmem_cache_destroy(buf_cache);
}
______unchanged_portion_omitted_

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

        while ((1ULL << (ht_masklen + zfs_arc_ht_base_masklen)) <
            physmem * PAGESIZE)
                ht_masklen++;
        buf_hash_table.ht_mask = (1ULL << ht_masklen) - 1;
        /*
         * The hash table is big enough to fill all of physical memory
         * with an average 64K block size.  The table will take up
         * totalmem*sizeof(void*)/64K (eg. 128KB/GB with 8-byte pointers).
         */
        while (hsize * 65536 < physmem * PAGESIZE)
                hsize <<= 1;
retry:
        buf_hash_table.ht_mask = hsize - 1;
        buf_hash_table.ht_table =
            kmem_zalloc((1ULL << ht_masklen) * sizeof (void *), KM_SLEEP);

        buf_hash_table.ht_num_locks = MAX((physmem * PAGESIZE) >>
            zfs_arc_ht_lock_shift, MIN_BUF_LOCKS);
        buf_hash_table.ht_lock_mask = buf_hash_table.ht_num_locks - 1;
        buf_hash_table.ht_locks = kmem_zalloc(sizeof (struct ht_lock) *
            buf_hash_table.ht_num_locks, KM_SLEEP);
        for (i = 0; i < buf_hash_table.ht_num_locks; i++) {
                mutex_init(&buf_hash_table.ht_locks[i].ht_lock,
                    NULL, MUTEX_DEFAULT, NULL);
            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);
        }
}
______unchanged_portion_omitted_
```