```
**********************************************************
  131644 Mon Jun  4 22:08:28 2012
new/usr/src/uts/common/fs/zfs/arc.c
*** NO COMMENTS ***
**********************************************************
/*
 * 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 by Delphix. 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 therefor 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 therefor choose a set of blocks to evict to make
 * space for a cache miss that approximates as closely as possible
 * the space used by the new block.
 *
 * See also:  "ARC: A Self-Tuning, Low Overhead Replacement Cache"
 * by N. Megiddo & D. Modha, FAST 2003
 */

/*
 * The locking model:
 *
 * A new reference to a cache buffer can be obtained in two
 * ways: 1) via a hash table lookup using the DVA as a key,
 * or 2) via one of the ARC lists.  The arc_read() interface
 * uses method 1, while the internal arc algorithms for
 * adjusting the cache use method 2.  We therefor 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).
 * Buffers do not have their own mutexs, rather they rely on the
 * hash table mutexs for the bulk of their protection (i.e. most
 * fields in the arc_buf_hdr_t are protected by these mutexs).
 *
 * 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/zfs_context.h>
#include <sys/arc.h>
```

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

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

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

#define ARC_REDUCE_DNLC_PERCENT 3
uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

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

______unchanged_portion_omitted_

```
/*
 * "Read" the block at the specified DVA (in bp) via the
 * "Read" the block 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.
 *
 * Normal callers should use arc_read and pass the arc buffer and offset
 * for the bp.  But if you know you don’t need locking, you can use
 * arc_read_bp.
 */
int
arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_buf_t *pbuf,
    arc_done_func_t *done, void *private, int priority, int zio_flags,
    uint32_t *arc_flags, const zbookmark_t *zb)
{
        int err;

        if (pbuf == NULL) {
                /*
                 * XXX This happens from traverse callback funcs, for
                 * the objset_phys_t block.
                 */
                return (arc_read_nolock(pio, spa, bp, done, private, priority,
```

```
                    zio_flags, arc_flags, zb));
        }

        ASSERT(!refcount_is_zero(&pbuf->b_hdr->b_refcnt));
        ASSERT3U((char *)bp - (char *)pbuf->b_data, <, pbuf->b_hdr->b_size);
        rw_enter(&pbuf->b_data_lock, RW_READER);

        err = arc_read_nolock(pio, spa, bp, done, private, priority,
            zio_flags, arc_flags, zb);
        rw_exit(&pbuf->b_data_lock);

        return (err);
}
```

______unchanged_portion_omitted_