```
**********************************************************
  134460 Wed May 29 14:09:48 2013
new/usr/src/uts/common/fs/zfs/arc.c
re #13729 assign each ARC hash bucket its own mutex
In ARC the number of buckets in buffer header hash table is
proportional to the size of physical RAM.
The number of locks protecting headers in the buckets is fixed to 256 though.
Hence, on systems with large memory (>= 128GB) too many unrelated buffer
headers are protected by the same mutex.
When the memory in the system is fragmented this may cause a deadlock:
- An arc_read thread may be trying to allocate a 128k buffer while holding
a header lock.
- The allocation uses KM_PUSHPAGE option that blocks the thread if no contigous
chunk of requested size is available.
- ARC eviction thread that is supposed to evict some buffers would call
an evict callback on one of the buffers.
- Before freing the memory, the callback will attempt to take a lock on buffer
header.
- Incidentally, this buffer header will be protected by the same lock as
he one in arc_read() thread.
The solution in this patch is not perfect - that is, it protects all headers
in the hash bucket by the same lock.
However, a probability of collision is very low and does not depend on memory
size.
By the same argument, padding locks to cacheline looks like a waste of memory
here since the probability of contention on a cacheline is quite low, given
he number of buckets, number of locks per cacheline (4) and the fact that
he hash function (crc64 % hash table size) is supposed to be a very good
randomizer.
This effect on memory usage is as follows:
Per hash table size n,
- Original code uses 16K + 16 + n * 8 bytes of memory
- This fix uses 2 * n * 8 + 8 bytes of memory
- The net memory overhead is therefore n * 8 - 16K - 8 bytes
The value of n grows proportionally to physical memory size.
For 128GB of physical memory it is 2M, so the memory overhead is
16M - 16K - 8 bytes.
For smaller memory configurations the overhead is proportionally smaller, and
for larger memory configurations it is propottionally bigger.
The patch has been tested for 30+ hours using vdbench script that reproduces
hang with original code 100% of times in 20-30 minutes.
**********************************************************
______unchanged_portion_omitted_

 489 static arc_buf_t *arc_eviction_list;
 490 static kmutex_t arc_eviction_mtx;
 491 static arc_buf_hdr_t arc_eviction_hdr;
 492 static void arc_get_data_buf(arc_buf_t *buf);
 493 static void arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock);
 494 static int arc_evict_needed(arc_buf_contents_t type);
 495 static void arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes);
 496 static void arc_buf_watch(arc_buf_t *buf);

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

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

 504 /*
 505  * Private ARC flags.  These flags are private ARC only flags that will show up
 506  * in b_flags in the arc_hdr_buf_t.  Some flags are publicly declared, and can
 507  * be passed in as arc_flags in things like arc_read.  However, these flags
 508  * should never be passed and should only be set by ARC code.  When adding new
 509  * public flags, make sure not to smash the private ones.
 510  */
```



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

 523 #define HDR_IN_HASH_TABLE(hdr)  ((hdr)->b_flags & ARC_IN_HASH_TABLE)
 524 #define HDR_IO_IN_PROGRESS(hdr) ((hdr)->b_flags & ARC_IO_IN_PROGRESS)
 525 #define HDR_IO_ERROR(hdr)       ((hdr)->b_flags & ARC_IO_ERROR)
 526 #define HDR_PREFETCH(hdr)       ((hdr)->b_flags & ARC_PREFETCH)
 527 #define HDR_FREED_IN_READ(hdr)  ((hdr)->b_flags & ARC_FREED_IN_READ)
 528 #define HDR_BUF_AVAILABLE(hdr)  ((hdr)->b_flags & ARC_BUF_AVAILABLE)
 529 #define HDR_FREE_IN_PROGRESS(hdr)       ((hdr)->b_flags & ARC_FREE_IN_PROGRESS)
 530 #define HDR_L2CACHE(hdr)        ((hdr)->b_flags & ARC_L2CACHE)
 531 #define HDR_L2_READING(hdr)     ((hdr)->b_flags & ARC_IO_IN_PROGRESS && \
 532                                     (hdr)->b_l2hdr != NULL)
 533 #define HDR_L2_WRITING(hdr)     ((hdr)->b_flags & ARC_L2_WRITING)
 534 #define HDR_L2_EVICTED(hdr)     ((hdr)->b_flags & ARC_L2_EVICTED)
 535 #define HDR_L2_WRITE_HEAD(hdr)  ((hdr)->b_flags & ARC_L2_WRITE_HEAD)

 537 /*
 538  * Other sizes
 539  */

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

 544 /*
 545  * Hash table routines
 546  */

 548 struct ht_table {
 549         arc_buf_hdr_t   *hdr;
 550         kmutex_t        lock;
 548 #define HT_LOCK_PAD     64

 550 struct ht_lock {
 551         kmutex_t        ht_lock;
 552 #ifdef _KERNEL
 553         unsigned char   pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
 554 #endif
 551 };

 557 #define BUF_LOCKS 256
 553 typedef struct buf_hash_table {
 554         uint64_t ht_mask;
 555         struct ht_table *ht_table;
 560         arc_buf_hdr_t **ht_table;
 561         struct ht_lock ht_locks[BUF_LOCKS];
 556 } buf_hash_table_t;

 558 static buf_hash_table_t buf_hash_table;

 560 #define BUF_HASH_INDEX(spa, dva, birth) \
 561         (buf_hash(spa, dva, birth) & buf_hash_table.ht_mask)
 562 #define BUF_HASH_LOCK(idx) (&buf_hash_table.ht_table[idx].lock)
 568 #define BUF_HASH_LOCK_NTRY(idx) (buf_hash_table.ht_locks[idx & (BUF_LOCKS-1)])
 569 #define BUF_HASH_LOCK(idx)      (&(BUF_HASH_LOCK_NTRY(idx).ht_lock))
 563 #define HDR_LOCK(hdr) \
 564         (BUF_HASH_LOCK(BUF_HASH_INDEX(hdr->b_spa, &hdr->b_dva, hdr->b_birth)))
```

```
566 uint64_t zfs_crc64_table[256];

568 /*
569  * Level 2 ARC
570  */

572 #define L2ARC_WRITE_SIZE        (8 * 1024 * 1024)       /* initial write max */
573 #define L2ARC_HEADROOM          2               /* num of writes */
574 #define L2ARC_FEED_SECS         1               /* caching interval secs */
575 #define L2ARC_FEED_MIN_MS       200             /* min caching interval ms */

577 #define l2arc_writes_sent       ARCSTAT(arcstat_l2_writes_sent)
578 #define l2arc_writes_done       ARCSTAT(arcstat_l2_writes_done)

580 /*
581  * L2ARC Performance Tunables
582  */
583 uint64_t l2arc_write_max = L2ARC_WRITE_SIZE;    /* default max write size */
584 uint64_t l2arc_write_boost = L2ARC_WRITE_SIZE;  /* extra write during warmup */
585 uint64_t l2arc_headroom = L2ARC_HEADROOM;       /* number of dev writes */
586 uint64_t l2arc_feed_secs = L2ARC_FEED_SECS;     /* interval seconds */
587 uint64_t l2arc_feed_min_ms = L2ARC_FEED_MIN_MS; /* min interval milliseconds */
588 boolean_t l2arc_noprefetch = B_TRUE;            /* don't cache prefetch bufs */
589 boolean_t l2arc_feed_again = B_TRUE;            /* turbo warmup */
590 boolean_t l2arc_norw = B_TRUE;                  /* no reads during writes */

592 /*
593  * L2ARC Internals
594  */
595 typedef struct l2arc_dev {
596         vdev_t                  *l2ad_vdev;     /* vdev */
597         spa_t                   *l2ad_spa;      /* spa */
598         uint64_t                l2ad_hand;      /* next write location */
599         uint64_t                l2ad_write;     /* desired write size, bytes */
600         uint64_t                l2ad_boost;     /* warmup write boost, bytes */
601         uint64_t                l2ad_start;     /* first addr on device */
602         uint64_t                l2ad_end;       /* last addr on device */
603         uint64_t                l2ad_evict;     /* last addr eviction reached */
604         boolean_t               l2ad_first;     /* first sweep through */
605         boolean_t               l2ad_writing;   /* currently writing */
606         list_t                  *l2ad_buflist;  /* buffer list */
607         list_node_t             l2ad_node;      /* device list node */
608 } l2arc_dev_t;
```

unchanged_portion_omitted_

```
691 static arc_buf_hdr_t *
692 buf_hash_find(uint64_t spa, const dva_t *dva, uint64_t birth, kmutex_t **lockp)
693 {
694         uint64_t idx = BUF_HASH_INDEX(spa, dva, birth);
695         kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
696         arc_buf_hdr_t *buf;

698         mutex_enter(hash_lock);
699         for (buf = buf_hash_table.ht_table[idx].hdr; buf != NULL;
706         for (buf = buf_hash_table.ht_table[idx]; buf != NULL;
700             buf = buf->b_hash_next) {
701                 if (BUF_EQUAL(spa, dva, birth, buf)) {
702                         *lockp = hash_lock;
703                         return (buf);
704                 }
705         }
706         mutex_exit(hash_lock);
707         *lockp = NULL;
708         return (NULL);
709 }
```



```
711 /*
712  * Insert an entry into the hash table.  If there is already an element
713  * equal to elem in the hash table, then the already existing element
714  * will be returned and the new element will not be inserted.
715  * Otherwise returns NULL.
716  */
717 static arc_buf_hdr_t *
718 buf_hash_insert(arc_buf_hdr_t *buf, kmutex_t **lockp)
719 {
720         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
721         kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
722         arc_buf_hdr_t *fbuf;
723         uint32_t i;

725         ASSERT(!HDR_IN_HASH_TABLE(buf));
726         *lockp = hash_lock;
727         mutex_enter(hash_lock);
728         for (fbuf = buf_hash_table.ht_table[idx].hdr, i = 0; fbuf != NULL;
735         for (fbuf = buf_hash_table.ht_table[idx], i = 0; fbuf != NULL;
729             fbuf = fbuf->b_hash_next, i++) {
730                 if (BUF_EQUAL(buf->b_spa, &buf->b_dva, buf->b_birth, fbuf))
731                         return (fbuf);
732         }

734         buf->b_hash_next = buf_hash_table.ht_table[idx].hdr;
735         buf_hash_table.ht_table[idx].hdr = buf;
741         buf->b_hash_next = buf_hash_table.ht_table[idx];
742         buf_hash_table.ht_table[idx] = buf;
736         buf->b_flags |= ARC_IN_HASH_TABLE;

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

744                 ARCSTAT_MAX(arcstat_hash_chain_max, i);
745         }

747         ARCSTAT_BUMP(arcstat_hash_elements);
748         ARCSTAT_MAXSTAT(arcstat_hash_elements);

750         return (NULL);
751 }

753 static void
754 buf_hash_remove(arc_buf_hdr_t *buf)
755 {
756         arc_buf_hdr_t *fbuf, **bufp;
757         uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);

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

762         bufp = &buf_hash_table.ht_table[idx].hdr;
769         bufp = &buf_hash_table.ht_table[idx];
763         while ((fbuf = *bufp) != buf) {
764                 ASSERT(fbuf != NULL);
765                 bufp = &fbuf->b_hash_next;
766         }
767         *bufp = buf->b_hash_next;
768         buf->b_hash_next = NULL;
769         buf->b_flags &= ~ARC_IN_HASH_TABLE;

771         /* collect some hash table performance data */
```

```
 772         ARCSTAT_BUMPDOWN(arcstat_hash_elements);

 774         if (buf_hash_table.ht_table[idx].hdr &&
 775             buf_hash_table.ht_table[idx].hdr->b_hash_next == NULL)
 781         if (buf_hash_table.ht_table[idx] &&
 782             buf_hash_table.ht_table[idx]->b_hash_next == NULL)
 776                 ARCSTAT_BUMPDOWN(arcstat_hash_chains);
 777 }

 779 /*
 780  * Global data structures and functions for the buf kmem cache.
 781  */
 782 static kmem_cache_t *hdr_cache;
 783 static kmem_cache_t *buf_cache;

 785 static void
 786 buf_fini(void)
 787 {
 788         int i;

 790         for (i = 0; i < buf_hash_table.ht_mask + 1; i++)
 791                 mutex_destroy(&buf_hash_table.ht_table[i].lock);
 792         kmem_free(buf_hash_table.ht_table,
 793             (buf_hash_table.ht_mask + 1) * sizeof (struct ht_table));
 798             (buf_hash_table.ht_mask + 1) * sizeof (void *));
 799         for (i = 0; i < BUF_LOCKS; i++)
 800                 mutex_destroy(&buf_hash_table.ht_locks[i].ht_lock);
 794         kmem_cache_destroy(hdr_cache);
 795         kmem_cache_destroy(buf_cache);
 796 }
______unchanged_portion_omitted_

 873 static void
 874 buf_init(void)
 875 {
 876         uint64_t *ct;
 877         uint64_t hsize = 1ULL << 12;
 878         int i, j;

 880         /*
 881          * The hash table is big enough to fill all of physical memory
 882          * with an average 64K block size.  The table will take up
 883          * totalmem*sizeof(void*)/64K (eg. 128KB/GB with 8-byte pointers).
 884          */
 885         while (hsize * 65536 < physmem * PAGESIZE)
 886                 hsize <<= 1;
 887 retry:
 888         buf_hash_table.ht_mask = hsize - 1;
 889         buf_hash_table.ht_table =
 890             kmem_zalloc(hsize * sizeof (struct ht_table), KM_NOSLEEP);
 897             kmem_zalloc(hsize * sizeof (void*), KM_NOSLEEP);
 891         if (buf_hash_table.ht_table == NULL) {
 892                 ASSERT(hsize > (1ULL << 8));
 893                 hsize >>= 1;
 894                 goto retry;
 895         }

 897         hdr_cache = kmem_cache_create("arc_buf_hdr_t", sizeof (arc_buf_hdr_t),
 898             0, hdr_cons, hdr_dest, hdr_recl, NULL, NULL, 0);
 899         buf_cache = kmem_cache_create("arc_buf_t", sizeof (arc_buf_t),
 900             0, buf_cons, buf_dest, NULL, NULL, NULL, 0);

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



```
 906         for (i = 0; i < hsize; i++) {
 907                 mutex_init(&buf_hash_table.ht_table[i].lock,
 913         for (i = 0; i < BUF_LOCKS; i++) {
 914                 mutex_init(&buf_hash_table.ht_locks[i].ht_lock,
 908                     NULL, MUTEX_DEFAULT, NULL);
 909         }
 910 }
______unchanged_portion_omitted_
```