```
**********************************************************
  149294 Tue Nov 11 06:35:52 2014
new/usr/src/uts/common/fs/zfs/arc.c
arc_get_data_buf should be more aggressive in eviction when memory is unavailabl
**********************************************************
______unchanged_portion_omitted_

2490 /*
2491  * The buffer, supplied as the first argument, needs a data block.
2492  * So, if we are at cache max, determine which cache should be victimized.
2493  * We have the following cases:
2494  *
2495  * 1. Insert for MRU, p > sizeof(arc_anon + arc_mru) ->
2496  * In this situation if we're out of space, but the resident size of the MFU is
2497  * under the limit, victimize the MFU cache to satisfy this insertion request.
2498  *
2499  * 2. Insert for MRU, p <= sizeof(arc_anon + arc_mru) ->
2500  * Here, we've used up all of the available space for the MRU, so we need to
2501  * evict from our own cache instead.  Evict from the set of resident MRU
2502  * entries.
2503  *
2504  * 3. Insert for MFU (c - p) > sizeof(arc_mfu) ->
2505  * c minus p represents the MFU space in the cache, since p is the size of the
2506  * cache that is dedicated to the MRU.  In this situation there's still space on
2507  * the MFU side, so the MRU side needs to be victimized.
2508  *
2509  * 4. Insert for MFU (c - p) < sizeof(arc_mfu) ->
2510  * MFU's resident set is consuming more space than it has been allotted.  In
2511  * this situation, we must victimize our own cache, the MFU, for this insertion.
2512  */
2513 static void
2514 arc_get_data_buf(arc_buf_t *buf)
2515 {
2516         arc_state_t             *state = buf->b_hdr->b_state;
2517         uint64_t                size = buf->b_hdr->b_size;
2518         arc_buf_contents_t      type = buf->b_hdr->b_type;

2520         arc_adapt(size, state);

2522 top:
2523         /*
2524          * We have not yet reached cache maximum size,
2525          * just allocate a new buffer.
2526          */
2527         if (!arc_evict_needed(type)) {
2528                 if (type == ARC_BUFC_METADATA) {
2529                         buf->b_data = zio_buf_alloc_canfail(size);
2530                         if (buf->b_data != NULL) {
2528                         buf->b_data = zio_buf_alloc(size);
2531                                 arc_space_consume(size, ARC_SPACE_DATA);
2532                                 goto out;
2533                         }
2534                 } else {
2535                         ASSERT(type == ARC_BUFC_DATA);
2536                         buf->b_data = zio_data_buf_alloc_canfail(size);
2537                         if (buf->b_data != NULL) {
2532                         buf->b_data = zio_data_buf_alloc(size);
2538                                 ARCSTAT_INCR(arcstat_data_size, size);
2539                                 atomic_add_64(&arc_size, size);
2535                         }
2540                                 goto out;
2541                         }
2542                 }
2543                 /*
2544                  * Memory allocation failed probably due to excessive
2545                  * fragmentation, we need to evict regardless.
```



```
2546                  */
2547         }

2549         /*
2550          * If we are prefetching from the mfu ghost list, this buffer
2551          * will end up on the mru list; so steal space from there.
2552          */
2553         if (state == arc_mfu_ghost)
2554                 state = buf->b_hdr->b_flags & ARC_PREFETCH ? arc_mru : arc_mfu;
2555         else if (state == arc_mru_ghost)
2556                 state = arc_mru;

2558         if (state == arc_mru || state == arc_anon) {
2559                 uint64_t mru_used = arc_anon->arcs_size + arc_mru->arcs_size;
2560                 state = (arc_mfu->arcs_lsize[type] >= size &&
2561                     arc_p > mru_used) ? arc_mfu : arc_mru;
2562         } else {
2563                 /* MFU cases */
2564                 uint64_t mfu_space = arc_c - arc_p;
2565                 state =  (arc_mru->arcs_lsize[type] >= size &&
2566                     mfu_space > arc_mfu->arcs_size) ? arc_mru : arc_mfu;
2567         }
2568         if ((buf->b_data = arc_evict(state, NULL, size, TRUE, type)) == NULL) {
2559                 if (type == ARC_BUFC_METADATA) {
2560                         buf->b_data = zio_buf_alloc(size);
2561                         arc_space_consume(size, ARC_SPACE_DATA);
2562                 } else {
2563                         ASSERT(type == ARC_BUFC_DATA);
2564                         buf->b_data = zio_data_buf_alloc(size);
2565                         ARCSTAT_INCR(arcstat_data_size, size);
2566                         atomic_add_64(&arc_size, size);
2567                 }
2569                 ARCSTAT_BUMP(arcstat_recycle_miss);
2570                 goto top;
2571         }
2572         ASSERT(buf->b_data != NULL);
2573 out:
2574         /*
2575          * Update the state size.  Note that ghost states have a
2576          * "ghost size" and so don't need to be updated.
2577          */
2578         if (!GHOST_STATE(buf->b_hdr->b_state)) {
2579                 arc_buf_hdr_t *hdr = buf->b_hdr;

2581                 atomic_add_64(&hdr->b_state->arcs_size, size);
2582                 if (list_link_active(&hdr->b_arc_node)) {
2583                         ASSERT(refcount_is_zero(&hdr->b_refcnt));
2584                         atomic_add_64(&hdr->b_state->arcs_lsize[type], size);
2585                 }
2586                 /*
2587                  * If we are growing the cache, and we are adding anonymous
2588                  * data, and we have outgrown arc_p, update arc_p
2589                  */
2590                 if (arc_size < arc_c && hdr->b_state == arc_anon &&
2591                     arc_anon->arcs_size + arc_mru->arcs_size > arc_p)
2592                         arc_p = MIN(arc_c, arc_p + size);
2593         }
2594 }
______unchanged_portion_omitted_
```

```
**********************************************************
   18196 Tue Nov 11 06:35:53 2014
new/usr/src/uts/common/fs/zfs/sys/zio.h
arc_get_data_buf should be more aggressive in eviction when memory is unavailabl
**********************************************************
______unchanged_portion_omitted_

 452 extern zio_t *zio_null(zio_t *pio, spa_t *spa, vdev_t *vd,
 453     zio_done_func_t *done, void *private, enum zio_flag flags);

 455 extern zio_t *zio_root(spa_t *spa,
 456     zio_done_func_t *done, void *private, enum zio_flag flags);

 458 extern zio_t *zio_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, void *data,
 459     uint64_t size, zio_done_func_t *done, void *private,
 460     zio_priority_t priority, enum zio_flag flags, const zbookmark_phys_t *zb);

 462 extern zio_t *zio_write(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
 463     void *data, uint64_t size, const zio_prop_t *zp,
 464     zio_done_func_t *ready, zio_done_func_t *physdone, zio_done_func_t *done,
 465     void *private,
 466     zio_priority_t priority, enum zio_flag flags, const zbookmark_phys_t *zb);

 468 extern zio_t *zio_rewrite(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
 469     void *data, uint64_t size, zio_done_func_t *done, void *private,
 470     zio_priority_t priority, enum zio_flag flags, zbookmark_phys_t *zb);

 472 extern void zio_write_override(zio_t *zio, blkptr_t *bp, int copies,
 473     boolean_t nopwrite);

 475 extern void zio_free(spa_t *spa, uint64_t txg, const blkptr_t *bp);

 477 extern zio_t *zio_claim(zio_t *pio, spa_t *spa, uint64_t txg,
 478     const blkptr_t *bp,
 479     zio_done_func_t *done, void *private, enum zio_flag flags);

 481 extern zio_t *zio_ioctl(zio_t *pio, spa_t *spa, vdev_t *vd, int cmd,
 482     zio_done_func_t *done, void *private, enum zio_flag flags);

 484 extern zio_t *zio_read_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
 485     uint64_t size, void *data, int checksum,
 486     zio_done_func_t *done, void *private, zio_priority_t priority,
 487     enum zio_flag flags, boolean_t labels);

 489 extern zio_t *zio_write_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
 490     uint64_t size, void *data, int checksum,
 491     zio_done_func_t *done, void *private, zio_priority_t priority,
 492     enum zio_flag flags, boolean_t labels);

 494 extern zio_t *zio_free_sync(zio_t *pio, spa_t *spa, uint64_t txg,
 495     const blkptr_t *bp, enum zio_flag flags);

 497 extern int zio_alloc_zil(spa_t *spa, uint64_t txg, blkptr_t *new_bp,
 498     blkptr_t *old_bp, uint64_t size, boolean_t use_slog);
 499 extern void zio_free_zil(spa_t *spa, uint64_t txg, blkptr_t *bp);
 500 extern void zio_flush(zio_t *zio, vdev_t *vd);
 501 extern void zio_shrink(zio_t *zio, uint64_t size);

 503 extern int zio_wait(zio_t *zio);
 504 extern void zio_nowait(zio_t *zio);
 505 extern void zio_execute(zio_t *zio);
 506 extern void zio_interrupt(zio_t *zio);

 508 extern zio_t *zio_walk_parents(zio_t *cio);
 509 extern zio_t *zio_walk_children(zio_t *pio);
 510 extern zio_t *zio_unique_parent(zio_t *cio);
```



```
 511 extern void zio_add_child(zio_t *pio, zio_t *cio);

 513 extern void *zio_buf_alloc(size_t size);
 514 extern void *zio_buf_alloc_canfail(size_t size);
 515 extern void zio_buf_free(void *buf, size_t size);
 516 extern void *zio_data_buf_alloc(size_t size);
 517 extern void *zio_data_buf_alloc_canfail(size_t size);
 518 extern void zio_data_buf_free(void *buf, size_t size);

 520 extern void zio_resubmit_stage_async(void *);

 522 extern zio_t *zio_vdev_child_io(zio_t *zio, blkptr_t *bp, vdev_t *vd,
 523     uint64_t offset, void *data, uint64_t size, int type,
 524     zio_priority_t priority, enum zio_flag flags,
 525     zio_done_func_t *done, void *private);

 527 extern zio_t *zio_vdev_delegated_io(vdev_t *vd, uint64_t offset,
 528     void *data, uint64_t size, int type, zio_priority_t priority,
 529     enum zio_flag flags, zio_done_func_t *done, void *private);

 531 extern void zio_vdev_io_bypass(zio_t *zio);
 532 extern void zio_vdev_io_reissue(zio_t *zio);
 533 extern void zio_vdev_io_redone(zio_t *zio);

 535 extern void zio_checksum_verified(zio_t *zio);
 536 extern int zio_worst_error(int e1, int e2);

 538 extern enum zio_checksum zio_checksum_select(enum zio_checksum child,
 539     enum zio_checksum parent);
 540 extern enum zio_checksum zio_checksum_dedup_select(spa_t *spa,
 541     enum zio_checksum child, enum zio_checksum parent);
 542 extern enum zio_compress zio_compress_select(enum zio_compress child,
 543     enum zio_compress parent);

 545 extern void zio_suspend(spa_t *spa, zio_t *zio);
 546 extern int zio_resume(spa_t *spa);
 547 extern void zio_resume_wait(spa_t *spa);

 549 /*
 550  * Initial setup and teardown.
 551  */
 552 extern void zio_init(void);
 553 extern void zio_fini(void);

 555 /*
 556  * Fault injection
 557  */
 558 struct zinject_record;
 559 extern uint32_t zio_injection_enabled;
 560 extern int zio_inject_fault(char *name, int flags, int *id,
 561     struct zinject_record *record);
 562 extern int zio_inject_list_next(int *id, char *name, size_t buflen,
 563     struct zinject_record *record);
 564 extern int zio_clear_fault(int id);
 565 extern void zio_handle_panic_injection(spa_t *spa, char *tag, uint64_t type);
 566 extern int zio_handle_fault_injection(zio_t *zio, int error);
 567 extern int zio_handle_device_injection(vdev_t *vd, zio_t *zio, int error);
 568 extern int zio_handle_label_injection(zio_t *zio, int error);
 569 extern void zio_handle_ignored_writes(zio_t *zio);
 570 extern uint64_t zio_handle_io_delay(zio_t *zio);

 572 /*
 573  * Checksum ereport functions
 574  */
 575 extern void zfs_ereport_start_checksum(spa_t *spa, vdev_t *vd, struct zio *zio,
 576     uint64_t offset, uint64_t length, void *arg, struct zio_bad_cksum *info);
```

```
 577 extern void zfs_ereport_finish_checksum(zio_cksum_report_t *report,
 578     const void *good_data, const void *bad_data, boolean_t drop_if_identical);

 580 extern void zfs_ereport_send_interim_checksum(zio_cksum_report_t *report);
 581 extern void zfs_ereport_free_checksum(zio_cksum_report_t *report);

 583 /* If we have the good data in hand, this function can be used */
 584 extern void zfs_ereport_post_checksum(spa_t *spa, vdev_t *vd,
 585     struct zio *zio, uint64_t offset, uint64_t length,
 586     const void *good_data, const void *bad_data, struct zio_bad_cksum *info);

 588 /* Called from spa_sync(), but primarily an injection handler */
 589 extern void spa_handle_ignored_writes(spa_t *spa);

 591 /* zbookmark_phys functions */
 592 boolean_t zbookmark_is_before(const struct dnode_phys *dnp,
 593     const zbookmark_phys_t *zb1, const zbookmark_phys_t *zb2);

 595 #ifdef  __cplusplus
 596 }
```

unchanged_portion_omitted_

****
94148 Tue Nov 11 06:35:53 2014
new/usr/src/uts/common/fs/zfs/zio.c
arc_get_data_buf should be more aggressive in eviction when memory is unavailabl
****
_____***unchanged_portion_omitted_***

```
223 /*
224  * Same as zio_buf_alloc, but won't sleep in case memory cannot be allocated
225  * and will instead return immediately with a failure.
226  */
227 void *
228 zio_buf_alloc_canfail(size_t size)
229 {
230         size_t c = (size - 1) >> SPA_MINBLOCKSHIFT;

232         ASSERT(c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT);

234         return (kmem_cache_alloc(zio_buf_cache[c], KM_NOSLEEP | KM_NORMALPRI));
235 }

237 /*
238  * Use zio_data_buf_alloc to allocate data.  The data will not appear in a
239  * crashdump if the kernel panics.  This exists so that we will limit the amount
240  * of ZFS data that shows up in a kernel crashdump.  (Thus reducing the amount
241  * of kernel heap dumped to disk when the kernel panics)
242  */
243 void *
244 zio_data_buf_alloc(size_t size)
245 {
246         size_t c = (size - 1) >> SPA_MINBLOCKSHIFT;

248         ASSERT(c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT);

250         return (kmem_cache_alloc(zio_data_buf_cache[c], KM_PUSHPAGE));
251 }

253 /*
254  * Same as zio_data_buf_alloc, but won't sleep in case memory cannot be
255  * allocated and will instead return immediately with a failure.
256  */
257 void *
258 zio_data_buf_alloc_canfail(size_t size)
259 {
260         size_t c = (size - 1) >> SPA_MINBLOCKSHIFT;

262         ASSERT(c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT);

264         return (kmem_cache_alloc(zio_data_buf_cache[c],
265             KM_NOSLEEP | KM_NORMALPRI));
266 }

268 void
269 zio_buf_free(void *buf, size_t size)
270 {
271         size_t c = (size - 1) >> SPA_MINBLOCKSHIFT;

273         ASSERT(c < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT);

275         kmem_cache_free(zio_buf_cache[c], buf);
276 }
```

_____***unchanged_portion_omitted_***