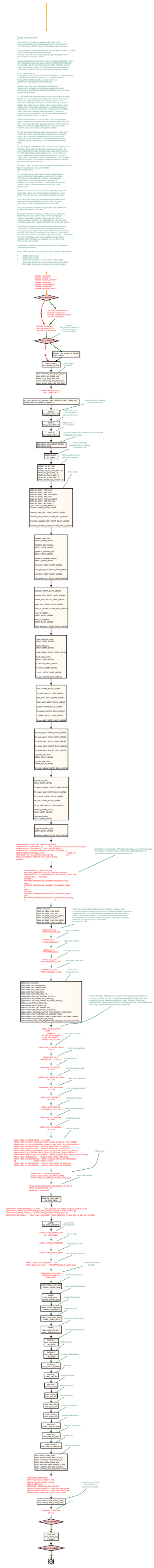
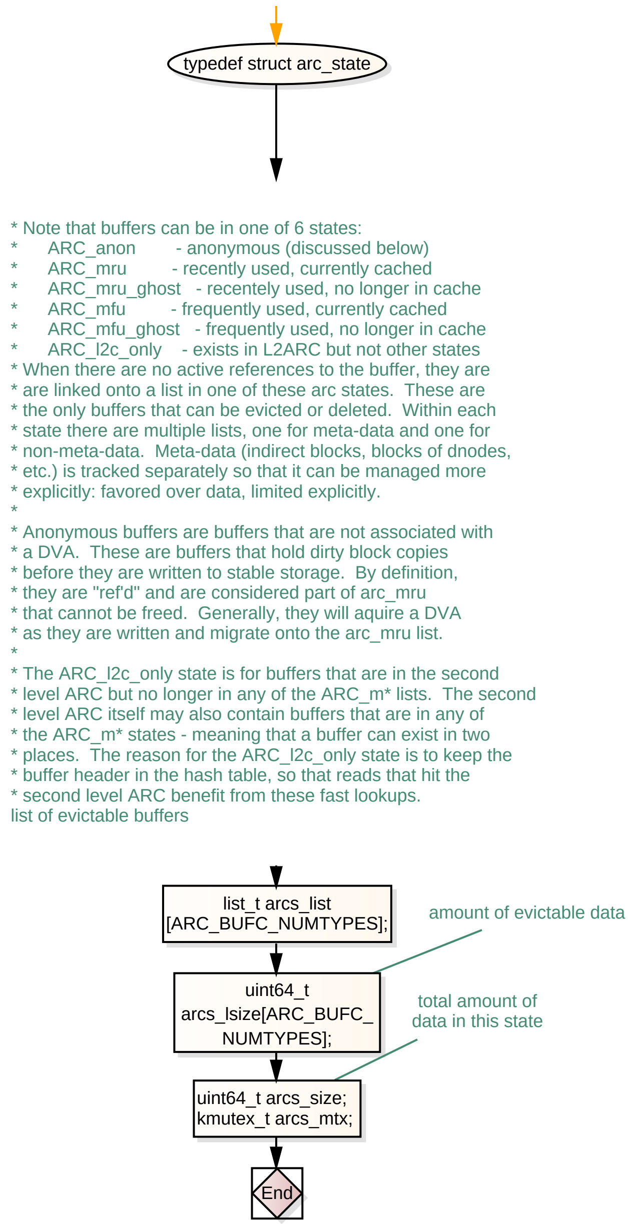
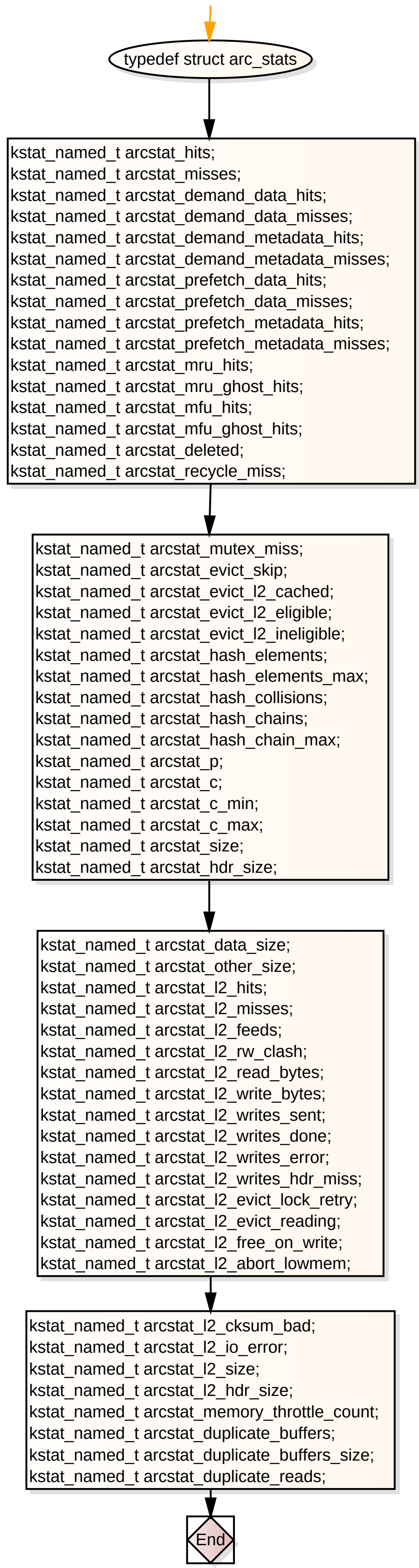
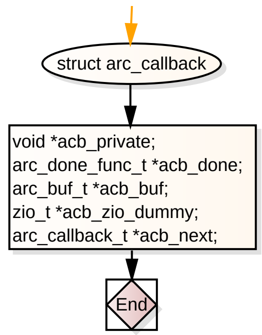
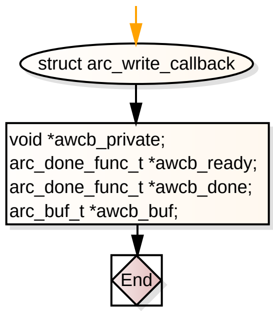
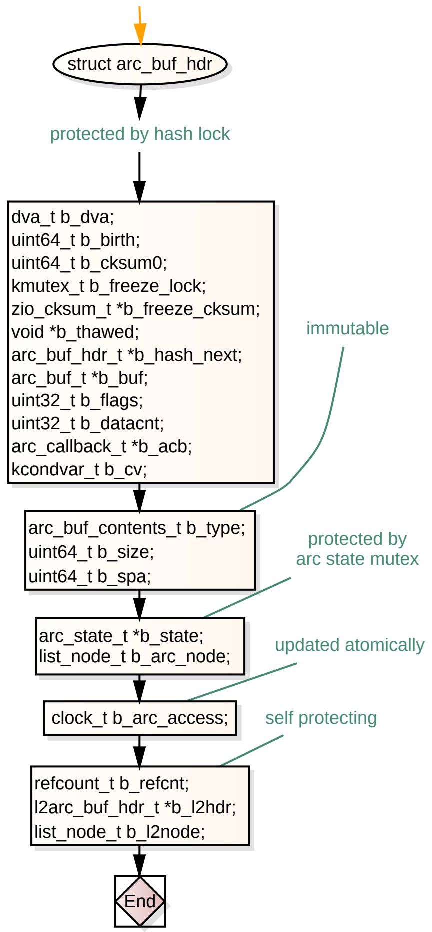


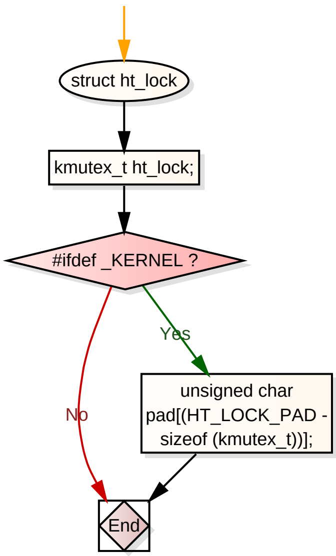

typedef struct arc_state
* Note that buffers can be in one of 6 states:
*   ARC_anon   - anonymous (discussed below)
*   ARC_mru   - recently used, currently cached
*   ARC_mru_ghost   - recentely used, no longer in cache
*   ARC_mfu   - frequently used, currently cached
*   ARC_mfu_ghost   - frequently 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.
list of evictable buffers
list_t arcs_list
[ARC_BUFC_NUMTYPES];
amount of evictable data
uint64_t
arcs_lsize[ARC_BUFC_
NUMTYPES];
total amount of
data in this state
uint64_t arcs_size;
kmutex_t arcs_mtx;
End

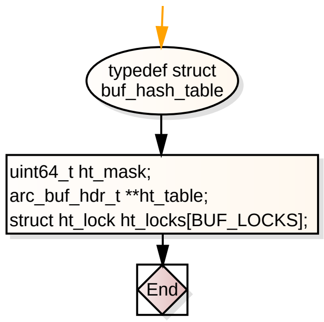

typedef struct arc_stats
kstat_named_t arcstat_hits;
kstat_named_t arcstat_misses;
kstat_named_t arcstat_demand_data_hits;
kstat_named_t arcstat_demand_data_misses;
kstat_named_t arcstat_demand_metadata_hits;
kstat_named_t arcstat_demand_metadata_misses;
kstat_named_t arcstat_prefetch_data_hits;
kstat_named_t 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;
kstat_named_t arcstat_mutex_miss;
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_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_hdr_size;
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;
End

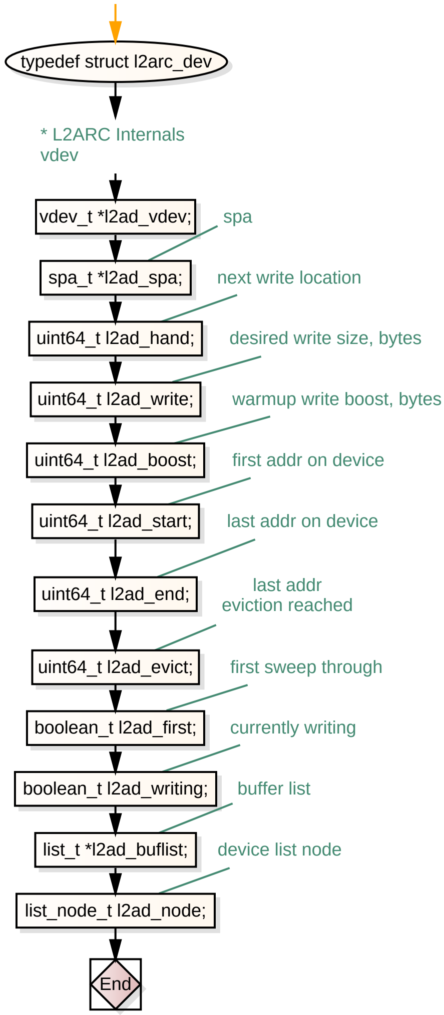

struct arc_callback
void *acb_private;
arc_done_func_t *acb_done;
arc_buf_t *acb_buf;
zio_t *acb_zio_dummy;
arc_callback_t *acb_next;
End

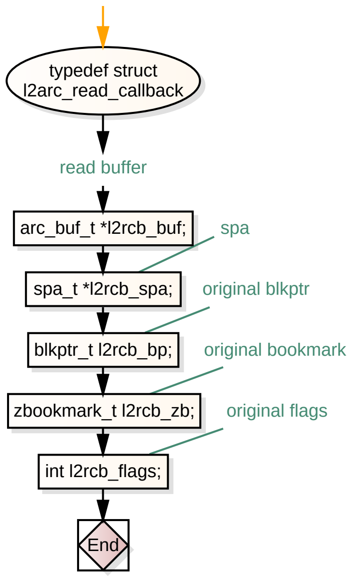

struct arc_write_callback
void *awcb_private;
arc_done_func_t *awcb_ready;
arc_done_func_t *awcb_done;
arc_buf_t *awcb_buf;
End


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;
End

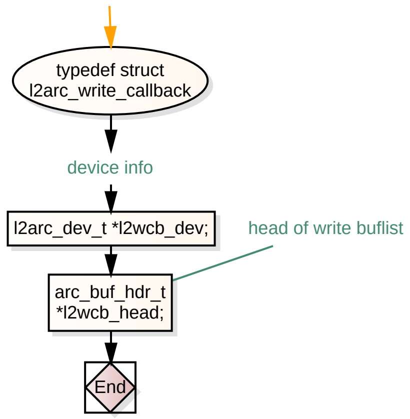

struct ht_lock
kmutex_t ht_lock;
#ifdef _KERNEL ?
Yes
No
unsigned char pad[(HT_LOCK_PAD - sizeof (kmutex_t))];
End

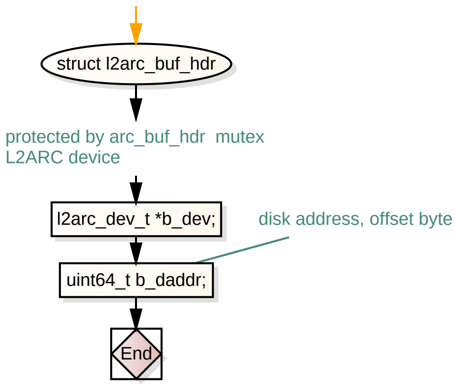

typedef struct
buf_hash_table
uint64_t ht_mask;
arc_buf_hdr_t **ht_table;
struct ht_lock ht_locks[BUF_LOCKS];
End

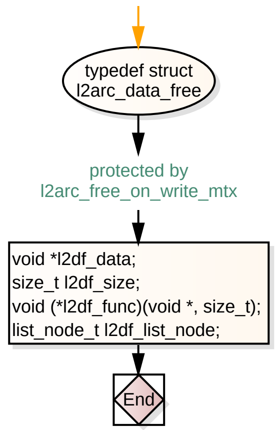

typedef struct l2arc_dev
* L2ARC Internals
vdev
vdev_t *l2ad_vdev;
spa
spa_t *l2ad_spa;
next write location
uint64_t l2ad_hand;
desired write size, bytes
uint64_t l2ad_write;
warmup write boost, bytes
uint64_t l2ad_boost;
first addr on device
uint64_t l2ad_start;
last addr on device
uint64_t l2ad_end;
last addr
eviction reached
uint64_t l2ad_evict;
first sweep through
boolean_t l2ad_first;
currently writing
boolean_t l2ad_writing;
buffer list
list_t *l2ad_buflist;
device list node
list_node_t l2ad_node;
End

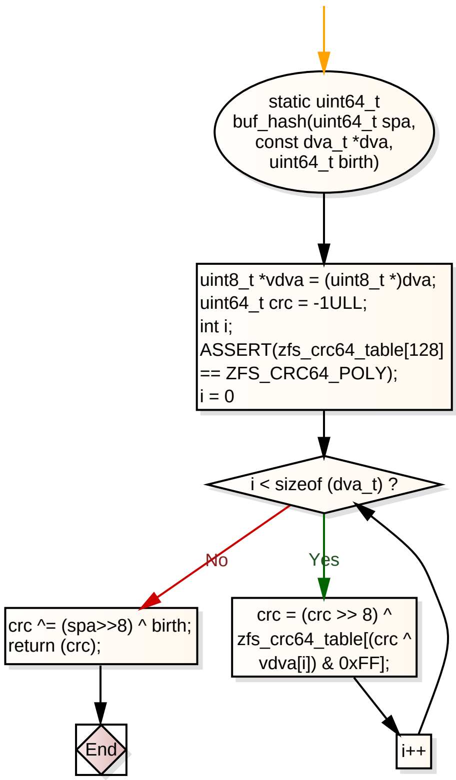


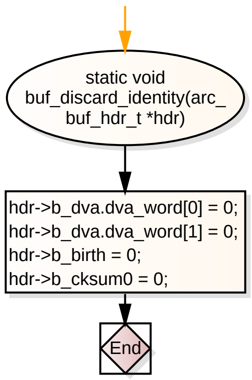
typedef struct l2arc_read_callback
read buffer
arc_buf_t *l2rcb_buf;
spa
spa_t *l2rcb_spa;
original blkptr
blkptr_t l2rcb_bp;
original bookmark
zbookmark_t l2rcb_zb;
original flags
int l2rcb_flags;
End

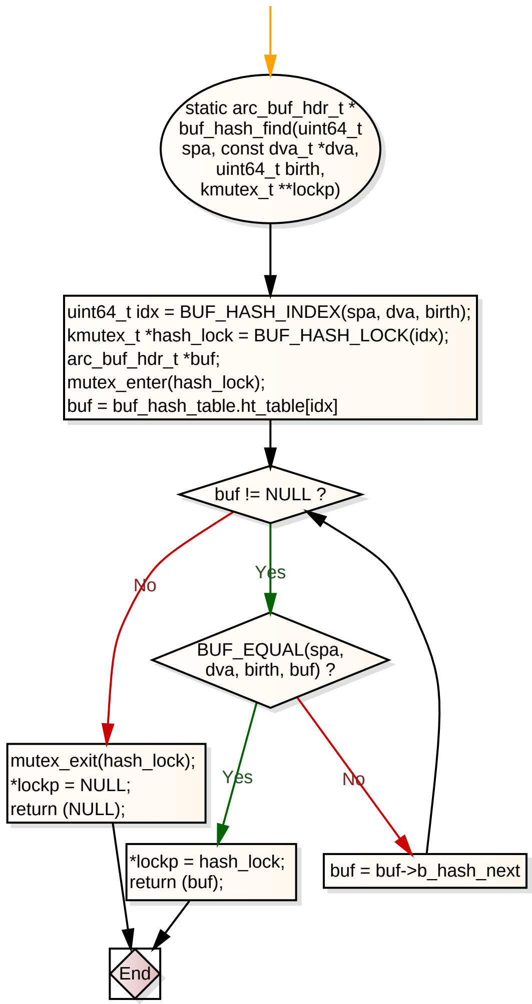

typedef struct l2arc_write_callback
device info
l2arc_dev_t *l2wcb_dev;
head of write buflist
arc_buf_hdr_t *l2wcb_head;
End

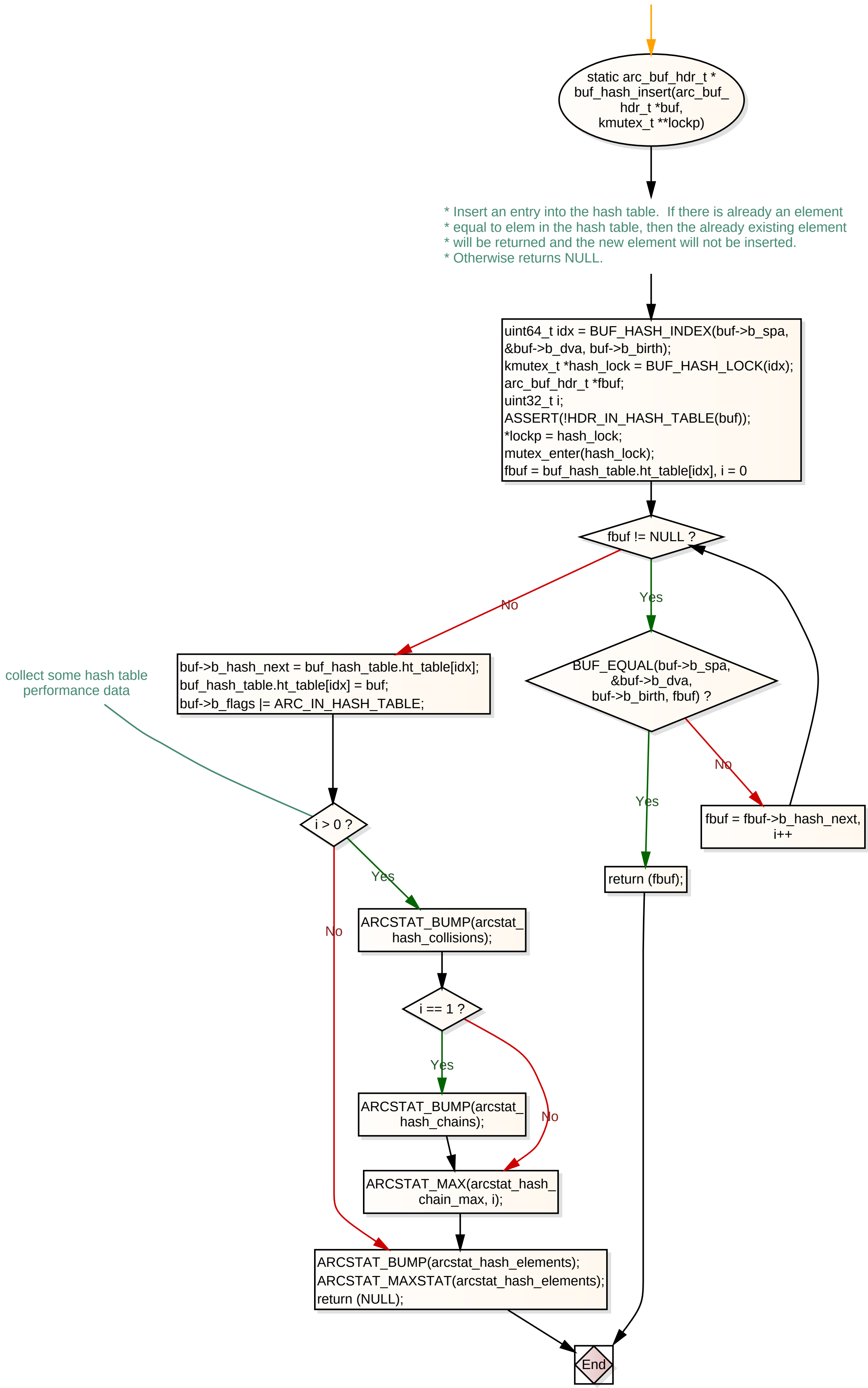

struct l2arc_buf_hdr
protected by arc_buf_hdr mutex
L2ARC device
l2arc_dev_t *b_dev;
disk address, offset byte
uint64_t b_daddr;
End

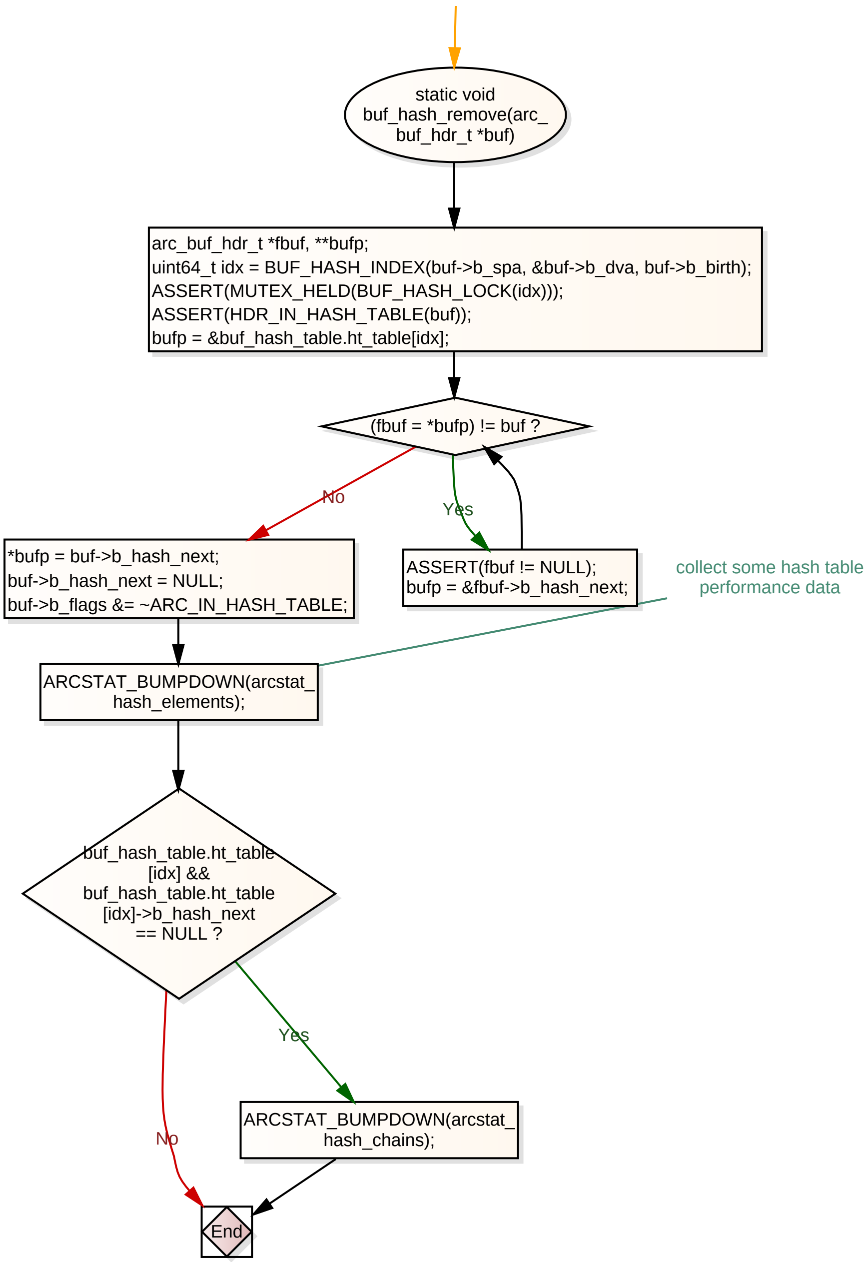

typedef struct
l2arc_data_free
protected by
l2arc_free_on_write_mtx
void *l2df_data;
size_t l2df_size;
void (*l2df_func)(void *, size_t);
list_node_t l2df_list_node;
End

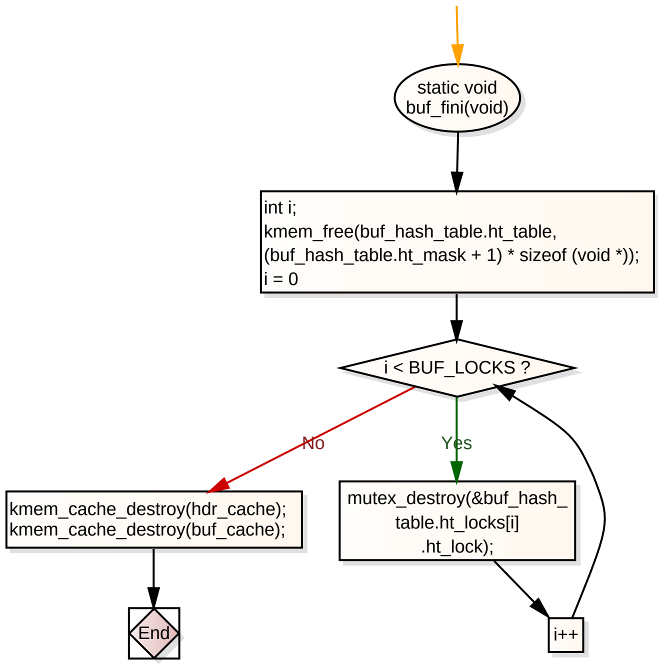

static uint64_t buf_hash(uint64_t spa, const dva_t *dva, uint64_t birth)
uint8_t *vdva = (uint8_t *)dva;
uint64_t crc = -1ULL;
int i;
ASSERT(zfs_crc64_table[128] == ZFS_CRC64_POLY);
i = 0
i < sizeof (dva_t) ?
No
Yes
crc ^= (spa>>8) ^ birth;
return (crc);
crc = (crc >> 8) ^ zfs_crc64_table[(crc ^ vdva[i]) & 0xFF];
i++
End

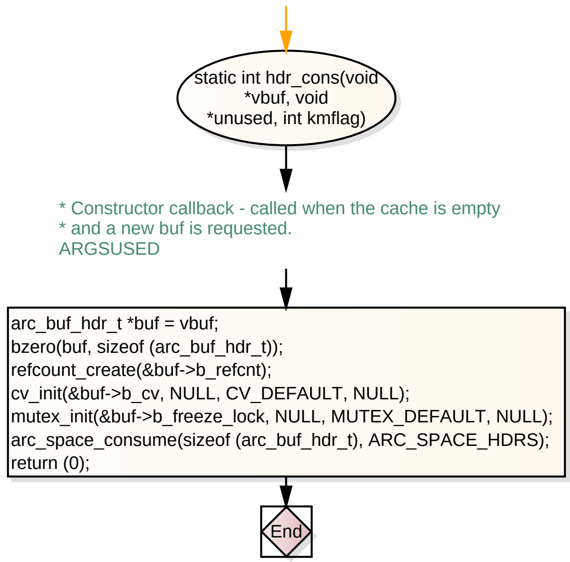

static void
buf_discard_identity(arc_
buf_hdr_t *hdr)
hdr->b_dva.dva_word[0] = 0;
hdr->b_dva.dva_word[1] = 0;
hdr->b_birth = 0;
hdr->b_cksum0 = 0;
End

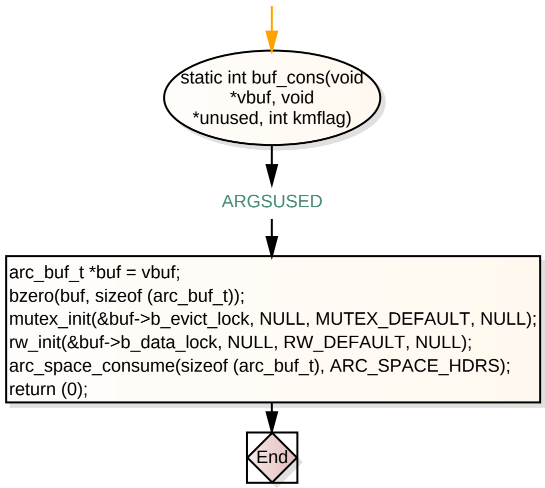

static arc_buf_hdr_t * buf_hash_find(uint64_t spa, const dva_t *dva, uint64_t birth, kmutex_t **lockp)
uint64_t idx = BUF_HASH_INDEX(spa, dva, birth);
kmutex_t *hash_lock = BUF_HASH_LOCK(idx);
arc_buf_hdr_t *buf;
mutex_enter(hash_lock);
buf = buf_hash_table.ht_table[idx]
buf != NULL ?
Yes
No
BUF_EQUAL(spa, dva, birth, buf) ?
mutex_exit(hash_lock);
*lockp = NULL;
return (NULL);
Yes
No
*lockp = hash_lock;
return (buf);
buf = buf->b_hash_next
End

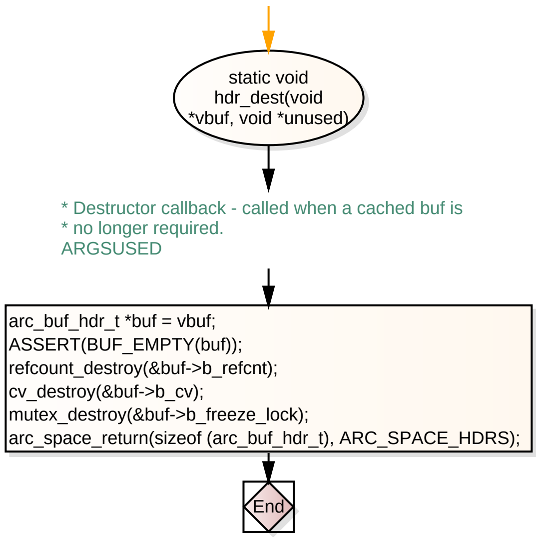

static arc_buf_hdr_t *
buf_hash_insert(arc_buf_
hdr_t *buf,
kmutex_t **lockp)
* Insert an entry into the hash table. If there is already an element
* equal to elem in the hash table, then the already existing element
* will be returned and the new element will not be inserted.
* Otherwise returns NULL.
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(!HDR_IN_HASH_TABLE(buf));
*lockp = hash_lock;
mutex_enter(hash_lock);
fbuf = buf_hash_table.ht_table[idx], i = 0
fbuf != NULL ?
No
Yes
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
BUF_EQUAL(buf->b_spa,
&buf->b_dva,
buf->b_birth, fbuf) ?
No
Yes
fbuf = fbuf->b_hash_next,
i++
return (fbuf);
i > 0 ?
Yes
No
ARCSTAT_BUMP(arcstat_
hash_collisions);
i == 1 ?
Yes
No
ARCSTAT_BUMP(arcstat_
hash_chains);
ARCSTAT_MAX(arcstat_hash_
chain_max, i);
ARCSTAT_BUMP(arcstat_hash_elements);
ARCSTAT_MAXSTAT(arcstat_hash_elements);
return (NULL);
End

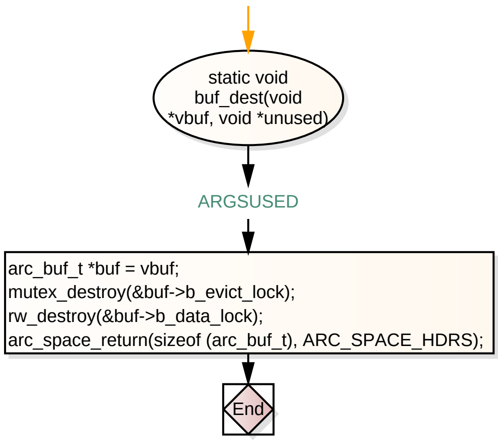

static void buf_hash_remove(arc_buf_hdr_t *buf)
arc_buf_hdr_t *fbuf, **bufp;
uint64_t idx = BUF_HASH_INDEX(buf->b_spa, &buf->b_dva, buf->b_birth);
ASSERT(MUTEX_HELD(BUF_HASH_LOCK(idx)));
ASSERT(HDR_IN_HASH_TABLE(buf));
bufp = &buf_hash_table.ht_table[idx];
(fbuf = *bufp) != buf ?
No
Yes
*bufp = buf->b_hash_next;
buf->b_hash_next = NULL;
buf->b_flags &= ~ARC_IN_HASH_TABLE;
ASSERT(fbuf != NULL);
bufp = &fbuf->b_hash_next;
collect some hash table performance data
ARCSTAT_BUMPDOWN(arcstat_hash_elements);
buf_hash_table.ht_table[idx] && buf_hash_table.ht_table[idx]->b_hash_next == NULL ?
Yes
No
ARCSTAT_BUMPDOWN(arcstat_hash_chains);
End

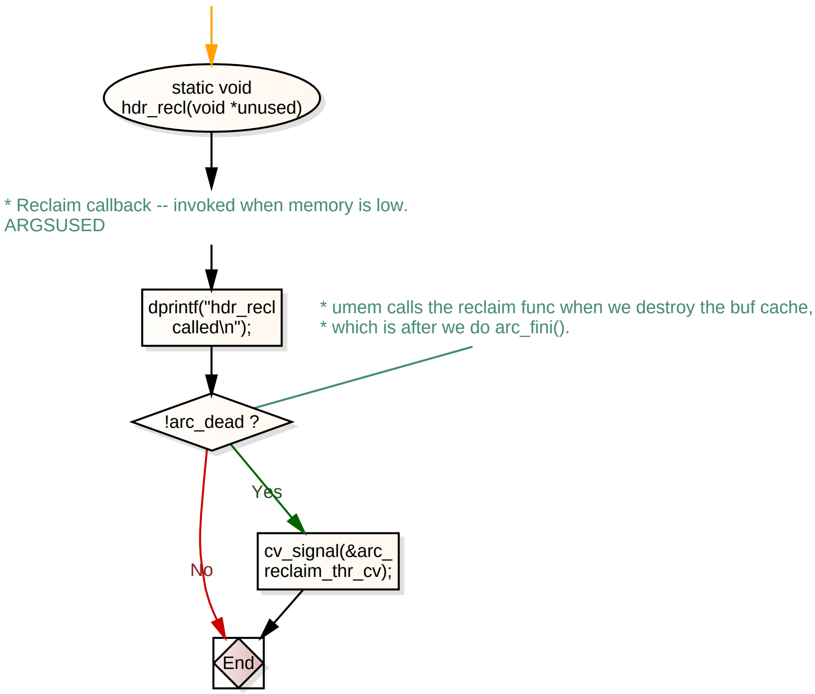

static void
buf_fini(void)
int i;
kmem_free(buf_hash_table.ht_table,
(buf_hash_table.ht_mask + 1) * sizeof (void *));
i = 0
i < BUF_LOCKS ?
No
Yes
kmem_cache_destroy(hdr_cache);
kmem_cache_destroy(buf_cache);
mutex_destroy(&buf_hash_
table.ht_locks[i]
.ht_lock);
End
i++

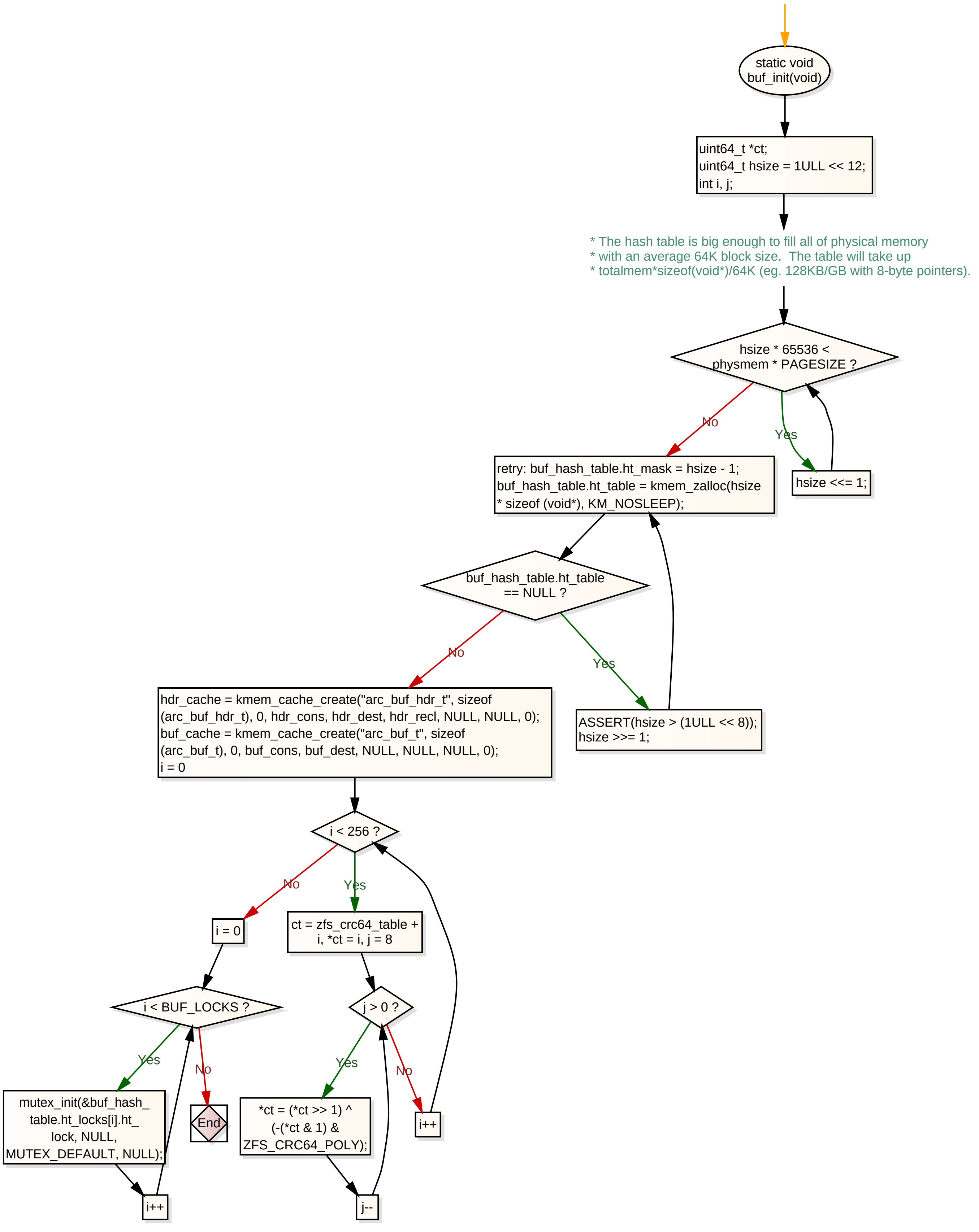

static int hdr_cons(void *vbuf, void *unused, int kmflag)
* Constructor callback - called when the cache is empty
* and a new buf is requested.
ARGSUSED
arc_buf_hdr_t *buf = vbuf;
bzero(buf, sizeof (arc_buf_hdr_t));
refcount_create(&buf->b_refcnt);
cv_init(&buf->b_cv, NULL, CV_DEFAULT, NULL);
mutex_init(&buf->b_freeze_lock, NULL, MUTEX_DEFAULT, NULL);
arc_space_consume(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
return (0);
End


static int buf_cons(void *vbuf, void *unused, int kmflag)
ARGSUSED
arc_buf_t *buf = vbuf;
bzero(buf, sizeof (arc_buf_t));
mutex_init(&buf->b_evict_lock, NULL, MUTEX_DEFAULT, NULL);
rw_init(&buf->b_data_lock, NULL, RW_DEFAULT, NULL);
arc_space_consume(sizeof (arc_buf_t), ARC_SPACE_HDRS);
return (0);
End

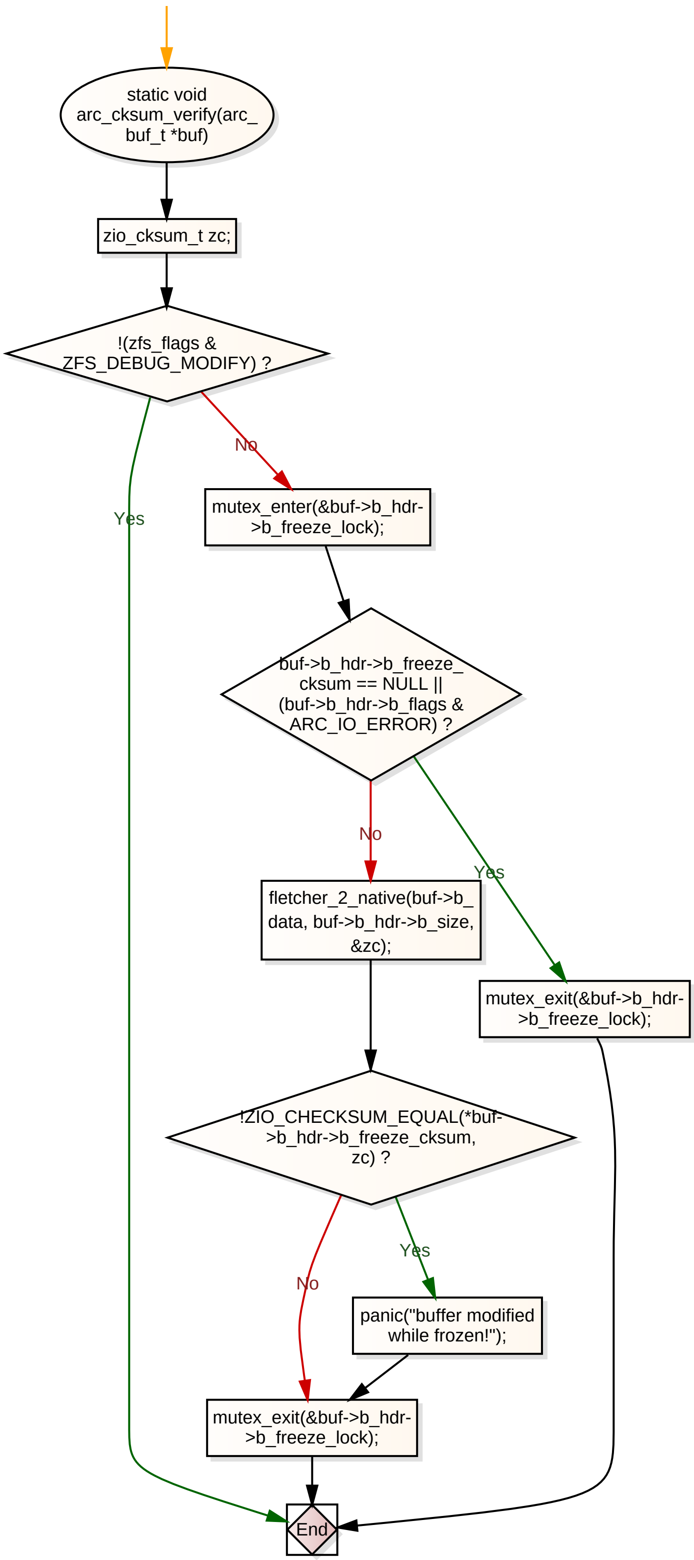

static void
hdr_dest(void
*vbuf, void *unused)
* Destructor callback - called when a cached buf is
* no longer required.
ARGSUSED
arc_buf_hdr_t *buf = vbuf;
ASSERT(BUF_EMPTY(buf));
refcount_destroy(&buf->b_refcnt);
cv_destroy(&buf->b_cv);
mutex_destroy(&buf->b_freeze_lock);
arc_space_return(sizeof (arc_buf_hdr_t), ARC_SPACE_HDRS);
End


static void buf_dest(void *vbuf, void *unused)
ARGSUSED
arc_buf_t *buf = vbuf;
mutex_destroy(&buf->b_evict_lock);
rw_destroy(&buf->b_data_lock);
arc_space_return(sizeof (arc_buf_t), ARC_SPACE_HDRS);
End

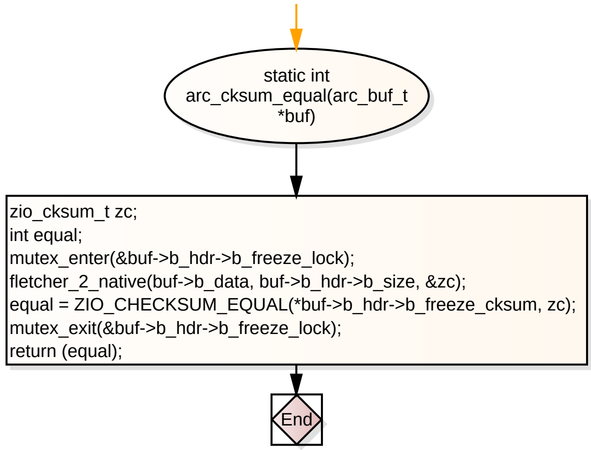

static void
hdr_recl(void *unused)
* Reclaim callback -- invoked when memory is low.
ARGSUSED
dprintf("hdr_recl called\n");
* umem calls the reclaim func when we destroy the buf cache,
* which is after we do arc_fini().
!arc_dead ?
Yes
No
cv_signal(&arc_reclaim_thr_cv);
End

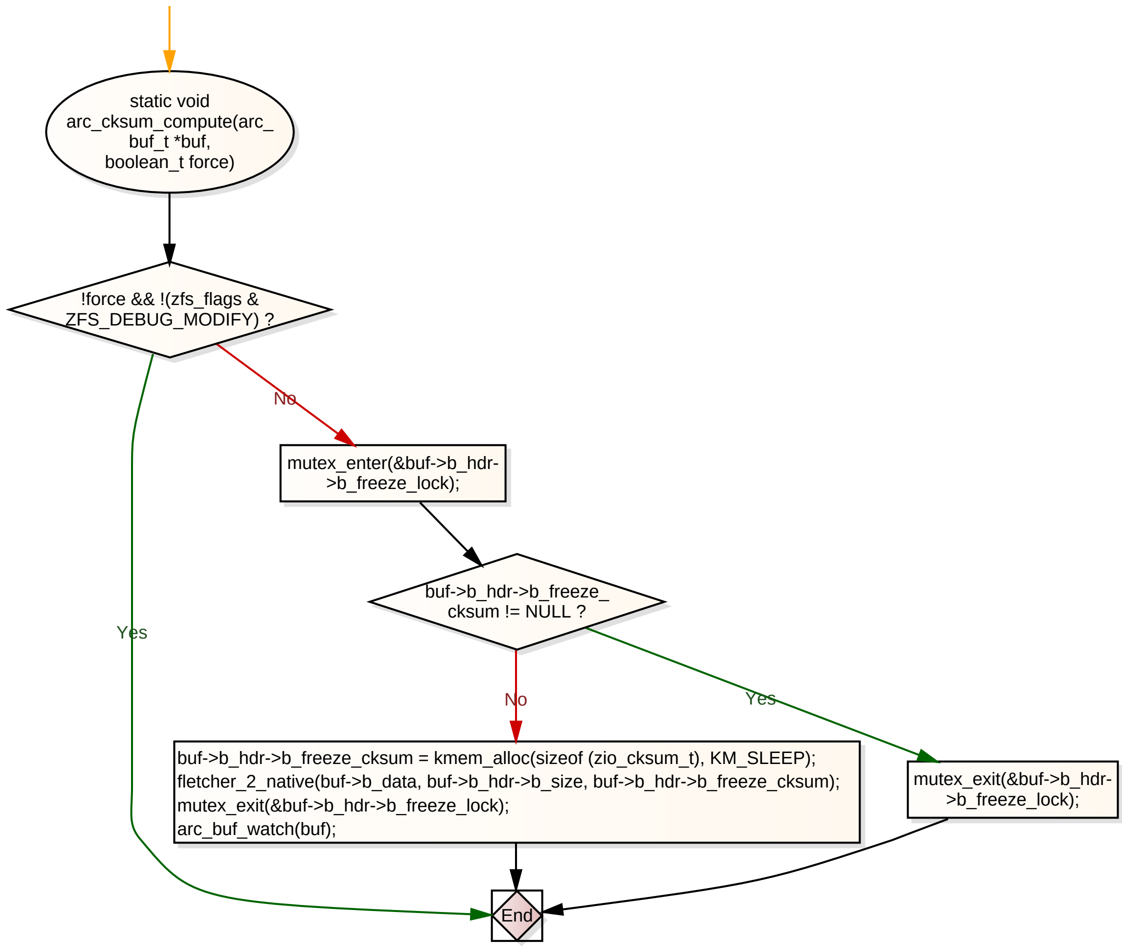


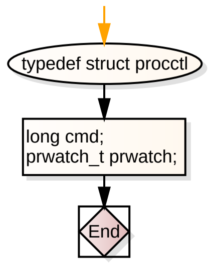
static void
buf_init(void)
uint64_t *ct;
uint64_t hsize = 1ULL << 12;
int i, j;
* The hash table is big enough to fill all of physical memory
* with an average 64K block size. The table will take up
* totalmem*sizeof(void*)/64K (eg. 128KB/GB with 8-byte pointers).
hsize * 65536 <
physmem * PAGESIZE ?
No
Yes
retry: buf_hash_table.ht_mask = hsize - 1;
buf_hash_table.ht_table = kmem_zalloc(hsize
* sizeof (void*), KM_NOSLEEP);
hsize <<= 1;
buf_hash_table.ht_table
== NULL ?
No
Yes
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);
i = 0
ASSERT(hsize > (1ULL << 8));
hsize >>= 1;
i < 256 ?
No
Yes
i = 0
ct = zfs_crc64_table +
i, *ct = i, j = 8
i < BUF_LOCKS ?
j > 0 ?
Yes
No
Yes
No
mutex_init(&buf_hash_
table.ht_locks[i].ht_
lock, NULL,
MUTEX_DEFAULT, NULL);
End
*ct = (*ct >> 1) ^
(-(*ct & 1) &
ZFS_CRC64_POLY);
i++
i++
j--

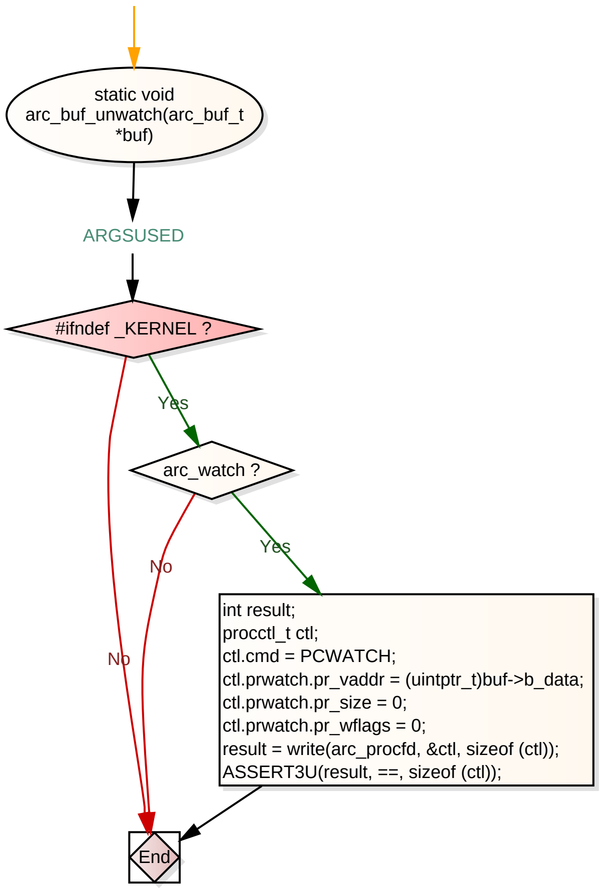


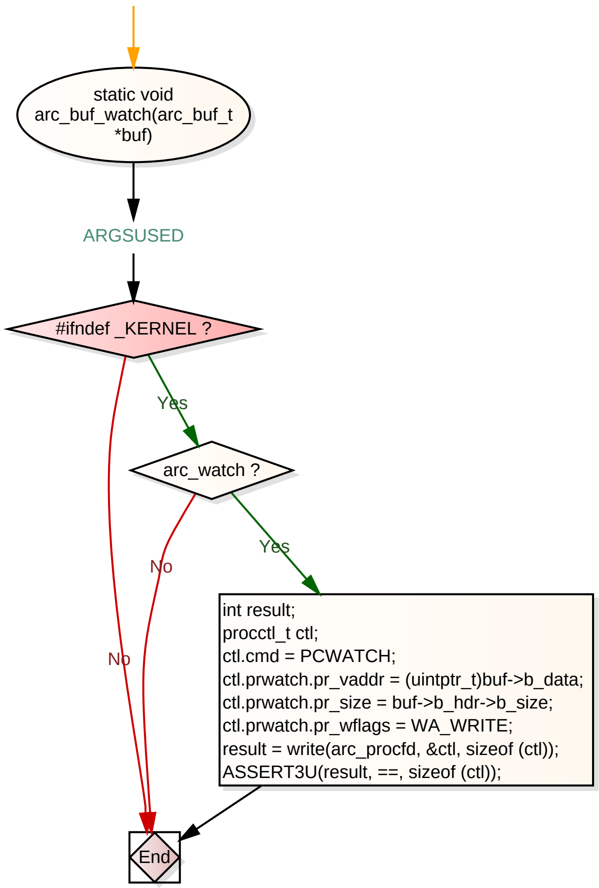
static void arc_cksum_verify(arc_buf_t *buf)
zio_cksum_t zc;
!(zfs_flags & ZFS_DEBUG_MODIFY) ?
No
Yes
mutex_enter(&buf->b_hdr->b_freeze_lock);
buf->b_hdr->b_freeze_cksum == NULL || (buf->b_hdr->b_flags & ARC_IO_ERROR) ?
No
Yes
fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
mutex_exit(&buf->b_hdr->b_freeze_lock);
!ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc) ?
Yes
No
panic("buffer modified while frozen!");
mutex_exit(&buf->b_hdr->b_freeze_lock);
End

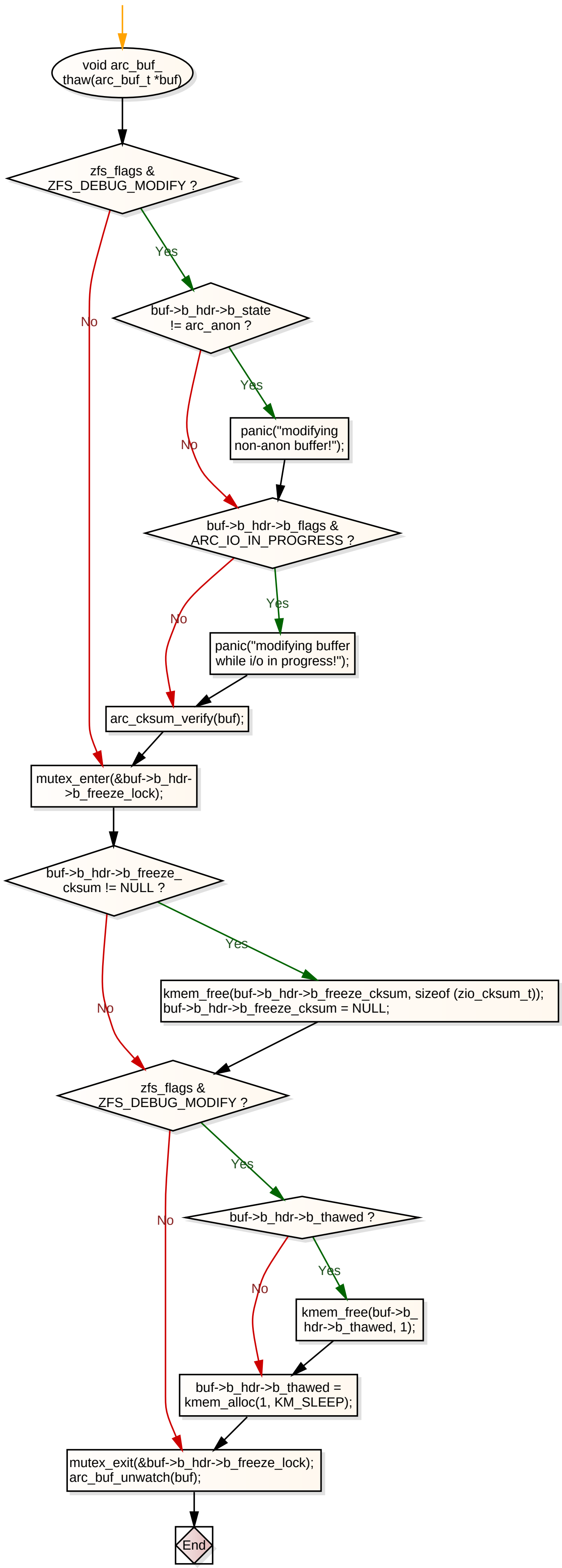

static int
arc_cksum_equal(arc_buf_t
*buf)
zio_cksum_t zc;
int equal;
mutex_enter(&buf->b_hdr->b_freeze_lock);
fletcher_2_native(buf->b_data, buf->b_hdr->b_size, &zc);
equal = ZIO_CHECKSUM_EQUAL(*buf->b_hdr->b_freeze_cksum, zc);
mutex_exit(&buf->b_hdr->b_freeze_lock);
return (equal);
End

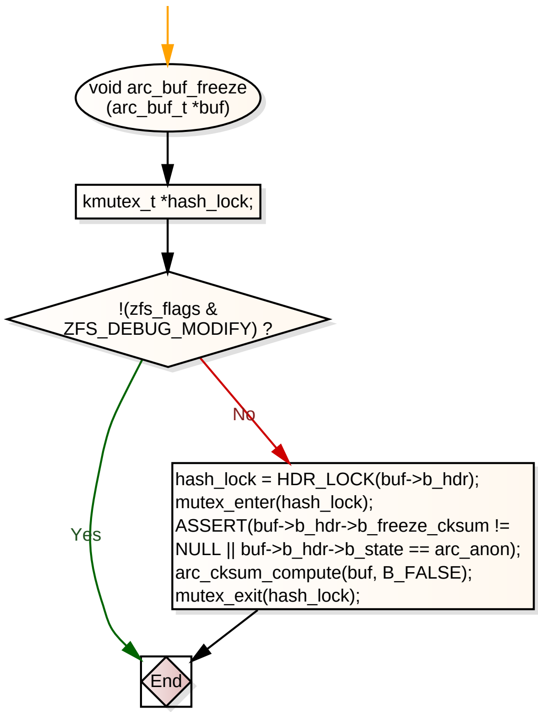

static void arc_cksum_compute(arc_buf_t *buf, boolean_t force)
!force && !(zfs_flags & ZFS_DEBUG_MODIFY) ?
No
Yes
mutex_enter(&buf->b_hdr->b_freeze_lock);
buf->b_hdr->b_freeze_cksum != NULL ?
No
Yes
buf->b_hdr->b_freeze_cksum = kmem_alloc(sizeof (zio_cksum_t), KM_SLEEP);
fletcher_2_native(buf->b_data, buf->b_hdr->b_size, buf->b_hdr->b_freeze_cksum);
mutex_exit(&buf->b_hdr->b_freeze_lock);
arc_buf_watch(buf);
mutex_exit(&buf->b_hdr->b_freeze_lock);
End

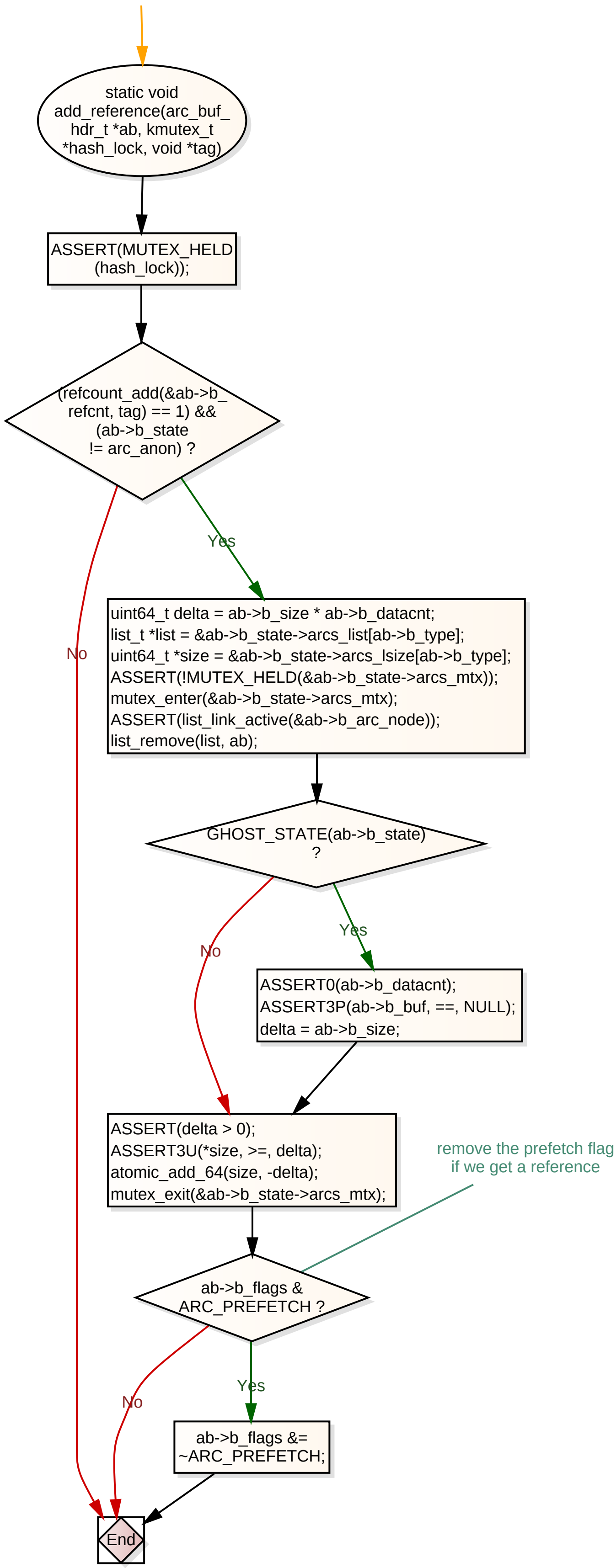

typedef struct procctl
long cmd;
prwatch_t prwatch;
End

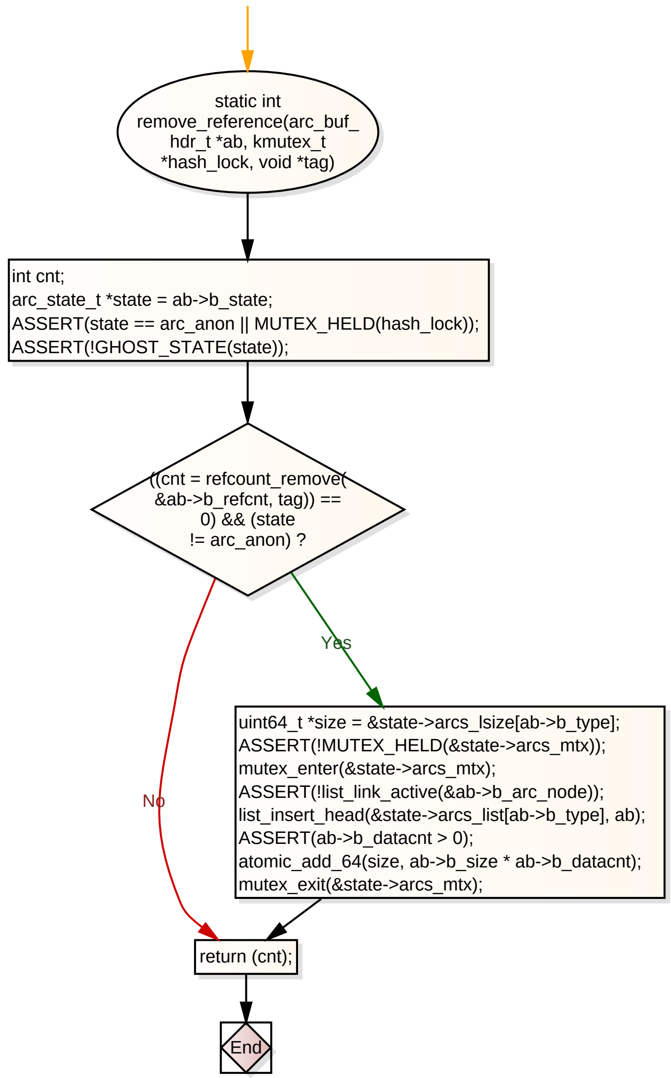

static void
arc_buf_unwatch(arc_buf_t
*buf)
ARGSUSED
#ifndef _KERNEL ?
Yes
arc_watch ?
Yes
No
No
int result;
procctl_t ctl;
ctl.cmd = PCWATCH;
ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
ctl.prwatch.pr_size = 0;
ctl.prwatch.pr_wflags = 0;
result = write(arc_procfd, &ctl, sizeof (ctl));
ASSERT3U(result, ==, sizeof (ctl));
End

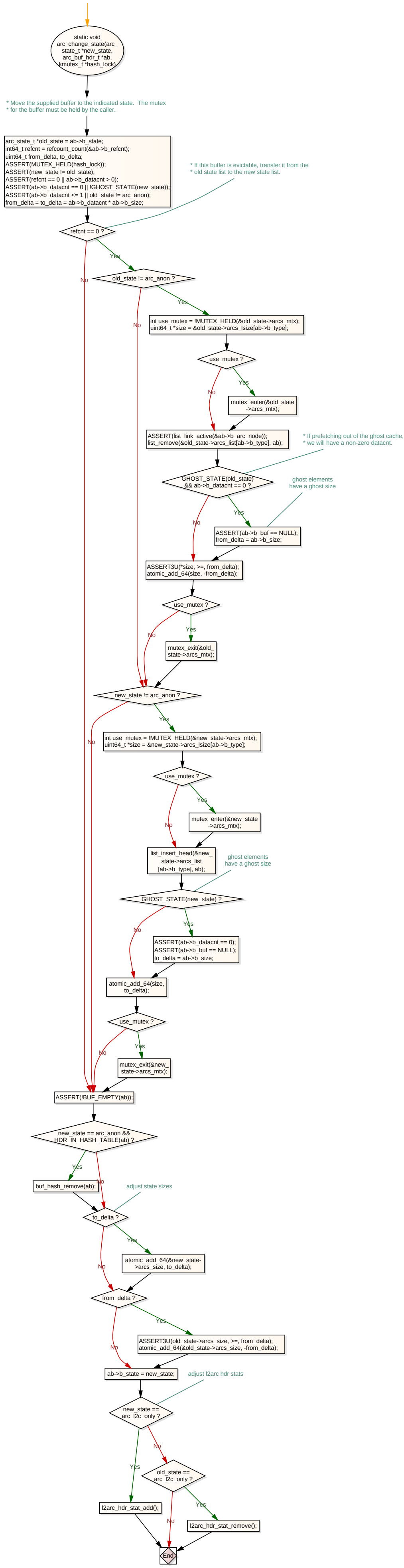

static void
arc_buf_watch(arc_buf_t
*buf)
ARGSUSED
#ifndef _KERNEL ?
Yes
arc_watch ?
Yes
No
No
int result;
procctl_t ctl;
ctl.cmd = PCWATCH;
ctl.prwatch.pr_vaddr = (uintptr_t)buf->b_data;
ctl.prwatch.pr_size = buf->b_hdr->b_size;
ctl.prwatch.pr_wflags = WA_WRITE;
result = write(arc_procfd, &ctl, sizeof (ctl));
ASSERT3U(result, ==, sizeof (ctl));
End


void arc_buf_
thaw(arc_buf_t *buf)
zfs_flags &
ZFS_DEBUG_MODIFY ?
Yes
No
buf->b_hdr->b_state
!= arc_anon ?
Yes
No
panic("modifying
non-anon buffer!");
buf->b_hdr->b_flags &
ARC_IO_IN_PROGRESS ?
Yes
No
panic("modifying buffer
while i/o in progress!");
arc_cksum_verify(buf);
mutex_enter(&buf->b_hdr-
>b_freeze_lock);
buf->b_hdr->b_freeze_
cksum != NULL ?
Yes
No
kmem_free(buf->b_hdr->b_freeze_cksum, sizeof (zio_cksum_t));
buf->b_hdr->b_freeze_cksum = NULL;
zfs_flags &
ZFS_DEBUG_MODIFY ?
Yes
No
buf->b_hdr->b_thawed ?
Yes
No
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);
End

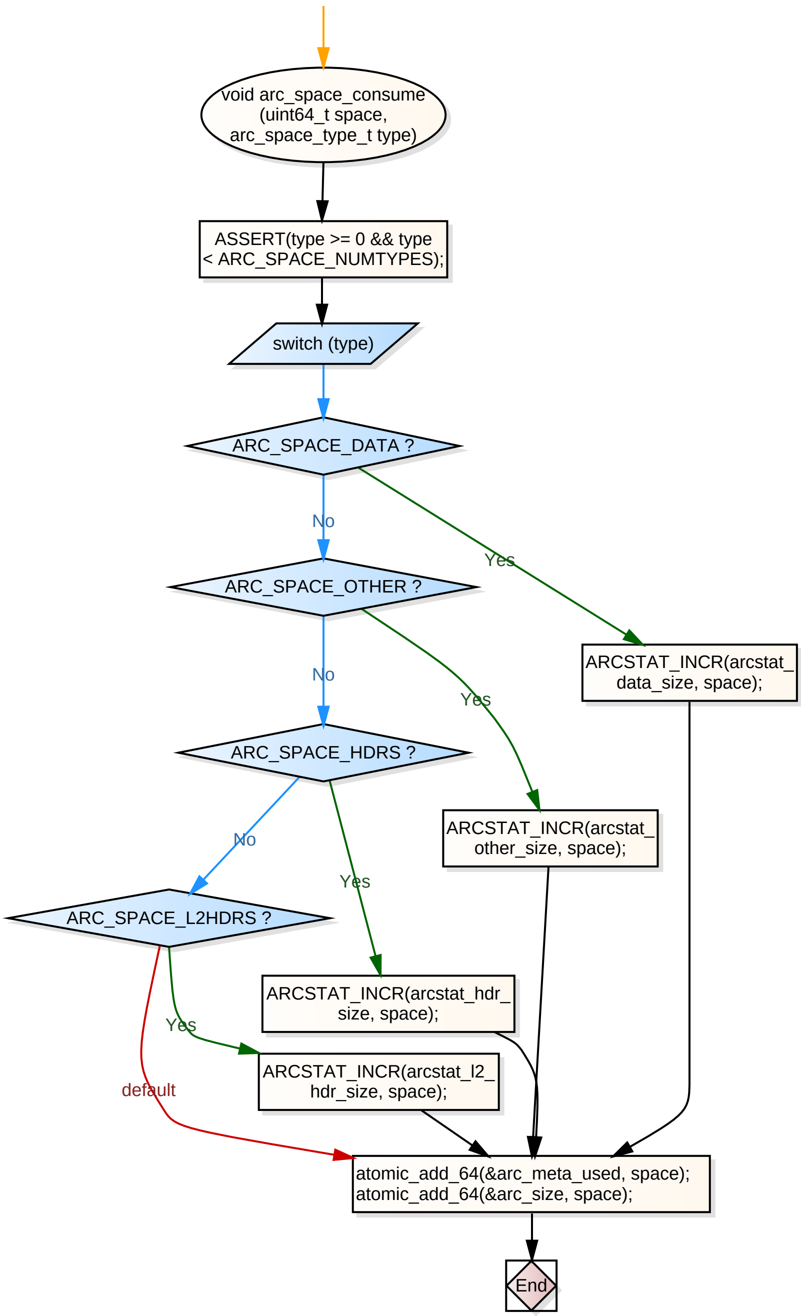

void arc_buf_freeze
(arc_buf_t *buf)
kmutex_t *hash_lock;
!(zfs_flags &
ZFS_DEBUG_MODIFY) ?
Yes
No
hash_lock = HDR_LOCK(buf->b_hdr);
mutex_enter(hash_lock);
ASSERT(buf->b_hdr->b_freeze_cksum !=
NULL || buf->b_hdr->b_state == arc_anon);
arc_cksum_compute(buf, B_FALSE);
mutex_exit(hash_lock);
End


static void add_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
ASSERT(MUTEX_HELD (hash_lock));
(refcount_add(&ab->b_refcnt, tag) == 1) && (ab->b_state != arc_anon) ?
Yes
No
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);
GHOST_STATE(ab->b_state) ?
Yes
No
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
ab->b_flags & ARC_PREFETCH ?
Yes
No
ab->b_flags &= ~ARC_PREFETCH;
End

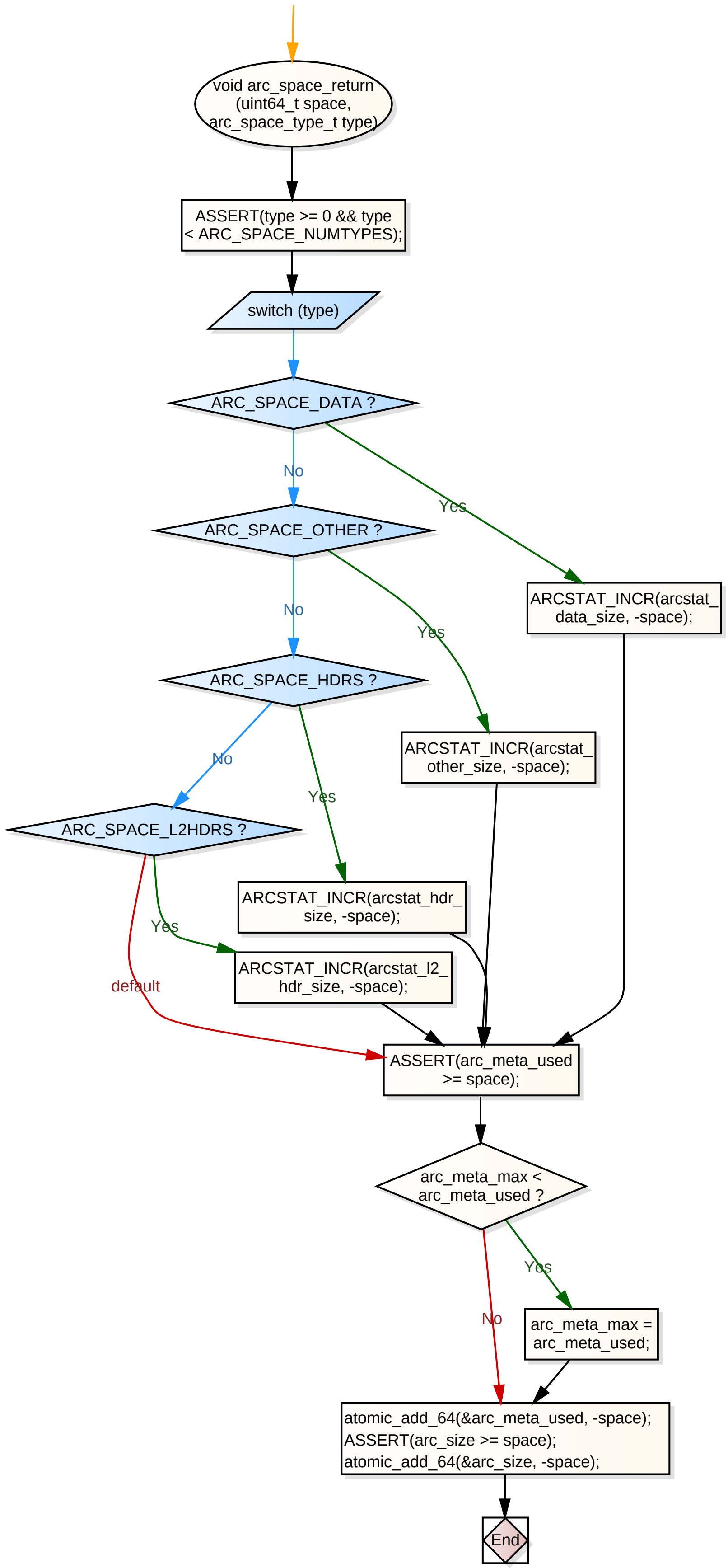

static int remove_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
int cnt;
arc_state_t *state = ab->b_state;
ASSERT(state == arc_anon || MUTEX_HELD(hash_lock));
ASSERT(!GHOST_STATE(state));
((cnt = refcount_remove(&ab->b_refcnt, tag)) == 0) && (state != arc_anon) ?
Yes
No
uint64_t *size = &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);
End

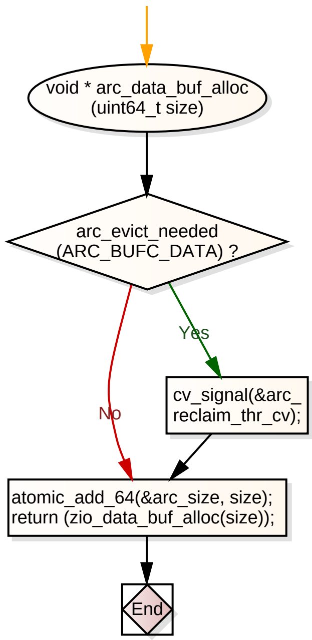


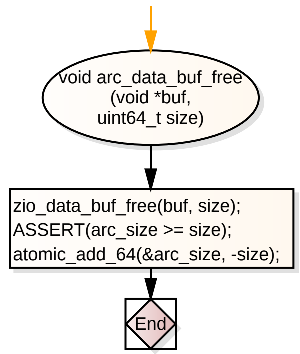
static void arc_change_state(arc_state_t *new_state, arc_buf_hdr_t *ab, kmutex_t *hash_lock)
* Move the supplied buffer to the indicated state. The mutex
* for the buffer must be held by the caller.
arc_state_t *old_state = ab->b_state;
int64_t refcnt = refcount_count(&ab->b_refcnt);
uint64_t from_delta, to_delta;
ASSERT(MUTEX_HELD(hash_lock));
ASSERT(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.
refcnt == 0 ?
Yes
No
old_state != arc_anon ?
Yes
No
int use_mutex = !MUTEX_HELD(&old_state->arcs_mtx);
uint64_t *size = &old_state->arcs_lsize[ab->b_type];
use_mutex ?
Yes
No
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.
GHOST_STATE(old_state) && ab->b_datacnt == 0 ?
ghost elements have a ghost size
Yes
No
ASSERT(ab->b_buf == NULL);
from_delta = ab->b_size;
ASSERT3U(*size, >=, from_delta);
atomic_add_64(size, -from_delta);
use_mutex ?
Yes
No
mutex_exit(&old_state->arcs_mtx);
new_state != arc_anon ?
Yes
No
int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
uint64_t *size = &new_state->arcs_lsize[ab->b_type];
use_mutex ?
Yes
No
mutex_enter(&new_state->arcs_mtx);
list_insert_head(&new_state->arcs_list[ab->b_type], ab);
ghost elements have a ghost size
GHOST_STATE(new_state) ?
Yes
No
ASSERT(ab->b_datacnt == 0);
ASSERT(ab->b_buf == NULL);
to_delta = ab->b_size;
atomic_add_64(size, to_delta);
use_mutex ?
Yes
No
mutex_exit(&new_state->arcs_mtx);
ASSERT(!BUF_EMPTY(ab));
new_state == arc_anon && HDR_IN_HASH_TABLE(ab) ?
Yes
No
buf_hash_remove(ab);
adjust state sizes
to_delta ?
Yes
No
atomic_add_64(&new_state->arcs_size, to_delta);
from_delta ?
Yes
No
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
new_state == arc_l2c_only ?
Yes
No
old_state == arc_l2c_only ?
Yes
No
l2arc_hdr_stat_add();
l2arc_hdr_stat_remove();
End

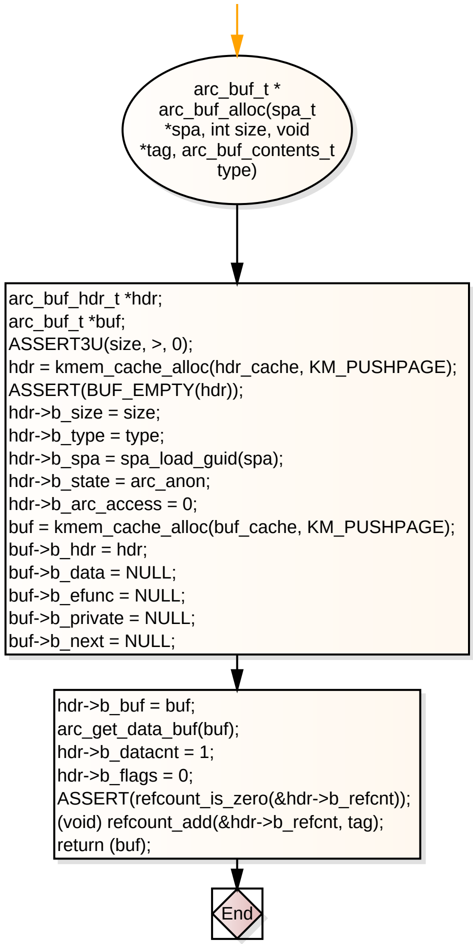


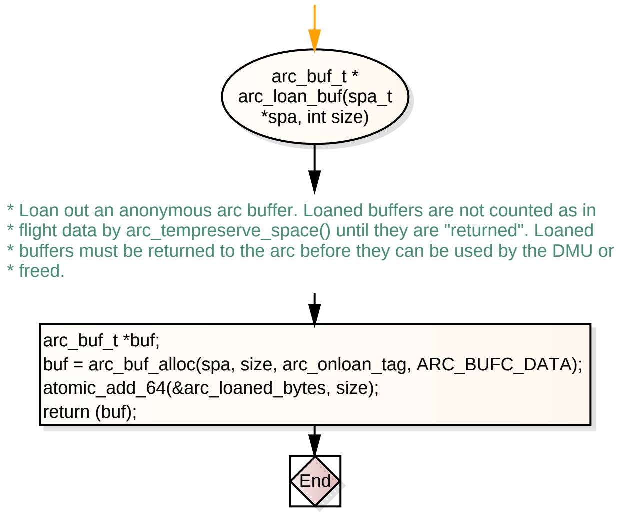
void arc_space_consume
(uint64_t space,
arc_space_type_t type)
ASSERT(type >= 0 && type
< ARC_SPACE_NUMTYPES);
switch (type)
ARC_SPACE_DATA ?
No
Yes
ARC_SPACE_OTHER ?
ARCSTAT_INCR(arcstat_
data_size, space);
No
Yes
ARC_SPACE_HDRS ?
ARCSTAT_INCR(arcstat_
other_size, space);
No
Yes
ARC_SPACE_L2HDRS ?
ARCSTAT_INCR(arcstat_hdr_
size, space);
Yes
ARCSTAT_INCR(arcstat_l2_
hdr_size, space);
default
atomic_add_64(&arc_meta_used, space);
atomic_add_64(&arc_size, space);
End

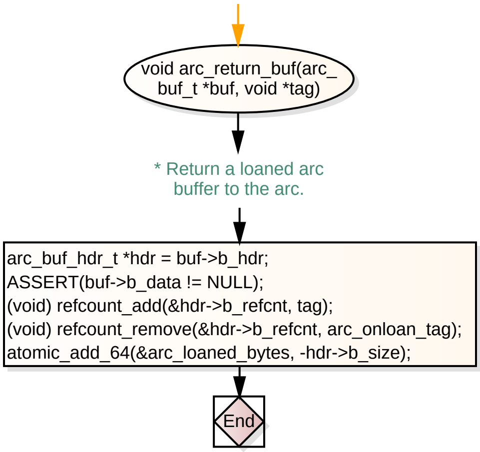

void arc_space_return (uint64_t space, arc_space_type_t type)
ASSERT(type >= 0 && type < ARC_SPACE_NUMTYPES);
switch (type)
ARC_SPACE_DATA ?
No
Yes
ARC_SPACE_OTHER ?
ARCSTAT_INCR(arcstat_data_size, -space);
No
Yes
ARC_SPACE_HDRS ?
ARCSTAT_INCR(arcstat_other_size, -space);
No
Yes
ARC_SPACE_L2HDRS ?
ARCSTAT_INCR(arcstat_hdr_size, -space);
Yes
ARCSTAT_INCR(arcstat_l2_hdr_size, -space);
default
ASSERT(arc_meta_used >= space);
arc_meta_max < arc_meta_used ?
Yes
No
arc_meta_max = arc_meta_used;
atomic_add_64(&arc_meta_used, -space);
ASSERT(arc_size >= space);
atomic_add_64(&arc_size, -space);
End

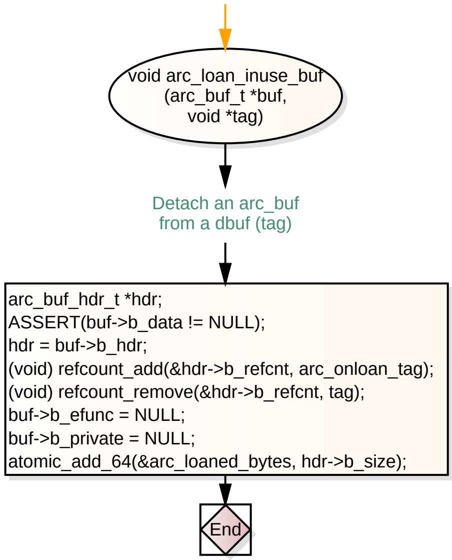

void * arc_data_buf_alloc
(uint64_t size)
arc_evict_needed
(ARC_BUFC_DATA) ?
Yes
No
cv_signal(&arc_
reclaim_thr_cv);
atomic_add_64(&arc_size, size);
return (zio_data_buf_alloc(size));
End

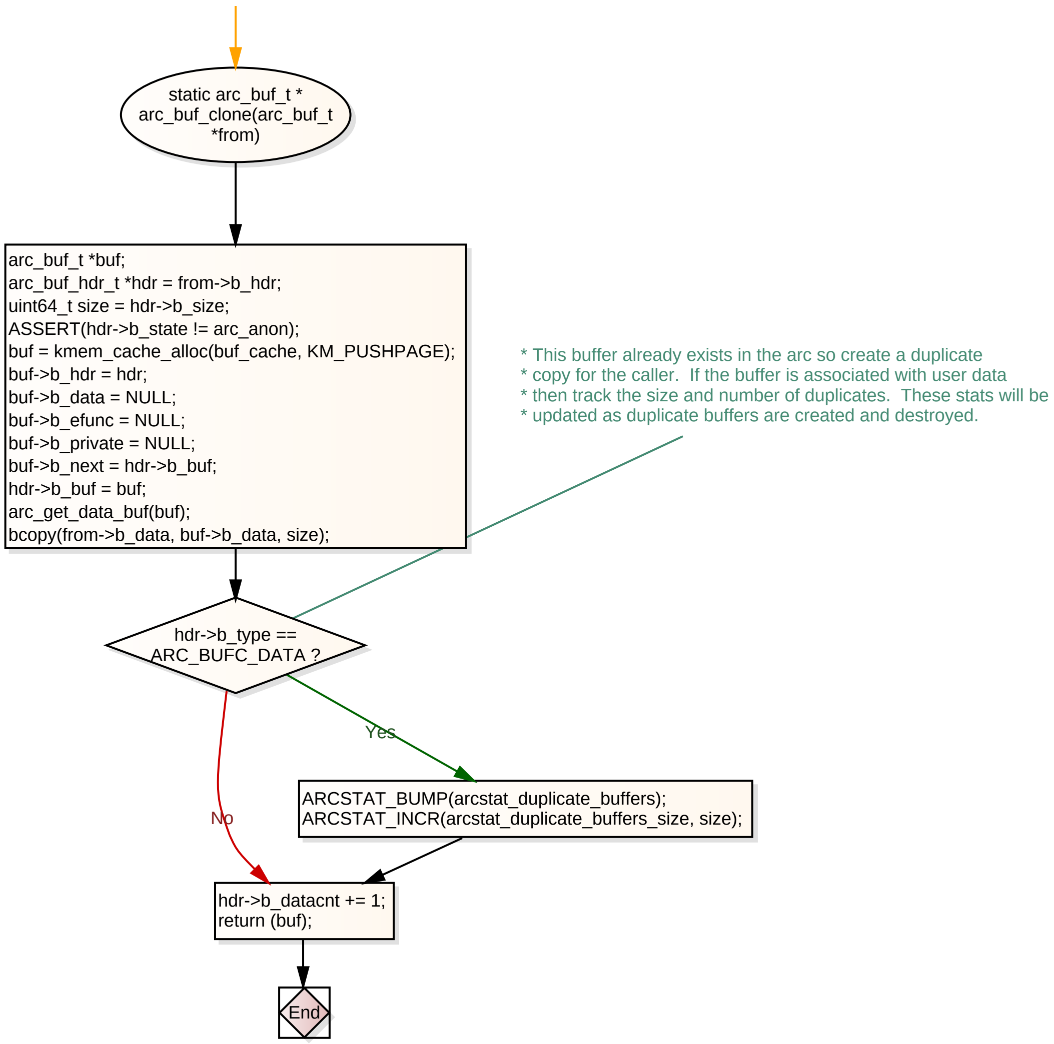

void arc_data_buf_free
(void *buf,
uint64_t size)
zio_data_buf_free(buf, size);
ASSERT(arc_size >= size);
atomic_add_64(&arc_size, -size);
End

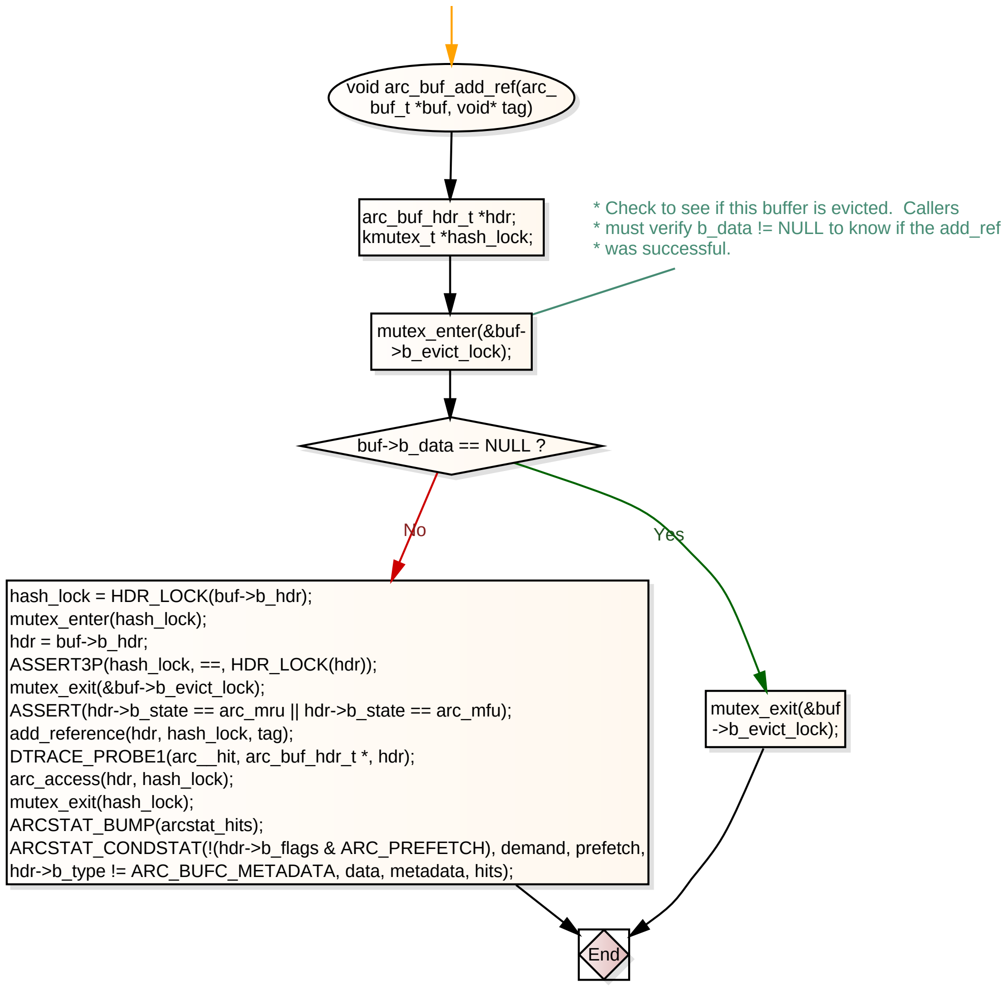

arc_buf_t * arc_buf_alloc(spa_t *spa, int size, void *tag, arc_buf_contents_t type)
arc_buf_hdr_t *hdr;
arc_buf_t *buf;
ASSERT3U(size, >, 0);
hdr = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
ASSERT(BUF_EMPTY(hdr));
hdr->b_size = size;
hdr->b_type = type;
hdr->b_spa = spa_load_guid(spa);
hdr->b_state = 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);
End

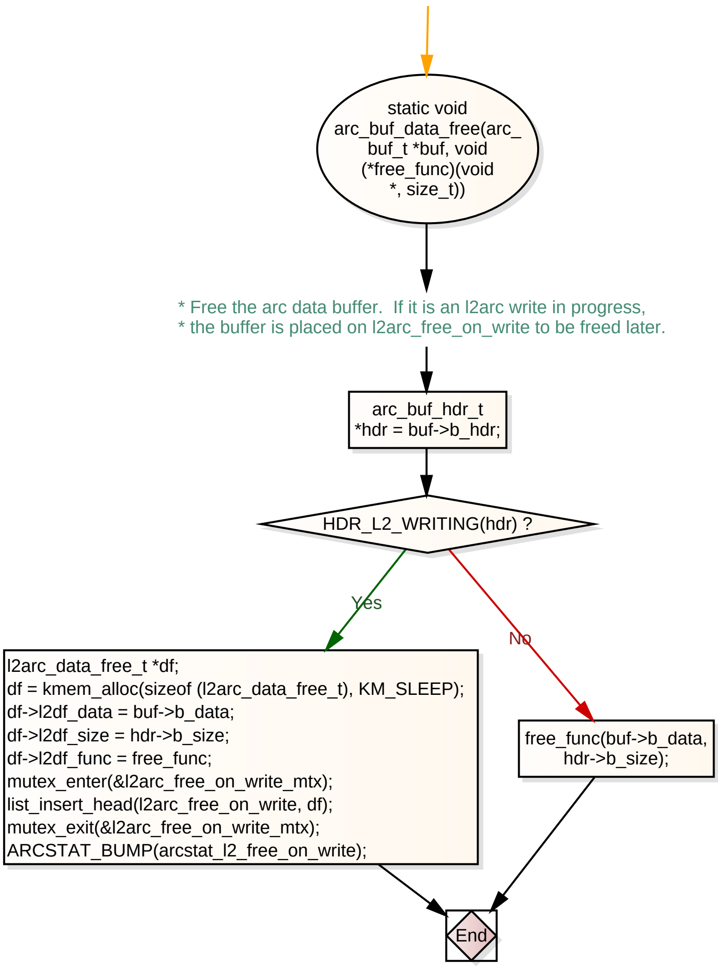

arc_buf_t *
arc_loan_buf(spa_t
*spa, int size)
* Loan out an anonymous arc buffer. Loaned buffers are not counted as in
* flight data by arc_tempreserve_space() until they are "returned". Loaned
* buffers must be returned to the arc before they can be used by the DMU or
* freed.
arc_buf_t *buf;
buf = arc_buf_alloc(spa, size, arc_onloan_tag, ARC_BUFC_DATA);
atomic_add_64(&arc_loaned_bytes, size);
return (buf);
End

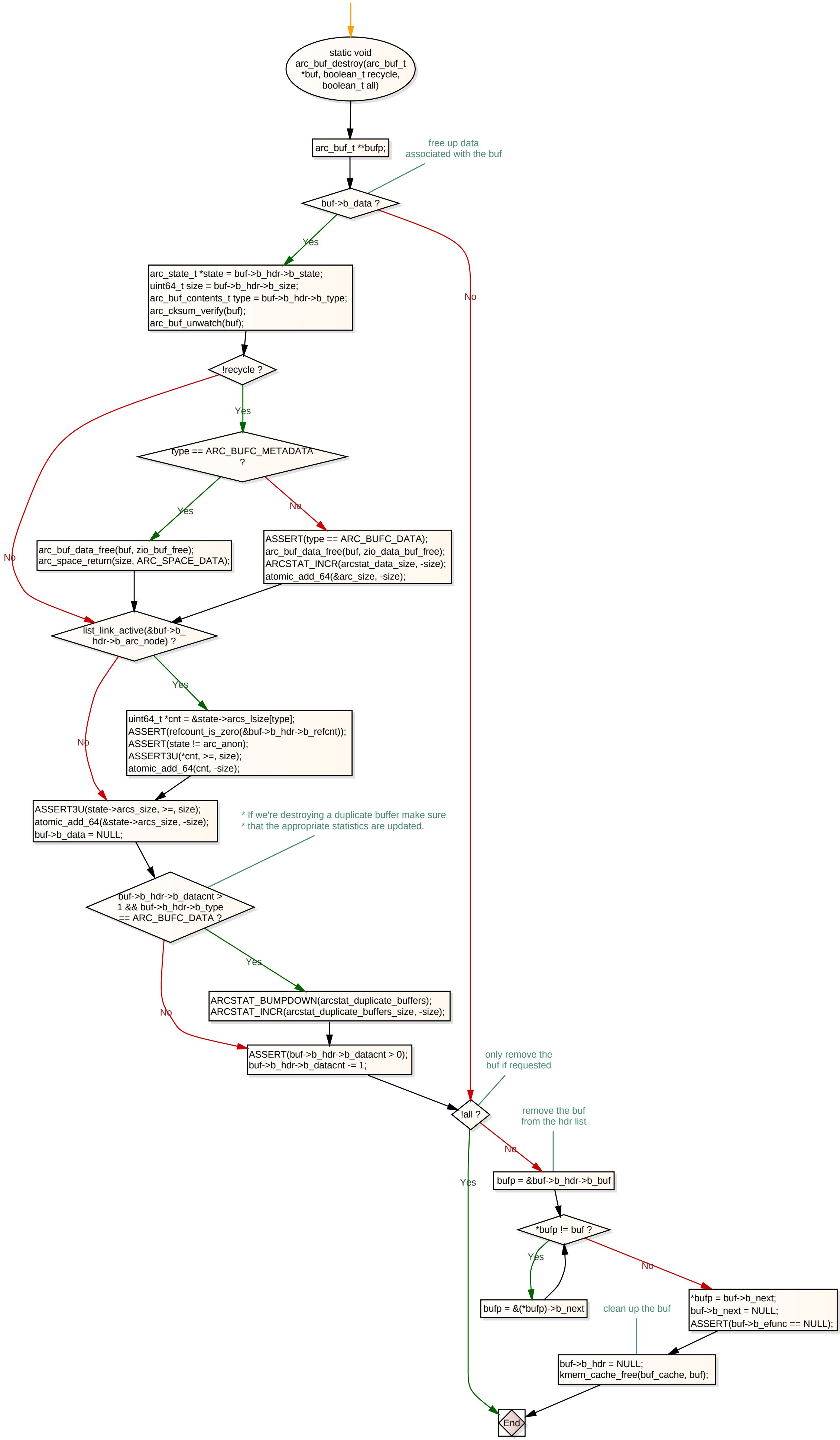

void arc_return_buf(arc_
buf_t *buf, void *tag)
* Return a loaned arc
buffer to the arc.
arc_buf_hdr_t *hdr = buf->b_hdr;
ASSERT(buf->b_data != NULL);
(void) refcount_add(&hdr->b_refcnt, tag);
(void) refcount_remove(&hdr->b_refcnt, arc_onloan_tag);
atomic_add_64(&arc_loaned_bytes, -hdr->b_size);
End

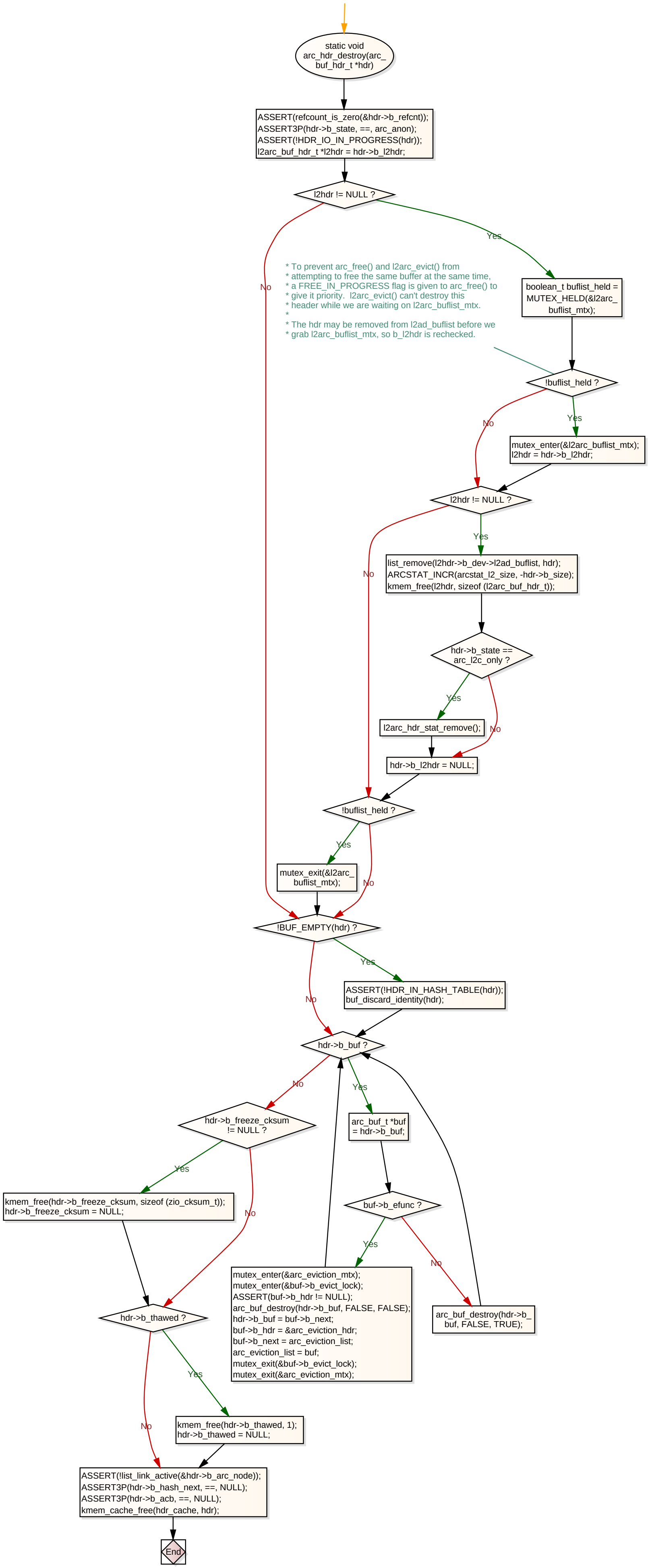

void arc_loan_inuse_buf
(arc_buf_t *buf,
void *tag)
Detach an arc_buf
from a dbuf (tag)
arc_buf_hdr_t *hdr;
ASSERT(buf->b_data != NULL);
hdr = buf->b_hdr;
(void) refcount_add(&hdr->b_refcnt, arc_onloan_tag);
(void) refcount_remove(&hdr->b_refcnt, tag);
buf->b_efunc = NULL;
buf->b_private = NULL;
atomic_add_64(&arc_loaned_bytes, hdr->b_size);
End


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.
hdr->b_type == ARC_BUFC_DATA ?
Yes
No
ARCSTAT_BUMP(arcstat_duplicate_buffers);
ARCSTAT_INCR(arcstat_duplicate_buffers_size, size);
hdr->b_datacnt += 1;
return (buf);
End

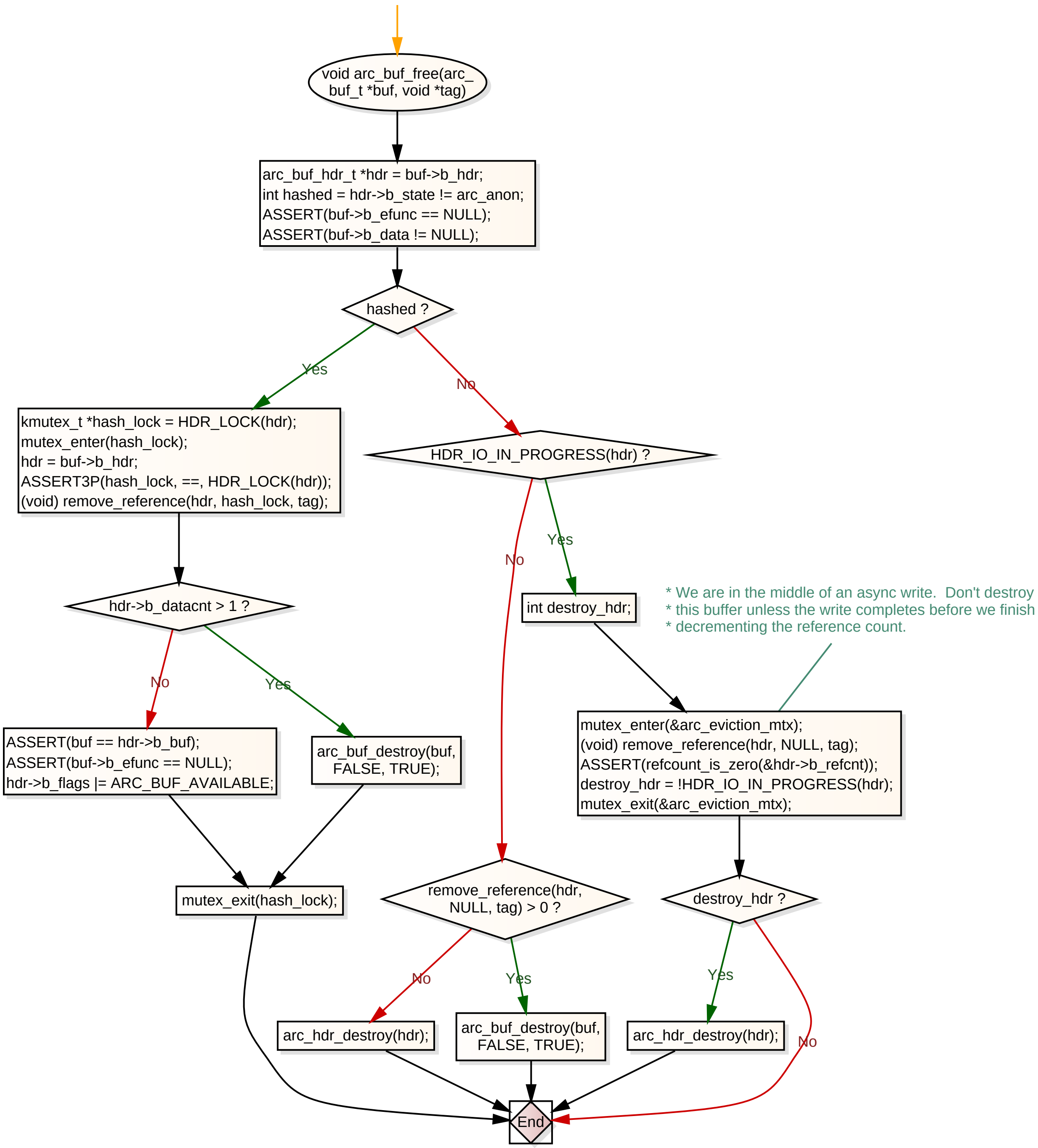

void arc_buf_add_ref(arc_buf_t *buf, void* tag)
arc_buf_hdr_t *hdr;
kmutex_t *hash_lock;
* Check to see if this buffer is evicted. Callers
* must verify b_data != NULL to know if the add_ref
* was successful.
mutex_enter(&buf->b_evict_lock);
buf->b_data == NULL ?
No
Yes
hash_lock = HDR_LOCK(buf->b_hdr);
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);
mutex_exit(&buf->b_evict_lock);
End

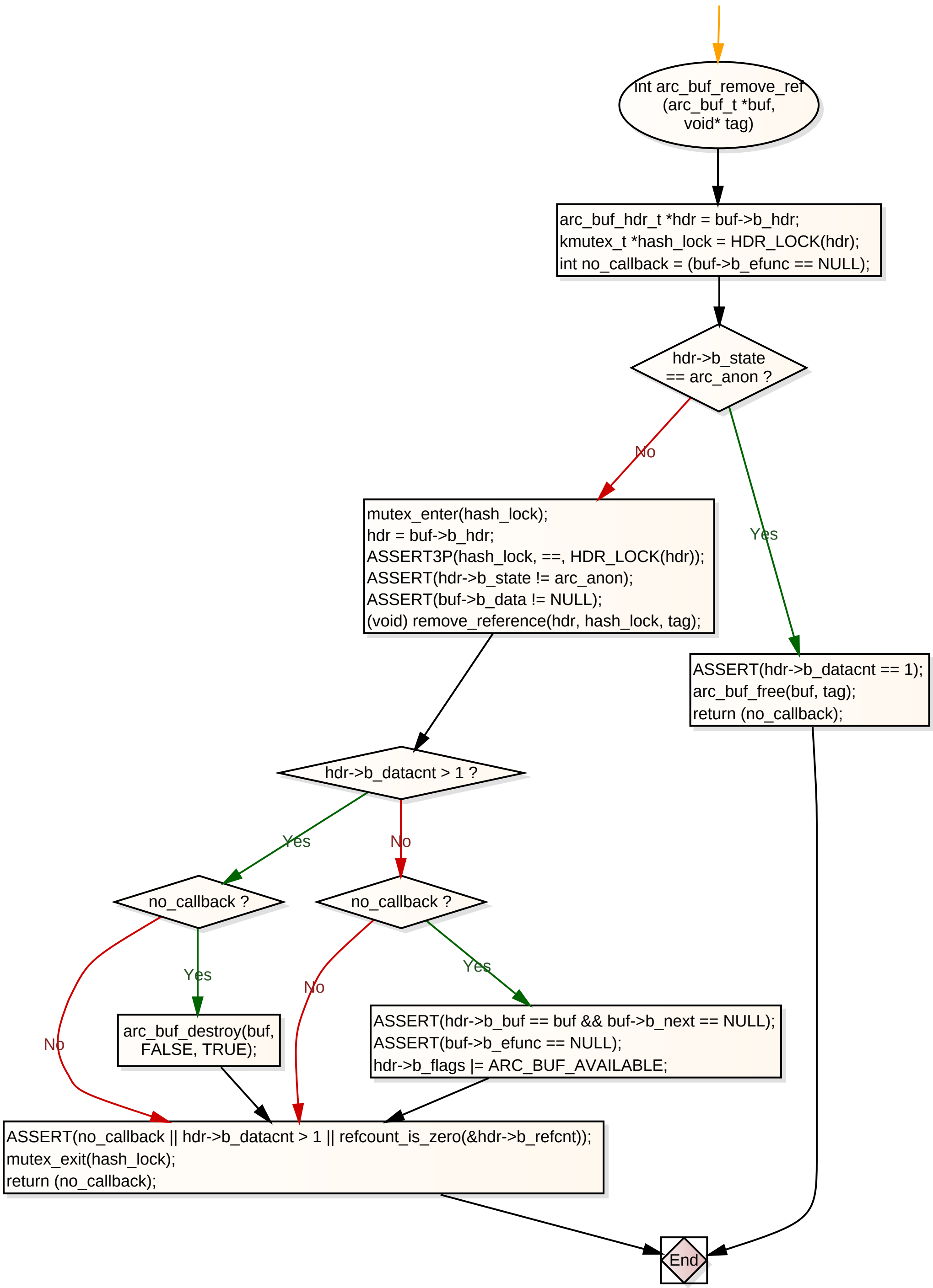

static void arc_buf_data_free(arc_buf_t *buf, void (*free_func)(void *, size_t))
* Free the arc data buffer. If it is an l2arc write in progress,
* the buffer is placed on l2arc_free_on_write to be freed later.
arc_buf_hdr_t *hdr = buf->b_hdr;
HDR_L2_WRITING(hdr) ?
Yes
No
l2arc_data_free_t *df;
df = 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);
free_func(buf->b_data, hdr->b_size);
End

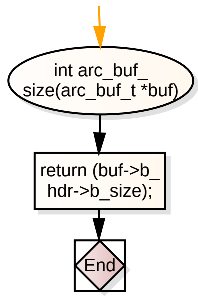

static void arc_buf_destroy(arc_buf_t *buf, boolean_t recycle, boolean_t all)
arc_buf_t **bufp;
free up data associated with the buf
buf->b_data ?
Yes
No
arc_state_t *state = buf->b_hdr->b_state;
uint64_t size = buf->b_hdr->b_size;
arc_buf_contents_t type = buf->b_hdr->b_type;
arc_cksum_verify(buf);
arc_buf_unwatch(buf);
!recycle ?
Yes
No
type == ARC_BUFC_METADATA ?
Yes
No
arc_buf_data_free(buf, zio_buf_free);
arc_space_return(size, ARC_SPACE_DATA);
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);
list_link_active(&buf->b_hdr->b_arc_node) ?
Yes
No
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.
buf->b_hdr->b_datacnt > 1 && buf->b_hdr->b_type == ARC_BUFC_DATA ?
Yes
No
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
!all ?
No
Yes
remove the buf from the hdr list
bufp = &buf->b_hdr->b_buf
*bufp != buf ?
Yes
No
bufp = &(*bufp)->b_next
*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);
End

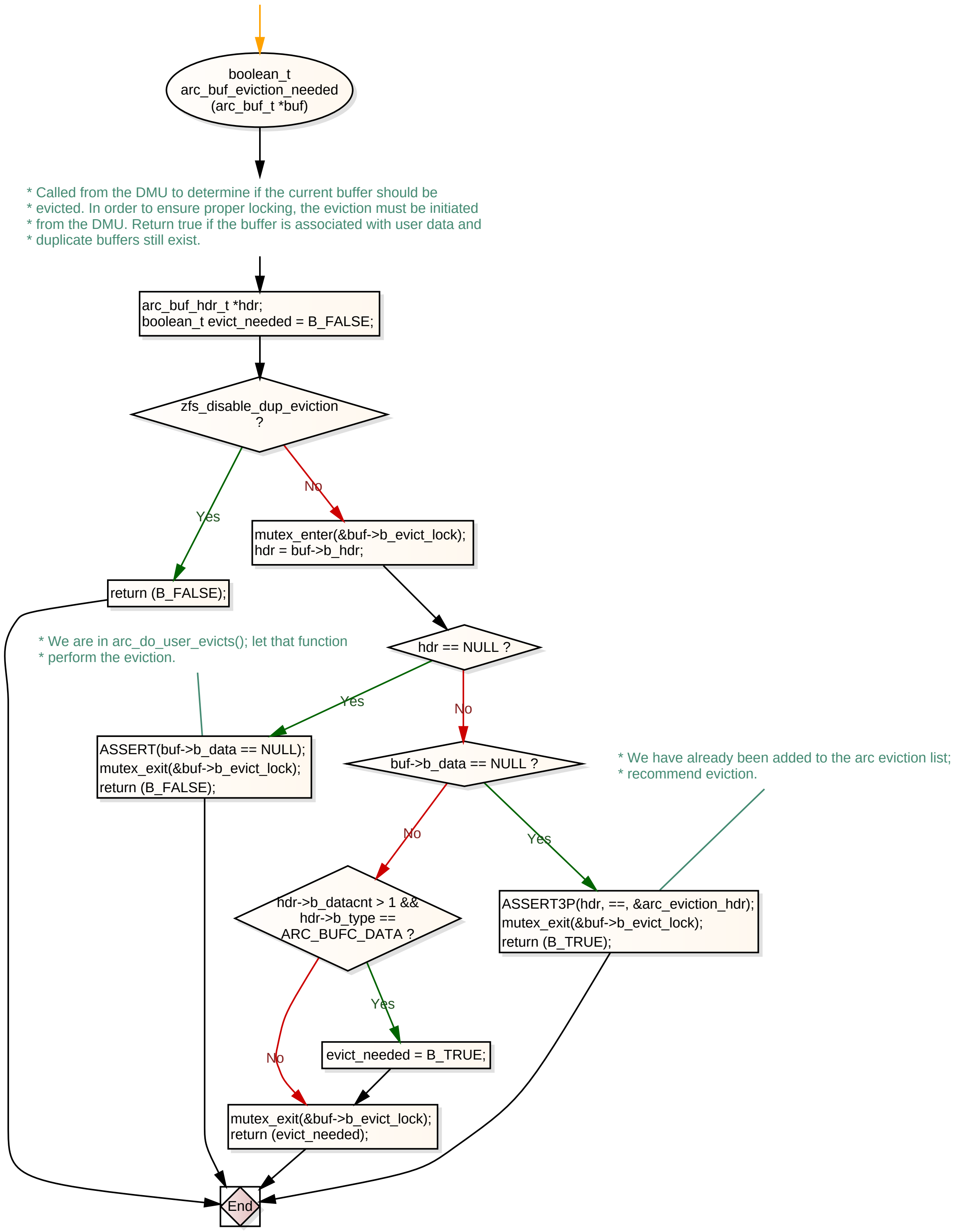


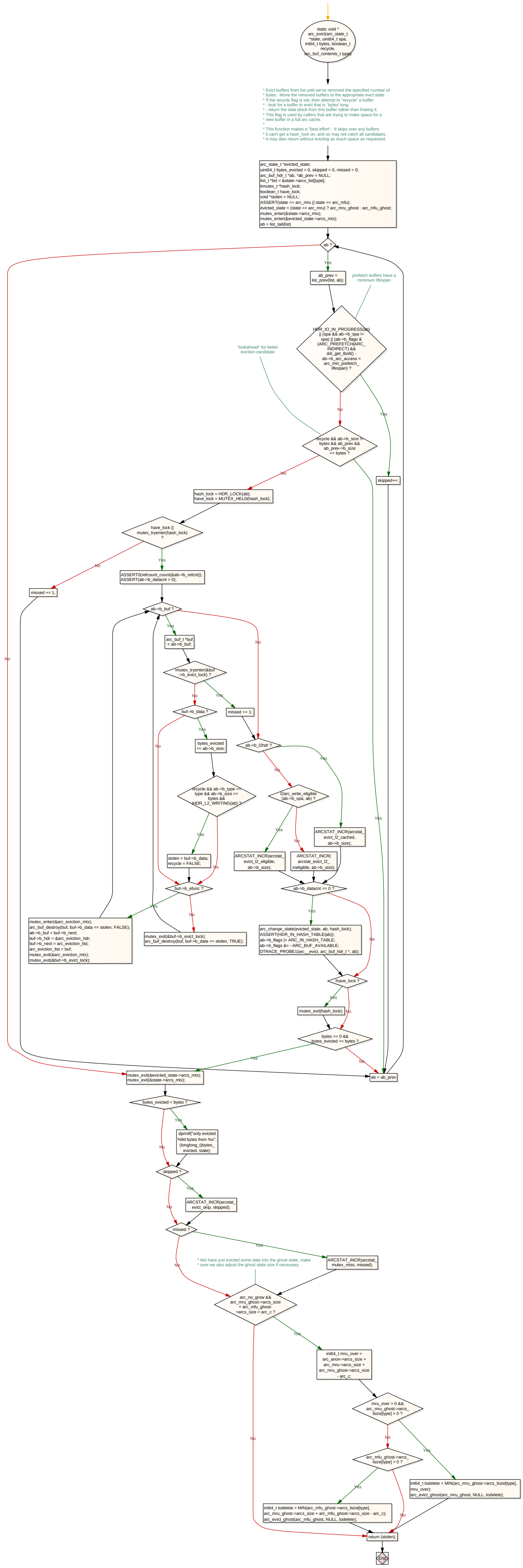
static void arc_hdr_destroy(arc_buf_hdr_t *hdr)
ASSERT(refcount_is_zero(&hdr->b_refcnt));
ASSERT3P(hdr->b_state, ==, arc_anon);
ASSERT(!HDR_IO_IN_PROGRESS(hdr));
l2arc_buf_hdr_t *l2hdr = hdr->b_l2hdr;
l2hdr != NULL ?
Yes
No
* To prevent arc_free() and l2arc_evict() from
* attempting to free the same 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.
boolean_t buflist_held = MUTEX_HELD(&l2arc_buflist_mtx);
!buflist_held ?
No
Yes
mutex_enter(&l2arc_buflist_mtx);
l2hdr = hdr->b_l2hdr;
l2hdr != NULL ?
Yes
No
list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
ARCSTAT_INCR(arcstat_l2_size, -hdr->b_size);
kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
hdr->b_state == arc_l2c_only ?
Yes
No
l2arc_hdr_stat_remove();
hdr->b_l2hdr = NULL;
!buflist_held ?
Yes
No
mutex_exit(&l2arc_buflist_mtx);
!BUF_EMPTY(hdr) ?
Yes
No
ASSERT(!HDR_IN_HASH_TABLE(hdr));
buf_discard_identity(hdr);
hdr->b_buf ?
No
Yes
hdr->b_freeze_cksum != NULL ?
arc_buf_t *buf = hdr->b_buf;
Yes
No
kmem_free(hdr->b_freeze_cksum, sizeof (zio_cksum_t));
hdr->b_freeze_cksum = NULL;
buf->b_efunc ?
Yes
No
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);
arc_buf_destroy(hdr->b_buf, FALSE, TRUE);
hdr->b_thawed ?
Yes
No
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);
End


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);
hashed ?
Yes
No
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);
HDR_IO_IN_PROGRESS(hdr) ?
Yes
No
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.
hdr->b_datacnt > 1 ?
No
Yes
ASSERT(buf == hdr->b_buf);
ASSERT(buf->b_efunc == NULL);
hdr->b_flags |= ARC_BUF_AVAILABLE;
arc_buf_destroy(buf, FALSE, TRUE);
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);
mutex_exit(hash_lock);
remove_reference(hdr, NULL, tag) > 0 ?
destroy_hdr ?
No
Yes
Yes
No
arc_hdr_destroy(hdr);
arc_buf_destroy(buf, FALSE, TRUE);
arc_hdr_destroy(hdr);
End

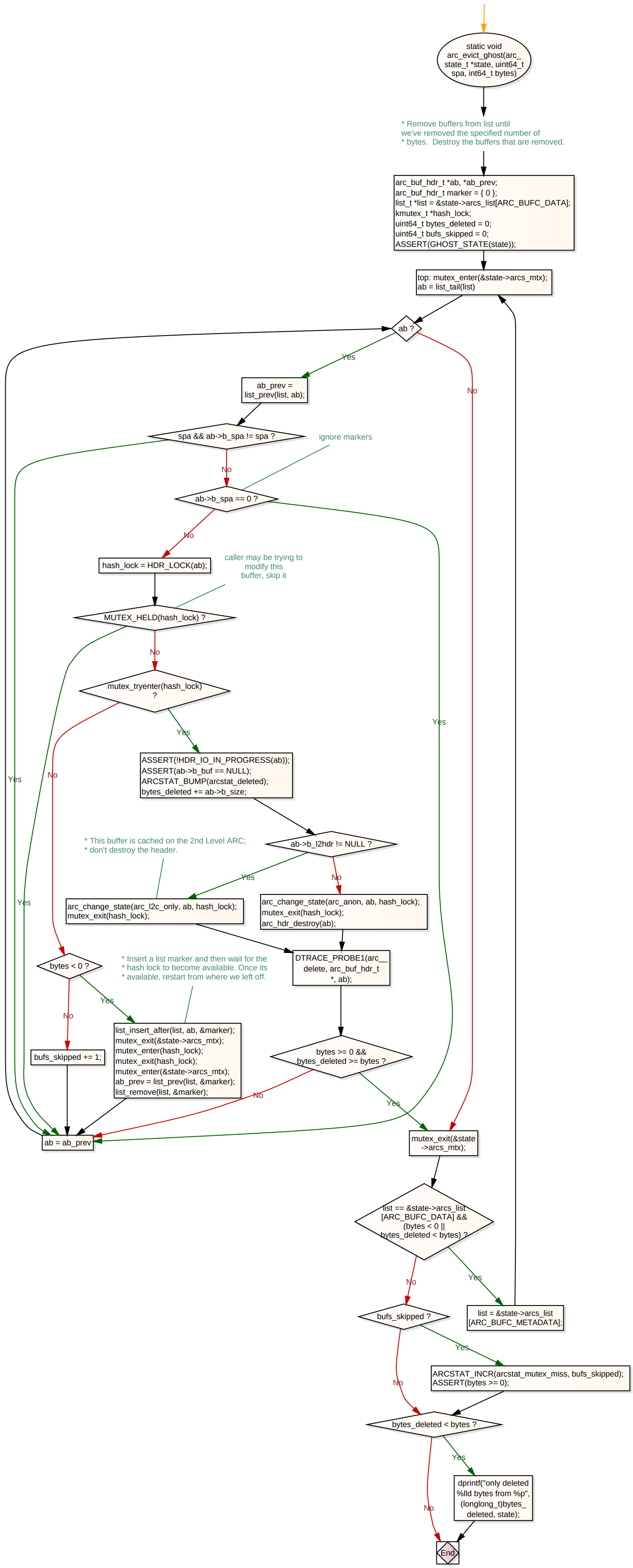

int arc_buf_remove_ref
(arc_buf_t *buf,
void* tag)
arc_buf_hdr_t *hdr = buf->b_hdr;
kmutex_t *hash_lock = HDR_LOCK(hdr);
int no_callback = (buf->b_efunc == NULL);
hdr->b_state
== arc_anon ?
No
Yes
mutex_enter(hash_lock);
hdr = buf->b_hdr;
ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
ASSERT(hdr->b_state != arc_anon);
ASSERT(buf->b_data != NULL);
(void) remove_reference(hdr, hash_lock, tag);
ASSERT(hdr->b_datacnt == 1);
arc_buf_free(buf, tag);
return (no_callback);
hdr->b_datacnt > 1 ?
Yes
No
no_callback ?
no_callback ?
Yes
No
Yes
No
arc_buf_destroy(buf,
FALSE, TRUE);
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);
End


int arc_buf_
size(arc_buf_t *buf)
return (buf->b_
hdr->b_size);
End

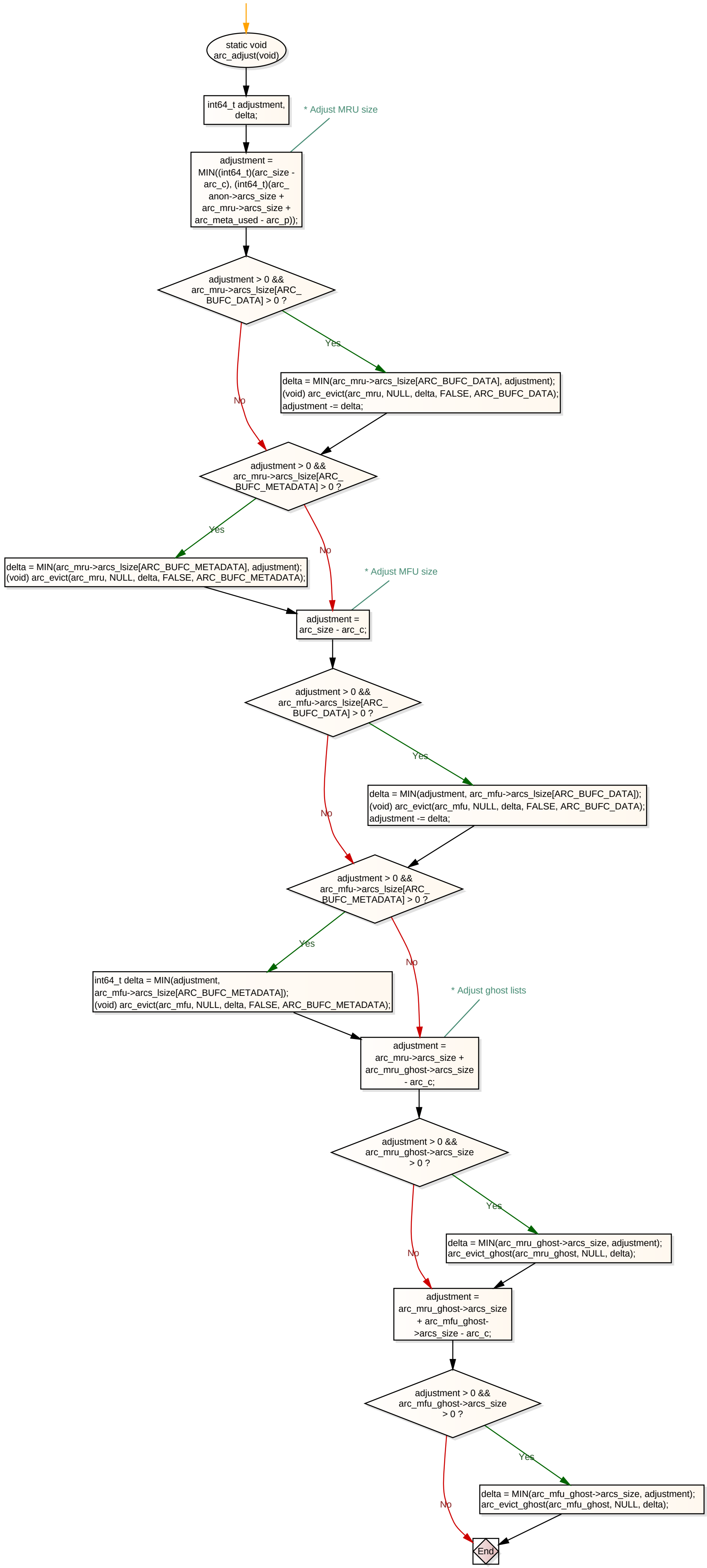

boolean_t
arc_buf_eviction_needed
(arc_buf_t *buf)
* Called from the DMU to determine if the current buffer should be
* evicted. In order to ensure proper locking, the eviction must be initiated
* from the DMU. Return true if the buffer is associated with user data and
* duplicate buffers still exist.
arc_buf_hdr_t *hdr;
boolean_t evict_needed = B_FALSE;
zfs_disable_dup_eviction
?
Yes
No
mutex_enter(&buf->b_evict_lock);
hdr = buf->b_hdr;
return (B_FALSE);
* We are in arc_do_user_evicts(); let that function
* perform the eviction.
hdr == NULL ?
Yes
No
ASSERT(buf->b_data == NULL);
mutex_exit(&buf->b_evict_lock);
return (B_FALSE);
buf->b_data == NULL ?
* We have already been added to the arc eviction list;
* recommend eviction.
No
Yes
hdr->b_datacnt > 1 &&
hdr->b_type ==
ARC_BUFC_DATA ?
ASSERT3P(hdr, ==, &arc_eviction_hdr);
mutex_exit(&buf->b_evict_lock);
return (B_TRUE);
Yes
No
evict_needed = B_TRUE;
mutex_exit(&buf->b_evict_lock);
return (evict_needed);
End

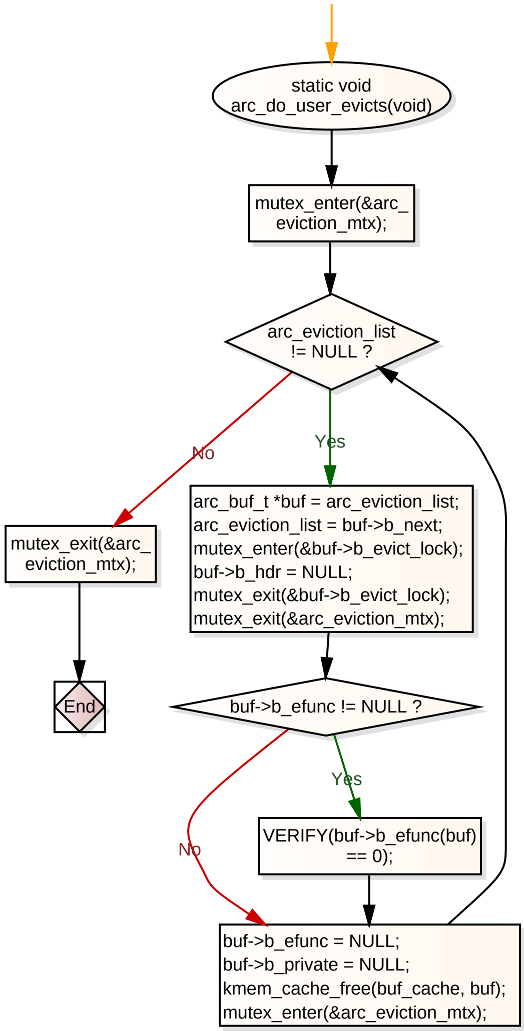


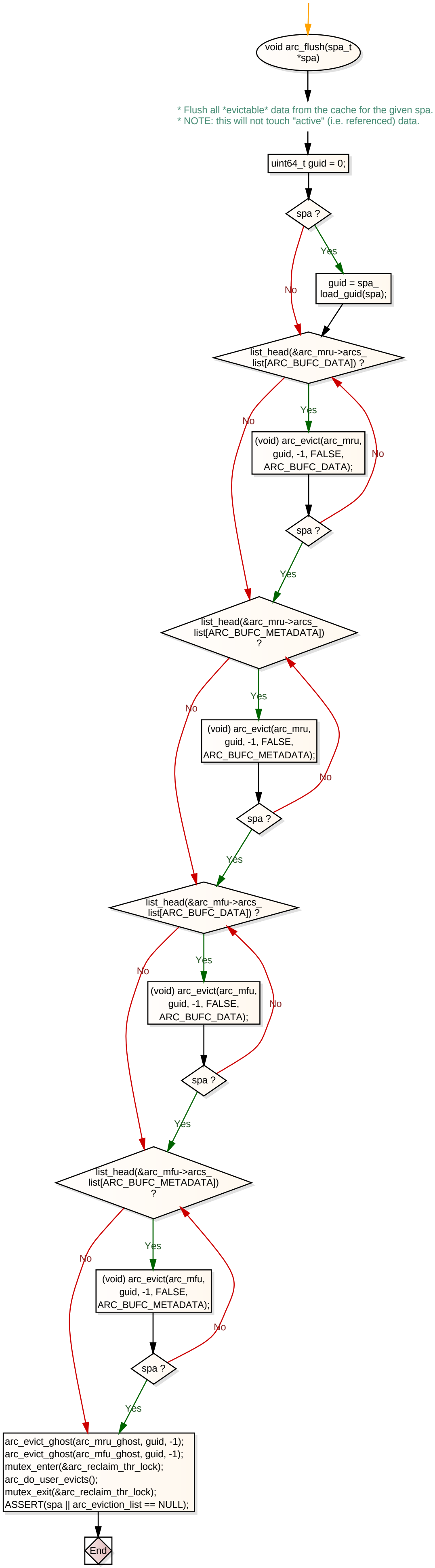
static void * arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle, arc_buf_contents_t type)
* Evict buffers from list until we've removed the specified number of
* bytes. Move the removed buffers to the appropriate evict state.
* If the recycle flag is set, then attempt to "recycle" 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.
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;
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);
ab = list_tail(list)
ab ?
ab_prev = list_prev(list, ab);
prefetch buffers have a minimum lifespan
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) ?
"lookahead" for better eviction candidate
recycle && ab->b_size != bytes && ab_prev && ab_prev->b_size == bytes ?
skipped++;
hash_lock = HDR_LOCK(ab);
have_lock = MUTEX_HELD(hash_lock);
have_lock || mutex_tryenter(hash_lock) ?
missed += 1;
ASSERT0(refcount_count(&ab->b_refcnt));
ASSERT(ab->b_datacnt > 0);
ab->b_buf ?
arc_buf_t *buf = ab->b_buf;
!mutex_tryenter(&buf->b_evict_lock) ?
buf->b_data ?
missed += 1;
bytes_evicted += ab->b_size;
ab->b_l2hdr ?
recycle && ab->b_type == type && ab->b_size == bytes && !HDR_L2_WRITING(ab) ?
l2arc_write_eligible (ab->b_spa, ab) ?
ARCSTAT_INCR(arcstat_evict_l2_cached, ab->b_size);
stolen = buf->b_data;
recycle = FALSE;
ARCSTAT_INCR(arcstat_evict_l2_eligible, ab->b_size);
ARCSTAT_INCR(arcstat_evict_l2_ineligible, ab->b_size);
buf->b_efunc ?
ab->b_datacnt == 0 ?
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);
mutex_exit(&buf->b_evict_lock);
arc_buf_destroy(buf, buf->b_data == stolen, TRUE);
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);
!have_lock ?
mutex_exit(hash_lock);
bytes >= 0 && bytes_evicted >= bytes ?
mutex_exit(&evicted_state->arcs_mtx);
mutex_exit(&state->arcs_mtx);
ab = ab_prev
bytes_evicted < bytes ?
dprintf("only evicted %lld bytes from %x", (longlong_t)bytes_evicted, state);
skipped ?
ARCSTAT_INCR(arcstat_evict_skip, skipped);
missed ?
ARCSTAT_INCR(arcstat_mutex_miss, missed);
* We have just evicted some date into the ghost state, make
* sure we also adjust the ghost state size if necessary.
arc_no_grow && arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size > arc_c ?
int64_t mru_over = arc_anon->arcs_size + arc_mru->arcs_size + arc_mru_ghost->arcs_size - arc_c;
mru_over > 0 && arc_mru_ghost->arcs_lsize[type] > 0 ?
arc_mfu_ghost->arcs_lsize[type] > 0 ?
int64_t todelete = MIN(arc_mru_ghost->arcs_lsize[type], mru_over);
arc_evict_ghost(arc_mru_ghost, NULL, todelete);
int64_t todelete = MIN(arc_mfu_ghost->arcs_lsize[type], arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size - arc_c);
arc_evict_ghost(arc_mfu_ghost, NULL, todelete);
return (stolen);
End

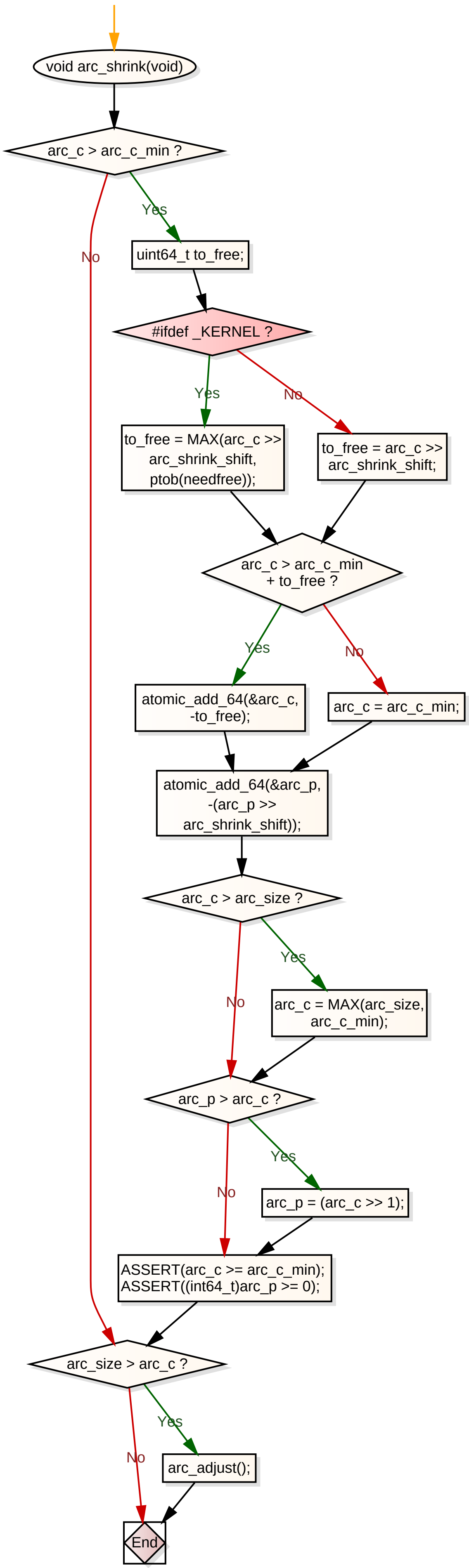


static void arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes)
* Remove buffers from list until we've removed the specified number of
* bytes. Destroy the buffers that are removed.
arc_buf_hdr_t *ab, *ab_prev;
arc_buf_hdr_t marker = { 0 };
list_t *list = &state->arcs_list[ARC_BUFC_DATA];
kmutex_t *hash_lock;
uint64_t bytes_deleted = 0;
uint64_t bufs_skipped = 0;
ASSERT(GHOST_STATE(state));
top: mutex_enter(&state->arcs_mtx);
ab = list_tail(list)
ab ?
Yes
No
ab_prev = list_prev(list, ab);
spa && ab->b_spa != spa ?
ignore markers
No
ab->b_spa == 0 ?
No
hash_lock = HDR_LOCK(ab);
caller may be trying to modify this buffer, skip it
MUTEX_HELD(hash_lock) ?
No
mutex_tryenter(hash_lock) ?
Yes
ASSERT(!HDR_IO_IN_PROGRESS(ab));
ASSERT(ab->b_buf == NULL);
ARCSTAT_BUMP(arcstat_deleted);
bytes_deleted += ab->b_size;
* This buffer is cached on the 2nd Level ARC;
* don't destroy the header.
ab->b_l2hdr != NULL ?
Yes
No
arc_change_state(arc_l2c_only, ab, hash_lock);
mutex_exit(hash_lock);
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);
No
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.
Yes
No
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);
bufs_skipped += 1;
bytes >= 0 && bytes_deleted >= bytes ?
No
Yes
Yes
Yes
Yes
ab = ab_prev
mutex_exit(&state->arcs_mtx);
list == &state->arcs_list[ARC_BUFC_DATA] && (bytes < 0 || bytes_deleted < bytes) ?
No
Yes
bufs_skipped ?
list = &state->arcs_list[ARC_BUFC_METADATA];
Yes
No
ARCSTAT_INCR(arcstat_mutex_miss, bufs_skipped);
ASSERT(bytes >= 0);
bytes_deleted < bytes ?
Yes
No
dprintf("only deleted %lld bytes from %p", (longlong_t)bytes_deleted, state);
End

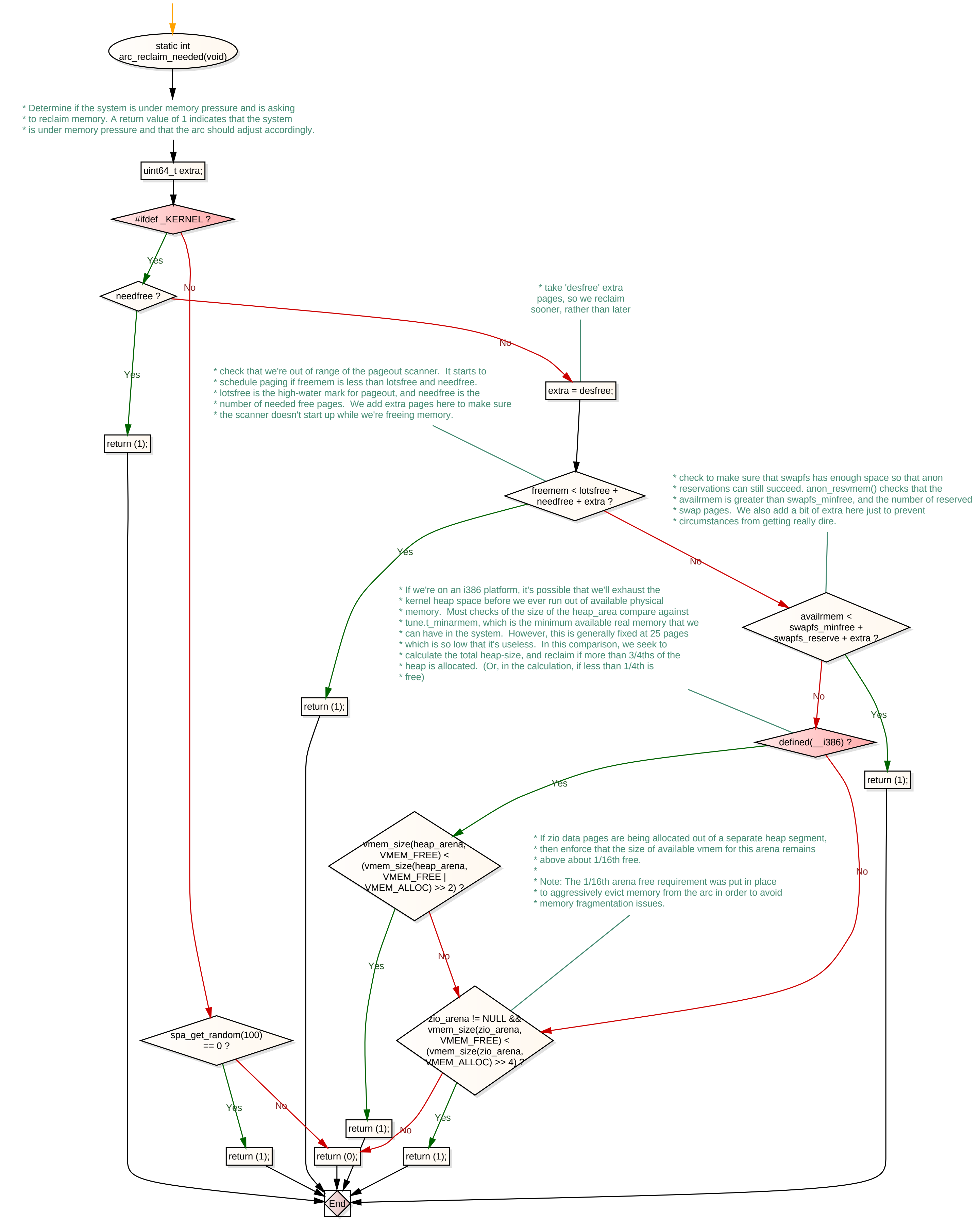

static void arc_adjust(void)
int64_t adjustment, delta;
* Adjust MRU size
adjustment = MIN((int64_t)(arc_size - arc_c), (int64_t)(arc_anon->arcs_size + arc_mru->arcs_size + arc_meta_used - arc_p));
adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_DATA] > 0 ?
Yes
delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_DATA], adjustment);
(void) arc_evict(arc_mru, NULL, delta, FALSE, ARC_BUFC_DATA);
adjustment -= delta;
No
adjustment > 0 && arc_mru->arcs_lsize[ARC_BUFC_METADATA] > 0 ?
Yes
delta = MIN(arc_mru->arcs_lsize[ARC_BUFC_METADATA], adjustment);
(void) arc_evict(arc_mru, NULL, delta, FALSE, ARC_BUFC_METADATA);
No
* Adjust MFU size
adjustment = arc_size - arc_c;
adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_DATA] > 0 ?
Yes
delta = MIN(adjustment, arc_mfu->arcs_lsize[ARC_BUFC_DATA]);
(void) arc_evict(arc_mfu, NULL, delta, FALSE, ARC_BUFC_DATA);
adjustment -= delta;
No
adjustment > 0 && arc_mfu->arcs_lsize[ARC_BUFC_METADATA] > 0 ?
Yes
int64_t delta = MIN(adjustment, arc_mfu->arcs_lsize[ARC_BUFC_METADATA]);
(void) arc_evict(arc_mfu, NULL, delta, FALSE, ARC_BUFC_METADATA);
No
* Adjust ghost lists
adjustment = arc_mru->arcs_size + arc_mru_ghost->arcs_size - arc_c;
adjustment > 0 && arc_mru_ghost->arcs_size > 0 ?
Yes
delta = MIN(arc_mru_ghost->arcs_size, adjustment);
arc_evict_ghost(arc_mru_ghost, NULL, delta);
No
adjustment = arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size - arc_c;
adjustment > 0 && arc_mfu_ghost->arcs_size > 0 ?
Yes
delta = MIN(arc_mfu_ghost->arcs_size, adjustment);
arc_evict_ghost(arc_mfu_ghost, NULL, delta);
No
End

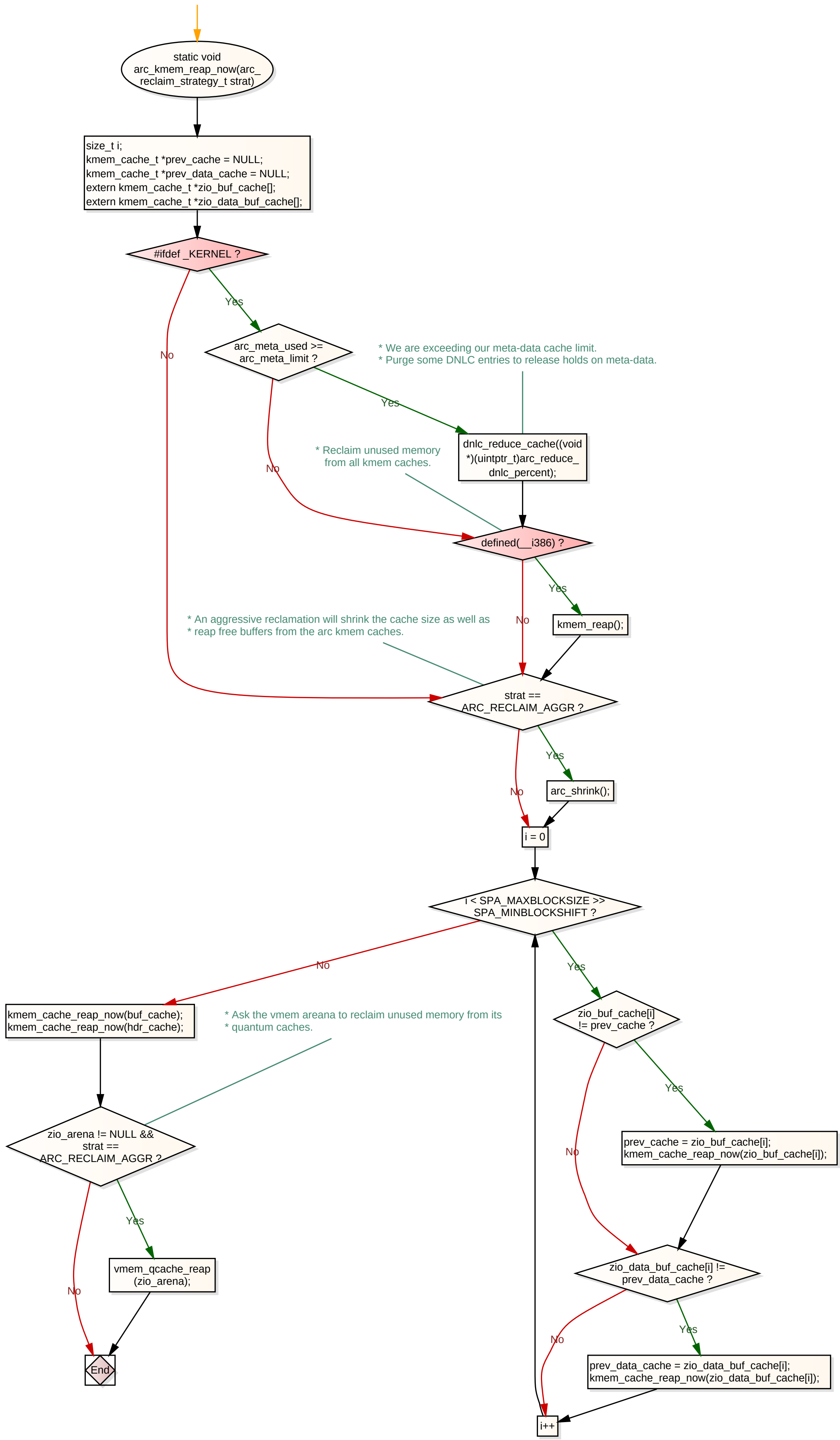

static void
arc_do_user_evicts(void)
mutex_enter(&arc_
eviction_mtx);
arc_eviction_list
!= NULL ?
No
Yes
mutex_exit(&arc_
eviction_mtx);
arc_buf_t *buf = arc_eviction_list;
arc_eviction_list = buf->b_next;
mutex_enter(&buf->b_evict_lock);
buf->b_hdr = NULL;
mutex_exit(&buf->b_evict_lock);
mutex_exit(&arc_eviction_mtx);
End
buf->b_efunc != NULL ?
Yes
No
VERIFY(buf->b_efunc(buf)
== 0);
buf->b_efunc = NULL;
buf->b_private = NULL;
kmem_cache_free(buf_cache, buf);
mutex_enter(&arc_eviction_mtx);


void arc_flush(spa_t *spa)
* Flush all *evictable* data from the cache for the given spa.
* NOTE: this will not touch "active" (i.e. referenced) data.
uint64_t guid = 0;
spa ?
Yes
No
guid = spa_load_guid(spa);
list_head(&arc_mru->arcs_list[ARC_BUFC_DATA]) ?
Yes
No
(void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_DATA);
spa ?
Yes
No
list_head(&arc_mru->arcs_list[ARC_BUFC_METADATA]) ?
Yes
No
(void) arc_evict(arc_mru, guid, -1, FALSE, ARC_BUFC_METADATA);
spa ?
Yes
No
list_head(&arc_mfu->arcs_list[ARC_BUFC_DATA]) ?
Yes
No
(void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_DATA);
spa ?
Yes
No
list_head(&arc_mfu->arcs_list[ARC_BUFC_METADATA]) ?
Yes
No
(void) arc_evict(arc_mfu, guid, -1, FALSE, ARC_BUFC_METADATA);
spa ?
Yes
No
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);
End

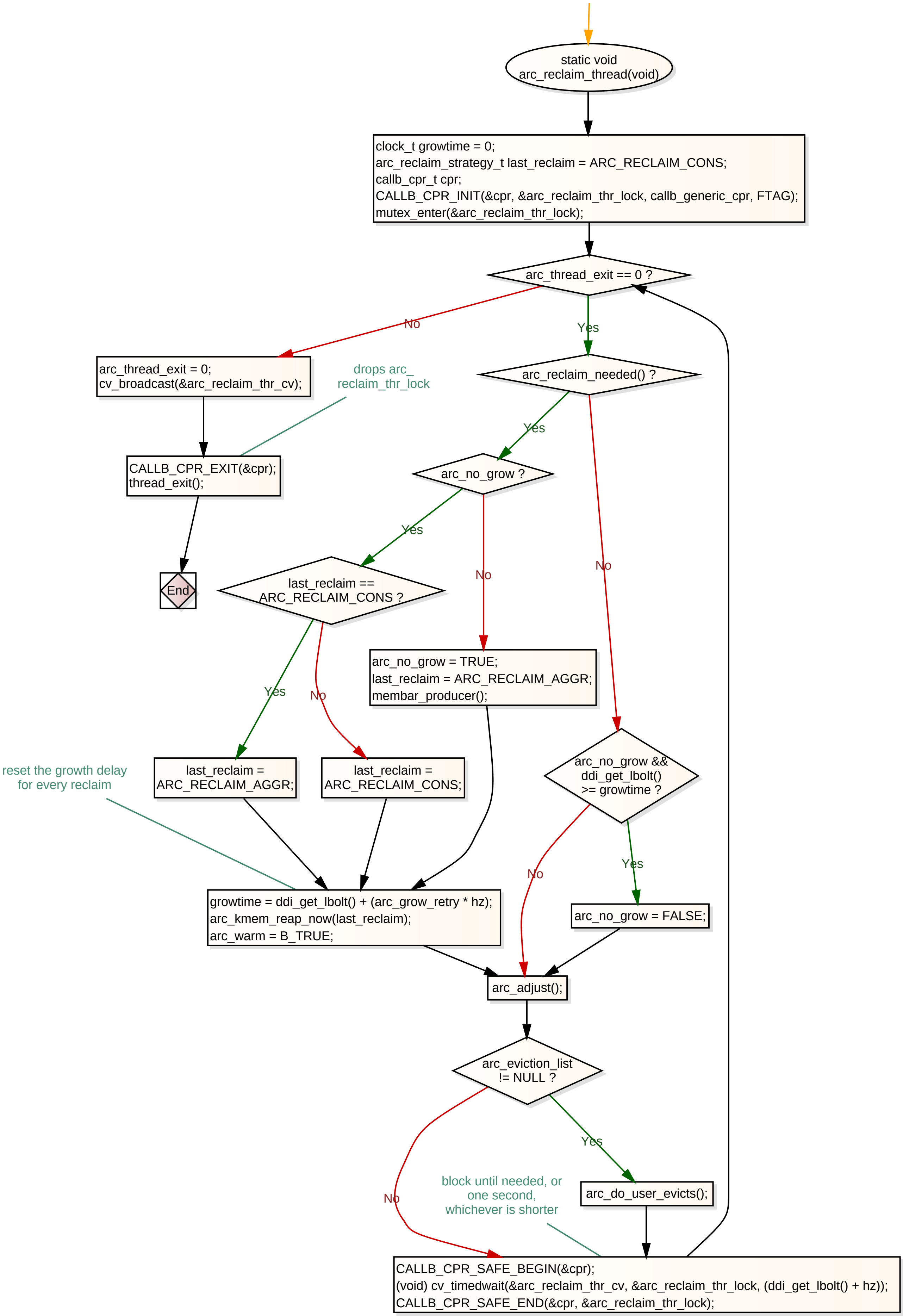


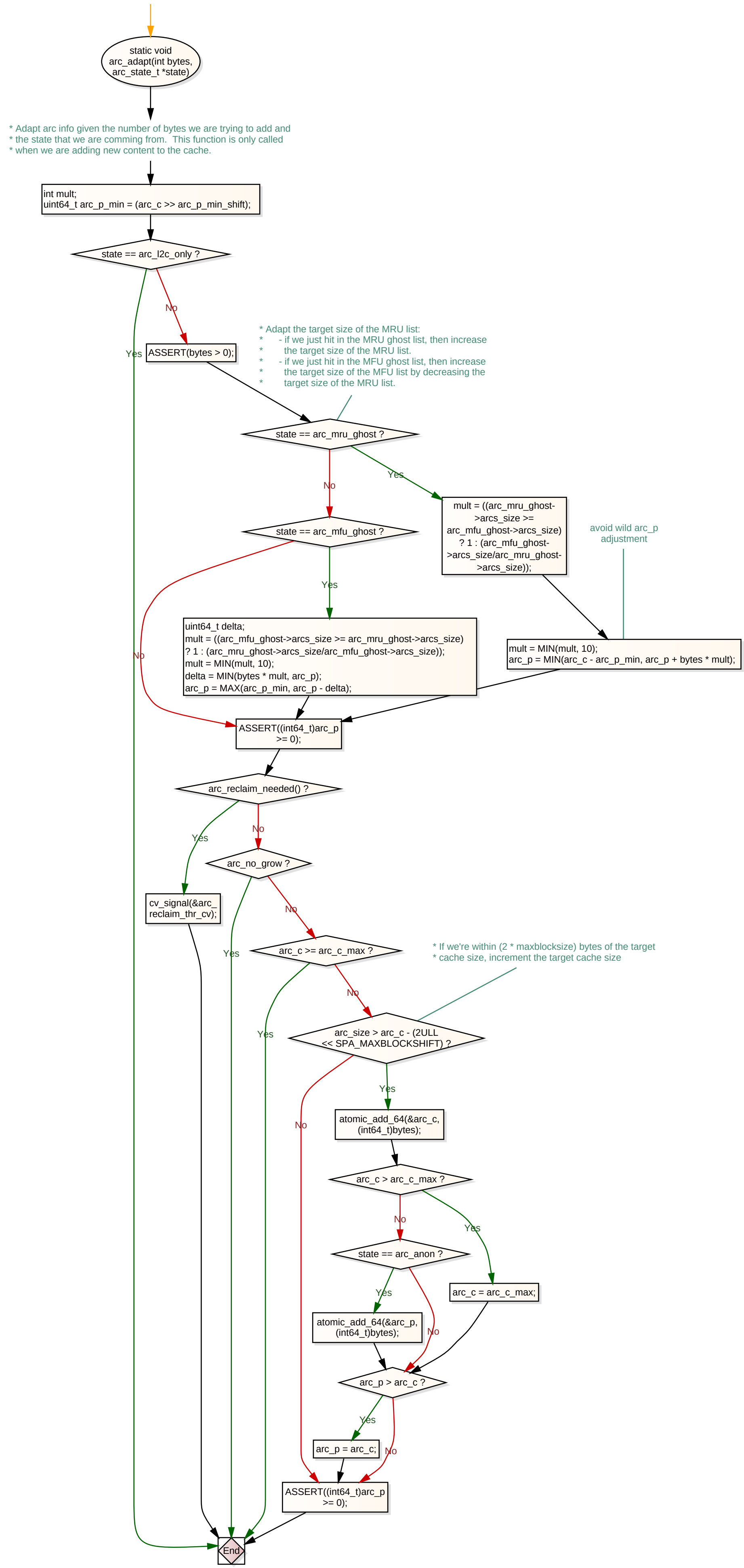
void arc_shrink(void)
arc_c > arc_c_min ?
Yes
No
uint64_t to_free;
#ifdef _KERNEL ?
Yes
No
to_free = MAX(arc_c >> arc_shrink_shift, ptob(needfree));
to_free = arc_c >> arc_shrink_shift;
arc_c > arc_c_min + to_free ?
Yes
No
atomic_add_64(&arc_c, -to_free);
arc_c = arc_c_min;
atomic_add_64(&arc_p, -(arc_p >> arc_shrink_shift));
arc_c > arc_size ?
Yes
No
arc_c = MAX(arc_size, arc_c_min);
arc_p > arc_c ?
Yes
No
arc_p = (arc_c >> 1);
ASSERT(arc_c >= arc_c_min);
ASSERT((int64_t)arc_p >= 0);
arc_size > arc_c ?
Yes
No
arc_adjust();
End


static int
arc_reclaim_needed(void)
* 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.
uint64_t extra;
#ifdef _KERNEL ?
Yes
No
needfree ?
Yes
return (1);
No
* 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.
freemem < lotsfree +
needfree + extra ?
Yes
return (1);
No
* 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.
availrmem <
swapfs_minfree +
swapfs_reserve + extra ?
Yes
return (1);
No
* 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)
defined(__i386) ?
Yes
No
vmem_size(heap_arena,
VMEM_FREE) <
(vmem_size(heap_arena,
VMEM_FREE |
VMEM_ALLOC) >> 2) ?
Yes
return (1);
No
* 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.
zio_arena != NULL &&
vmem_size(zio_arena,
VMEM_FREE) <
(vmem_size(zio_arena,
VMEM_ALLOC) >> 4) ?
Yes
return (1);
No
spa_get_random(100)
== 0 ?
Yes
return (1);
No
return (0);
End

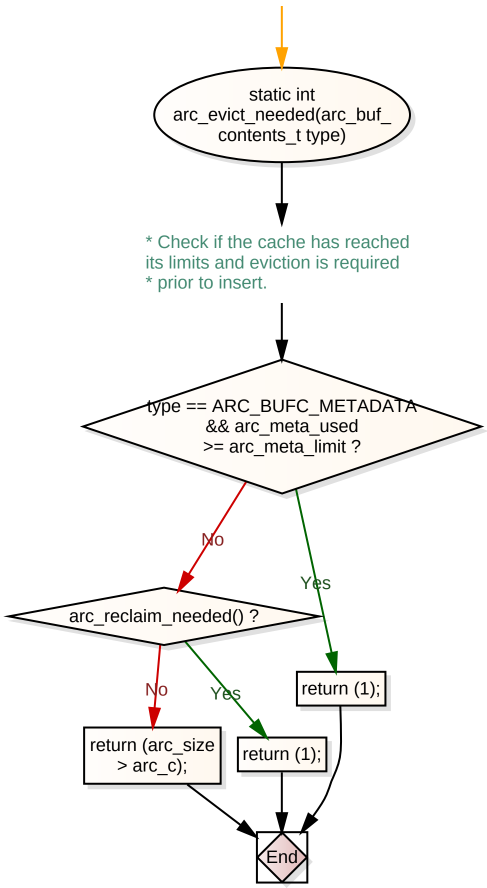


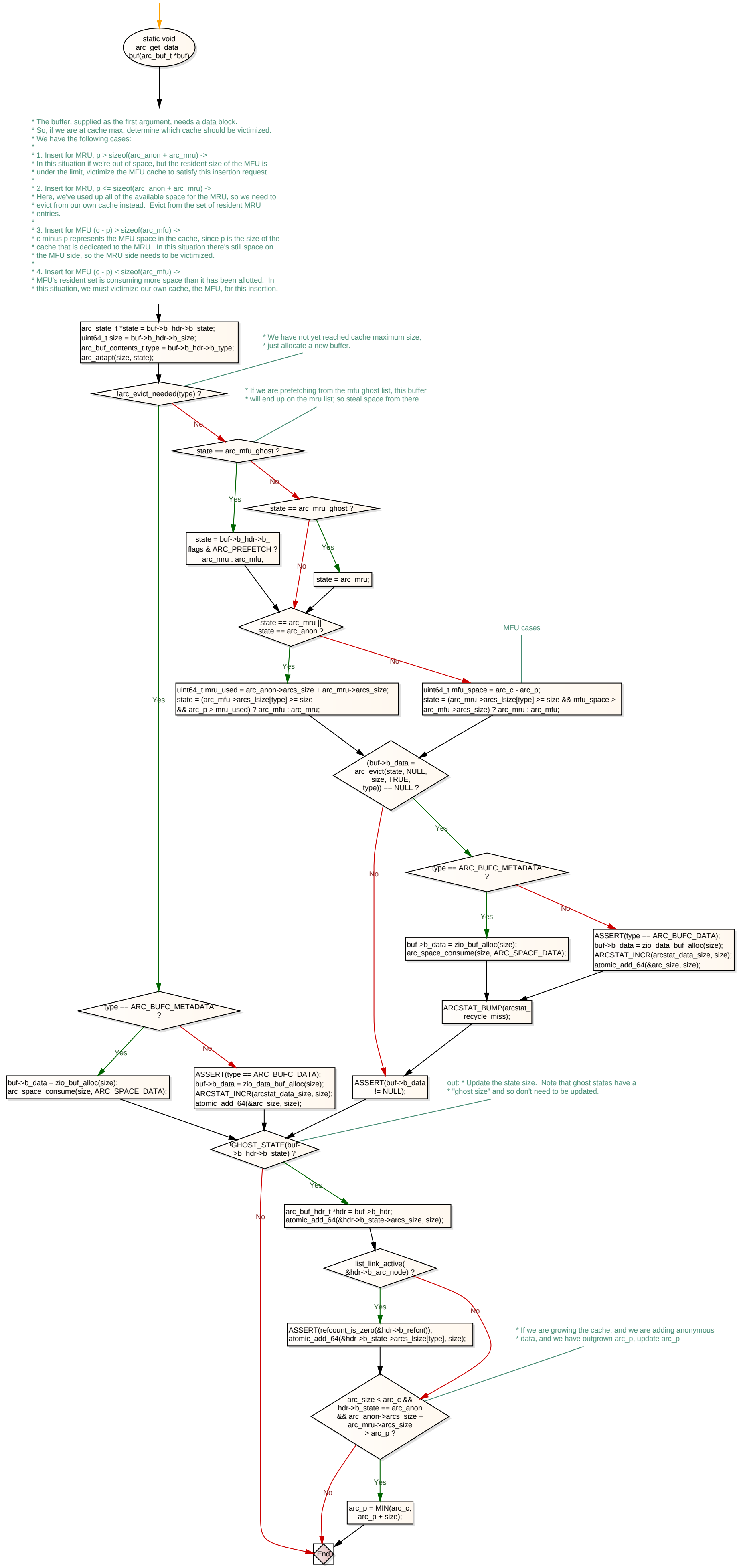
static void arc_kmem_reap_now(arc_reclaim_strategy_t strat)
size_t i;
kmem_cache_t *prev_cache = NULL;
kmem_cache_t *prev_data_cache = NULL;
extern kmem_cache_t *zio_buf_cache[];
extern kmem_cache_t *zio_data_buf_cache[];
#ifdef _KERNEL ?
Yes
No
arc_meta_used >= arc_meta_limit ?
* We are exceeding our meta-data cache limit.
* Purge some DNLC entries to release holds on meta-data.
Yes
No
dnlc_reduce_cache((void *)(uintptr_t)arc_reduce_dnlc_percent);
* Reclaim unused memory from all kmem caches.
defined(__i386) ?
Yes
No
kmem_reap();
* An aggressive reclamation will shrink the cache size as well as
* reap free buffers from the arc kmem caches.
strat == ARC_RECLAIM_AGGR ?
Yes
No
arc_shrink();
i = 0
i < SPA_MAXBLOCKSIZE >> SPA_MINBLOCKSHIFT ?
No
Yes
kmem_cache_reap_now(buf_cache);
kmem_cache_reap_now(hdr_cache);
* Ask the vmem areana to reclaim unused memory from its
* quantum caches.
zio_buf_cache[i] != prev_cache ?
Yes
No
prev_cache = zio_buf_cache[i];
kmem_cache_reap_now(zio_buf_cache[i]);
zio_arena != NULL && strat == ARC_RECLAIM_AGGR ?
Yes
No
vmem_qcache_reap (zio_arena);
zio_data_buf_cache[i] != prev_data_cache ?
Yes
No
prev_data_cache = zio_data_buf_cache[i];
kmem_cache_reap_now(zio_data_buf_cache[i]);
End
i++


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);
arc_thread_exit == 0 ?
No
Yes
arc_thread_exit = 0;
cv_broadcast(&arc_reclaim_thr_cv);
drops arc_
reclaim_thr_lock
arc_reclaim_needed() ?
Yes
CALLB_CPR_EXIT(&cpr);
thread_exit();
arc_no_grow ?
Yes
End
No
No
last_reclaim ==
ARC_RECLAIM_CONS ?
arc_no_grow = TRUE;
last_reclaim = ARC_RECLAIM_AGGR;
membar_producer();
Yes
No
reset the growth delay
for every reclaim
last_reclaim =
ARC_RECLAIM_AGGR;
last_reclaim =
ARC_RECLAIM_CONS;
arc_no_grow &&
ddi_get_lbolt()
>= growtime ?
No
Yes
growtime = ddi_get_lbolt() + (arc_grow_retry * hz);
arc_kmem_reap_now(last_reclaim);
arc_warm = B_TRUE;
arc_no_grow = FALSE;
arc_adjust();
arc_eviction_list
!= NULL ?
Yes
No
block until needed, or
one second,
whichever is shorter
arc_do_user_evicts();
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);

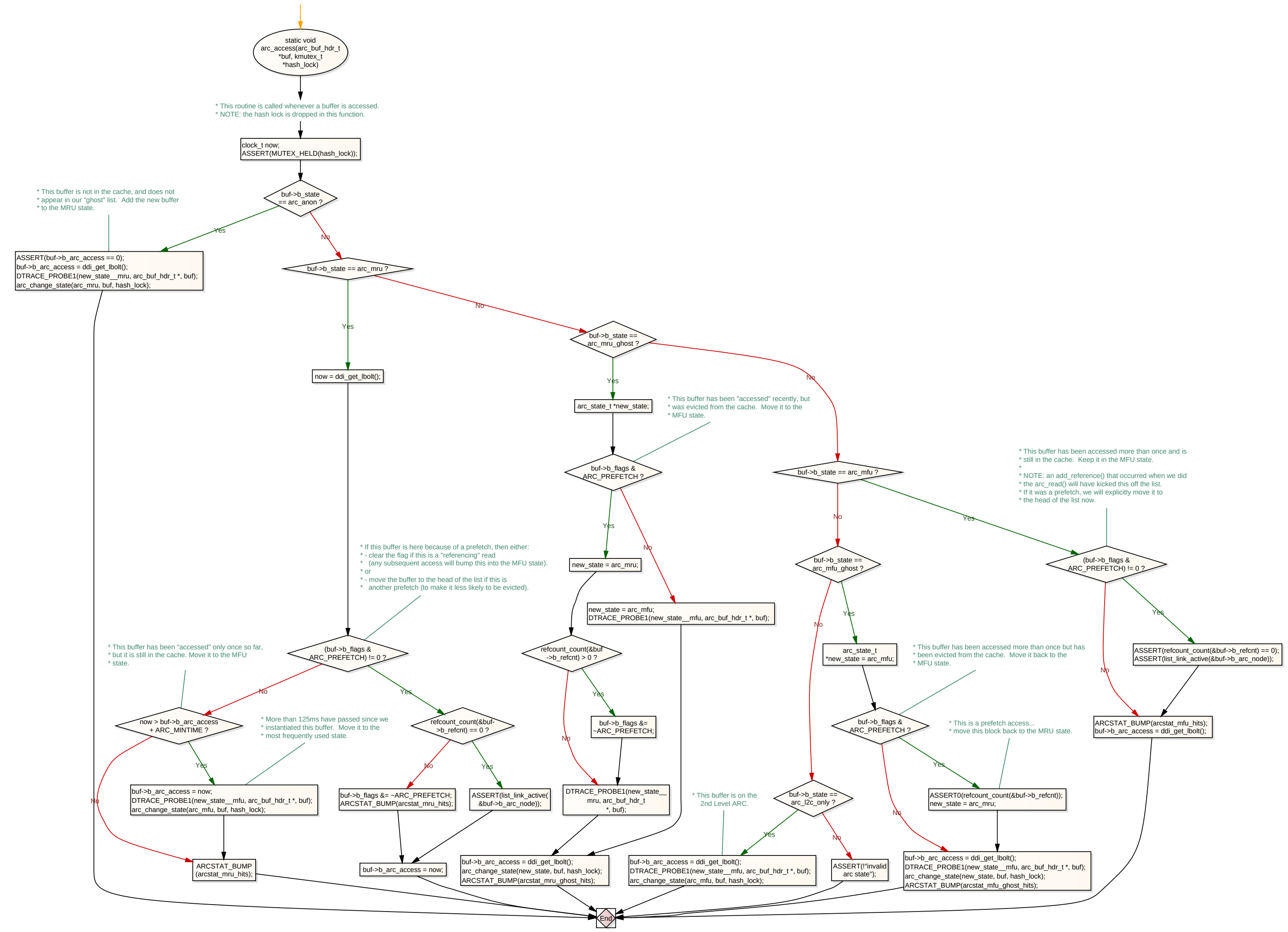


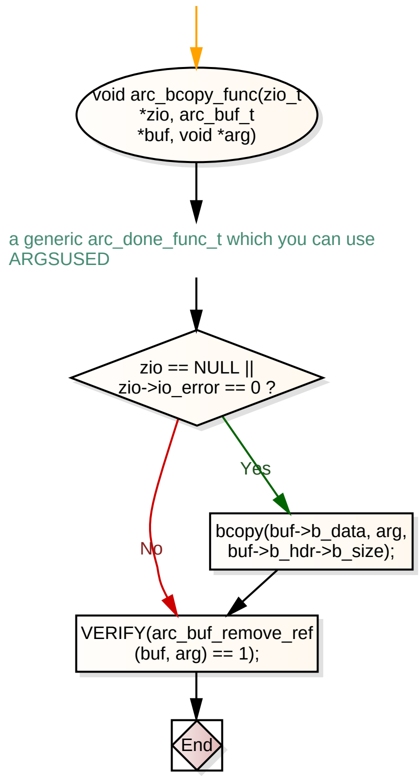
static void
arc_adapt(int bytes,
arc_state_t *state)
* Adapt arc info given the number of bytes we are trying to add and
* the state that we are comming from. This function is only called
* when we are adding new content to the cache.
int mult;
uint64_t arc_p_min = (arc_c >> arc_p_min_shift);
state == arc_l2c_only ?
No
Yes
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.
state == arc_mru_ghost ?
No
Yes
state == arc_mfu_ghost ?
mult = ((arc_mru_ghost->arcs_size >= arc_mfu_ghost->arcs_size) ? 1 : (arc_mfu_ghost->arcs_size/arc_mru_ghost->arcs_size));
avoid wild arc_p adjustment
Yes
No
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);
mult = MIN(mult, 10);
arc_p = MIN(arc_c - arc_p_min, arc_p + bytes * mult);
ASSERT((int64_t)arc_p >= 0);
arc_reclaim_needed() ?
Yes
No
arc_no_grow ?
cv_signal(&arc_reclaim_thr_cv);
No
Yes
arc_c >= arc_c_max ?
* If we're within (2 * maxblocksize) bytes of the target
* cache size, increment the target cache size
No
Yes
arc_size > arc_c - (2ULL << SPA_MAXBLOCKSHIFT) ?
Yes
No
atomic_add_64(&arc_c, (int64_t)bytes);
arc_c > arc_c_max ?
No
Yes
state == arc_anon ?
arc_c = arc_c_max;
Yes
No
atomic_add_64(&arc_p, (int64_t)bytes);
arc_p > arc_c ?
Yes
No
arc_p = arc_c;
ASSERT((int64_t)arc_p >= 0);
End

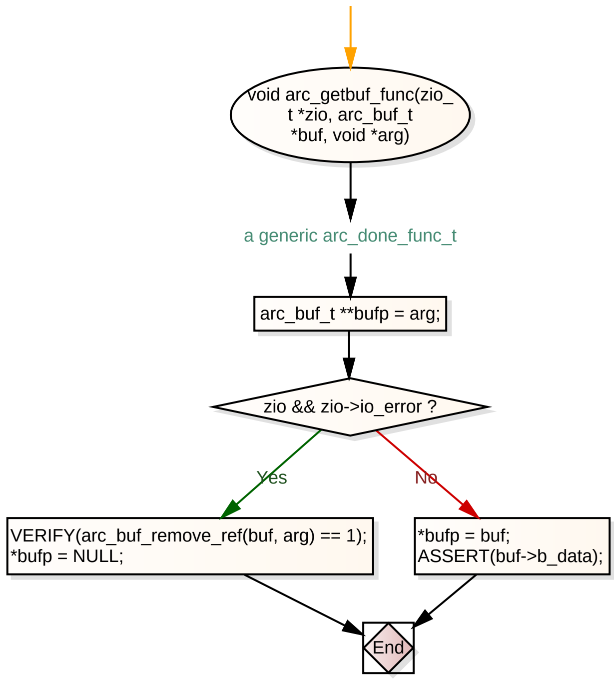

static int
arc_evict_needed(arc_buf_
contents_t type)
* Check if the cache has reached
its limits and eviction is required
* prior to insert.
type == ARC_BUFC_METADATA
&& arc_meta_used
>= arc_meta_limit ?
No
Yes
arc_reclaim_needed() ?
No
Yes
return (1);
return (arc_size
> arc_c);
return (1);
End

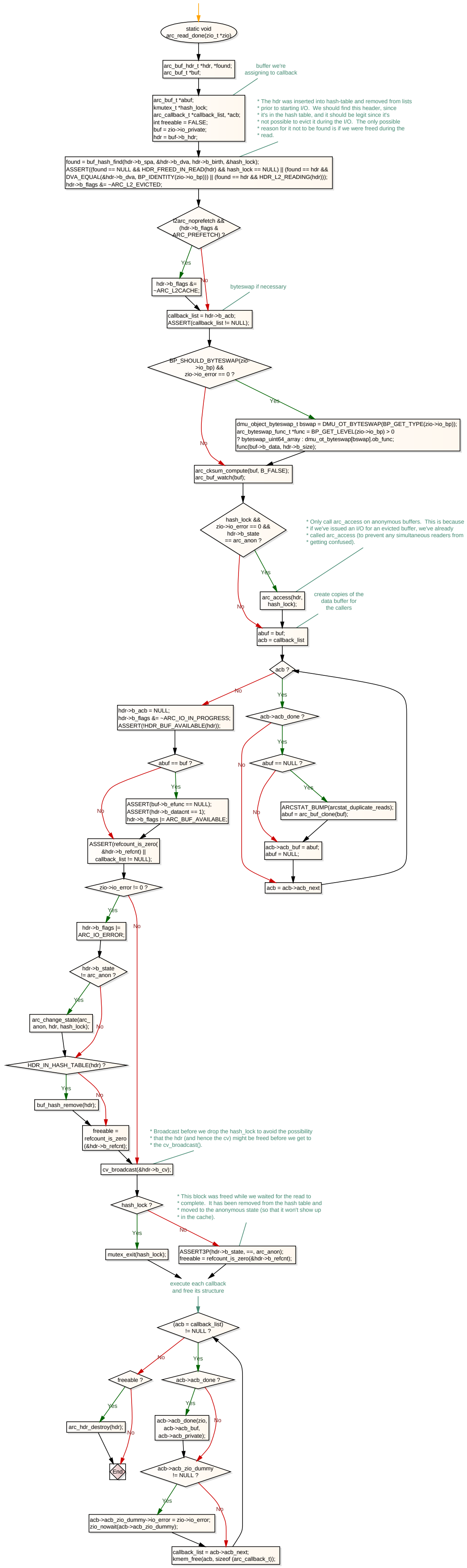


static void arc_get_data_ buf(arc_buf_t *buf)
* The buffer, supplied as the first argument, needs a data block.
* So, if we are at cache max, determine which cache should be victimized.
* We have the following cases:
*
* 1. Insert for MRU, p > sizeof(arc_anon + arc_mru) ->
* In this situation if we're 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.
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.
!arc_evict_needed(type) ?
* If we are prefetching from the mfu ghost list, this buffer
* will end up on the mru list; so steal space from there.
No
state == arc_mfu_ghost ?
No
Yes
state == arc_mru_ghost ?
state = buf->b_hdr->b_ flags & ARC_PREFETCH ? arc_mru : arc_mfu;
Yes
No
state = arc_mru;
state == arc_mru || state == arc_anon ?
MFU cases
No
Yes
Yes
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;
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;
(buf->b_data = arc_evict(state, NULL, size, TRUE, type)) == NULL ?
Yes
No
type == ARC_BUFC_METADATA ?
Yes
No
buf->b_data = zio_buf_alloc(size);
arc_space_consume(size, ARC_SPACE_DATA);
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);
type == ARC_BUFC_METADATA ?
Yes
No
buf->b_data = zio_buf_alloc(size);
arc_space_consume(size, ARC_SPACE_DATA);
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);
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.
!GHOST_STATE(buf->b_hdr->b_state) ?
Yes
No
arc_buf_hdr_t *hdr = buf->b_hdr;
atomic_add_64(&hdr->b_state->arcs_size, size);
list_link_active(&hdr->b_arc_node) ?
Yes
No
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
arc_size < arc_c && hdr->b_state == arc_anon && arc_anon->arcs_size + arc_mru->arcs_size > arc_p ?
Yes
No
arc_p = MIN(arc_c, arc_p + size);
End

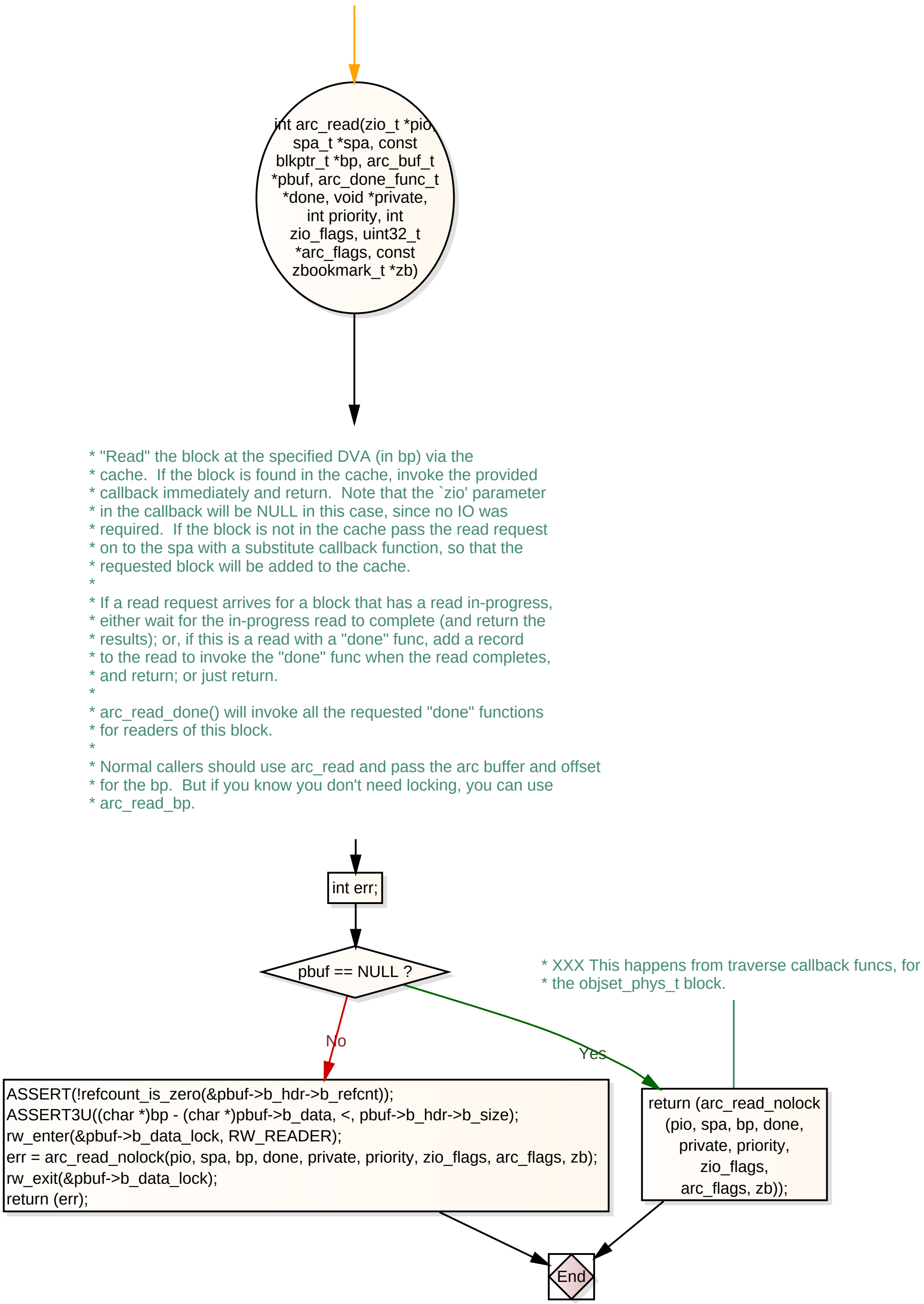

static void arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock)
* This routine is called whenever a buffer is accessed.
* NOTE: the hash lock is dropped in this function.
clock_t now;
ASSERT(MUTEX_HELD(hash_lock));
buf->b_state == arc_anon ?
* This buffer is not in the cache, and does not
* appear in our "ghost" list. Add the new buffer
* to the MRU state.
Yes
No
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);
buf->b_state == arc_mru ?
Yes
No
now = ddi_get_lbolt();
buf->b_state == arc_mru_ghost ?
Yes
No
arc_state_t *new_state;
* This buffer has been "accessed" recently, but
* was evicted from the cache. Move it to the
* MFU state.
buf->b_flags & ARC_PREFETCH ?
Yes
No
new_state = arc_mru;
new_state = arc_mfu;
DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
refcount_count(&buf->b_refcnt) > 0 ?
Yes
No
buf->b_flags &= ~ARC_PREFETCH;
DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
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.
Yes
No
(buf->b_flags & ARC_PREFETCH) != 0 ?
Yes
No
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();
buf->b_state == arc_mfu_ghost ?
Yes
No
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.
buf->b_flags & ARC_PREFETCH ?
* This is a prefetch access...
* move this block back to the MRU state.
Yes
No
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);
buf->b_state == arc_l2c_only ?
* This buffer is on the 2nd Level ARC.
Yes
No
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);
ASSERT(!"invalid arc state");
* 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).
(buf->b_flags & ARC_PREFETCH) != 0 ?
Yes
No
* This buffer has been "accessed" only once so far,
* but it is still in the cache. Move it to the MFU
* state.
now > buf->b_arc_access + ARC_MINTIME ?
Yes
No
* 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);
refcount_count(&buf->b_refcnt) == 0 ?
Yes
No
buf->b_flags &= ~ARC_PREFETCH;
ARCSTAT_BUMP(arcstat_mru_hits);
ASSERT(list_link_active(&buf->b_arc_node));
buf->b_arc_access = now;
buf->b_arc_access = ddi_get_lbolt();
arc_change_state(new_state, buf, hash_lock);
ARCSTAT_BUMP(arcstat_mru_ghost_hits);
End

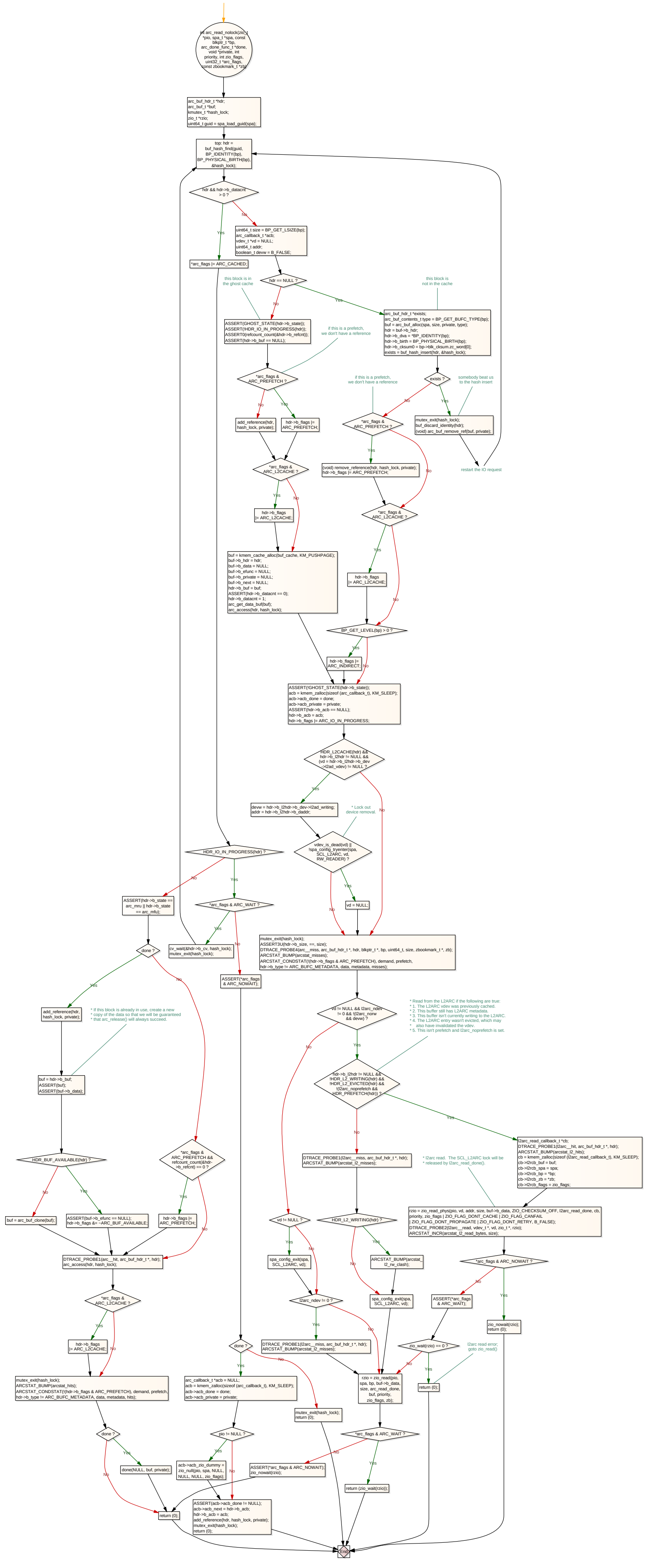

void arc_bcopy_func(zio_t *zio, arc_buf_t *buf, void *arg)
a generic arc_done_func_t which you can use
ARGSUSED
zio == NULL || zio->io_error == 0 ?
Yes
No
bcopy(buf->b_data, arg, buf->b_hdr->b_size);
VERIFY(arc_buf_remove_ref(buf, arg) == 1);
End


void arc_getbuf_func(zio_t *zio, arc_buf_t *buf, void *arg)
a generic arc_done_func_t
arc_buf_t **bufp = arg;
zio && zio->io_error ?
Yes
No
VERIFY(arc_buf_remove_ref(buf, arg) == 1);
*bufp = NULL;
*bufp = buf;
ASSERT(buf->b_data);
End

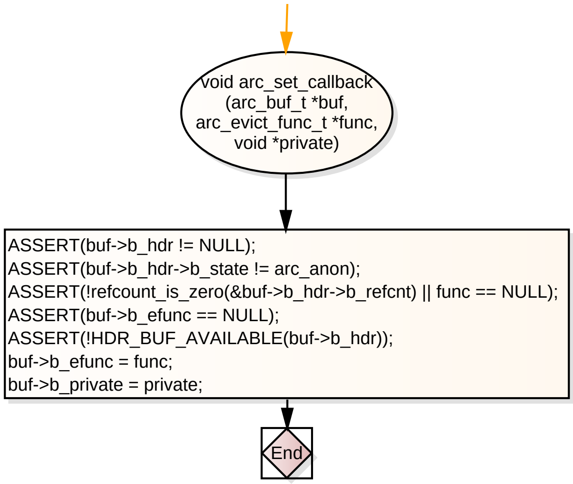


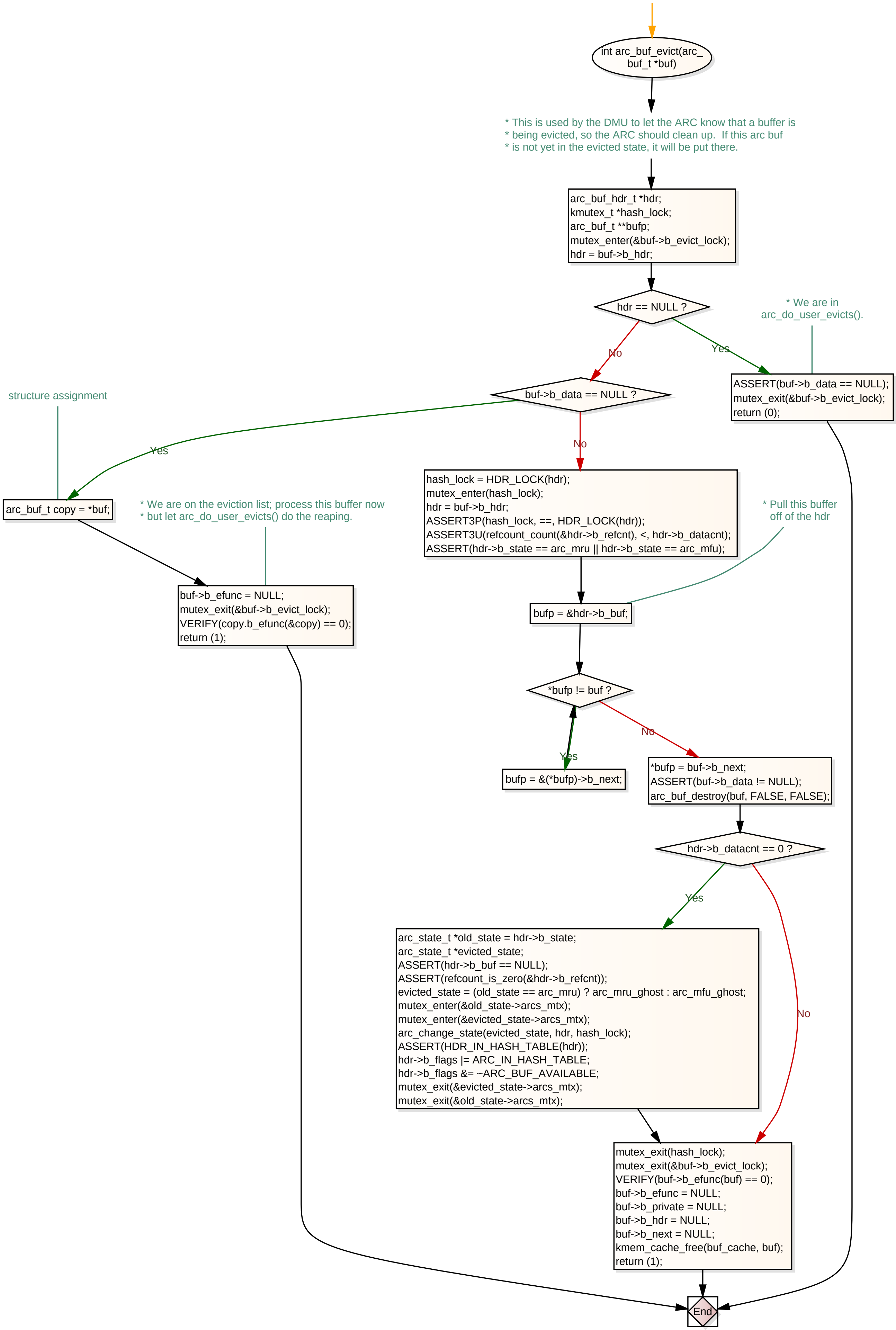
static void arc_read_done(zio_t *zio)
arc_buf_hdr_t *hdr, *found; arc_buf_t *buf;
buffer we're assigning to callback
arc_buf_t *abuf; kmutex_t *hash_lock; arc_callback_t *callback_list, *acb; int freeable = FALSE; buf = zio->io_private; hdr = buf->b_hdr;
* The hdr was inserted into hash-table and removed 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.
found = buf_hash_find(hdr->b_spa, &hdr->b_dva, hdr->b_birth, &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;
l2arc_noprefetch && (hdr->b_flags & ARC_PREFETCH) ?
Yes
No
hdr->b_flags &= ~ARC_L2CACHE;
byteswap if necessary
callback_list = hdr->b_acb; ASSERT(callback_list != NULL);
BP_SHOULD_BYTESWAP(zio->io_bp) && zio->io_error == 0 ?
Yes
No
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);
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).
Yes
No
arc_access(hdr, hash_lock);
create copies of the data buffer for the callers
abuf = buf; acb = callback_list
acb ?
No
Yes
hdr->b_acb = NULL; hdr->b_flags &= ~ARC_IO_IN_PROGRESS; ASSERT(!HDR_BUF_AVAILABLE(hdr));
acb->acb_done ?
Yes
No
abuf == NULL ?
Yes
No
ARCSTAT_BUMP(arcstat_duplicate_reads); abuf = arc_buf_clone(buf);
acb->acb_buf = abuf; abuf = NULL;
acb = acb->acb_next
abuf == buf ?
Yes
No
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);
zio->io_error != 0 ?
Yes
No
hdr->b_flags |= ARC_IO_ERROR;
hdr->b_state != arc_anon ?
Yes
No
arc_change_state(arc_anon, hdr, hash_lock);
HDR_IN_HASH_TABLE(hdr) ?
Yes
No
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);
hash_lock ?
* 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).
Yes
No
mutex_exit(hash_lock);
ASSERT3P(hdr->b_state, ==, arc_anon); freeable = refcount_is_zero(&hdr->b_refcnt);
execute each callback and free its structure
(acb = callback_list) != NULL ?
No
Yes
freeable ?
acb->acb_done ?
Yes
No
arc_hdr_destroy(hdr);
acb->acb_done(zio, acb->acb_buf, acb->acb_private);
End
acb->acb_zio_dummy != NULL ?
Yes
No
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));

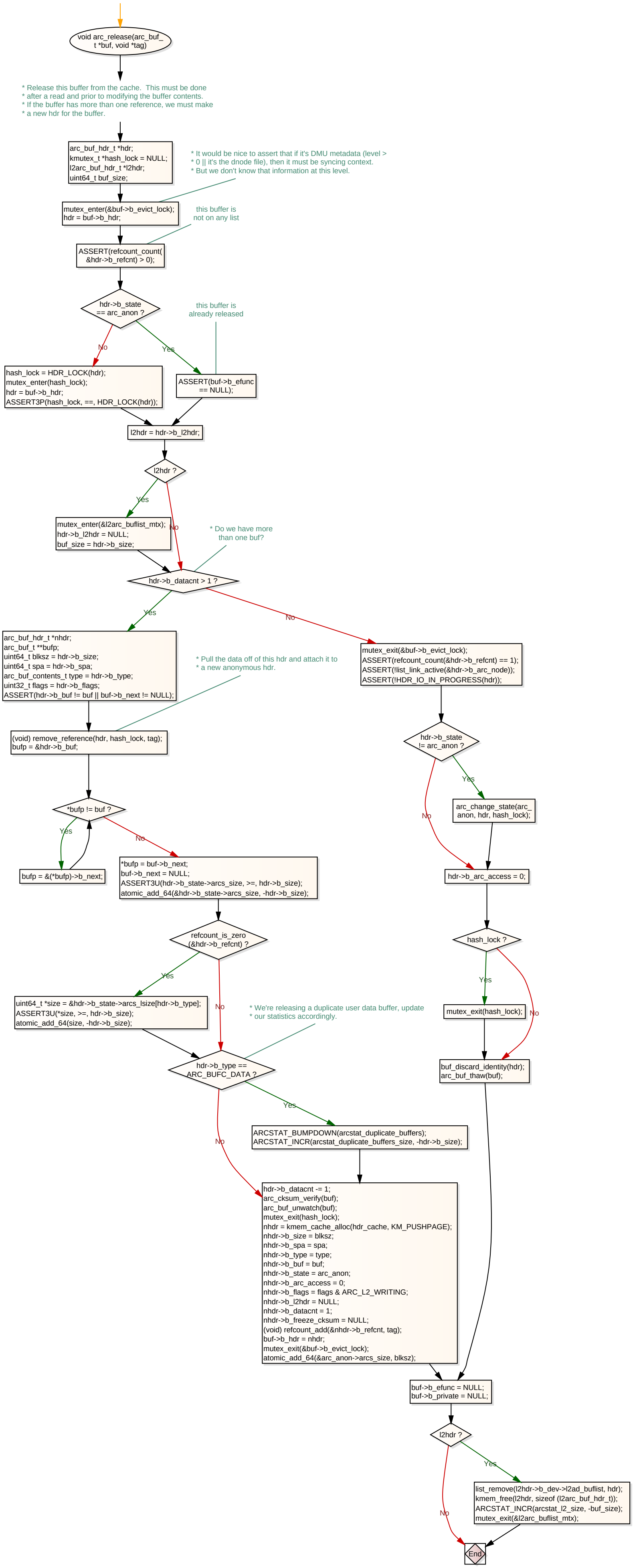

int arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_buf_t *pbuf, arc_done_func_t *done, void *private, int priority, int zio_flags, uint32_t *arc_flags, const zbookmark_t *zb)
* "Read" the block at the specified DVA (in bp) via the
* cache. If the block is found in the cache, invoke the provided
* 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.
*
* 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 err;
pbuf == NULL ?
* XXX This happens from traverse callback funcs, for
* the objset_phys_t block.
No
Yes
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);
return (arc_read_nolock (pio, spa, bp, done, private, priority, zio_flags, arc_flags, zb));
End

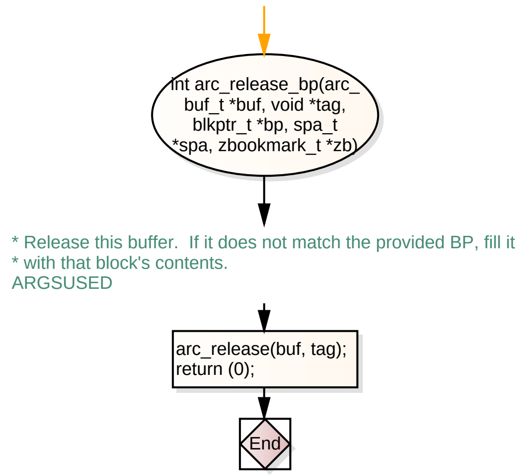


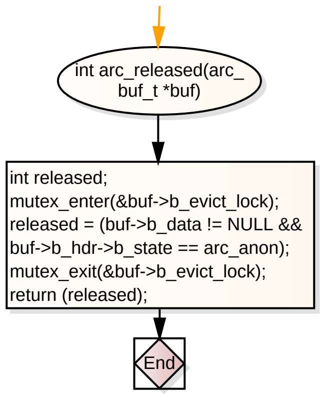
int arc_read_nolock(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done, void *private, int priority, int zio_flags, uint32_t *arc_flags, const zbookmark_t *zb)
arc_buf_hdr_t *hdr; arc_buf_t *buf; kmutex_t *hash_lock; zio_t *rzio; uint64_t guid = spa_load_guid(spa);
top: hdr = buf_hash_find(guid, BP_IDENTITY(bp), BP_PHYSICAL_BIRTH(bp), &hash_lock);
hdr && hdr->b_datacnt > 0 ?
*arc_flags |= ARC_CACHED;
uint64_t size = BP_GET_LSIZE(bp); arc_callback_t *acb; vdev_t *vd = NULL; uint64_t addr; boolean_t devw = B_FALSE;
hdr == NULL ?
this block is in the ghost cache
this block is not in the cache
ASSERT(GHOST_STATE(hdr->b_state)); ASSERT(!HDR_IO_IN_PROGRESS(hdr)); ASSERT0(refcount_count(&hdr->b_refcnt)); ASSERT(hdr->b_buf == NULL);
arc_buf_hdr_t *exists; arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp); buf = arc_buf_alloc(spa, size, private, type); hdr = buf->b_hdr; 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 this is a prefetch, we don't have a reference
*arc_flags & ARC_PREFETCH ?
exists ?
somebody beat us to the hash insert
add_reference(hdr, hash_lock, private);
hdr->b_flags |= ARC_PREFETCH;
mutex_exit(hash_lock); buf_discard_identity(hdr); (void) arc_buf_remove_ref(buf, private);
restart the IO request
(void) remove_reference(hdr, hash_lock, private); hdr->b_flags |= ARC_PREFETCH;
*arc_flags & ARC_L2CACHE ?
hdr->b_flags |= ARC_L2CACHE;
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);
BP_GET_LEVEL(bp) > 0 ?
hdr->b_flags |= ARC_INDIRECT;
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;
HDR_L2CACHE(hdr) && 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;
* Lock out device removal.
vdev_is_dead(vd) || !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER) ?
vd = NULL;
HDR_IO_IN_PROGRESS(hdr) ?
ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);
*arc_flags & ARC_WAIT ?
cv_wait(&hdr->b_cv, hash_lock); mutex_exit(hash_lock);
ASSERT(*arc_flags & ARC_NOWAIT);
mutex_exit(hash_lock); ASSERT3U(hdr->b_size, ==, size); DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp, uint64_t, size, zbookmark_t *, zb); ARCSTAT_BUMP(arcstat_misses); ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH), demand, prefetch, hdr->b_type != ARC_BUFC_METADATA, data, metadata, misses);
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.
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);
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;
HDR_BUF_AVAILABLE(hdr) ?
*arc_flags & ARC_PREFETCH && refcount_count(&hdr->b_refcnt) == 0 ?
DTRACE_PROBE1(l2arc__miss, arc_buf_hdr_t *, hdr); ARCSTAT_BUMP(arcstat_l2_misses);
* l2arc read. The SCL_L2ARC lock will be
* released by l2arc_read_done().
rzio = zio_read_phys(pio, vd, addr, size, 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, size);
buf = arc_buf_clone(buf);
ASSERT(buf->b_efunc == NULL); hdr->b_flags &= ~ARC_BUF_AVAILABLE;
hdr->b_flags |= ARC_PREFETCH;
vd != NULL ?
HDR_L2_WRITING(hdr) ?
*arc_flags & ARC_NOWAIT ?
DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr); arc_access(hdr, hash_lock);
spa_config_exit(spa, SCL_L2ARC, vd);
ARCSTAT_BUMP(arcstat_l2_rw_clash);
*arc_flags & ARC_L2CACHE ?
l2arc_ndev != 0 ?
spa_config_exit(spa, SCL_L2ARC, vd);
ASSERT(*arc_flags & ARC_WAIT);
zio_nowait(rzio); return (0);
hdr->b_flags |= ARC_L2CACHE;
done ?
DTRACE_PROBE1(l2arc__miss, arc_buf_hdr_t *, hdr); ARCSTAT_BUMP(arcstat_l2_misses);
zio_wait(rzio) == 0 ?
l2arc read error; goto zio_read()
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);
arc_callback_t *acb = NULL; acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP); acb->acb_done = done; acb->acb_private = private;
rzio = zio_read(pio, spa, bp, buf->b_data, size, arc_read_done, buf, priority, zio_flags, zb);
return (0);
mutex_exit(hash_lock); return (0);
done ?
pio != NULL ?
*arc_flags & ARC_WAIT ?
done(NULL, buf, private);
acb->acb_zio_dummy = zio_null(pio, spa, NULL, NULL, NULL, zio_flags);
ASSERT(*arc_flags & ARC_NOWAIT); zio_nowait(rzio);
return (zio_wait(rzio));
return (0);
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);
End

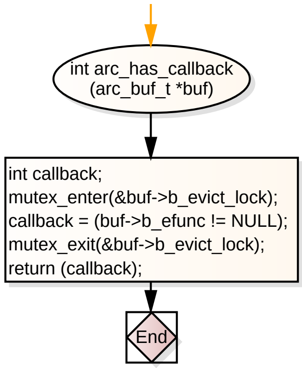

void arc_set_callback
(arc_buf_t *buf,
arc_evict_func_t *func,
void *private)
ASSERT(buf->b_hdr != NULL);
ASSERT(buf->b_hdr->b_state != arc_anon);
ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt) || func == NULL);
ASSERT(buf->b_efunc == NULL);
ASSERT(!HDR_BUF_AVAILABLE(buf->b_hdr));
buf->b_efunc = func;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

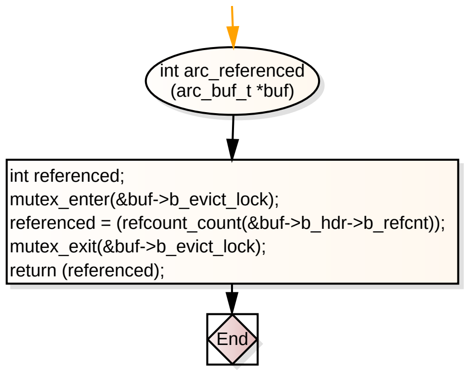

int arc_buf_evict(arc_buf_t *buf)
* This is used by the DMU to let the ARC know that a buffer is
* being evicted, so the ARC should clean up. If this arc buf
* is not yet in the evicted state, it will be put there.
arc_buf_hdr_t *hdr;
kmutex_t *hash_lock;
arc_buf_t **bufp;
mutex_enter(&buf->b_evict_lock);
hdr = buf->b_hdr;
hdr == NULL ?
* We are in
arc_do_user_evicts().
Yes
No
ASSERT(buf->b_data == NULL);
mutex_exit(&buf->b_evict_lock);
return (0);
buf->b_data == NULL ?
structure assignment
Yes
No
arc_buf_t copy = *buf;
* We are on the eviction list; process this buffer now
* but let arc_do_user_evicts() do the reaping.
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);
* Pull this buffer
off of the hdr
buf->b_efunc = NULL;
mutex_exit(&buf->b_evict_lock);
VERIFY(copy.b_efunc(&copy) == 0);
return (1);
bufp = &hdr->b_buf;
*bufp != buf ?
Yes
No
bufp = &(*bufp)->b_next;
*bufp = buf->b_next;
ASSERT(buf->b_data != NULL);
arc_buf_destroy(buf, FALSE, FALSE);
hdr->b_datacnt == 0 ?
Yes
No
arc_state_t *old_state = hdr->b_state;
arc_state_t *evicted_state;
ASSERT(hdr->b_buf == NULL);
ASSERT(refcount_is_zero(&hdr->b_refcnt));
evicted_state = (old_state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;
mutex_enter(&old_state->arcs_mtx);
mutex_enter(&evicted_state->arcs_mtx);
arc_change_state(evicted_state, hdr, hash_lock);
ASSERT(HDR_IN_HASH_TABLE(hdr));
hdr->b_flags |= ARC_IN_HASH_TABLE;
hdr->b_flags &= ~ARC_BUF_AVAILABLE;
mutex_exit(&evicted_state->arcs_mtx);
mutex_exit(&old_state->arcs_mtx);
mutex_exit(hash_lock);
mutex_exit(&buf->b_evict_lock);
VERIFY(buf->b_efunc(buf) == 0);
buf->b_efunc = NULL;
buf->b_private = NULL;
buf->b_hdr = NULL;
buf->b_next = NULL;
kmem_cache_free(buf_cache, buf);
return (1);
End

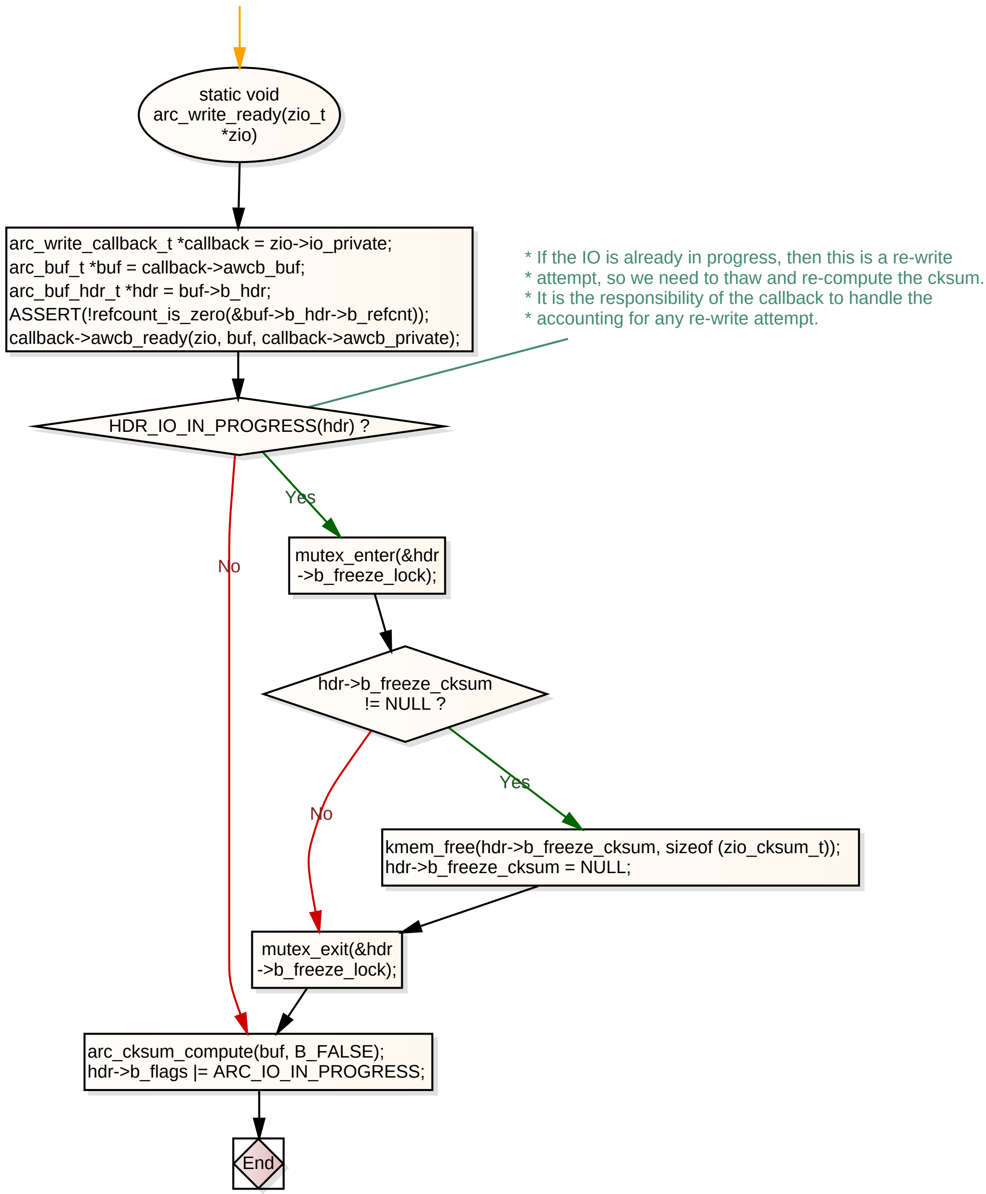

void arc_release(arc_buf_t *buf, void *tag)
* Release this buffer from the cache. This must be done
* after a read and prior to modifying the buffer contents.
* If the buffer has more than one reference, we must make
* a new hdr for the buffer.
arc_buf_hdr_t *hdr;
kmutex_t *hash_lock = NULL;
l2arc_buf_hdr_t *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);
hdr->b_state
== arc_anon ?
this buffer is
already released
No
Yes
hash_lock = HDR_LOCK(hdr);
mutex_enter(hash_lock);
hdr = buf->b_hdr;
ASSERT3P(hash_lock, ==, HDR_LOCK(hdr));
ASSERT(buf->b_efunc
== NULL);
l2hdr = hdr->b_l2hdr;
l2hdr ?
Yes
No
mutex_enter(&l2arc_buflist_mtx);
hdr->b_l2hdr = NULL;
buf_size = hdr->b_size;
* Do we have more
than one buf?
hdr->b_datacnt > 1 ?
Yes
No
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.
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));
(void) remove_reference(hdr, hash_lock, tag);
bufp = &hdr->b_buf;
hdr->b_state
!= arc_anon ?
Yes
No
arc_change_state(arc_anon, hdr, hash_lock);
*bufp != buf ?
Yes
No
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);
hdr->b_arc_access = 0;
refcount_is_zero
(&hdr->b_refcnt) ?
hash_lock ?
Yes
No
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.
mutex_exit(hash_lock);
hdr->b_type ==
ARC_BUFC_DATA ?
buf_discard_identity(hdr);
arc_buf_thaw(buf);
Yes
No
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);
buf->b_efunc = NULL;
buf->b_private = NULL;
l2hdr ?
Yes
No
list_remove(l2hdr->b_dev->l2ad_buflist, hdr);
kmem_free(l2hdr, sizeof (l2arc_buf_hdr_t));
ARCSTAT_INCR(arcstat_l2_size, -buf_size);
mutex_exit(&l2arc_buflist_mtx);
End

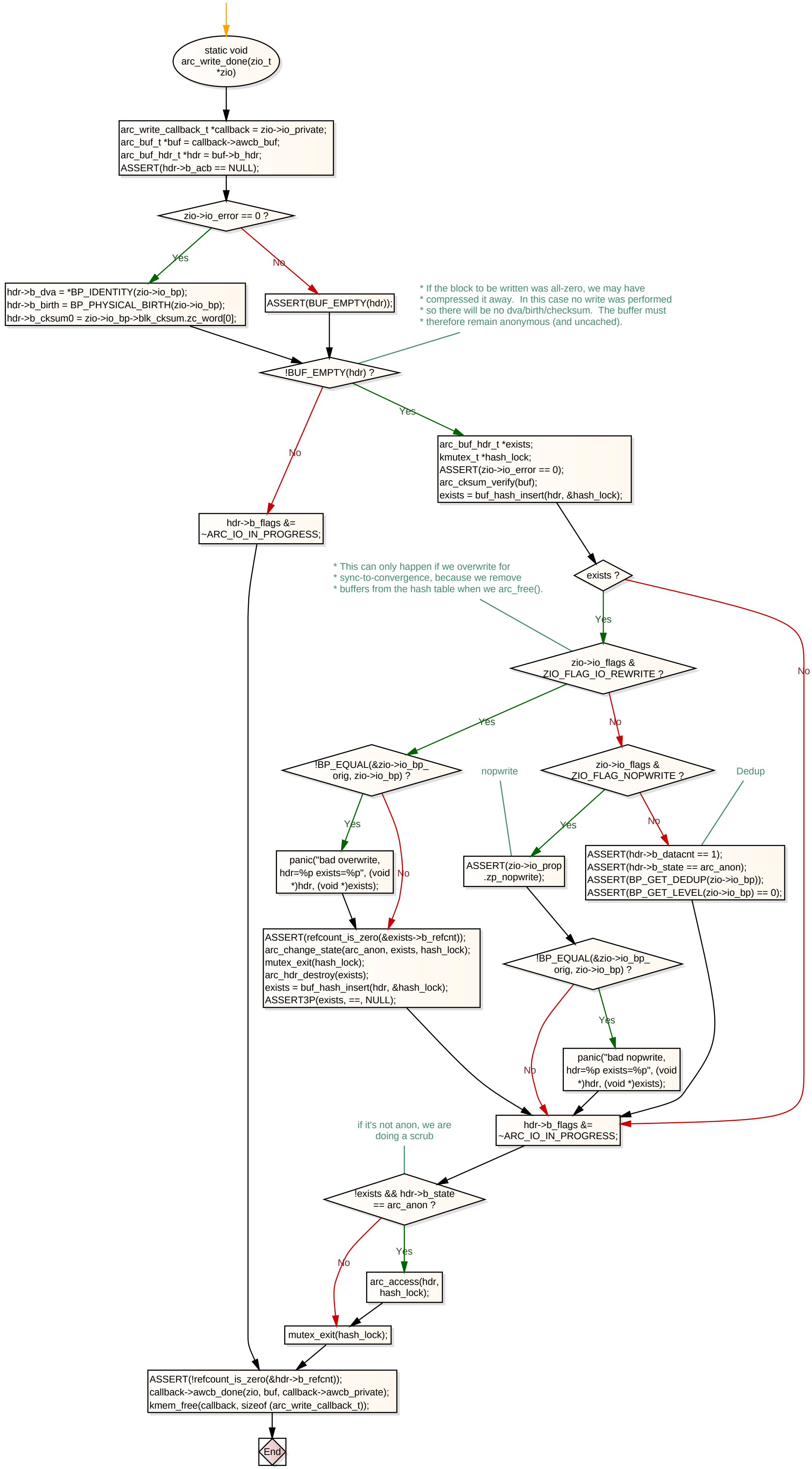

int arc_release_bp(arc_buf_t *buf, void *tag, blkptr_t *bp, spa_t *spa, zbookmark_t *zb)
* Release this buffer. If it does not match the provided BP, fill it
* with that block's contents.
ARGSUSED
arc_release(buf, tag);
return (0);
End

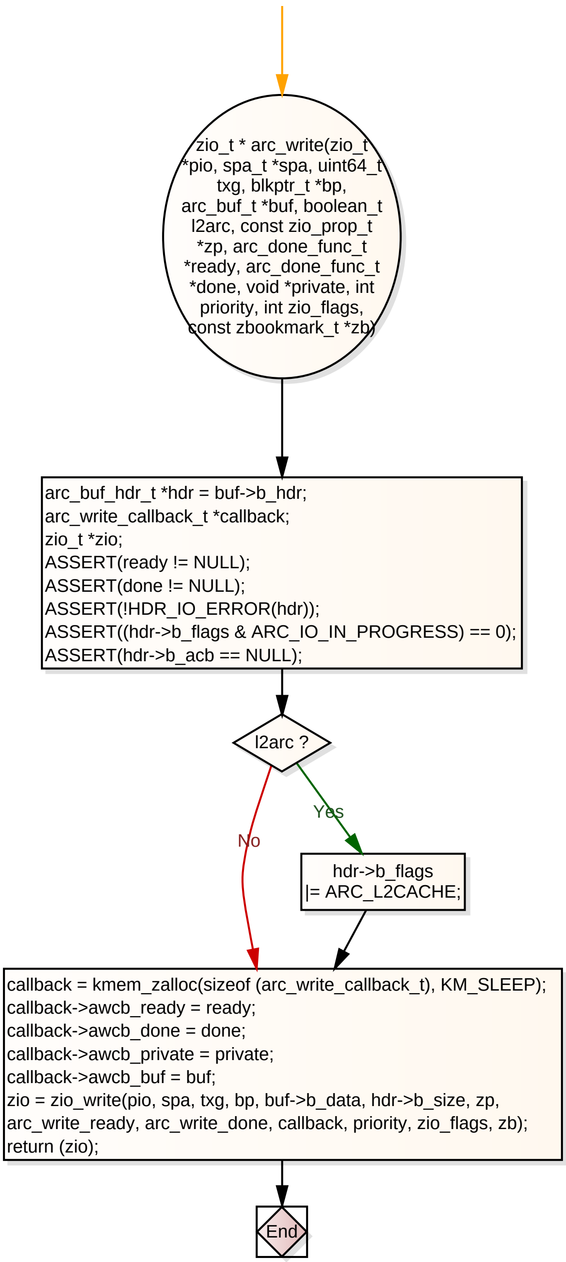

int arc_released(arc_
buf_t *buf)
int released;
mutex_enter(&buf->b_evict_lock);
released = (buf->b_data != NULL &&
buf->b_hdr->b_state == arc_anon);
mutex_exit(&buf->b_evict_lock);
return (released);
End

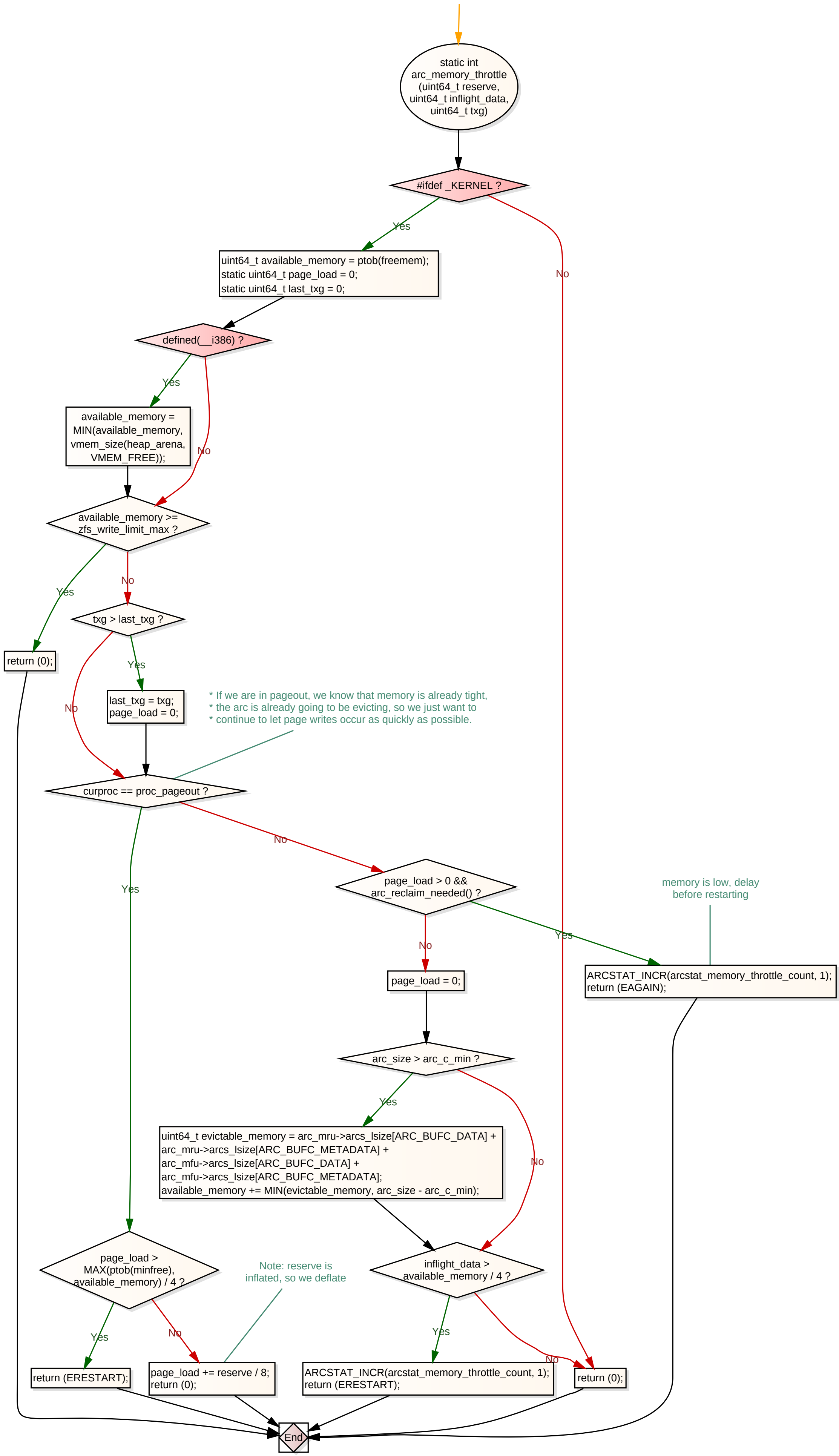

int arc_has_callback
(arc_buf_t *buf)
int callback;
mutex_enter(&buf->b_evict_lock);
callback = (buf->b_efunc != NULL);
mutex_exit(&buf->b_evict_lock);
return (callback);
End

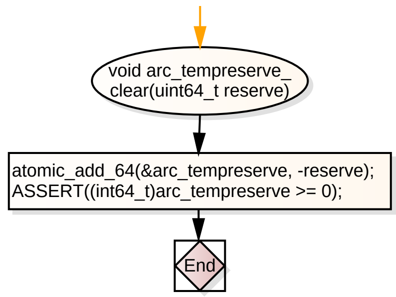

int arc_referenced
(arc_buf_t *buf)
int referenced;
mutex_enter(&buf->b_evict_lock);
referenced = (refcount_count(&buf->b_hdr->b_refcnt));
mutex_exit(&buf->b_evict_lock);
return (referenced);
End

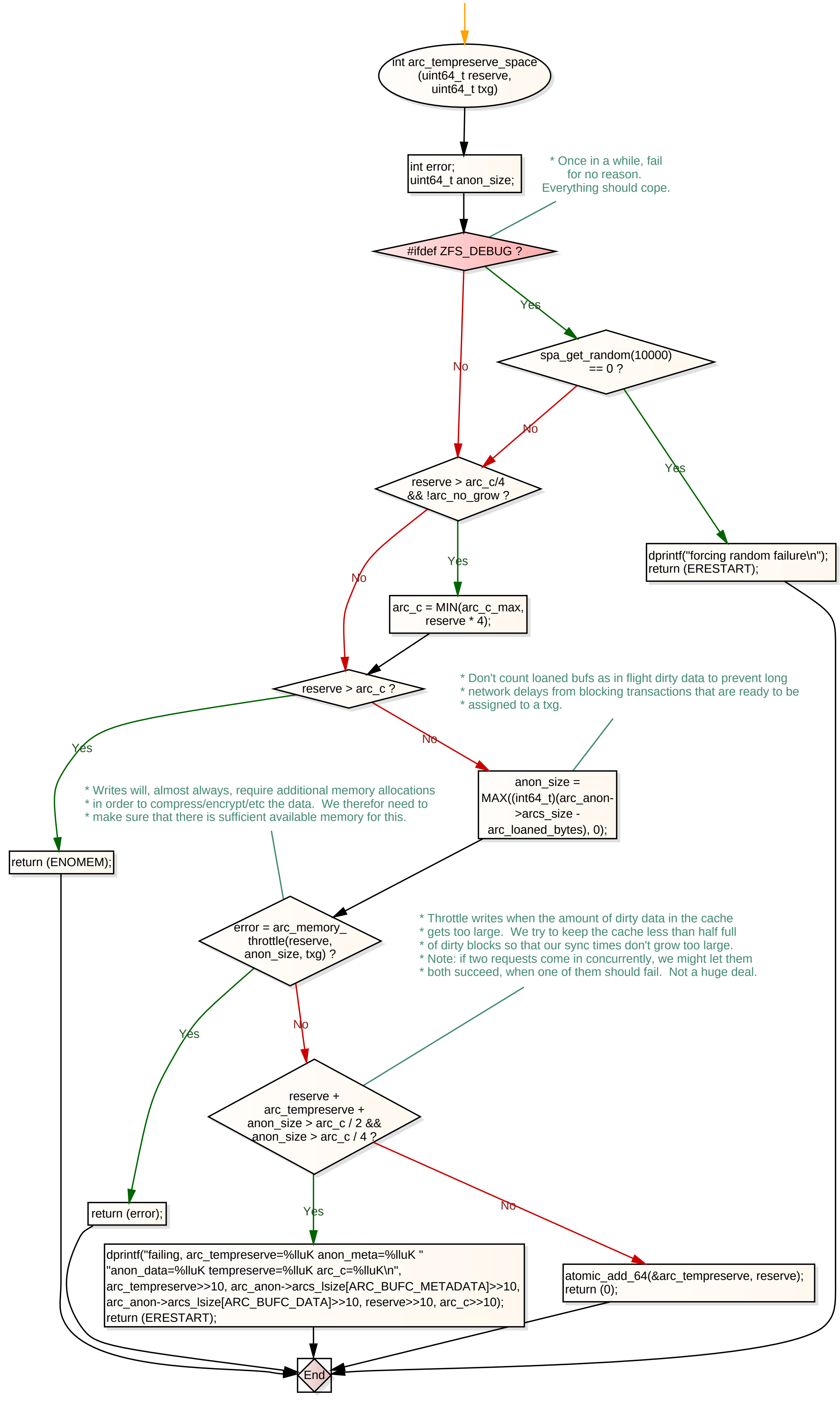

static void
arc_write_ready(zio_t
*zio)
arc_write_callback_t *callback = zio->io_private;
arc_buf_t *buf = callback->awcb_buf;
arc_buf_hdr_t *hdr = buf->b_hdr;
ASSERT(!refcount_is_zero(&buf->b_hdr->b_refcnt));
callback->awcb_ready(zio, buf, callback->awcb_private);
* If the IO is already in progress, 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.
HDR_IO_IN_PROGRESS(hdr) ?
Yes
No
mutex_enter(&hdr
->b_freeze_lock);
hdr->b_freeze_cksum
!= NULL ?
Yes
No
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;
End


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);
zio->io_error == 0 ?
Yes
No
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];
ASSERT(BUF_EMPTY(hdr));
* If the block to be written was all-zero, we may have
* compressed it away. In this case no write was performed
* so there will be no dva/birth/checksum. The buffer must
* therefore remain anonymous (and uncached).
!BUF_EMPTY(hdr) ?
Yes
No
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);
hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
* This can only happen if we overwrite for
* sync-to-convergence, because we remove
* buffers from the hash table when we arc_free().
exists ?
Yes
No
zio->io_flags & ZIO_FLAG_IO_REWRITE ?
Yes
No
!BP_EQUAL(&zio->io_bp_orig, zio->io_bp) ?
nopwrite
zio->io_flags & ZIO_FLAG_NOPWRITE ?
Dedup
Yes
No
panic("bad overwrite, hdr=%p exists=%p", (void *)hdr, (void *)exists);
ASSERT(zio->io_prop.zp_nopwrite);
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);
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);
!BP_EQUAL(&zio->io_bp_orig, zio->io_bp) ?
Yes
No
panic("bad nopwrite, hdr=%p exists=%p", (void *)hdr, (void *)exists);
hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
if it's not anon, we are doing a scrub
!exists && hdr->b_state == arc_anon ?
Yes
No
arc_access(hdr, hash_lock);
mutex_exit(hash_lock);
ASSERT(!refcount_is_zero(&hdr->b_refcnt));
callback->awcb_done(zio, buf, callback->awcb_private);
kmem_free(callback, sizeof (arc_write_callback_t));
End

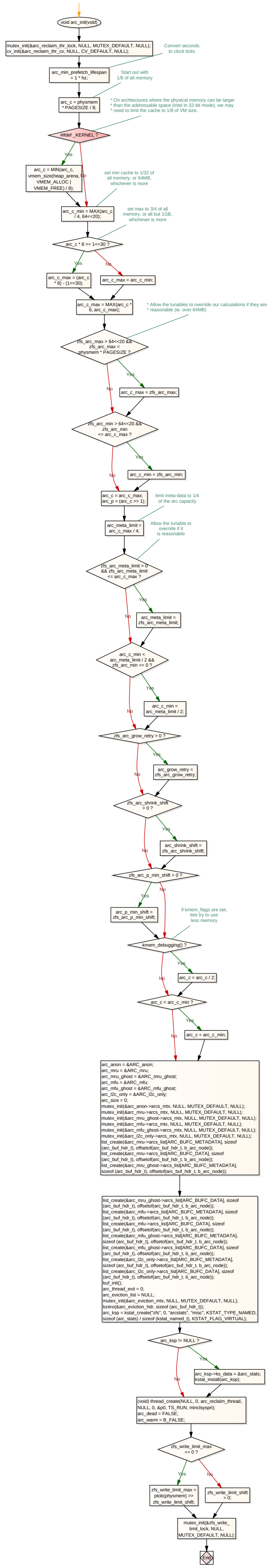

zio_t * arc_write(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *done, void *private, int priority, int zio_flags, const zbookmark_t *zb)
arc_buf_hdr_t *hdr = buf->b_hdr;
arc_write_callback_t *callback;
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);
l2arc ?
Yes
No
hdr->b_flags |= ARC_L2CACHE;
callback = kmem_zalloc(sizeof (arc_write_callback_t), KM_SLEEP);
callback->awcb_ready = ready;
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_done, callback, priority, zio_flags, zb);
return (zio);
End

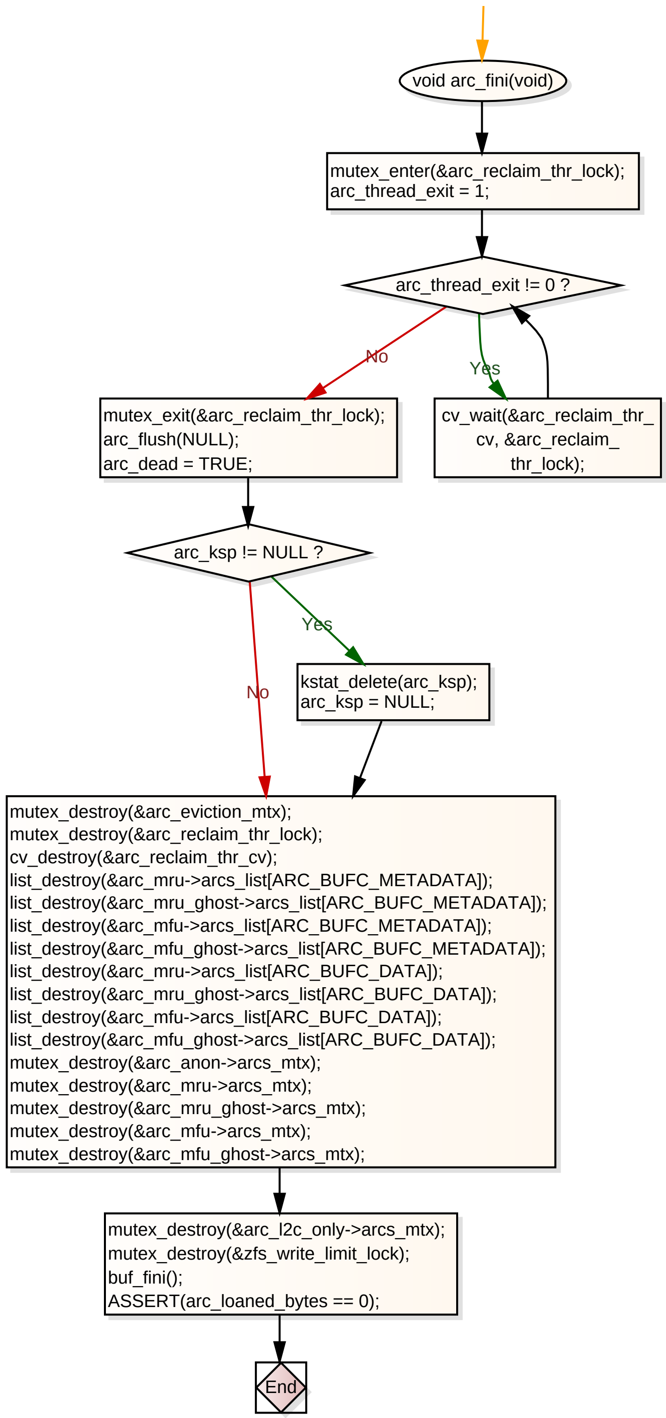

static int arc_memory_throttle (uint64_t reserve, uint64_t inflight_data, uint64_t txg)
#ifdef _KERNEL ?
Yes
No
uint64_t available_memory = ptob(freemem);
static uint64_t page_load = 0;
static uint64_t last_txg = 0;
defined(__i386) ?
Yes
No
available_memory = MIN(available_memory, vmem_size(heap_arena, VMEM_FREE));
available_memory >= zfs_write_limit_max ?
Yes
No
return (0);
txg > last_txg ?
Yes
No
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.
curproc == proc_pageout ?
No
Yes
page_load > 0 && arc_reclaim_needed() ?
memory is low, delay before restarting
Yes
No
ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
return (EAGAIN);
page_load = 0;
arc_size > arc_c_min ?
Yes
No
uint64_t evictable_memory = arc_mru->arcs_lsize[ARC_BUFC_DATA] +
arc_mru->arcs_lsize[ARC_BUFC_METADATA] +
arc_mfu->arcs_lsize[ARC_BUFC_DATA] +
arc_mfu->arcs_lsize[ARC_BUFC_METADATA];
available_memory += MIN(evictable_memory, arc_size - arc_c_min);
page_load > MAX(ptob(minfree), available_memory) / 4 ?
Note: reserve is inflated, so we deflate
inflight_data > available_memory / 4 ?
Yes
No
Yes
No
return (ERESTART);
page_load += reserve / 8;
return (0);
ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
return (ERESTART);
return (0);
End

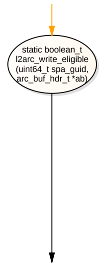

void arc_tempreserve_
clear(uint64_t reserve)
atomic_add_64(&arc_tempreserve, -reserve);
ASSERT((int64_t)arc_tempreserve >= 0);
End



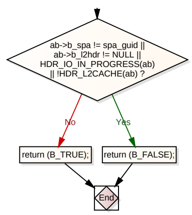
int arc_tempreserve_space (uint64_t reserve, uint64_t txg)
int error;
uint64_t anon_size;
* Once in a while, fail for no reason. Everything should cope.
#ifdef ZFS_DEBUG ?
Yes
No
spa_get_random(10000) == 0 ?
No
Yes
reserve > arc_c/4 && !arc_no_grow ?
dprintf("forcing random failure\n");
return (ERESTART);
No
Yes
arc_c = MIN(arc_c_max, reserve * 4);
reserve > arc_c ?
* 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.
Yes
No
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 therefor need to
* make sure that there is sufficient available memory for this.
return (ENOMEM);
error = arc_memory_throttle(reserve, anon_size, txg) ?
* 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.
Yes
No
reserve + arc_tempreserve + anon_size > arc_c / 2 && anon_size > arc_c / 4 ?
return (error);
Yes
No
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 (ERESTART);
atomic_add_64(&arc_tempreserve, reserve);
return (0);
End

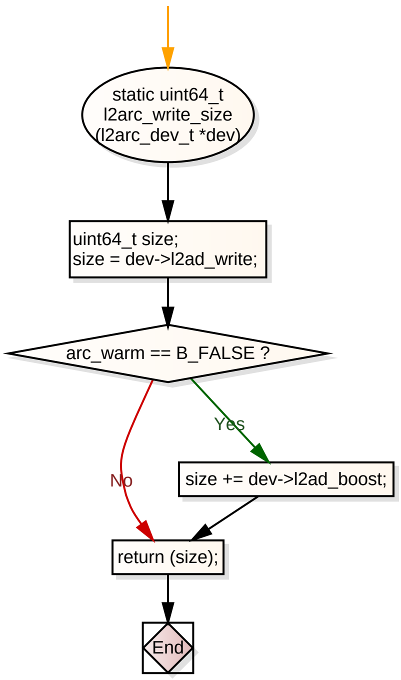

void arc_init(void)
mutex_init(&arc_reclaim_thr_lock, NULL, MUTEX_DEFAULT, NULL);
cv_init(&arc_reclaim_thr_cv, NULL, CV_DEFAULT, NULL);
Convert seconds to clock ticks
arc_min_prefetch_lifespan = 1 * hz;
Start out with 1/8 of all memory
arc_c = physmem * PAGESIZE / 8;
* On architectures where the physical 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.
#ifdef _KERNEL ?
Yes
No
arc_c = MIN(arc_c, vmem_size(heap_arena, VMEM_ALLOC | VMEM_FREE) / 8);
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
arc_c * 8 >= 1<<30 ?
Yes
No
arc_c_max = (arc_c * 8) - (1<<30);
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)
zfs_arc_max > 64<<20 && zfs_arc_max < physmem * PAGESIZE ?
Yes
No
arc_c_max = zfs_arc_max;
zfs_arc_min > 64<<20 && zfs_arc_min <= arc_c_max ?
Yes
No
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
zfs_arc_meta_limit > 0 && zfs_arc_meta_limit <= arc_c_max ?
Yes
No
arc_meta_limit = zfs_arc_meta_limit;
arc_c_min < arc_meta_limit / 2 && zfs_arc_min == 0 ?
Yes
No
arc_c_min = arc_meta_limit / 2;
zfs_arc_grow_retry > 0 ?
Yes
No
arc_grow_retry = zfs_arc_grow_retry;
zfs_arc_shrink_shift > 0 ?
Yes
No
arc_shrink_shift = zfs_arc_shrink_shift;
zfs_arc_p_min_shift > 0 ?
Yes
No
arc_p_min_shift = zfs_arc_p_min_shift;
if kmem_flags are set, lets try to use less memory
kmem_debugging() ?
Yes
No
arc_c = arc_c / 2;
arc_c < arc_c_min ?
Yes
No
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);
arc_ksp != NULL ?
Yes
No
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;
zfs_write_limit_max == 0 ?
Yes
No
zfs_write_limit_max = ptob(physmem) >> zfs_write_limit_shift;
zfs_write_limit_shift = 0;
mutex_init(&zfs_write_limit_lock, NULL, MUTEX_DEFAULT, NULL);
End


void arc_fini(void)
mutex_enter(&arc_reclaim_thr_lock);
arc_thread_exit = 1;
arc_thread_exit != 0 ?
No
Yes
mutex_exit(&arc_reclaim_thr_lock);
arc_flush(NULL);
arc_dead = TRUE;
cv_wait(&arc_reclaim_thr_cv, &arc_reclaim_thr_lock);
arc_ksp != NULL ?
Yes
No
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);
mutex_destroy(&zfs_write_limit_lock);
buf_fini();
ASSERT(arc_loaned_bytes == 0);
End

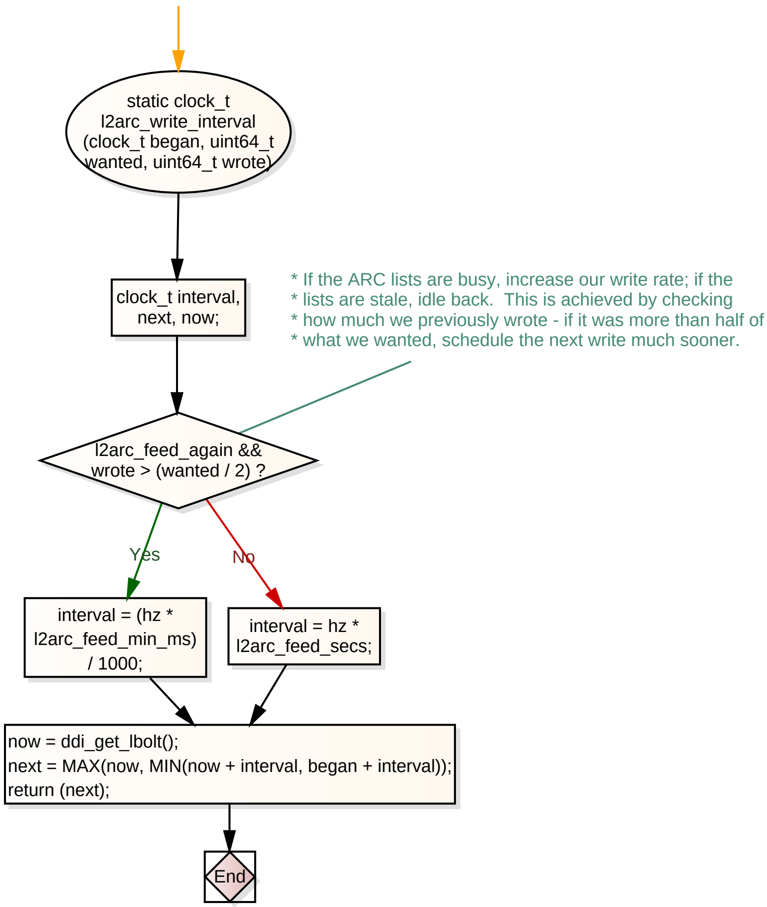


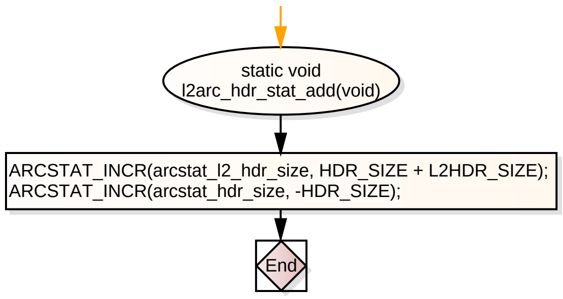

```
* Level 2 ARC
*
* The level 2 ARC (L2ARC) is a cache layer in-between main memory and disk.
* It uses dedicated storage devices to hold cached data, which are populated
* using large infrequent writes.  The main role of this cache is to boost
* the performance of random read workloads.  The intended 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.  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_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.
* 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).
```

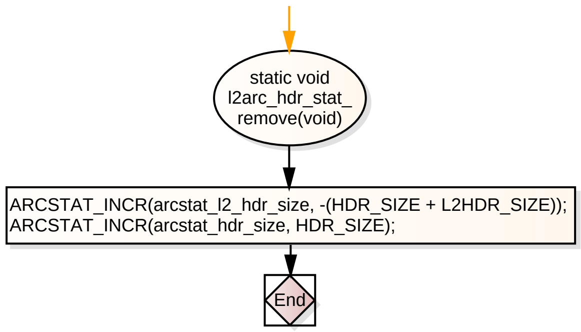


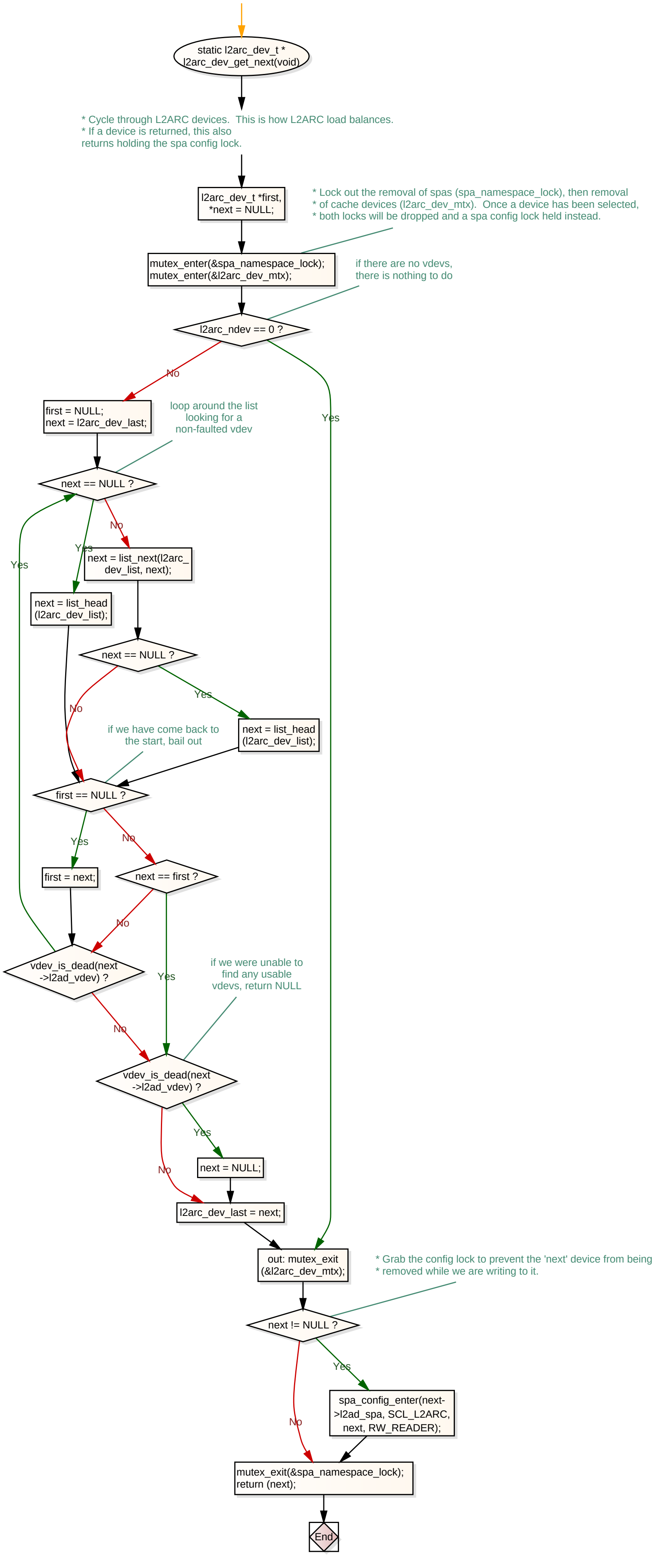


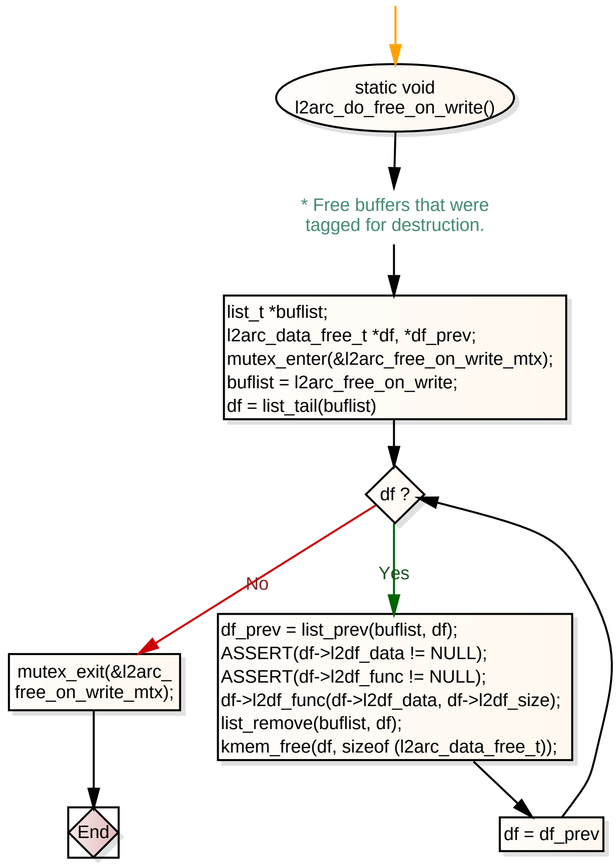
static uint64_t
l2arc_write_size
(l2arc_dev_t *dev)
uint64_t size;
size = dev->l2ad_write;
arc_warm == B_FALSE ?
Yes
No
size += dev->l2ad_boost;
return (size);
End

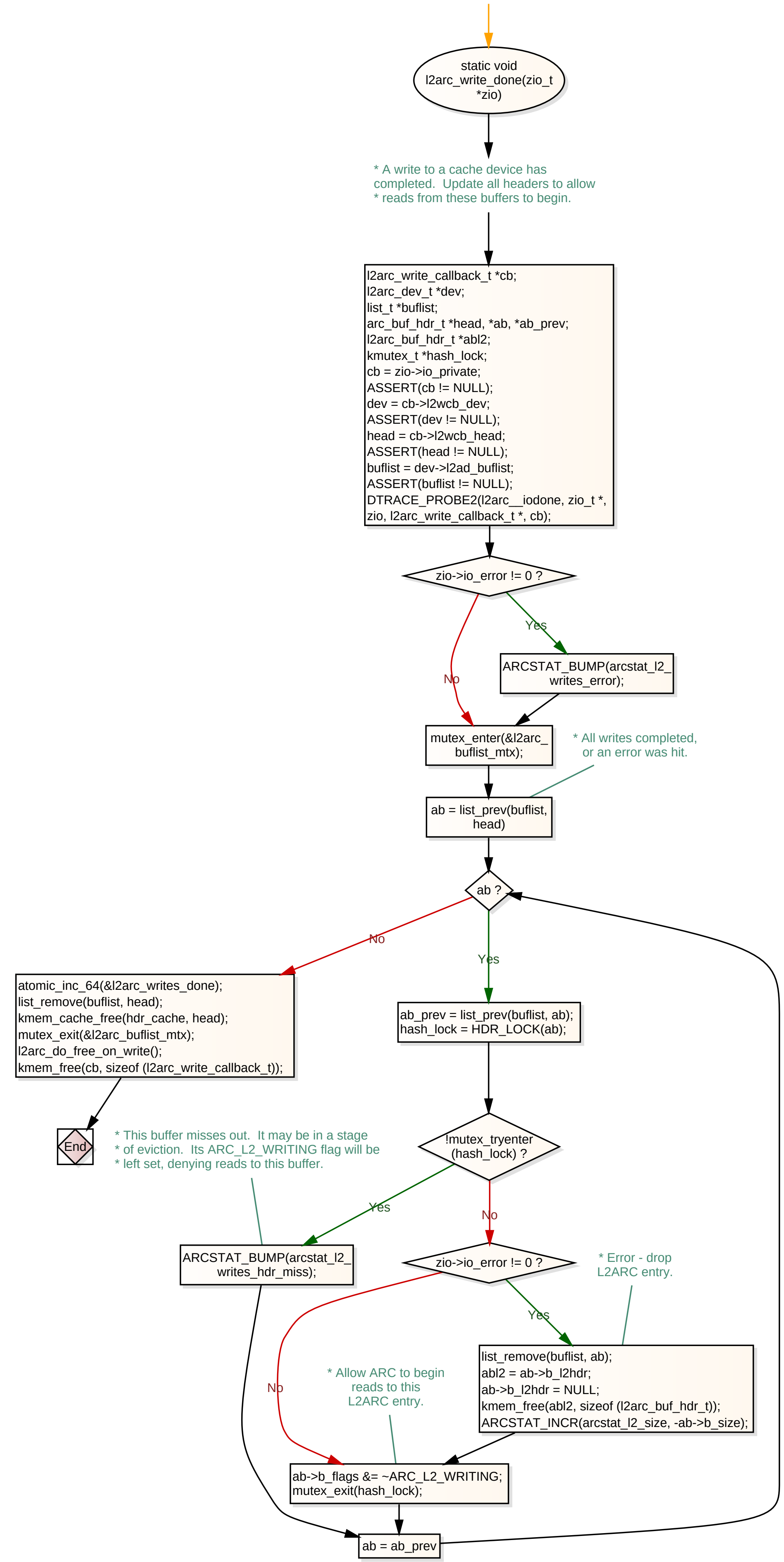

static clock_t l2arc_write_interval (clock_t began, uint64_t wanted, uint64_t wrote)
clock_t interval, next, now;
* If the ARC lists are busy, increase our write rate; if the
* lists are stale, idle back. This is achieved by checking
* how much we previously wrote - if it was more than half of
* what we wanted, schedule the next write much sooner.
l2arc_feed_again && wrote > (wanted / 2) ?
Yes
No
interval = (hz * l2arc_feed_min_ms) / 1000;
interval = hz * l2arc_feed_secs;
now = ddi_get_lbolt();
next = MAX(now, MIN(now + interval, began + interval));
return (next);
End


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

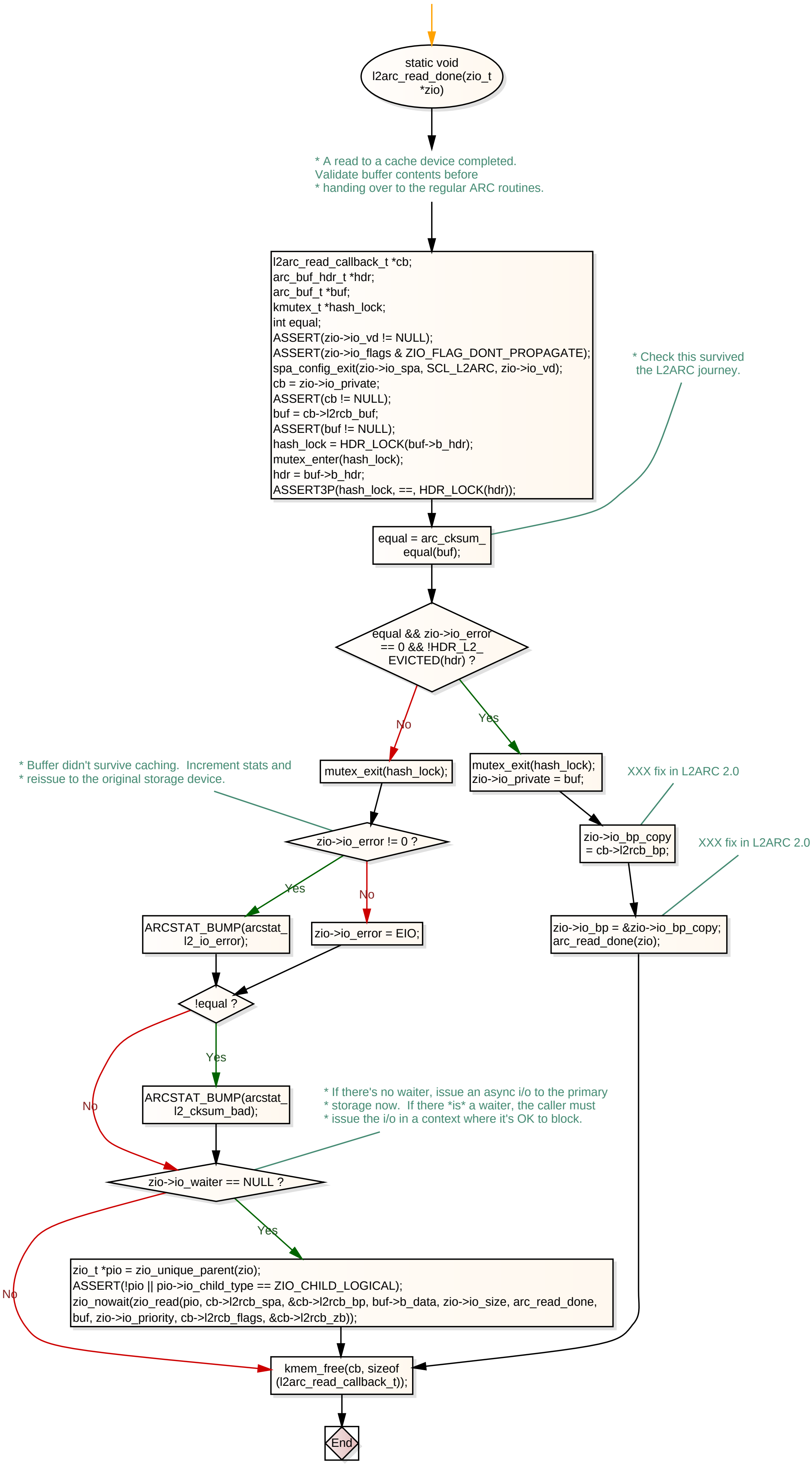

static void
l2arc_hdr_stat_
remove(void)
ARCSTAT_INCR(arcstat_l2_hdr_size, -(HDR_SIZE + L2HDR_SIZE));
ARCSTAT_INCR(arcstat_hdr_size, HDR_SIZE);
End

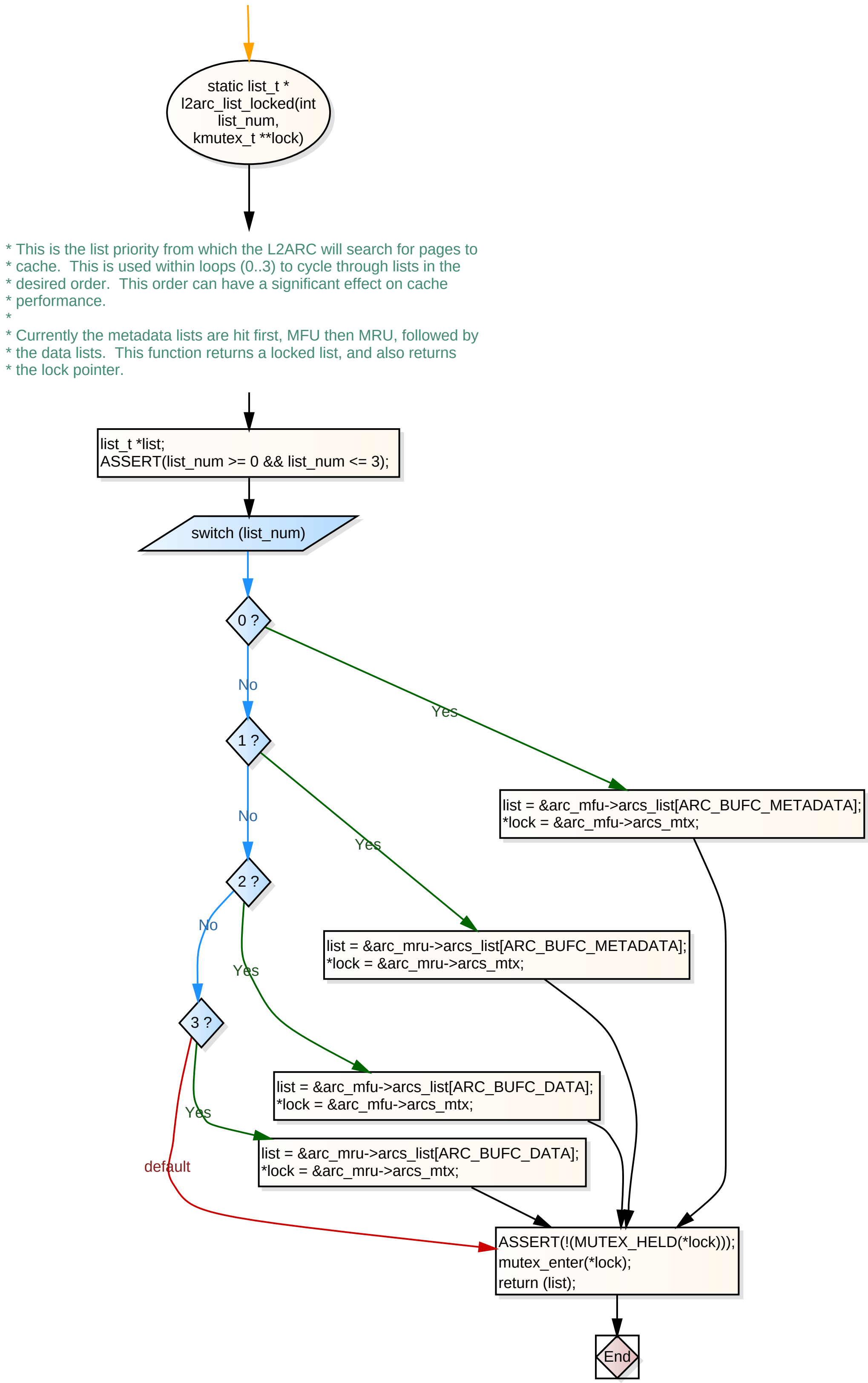

static l2arc_dev_t *
l2arc_dev_get_next(void)
* Cycle through L2ARC devices. This is how L2ARC load balances.
* If a device is returned, this also
returns holding the spa config lock.
l2arc_dev_t *first,
*next = NULL;
* Lock out the removal of spas (spa_namespace_lock), then removal
* of cache devices (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
l2arc_ndev == 0 ?
No
Yes
first = NULL;
next = l2arc_dev_last;
loop around the list
looking for a
non-faulted vdev
next == NULL ?
No
Yes
Yes
next = list_next(l2arc_
dev_list, next);
next = list_head
(l2arc_dev_list);
next == NULL ?
Yes
No
if we have come back to
the start, bail out
next = list_head
(l2arc_dev_list);
first == NULL ?
Yes
No
first = next;
next == first ?
No
Yes
vdev_is_dead(next
->l2ad_vdev) ?
if we were unable to
find any usable
vdevs, return NULL
No
vdev_is_dead(next
->l2ad_vdev) ?
Yes
No
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.
next != NULL ?
Yes
No
spa_config_enter(next-
>l2ad_spa, SCL_L2ARC,
next, RW_READER);
mutex_exit(&spa_namespace_lock);
return (next);
End

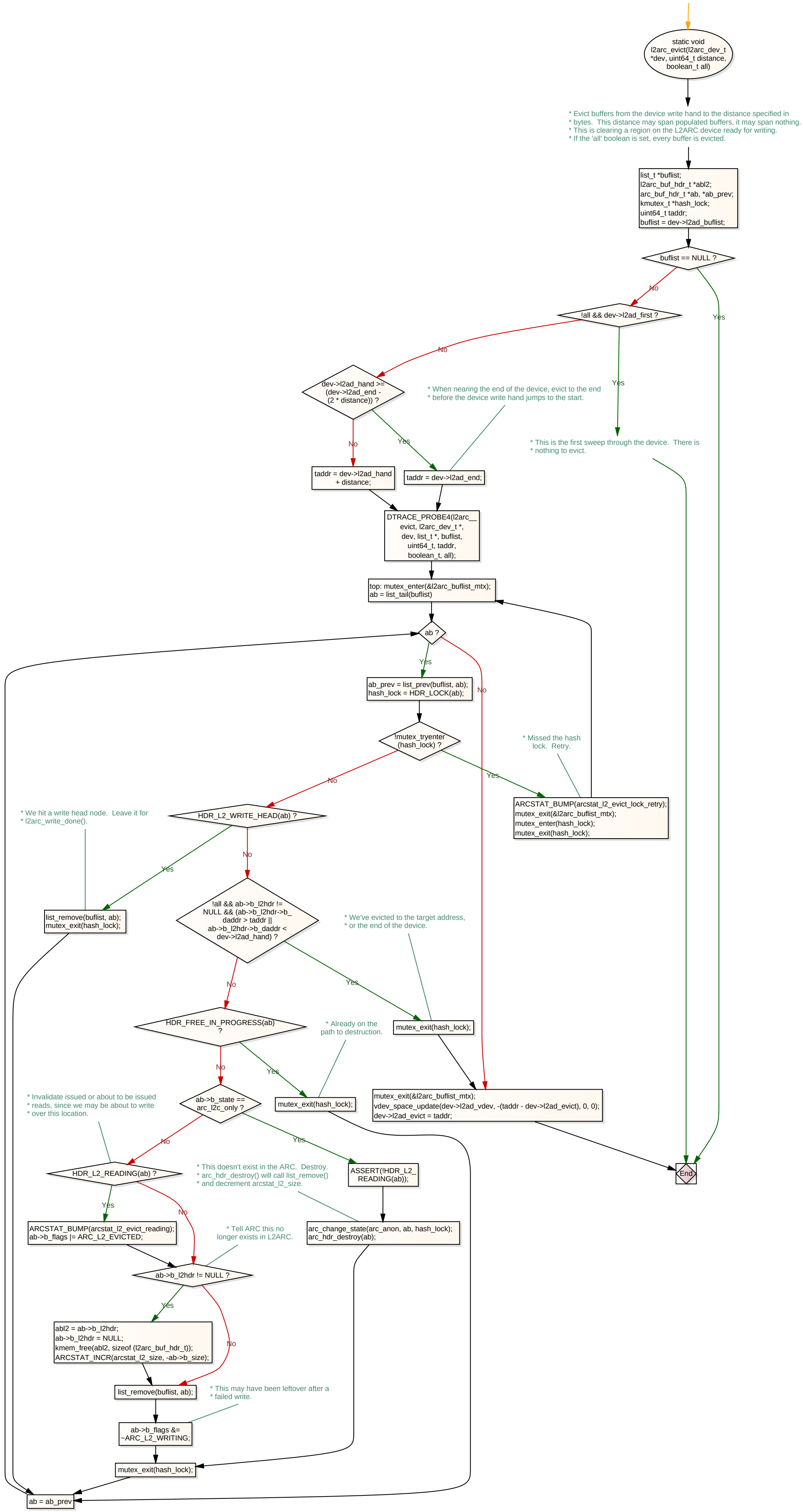

static void
l2arc_do_free_on_write()
* Free buffers that were
tagged for destruction.
list_t *buflist;
l2arc_data_free_t *df, *df_prev;
mutex_enter(&l2arc_free_on_write_mtx);
buflist = l2arc_free_on_write;
df = list_tail(buflist)
df ?
No
Yes
mutex_exit(&l2arc_
free_on_write_mtx);
df_prev = list_prev(buflist, 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));
End
df = df_prev

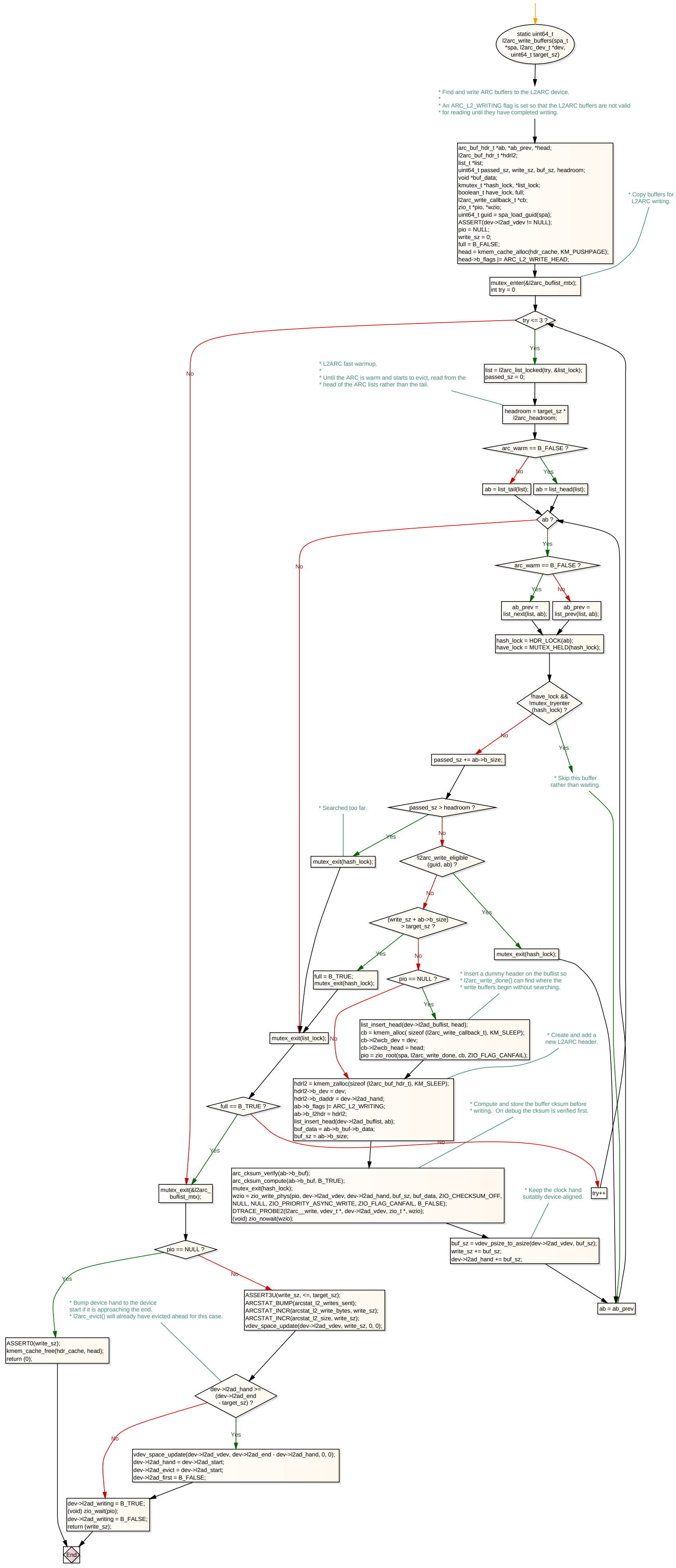


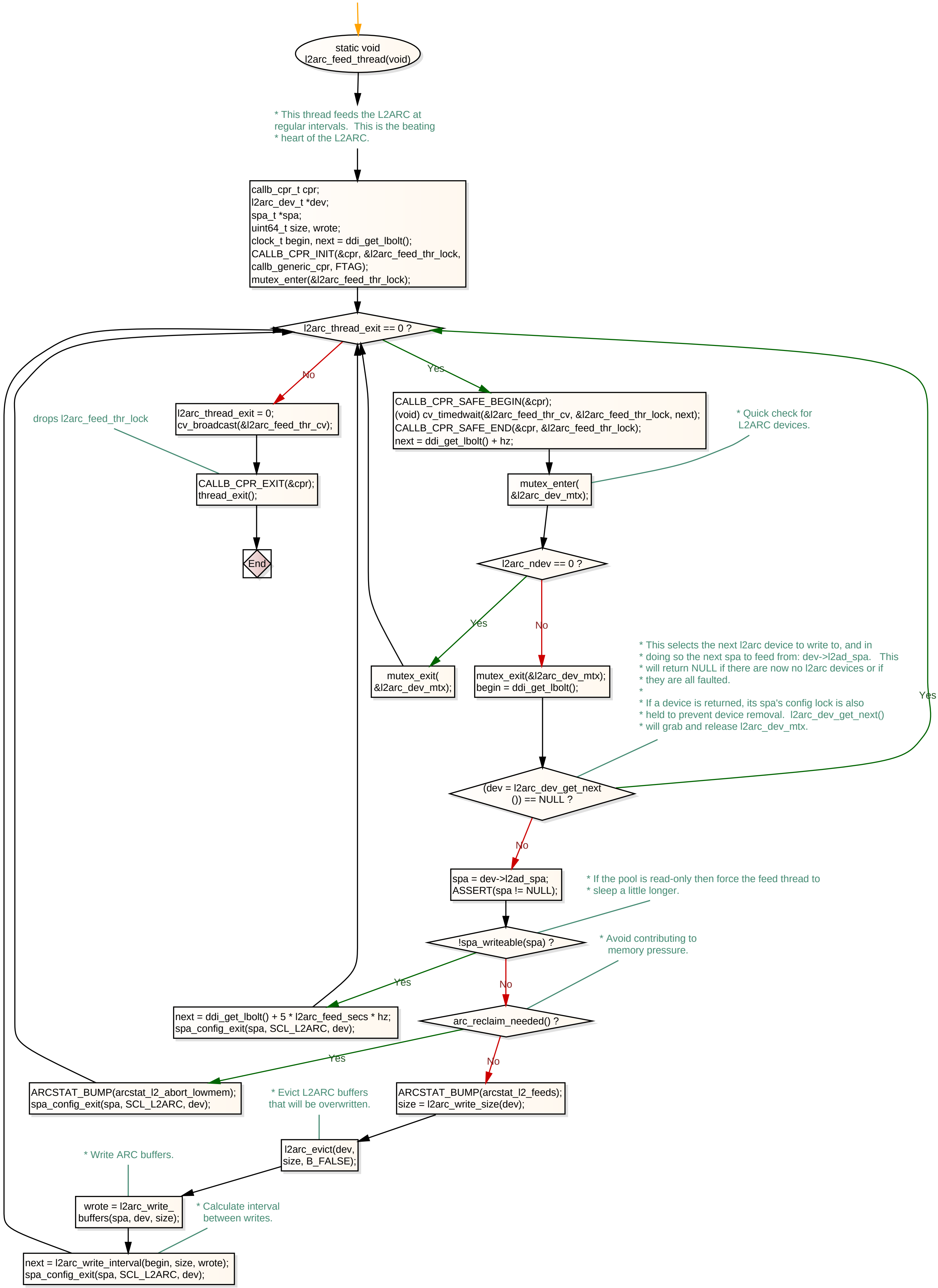
static void
l2arc_write_done(zio_t
*zio)
* A write to a cache device has
completed. Update all headers to allow
* reads from these buffers to begin.
l2arc_write_callback_t *cb;
l2arc_dev_t *dev;
list_t *buflist;
arc_buf_hdr_t *head, *ab, *ab_prev;
l2arc_buf_hdr_t *abl2;
kmutex_t *hash_lock;
cb = zio->io_private;
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);
zio->io_error != 0 ?
Yes
ARCSTAT_BUMP(arcstat_l2_
writes_error);
No
mutex_enter(&l2arc_
buflist_mtx);
* All writes completed,
or an error was hit.
ab = list_prev(buflist,
head)
ab ?
No
Yes
atomic_inc_64(&l2arc_writes_done);
list_remove(buflist, head);
kmem_cache_free(hdr_cache, head);
mutex_exit(&l2arc_buflist_mtx);
l2arc_do_free_on_write();
kmem_free(cb, sizeof (l2arc_write_callback_t));
ab_prev = list_prev(buflist, ab);
hash_lock = HDR_LOCK(ab);
End
* 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.
!mutex_tryenter
(hash_lock) ?
Yes
No
ARCSTAT_BUMP(arcstat_l2_
writes_hdr_miss);
zio->io_error != 0 ?
* Error - drop
L2ARC entry.
Yes
No
list_remove(buflist, ab);
abl2 = ab->b_l2hdr;
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);
ab = ab_prev

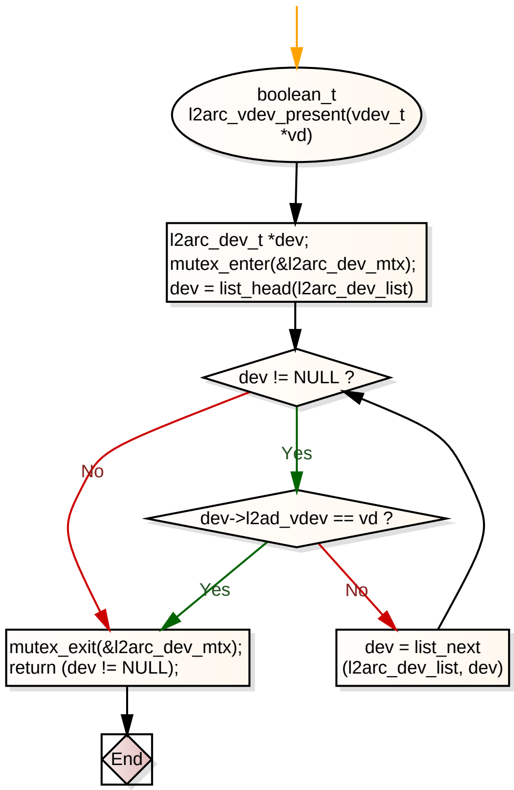

static void l2arc_read_done(zio_t *zio)
* A read to a cache device completed. Validate buffer contents before
* handing over to the regular ARC routines.
l2arc_read_callback_t *cb;
arc_buf_hdr_t *hdr;
arc_buf_t *buf;
kmutex_t *hash_lock;
int equal;
ASSERT(zio->io_vd != NULL);
ASSERT(zio->io_flags & ZIO_FLAG_DONT_PROPAGATE);
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));
* Check this survived the L2ARC journey.
equal = arc_cksum_equal(buf);
equal && zio->io_error == 0 && !HDR_L2_EVICTED(hdr) ?
No
Yes
mutex_exit(hash_lock);
mutex_exit(hash_lock);
zio->io_private = buf;
XXX fix in L2ARC 2.0
zio->io_bp_copy = cb->l2rcb_bp;
XXX fix in L2ARC 2.0
zio->io_bp = &zio->io_bp_copy;
arc_read_done(zio);
* Buffer didn't survive caching. Increment stats and
* reissue to the original storage device.
zio->io_error != 0 ?
Yes
No
ARCSTAT_BUMP(arcstat_l2_io_error);
zio->io_error = EIO;
!equal ?
Yes
No
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.
zio->io_waiter == NULL ?
Yes
No
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));
End

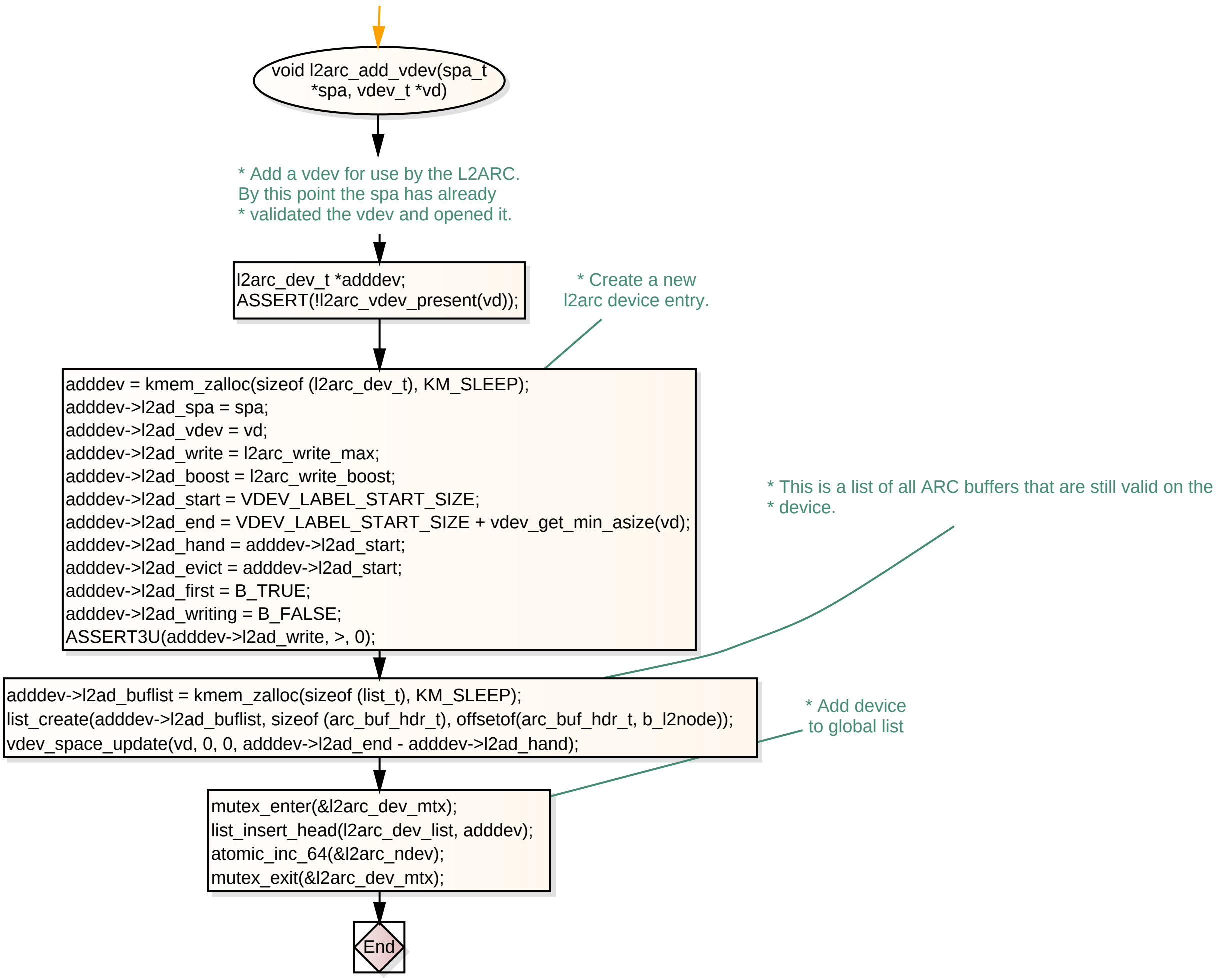

static list_t *
l2arc_list_locked(int
list_num,
kmutex_t **lock)
* This is the list priority from which the L2ARC will search for pages to
* cache. This is used within loops (0..3) to cycle through lists in the
* desired order. This order can have a significant effect on cache
* performance.
*
* Currently 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.
list_t *list;
ASSERT(list_num >= 0 && list_num <= 3);
switch (list_num)
0 ?
Yes
No
1 ?
list = &arc_mfu->arcs_list[ARC_BUFC_METADATA];
*lock = &arc_mfu->arcs_mtx;
No
Yes
2 ?
list = &arc_mru->arcs_list[ARC_BUFC_METADATA];
*lock = &arc_mru->arcs_mtx;
No
Yes
3 ?
list = &arc_mfu->arcs_list[ARC_BUFC_DATA];
*lock = &arc_mfu->arcs_mtx;
Yes
list = &arc_mru->arcs_list[ARC_BUFC_DATA];
*lock = &arc_mru->arcs_mtx;
default
ASSERT(!(MUTEX_HELD(*lock)));
mutex_enter(*lock);
return (list);
End

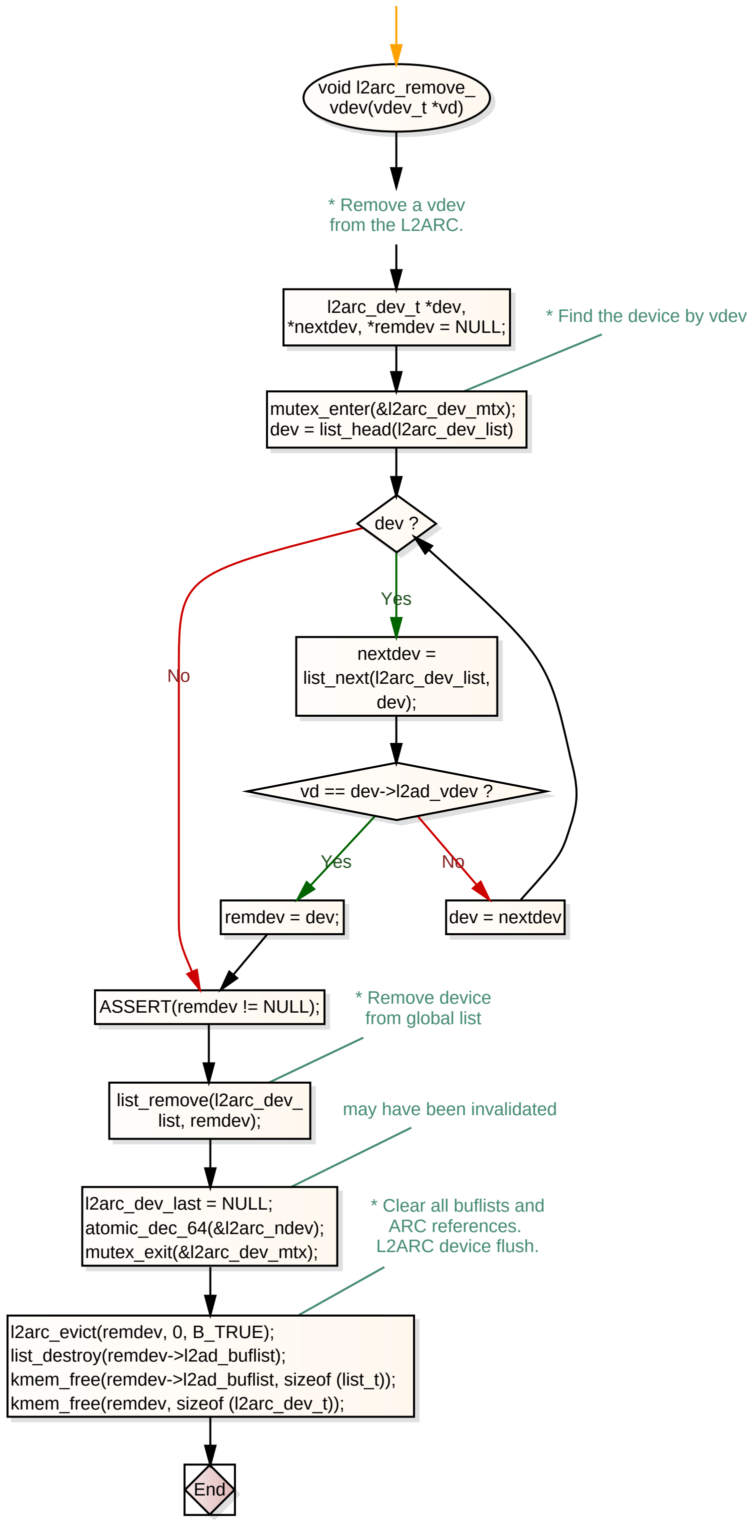

static void l2arc_evict(l2arc_dev_t *dev, uint64_t distance, boolean_t all)
* Evict buffers from the device write hand to the distance specified in
* bytes. This distance may span populated buffers, it may span nothing.
* This is clearing a region on the L2ARC device ready for writing.
* If the 'all' boolean is set, every buffer is evicted.
list_t *buflist;
l2arc_buf_hdr_t *abl2;
arc_buf_hdr_t *ab, *ab_prev;
kmutex_t *hash_lock;
uint64_t taddr;
buflist = dev->l2ad_buflist;
buflist == NULL ?
No
Yes
!all && dev->l2ad_first ?
No
Yes
* This is the first sweep through the device. There is
* nothing to evict.
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.
No
Yes
taddr = dev->l2ad_hand + distance;
taddr = dev->l2ad_end;
DTRACE_PROBE4(l2arc__evict, l2arc_dev_t *, dev, list_t *, buflist, uint64_t, taddr, boolean_t, all);
top: mutex_enter(&l2arc_buflist_mtx);
ab = list_tail(buflist)
ab ?
Yes
No
ab_prev = list_prev(buflist, ab);
hash_lock = HDR_LOCK(ab);
!mutex_tryenter (hash_lock) ?
No
Yes
* 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);
HDR_L2_WRITE_HEAD(ab) ?
* We hit a write head node. Leave it for
* l2arc_write_done().
Yes
No
list_remove(buflist, ab);
mutex_exit(hash_lock);
!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.
Yes
No
mutex_exit(hash_lock);
HDR_FREE_IN_PROGRESS(ab) ?
* Already on the path to destruction.
Yes
No
mutex_exit(hash_lock);
mutex_exit(&l2arc_buflist_mtx);
vdev_space_update(dev->l2ad_vdev, -(taddr - dev->l2ad_evict), 0, 0);
dev->l2ad_evict = taddr;
ab->b_state == arc_l2c_only ?
* Invalidate issued or about to be issued
* reads, since we may be about to write
* over this location.
No
Yes
HDR_L2_READING(ab) ?
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.
Yes
No
ARCSTAT_BUMP(arcstat_l2_evict_reading);
ab->b_flags |= ARC_L2_EVICTED;
arc_change_state(arc_anon, ab, hash_lock);
arc_hdr_destroy(ab);
* Tell ARC this no longer exists in L2ARC.
ab->b_l2hdr != NULL ?
Yes
No
abl2 = ab->b_l2hdr;
ab->b_l2hdr = NULL;
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);
ab = ab_prev
End

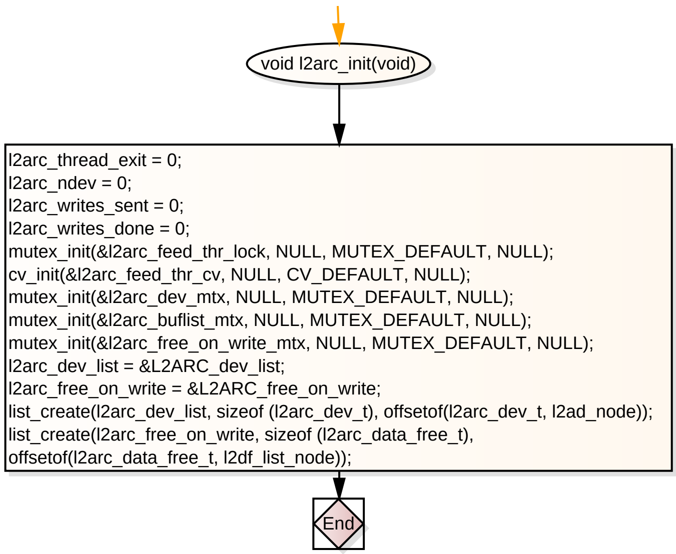

static uint64_t l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz)
* Find and write ARC buffers to the L2ARC device.
*
* An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
* for reading until they have completed writing.
arc_buf_hdr_t *ab, *ab_prev, *head;
l2arc_buf_hdr_t *hdrl2;
list_t *list;
uint64_t passed_sz, write_sz, buf_sz, headroom;
void *buf_data;
kmutex_t *hash_lock, *list_lock;
boolean_t have_lock, full;
l2arc_write_callback_t *cb;
zio_t *pio, *wzio;
uint64_t guid = spa_load_guid(spa);
ASSERT(dev->l2ad_vdev != NULL);
pio = NULL;
write_sz = 0;
full = B_FALSE;
head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
head->b_flags |= ARC_L2_WRITE_HEAD;
* Copy buffers for L2ARC writing.
mutex_enter(&l2arc_buflist_mtx);
int try = 0
try <= 3 ?
No
Yes
list = l2arc_list_locked(try, &list_lock);
passed_sz = 0;
* L2ARC fast warmup.
*
* Until the ARC is warm and starts to evict, read from the
* head of the ARC lists rather than the tail.
headroom = target_sz * l2arc_headroom;
arc_warm == B_FALSE ?
No
Yes
ab = list_tail(list);
ab = list_head(list);
ab ?
No
Yes
arc_warm == B_FALSE ?
Yes
No
ab_prev = list_next(list, ab);
ab_prev = list_prev(list, ab);
hash_lock = HDR_LOCK(ab);
have_lock = MUTEX_HELD(hash_lock);
!have_lock && !mutex_tryenter (hash_lock) ?
No
Yes
passed_sz += ab->b_size;
* Skip this buffer rather than waiting.
* Searched too far.
passed_sz > headroom ?
Yes
No
mutex_exit(hash_lock);
!l2arc_write_eligible (guid, ab) ?
No
Yes
(write_sz + ab->b_size) > target_sz ?
Yes
No
mutex_exit(hash_lock);
full = B_TRUE;
mutex_exit(hash_lock);
pio == NULL ?
* Insert a dummy header on the buflist so
* l2arc_write_done() can find where the
* write buffers begin without searching.
Yes
No
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.
mutex_exit(list_lock);
hdrl2 = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
hdrl2->b_dev = dev;
hdrl2->b_daddr = dev->l2ad_hand;
ab->b_flags |= ARC_L2_WRITING;
ab->b_l2hdr = hdrl2;
list_insert_head(dev->l2ad_buflist, ab);
buf_data = ab->b_buf->b_data;
buf_sz = ab->b_size;
full == B_TRUE ?
No
Yes
* 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);
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);
try++
mutex_exit(&l2arc_buflist_mtx);
* Keep the clock hand suitably device-aligned.
buf_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
write_sz += buf_sz;
dev->l2ad_hand += buf_sz;
ab = ab_prev
pio == NULL ?
Yes
No
ASSERT3U(write_sz, <=, target_sz);
ARCSTAT_BUMP(arcstat_l2_writes_sent);
ARCSTAT_INCR(arcstat_l2_write_bytes, write_sz);
ARCSTAT_INCR(arcstat_l2_size, write_sz);
vdev_space_update(dev->l2ad_vdev, write_sz, 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.
ASSERT0(write_sz);
kmem_cache_free(hdr_cache, head);
return (0);
dev->l2ad_hand >= (dev->l2ad_end - target_sz) ?
No
Yes
vdev_space_update(dev->l2ad_vdev, dev->l2ad_end - dev->l2ad_hand, 0, 0);
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_sz);
End

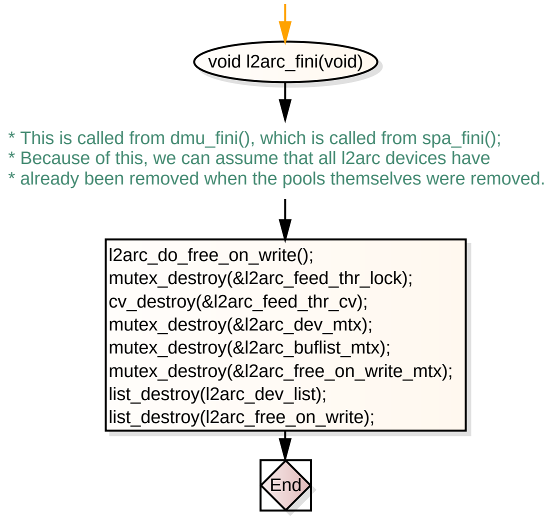

static void
l2arc_feed_thread(void)
* This thread feeds the L2ARC at
regular intervals. This is the beating
* heart of the L2ARC.
callb_cpr_t cpr;
l2arc_dev_t *dev;
spa_t *spa;
uint64_t size, wrote;
clock_t begin, next = ddi_get_lbolt();
CALLB_CPR_INIT(&cpr, &l2arc_feed_thr_lock,
callb_generic_cpr, FTAG);
mutex_enter(&l2arc_feed_thr_lock);
l2arc_thread_exit == 0 ?
No
Yes
l2arc_thread_exit = 0;
cv_broadcast(&l2arc_feed_thr_cv);
drops l2arc_feed_thr_lock
CALLB_CPR_EXIT(&cpr);
thread_exit();
End
CALLB_CPR_SAFE_BEGIN(&cpr);
(void) cv_timedwait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock, next);
CALLB_CPR_SAFE_END(&cpr, &l2arc_feed_thr_lock);
next = ddi_get_lbolt() + hz;
* Quick check for
L2ARC devices.
mutex_enter(
&l2arc_dev_mtx);
l2arc_ndev == 0 ?
Yes
No
mutex_exit(
&l2arc_dev_mtx);
mutex_exit(&l2arc_dev_mtx);
begin = ddi_get_lbolt();
* This selects the next l2arc device to write to, and in
* doing so the next spa to feed from: dev->l2ad_spa. This
* will return NULL if there are now no l2arc devices or if
* they are all faulted.
*
* If a device is returned, its spa's config lock is also
* held to prevent device removal. l2arc_dev_get_next()
* will grab and release l2arc_dev_mtx.
Yes
(dev = l2arc_dev_get_next
()) == NULL ?
No
spa = dev->l2ad_spa;
ASSERT(spa != NULL);
* If the pool is read-only then force the feed thread to
* sleep a little longer.
!spa_writeable(spa) ?
* Avoid contributing to
memory pressure.
Yes
No
next = ddi_get_lbolt() + 5 * l2arc_feed_secs * hz;
spa_config_exit(spa, SCL_L2ARC, dev);
arc_reclaim_needed() ?
Yes
No
ARCSTAT_BUMP(arcstat_l2_abort_lowmem);
spa_config_exit(spa, SCL_L2ARC, dev);
* Evict L2ARC buffers
that will be overwritten.
ARCSTAT_BUMP(arcstat_l2_feeds);
size = l2arc_write_size(dev);
l2arc_evict(dev,
size, B_FALSE);
* Write ARC buffers.
wrote = l2arc_write_
buffers(spa, dev, size);
* Calculate interval
between writes.
next = l2arc_write_interval(begin, size, wrote);
spa_config_exit(spa, SCL_L2ARC, dev);


boolean_t l2arc_vdev_present(vdev_t *vd)
l2arc_dev_t *dev;
mutex_enter(&l2arc_dev_mtx);
dev = list_head(l2arc_dev_list)
dev != NULL ?
No
Yes
dev->l2ad_vdev == vd ?
Yes
No
mutex_exit(&l2arc_dev_mtx);
return (dev != NULL);
dev = list_next (l2arc_dev_list, dev)
End

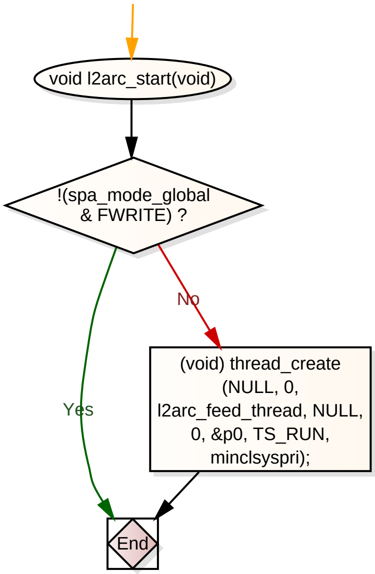

void l2arc_add_vdev(spa_t *spa, vdev_t *vd)
* Add a vdev for use by the L2ARC.
By this point the spa has already
* validated the vdev and opened it.
l2arc_dev_t *adddev;
ASSERT(!l2arc_vdev_present(vd));
* Create a new
l2arc device entry.
adddev = kmem_zalloc(sizeof (l2arc_dev_t), KM_SLEEP);
adddev->l2ad_spa = spa;
adddev->l2ad_vdev = vd;
adddev->l2ad_write = l2arc_write_max;
adddev->l2ad_boost = l2arc_write_boost;
adddev->l2ad_start = VDEV_LABEL_START_SIZE;
adddev->l2ad_end = VDEV_LABEL_START_SIZE + vdev_get_min_asize(vd);
adddev->l2ad_hand = adddev->l2ad_start;
adddev->l2ad_evict = adddev->l2ad_start;
adddev->l2ad_first = B_TRUE;
adddev->l2ad_writing = B_FALSE;
ASSERT3U(adddev->l2ad_write, >, 0);
* This is a list of all ARC buffers that are still valid on the
* device.
adddev->l2ad_buflist = kmem_zalloc(sizeof (list_t), KM_SLEEP);
list_create(adddev->l2ad_buflist, sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_l2node));
vdev_space_update(vd, 0, 0, adddev->l2ad_end - adddev->l2ad_hand);
* Add device
to global list
mutex_enter(&l2arc_dev_mtx);
list_insert_head(l2arc_dev_list, adddev);
atomic_inc_64(&l2arc_ndev);
mutex_exit(&l2arc_dev_mtx);
End

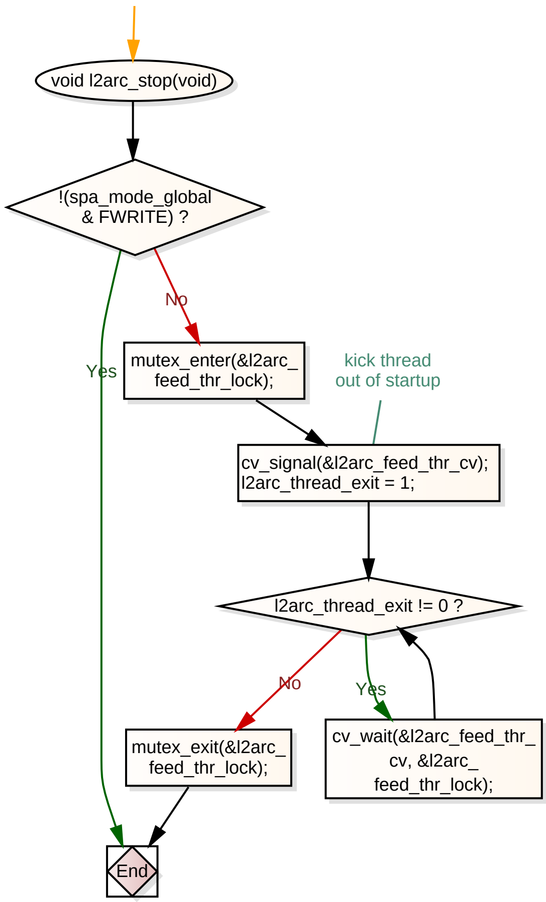

void l2arc_remove_
vdev(vdev_t *vd)
* Remove a vdev
from the L2ARC.
l2arc_dev_t *dev,
*nextdev, *remdev = NULL;
* Find the device by vdev
mutex_enter(&l2arc_dev_mtx);
dev = list_head(l2arc_dev_list)
dev ?
Yes
No
nextdev =
list_next(l2arc_dev_list,
dev);
vd == dev->l2ad_vdev ?
Yes
No
remdev = dev;
dev = nextdev
ASSERT(remdev != NULL);
* Remove device
from global list
list_remove(l2arc_dev_
list, remdev);
may have been invalidated
l2arc_dev_last = NULL;
atomic_dec_64(&l2arc_ndev);
mutex_exit(&l2arc_dev_mtx);
* Clear all buflists and
ARC references.
L2ARC device flush.
l2arc_evict(remdev, 0, B_TRUE);
list_destroy(remdev->l2ad_buflist);
kmem_free(remdev->l2ad_buflist, sizeof (list_t));
kmem_free(remdev, sizeof (l2arc_dev_t));
End


void l2arc_init(void)
l2arc_thread_exit = 0;
l2arc_ndev = 0;
l2arc_writes_sent = 0;
l2arc_writes_done = 0;
mutex_init(&l2arc_feed_thr_lock, NULL, MUTEX_DEFAULT, NULL);
cv_init(&l2arc_feed_thr_cv, NULL, CV_DEFAULT, NULL);
mutex_init(&l2arc_dev_mtx, NULL, MUTEX_DEFAULT, NULL);
mutex_init(&l2arc_buflist_mtx, NULL, MUTEX_DEFAULT, NULL);
mutex_init(&l2arc_free_on_write_mtx, NULL, MUTEX_DEFAULT, NULL);
l2arc_dev_list = &L2ARC_dev_list;
l2arc_free_on_write = &L2ARC_free_on_write;
list_create(l2arc_dev_list, sizeof (l2arc_dev_t), offsetof(l2arc_dev_t, l2ad_node));
list_create(l2arc_free_on_write, sizeof (l2arc_data_free_t),
offsetof(l2arc_data_free_t, l2df_list_node));
End



void l2arc_fini(void)
* This is called from dmu_fini(), which is called from spa_fini();
* Because of this, we can assume that all l2arc devices have
* already been removed when the pools themselves were removed.
l2arc_do_free_on_write();
mutex_destroy(&l2arc_feed_thr_lock);
cv_destroy(&l2arc_feed_thr_cv);
mutex_destroy(&l2arc_dev_mtx);
mutex_destroy(&l2arc_buflist_mtx);
mutex_destroy(&l2arc_free_on_write_mtx);
list_destroy(l2arc_dev_list);
list_destroy(l2arc_free_on_write);
End


void l2arc_start(void)
!(spa_mode_global & FWRITE) ?
No
Yes
(void) thread_create (NULL, 0, l2arc_feed_thread, NULL, 0, &p0, TS_RUN, minclsyspri);
End


void l2arc_stop(void)
!(spa_mode_global & FWRITE) ?
No
Yes
mutex_enter(&l2arc_feed_thr_lock);
kick thread out of startup
cv_signal(&l2arc_feed_thr_cv);
l2arc_thread_exit = 1;
l2arc_thread_exit != 0 ?
No
Yes
mutex_exit(&l2arc_feed_thr_lock);
cv_wait(&l2arc_feed_thr_cv, &l2arc_feed_thr_lock);
End