

```

*****
99349 Fri Aug 11 03:00:25 2017
new/usr/src/uts/common/fs/zfs/metablab.c
7938 disable LBA weighting on files and SSDs
Reviewed by: Yuri Pankov <yuripv@gmx.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
_____unchanged_portion_omitted_____

1613 /*
1614  * Compute a weight -- a selection preference value -- for the given metablab.
1615  * This is based on the amount of free space, the level of fragmentation,
1616  * the LBA range, and whether the metablab is loaded.
1617  */
1618 static uint64_t
1619 metablab_space_weight(metablab_t *msp)
1620 {
1621     metablab_group_t *mg = msp->ms_group;
1622     vdev_t *vd = mg->mg_vd;
1623     uint64_t weight, space;

1625     ASSERT(MUTEX_HELD(&msp->ms_lock));
1626     ASSERT(!vd->vdev_removing);

1628     /*
1629      * The baseline weight is the metablab's free space.
1630      */
1631     space = msp->ms_size - space_map_allocated(msp->ms_sm);

1633     if (metablab_fragmentation_factor_enabled &&
1634         msp->ms_fragmentation != ZFS_FRAG_INVALID) {
1635         /*
1636          * Use the fragmentation information to inversely scale
1637          * down the baseline weight. We need to ensure that we
1638          * don't exclude this metablab completely when it's 100%
1639          * fragmented. To avoid this we reduce the fragmented value
1640          * by 1.
1641          */
1642         space = (space * (100 - (msp->ms_fragmentation - 1))) / 100;

1644         /*
1645          * If space < SPA_MINBLOCKSIZE, then we will not allocate from
1646          * this metablab again. The fragmentation metric may have
1647          * decreased the space to something smaller than
1648          * SPA_MINBLOCKSIZE, so reset the space to SPA_MINBLOCKSIZE
1649          * so that we can consume any remaining space.
1650          */
1651         if (space > 0 && space < SPA_MINBLOCKSIZE)
1652             space = SPA_MINBLOCKSIZE;
1653     }
1654     weight = space;

1656     /*
1657      * Modern disks have uniform bit density and constant angular velocity.
1658      * Therefore, the outer recording zones are faster (higher bandwidth)
1659      * than the inner zones by the ratio of outer to inner track diameter,
1660      * which is typically around 2:1. We account for this by assigning
1661      * higher weight to lower metablabs (multiplier ranging from 2x to 1x).
1662      * In effect, this means that we'll select the metablab with the most
1663      * free bandwidth rather than simply the one with the most free space.
1664      */
1665     if (!vd->vdev_nonrot && metablab_lba_weighting_enabled) {
1666     if (metablab_lba_weighting_enabled) {
1666         weight = 2 * weight - (msp->ms_id * weight) / vd->vdev_ms_count;
1667         ASSERT(weight >= space && weight <= 2 * space);
1668     }

```

```

1670     /*
1671      * If this metablab is one we're actively using, adjust its
1672      * weight to make it preferable to any inactive metablab so
1673      * we'll polish it off. If the fragmentation on this metablab
1674      * has exceed our threshold, then don't mark it active.
1675      */
1676     if (msp->ms_loaded && msp->ms_fragmentation != ZFS_FRAG_INVALID &&
1677         msp->ms_fragmentation <= zfs_metablab_fragmentation_threshold) {
1678         weight |= (msp->ms_weight & METABLAB_ACTIVE_MASK);
1679     }

1681     WEIGHT_SET_SPACEBASED(weight);
1682     return (weight);
1683 }
_____unchanged_portion_omitted_____

```

```

*****
12655 Fri Aug 11 03:00:26 2017
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
7938 disable LBA weighting on files and SSDs
Reviewed by: Yuri Pankov <yuripv@gmx.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
unchanged_portion_omitted

127 /*
128 * Virtual device descriptor
129 */
130 struct vdev {
131     /*
132      * Common to all vdev types.
133      */
134     uint64_t    vdev_id;        /* child number in vdev parent */
135     uint64_t    vdev_guid;      /* unique ID for this vdev */
136     uint64_t    vdev_guid_sum; /* self guid + all child guids */
137     uint64_t    vdev_orig_guid; /* orig. guid prior to remove */
138     uint64_t    vdev_asize;     /* allocatable device capacity */
139     uint64_t    vdev_min_asize; /* min acceptable asize */
140     uint64_t    vdev_max_asize; /* max acceptable asize */
141     uint64_t    vdev_ashift;    /* block alignment shift */
142     uint64_t    vdev_state;     /* see VDEV_STATE_* #defines */
143     uint64_t    vdev_prevstate; /* used when reopening a vdev */
144     vdev_ops_t  *vdev_ops;      /* vdev operations */
145     spa_t        vdev_spa;      /* spa for this vdev */
146     void        *vdev_tsd;     /* type-specific data */
147     vnode_t     *vdev_name_vp;  /* vnode for pathname */
148     vnode_t     *vdev_devid_vp; /* vnode for devid */
149     vdev_t      *vdev_top;      /* top-level vdev */
150     vdev_t      *vdev_parent;   /* parent vdev */
151     vdev_t      **vdev_child;   /* array of children */
152     uint64_t    vdev_children;  /* number of children */
153     vdev_stat_t vdev_stat;     /* virtual device statistics */
154     boolean_t   vdev_expanding; /* expand the vdev? */
155     boolean_t   vdev_reopening; /* reopen in progress? */
156     boolean_t   vdev_nonrot;    /* true if SSD, file, or Virtio */
157     int         vdev_open_error; /* error on last open */
158     kthread_t   *vdev_open_thread; /* thread opening children */
159     uint64_t    vdev_crtxg;     /* txg when top-level was added */

161     /*
162      * Top-level vdev state.
163      */
164     uint64_t    vdev_ms_array;  /* metaslab array object */
165     uint64_t    vdev_ms_shift;  /* metaslab size shift */
166     uint64_t    vdev_ms_count;  /* number of metaslabs */
167     metaslab_group_t *vdev_mg; /* metaslab group */
168     metaslab_t  **vdev_ms;     /* metaslab array */
169     txg_list_t  vdev_ms_list;  /* per-txg dirty metaslab lists */
170     txg_list_t  vdev_dtl_list; /* per-txg dirty DTL lists */
171     txg_node_t  vdev_txg_node; /* per-txg dirty vdev linkage */
172     boolean_t   vdev_remove_wanted; /* async remove wanted? */
173     boolean_t   vdev_probe_wanted; /* async probe wanted? */
174     list_node_t vdev_config_dirty_node; /* config dirty list */
175     list_node_t vdev_state_dirty_node; /* state dirty list */
176     uint64_t    vdev_deflate_ratio; /* deflation ratio (x512) */
177     uint64_t    vdev_islog;     /* is an intent log device */
178     uint64_t    vdev_removing; /* device is being removed? */
179     boolean_t   vdev_ishole;    /* is a hole in the namespace */
180     kmutex_t    vdev_queue_lock; /* protects vdev_queue_depth */
181     uint64_t    vdev_top_zap;

```

```

183     /*

```

```

184     * The queue depth parameters determine how many async writes are
185     * still pending (i.e. allocated by net yet issued to disk) per
186     * top-level (vdev_async_write_queue_depth) and the maximum allowed
187     * (vdev_max_async_write_queue_depth). These values only apply to
188     * top-level vdevs.
189     */
190     uint64_t    vdev_async_write_queue_depth;
191     uint64_t    vdev_max_async_write_queue_depth;

193     /*
194     * Leaf vdev state.
195     */
196     range_tree_t *vdev_dtl[DTL_TYPES]; /* dirty time logs */
197     space_map_t  *vdev_dtl_sm; /* dirty time log space map */
198     txg_node_t   vdev_dtl_node; /* per-txg dirty DTL linkage */
199     uint64_t     vdev_dtl_object; /* DTL object */
200     uint64_t     vdev_psize; /* physical device capacity */
201     uint64_t     vdev_wholeldisk; /* true if this is a whole disk */
202     uint64_t     vdev_offline; /* persistent offline state */
203     uint64_t     vdev_faulted; /* persistent faulted state */
204     uint64_t     vdev_degraded; /* persistent degraded state */
205     uint64_t     vdev_removed; /* persistent removed state */
206     uint64_t     vdev_resilver_txg; /* persistent resilvering state */
207     uint64_t     vdev_nparity; /* number of parity devices for raidz */
208     char         *vdev_path; /* vdev path (if any) */
209     char         *vdev_devid; /* vdev devid (if any) */
210     char         *vdev_physpath; /* vdev device path (if any) */
211     char         *vdev_fru; /* physical FRU location */
212     uint64_t     vdev_not_present; /* not present during import */
213     uint64_t     vdev_unspare; /* unspare when resilvering done */
214     boolean_t    vdev_nowritecache; /* true if flushwritecache failed */
215     boolean_t    vdev_checkremove; /* temporary online test */
216     boolean_t    vdev_forcefault; /* force online fault */
217     boolean_t    vdev_splitting; /* split or repair in progress */
218     boolean_t    vdev_delayed_close; /* delayed device close? */
219     boolean_t    vdev_tmpoffline; /* device taken offline temporarily? */
220     boolean_t    vdev_detached; /* device detached? */
221     boolean_t    vdev_cant_read; /* vdev is failing all reads */
222     boolean_t    vdev_cant_write; /* vdev is failing all writes */
223     boolean_t    vdev_isspare; /* was a hot spare */
224     boolean_t    vdev_isl2cache; /* was a l2cache device */
225     vdev_queue_t vdev_queue; /* I/O deadline schedule queue */
226     vdev_cache_t vdev_cache; /* physical block cache */
227     spa_aux_vdev_t *vdev_aux; /* for l2cache and spares vdevs */
228     zio_t        *vdev_probe_zio; /* root of current probe */
229     vdev_aux_t   vdev_label_aux; /* on-disk aux state */
230     uint64_t     vdev_leaf_zap;

232     /*
233     * For DTrace to work in userland (libzpool) context, these fields must
234     * remain at the end of the structure. DTrace will use the kernel's
235     * CTF definition for 'struct vdev', and since the size of a kmutex_t is
236     * larger in userland, the offsets for the rest of the fields would be
237     * incorrect.
238     */
239     kmutex_t     vdev_dtl_lock; /* vdev_dtl_{map,resilver} */
240     kmutex_t     vdev_stat_lock; /* vdev_stat */
241     kmutex_t     vdev_probe_lock; /* protects vdev_probe_zio */
242 };
unchanged_portion_omitted

```

```
*****
94381 Fri Aug 11 03:00:26 2017
new/usr/src/uts/common/fs/zfs/vdev.c
7938 disable LBA weighting on files and SSDs
Reviewed by: Yuri Pankov <yuripv@gmx.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
_____unchanged_portion_omitted_____
```

```
1104 static void
1105 vdev_open_child(void *arg)
1106 {
1107     vdev_t *vd = arg;
1109     vd->vdev_open_thread = curthread;
1110     vd->vdev_open_error = vdev_open(vd);
1111     vd->vdev_open_thread = NULL;
1112     vd->vdev_parent->vdev_nonrot &= vd->vdev_nonrot;
1113 }
```

```
_____unchanged_portion_omitted_____
```

```
1127 void
1128 vdev_open_children(vdev_t *vd)
1129 {
1130     taskq_t *tq;
1131     int children = vd->vdev_children;
1133     vd->vdev_nonrot = B_TRUE;
1135     /*
1136     * in order to handle pools on top of zvols, do the opens
1137     * in a single thread so that the same thread holds the
1138     * spa_namespace_lock
1139     */
1140     if (vdev_uses_zvols(vd)) {
1141         for (int c = 0; c < children; c++) {
1142             for (int c = 0; c < children; c++)
1143                 vd->vdev_child[c]->vdev_open_error =
1144                     vdev_open(vd->vdev_child[c]);
1145             vd->vdev_nonrot &= vd->vdev_child[c]->vdev_nonrot;
1146         }
1147     }
1148     tq = taskq_create("vdev_open", children, minclsyspri,
1149                     children, TASKQ_PREPOPULATE);
1151     for (int c = 0; c < children; c++)
1152         VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
1153                             TQ_SLEEP) != NULL);
1155     taskq_destroy(tq);
1157     for (int c = 0; c < children; c++)
1158         vd->vdev_nonrot &= vd->vdev_child[c]->vdev_nonrot;
1159 }
```

```
_____unchanged_portion_omitted_____
```

```

*****
23663 Fri Aug 11 03:00:27 2017
new/usr/src/uts/common/fs/zfs/vdev_disk.c
7938 disable LBA weighting on files and SSDs
Reviewed by: Yuri Pankov <yuripv@gmx.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012, 2015 by Delphix. All rights reserved.
24 * Copyright 2016 Nexenta Systems, Inc. All rights reserved.
25 * Copyright (c) 2013 Joyent, Inc. All rights reserved.
26 * Copyright (c) 2017 James S Blachly, MD <james.blachly@gmail.com>
27 */

29 #include <sys/zfs_context.h>
30 #include <sys/spa_impl.h>
31 #include <sys/refcount.h>
32 #include <sys/vdev_disk.h>
33 #include <sys/vdev_impl.h>
34 #include <sys/abd.h>
35 #include <sys/fs/zfs.h>
36 #include <sys/zio.h>
37 #include <sys/sunldi.h>
38 #include <sys/efi_partition.h>
39 #include <sys/fm/fs/zfs.h>

41 /*
42  * Virtual device vector for disks.
43  */

45 extern ldi_ident_t zfs_li;

47 static void vdev_disk_close(vdev_t *);

49 typedef struct vdev_disk_ldi_cb {
50     list_node_t      lcb_next;
51     ldi_callback_id_t lcb_id;
52 } vdev_disk_ldi_cb_t;
unchanged_portion_omitted

246 /*
247  * We want to be loud in DEBUG kernels when DKIOCGMEDIAINFOEXT fails, or when
248  * even a fallback to DKIOCGMEDIAINFO fails.
249  */
250 #ifdef DEBUG

```

```

251 #define VDEV_DEBUG(...) cmn_err(CE_NOTE, __VA_ARGS__)
252 #else
253 #define VDEV_DEBUG(...) /* Nothing... */
254 #endif

256 static int
257 vdev_disk_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,
258               uint64_t *ashift)
259 {
260     spa_t *spa = vd->vdev_spa;
261     vdev_disk_t *dvd = vd->vdev_tsd;
262     ldi_ev_cookie_t ecookie;
263     vdev_disk_ldi_cb_t *lcb;
264     union {
265         struct dk_minfo_ext ude;
266         struct dk_minfo ud;
267     } dks;
268     struct dk_minfo_ext *dkmext = &dks.ude;
269     struct dk_minfo *dkm = &dks.ud;
270     int error;
271     dev_t dev;
272     int otyp;
273     boolean_t validate_devid = B_FALSE;
274     ddi_devid_t devid;
275     uint64_t capacity = 0, blksize = 0, pbsize;
276     int device_rotational;

278     /*
279      * We must have a pathname, and it must be absolute.
280      */
281     if (vd->vdev_path == NULL || vd->vdev_path[0] != '/') {
282         vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
283         return (SET_ERROR(EINVAL));
284     }

286     /*
287      * Reopen the device if it's not currently open. Otherwise,
288      * just update the physical size of the device.
289      */
290     if (dvd != NULL) {
291         if (dvd->vd_ldi_offline && dvd->vd_lh == NULL) {
292             /*
293              * If we are opening a device in its offline notify
294              * context, the LDI handle was just closed. Clean
295              * up the LDI event callbacks and free vd->vdev_tsd.
296              */
297             vdev_disk_free(vd);
298         } else {
299             ASSERT(vd->vdev_reopening);
300             goto skip_open;
301         }
302     }

304     /*
305      * Create vd->vdev_tsd.
306      */
307     vdev_disk_alloc(vd);
308     dvd = vd->vdev_tsd;

310     /*
311      * When opening a disk device, we want to preserve the user's original
312      * intent. We always want to open the device by the path the user gave
313      * us, even if it is one of multiple paths to the same device. But we
314      * also want to be able to survive disks being removed/recabled.
315      * Therefore the sequence of opening devices is:
316

```

```

317 * 1. Try opening the device by path. For legacy pools without the
318 * 'whole_disk' property, attempt to fix the path by appending 's0'.
319 *
320 * 2. If the devid of the device matches the stored value, return
321 * success.
322 *
323 * 3. Otherwise, the device may have moved. Try opening the device
324 * by the devid instead.
325 */
326 if (vd->vdev_devid != NULL) {
327     if (ddi_devid_str_decode(vd->vdev_devid, &dvd->vd_devid,
328         &dvd->vd_minor) != 0) {
329         vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
330         return (SET_ERROR(EINVAL));
331     }
332 }
334 error = EINVAL; /* presume failure */
336 if (vd->vdev_path != NULL) {
338     if (vd->vdev_wholedisk == -1ULL) {
339         size_t len = strlen(vd->vdev_path) + 3;
340         char *buf = kmem_alloc(len, KM_SLEEP);
342         (void) snprintf(buf, len, "%s0", vd->vdev_path);
344         error = ldi_open_by_name(buf, spa_mode(spa), kcred,
345             &dvd->vd_lh, zfs_li);
346         if (error == 0) {
347             spa_strfree(vd->vdev_path);
348             vd->vdev_path = buf;
349             vd->vdev_wholedisk = 1ULL;
350         } else {
351             kmem_free(buf, len);
352         }
353     }
355     /*
356     * If we have not yet opened the device, try to open it by the
357     * specified path.
358     */
359     if (error != 0) {
360         error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
361             kcred, &dvd->vd_lh, zfs_li);
362     }
364     /*
365     * Compare the devid to the stored value.
366     */
367     if (error == 0 && vd->vdev_devid != NULL &&
368         ldi_get_devid(dvd->vd_lh, &devid) == 0) {
369         if (ddi_devid_compare(devid, dvd->vd_devid) != 0) {
370             error = SET_ERROR(EINVAL);
371             (void) ldi_close(dvd->vd_lh, spa_mode(spa),
372                 kcred);
373             dvd->vd_lh = NULL;
374         }
375         ddi_devid_free(devid);
376     }
378     /*
379     * If we succeeded in opening the device, but 'vdev_wholedisk'
380     * is not yet set, then this must be a slice.
381     */
382     if (error == 0 && vd->vdev_wholedisk == -1ULL)

```

```

383         vd->vdev_wholedisk = 0;
384     }
386     /*
387     * If we were unable to open by path, or the devid check fails, open by
388     * devid instead.
389     */
390     if (error != 0 && vd->vdev_devid != NULL) {
391         error = ldi_open_by_devid(dvd->vd_devid, dvd->vd_minor,
392             spa_mode(spa), kcred, &dvd->vd_lh, zfs_li);
393     }
395     /*
396     * If all else fails, then try opening by physical path (if available)
397     * or the logical path (if we failed due to the devid check). While not
398     * as reliable as the devid, this will give us something, and the higher
399     * level vdev validation will prevent us from opening the wrong device.
400     */
401     if (error) {
402         if (vd->vdev_devid != NULL)
403             validate_devid = B_TRUE;
405         if (vd->vdev_physpath != NULL &&
406             (dev = ddi_pathname_to_dev_t(vd->vdev_physpath)) != NODEV)
407             error = ldi_open_by_dev(&dev, OTYP_BLK, spa_mode(spa),
408                 kcred, &dvd->vd_lh, zfs_li);
410         /*
411         * Note that we don't support the legacy auto-wholedisk support
412         * as above. This hasn't been used in a very long time and we
413         * don't need to propagate its oddities to this edge condition.
414         */
415         if (error && vd->vdev_path != NULL)
416             error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
417                 kcred, &dvd->vd_lh, zfs_li);
418     }
420     if (error) {
421         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
422         return (error);
423     }
425     /*
426     * Now that the device has been successfully opened, update the devid
427     * if necessary.
428     */
429     if (validate_devid && spa_writeable(spa) &&
430         ldi_get_devid(dvd->vd_lh, &devid) == 0) {
431         if (ddi_devid_compare(devid, dvd->vd_devid) != 0) {
432             char *vd_devid;
434             vd_devid = ddi_devid_str_encode(devid, dvd->vd_minor);
435             zfs_dbgmsg("vdev %s: update devid from %s, "
436                 "to %s", vd->vdev_path, vd->vdev_devid, vd_devid);
437             spa_strfree(vd->vdev_devid);
438             vd->vdev_devid = spa_strdup(vd_devid);
439             ddi_devid_str_free(vd_devid);
440         }
441         ddi_devid_free(devid);
442     }
444     /*
445     * Once a device is opened, verify that the physical device path (if
446     * available) is up to date.
447     */
448     if (ldi_get_dev(dvd->vd_lh, &dev) == 0 &&

```

```

449     ldi_get_otyp(dvd->vd_lh, &otyp) == 0) {
450         char *physpath, *minorname;

452         physpath = kmem_alloc(MAXPATHLEN, KM_SLEEP);
453         minorname = NULL;
454         if (ddi_dev_pathname(dev, otyp, physpath) == 0 &&
455             ldi_get_minor_name(dvd->vd_lh, &minorname) == 0 &&
456             (vd->vdev_physpath == NULL ||
457              strcmp(vd->vdev_physpath, physpath) != 0)) {
458             if (vd->vdev_physpath)
459                 spa_strfree(vd->vdev_physpath);
460             (void) strlcat(physpath, ":", MAXPATHLEN);
461             (void) strlcat(physpath, minorname, MAXPATHLEN);
462             vd->vdev_physpath = spa_strdup(physpath);
463         }
464         if (minorname)
465             kmem_free(minorname, strlen(minorname) + 1);
466         kmem_free(physpath, MAXPATHLEN);
467     }

469     /*
470      * Register callbacks for the LDI offline event.
471      */
472     if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_OFFLINE, &ecookie) ==
473         LDI_EV_SUCCESS) {
474         lcb = kmem_zalloc(sizeof (vdev_disk_ldi_cb_t), KM_SLEEP);
475         list_insert_tail(&dvd->vd_ldi_cbs, lcb);
476         (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
477             &vdev_disk_off_callb, (void *) vd, &lcb->lcb_id);
478     }

480     /*
481      * Register callbacks for the LDI degrade event.
482      */
483     if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_DEGRADE, &ecookie) ==
484         LDI_EV_SUCCESS) {
485         lcb = kmem_zalloc(sizeof (vdev_disk_ldi_cb_t), KM_SLEEP);
486         list_insert_tail(&dvd->vd_ldi_cbs, lcb);
487         (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
488             &vdev_disk_dgrd_callb, (void *) vd, &lcb->lcb_id);
489     }
490 skip_open:
491     /*
492      * Determine the actual size of the device.
493      */
494     if (ldi_get_size(dvd->vd_lh, psize) != 0) {
495         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
496         return (SET_ERROR(EINVAL));
497     }

499     *max_psize = *psize;

501     /*
502      * Determine the device's minimum transfer size.
503      * If the ioctl isn't supported, assume DEV_BSIZE.
504      */
505     if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFOEXT,
506         (intptr_t)dkmext, FKIOCTL, kcred, NULL)) == 0) {
507         capacity = dkmext->dki_capacity - 1;
508         blkksz = dkmext->dki_lbsize;
509         pbsize = dkmext->dki_pbsize;
510     } else if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFO,
511         (intptr_t)dkm, FKIOCTL, kcred, NULL)) == 0) {
512         VDEV_DEBUG(
513             "vdev_disk_open(\"%s\"): fallback to DKIOCGMEDIAINFO\n",
514             vd->vdev_path);

```

```

515         capacity = dkm->dki_capacity - 1;
516         blkksz = dkm->dki_lbsize;
517         pbsize = blkksz;
518     } else {
519         VDEV_DEBUG("vdev_disk_open(\"%s\"): "
520             "both DKIOCGMEDIAINFO{,EXT} calls failed, %d\n",
521             vd->vdev_path, error);
522         pbsize = DEV_BSIZE;
523     }

525     *ashift = highbit64(MAX(pbsize, SPA_MINBLOCKSIZE)) - 1;

527     if (vd->vdev_wholedisk == 1) {
528         int wce = 1;

530         if (error == 0) {
531             /*
532              * If we have the capability to expand, we'd have
533              * found out via success from DKIOCGMEDIAINFO{,EXT}.
534              * Adjust max_psize upward accordingly since we know
535              * we own the whole disk now.
536              */
537             *max_psize = capacity * blkksz;
538         }

540         /*
541          * Since we own the whole disk, try to enable disk write
542          * caching. We ignore errors because it's OK if we can't do it.
543          */
544         (void) ldi_ioctl(dvd->vd_lh, DKIOCSETWCE, (intptr_t)&wce,
545             FKIOCTL, kcred, NULL);
546     }

548     /*
549      * Inform the ZIO pipeline if we are non-rotational
550      */
551     device_rotational = ldi_prop_get_int(dvd->vd_lh, LDI_DEV_T_ANY,
552         "device-rotational", 1);
553     vd->vdev_nonrot = (device_rotational ? B_FALSE : B_TRUE);

555     /*
556      * Clear the nowritecache bit, so that on a vdev_reopen() we will
557      * try again.
558      */
559     vd->vdev_nowritecache = B_FALSE;

561     return (0);
562 }

```

unchanged_portion_omitted

```

*****
6697 Fri Aug 11 03:00:27 2017
new/usr/src/uts/common/fs/zfs/vdev_file.c
7938 disable LBA weighting on files and SSDs
Reviewed by: Yuri Pankov <yuripv@gmx.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
*****
_____unchanged_portion_omitted_____

52 static int
53 vdev_file_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,
54               uint64_t *ashift)
55 {
56     vdev_file_t *vf;
57     vnode_t *vp;
58     vattr_t vattr;
59     int error;

61     /*
62      * Rotational optimizations only make sense on block devices
63      */
64     vd->vdev_nonrot = B_TRUE;

66     /*
67      * We must have a pathname, and it must be absolute.
68      */
69     if (vd->vdev_path == NULL || vd->vdev_path[0] != '/') {
70         vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
71         return (SET_ERROR(EINVAL));
72     }

74     /*
75      * Reopen the device if it's not currently open.  Otherwise,
76      * just update the physical size of the device.
77      */
78     if (vd->vdev_tsd != NULL) {
79         ASSERT(vd->vdev_reopening);
80         vf = vd->vdev_tsd;
81         goto skip_open;
82     }

84     vf = vd->vdev_tsd = kmem_zalloc(sizeof (vdev_file_t), KM_SLEEP);

86     /*
87      * We always open the files from the root of the global zone, even if
88      * we're in a local zone.  If the user has gotten to this point, the
89      * administrator has already decided that the pool should be available
90      * to local zone users, so the underlying devices should be as well.
91      */
92     ASSERT(vd->vdev_path != NULL && vd->vdev_path[0] == '/');
93     error = vn_openat(vd->vdev_path + 1, UIO_SYSSPACE,
94                     spa_mode(vd->vdev_spa) | FOFPMAX, 0, &vp, 0, 0, rootdir, -1);

96     if (error) {
97         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
98         return (error);
99     }

101     vf->vf_vnode = vp;

103 #ifdef _KERNEL
104     /*
105      * Make sure it's a regular file.
106      */
107     if (vp->v_type != VREG) {
108         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;

```

```

109         return (SET_ERROR(ENODEV));
110     }
111 #endif

113 skip_open:
114     /*
115      * Determine the physical size of the file.
116      */
117     vattr.va_mask = AT_SIZE;
118     error = VOP_GETATTR(vf->vf_vnode, &vattr, 0, kcred, NULL);
119     if (error) {
120         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
121         return (error);
122     }

124     *max_psize = *psize = vattr.va_size;
125     *ashift = SPA_MINBLOCKSHIFT;

127     return (0);
128 }
_____unchanged_portion_omitted_____

```