

new/usr/src/uts/common/fs/zfs/metaslab.c

1

```
*****  
98778 Sun Mar 5 21:45:56 2017  
new/usr/src/uts/common/fs/zfs/metaslab.c  
7938 Port ZOL #3712 disable LBA weighting on files and SSDs  
*****  
_____unchanged_portion_omitted_____  
1600 /*  
1601 * Compute a weight -- a selection preference value -- for the given metaslab.  
1602 * This is based on the amount of free space, the level of fragmentation,  
1603 * the LBA range, and whether the metaslab is loaded.  
1604 */  
1605 static uint64_t  
1606 metaslab_space_weight(metaslab_t *msp)  
1607 {  
1608     metaslab_group_t *mg = msp->ms_group;  
1609     vdev_t *vd = mg->mg_vd;  
1610     uint64_t weight, space;  
1611  
1612     ASSERT(MUTEX_HELD(&msp->ms_lock));  
1613     ASSERT(!vd->vdev_removing);  
1614  
1615     /*  
1616      * The baseline weight is the metaslab's free space.  
1617      */  
1618     space = msp->ms_size - space_map_allocated(msp->ms_sm);  
1619  
1620     if (metaslab_fragmentation_factor_enabled &&  
1621         msp->ms_fragmentation != ZFS_FRAG_INVALID) {  
1622         /*  
1623          * Use the fragmentation information to inversely scale  
1624          * down the baseline weight. We need to ensure that we  
1625          * don't exclude this metaslab completely when it's 100%  
1626          * fragmented. To avoid this we reduce the fragmented value  
1627          * by 1.  
1628          */  
1629         space = (space * (100 - (msp->ms_fragmentation - 1))) / 100;  
1630  
1631         /*  
1632          * If space < SPA_MINBLOCKSIZE, then we will not allocate from  
1633          * this metaslab again. The fragmentation metric may have  
1634          * decreased the space to something smaller than  
1635          * SPA_MINBLOCKSIZE, so reset the space to SPA_MINBLOCKSIZE  
1636          * so that we can consume any remaining space.  
1637          */  
1638         if (space > 0 && space < SPA_MINBLOCKSIZE)  
1639             space = SPA_MINBLOCKSIZE;  
1640     }  
1641     weight = space;  
1642  
1643     /*  
1644      * Modern disks have uniform bit density and constant angular velocity.  
1645      * Therefore, the outer recording zones are faster (higher bandwidth)  
1646      * than the inner zones by the ratio of outer to inner track diameter,  
1647      * which is typically around 2:1. We account for this by assigning  
1648      * higher weight to lower metaslabs (multiplier ranging from 2x to 1x).  
1649      * In effect, this means that we'll select the metaslab with the most  
1650      * free bandwidth rather than simply the one with the most free space.  
1651      */  
1652     if (!vd->vdev_nonrot && metaslab_lba_weighting_enabled) {  
1653         if (metaslab_lba_weighting_enabled) {  
1654             weight = 2 * weight - (msp->ms_id * weight) / vd->vdev_ms_count;  
1655             ASSERT(weight >= space && weight <= 2 * space);  
1656         }  
1657     }/*
```

new/usr/src/uts/common/fs/zfs/metaslab.c

2

```
1658     * If this metaslab is one we're actively using, adjust its  
1659     * weight to make it preferable to any inactive metaslab so  
1660     * we'll polish it off. If the fragmentation on this metaslab  
1661     * has exceed our threshold, then don't mark it active.  
1662     */  
1663     if (msp->ms_loaded && msp->ms_fragmentation != ZFS_FRAG_INVALID &&  
1664         msp->ms_fragmentation <= zfs_metaslab_fragmentation_threshold) {  
1665             weight |= (msp->ms_weight & METASLAB_ACTIVE_MASK);  
1666         }  
1667  
1668     WEIGHT_SET_SPACEBASED(weight);  
1669     return (weight);  
1670 }  
_____unchanged_portion_omitted_____
```

```
*****  
12639 Sun Mar 5 21:45:57 2017  
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h  
7938 Port ZOL #3712 disable LBA weighting on files and SSDs  
*****  
_____ unchanged_portion_omitted _____
```

```
126 /*  
127 * Virtual device descriptor  
128 */  
129 struct vdev {  
130     /*  
131     * Common to all vdev types.  
132     */  
133     uint64_t    vdev_id;        /* child number in vdev parent */  
134     uint64_t    vdev_guid;      /* unique ID for this vdev */  
135     uint64_t    vdev_guid_sum;  /* self guid + all child guids */  
136     uint64_t    vdev_orig_guid; /* orig. guid prior to remove */  
137     uint64_t    vdev_asize;     /* allocatable device capacity */  
138     uint64_t    vdev_min_asize; /* min acceptable asize */  
139     uint64_t    vdev_max_asize; /* max acceptable asize */  
140     uint64_t    vdev_ashift;    /* block alignment shift */  
141     uint64_t    vdev_state;    /* see VDEV_STATE_* #defines */  
142     uint64_t    vdev_prevstate; /* used when reopening a vdev */  
143     vdev_ops_t  *vdev_ops;     /* vdev operations */  
144     spa_t       *vdev_spa;      /* spa for this vdev */  
145     void        *vdev_tsd;      /* type-specific data */  
146     vnode_t     *vdev_name_vp;  /* vnode for pathname */  
147     vnode_t     *vdev_devid_vp; /* vnode for devid */  
148     vdev_t      *vdev_top;     /* top-level vdev */  
149     vdev_t      *vdev_parent;   /* parent vdev */  
150     vdev_t      **vdev_child;   /* array of children */  
151     uint64_t    vdev_children;  /* number of children */  
152     vdev_stat_t vdev_stat;    /* virtual device statistics */  
153     boolean_t   vdev_expanding; /* expand the vdev? */  
154     boolean_t   vdev_reopening; /* reopen in progress? */  
155     boolean_t   vdev_nonrot;    /* true if SSD, file, or Virtio */  
156     int         vdev_open_error; /* error on last open */  
157     kthread_t   *vdev_open_thread; /* thread opening children */  
158     uint64_t    vdev_crtxg;    /* txg when top-level was added */  
159  
160     /*  
161     * Top-level vdev state.  
162     */  
163     uint64_t    vdev_ms_array;  /* metaslab array object */  
164     uint64_t    vdev_ms_shift;  /* metaslab size shift */  
165     uint64_t    vdev_ms_count;  /* number of metaslabs */  
166     metaslab_group_t *vdev_mg;  /* metaslab group */  
167     metaslab_t   **vdev_ms;    /* metaslab array */  
168     txg_list_t  vdev_ms_list;  /* per-txg dirty metaslab lists */  
169     txg_list_t  vdev_dtl_list; /* per-txg dirty DTL lists */  
170     txg_node_t  vdev_txg_node; /* per-txg dirty vdev linkage */  
171     boolean_t   vdev_remove_wanted; /* async remove wanted? */  
172     boolean_t   vdev_probe_wanted; /* async probe wanted? */  
173     list_node_t vdev_config_dirty_node; /* config dirty list */  
174     list_node_t vdev_state_dirty_node; /* state dirty list */  
175     uint64_t    vdev_deflate_ratio; /* deflation ratio (x512) */  
176     uint64_t    vdev_islog;      /* is an intent log device */  
177     uint64_t    vdev_removing;  /* device is being removed? */  
178     boolean_t   vdev_ishole;    /* is a hole in the namespace */  
179     kmutex_t    vdev_queue_lock; /* protects vdev_queue_depth */  
180     uint64_t    vdev_top_zap;  
181  
182     /*  
183     * The queue depth parameters determine how many async writes are  
184     * still pending (i.e. allocated by net yet issued to disk) per
```

```
185     * top-level (vdev_async_write_queue_depth) and the maximum allowed  
186     * (vdev_max_async_write_queue_depth). These values only apply to  
187     * top-level vdevs.  
188     */  
189     uint64_t    vdev_async_write_queue_depth;  
190     uint64_t    vdev_max_async_write_queue_depth;  
191  
192     /*  
193     * Leaf vdev state.  
194     */  
195     range_tree_t *vdev_dtl[DTL_TYPES]; /* dirty time logs */  
196     space_map_t  *vdev_dtl_sm; /* dirty time log space map */  
197     txg_node_t   vdev_dtl_node; /* per-txg dirty DTL linkage */  
198     uint64_t    vdev_dtl_object; /* DTL object */  
199     uint64_t    vdev_psize; /* physical device capacity */  
200     uint64_t    vdev_wholedisk; /* true if this is a whole disk */  
201     uint64_t    vdev_offline; /* persistent offline state */  
202     uint64_t    vdev_faulted; /* persistent faulted state */  
203     uint64_t    vdev_degraded; /* persistent degraded state */  
204     uint64_t    vdev_removed; /* persistent removed state */  
205     uint64_t    vdev_resilver_txg; /* persistent resilvering state */  
206     uint64_t    vdev_nparity; /* number of parity devices for raidz */  
207     char        *vdev_path; /* vdev path (if any) */  
208     char        *vdev_devid; /* vdev devid (if any) */  
209     char        *vdev_physpath; /* vdev device path (if any) */  
210     char        *vdev_fru; /* physical FRU location */  
211     uint64_t    vdev_not_present; /* not present during import */  
212     uint64_t    vdev_unspare; /* unspare when resilvering done */  
213     boolean_t   vdev_nowritecache; /* true if flushwritecache failed */  
214     boolean_t   vdev_checkremove; /* temporary online test */  
215     boolean_t   vdev_forcefault; /* force online fault */  
216     boolean_t   vdev_splitting; /* split or repair in progress */  
217     boolean_t   vdev_delayed_close; /* delayed device close? */  
218     boolean_t   vdev_tmppofline; /* device taken offline temporarily? */  
219     boolean_t   vdev_detached; /* device detached? */  
220     boolean_t   vdev_cant_read; /* vdev is failing all reads */  
221     boolean_t   vdev_cant_write; /* vdev is failing all writes */  
222     boolean_t   vdev_isspare; /* was a hot spare */  
223     boolean_t   vdev_isl2cache; /* was a l2cache device */  
224     vdev_queue_t vdev_queue; /* I/O deadline schedule queue */  
225     vdev_cache_t vdev_cache; /* physical block cache */  
226     spa_aux_vdev_t *vdev_aux; /* for l2cache and spares vdevs */  
227     zio_t       *vdev_probe_zio; /* root of current probe */  
228     vdev_aux_t  vdev_label_aux; /* on-disk aux state */  
229     uint64_t    vdev_leaf_zap;  
230  
231     /*  
232     * For DTrace to work in userland (libzpool) context, these fields must  
233     * remain at the end of the structure. DTrace will use the kernel's  
234     * CTF definition for 'struct vdev', and since the size of a kmutex_t is  
235     * larger in userland, the offsets for the rest of the fields would be  
236     * incorrect.  
237     */  
238     kmutex_t    vdev_dtl_lock; /* vdev_dtl_{map,resilver} */  
239     kmutex_t    vdev_stat_lock; /* vdev_stat */  
240     kmutex_t    vdev_probe_lock; /* protects vdev_probe_zio */  
241 };  
_____ unchanged_portion_omitted _____
```

```
*****  
93729 Sun Mar 5 21:45:57 2017  
new/usr/src/uts/common/fs/zfs/vdev.c  
7938 Port ZOL #3712 disable LBA weighting on files and SSDs  
*****  
_____unchanged_portion_omitted_____
```

```
1101 static void  
1102 vdev_open_child(void *arg)  
1103 {  
1104     vdev_t *vd = arg;  
  
1106     vd->vdev_open_thread = curthread;  
1107     vd->vdev_open_error = vdev_open(vd);  
1108     vd->vdev_open_thread = NULL;  
1109     vd->vdev_parent->vdev_nonrot &= vd->vdev_nonrot;  
1110 }  
_____unchanged_portion_omitted_____  
  
1124 void  
1125 vdev_open_children(vdev_t *vd)  
1126 {  
1127     taskq_t *tq;  
1128     int children = vd->vdev_children;  
  
1130     vd->vdev_nonrot = B_TRUE;  
  
1132     /*  
1133      * in order to handle pools on top of zvols, do the opens  
1134      * in a single thread so that the same thread holds the  
1135      * spa_namespace_lock  
1136      */  
1137     if (vdev_uses_zvols(vd)) {  
1138         for (int c = 0; c < children; c++) {  
1139             for (int c = 0; c < children; c++)  
1140                 vd->vdev_child[c]->vdev_open_error =  
1141                 vdev_open(vd->vdev_child[c]);  
1142             vd->vdev_nonrot &= vd->vdev_child[c]->vdev_nonrot;  
1143         }  
1144     }  
1145     tq = taskq_create("vdev_open", children, minclsy wholepri,  
1146     children, children, TASKQ_PREPOPULATE);  
  
1148     for (int c = 0; c < children; c++)  
1149         VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],  
1150             TQ_SLEEP) != NULL);  
  
1152     taskq_destroy(tq);  
  
1154     for (int c = 0; c < children; c++)  
1155         vd->vdev_nonrot &= vd->vdev_child[c]->vdev_nonrot;  
1156 }  
_____unchanged_portion_omitted_____
```

new/usr/src/uts/common/fs/zfs/vdev_disk.c

1

```
*****
23899 Sun Mar 5 21:45:57 2017
new/usr/src/uts/common/fs/zfs/vdev_disk.c
7938 Port ZOL #3712 disable LBA weighting on files and SSDs
*****  
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/fs/zfs.h>  
35 #include <sys/zio.h>  
36 #include <sys/sunldi.h>  
37 #include <sys/efi_partition.h>  
38 #include <sys/fm/fs/zfs.h>  
  
40 /*  
41 * Virtual device vector for disks.  
42 */  
  
44 extern ldi_ident_t zfs_li;  
  
46 static void vdev_disk_close(vdev_t *);  
  
48 typedef struct vdev_disk_ldi_cb {  
49     list_node_t lcb_next;  
50     ldi_callback_id_t lcb_id;  
51 } vdev_disk_ldi_cb_t;  
_____unchanged_portion_omitted_____  
  
245 /*  
246 * We want to be loud in DEBUG kernels when DKIOCGMEDIAINFOEXT fails, or when  
247 * even a fallback to DKIOCGMEDIAINFO fails.  
248 */  
249 #ifdef DEBUG  
250 #define VDEV_DEBUG(...) cmn_err(CE_NOTE, __VA_ARGS__)  
251 #else  
252 #define VDEV_DEBUG(...) /* Nothing... */
```

new/usr/src/uts/common/fs/zfs/vdev_disk.c

2

```
253 #endif  
  
255 static int  
256 vdev_disk_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,  
257     uint64_t *ashift)  
258 {  
259     spa_t *spa = vd->vdev_spa;  
260     vdev_disk_t *dvd = vd->vdev_tsd;  
261     ldi_ev_cookie_t ecookie;  
262     vdev_disk_ldi_cb_t *lcb;  
263     union {  
264         struct dk_minfo_ext ude;  
265         struct dk_minfo ud;  
266     } dks;  
267     struct dk_minfo_ext *dkmext = &dks.udc;  
268     struct dk_minfo *dkm = &dks.ud;  
269     int error;  
270     dev_t dev;  
271     int otyp;  
272     boolean_t validate_devid = B_FALSE;  
273     ddi_devid_t devid;  
274     uint64_t capacity = 0, blksz = 0, pbsize;  
275     int device_solid_state;  
276     char *vendorp; /* will point to inquiry-vendor-id */  
  
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     }
333     error = EINVAL; /* presume failure */
334
335     if (vd->vdev_path != NULL) {
336
337         if (vd->vdev_wholedisk == -1ULL) {
338             size_t len = strlen(vd->vdev_path) + 3;
339             char *buf = kmem_alloc(len, KM_SLEEP);
340
341             (void) snprintf(buf, len, "%s0", vd->vdev_path);
342
343             error = ldi_open_by_name(buf, spa_mode(spa), kcred,
344                 &dvd->vd_lh, zfs_li);
345             if (error == 0) {
346                 spa_strfree(vd->vdev_path);
347                 vd->vdev_path = buf;
348                 vd->vdev_wholedisk = 1ULL;
349             } else {
350                 kmem_free(buf, len);
351             }
352         }
353
354         /*
355          * If we have not yet opened the device, try to open it by the
356          * specified path.
357         */
358         if (error != 0) {
359             error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
360                 kcrid, &dvd->vd_lh, zfs_li);
361         }
362
363         /*
364          * Compare the devid to the stored value.
365         */
366         if (error == 0 && vd->vdev_devid != NULL &&
367             ldi_get_devid(dvd->vd_lh, &devid) == 0) {
368             if (ddi_devid_compare(devid, dvd->vd_devid) != 0) {
369                 error = SET_ERROR(EINVAL);
370                 (void) ldi_close(dvd->vd_lh, spa_mode(spa),
371                     kcrid);
372                 dvd->vd_lh = NULL;
373             }
374             ddi_devid_free(devid);
375         }
376
377         /*
378          * If we succeeded in opening the device, but 'vdev_wholedisk'
379          * is not yet set, then this must be a slice.
380         */
381         if (error == 0 && vd->vdev_wholedisk == -1ULL)
382             vd->vdev_wholedisk = 0;
383     }
384

```

```

386
387     /*
388      * If we were unable to open by path, or the devid check fails, open by
389      * devid instead.
390      */
391     if (error != 0 && vd->vdev_devid != NULL) {
392         error = ldi_open_by_devid(dvd->vd_devid, dvd->vd_minor,
393             spa_mode(spa), kcrid, &dvd->vd_lh, zfs_li);
394     }
395
396     /*
397      * If all else fails, then try opening by physical path (if available)
398      * or the logical path (if we failed due to the devid check). While not
399      * as reliable as the devid, this will give us something, and the higher
400      * level vdev validation will prevent us from opening the wrong device.
401      */
402     if (error) {
403         if (vd->vdev_devid != NULL)
404             validate_devid = B_TRUE;
405
406         if (vd->vdev_physpath != NULL &&
407             (dev = ddi_pathname_to_dev_t(vd->vdev_physpath)) != NODEV)
408             error = ldi_open_by_dev(&dev, OTYP_BLK, spa_mode(spa),
409                 kcrid, &dvd->vd_lh, zfs_li);
410
411         /*
412          * Note that we don't support the legacy auto-wholedisk support
413          * as above. This hasn't been used in a very long time and we
414          * don't need to propagate its oddities to this edge condition.
415         */
416         if (error && vd->vdev_path != NULL)
417             error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
418                 kcrid, &dvd->vd_lh, zfs_li);
419
420         if (error) {
421             vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
422             return (error);
423         }
424
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;
433
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         }
443
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_strdup(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 }
468 */
469 /* Register callbacks for the LDI offline event.
470 */
471 if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_OFFLINE, &ecookie) ==
472     LDI_EV_SUCCESS) {
473     lcb = kmem_zalloc(sizeof(vdev_disk_ldi_cb_t), KM_SLEEP);
474     list_insert_tail(&dvd->vd_ldi_cbs, lcb);
475     (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
476                                     &vdev_disk_off_callb, (void *) vd, &lcb->lcb_id);
477 }
478 */
479 /*
480 * Register callbacks for the LDI degrade event.
481 */
482 if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_DEGRADE, &ecookie) ==
483     LDI_EV_SUCCESS) {
484     lcb = kmem_zalloc(sizeof(vdev_disk_ldi_cb_t), KM_SLEEP);
485     list_insert_tail(&dvd->vd_ldi_cbs, lcb);
486     (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
487                                     &vdev_disk_dgrd_callb, (void *) vd, &lcb->lcb_id);
488 }
489 skip_open:
490 /*
491 * Determine the actual size of the device.
492 */
493 if (ldi_get_size(dvd->vd_lh, psize) != 0) {
494     vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
495     return (SET_ERROR(EINVAL));
496 }
497 */
498 *max_psize = *psize;
499 */
500 /*
501 * Determine the device's minimum transfer size.
502 * If the ioctl isn't supported, assume DEV_BSIZE.
503 */
504 if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFOEXT,
505     (intptr_t)dkmext, FKIOCTL, kcred, NULL)) == 0) {
506     capacity = dkmext->dki_capacity - 1;
507     blksize = dkmext->dki_lbsize;
508     pbsize = dkmext->dki_pbsize;
509 } else if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFO,
510     (intptr_t)dkm, FKIOCTL, kcred, NULL)) == 0) {
511     VDEV_DEBUG(
512         "vdev_disk_open(\"%s\"): fallback to DKIOCGMEDIAINFO\n",
513         vd->vdev_path);
514     capacity = dkm->dki_capacity - 1;
515     blksize = dkm->dki_lbsize;
516 }

```

```

517             pbsize = blksize;
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         }
524         *ashift = highbit64(MAX(pbsize, SPA_MINBLOCKSIZE)) - 1;
525         if (vd->vdev_wholedisk == 1) {
526             int wce = 1;
527             if (error == 0) {
528                 /*
529                  * If we have the capability to expand, we'd have
530                  * found out via success from DKIOCGMEDIAINFO{,EXT}.
531                  * Adjust max_psize upward accordingly since we know
532                  * we own the whole disk now.
533                  */
534                 *max_psize = capacity * blksize;
535             }
536             /*
537              * Since we own the whole disk, try to enable disk write
538              * caching. We ignore errors because it's OK if we can't do it.
539              */
540             (void) ldi_ioctl(dvd->vd_lh, DKIOCSETWCE, (intptr_t)&wce,
541                             FKIOCTL, kcred, NULL);
542         }
543         /*
544          * Inform the ZIO pipeline if we are non-rotational:
545          * 1. Check if device is SSD
546          * 2. If not SSD, check if device is Virtio
547          */
548         device_solid_state = ldi_prop_get_int(dvd->vd_lh, LDI_DEV_T_ANY,
549                                             "device-solid-state", 0);
550         vd->vdev_nonrot = (device_solid_state ? B_TRUE : B_FALSE);
551
552         if (device_solid_state == 0 &&
553             ldi_prop_exists(dvd->vd_lh, LDI_DEV_T_ANY, "inquiry-vendor-id")) {
554             ldi_prop_lookup_string(dvd->vd_lh, LDI_DEV_T_ANY,
555                                     "inquiry-vendor-id", &vendorp);
556             if (strncmp(vendorp, "Virtio", 6) == 0)
557                 vd->vdev_nonrot = B_TRUE;
558             ddi_prop_free(vendorp);
559         }
560
561         cmn_err(CE_NOTE, "[vdev_disk_open] %s :: device-solid-state "
562                 "== %d :: vd->vdev_nonrot == %d\n", vd->vdev_path,
563                 device_solid_state, (int) vd->vdev_nonrot);
564
565         /*
566          * Clear the nowritecache bit, so that on a vdev_reopen() we will
567          * try again.
568          */
569         vd->vdev_nowritecache = B_FALSE;
570
571         return (0);
572     }
573 }
574
575 /*
576  * Clear the nowritecache bit, so that on a vdev_reopen() we will
577  * try again.
578  */
579 vd->vdev_nowritecache = B_FALSE;
580
581 return (0);
582 }
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529

```

new/usr/src/uts/common/fs/zfs/vdev_file.c

```
*****
6332 Sun Mar  5 21:45:58 2017
new/usr/src/uts/common/fs/zfs/vdev_file.c
7938 Port ZOL #3712 disable LBA weighting on files and SSDs
*****
_____unchanged_portion_omitted_____
51 static int
52 vdev_file_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,
53     uint64_t *ashift)
54 {
55     vdev_file_t *vf;
56     vnode_t *vp;
57     vattr_t vattr;
58     int error;
59
60     /*
61      * Rotational optimizations only make sense on block devices
62      */
63     vd->vdev_nonrot = B_TRUE;
64
65     /*
66      * We must have a pathname, and it must be absolute.
67      */
68     if (vd->vdev_path == NULL || vd->vdev_path[0] != '/') {
69         vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
70         return (SET_ERROR(EINVAL));
71     }
72
73     /*
74      * Reopen the device if it's not currently open. Otherwise,
75      * just update the physical size of the device.
76      */
77     if (vd->vdev_tsdl != NULL) {
78         ASSERT(vd->vdev_reopening);
79         vf = vd->vdev_tsdl;
80         goto skip_open;
81     }
82
83     vf = vd->vdev_tsdl = kmem_zalloc(sizeof (vdev_file_t), KM_SLEEP);
84
85     /*
86      * We always open the files from the root of the global zone, even if
87      * we're in a local zone. If the user has gotten to this point, the
88      * administrator has already decided that the pool should be available
89      * to local zone users, so the underlying devices should be as well.
90      */
91     ASSERT(vd->vdev_path != NULL && vd->vdev_path[0] == '/');
92     error = vn_openat(vd->vdev_path + 1, UIO_SYSSPACE,
93         spa_mode(vd->vdev_spa) | FOFFMAX, 0, &vp, 0, 0, rootdir, -1);
94
95     if (error) {
96         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
97         return (error);
98     }
99
100    vf->vf_vnode = vp;
101
102 #ifdef _KERNEL
103     /*
104      * Make sure it's a regular file.
105      */
106     if (vp->v_type != VREG) {
107         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
108         return (SET_ERROR(ENODEV));
109     }
110 }
```

1

new/usr/src/uts/common/fs/zfs/vdev_file.c

```
110 #endif
111
112 skip_open:
113     /*
114      * Determine the physical size of the file.
115      */
116     vattr.va_mask = AT_SIZE;
117     error = VOP_GETATTR(vf->vf_vnode, &vattr, 0, kcred, NULL);
118     if (error) {
119         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
120         return (error);
121     }
122
123     *max_psize = *psize = vattr.va_size;
124     *ashift = SPA_MINBLOCKSHIFT;
125
126 }
127
128 _____unchanged_portion_omitted_____
129
130 }
```

2