```
**********************************************************
   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;

1612         ASSERT(MUTEX_HELD(&msp->ms_lock));
1613         ASSERT(!vd->vdev_removing);

1615         /*
1616          * The baseline weight is the metaslab's free space.
1617          */
1618         space = msp->ms_size - space_map_allocated(msp->ms_sm);

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;

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;

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) {
1652         if (metaslab_lba_weighting_enabled) {
1653                 weight = 2 * weight - (msp->ms_id * weight) / vd->vdev_ms_count;
1654                 ASSERT(weight >= space && weight <= 2 * space);
1655         }

1657         /*
```



```
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         }

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 */

 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;

 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;

 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_tmpoffline; /* 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;

 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++) {
1135                 for (int c = 0; c < children; c++)
1139                         vd->vdev_child[c]->vdev_open_error =
1140                             vdev_open(vd->vdev_child[c]);
1141                         vd->vdev_nonrot &= vd->vdev_child[c]->vdev_nonrot;
1142                 }
1143                 return;
1144         }
1145         tq = taskq_create("vdev_open", children, minclsyspri,
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_*

```
**********************************************************
   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
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, 2015 by Delphix. All rights reserved.
 * Copyright 2016 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 Joyent, Inc.  All rights reserved.
 * Copyright (c) 2017 James S Blachly, MD <james.blachly@gmail.com>
 */

#include <sys/zfs_context.h>
#include <sys/spa_impl.h>
#include <sys/refcount.h>
#include <sys/vdev_disk.h>
#include <sys/vdev_impl.h>
#include <sys/fs/zfs.h>
#include <sys/zio.h>
#include <sys/sunldi.h>
#include <sys/efi_partition.h>
#include <sys/fm/fs/zfs.h>

/*
 * Virtual device vector for disks.
 */

extern ldi_ident_t zfs_li;

static void vdev_disk_close(vdev_t *);

typedef struct vdev_disk_ldi_cb {
        list_node_t             lcb_next;
        ldi_callback_id_t       lcb_id;
} vdev_disk_ldi_cb_t;
```

*unchanged_portion_omitted_*

```
/*
 * We want to be loud in DEBUG kernels when DKIOCGMEDIAINFOEXT fails, or when
 * even a fallback to DKIOCGMEDIAINFO fails.
 */
#ifdef DEBUG
#define VDEV_DEBUG(...) cmn_err(CE_NOTE, __VA_ARGS__)
#else
#define VDEV_DEBUG(...) /* Nothing... */
```



```
#endif

static int
vdev_disk_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,
    uint64_t *ashift)
{
        spa_t *spa = vd->vdev_spa;
        vdev_disk_t *dvd = vd->vdev_tsd;
        ldi_ev_cookie_t ecookie;
        vdev_disk_ldi_cb_t *lcb;
        union {
                struct dk_minfo_ext ude;
                struct dk_minfo ud;
        } dks;
        struct dk_minfo_ext *dkmext = &dks.ude;
        struct dk_minfo *dkm = &dks.ud;
        int error;
        dev_t dev;
        int otyp;
        boolean_t validate_devid = B_FALSE;
        ddi_devid_t devid;
        uint64_t capacity = 0, blksz = 0, pbsize;
        int device_solid_state;
        char *vendorp;  /* will point to inquiry-vendor-id */

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

        /*
         * Reopen the device if it's not currently open. Otherwise,
         * just update the physical size of the device.
         */
        if (dvd != NULL) {
                if (dvd->vd_ldi_offline && dvd->vd_lh == NULL) {
                        /*
                         * If we are opening a device in its offline notify
                         * context, the LDI handle was just closed. Clean
                         * up the LDI event callbacks and free vd->vdev_tsd.
                         */
                        vdev_disk_free(vd);
                } else {
                        ASSERT(vd->vdev_reopening);
                        goto skip_open;
                }
        }

        /*
         * Create vd->vdev_tsd.
         */
        vdev_disk_alloc(vd);
        dvd = vd->vdev_tsd;

        /*
         * When opening a disk device, we want to preserve the user's original
         * intent.  We always want to open the device by the path the user gave
         * us, even if it is one of multiple paths to the same device.  But we
         * also want to be able to survive disks being removed/recabled.
         * Therefore the sequence of opening devices is:
         *
         * 1. Try opening the device by path.  For legacy pools without the
         *    '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, "%ss0", 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                 blksz = 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                 blksz = dkm->dki_lbsize;
```



```
517                 pbsize = blksz;
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 * blksz;
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          * 1. Check if device is SSD
551          * 2. If not SSD, check if device is Virtio
552          */
553         device_solid_state = ldi_prop_get_int(dvd->vd_lh, LDI_DEV_T_ANY,
554             "device-solid-state", 0);
555         vd->vdev_nonrot = (device_solid_state ? B_TRUE : B_FALSE);

557         if (device_solid_state == 0 &&
558             ldi_prop_exists(dvd->vd_lh, LDI_DEV_T_ANY, "inquiry-vendor-id")) {
559                 ldi_prop_lookup_string(dvd->vd_lh, LDI_DEV_T_ANY,
560                     "inquiry-vendor-id", &vendorp);
561                 if (strncmp(vendorp, "Virtio", 6) == 0)
562                     vd->vdev_nonrot = B_TRUE;
563                 ddi_prop_free(vendorp);
564         }

566         cmn_err(CE_NOTE, "[vdev_disk_open] %s :: device-solid-state "
567             "== %d :: vd->vdev_nonrot == %d\n", vd->vdev_path,
568             device_solid_state, (int) vd->vdev_nonrot);

570         /*
571          * Clear the nowritecache bit, so that on a vdev_reopen() we will
572          * try again.
573          */
574         vd->vdev_nowritecache = B_FALSE;

576         return (0);
577 }
```

_____*unchanged_portion_omitted_*

****
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;

 60         /*
 61          * Rotational optimizations only make sense on block devices
 62          */
 63         vd->vdev_nonrot = B_TRUE;

 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         }

 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_tsd != NULL) {
 78                 ASSERT(vd->vdev_reopening);
 79                 vf = vd->vdev_tsd;
 80                 goto skip_open;
 81         }

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

 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);

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

100         vf->vf_vnode = vp;

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 #endif

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         }

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

126         return (0);
127 }
```

_____unchanged_portion_omitted_