

```

*****
27864 Mon Apr 7 00:45:25 2008
new/usr/src/uts/common/fs/zfs/metablab.c
expandable RAID-Z
*****
_____unchanged_portion_omitted_____

714 /*
715  * Allocate a block for the specified i/o.
716  */
717 static int
718 metablab_alloc_dva(spa_t *spa, metablab_class_t *mc, uint64_t psize,
719     dva_t *dva, int d, dva_t *hintdva, uint64_t txg, boolean_t hintdva_avoid)
720 {
721     metablab_group_t *mg, *rotor;
722     vdev_t *vd;
723     int dshift = 3;
724     int all_zero;
725     uint64_t offset = -1ULL;
726     uint64_t asize;
727     uint64_t distance;

729     ASSERT(!DVA_IS_VALID(&dva[d]));

731     /*
732      * For testing, make some blocks above a certain size be gang blocks.
733      */
734     if (psize >= metablab_gang_bang && (lbolt & 3) == 0)
735         return (ENOSPC);

737     /*
738      * Start at the rotor and loop through all mgs until we find something.
739      * Note that there's no locking on mc_rotor or mc_allocated because
740      * nothing actually breaks if we miss a few updates -- we just won't
741      * allocate quite as evenly. It all balances out over time.
742      *
743      * If we are doing ditto or log blocks, try to spread them across
744      * consecutive vdevs. If we're forced to reuse a vdev before we've
745      * allocated all of our ditto blocks, then try and spread them out on
746      * that vdev as much as possible. If it turns out to not be possible,
747      * gradually lower our standards until anything becomes acceptable.
748      * Also, allocating on consecutive vdevs (as opposed to random vdevs)
749      * gives us hope of containing our fault domains to something we're
750      * able to reason about. Otherwise, any two top-level vdev failures
751      * will guarantee the loss of data. With consecutive allocation,
752      * only two adjacent top-level vdev failures will result in data loss.
753      *
754      * If we are doing gang blocks (hintdva is non-NULL), try to keep
755      * ourselves on the same vdev as our gang block header. That
756      * way, we can hope for locality in vdev_cache, plus it makes our
757      * fault domains something tractable.
758      */
759     if (hintdva) {
760         vd = vdev_lookup_top(spa, DVA_GET_VDEV(&hintdva[d]));
761         if (hintdva_avoid)
762             mg = vd->vdev_mg->mg_next;
763         else
764             mg = vd->vdev_mg;
765     } else if (d != 0) {
766         vd = vdev_lookup_top(spa, DVA_GET_VDEV(&dva[d - 1]));
767         mg = vd->vdev_mg->mg_next;
768     } else {
769         mg = mc->mc_rotor;
770     }

772     /*
773      * If the hint put us into the wrong class, just follow the rotor.
774      */
775     if (mg->mg_class != mc)
776         mg = mc->mc_rotor;

```

```

778     rotor = mg;
779 top:
780     all_zero = B_TRUE;
781     do {
782         vd = mg->mg_vd;
783         /*
784          * Dont allocate from faulted devices
785          */
786         if (!vdev_writeable(vd))
787             goto next;
788         /*
789          * Avoid writing single-copy data to a failing vdev
790          */
791         if ((vd->vdev_stat.vs_write_errors > 0 ||
792             vd->vdev_state < VDEV_STATE_HEALTHY) &&
793             d == 0 && dshift == 3) {
794             all_zero = B_FALSE;
795             goto next;
796         }

798         ASSERT(mg->mg_class == mc);

800         distance = vd->vdev_asize >> dshift;
801         if (distance <= (1ULL << vd->vdev_ms_shift))
802             distance = 0;
803         else
804             all_zero = B_FALSE;

806         asize = vdev_psize_to_asize(vd, psize);
807         ASSERT(P2PHASE(asize, 1ULL << vd->vdev_ashift) == 0);

809         offset = metablab_group_alloc(mg, asize, txg, distance, dva, d);
810         if (offset != -1ULL) {
811             /*
812              * If we've just selected this metablab group,
813              * figure out whether the corresponding vdev is
814              * over- or under-used relative to the pool,
815              * and set an allocation bias to even it out.
816              */
817             if (mc->mc_allocated == 0) {
818                 vdev_stat_t *vs = &vd->vdev_stat;
819                 uint64_t alloc, space;
820                 int64_t vu, su;

822                 alloc = spa_get_alloc(spa);
823                 space = spa_get_space(spa);

825                 /*
826                  * Determine percent used in units of 0..1024.
827                  * (This is just to avoid floating point.)
828                  */
829                 vu = (vs->vs_alloc << 10) / (vs->vs_space + 1);
830                 su = (alloc << 10) / (space + 1);

832                 /*
833                  * Bias by at most +/- 25% of the aliquot.
834                  */
835                 mg->mg_bias = ((su - vu) *
836                     (int64_t)mg->mg_aliquot) / (1024 * 4);
837             }

839             if (atomic_add_64_nv(&mc->mc_allocated, asize) >=
840                 mg->mg_aliquot + mg->mg_bias) {
841                 mc->mc_rotor = mg->mg_next;
842                 mc->mc_allocated = 0;
843             }

```

```
845             DVA_SET_VDEV(&dva[d], vd->vdev_id);
846             if (vd->vdev_ops->vdev_op_grid != NULL)
847                 DVA_SET_GRID(&dva[d],
848                     vd->vdev_ops->vdev_op_grid(vd);
849             DVA_SET_ASIZE(&dva[d], asize);
850             DVA_SET_GANG(&dva[d], 0);
851             DVA_SET_OFFSET(&dva[d], offset);
847             DVA_SET_GANG(&dva[d], 0);
848             DVA_SET_ASIZE(&dva[d], asize);

853             return (0);
854         }
855 next:
856         mc->mc_rotor = mg->mg_next;
857         mc->mc_allocated = 0;
858     } while ((mg = mg->mg_next) != rotor);

860     if (!all_zero) {
861         dshift++;
862         ASSERT(dshift < 64);
863         goto top;
864     }

866     bzero(&dva[d], sizeof (dva_t));

868     return (ENOSPC);
869 }
unchanged_portion_omitted
```

```

*****
13017 Mon Apr 7 00:45:25 2008
new/usr/src/uts/common/fs/zfs/space_map.c
expandable RAID-Z
*****
_____unchanged_portion_omitted_____

185 int
186 space_map_fold(space_map_t *sm, uint8_t oldwidth, uint8_t newwidth)
187 {
188     space_seg_t *ss;
189     uint64_t d, r;

191     ASSERT(MUTEX_HELD(sm->sm_lock));

193     for (ss = avl_first(&sm->sm_root); ss != NULL;
194          ss = AVL_NEXT(&sm->sm_root, ss)) {
195         d = ss->ss_start / oldwidth;
196         r = ss->ss_start % oldwidth;
197         ss->ss_start = d * newwidth + r;

199         d = ss->ss_end / oldwidth;
200         r = ss->ss_end % oldwidth;
201         ss->ss_end = d * newwidth + r;
202     }
203 }

205 int
206 space_map_contains(space_map_t *sm, uint64_t start, uint64_t size)
207 {
208     avl_index_t where;
209     space_seg_t ssearch, *ss;
210     uint64_t end = start + size;

212     ASSERT(MUTEX_HELD(sm->sm_lock));
213     VERIFY(size != 0);
214     VERIFY(P2PHASE(start, 1ULL << sm->sm_shift) == 0);
215     VERIFY(P2PHASE(size, 1ULL << sm->sm_shift) == 0);

217     ssearch.ss_start = start;
218     ssearch.ss_end = end;
219     ss = avl_find(&sm->sm_root, &ssearch, &where);

221     return (ss != NULL && ss->ss_start <= start && ss->ss_end >= end);
222 }
_____unchanged_portion_omitted_____

312 /*
313  * Note: space_map_load() will drop sm_lock across dmu_read() calls.
314  * The caller must be OK with this.
315  */
316 int
317 space_map_load(space_map_t *sm, space_map_ops_t *ops, uint8_t maptype,
318               space_map_obj_t *smo, objset_t *os)
319 {
320     uint64_t *entry, *entry_map, *entry_map_end;
321     uint64_t bufsize, size, offset, end, space;
322     uint64_t mapstart = sm->sm_start;
323     int error = 0;

325     ASSERT(MUTEX_HELD(sm->sm_lock));

327     space_map_load_wait(sm);

329     if (sm->sm_loaded)
330         return (0);

332     sm->sm_loading = B_TRUE;
333     end = smo->smo_objsize;

```

```

334     space = smo->smo_alloc;

336     ASSERT(sm->sm_ops == NULL);
337     VERIFY3U(sm->sm_space, ==, 0);

339     if (maptype == SM_FREE) {
340         space_map_add(sm, sm->sm_start, sm->sm_size);
341         space = sm->sm_size - space;
342     }

344     bufsize = 1ULL << SPACE_MAP_BLOCKSHIFT;
345     entry_map = zio_buf_alloc(bufsize);

347     mutex_exit(sm->sm_lock);
348     if (end > bufsize)
349         dmu_prefetch(os, smo->smo_object, bufsize, end - bufsize);
350     mutex_enter(sm->sm_lock);

352     for (offset = 0; offset < end; offset += bufsize) {
353         size = MIN(end - offset, bufsize);
354         VERIFY(P2PHASE(size, sizeof(uint64_t)) == 0);
355         VERIFY(size != 0);

357         dprintf("object=%llu offset=%llx size=%llx\n",
358                smo->smo_object, offset, size);

360         mutex_exit(sm->sm_lock);
361         error = dmu_read(os, smo->smo_object, offset, size, entry_map);
362         mutex_enter(sm->sm_lock);
363         if (error != 0)
364             break;

366         entry_map_end = entry_map + (size / sizeof(uint64_t));
367         for (entry = entry_map; entry < entry_map_end; entry++) {
368             uint64_t e = *entry;

370             if (SM_IS_DEBUG(e)) /* Skip debug entries */
371                 if (SM_DEBUG_DECODE(e)) /* Skip debug entries */
372                     continue;

373             if (SM_IS_SPECIAL(e)) {
374                 uint8_t oldwidth, newwidth;
375                 ASSERT(SM_SPECIAL_ACTION_DECODE(e) == SM_FOLD);
376                 oldwidth = BF64_ENCODE(e, 0, 8);
377                 newwidth = BF64_ENCODE(e, 8, 8);
378                 space_map_fold(sm, oldwidth, newwidth);
379                 continue;
380             }

382             (SM_TYPE_DECODE(e) == maptype ?
383              space_map_add : space_map_remove)(sm,
384              (SM_OFFSET_DECODE(e) << sm->sm_shift) + mapstart,
385              SM_RUN_DECODE(e) << sm->sm_shift);
386         }
387     }

389     if (error == 0) {
390         VERIFY3U(sm->sm_space, ==, space);

392         sm->sm_loaded = B_TRUE;
393         sm->sm_ops = ops;
394         if (ops != NULL)
395             ops->smop_load(sm);
396     } else {
397         space_map_vacate(sm, NULL, NULL);
398     }

```

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

3

```
400     zio_buf_free(entry_map, bufsize);
402     sm->sm_loading = B_FALSE;
404     cv_broadcast(&sm->sm_load_cv);
406     return (error);
407 }
```

_____unchanged_portion_omitted_____

```

*****
6514 Mon Apr 7 00:45:26 2008
new/usr/src/uts/common/fs/zfs/sys/space_map.h
expandable RAID-Z
*****
unchanged_portion_omitted_

75 /*
76 * allocation/free entry
76 * debug entry
77 *
78 * 1 47 1 15
79 *
80 * [ 0 | offset (sm_shift units) | type | run ]
81 *
82 * 63 62 17 16 15 0
78 * 1 3 10 50
79 *
80 * [ 1 | action | syncpass | txg (lower bits) ]
81 *
82 * 63 62 60 59 50 49 0
83 *
84 * special entry
85 *
86 * 1 1 6 56
87 *
88 * [ 1 | 1 | action | <action specific> ]
89 *
90 * 63 62 61 56 55 0
91 *
92 * debug entry
86 * non-debug entry
93 *
94 * 1 1 2 10 50
95 *
96 * [ 1 | 0 | action | syncpass | txg (lower bits) ]
97 *
98 * 63 62 61 60 59 50 49 0
88 * 1 47 1 15
89 *
90 * [ 0 | offset (sm_shift units) | type | run ]
91 *
92 * 63 62 17 16 15 0
93 */

101 /* All this stuff takes and returns bytes */
102 #define SM_RUN_DECODE(x) (BF64_DECODE(x, 0, 15) + 1)
103 #define SM_RUN_ENCODE(x) BF64_ENCODE(x, -1, 0, 15)
104 #define SM_TYPE_DECODE(x) BF64_DECODE(x, 15, 1)
105 #define SM_TYPE_ENCODE(x) BF64_ENCODE(x, 15, 1)
106 #define SM_OFFSET_DECODE(x) BF64_DECODE(x, 16, 47)
107 #define SM_OFFSET_ENCODE(x) BF64_ENCODE(x, 16, 47)
108 #define SM_DEBUG_DECODE(x) BF64_DECODE(x, 63, 1)
109 #define SM_DEBUG_ENCODE(x) BF64_ENCODE(x, 63, 1)

110 #define SM_IS_SPECIAL(x) (BF64_DECODE(x, 62, 2) == 3)
111 #define SM_SPECIAL_ENCODE() BF64_ENCODE(3, 62, 2)
112 #define SM_IS_DEBUG(x) (BF64_DECODE(x, 62, 2) == 2)
113 #define SM_DEBUG_ENCODE() BF64_ENCODE(2, 62, 2)

114 #define SM_SPECIAL_ACTION_DECODE(x) BF64_DECODE(x, 56, 6)
115 #define SM_SPECIAL_ACTION_ENCODE(x) BF64_ENCODE(x, 56, 6)

117 #define SM_DEBUG_ACTION_DECODE(x) BF64_DECODE(x, 60, 3)
118 #define SM_DEBUG_ACTION_ENCODE(x) BF64_ENCODE(x, 60, 3)

119 #define SM_DEBUG_SYNCPASS_DECODE(x) BF64_DECODE(x, 50, 10)
120 #define SM_DEBUG_SYNCPASS_ENCODE(x) BF64_ENCODE(x, 50, 10)

121 #define SM_DEBUG_TXG_DECODE(x) BF64_DECODE(x, 0, 50)
122 #define SM_DEBUG_TXG_ENCODE(x) BF64_ENCODE(x, 0, 50)

```

```

124 #define SM_RUN_MAX SM_RUN_DECODE(~0ULL)

126 #define SM_ALLOC 0x0
127 #define SM_FREE 0x1

129 #define SM_FOLD 0x01

131 /*
132 * The data for a given space map can be kept on blocks of any size.
133 * Larger blocks entail fewer i/o operations, but they also cause the
134 * DMU to keep more data in-core, and also to waste more i/o bandwidth
135 * when only a few blocks have changed since the last transaction group.
136 * This could use a lot more research, but for now, set the freelist
137 * block size to 4k (2^12).
138 */
139 #define SPACE_MAP_BLOCKSHIFT 12

141 typedef void space_map_func_t(space_map_t *sm, uint64_t start, uint64_t size);

143 extern void space_map_create(space_map_t *sm, uint64_t start, uint64_t size,
144 uint8_t shift, kmutex_t *lp);
145 extern void space_map_destroy(space_map_t *sm);
146 extern void space_map_add(space_map_t *sm, uint64_t start, uint64_t size);
147 extern void space_map_remove(space_map_t *sm, uint64_t start, uint64_t size);
148 extern int space_map_contains(space_map_t *sm, uint64_t start, uint64_t size);
149 extern void space_map_vacate(space_map_t *sm,
150 space_map_func_t *func, space_map_t *mdest);
151 extern void space_map_walk(space_map_t *sm,
152 space_map_func_t *func, space_map_t *mdest);
153 extern void space_map_excise(space_map_t *sm, uint64_t start, uint64_t size);
154 extern void space_map_union(space_map_t *smd, space_map_t *sms);

156 extern void space_map_load_wait(space_map_t *sm);
157 extern int space_map_load(space_map_t *sm, space_map_ops_t *ops,
158 uint8_t matype, space_map_obj_t *smo, objset_t *os);
159 extern void space_map_unload(space_map_t *sm);

161 extern uint64_t space_map_alloc(space_map_t *sm, uint64_t size);
162 extern void space_map_claim(space_map_t *sm, uint64_t start, uint64_t size);
163 extern void space_map_free(space_map_t *sm, uint64_t start, uint64_t size);

165 extern void space_map_sync(space_map_t *sm, uint8_t matype,
166 space_map_obj_t *smo, objset_t *os, dmu_tx_t *tx);
167 extern void space_map_truncate(space_map_obj_t *smo,
168 objset_t *os, dmu_tx_t *tx);

170 #ifdef __cplusplus
171 }
unchanged_portion_omitted_

```

new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h

1

```
*****
 9987 Mon Apr  7 00:45:26 2008
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
expandable_RAID-Z
*****
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 2007 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #ifndef _SYS_VDEV_IMPL_H
27 #define _SYS_VDEV_IMPL_H

29 #pragma ident      "%Z%M% %I%      %E% SMI"

31 #include <sys/avl.h>
32 #include <sys/dmu.h>
33 #include <sys/metaslab.h>
34 #include <sys/nvpair.h>
35 #include <sys/space_map.h>
36 #include <sys/vdev.h>
37 #include <sys/dkio.h>
38 #include <sys/uberblock_impl.h>

40 #ifdef __cplusplus
41 extern "C" {
42 #endif

44 /*
45  * Virtual device descriptors.
46  *
47  * All storage pool operations go through the virtual device framework,
48  * which provides data replication and I/O scheduling.
49  */

51 /*
52  * Forward declarations that lots of things need.
53  */
54 typedef struct vdev_queue vdev_queue_t;
55 typedef struct vdev_cache vdev_cache_t;
56 typedef struct vdev_cache_entry vdev_cache_entry_t;

58 /*
59  * Virtual device operations
60  */
61 typedef int      vdev_open_func_t(vdev_t *vd, uint64_t *size, uint64_t *ashift);
```

new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h

2

```
62 typedef void      vdev_close_func_t(vdev_t *vd);
63 typedef int       vdev_probe_func_t(vdev_t *vd);
64 typedef uint64_t  vdev_asize_func_t(vdev_t *vd, uint64_t psize);
65 typedef int       vdev_io_start_func_t(zio_t *zio);
66 typedef int       vdev_io_done_func_t(zio_t *zio);
67 typedef void      vdev_state_change_func_t(vdev_t *vd, int failed, int degraded);
68 typedef uint8_t   vdev_grid_func_t(vdev_t *vd);
69 typedef void      vdev_state_change_func_t(vdev_t *vd, int, int);

70 typedef struct vdev_ops {
71     vdev_open_func_t      *vdev_op_open;
72     vdev_close_func_t     *vdev_op_close;
73     vdev_probe_func_t     *vdev_op_probe;
74     vdev_asize_func_t     *vdev_op_asize;
75     vdev_io_start_func_t  *vdev_op_io_start;
76     vdev_io_done_func_t   *vdev_op_io_done;
77     vdev_state_change_func_t *vdev_op_state_change;
78     vdev_grid_func_t      *vdev_op_grid;
79     char                   vdev_op_type[16];
80     boolean_t              vdev_op_leaf;
81 } vdev_ops_t;
_____ unchanged_portion_omitted_
```

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

1

13902 Mon Apr 7 00:45:26 2008

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

expandable RAID-Z

unchanged portion omitted

```
550 vdev_ops_t vdev_disk_ops = {
551     vdev_disk_open,
552     vdev_disk_close,
553     vdev_disk_probe,
554     vdev_default_asize,
555     vdev_disk_io_start,
556     vdev_disk_io_done,
557     NULL,
558     NULL,
559     VDEV_TYPE_DISK,      /* name of this vdev type */
560     B_TRUE               /* leaf vdev */
561 };
```

unchanged portion omitted

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

1

7713 Mon Apr 7 00:45:27 2008

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

expandable RAID-Z

unchanged portion omitted

```
311 vdev_ops_t vdev_file_ops = {
312     vdev_file_open,
313     vdev_file_close,
314     vdev_file_probe,
315     vdev_default_asize,
316     vdev_file_io_start,
317     vdev_file_io_done,
318     NULL,
319     NULL,
320     VDEV_TYPE_FILE,      /* name of this vdev type */
321     B_TRUE               /* leaf vdev */
322 };
```

unchanged portion omitted

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

1

12339 Mon Apr 7 00:45:27 2008

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

expandable RAID-Z

unchanged portion omitted

```
462 vdev_ops_t vdev_mirror_ops = {
463     vdev_mirror_open,
464     vdev_mirror_close,
465     NULL,
466     vdev_default_asize,
467     vdev_mirror_io_start,
468     vdev_mirror_io_done,
469     vdev_mirror_state_change,
470     NULL,
471     VDEV_TYPE_MIRROR,      /* name of this vdev type */
472     B_FALSE               /* not a leaf vdev */
473 };
```

```
475 vdev_ops_t vdev_replacing_ops = {
476     vdev_mirror_open,
477     vdev_mirror_close,
478     NULL,
479     vdev_default_asize,
480     vdev_mirror_io_start,
481     vdev_mirror_io_done,
482     vdev_mirror_state_change,
483     NULL,
484     VDEV_TYPE_REPLACING,  /* name of this vdev type */
485     B_FALSE               /* not a leaf vdev */
486 };
```

```
488 vdev_ops_t vdev_spare_ops = {
489     vdev_mirror_open,
490     vdev_mirror_close,
491     NULL,
492     vdev_default_asize,
493     vdev_mirror_io_start,
494     vdev_mirror_io_done,
495     vdev_mirror_state_change,
496     NULL,
497     VDEV_TYPE_SPARE,      /* name of this vdev type */
498     B_FALSE               /* not a leaf vdev */
499 };
```

unchanged portion omitted

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

1

2583 Mon Apr 7 00:45:28 2008

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

expandable RAID-Z

unchanged_portion_omitted

```
86 vdev_ops_t vdev_missing_ops = {
87     vdev_missing_open,
88     vdev_missing_close,
89     vdev_missing_probe,
90     vdev_default_asize,
91     vdev_missing_io_start,
92     vdev_missing_io_done,
93     NULL,
94     NULL,
95     VDEV_TYPE_MISSING, /* name of this vdev type */
96     B_TRUE /* leaf vdev */
97 };
```

unchanged_portion_omitted

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

1

34798 Mon Apr 7 00:45:28 2008

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

expandable RAID-Z

_____unchanged_portion_omitted_____

```
1229 static uint8_t
1230 vdev_raidz_grid(vdev_t *vd)
1231 {
1232     ASSERT(vd->vdev_nparity - 1 <= 1);
1233     return (((vd->vdev_nparity - 1) << 6) | vd->vdev_children);
1234 }
```

```
1236 vdev_ops_t vdev_raidz_ops = {
1237     vdev_raidz_open,
1238     vdev_raidz_close,
1239     NULL,
1240     vdev_raidz_asize,
1241     vdev_raidz_io_start,
1242     vdev_raidz_io_done,
1243     vdev_raidz_state_change,
1244     vdev_raidz_grid,
1245     VDEV_TYPE_RAIDZ, /* name of this vdev type */
1246     B_FALSE /* not a leaf vdev */
1247 };
_____unchanged_portion_omitted_____
```

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

1

3425 Mon Apr 7 00:45:28 2008

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

expandable RAID-Z

unchanged portion omitted

```
120 vdev_ops_t vdev_root_ops = {
121     vdev_root_open,
122     vdev_root_close,
123     NULL,
124     vdev_default_asize,
125     NULL,                /* io_start - not applicable to the root */
126     NULL,                /* io_done - not applicable to the root */
127     vdev_root_state_change,
128     NULL,
129     VDEV_TYPE_ROOT,     /* name of this vdev type */
130     B_FALSE             /* not a leaf vdev */
131 };
```

unchanged portion omitted