```
**********************************************************
    6014 Thu Nov 21 00:19:07 2013
new/usr/src/uts/common/fs/zfs/sys/vdev.h
4334 Improve ZFS N-way mirror read performance
**********************************************************
   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  */

  22 /*
  23  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright (c) 2012 by Delphix. All rights reserved.
  25  * Copyright (c) 2013 Steven Hartland. All rights reserved.
  26 #endif /* ! codereview */
  27  */

  29 #ifndef _SYS_VDEV_H
  30 #define _SYS_VDEV_H

  32 #include <sys/spa.h>
  33 #include <sys/zio.h>
  34 #include <sys/dmu.h>
  35 #include <sys/space_map.h>
  36 #include <sys/fs/zfs.h>

  38 #ifdef  __cplusplus
  39 extern "C" {
  40 #endif

  42 typedef enum vdev_dtl_type {
  43         DTL_MISSING,    /* 0% replication: no copies of the data */
  44         DTL_PARTIAL,    /* less than 100% replication: some copies missing */
  45         DTL_SCRUB,      /* unable to fully repair during scrub/resilver */
  46         DTL_OUTAGE,     /* temporarily missing (used to attempt detach) */
  47         DTL_TYPES
  48 } vdev_dtl_type_t;

  50 extern boolean_t zfs_nocacheflush;

  52 extern int vdev_open(vdev_t *);
  53 extern void vdev_open_children(vdev_t *);
  54 extern boolean_t vdev_uses_zvols(vdev_t *);
  55 extern int vdev_validate(vdev_t *, boolean_t);
  56 extern void vdev_close(vdev_t *);
  57 extern int vdev_create(vdev_t *, uint64_t txg, boolean_t isreplace);
  58 extern void vdev_reopen(vdev_t *);
  59 extern int vdev_validate_aux(vdev_t *vd);
  60 extern zio_t *vdev_probe(vdev_t *vd, zio_t *pio);
```

```
  62 extern boolean_t vdev_is_bootable(vdev_t *vd);
  63 extern vdev_t *vdev_lookup_top(spa_t *spa, uint64_t vdev);
  64 extern vdev_t *vdev_lookup_by_guid(vdev_t *vd, uint64_t guid);
  65 extern void vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t d,
  66     uint64_t txg, uint64_t size);
  67 extern boolean_t vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t d,
  68     uint64_t txg, uint64_t size);
  69 extern boolean_t vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t d);
  70 extern void vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg,
  71     int scrub_done);
  72 extern boolean_t vdev_dtl_required(vdev_t *vd);
  73 extern boolean_t vdev_resilver_needed(vdev_t *vd,
  74     uint64_t *minp, uint64_t *maxp);

  76 extern void vdev_hold(vdev_t *);
  77 extern void vdev_rele(vdev_t *);

  79 extern int vdev_metaslab_init(vdev_t *vd, uint64_t txg);
  80 extern void vdev_metaslab_fini(vdev_t *vd);
  81 extern void vdev_metaslab_set_size(vdev_t *);
  82 extern void vdev_expand(vdev_t *vd, uint64_t txg);
  83 extern void vdev_split(vdev_t *vd);
  84 extern void vdev_deadman(vdev_t *vd);

  87 extern void vdev_get_stats(vdev_t *vd, vdev_stat_t *vs);
  88 extern void vdev_clear_stats(vdev_t *vd);
  89 extern void vdev_stat_update(zio_t *zio, uint64_t psize);
  90 extern void vdev_scan_stat_init(vdev_t *vd);
  91 extern void vdev_propagate_state(vdev_t *vd);
  92 extern void vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state,
  93     vdev_aux_t aux);

  95 extern void vdev_space_update(vdev_t *vd,
  96     int64_t alloc_delta, int64_t defer_delta, int64_t space_delta);

  98 extern uint64_t vdev_psize_to_asize(vdev_t *vd, uint64_t psize);

 100 extern int vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux);
 101 extern int vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux);
 102 extern int vdev_online(spa_t *spa, uint64_t guid, uint64_t flags,
 103     vdev_state_t *);
 104 extern int vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags);
 105 extern void vdev_clear(spa_t *spa, vdev_t *vd);

 107 extern boolean_t vdev_is_dead(vdev_t *vd);
 108 extern boolean_t vdev_readable(vdev_t *vd);
 109 extern boolean_t vdev_writeable(vdev_t *vd);
 110 extern boolean_t vdev_allocatable(vdev_t *vd);
 111 extern boolean_t vdev_accessible(vdev_t *vd, zio_t *zio);

 113 extern void vdev_cache_init(vdev_t *vd);
 114 extern void vdev_cache_fini(vdev_t *vd);
 115 extern int vdev_cache_read(zio_t *zio);
 116 extern void vdev_cache_write(zio_t *zio);
 117 extern void vdev_cache_purge(vdev_t *vd);

 119 extern void vdev_queue_init(vdev_t *vd);
 120 extern void vdev_queue_fini(vdev_t *vd);
 121 extern zio_t *vdev_queue_io(zio_t *zio);
 122 extern void vdev_queue_io_done(zio_t *zio);
 123 extern int vdev_queue_length(vdev_t *vd);
 124 extern uint64_t vdev_queue_last_queued_offset(vdev_t *vd);
 125 extern void vdev_queue_register_last_queued_offset(vdev_t *vd, zio_t *zio);
 126 #endif /* ! codereview */
```

```
128  extern void vdev_config_dirty(vdev_t *vd);
129  extern void vdev_config_clean(vdev_t *vd);
130  extern int vdev_config_sync(vdev_t **svd, int svdcount, uint64_t txg,
131      boolean_t);

133  extern void vdev_state_dirty(vdev_t *vd);
134  extern void vdev_state_clean(vdev_t *vd);

136  typedef enum vdev_config_flag {
137          VDEV_CONFIG_SPARE = 1 << 0,
138          VDEV_CONFIG_L2CACHE = 1 << 1,
139          VDEV_CONFIG_REMOVING = 1 << 2
140  } vdev_config_flag_t;

142  extern void vdev_top_config_generate(spa_t *spa, nvlist_t *config);
143  extern nvlist_t *vdev_config_generate(spa_t *spa, vdev_t *vd,
144      boolean_t getstats, vdev_config_flag_t flags);

146  /*
147   * Label routines
148   */
149  struct uberblock;
150  extern uint64_t vdev_label_offset(uint64_t psize, int l, uint64_t offset);
151  extern int vdev_label_number(uint64_t psise, uint64_t offset);
152  extern nvlist_t *vdev_label_read_config(vdev_t *vd, uint64_t txg);
153  extern void vdev_uberblock_load(vdev_t *, struct uberblock *, nvlist_t **);

155  typedef enum {
156          VDEV_LABEL_CREATE,      /* create/add a new device */
157          VDEV_LABEL_REPLACE,     /* replace an existing device */
158          VDEV_LABEL_SPARE,       /* add a new hot spare */
159          VDEV_LABEL_REMOVE,      /* remove an existing device */
160          VDEV_LABEL_L2CACHE,     /* add an L2ARC cache device */
161          VDEV_LABEL_SPLIT        /* generating new label for split-off dev */
162  } vdev_labeltype_t;

164  extern int vdev_label_init(vdev_t *vd, uint64_t txg, vdev_labeltype_t reason);

166  #ifdef  __cplusplus
167  }
168  #endif

170  #endif  /* _SYS_VDEV_H */
```

```
**********************************************************
   12026 Thu Nov 21 00:19:07 2013
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
4334 Improve ZFS N-way mirror read performance
**********************************************************
   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) 2013 by Delphix. All rights reserved.
  24  * Copyright (c) 2013 Steven Hartland. All rights reserved.
  25 #endif /* ! codereview */
  26  */

  28 #ifndef _SYS_VDEV_IMPL_H
  29 #define _SYS_VDEV_IMPL_H

  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 *max_size,
```

```
  62     uint64_t *ashift);
  63 typedef void    vdev_close_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 void    vdev_io_done_func_t(zio_t *zio);
  67 typedef void    vdev_state_change_func_t(vdev_t *vd, int, int);
  68 typedef void    vdev_hold_func_t(vdev_t *vd);
  69 typedef void    vdev_rele_func_t(vdev_t *vd);

  71 typedef struct vdev_ops {
  72         vdev_open_func_t                *vdev_op_open;
  73         vdev_close_func_t               *vdev_op_close;
  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_hold_func_t                *vdev_op_hold;
  79         vdev_rele_func_t                *vdev_op_rele;
  80         char                            vdev_op_type[16];
  81         boolean_t                       vdev_op_leaf;
  82 } vdev_ops_t;

  84 /*
  85  * Virtual device properties
  86  */
  87 struct vdev_cache_entry {
  88         char            *ve_data;
  89         uint64_t        ve_offset;
  90         uint64_t        ve_lastused;
  91         avl_node_t      ve_offset_node;
  92         avl_node_t      ve_lastused_node;
  93         uint32_t        ve_hits;
  94         uint16_t        ve_missed_update;
  95         zio_t           *ve_fill_io;
  96 };

  98 struct vdev_cache {
  99         avl_tree_t      vc_offset_tree;
 100         avl_tree_t      vc_lastused_tree;
 101         kmutex_t        vc_lock;
 102 };

 104 typedef struct vdev_queue_class {
 105         uint32_t        vqc_active;

 107         /*
 108          * Sorted by offset or timestamp, depending on if the queue is
 109          * LBA-ordered vs FIFO.
 110          */
 111         avl_tree_t      vqc_queued_tree;
 112 } vdev_queue_class_t;

 114 struct vdev_queue {
 115         vdev_t          *vq_vdev;
 116         vdev_queue_class_t vq_class[ZIO_PRIORITY_NUM_QUEUEABLE];
 117         avl_tree_t      vq_active_tree;
 118         uint64_t        vq_last_offset;
 119         hrtime_t        vq_io_complete_ts; /* time last i/o completed */
 120         kmutex_t        vq_lock;
 121         uint64_t        vq_last_queued_offset;
 122 #endif /* ! codereview */
 123 };

 125 /*
 126  * Virtual device descriptor
 127  */
```

```
128 struct vdev {
129         /*
130          * Common to all vdev types.
131          */
132         uint64_t        vdev_id;        /* child number in vdev parent  */
133         uint64_t        vdev_guid;      /* unique ID for this vdev      */
134         uint64_t        vdev_guid_sum;  /* self guid + all child guids  */
135         uint64_t        vdev_orig_guid; /* orig. guid prior to remove   */
136         uint64_t        vdev_asize;     /* allocatable device capacity  */
137         uint64_t        vdev_min_asize; /* min acceptable asize         */
138         uint64_t        vdev_max_asize; /* max acceptable asize         */
139         uint64_t        vdev_ashift;    /* block alignment shift        */
140         uint64_t        vdev_state;     /* see VDEV_STATE_* #defines    */
141         uint64_t        vdev_prevstate; /* used when reopening a vdev   */
142         vdev_ops_t      *vdev_ops;      /* vdev operations              */
143         spa_t           *vdev_spa;      /* spa for this vdev            */
144         void            *vdev_tsd;      /* type-specific data           */
145         vnode_t         *vdev_name_vp;  /* vnode for pathname           */
146         vnode_t         *vdev_devid_vp; /* vnode for devid              */
147         vdev_t          *vdev_top;      /* top-level vdev               */
148         vdev_t          *vdev_parent;   /* parent vdev                  */
149         vdev_t          **vdev_child;   /* array of children            */
150         uint64_t        vdev_children;  /* number of children           */
151         vdev_stat_t     vdev_stat;      /* virtual device statistics    */
152         boolean_t       vdev_expanding; /* expand the vdev?             */
153         boolean_t       vdev_reopening; /* reopen in progress?          */
154         int             vdev_open_error; /* error on last open          */
155         kthread_t       *vdev_open_thread; /* thread opening children   */
156         uint64_t        vdev_crtxg;     /* txg when top-level was added */
157
158         /*
159          * Top-level vdev state.
160          */
161         uint64_t        vdev_ms_array;  /* metaslab array object        */
162         uint64_t        vdev_ms_shift;  /* metaslab size shift          */
163         uint64_t        vdev_ms_count;  /* number of metaslabs          */
164         metaslab_group_t *vdev_mg;      /* metaslab group               */
165         metaslab_t      **vdev_ms;      /* metaslab array               */
166         txg_list_t      vdev_ms_list;   /* per-txg dirty metaslab lists */
167         txg_list_t      vdev_dtl_list;  /* per-txg dirty DTL lists      */
168         txg_node_t      vdev_txg_node;  /* per-txg dirty vdev linkage   */
169         boolean_t       vdev_remove_wanted; /* async remove wanted?     */
170         boolean_t       vdev_probe_wanted;  /* async probe wanted?      */
171         list_node_t     vdev_config_dirty_node; /* config dirty list    */
172         list_node_t     vdev_state_dirty_node; /* state dirty list      */
173         uint64_t        vdev_deflate_ratio; /* deflation ratio (x512)   */
174         uint64_t        vdev_islog;     /* is an intent log device      */
175         uint64_t        vdev_removing;  /* device is being removed?     */
176         boolean_t       vdev_ishole;    /* is a hole in the namespace   */
177
178         /*
179          * Leaf vdev state.
180          */
181         range_tree_t    *vdev_dtl[DTL_TYPES]; /* dirty time logs        */
182         space_map_t     *vdev_dtl_sm;   /* dirty time log space map     */
183         txg_node_t      vdev_dtl_node;  /* per-txg dirty DTL linkage    */
184         uint64_t        vdev_dtl_object; /* DTL object                  */
185         uint64_t        vdev_psize;     /* physical device capacity     */
186         uint64_t        vdev_wholedisk; /* true if this is a whole disk */
187         uint64_t        vdev_offline;   /* persistent offline state     */
188         uint64_t        vdev_faulted;   /* persistent faulted state     */
189         uint64_t        vdev_degraded;  /* persistent degraded state    */
190         uint64_t        vdev_removed;   /* persistent removed state     */
191         uint64_t        vdev_resilver_txg; /* persistent resilvering state */
192         uint64_t        vdev_nparity;   /* number of parity devices for raidz */
193         char            *vdev_path;     /* vdev path (if any)           */
```

```
194         char            *vdev_devid;    /* vdev devid (if any)          */
195         char            *vdev_physpath; /* vdev device path (if any)    */
196         char            *vdev_fru;      /* physical FRU location        */
197         uint64_t        vdev_not_present; /* not present during import  */
198         uint64_t        vdev_unspare;   /* unspare when resilvering done */
199         boolean_t       vdev_nowritecache; /* true if flushwritecache failed */
200         boolean_t       vdev_checkremove; /* temporary online test      */
201         boolean_t       vdev_forcefault; /* force online fault          */
202         boolean_t       vdev_splitting; /* split or repair in progress  */
203         boolean_t       vdev_delayed_close; /* delayed device close?    */
204         boolean_t       vdev_tmpoffline; /* device taken offline temporarily? */
205         boolean_t       vdev_detached;  /* device detached?             */
206         boolean_t       vdev_cant_read; /* vdev is failing all reads    */
207         boolean_t       vdev_cant_write; /* vdev is failing all writes  */
208         boolean_t       vdev_isspare;   /* was a hot spare              */
209         boolean_t       vdev_isl2cache; /* was a l2cache device         */
210         vdev_queue_t    vdev_queue;     /* I/O deadline schedule queue  */
211         vdev_cache_t    vdev_cache;     /* physical block cache         */
212         spa_aux_vdev_t  *vdev_aux;      /* for l2cache vdevs            */
213         zio_t           *vdev_probe_zio; /* root of current probe       */
214         vdev_aux_t      vdev_label_aux; /* on-disk aux state            */
215         uint16_t        vdev_rotation_rate; /* rotational rate of the media */
216 #define VDEV_RATE_UNKNOWN       0
217 #define VDEV_RATE_NON_ROTATING  1
218 #endif /* ! codereview */
219
220         /*
221          * For DTrace to work in userland (libzpool) context, these fields must
222          * remain at the end of the structure.  DTrace will use the kernel's
223          * CTF definition for 'struct vdev', and since the size of a kmutex_t is
224          * larger in userland, the offsets for the rest of the fields would be
225          * incorrect.
226          */
227         kmutex_t        vdev_dtl_lock;  /* vdev_dtl_{map,resilver}      */
228         kmutex_t        vdev_stat_lock; /* vdev_stat                   */
229         kmutex_t        vdev_probe_lock; /* protects vdev_probe_zio     */
230 };
231
232 #define VDEV_RAIDZ_MAXPARITY    3
233
234 #define VDEV_PAD_SIZE           (8 << 10)
235 /* 2 padding areas (vl_pad1 and vl_pad2) to skip */
236 #define VDEV_SKIP_SIZE          VDEV_PAD_SIZE * 2
237 #define VDEV_PHYS_SIZE          (112 << 10)
238 #define VDEV_UBERBLOCK_RING     (128 << 10)
239
240 #define VDEV_UBERBLOCK_SHIFT(vd)        \
241         MAX((vd)->vdev_top->vdev_ashift, UBERBLOCK_SHIFT)
242 #define VDEV_UBERBLOCK_COUNT(vd)        \
243         (VDEV_UBERBLOCK_RING >> VDEV_UBERBLOCK_SHIFT(vd))
244 #define VDEV_UBERBLOCK_OFFSET(vd, n)    \
245         offsetof(vdev_label_t, vl_uberblock[(n) << VDEV_UBERBLOCK_SHIFT(vd)])
246 #define VDEV_UBERBLOCK_SIZE(vd)         (1ULL << VDEV_UBERBLOCK_SHIFT(vd))
247
248 typedef struct vdev_phys {
249         char            vp_nvlist[VDEV_PHYS_SIZE - sizeof (zio_eck_t)];
250         zio_eck_t       vp_zbt;
251 } vdev_phys_t;
252
253 typedef struct vdev_label {
254         char            vl_pad1[VDEV_PAD_SIZE];                  /*  8K */
255         char            vl_pad2[VDEV_PAD_SIZE];                  /*  8K */
256         vdev_phys_t     vl_vdev_phys;                           /* 112K */
257         char            vl_uberblock[VDEV_UBERBLOCK_RING];      /* 128K */
258 } vdev_label_t;                                                 /* 256K total */
```

```
260 /*
261  * vdev_dirty() flags
262  */
263 #define VDD_METASLAB    0x01
264 #define VDD_DTL         0x02

266 /* Offset of embedded boot loader region on each label */
267 #define VDEV_BOOT_OFFSET        (2 * sizeof (vdev_label_t))
268 /*
269  * Size of embedded boot loader region on each label.
270  * The total size of the first two labels plus the boot area is 4MB.
271  */
272 #define VDEV_BOOT_SIZE          (7ULL << 19)                            /* 3.5M */

274 /*
275  * Size of label regions at the start and end of each leaf device.
276  */
277 #define VDEV_LABEL_START_SIZE   (2 * sizeof (vdev_label_t) + VDEV_BOOT_SIZE)
278 #define VDEV_LABEL_END_SIZE     (2 * sizeof (vdev_label_t))
279 #define VDEV_LABELS             4
280 #define VDEV_BEST_LABEL         VDEV_LABELS

282 #define VDEV_ALLOC_LOAD         0
283 #define VDEV_ALLOC_ADD          1
284 #define VDEV_ALLOC_SPARE        2
285 #define VDEV_ALLOC_L2CACHE      3
286 #define VDEV_ALLOC_ROOTPOOL     4
287 #define VDEV_ALLOC_SPLIT        5
288 #define VDEV_ALLOC_ATTACH       6

290 /*
291  * Allocate or free a vdev
292  */
293 extern vdev_t *vdev_alloc_common(spa_t *spa, uint_t id, uint64_t guid,
294     vdev_ops_t *ops);
295 extern int vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *config,
296     vdev_t *parent, uint_t id, int alloctype);
297 extern void vdev_free(vdev_t *vd);

299 /*
300  * Add or remove children and parents
301  */
302 extern void vdev_add_child(vdev_t *pvd, vdev_t *cvd);
303 extern void vdev_remove_child(vdev_t *pvd, vdev_t *cvd);
304 extern void vdev_compact_children(vdev_t *pvd);
305 extern vdev_t *vdev_add_parent(vdev_t *cvd, vdev_ops_t *ops);
306 extern void vdev_remove_parent(vdev_t *cvd);

308 /*
309  * vdev sync load and sync
310  */
311 extern void vdev_load_log_state(vdev_t *nvd, vdev_t *ovd);
312 extern boolean_t vdev_log_state_valid(vdev_t *vd);
313 extern void vdev_load(vdev_t *vd);
314 extern int vdev_dtl_load(vdev_t *vd);
315 extern void vdev_sync(vdev_t *vd, uint64_t txg);
316 extern void vdev_sync_done(vdev_t *vd, uint64_t txg);
317 extern void vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg);
318 extern void vdev_dirty_leaves(vdev_t *vd, int flags, uint64_t txg);

320 /*
321  * Available vdev types.
322  */
323 extern vdev_ops_t vdev_root_ops;
324 extern vdev_ops_t vdev_mirror_ops;
325 extern vdev_ops_t vdev_replacing_ops;
```

```
326 extern vdev_ops_t vdev_raidz_ops;
327 extern vdev_ops_t vdev_disk_ops;
328 extern vdev_ops_t vdev_file_ops;
329 extern vdev_ops_t vdev_missing_ops;
330 extern vdev_ops_t vdev_hole_ops;
331 extern vdev_ops_t vdev_spare_ops;

333 /*
334  * Common size functions
335  */
336 extern uint64_t vdev_default_asize(vdev_t *vd, uint64_t psize);
337 extern uint64_t vdev_get_min_asize(vdev_t *vd);
338 extern void vdev_set_min_asize(vdev_t *vd);

340 /*
341  * Global variables
342  */
343 /* zdb uses this tunable, so it must be declared here to make lint happy. */
344 extern int zfs_vdev_cache_size;

346 /*
347  * The vdev_buf_t is used to translate between zio_t and buf_t, and back again.
348  */
349 typedef struct vdev_buf {
350         buf_t   vb_buf;                 /* buffer that describes the io */
351         zio_t   *vb_io;                 /* pointer back to the original zio_t */
352 } vdev_buf_t;

354 #ifdef  __cplusplus
355 }
356 #endif

358 #endif  /* _SYS_VDEV_IMPL_H */
```

```
*********************************************************
    24357 Thu Nov 21 00:19:08 2013
new/usr/src/uts/common/fs/zfs/vdev_disk.c
4334 Improve ZFS N-way mirror read performance
*********************************************************
_____unchanged_portion_omitted_

274 /*
275  * We want to be loud in DEBUG kernels when DKIOCGMEDIAINFOEXT fails, or when
276  * even a fallback to DKIOCGMEDIAINFO fails.
277  */
278 #ifdef DEBUG
279 #define VDEV_DEBUG(...) cmn_err(CE_NOTE, __VA_ARGS__)
280 #else
281 #define VDEV_DEBUG(...) /* Nothing... */
282 #endif

284 static int
285 vdev_disk_open(vdev_t *vd, uint64_t *psize, uint64_t *max_psize,
286     uint64_t *ashift)
287 {
288         spa_t *spa = vd->vdev_spa;
289         vdev_disk_t *dvd = vd->vdev_tsd;
290         ldi_ev_cookie_t ecookie;
291         vdev_disk_ldi_cb_t *lcb;
292         union {
293                 struct dk_minfo_ext ude;
294                 struct dk_minfo ud;
295         } dks;
296         struct dk_minfo_ext *dkmext = &dks.ude;
297         struct dk_minfo *dkm = &dks.ud;
298         int error;
299         dev_t dev;
300         int otyp;
301         boolean_t validate_devid = B_FALSE;
302         ddi_devid_t devid;
303         uint64_t capacity = 0, blksz = 0, pbsize;

305         /*
306          * We must have a pathname, and it must be absolute.
307          */
308         if (vd->vdev_path == NULL || vd->vdev_path[0] != '/') {
309                 vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
310                 return (SET_ERROR(EINVAL));
311         }

313         /*
314          * Reopen the device if it's not currently open. Otherwise,
315          * just update the physical size of the device.
316          */
317         if (dvd != NULL) {
318                 if (dvd->vd_ldi_offline && dvd->vd_lh == NULL) {
319                         /*
320                          * If we are opening a device in its offline notify
321                          * context, the LDI handle was just closed. Clean
322                          * up the LDI event callbacks and free vd->vdev_tsd.
323                          */
324                         vdev_disk_free(vd);
325                 } else {
326                         ASSERT(vd->vdev_reopening);
327                         goto skip_open;
328                 }
329         }

331         /*
332          * Create vd->vdev_tsd.
```

```
333          */
334         vdev_disk_alloc(vd);
335         dvd = vd->vdev_tsd;

337         /*
338          * When opening a disk device, we want to preserve the user's original
339          * intent.  We always want to open the device by the path the user gave
340          * us, even if it is one of multiple paths to the same device.  But we
341          * also want to be able to survive disks being removed/recabled.
342          * Therefore the sequence of opening devices is:
343          *
344          * 1. Try opening the device by path.  For legacy pools without the
345          *    'whole_disk' property, attempt to fix the path by appending 's0'.
346          *
347          * 2. If the devid of the device matches the stored value, return
348          *    success.
349          *
350          * 3. Otherwise, the device may have moved.  Try opening the device
351          *    by the devid instead.
352          */
353         if (vd->vdev_devid != NULL) {
354                 if (ddi_devid_str_decode(vd->vdev_devid, &dvd->vd_devid,
355                     &dvd->vd_minor) != 0) {
356                         vd->vdev_stat.vs_aux = VDEV_AUX_BAD_LABEL;
357                         return (SET_ERROR(EINVAL));
358                 }
359         }

361         error = EINVAL;         /* presume failure */

363         if (vd->vdev_path != NULL) {

365                 if (vd->vdev_wholedisk == -1ULL) {
366                         size_t len = strlen(vd->vdev_path) + 3;
367                         char *buf = kmem_alloc(len, KM_SLEEP);

369                         (void) snprintf(buf, len, "%ss0", vd->vdev_path);

371                         error = ldi_open_by_name(buf, spa_mode(spa), kcred,
372                             &dvd->vd_lh, zfs_li);
373                         if (error == 0) {
374                                 spa_strfree(vd->vdev_path);
375                                 vd->vdev_path = buf;
376                                 vd->vdev_wholedisk = 1ULL;
377                         } else {
378                                 kmem_free(buf, len);
379                         }
380                 }

382                 /*
383                  * If we have not yet opened the device, try to open it by the
384                  * specified path.
385                  */
386                 if (error != 0) {
387                         error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
388                             kcred, &dvd->vd_lh, zfs_li);
389                 }

391                 /*
392                  * Compare the devid to the stored value.
393                  */
394                 if (error == 0 && vd->vdev_devid != NULL &&
395                     ldi_get_devid(dvd->vd_lh, &devid) == 0) {
396                         if (ddi_devid_compare(devid, dvd->vd_devid) != 0) {
397                                 error = SET_ERROR(EINVAL);
398                                 (void) ldi_close(dvd->vd_lh, spa_mode(spa),
```

```
 399                             kcred);
 400                     dvd->vd_lh = NULL;
 401             }
 402             ddi_devid_free(devid);
 403         }

 405         /*
 406          * If we succeeded in opening the device, but 'vdev_wholedisk'
 407          * is not yet set, then this must be a slice.
 408          */
 409         if (error == 0 && vd->vdev_wholedisk == -1ULL)
 410             vd->vdev_wholedisk = 0;
 411     }

 413     /*
 414      * If we were unable to open by path, or the devid check fails, open by
 415      * devid instead.
 416      */
 417     if (error != 0 && vd->vdev_devid != NULL) {
 418         error = ldi_open_by_devid(dvd->vd_devid, dvd->vd_minor,
 419             spa_mode(spa), kcred, &dvd->vd_lh, zfs_li);
 420     }

 422     /*
 423      * If all else fails, then try opening by physical path (if available)
 424      * or the logical path (if we failed due to the devid check).  While not
 425      * as reliable as the devid, this will give us something, and the higher
 426      * level vdev validation will prevent us from opening the wrong device.
 427      */
 428     if (error) {
 429         if (vd->vdev_devid != NULL)
 430             validate_devid = B_TRUE;

 432         if (vd->vdev_physpath != NULL &&
 433             (dev = ddi_pathname_to_dev_t(vd->vdev_physpath)) != NODEV)
 434             error = ldi_open_by_dev(&dev, OTYP_BLK, spa_mode(spa),
 435                 kcred, &dvd->vd_lh, zfs_li);

 437         /*
 438          * Note that we don't support the legacy auto-wholedisk support
 439          * as above.  This hasn't been used in a very long time and we
 440          * don't need to propagate its oddities to this edge condition.
 441          */
 442         if (error && vd->vdev_path != NULL)
 443             error = ldi_open_by_name(vd->vdev_path, spa_mode(spa),
 444                 kcred, &dvd->vd_lh, zfs_li);
 445     }

 447     if (error) {
 448         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
 449         return (error);
 450     }

 452     /*
 453      * Now that the device has been successfully opened, update the devid
 454      * if necessary.
 455      */
 456     if (validate_devid && spa_writeable(spa) &&
 457         ldi_get_devid(dvd->vd_lh, &devid) == 0) {
 458         if (ddi_devid_compare(devid, dvd->vd_devid) != 0) {
 459             char *vd_devid;

 461             vd_devid = ddi_devid_str_encode(devid, dvd->vd_minor);
 462             zfs_dbgmsg("vdev %s: update devid from %s, "
 463                 "to %s", vd->vdev_path, vd->vdev_devid, vd_devid);
 464             spa_strfree(vd->vdev_devid);
```

```
 465             vd->vdev_devid = spa_strdup(vd_devid);
 466             ddi_devid_str_free(vd_devid);
 467         }
 468         ddi_devid_free(devid);
 469     }

 471     /*
 472      * Once a device is opened, verify that the physical device path (if
 473      * available) is up to date.
 474      */
 475     if (ldi_get_dev(dvd->vd_lh, &dev) == 0 &&
 476         ldi_get_otyp(dvd->vd_lh, &otyp) == 0) {
 477         char *physpath, *minorname;

 479         physpath = kmem_alloc(MAXPATHLEN, KM_SLEEP);
 480         minorname = NULL;
 481         if (ddi_dev_pathname(dev, otyp, physpath) == 0 &&
 482             ldi_get_minor_name(dvd->vd_lh, &minorname) == 0 &&
 483             (vd->vdev_physpath == NULL ||
 484             strcmp(vd->vdev_physpath, physpath) != 0)) {
 485             if (vd->vdev_physpath)
 486                 spa_strfree(vd->vdev_physpath);
 487             (void) strlcat(physpath, ":", MAXPATHLEN);
 488             (void) strlcat(physpath, minorname, MAXPATHLEN);
 489             vd->vdev_physpath = spa_strdup(physpath);
 490         }
 491         if (minorname)
 492             kmem_free(minorname, strlen(minorname) + 1);
 493         kmem_free(physpath, MAXPATHLEN);
 494     }

 496     /*
 497      * Register callbacks for the LDI offline event.
 498      */
 499     if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_OFFLINE, &ecookie) ==
 500         LDI_EV_SUCCESS) {
 501         lcb = kmem_zalloc(sizeof (vdev_disk_ldi_cb_t), KM_SLEEP);
 502         list_insert_tail(&dvd->vd_ldi_cbs, lcb);
 503         (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
 504             &vdev_disk_off_callb, (void *) vd, &lcb->lcb_id);
 505     }

 507     /*
 508      * Register callbacks for the LDI degrade event.
 509      */
 510     if (ldi_ev_get_cookie(dvd->vd_lh, LDI_EV_DEGRADE, &ecookie) ==
 511         LDI_EV_SUCCESS) {
 512         lcb = kmem_zalloc(sizeof (vdev_disk_ldi_cb_t), KM_SLEEP);
 513         list_insert_tail(&dvd->vd_ldi_cbs, lcb);
 514         (void) ldi_ev_register_callbacks(dvd->vd_lh, ecookie,
 515             &vdev_disk_dgrd_callb, (void *) vd, &lcb->lcb_id);
 516     }
 517 skip_open:
 518     /*
 519      * Determine the actual size of the device.
 520      */
 521     if (ldi_get_size(dvd->vd_lh, psize) != 0) {
 522         vd->vdev_stat.vs_aux = VDEV_AUX_OPEN_FAILED;
 523         return (SET_ERROR(EINVAL));
 524     }

 526     *max_psize = *psize;

 528     /*
 529      * Determine the device's minimum transfer size.
 530      * If the ioctl isn't supported, assume DEV_BSIZE.
```

```
531                 */
532                if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFOEXT,
533                    (intptr_t)dkmext, FKIOCTL, kcred, NULL)) == 0) {
534                        capacity = dkmext->dki_capacity - 1;
535                        blksz = dkmext->dki_lbsize;
536                        pbsize = dkmext->dki_pbsize;
537                } else if ((error = ldi_ioctl(dvd->vd_lh, DKIOCGMEDIAINFO,
538                    (intptr_t)dkm, FKIOCTL, kcred, NULL)) == 0) {
539                        VDEV_DEBUG(
540                            "vdev_disk_open(\"%s\"): fallback to DKIOCGMEDIAINFO\n",
541                            vd->vdev_path);
542                        capacity = dkm->dki_capacity - 1;
543                        blksz = dkm->dki_lbsize;
544                        pbsize = blksz;
545                } else {
546                        VDEV_DEBUG("vdev_disk_open(\"%s\"): "
547                            "both DKIOCGMEDIAINFO{,EXT} calls failed, %d\n",
548                            vd->vdev_path, error);
549                        pbsize = DEV_BSIZE;
550                }

552                /*
553                 * Determine the rotation
554                 */
555                vd->vdev_rotation_rate = VDEV_RATE_UNKNOWN;
556                /* TODO: Implement when there's an ioctl which provides this info. */

558 #endif /* ! codereview */
559                *ashift = highbit(MAX(pbsize, SPA_MINBLOCKSIZE)) - 1;

561        if (vd->vdev_wholedisk == 1) {
562                int wce = 1;

564                if (error == 0) {
565                        /*
566                         * If we have the capability to expand, we'd have
567                         * found out via success from DKIOCGMEDIAINFO{,EXT}.
568                         * Adjust max_psize upward accordingly since we know
569                         * we own the whole disk now.
570                         */
571                        *max_psize += vdev_disk_get_space(vd, capacity, blksz);
572                        zfs_dbgmsg("capacity change: vdev %s, psize %llu, "
573                            "max_psize %llu", vd->vdev_path, *psize,
574                            *max_psize);
575                }

577                /*
578                 * Since we own the whole disk, try to enable disk write
579                 * caching.  We ignore errors because it's OK if we can't do it.
580                 */
581                (void) ldi_ioctl(dvd->vd_lh, DKIOCSETWCE, (intptr_t)&wce,
582                    FKIOCTL, kcred, NULL);
583        }

585        /*
586         * Clear the nowritecache bit, so that on a vdev_reopen() we will
587         * try again.
588         */
589        vd->vdev_nowritecache = B_FALSE;

591        return (0);
592 }

594 static void
595 vdev_disk_close(vdev_t *vd)
596 {
```

```
597        vdev_disk_t *dvd = vd->vdev_tsd;

599        if (vd->vdev_reopening || dvd == NULL)
600                return;

602        if (dvd->vd_minor != NULL) {
603                ddi_devid_str_free(dvd->vd_minor);
604                dvd->vd_minor = NULL;
605        }

607        if (dvd->vd_devid != NULL) {
608                ddi_devid_free(dvd->vd_devid);
609                dvd->vd_devid = NULL;
610        }

612        if (dvd->vd_lh != NULL) {
613                (void) ldi_close(dvd->vd_lh, spa_mode(vd->vdev_spa), kcred);
614                dvd->vd_lh = NULL;
615        }

617        vd->vdev_delayed_close = B_FALSE;
618        /*
619         * If we closed the LDI handle due to an offline notify from LDI,
620         * don't free vd->vdev_tsd or unregister the callbacks here;
621         * the offline finalize callback or a reopen will take care of it.
622         */
623        if (dvd->vd_ldi_offline)
624                return;

626        vdev_disk_free(vd);
627 }

629 int
630 vdev_disk_physio(vdev_t *vd, caddr_t data,
631     size_t size, uint64_t offset, int flags, boolean_t isdump)
632 {
633        vdev_disk_t *dvd = vd->vdev_tsd;

635        /*
636         * If the vdev is closed, it's likely in the REMOVED or FAULTED state.
637         * Nothing to be done here but return failure.
638         */
639        if (dvd == NULL || (dvd->vd_ldi_offline && dvd->vd_lh == NULL))
640                return (EIO);

642        ASSERT(vd->vdev_ops == &vdev_disk_ops);

644        /*
645         * If in the context of an active crash dump, use the ldi_dump(9F)
646         * call instead of ldi_strategy(9F) as usual.
647         */
648        if (isdump) {
649                ASSERT3P(dvd, !=, NULL);
650                return (ldi_dump(dvd->vd_lh, data, lbtodb(offset),
651                    lbtodb(size)));
652        }

654        return (vdev_disk_ldi_physio(dvd->vd_lh, data, size, offset, flags));
655 }

657 int
658 vdev_disk_ldi_physio(ldi_handle_t vd_lh, caddr_t data,
659     size_t size, uint64_t offset, int flags)
660 {
661        buf_t *bp;
662        int error = 0;
```

```
 664            if (vd_lh == NULL)
 665                    return (SET_ERROR(EINVAL));

 667            ASSERT(flags & B_READ || flags & B_WRITE);

 669            bp = getrbuf(KM_SLEEP);
 670            bp->b_flags = flags | B_BUSY | B_NOCACHE | B_FAILFAST;
 671            bp->b_bcount = size;
 672            bp->b_un.b_addr = (void *)data;
 673            bp->b_lblkno = lbtodb(offset);
 674            bp->b_bufsize = size;

 676            error = ldi_strategy(vd_lh, bp);
 677            ASSERT(error == 0);
 678            if ((error = biowait(bp)) == 0 && bp->b_resid != 0)
 679                    error = SET_ERROR(EIO);
 680            freerbuf(bp);

 682            return (error);
 683 }

 685 static void
 686 vdev_disk_io_intr(buf_t *bp)
 687 {
 688            vdev_buf_t *vb = (vdev_buf_t *)bp;
 689            zio_t *zio = vb->vb_io;

 691            /*
 692             * The rest of the zio stack only deals with EIO, ECKSUM, and ENXIO.
 693             * Rather than teach the rest of the stack about other error
 694             * possibilities (EFAULT, etc), we normalize the error value here.
 695             */
 696            zio->io_error = (geterror(bp) != 0 ? EIO : 0);

 698            if (zio->io_error == 0 && bp->b_resid != 0)
 699                    zio->io_error = SET_ERROR(EIO);

 701            kmem_free(vb, sizeof (vdev_buf_t));

 703            zio_interrupt(zio);
 704 }

 706 static void
 707 vdev_disk_ioctl_free(zio_t *zio)
 708 {
 709            kmem_free(zio->io_vsd, sizeof (struct dk_callback));
 710 }

 712 static const zio_vsd_ops_t vdev_disk_vsd_ops = {
 713            vdev_disk_ioctl_free,
 714            zio_vsd_default_cksum_report
 715 };

 717 static void
 718 vdev_disk_ioctl_done(void *zio_arg, int error)
 719 {
 720            zio_t *zio = zio_arg;

 722            zio->io_error = error;

 724            zio_interrupt(zio);
 725 }

 727 static int
 728 vdev_disk_io_start(zio_t *zio)
```

```
 729 {
 730            vdev_t *vd = zio->io_vd;
 731            vdev_disk_t *dvd = vd->vdev_tsd;
 732            vdev_buf_t *vb;
 733            struct dk_callback *dkc;
 734            buf_t *bp;
 735            int error;

 737            /*
 738             * If the vdev is closed, it's likely in the REMOVED or FAULTED state.
 739             * Nothing to be done here but return failure.
 740             */
 741            if (dvd == NULL || (dvd->vd_ldi_offline && dvd->vd_lh == NULL)) {
 742                    zio->io_error = ENXIO;
 743                    return (ZIO_PIPELINE_CONTINUE);
 744            }

 746            if (zio->io_type == ZIO_TYPE_IOCTL) {
 747                    /* XXPOLICY */
 748                    if (!vdev_readable(vd)) {
 749                            zio->io_error = SET_ERROR(ENXIO);
 750                            return (ZIO_PIPELINE_CONTINUE);
 751                    }

 753                    switch (zio->io_cmd) {

 755                    case DKIOCFLUSHWRITECACHE:

 757                            if (zfs_nocacheflush)
 758                                    break;

 760                            if (vd->vdev_nowritecache) {
 761                                    zio->io_error = SET_ERROR(ENOTSUP);
 762                                    break;
 763                            }

 765                            zio->io_vsd = dkc = kmem_alloc(sizeof (*dkc), KM_SLEEP);
 766                            zio->io_vsd_ops = &vdev_disk_vsd_ops;

 768                            dkc->dkc_callback = vdev_disk_ioctl_done;
 769                            dkc->dkc_flag = FLUSH_VOLATILE;
 770                            dkc->dkc_cookie = zio;

 772                            error = ldi_ioctl(dvd->vd_lh, zio->io_cmd,
 773                                (uintptr_t)dkc, FKIOCTL, kcred, NULL);

 775                            if (error == 0) {
 776                                    /*
 777                                     * The ioctl will be done asychronously,
 778                                     * and will call vdev_disk_ioctl_done()
 779                                     * upon completion.
 780                                     */
 781                                    return (ZIO_PIPELINE_STOP);
 782                            }

 784                            if (error == ENOTSUP || error == ENOTTY) {
 785                                    /*
 786                                     * If we get ENOTSUP or ENOTTY, we know that
 787                                     * no future attempts will ever succeed.
 788                                     * In this case we set a persistent bit so
 789                                     * that we don't bother with the ioctl in the
 790                                     * future.
 791                                     */
 792                                    vd->vdev_nowritecache = B_TRUE;
 793                            }
 794                            zio->io_error = error;
```

```
 796                             break;

 798                     default:
 799                             zio->io_error = SET_ERROR(ENOTSUP);
 800                     }

 802                     return (ZIO_PIPELINE_CONTINUE);
 803             }

 805             vb = kmem_alloc(sizeof (vdev_buf_t), KM_SLEEP);

 807             vb->vb_io = zio;
 808             bp = &vb->vb_buf;

 810             bioinit(bp);
 811             bp->b_flags = B_BUSY | B_NOCACHE |
 812                 (zio->io_type == ZIO_TYPE_READ ? B_READ : B_WRITE);
 813             if (!(zio->io_flags & (ZIO_FLAG_IO_RETRY | ZIO_FLAG_TRYHARD)))
 814                     bp->b_flags |= B_FAILFAST;
 815             bp->b_bcount = zio->io_size;
 816             bp->b_un.b_addr = zio->io_data;
 817             bp->b_lblkno = lbtodb(zio->io_offset);
 818             bp->b_bufsize = zio->io_size;
 819             bp->b_iodone = (int (*)())vdev_disk_io_intr;

 821             /* ldi_strategy() will return non-zero only on programming errors */
 822             VERIFY(ldi_strategy(dvd->vd_lh, bp) == 0);

 824             return (ZIO_PIPELINE_STOP);
 825     }

 827     static void
 828     vdev_disk_io_done(zio_t *zio)
 829     {
 830             vdev_t *vd = zio->io_vd;

 832             /*
 833              * If the device returned EIO, then attempt a DKIOCSTATE ioctl to see if
 834              * the device has been removed.  If this is the case, then we trigger an
 835              * asynchronous removal of the device. Otherwise, probe the device and
 836              * make sure it's still accessible.
 837              */
 838             if (zio->io_error == EIO && !vd->vdev_remove_wanted) {
 839                     vdev_disk_t *dvd = vd->vdev_tsd;
 840                     int state = DKIO_NONE;

 842                     if (ldi_ioctl(dvd->vd_lh, DKIOCSTATE, (intptr_t)&state,
 843                         FKIOCTL, kcred, NULL) == 0 && state != DKIO_INSERTED) {
 844                             /*
 845                              * We post the resource as soon as possible, instead of
 846                              * when the async removal actually happens, because the
 847                              * DE is using this information to discard previous I/O
 848                              * errors.
 849                              */
 850                             zfs_post_remove(zio->io_spa, vd);
 851                             vd->vdev_remove_wanted = B_TRUE;
 852                             spa_async_request(zio->io_spa, SPA_ASYNC_REMOVE);
 853                     } else if (!vd->vdev_delayed_close) {
 854                             vd->vdev_delayed_close = B_TRUE;
 855                     }
 856             }
 857     }

 859     vdev_ops_t vdev_disk_ops = {
 860             vdev_disk_open,
```

```
 861             vdev_disk_close,
 862             vdev_default_asize,
 863             vdev_disk_io_start,
 864             vdev_disk_io_done,
 865             NULL,
 866             vdev_disk_hold,
 867             vdev_disk_rele,
 868             VDEV_TYPE_DISK,         /* name of this vdev type */
 869             B_TRUE                  /* leaf vdev */
 870     };

 872     /*
 873      * Given the root disk device devid or pathname, read the label from
 874      * the device, and construct a configuration nvlist.
 875      */
 876     int
 877     vdev_disk_read_rootlabel(char *devpath, char *devid, nvlist_t **config)
 878     {
 879             ldi_handle_t vd_lh;
 880             vdev_label_t *label;
 881             uint64_t s, size;
 882             int l;
 883             ddi_devid_t tmpdevid;
 884             int error = -1;
 885             char *minor_name;

 887             /*
 888              * Read the device label and build the nvlist.
 889              */
 890             if (devid != NULL && ddi_devid_str_decode(devid, &tmpdevid,
 891                 &minor_name) == 0) {
 892                     error = ldi_open_by_devid(tmpdevid, minor_name,
 893                         FREAD, kcred, &vd_lh, zfs_li);
 894                     ddi_devid_free(tmpdevid);
 895                     ddi_devid_str_free(minor_name);
 896             }

 898             if (error && (error = ldi_open_by_name(devpath, FREAD, kcred, &vd_lh,
 899                 zfs_li)))
 900                     return (error);

 902             if (ldi_get_size(vd_lh, &s)) {
 903                     (void) ldi_close(vd_lh, FREAD, kcred);
 904                     return (SET_ERROR(EIO));
 905             }

 907             size = P2ALIGN_TYPED(s, sizeof (vdev_label_t), uint64_t);
 908             label = kmem_alloc(sizeof (vdev_label_t), KM_SLEEP);

 910             *config = NULL;
 911             for (l = 0; l < VDEV_LABELS; l++) {
 912                     uint64_t offset, state, txg = 0;

 914                     /* read vdev label */
 915                     offset = vdev_label_offset(size, l, 0);
 916                     if (vdev_disk_ldi_physio(vd_lh, (caddr_t)label,
 917                         VDEV_SKIP_SIZE + VDEV_PHYS_SIZE, offset, B_READ) != 0)
 918                             continue;

 920                     if (nvlist_unpack(label->vl_vdev_phys.vp_nvlist,
 921                         sizeof (label->vl_vdev_phys.vp_nvlist), config, 0) != 0) {
 922                             *config = NULL;
 923                             continue;
 924                     }

 926                     if (nvlist_lookup_uint64(*config, ZPOOL_CONFIG_POOL_STATE,
```

```
927                          &state) != 0 || state >= POOL_STATE_DESTROYED) {
928                                  nvlist_free(*config);
929                                  *config = NULL;
930                                  continue;
931                          }

933                          if (nvlist_lookup_uint64(*config, ZPOOL_CONFIG_POOL_TXG,
934                              &txg) != 0 || txg == 0) {
935                                  nvlist_free(*config);
936                                  *config = NULL;
937                                  continue;
938                          }

940                          break;
941                  }

943          kmem_free(label, sizeof (vdev_label_t));
944          (void) ldi_close(vd_lh, FREAD, kcred);
945          if (*config == NULL)
946                  error = SET_ERROR(EIDRM);

948          return (error);
949 }
```

```
**********************************************************
   16593 Thu Nov 21 00:19:08 2013
new/usr/src/uts/common/fs/zfs/vdev_mirror.c
4334 Improve ZFS N-way mirror read performance
**********************************************************
   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 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 /*
  27  * Copyright (c) 2013 by Delphix. All rights reserved.
  28  * Copyright (c) 2013 Steven Hartland. All rights reserved.
  29 #endif /* ! codereview */
  30  */

  32 #include <sys/zfs_context.h>
  33 #include <sys/spa.h>
  34 #include <sys/vdev_impl.h>
  35 #include <sys/zio.h>
  36 #include <sys/fs/zfs.h>

  38 /*
  39  * Virtual device vector for mirroring.
  40  */

  42 typedef struct mirror_child {
  43         vdev_t          *mc_vd;
  44         uint64_t        mc_offset;
  45         int             mc_error;
  46         int             mc_load;
  47 #endif /* ! codereview */
  48         uint8_t         mc_tried;
  49         uint8_t         mc_skipped;
  50         uint8_t         mc_speculative;
  51 } mirror_child_t;

  53 typedef struct mirror_map {
  54         int             *mm_preferred;
  55         int             mm_preferred_cnt;
  56 #endif /* ! codereview */
  57         int             mm_children;
  58         boolean_t       mm_replacing;
  59         boolean_t       mm_root;
  60         mirror_child_t  mm_child[];
  28         int             mm_replacing;
```

```
  29         int             mm_preferred;
  30         int             mm_root;
  31         mirror_child_t  mm_child[1];
  61 } mirror_map_t;

  63 int zfs_vdev_mirror_shift = 21;

  65 /*
  66  * The load configuration settings below are tuned by default for
  67  * the case where all devices are of the same rotational type.
  68  *
  69  * If there is a mixture of rotating and non-rotating media, setting
  70  * non_rotating_seek_inc to 0 may well provide better results as it
  71  * will direct more reads to the non-rotating vdevs which are more
  72  * likely to have a higher performance.
  73  */

  75 /* Rotating media load calculation configuration. */
  76 /* Rotating media load increment for non-seeking I/O's. */
  77 int zfs_vdev_mirror_rotating_inc = 0;

  79 /* Rotating media load increment for seeking I/O's. */
  80 int zfs_vdev_mirror_rotating_seek_inc = 5;

  82 /*
  83  * Offset in bytes from the last I/O which triggers a reduced rotating media
  84  * seek increment.
  85  */
  86 int zfs_vdev_mirror_rotating_seek_offset = 1 * 1024 * 1024;

  88 /* Non-rotating media load calculation configuration. */
  89 /* Non-rotating media load increment for non-seeking I/O's. */
  90 int zfs_vdev_mirror_non_rotating_inc = 0;

  92 /* Non-rotating media load increment for seeking I/O's. */
  93 int zfs_vdev_mirror_non_rotating_seek_inc = 1;

  95 static inline size_t
  96 vdev_mirror_map_size(int children)
  97 {
  98         return (offsetof(mirror_map_t, mm_child[children]) +
  99             sizeof (int) * children);
 100 }

 102 static inline mirror_map_t *
 103 vdev_mirror_map_alloc(int children, boolean_t replacing, boolean_t root)
 104 {
 105         mirror_map_t *mm;

 107         mm = kmem_zalloc(vdev_mirror_map_size(children), KM_SLEEP);
 108         mm->mm_children = children;
 109         mm->mm_replacing = replacing;
 110         mm->mm_root = root;
 111         mm->mm_preferred = (int *)((uintptr_t)mm +
 112             offsetof(mirror_map_t, mm_child[children]));

 114         return (mm);
 115 }
  34 int vdev_mirror_shift = 21;

 117 static void
 118 vdev_mirror_map_free(zio_t *zio)
 119 {
 120         mirror_map_t *mm = zio->io_vsd;

 122         kmem_free(mm, vdev_mirror_map_size(mm->mm_children));
```

```
    41          kmem_free(mm, offsetof(mirror_map_t, mm_child[mm->mm_children]));
   123 }
```
_____*unchanged_portion_omitted_*

```
   130 /*
   131  * Calculate and return the load of the specified vdev adjusted for a zio at
   132  * the given offset.
   133  *
   134  * The calcuation takes into account the vdev's:
   135  * 1. Rotation rate
   136  * 2. The distance of zio_offset from the last queued request
   137  */
   138 static int
   139 vdev_mirror_load(mirror_map_t *mm, vdev_t *vd, uint64_t zio_offset)
   140 {
   141          uint64_t lastoffset;
   142          int load;

   144          /* All DVAs have equal weight at the root. */
   145          if (mm->mm_root)
   146                  return (INT_MAX);

   148          /*
   149           * We don't return INT_MAX if the device is resilvering i.e.
   150           * vdev_resilver_txg != 0 as when tested performance was slightly
   151           * worse overall when resilvering with compared to without.
   152           */

   154          /* Standard load based on pending queue length. */
   155          load = vdev_queue_length(vd);
   156          lastoffset = vdev_queue_last_queued_offset(vd);

   158          if (vd->vdev_rotation_rate == VDEV_RATE_NON_ROTATING) {
   159                  /* Non-rotating media. */
   160                  if (lastoffset == zio_offset)
   161                          return (load + zfs_vdev_mirror_non_rotating_inc);

   163                  /*
   164                   * Apply a seek penalty even for non-rotating devices as
   165                   * sequential I/O'a can be aggregated into fewer operations
   166                   * on the device, thus avoiding unnecessary per-command
   167                   * overhead and boosting performance.
   168                   */
   169                  return (load + zfs_vdev_mirror_non_rotating_seek_inc);
   170          }

   172          /* Rotating media I/O's which directly follow the last I/O. */
   173          if (lastoffset == zio_offset)
   174                  return (load + zfs_vdev_mirror_rotating_inc);

   176          /*
   177           * Apply half the seek increment to I/O's within seek offset
   178           * of the last I/O queued to this vdev as they should incure less
   179           * of a seek increment.
   180           */
   181          if (ABS(lastoffset - zio_offset) <
   182              zfs_vdev_mirror_rotating_seek_offset)
   183                  return (load + (zfs_vdev_mirror_rotating_seek_inc / 2));

   185          /* Apply the full seek increment to all other I/O's. */
   186          return (load + zfs_vdev_mirror_rotating_seek_inc);
   187 }


   190 #endif /* ! codereview */
   191 static mirror_map_t *
```

```
   192 vdev_mirror_map_init(zio_t *zio)
    49 vdev_mirror_map_alloc(zio_t *zio)
   193 {
   194          mirror_map_t *mm = NULL;
   195          mirror_child_t *mc;
   196          vdev_t *vd = zio->io_vd;
   197          int c;
    54          int c, d;

   199          if (vd == NULL) {
   200                  dva_t *dva = zio->io_bp->blk_dva;
   201                  spa_t *spa = zio->io_spa;

   203                  mm = vdev_mirror_map_alloc(BP_GET_NDVAS(zio->io_bp), B_FALSE,
   204                      B_TRUE);
    60                  c = BP_GET_NDVAS(zio->io_bp);

    62                  mm = kmem_zalloc(offsetof(mirror_map_t, mm_child[c]), KM_SLEEP);
    63                  mm->mm_children = c;
    64                  mm->mm_replacing = B_FALSE;
    65                  mm->mm_preferred = spa_get_random(c);
    66                  mm->mm_root = B_TRUE;

    68                  /*
    69                   * Check the other, lower-index DVAs to see if they're on
    70                   * the same vdev as the child we picked.  If they are, use
    71                   * them since they are likely to have been allocated from
    72                   * the primary metaslab in use at the time, and hence are
    73                   * more likely to have locality with single-copy data.
    74                   */
    75                  for (c = mm->mm_preferred, d = c - 1; d >= 0; d--) {
    76                          if (DVA_GET_VDEV(&dva[d]) == DVA_GET_VDEV(&dva[c]))
    77                                  mm->mm_preferred = d;
    78                  }

   205                  for (c = 0; c < mm->mm_children; c++) {
   206                          mc = &mm->mm_child[c];

   207                          mc->mc_vd = vdev_lookup_top(spa, DVA_GET_VDEV(&dva[c]));
   208                          mc->mc_offset = DVA_GET_OFFSET(&dva[c]);
   209                  }
   210          } else {
   211                  mm = vdev_mirror_map_alloc(vd->vdev_children,
   212                      (vd->vdev_ops == &vdev_replacing_ops ||
   213                      vd->vdev_ops == &vdev_spare_ops), B_FALSE);
    87                  c = vd->vdev_children;

    89                  mm = kmem_zalloc(offsetof(mirror_map_t, mm_child[c]), KM_SLEEP);
    90                  mm->mm_children = c;
    91                  mm->mm_replacing = (vd->vdev_ops == &vdev_replacing_ops ||
    92                      vd->vdev_ops == &vdev_spare_ops);
    93                  mm->mm_preferred = mm->mm_replacing ? 0 :
    94                      (zio->io_offset >> vdev_mirror_shift) % c;
    95                  mm->mm_root = B_FALSE;

   214                  for (c = 0; c < mm->mm_children; c++) {
   215                          mc = &mm->mm_child[c];
   216                          mc->mc_vd = vd->vdev_child[c];
   217                          mc->mc_offset = zio->io_offset;
   218                  }
   219          }

   221          zio->io_vsd = mm;
   222          zio->io_vsd_ops = &vdev_mirror_vsd_ops;
   223          return (mm);
   224 }
```
_____*unchanged_portion_omitted_*

```
 304 /*
 305  * Check the other, lower-index DVAs to see if they're on the same
 306  * vdev as the child we picked.  If they are, use them since they
 307  * are likely to have been allocated from the primary metaslab in
 308  * use at the time, and hence are more likely to have locality with
 309  * single-copy data.
 310  */
 311 static int
 312 vdev_mirror_dva_select(zio_t *zio, int preferred)
 313 {
 314         dva_t *dva = zio->io_bp->blk_dva;
 315         int c;

 317         for (c = preferred - 1; c >= 0; c--) {
 318                 if (DVA_GET_VDEV(&dva[c]) == DVA_GET_VDEV(&dva[preferred]))
 319                         preferred = c;
 320         }
 321         return (preferred);
 322 }

 324 static int
 325 vdev_mirror_preferred_child_randomize(zio_t *zio)
 326 {
 327         mirror_map_t *mm = zio->io_vsd;
 328         int p;

 330         if (mm->mm_root) {
 331                 p = spa_get_random(mm->mm_preferred_cnt);
 332                 return (vdev_mirror_dva_select(zio, mm->mm_preferred[p]));
 333         }

 335         /*
 336          * To ensure we don't always favour the first matching vdev,
 337          * which could lead to wear leveling issues on SSD's, we
 338          * use the I/O offset as a pseudo random seed into the vdevs
 339          * which have the lowest load.
 340          */
 341         p = (zio->io_offset >> zfs_vdev_mirror_shift) % mm->mm_preferred_cnt;
 342         return (mm->mm_preferred[p]);
 343 }

 345 /*
 346  * Try to find a vdev whose DTL doesn't contain the block we want to read
 347  * prefering vdevs based on determined load.
 348  *
 188  * Try to find a child whose DTL doesn't contain the block we want to read.
 349  * If we can't, try the read on any vdev we haven't already tried.
 350  */
 351 static int
 352 vdev_mirror_child_select(zio_t *zio)
 353 {
 354         mirror_map_t *mm = zio->io_vsd;
 195         mirror_child_t *mc;
 355         uint64_t txg = zio->io_txg;
 356         int c, lowest_load;
 197         int i, c;

 358         ASSERT(zio->io_bp == NULL || BP_PHYSICAL_BIRTH(zio->io_bp) == txg);

 360         lowest_load = INT_MAX;
 361         mm->mm_preferred_cnt = 0;
 362         for (c = 0; c < mm->mm_children; c++) {
 363                 mirror_child_t *mc;

 201         /*
```

```
 202          * Try to find a child whose DTL doesn't contain the block to read.
 203          * If a child is known to be completely inaccessible (indicated by
 204          * vdev_readable() returning B_FALSE), don't even try.
 205          */
 206         for (i = 0, c = mm->mm_preferred; i < mm->mm_children; i++, c++) {
 207                 if (c >= mm->mm_children)
 208                         c = 0;
 365                 mc = &mm->mm_child[c];
 366                 if (mc->mc_tried || mc->mc_skipped)
 367                         continue;

 369 #endif /* ! codereview */
 370                 if (!vdev_readable(mc->mc_vd)) {
 371                         mc->mc_error = SET_ERROR(ENXIO);
 372                         mc->mc_tried = 1;       /* don't even try */
 373                         mc->mc_skipped = 1;
 374                         continue;
 375                 }

 377                 if (vdev_dtl_contains(mc->mc_vd, DTL_MISSING, txg, 1)) {
 212                 if (!vdev_dtl_contains(mc->mc_vd, DTL_MISSING, txg, 1))
 213                         return (c);
 378                         mc->mc_error = SET_ERROR(ESTALE);
 379                         mc->mc_skipped = 1;
 380                         mc->mc_speculative = 1;
 381                         continue;
 382                 }

 384                 mc->mc_load = vdev_mirror_load(mm, mc->mc_vd, mc->mc_offset);
 385                 if (mc->mc_load > lowest_load)
 386                         continue;

 388                 if (mc->mc_load < lowest_load) {
 389                         lowest_load = mc->mc_load;
 390                         mm->mm_preferred_cnt = 0;
 391                 }
 392                 mm->mm_preferred[mm->mm_preferred_cnt] = c;
 393                 mm->mm_preferred_cnt++;
 394         }

 396         if (mm->mm_preferred_cnt == 1) {
 397                 vdev_queue_register_last_queued_offset(
 398                     mm->mm_child[mm->mm_preferred[0]].mc_vd, zio);
 399                 return (mm->mm_preferred[0]);
 400         }

 402         if (mm->mm_preferred_cnt > 1) {
 403                 int c = vdev_mirror_preferred_child_randomize(zio);

 405                 vdev_queue_register_last_queued_offset(mm->mm_child[c].mc_vd,
 406                     zio);
 407                 return (c);
 408 #endif /* ! codereview */
 409         }

 411         /*
 412          * Every device is either missing or has this txg in its DTL.
 413          * Look for any child we haven't already tried before giving up.
 414          */
 415         for (c = 0; c < mm->mm_children; c++) {
 416                 if (!mm->mm_child[c].mc_tried) {
 417                         vdev_queue_register_last_queued_offset(
 418                             mm->mm_child[c].mc_vd, zio);
 217                 for (c = 0; c < mm->mm_children; c++)
 218                         if (!mm->mm_child[c].mc_tried)
 419                         return (c);
```

```
420                        }
421                }
422 #endif /* ! codereview */

424            /*
425             * Every child failed.  There's no place left to look.
426             */
427            return (-1);
428 }

430 static int
431 vdev_mirror_io_start(zio_t *zio)
432 {
433        mirror_map_t *mm;
434        mirror_child_t *mc;
435        int c, children;

437        mm = vdev_mirror_map_init(zio);
220        mm = vdev_mirror_map_alloc(zio);

439        if (zio->io_type == ZIO_TYPE_READ) {
440                if ((zio->io_flags & ZIO_FLAG_SCRUB) && !mm->mm_replacing) {
441                        /*
442                         * For scrubbing reads we need to allocate a read
443                         * buffer for each child and issue reads to all
444                         * children.  If any child succeeds, it will copy its
445                         * data into zio->io_data in vdev_mirror_scrub_done.
446                         */
447                        for (c = 0; c < mm->mm_children; c++) {
448                                mc = &mm->mm_child[c];
449                                zio_nowait(zio_vdev_child_io(zio, zio->io_bp,
450                                    mc->mc_vd, mc->mc_offset,
451                                        zio_buf_alloc(zio->io_size), zio->io_size,
452                                        zio->io_type, zio->io_priority, 0,
453                                        vdev_mirror_scrub_done, mc));
454                        }
455                        return (ZIO_PIPELINE_CONTINUE);
456                }
457                /*
458                 * For normal reads just pick one child.
459                 */
460                c = vdev_mirror_child_select(zio);
461                children = (c >= 0);
462        } else {
463                ASSERT(zio->io_type == ZIO_TYPE_WRITE);

465                /*
466                 * Writes go to all children.
467                 */
468                c = 0;
469                children = mm->mm_children;
470        }

472        while (children--) {
473                mc = &mm->mm_child[c];
474                zio_nowait(zio_vdev_child_io(zio, zio->io_bp,
475                    mc->mc_vd, mc->mc_offset, zio->io_data, zio->io_size,
476                    zio->io_type, zio->io_priority, 0,
477                    vdev_mirror_child_done, mc));
478                c++;
479        }

481        return (ZIO_PIPELINE_CONTINUE);
482 }
_____unchanged_portion_omitted_
```

     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 2009 Sun Microsystems, Inc. All rights reserved.
    23   * Use is subject to license terms.
    24   */

    26 /*
    27   * Copyright (c) 2013 by Delphix. All rights reserved.
    28   * Copyright (c) 2013 Steven Hartland. All rights reserved.
    29 #endif /* ! codereview */
    30   */

    32 #include <sys/zfs_context.h>
    33 #include <sys/vdev_impl.h>
    34 #include <sys/spa_impl.h>
    35 #include <sys/zio.h>
    36 #include <sys/avl.h>
    37 #include <sys/dsl_pool.h>

    39 /*
    40   * ZFS I/O Scheduler
    41   * ---------------
    42   *
    43   * ZFS issues I/O operations to leaf vdevs to satisfy and complete zios. The
    44   * I/O scheduler determines when and in what order those operations are
    45   * issued. The I/O scheduler divides operations into five I/O classes
    46   * prioritized in the following order: sync read, sync write, async read,
    47   * async write, and scrub/resilver. Each queue defines the minimum and
    48   * maximum number of concurrent operations that may be issued to the device.
    49   * In addition, the device has an aggregate maximum. Note that the sum of the
    50   * per-queue minimums must not exceed the aggregate maximum, and if the
    51   * aggregate maximum is equal to or greater than the sum of the per-queue
    52   * maximums, the per-queue minimum has no effect.
    53   *
    54   * For many physical devices, throughput increases with the number of
    55   * concurrent operations, but latency typically suffers. Further, physical
    56   * devices typically have a limit at which more concurrent operations have no
    57   * effect on throughput or can actually cause it to decrease.
    58   *
    59   * The scheduler selects the next operation to issue by first looking for an
    60   * I/O class whose minimum has not been satisfied. Once all are satisfied and
    61   * the aggregate maximum has not been hit, the scheduler looks for classes

    62   * whose maximum has not been satisfied. Iteration through the I/O classes is
    63   * done in the order specified above. No further operations are issued if the
    64   * aggregate maximum number of concurrent operations has been hit or if there
    65   * are no operations queued for an I/O class that has not hit its maximum.
    66   * Every time an i/o is queued or an operation completes, the I/O scheduler
    67   * looks for new operations to issue.
    68   *
    69   * All I/O classes have a fixed maximum number of outstanding operations
    70   * except for the async write class. Asynchronous writes represent the data
    71   * that is committed to stable storage during the syncing stage for
    72   * transaction groups (see txg.c). Transaction groups enter the syncing state
    73   * periodically so the number of queued async writes will quickly burst up and
    74   * then bleed down to zero. Rather than servicing them as quickly as possible,
    75   * the I/O scheduler changes the maximum number of active async write i/os
    76   * according to the amount of dirty data in the pool (see dsl_pool.c). Since
    77   * both throughput and latency typically increase with the number of
    78   * concurrent operations issued to physical devices, reducing the burstiness
    79   * in the number of concurrent operations also stabilizes the response time of
    80   * operations from other -- and in particular synchronous -- queues. In broad
    81   * strokes, the I/O scheduler will issue more concurrent operations from the
    82   * async write queue as there's more dirty data in the pool.
    83   *
    84   * Async Writes
    85   *
    86   * The number of concurrent operations issued for the async write I/O class
    87   * follows a piece-wise linear function defined by a few adjustable points.
    88   *
    89   * | o---------| <-- zfs_vdev_async_write_max_active
    90   *   ^ | /^
    91   * | | /
    92   * active | /
    93   * I/O | /
    94   * count | /
    95   * | /
    96   * |-----------o   /   <-- zfs_vdev_async_write_min_active
    97   *   0 |_____^_____/   100% of zfs_dirty_data_max
    98   *   0% |
    99   * |
   100   * | `-- zfs_vdev_async_write_active_max_dirty_percent
   101   *   `--------- zfs_vdev_async_write_active_min_dirty_percent
   102   *
   103   * Until the amount of dirty data exceeds a minimum percentage of the dirty
   104   * data allowed in the pool, the I/O scheduler will limit the number of
   105   * concurrent operations to the minimum. As that threshold is crossed, the
   106   * number of concurrent operations issued increases linearly to the maximum at
   107   * the specified maximum percentage of the dirty data allowed in the pool.
   108   *
   109   * Ideally, the amount of dirty data on a busy pool will stay in the sloped
   110   * part of the function between zfs_vdev_async_write_active_min_dirty_percent
   111   * and zfs_vdev_async_write_active_max_dirty_percent. If it exceeds the
   112   * maximum percentage, this indicates that the rate of incoming data is
   113   * greater than the rate that the backend storage can handle. In this case, we
   114   * must further throttle incoming writes (see dmu_tx_delay() for details).
   115   */

   117 /*
   118   * The maximum number of i/os active to each device. Ideally, this will be >=
   119   * the sum of each queue's max_active. It must be at least the sum of each
   120   * queue's min_active.
   121   */
   122 uint32_t zfs_vdev_max_active = 1000;

   124 /*
   125   * Per-queue limits on the number of i/os active to each device. If the
   126   * sum of the queue's max_active is < zfs_vdev_max_active, then the
   127   * min_active comes into play. We will send min_active from each queue,

```
 128        * and then select from queues in the order defined by zio_priority_t.
 129        *
 130        * In general, smaller max_active's will lead to lower latency of synchronous
 131        * operations.  Larger max_active's may lead to higher overall throughput,
 132        * depending on underlying storage.
 133        *
 134        * The ratio of the queues' max_actives determines the balance of performance
 135        * between reads, writes, and scrubs.  E.g., increasing
 136        * zfs_vdev_scrub_max_active will cause the scrub or resilver to complete
 137        * more quickly, but reads and writes to have higher latency and lower
 138        * throughput.
 139        */
 140 uint32_t zfs_vdev_sync_read_min_active = 10;
 141 uint32_t zfs_vdev_sync_read_max_active = 10;
 142 uint32_t zfs_vdev_sync_write_min_active = 10;
 143 uint32_t zfs_vdev_sync_write_max_active = 10;
 144 uint32_t zfs_vdev_async_read_min_active = 1;
 145 uint32_t zfs_vdev_async_read_max_active = 3;
 146 uint32_t zfs_vdev_async_write_min_active = 1;
 147 uint32_t zfs_vdev_async_write_max_active = 10;
 148 uint32_t zfs_vdev_scrub_min_active = 1;
 149 uint32_t zfs_vdev_scrub_max_active = 2;

 151 /*
 152  * When the pool has less than zfs_vdev_async_write_active_min_dirty_percent
 153  * dirty data, use zfs_vdev_async_write_min_active.  When it has more than
 154  * zfs_vdev_async_write_active_max_dirty_percent, use
 155  * zfs_vdev_async_write_max_active. The value is linearly interpolated
 156  * between min and max.
 157  */
 158 int zfs_vdev_async_write_active_min_dirty_percent = 30;
 159 int zfs_vdev_async_write_active_max_dirty_percent = 60;

 161 /*
 162  * To reduce IOPs, we aggregate small adjacent I/Os into one large I/O.
 163  * For read I/Os, we also aggregate across small adjacency gaps; for writes
 164  * we include spans of optional I/Os to aid aggregation at the disk even when
 165  * they aren't able to help us aggregate at this level.
 166  */
 167 int zfs_vdev_aggregation_limit = SPA_MAXBLOCKSIZE;
 168 int zfs_vdev_read_gap_limit = 32 << 10;
 169 int zfs_vdev_write_gap_limit = 4 << 10;

 171 int
 172 vdev_queue_offset_compare(const void *x1, const void *x2)
 173 {
 174        const zio_t *z1 = x1;
 175        const zio_t *z2 = x2;

 177        if (z1->io_offset < z2->io_offset)
 178                return (-1);
 179        if (z1->io_offset > z2->io_offset)
 180                return (1);

 182        if (z1 < z2)
 183                return (-1);
 184        if (z1 > z2)
 185                return (1);

 187        return (0);
 188 }

 190 int
 191 vdev_queue_timestamp_compare(const void *x1, const void *x2)
 192 {
 193        const zio_t *z1 = x1;
```

```
 194        const zio_t *z2 = x2;

 196        if (z1->io_timestamp < z2->io_timestamp)
 197                return (-1);
 198        if (z1->io_timestamp > z2->io_timestamp)
 199                return (1);

 201        if (z1 < z2)
 202                return (-1);
 203        if (z1 > z2)
 204                return (1);

 206        return (0);
 207 }

 209 void
 210 vdev_queue_init(vdev_t *vd)
 211 {
 212        vdev_queue_t *vq = &vd->vdev_queue;

 214        mutex_init(&vq->vq_lock, NULL, MUTEX_DEFAULT, NULL);
 215        vq->vq_vdev = vd;

 217        avl_create(&vq->vq_active_tree, vdev_queue_offset_compare,
 218            sizeof (zio_t), offsetof(struct zio, io_queue_node));

 220        for (zio_priority_t p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++) {
 221                /*
 222                 * The synchronous i/o queues are FIFO rather than LBA ordered.
 223                 * This provides more consistent latency for these i/os, and
 224                 * they tend to not be tightly clustered anyway so there is
 225                 * little to no throughput loss.
 226                 */
 227                boolean_t fifo = (p == ZIO_PRIORITY_SYNC_READ ||
 228                    p == ZIO_PRIORITY_SYNC_WRITE);
 229                avl_create(&vq->vq_class[p].vqc_queued_tree,
 230                    fifo ? vdev_queue_timestamp_compare :
 231                    vdev_queue_offset_compare,
 232                    sizeof (zio_t), offsetof(struct zio, io_queue_node));
 233        }

 235        vq->vq_last_queued_offset = 0;
 236 #endif /* ! codereview */
 237 }

 239 void
 240 vdev_queue_fini(vdev_t *vd)
 241 {
 242        vdev_queue_t *vq = &vd->vdev_queue;

 244        for (zio_priority_t p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++)
 245                avl_destroy(&vq->vq_class[p].vqc_queued_tree);
 246        avl_destroy(&vq->vq_active_tree);

 248        mutex_destroy(&vq->vq_lock);
 249 }

 251 static void
 252 vdev_queue_io_add(vdev_queue_t *vq, zio_t *zio)
 253 {
 254        spa_t *spa = zio->io_spa;
 255        ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
 256        avl_add(&vq->vq_class[zio->io_priority].vqc_queued_tree, zio);

 258        mutex_enter(&spa->spa_iokstat_lock);
 259        spa->spa_queue_stats[zio->io_priority].spa_queued++;
```

```
260            if (spa->spa_iokstat != NULL)
261                    kstat_waitq_enter(spa->spa_iokstat->ks_data);
262            mutex_exit(&spa->spa_iokstat_lock);
263 }

265 static void
266 vdev_queue_io_remove(vdev_queue_t *vq, zio_t *zio)
267 {
268            spa_t *spa = zio->io_spa;
269            ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
270            avl_remove(&vq->vq_class[zio->io_priority].vqc_queued_tree, zio);

272            mutex_enter(&spa->spa_iokstat_lock);
273            ASSERT3U(spa->spa_queue_stats[zio->io_priority].spa_queued, >, 0);
274            spa->spa_queue_stats[zio->io_priority].spa_queued--;
275            if (spa->spa_iokstat != NULL)
276                    kstat_waitq_exit(spa->spa_iokstat->ks_data);
277            mutex_exit(&spa->spa_iokstat_lock);
278 }

280 static void
281 vdev_queue_pending_add(vdev_queue_t *vq, zio_t *zio)
282 {
283            spa_t *spa = zio->io_spa;
284            ASSERT(MUTEX_HELD(&vq->vq_lock));
285            ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
286            vq->vq_class[zio->io_priority].vqc_active++;
287            avl_add(&vq->vq_active_tree, zio);

289            mutex_enter(&spa->spa_iokstat_lock);
290            spa->spa_queue_stats[zio->io_priority].spa_active++;
291            if (spa->spa_iokstat != NULL)
292                    kstat_runq_enter(spa->spa_iokstat->ks_data);
293            mutex_exit(&spa->spa_iokstat_lock);
294 }

296 static void
297 vdev_queue_pending_remove(vdev_queue_t *vq, zio_t *zio)
298 {
299            spa_t *spa = zio->io_spa;
300            ASSERT(MUTEX_HELD(&vq->vq_lock));
301            ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
302            vq->vq_class[zio->io_priority].vqc_active--;
303            avl_remove(&vq->vq_active_tree, zio);

305            mutex_enter(&spa->spa_iokstat_lock);
306            ASSERT3U(spa->spa_queue_stats[zio->io_priority].spa_active, >, 0);
307            spa->spa_queue_stats[zio->io_priority].spa_active--;
308            if (spa->spa_iokstat != NULL) {
309                    kstat_io_t *ksio = spa->spa_iokstat->ks_data;

311                    kstat_runq_exit(spa->spa_iokstat->ks_data);
312                    if (zio->io_type == ZIO_TYPE_READ) {
313                            ksio->reads++;
314                            ksio->nread += zio->io_size;
315                    } else if (zio->io_type == ZIO_TYPE_WRITE) {
316                            ksio->writes++;
317                            ksio->nwritten += zio->io_size;
318                    }
319            }
320            mutex_exit(&spa->spa_iokstat_lock);
321 }

323 static void
324 vdev_queue_agg_io_done(zio_t *aio)
325 {
```

```
326            if (aio->io_type == ZIO_TYPE_READ) {
327                    zio_t *pio;
328                    while ((pio = zio_walk_parents(aio)) != NULL) {
329                            bcopy((char *)aio->io_data + (pio->io_offset -
330                                aio->io_offset), pio->io_data, pio->io_size);
331                    }
332            }

334            zio_buf_free(aio->io_data, aio->io_size);
335 }

337 static int
338 vdev_queue_class_min_active(zio_priority_t p)
339 {
340            switch (p) {
341            case ZIO_PRIORITY_SYNC_READ:
342                    return (zfs_vdev_sync_read_min_active);
343            case ZIO_PRIORITY_SYNC_WRITE:
344                    return (zfs_vdev_sync_write_min_active);
345            case ZIO_PRIORITY_ASYNC_READ:
346                    return (zfs_vdev_async_read_min_active);
347            case ZIO_PRIORITY_ASYNC_WRITE:
348                    return (zfs_vdev_async_write_min_active);
349            case ZIO_PRIORITY_SCRUB:
350                    return (zfs_vdev_scrub_min_active);
351            default:
352                    panic("invalid priority %u", p);
353                    return (0);
354            }
355 }

357 static int
358 vdev_queue_max_async_writes(uint64_t dirty)
359 {
360            int writes;
361            uint64_t min_bytes = zfs_dirty_data_max *
362                zfs_vdev_async_write_active_min_dirty_percent / 100;
363            uint64_t max_bytes = zfs_dirty_data_max *
364                zfs_vdev_async_write_active_max_dirty_percent / 100;

366            if (dirty < min_bytes)
367                    return (zfs_vdev_async_write_min_active);
368            if (dirty > max_bytes)
369                    return (zfs_vdev_async_write_max_active);

371            /*
372             * linear interpolation:
373             * slope = (max_writes - min_writes) / (max_bytes - min_bytes)
374             * move right by min_bytes
375             * move up by min_writes
376             */
377            writes = (dirty - min_bytes) *
378                (zfs_vdev_async_write_max_active -
379                zfs_vdev_async_write_min_active) /
380                (max_bytes - min_bytes) +
381                zfs_vdev_async_write_min_active;
382            ASSERT3U(writes, >=, zfs_vdev_async_write_min_active);
383            ASSERT3U(writes, <=, zfs_vdev_async_write_max_active);
384            return (writes);
385 }

387 static int
388 vdev_queue_class_max_active(spa_t *spa, zio_priority_t p)
389 {
390            switch (p) {
391            case ZIO_PRIORITY_SYNC_READ:
```

```
392                         return (zfs_vdev_sync_read_max_active);
393                 case ZIO_PRIORITY_SYNC_WRITE:
394                         return (zfs_vdev_sync_write_max_active);
395                 case ZIO_PRIORITY_ASYNC_READ:
396                         return (zfs_vdev_async_read_max_active);
397                 case ZIO_PRIORITY_ASYNC_WRITE:
398                         return (vdev_queue_max_async_writes(
399                             spa->spa_dsl_pool->dp_dirty_total));
400                 case ZIO_PRIORITY_SCRUB:
401                         return (zfs_vdev_scrub_max_active);
402                 default:
403                         panic("invalid priority %u", p);
404                         return (0);
405                 }
406 }

408 /*
409  * Return the i/o class to issue from, or ZIO_PRIORITY_MAX_QUEUEABLE if
410  * there is no eligible class.
411  */
412 static zio_priority_t
413 vdev_queue_class_to_issue(vdev_queue_t *vq)
414 {
415         spa_t *spa = vq->vq_vdev->vdev_spa;
416         zio_priority_t p;

418         if (avl_numnodes(&vq->vq_active_tree) >= zfs_vdev_max_active)
419                 return (ZIO_PRIORITY_NUM_QUEUEABLE);

421         /* find a queue that has not reached its minimum # outstanding i/os */
422         for (p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++) {
423                 if (avl_numnodes(&vq->vq_class[p].vqc_queued_tree) > 0 &&
424                     vq->vq_class[p].vqc_active <
425                     vdev_queue_class_min_active(p))
426                         return (p);
427         }

429         /*
430          * If we haven't found a queue, look for one that hasn't reached its
431          * maximum # outstanding i/os.
432          */
433         for (p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++) {
434                 if (avl_numnodes(&vq->vq_class[p].vqc_queued_tree) > 0 &&
435                     vq->vq_class[p].vqc_active <
436                     vdev_queue_class_max_active(spa, p))
437                         return (p);
438         }

440         /* No eligible queued i/os */
441         return (ZIO_PRIORITY_NUM_QUEUEABLE);
442 }

444 /*
445  * Compute the range spanned by two i/os, which is the endpoint of the last
446  * (lio->io_offset + lio->io_size) minus start of the first (fio->io_offset).
447  * Conveniently, the gap between fio and lio is given by -IO_SPAN(lio, fio);
448  * thus fio and lio are adjacent if and only if IO_SPAN(lio, fio) == 0.
449  */
450 #define IO_SPAN(fio, lio) ((lio)->io_offset + (lio)->io_size - (fio)->io_offset)
451 #define IO_GAP(fio, lio) (-IO_SPAN(lio, fio))

453 static zio_t *
454 vdev_queue_aggregate(vdev_queue_t *vq, zio_t *zio)
455 {
456         zio_t *first, *last, *aio, *dio, *mandatory, *nio;
457         uint64_t maxgap = 0;
```

```
458         uint64_t size;
459         boolean_t stretch = B_FALSE;
460         vdev_queue_class_t *vqc = &vq->vq_class[zio->io_priority];
461         avl_tree_t *t = &vqc->vqc_queued_tree;
462         enum zio_flag flags = zio->io_flags & ZIO_FLAG_AGG_INHERIT;

464         if (zio->io_flags & ZIO_FLAG_DONT_AGGREGATE)
465                 return (NULL);

467         /*
468          * The synchronous i/o queues are not sorted by LBA, so we can't
469          * find adjacent i/os.  These i/os tend to not be tightly clustered,
470          * or too large to aggregate, so this has little impact on performance.
471          */
472         if (zio->io_priority == ZIO_PRIORITY_SYNC_READ ||
473             zio->io_priority == ZIO_PRIORITY_SYNC_WRITE)
474                 return (NULL);

476         first = last = zio;

478         if (zio->io_type == ZIO_TYPE_READ)
479                 maxgap = zfs_vdev_read_gap_limit;

481         /*
482          * We can aggregate I/Os that are sufficiently adjacent and of
483          * the same flavor, as expressed by the AGG_INHERIT flags.
484          * The latter requirement is necessary so that certain
485          * attributes of the I/O, such as whether it's a normal I/O
486          * or a scrub/resilver, can be preserved in the aggregate.
487          * We can include optional I/Os, but don't allow them
488          * to begin a range as they add no benefit in that situation.
489          */

491         /*
492          * We keep track of the last non-optional I/O.
493          */
494         mandatory = (first->io_flags & ZIO_FLAG_OPTIONAL) ? NULL : first;

496         /*
497          * Walk backwards through sufficiently contiguous I/Os
498          * recording the last non-option I/O.
499          */
500         while ((dio = AVL_PREV(t, first)) != NULL &&
501             (dio->io_flags & ZIO_FLAG_AGG_INHERIT) == flags &&
502             IO_SPAN(dio, last) <= zfs_vdev_aggregation_limit &&
503             IO_GAP(dio, first) <= maxgap) {
504                 first = dio;
505                 if (mandatory == NULL && !(first->io_flags & ZIO_FLAG_OPTIONAL))
506                         mandatory = first;
507         }

509         /*
510          * Skip any initial optional I/Os.
511          */
512         while ((first->io_flags & ZIO_FLAG_OPTIONAL) && first != last) {
513                 first = AVL_NEXT(t, first);
514                 ASSERT(first != NULL);
515         }

517         /*
518          * Walk forward through sufficiently contiguous I/Os.
519          */
520         while ((dio = AVL_NEXT(t, last)) != NULL &&
521             (dio->io_flags & ZIO_FLAG_AGG_INHERIT) == flags &&
522             IO_SPAN(first, dio) <= zfs_vdev_aggregation_limit &&
523             IO_GAP(last, dio) <= maxgap) {
```

```
524                         last = dio;
525                         if (!(last->io_flags & ZIO_FLAG_OPTIONAL))
526                                 mandatory = last;
527                 }

529                 /*
530                  * Now that we've established the range of the I/O aggregation
531                  * we must decide what to do with trailing optional I/Os.
532                  * For reads, there's nothing to do. While we are unable to
533                  * aggregate further, it's possible that a trailing optional
534                  * I/O would allow the underlying device to aggregate with
535                  * subsequent I/Os. We must therefore determine if the next
536                  * non-optional I/O is close enough to make aggregation
537                  * worthwhile.
538                  */
539                 if (zio->io_type == ZIO_TYPE_WRITE && mandatory != NULL) {
540                         zio_t *nio = last;
541                         while ((dio = AVL_NEXT(t, nio)) != NULL &&
542                             IO_GAP(nio, dio) == 0 &&
543                             IO_GAP(mandatory, dio) <= zfs_vdev_write_gap_limit) {
544                                 nio = dio;
545                                 if (!(nio->io_flags & ZIO_FLAG_OPTIONAL)) {
546                                         stretch = B_TRUE;
547                                         break;
548                                 }
549                         }
550                 }

552                 if (stretch) {
553                         /* This may be a no-op. */
554                         dio = AVL_NEXT(t, last);
555                         dio->io_flags &= ~ZIO_FLAG_OPTIONAL;
556                 } else {
557                         while (last != mandatory && last != first) {
558                                 ASSERT(last->io_flags & ZIO_FLAG_OPTIONAL);
559                                 last = AVL_PREV(t, last);
560                                 ASSERT(last != NULL);
561                         }
562                 }

564                 if (first == last)
565                         return (NULL);

567                 size = IO_SPAN(first, last);
568                 ASSERT3U(size, <=, zfs_vdev_aggregation_limit);

570                 aio = zio_vdev_delegated_io(first->io_vd, first->io_offset,
571                     zio_buf_alloc(size), size, first->io_type, zio->io_priority,
572                     flags | ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_QUEUE,
573                     vdev_queue_agg_io_done, NULL);
574                 aio->io_timestamp = first->io_timestamp;

576                 nio = first;
577                 do {
578                         dio = nio;
579                         nio = AVL_NEXT(t, dio);
580                         ASSERT3U(dio->io_type, ==, aio->io_type);

582                         if (dio->io_flags & ZIO_FLAG_NODATA) {
583                                 ASSERT3U(dio->io_type, ==, ZIO_TYPE_WRITE);
584                                 bzero((char *)aio->io_data + (dio->io_offset -
585                                     aio->io_offset), dio->io_size);
586                         } else if (dio->io_type == ZIO_TYPE_WRITE) {
587                                 bcopy(dio->io_data, (char *)aio->io_data +
588                                     (dio->io_offset - aio->io_offset),
589                                     dio->io_size);
```

```
590                         }

592                         zio_add_child(dio, aio);
593                         vdev_queue_io_remove(vq, dio);
594                         zio_vdev_io_bypass(dio);
595                         zio_execute(dio);
596                 } while (dio != last);

598                 return (aio);
599 }

601 static zio_t *
602 vdev_queue_io_to_issue(vdev_queue_t *vq)
603 {
604         zio_t *zio, *aio;
605         zio_priority_t p;
606         avl_index_t idx;
607         vdev_queue_class_t *vqc;
608         zio_t search;

610 again:
611         ASSERT(MUTEX_HELD(&vq->vq_lock));

613         p = vdev_queue_class_to_issue(vq);

615         if (p == ZIO_PRIORITY_NUM_QUEUEABLE) {
616                 /* No eligible queued i/os */
617                 return (NULL);
618         }

620         /*
621          * For LBA-ordered queues (async / scrub), issue the i/o which follows
622          * the most recently issued i/o in LBA (offset) order.
623          *
624          * For FIFO queues (sync), issue the i/o with the lowest timestamp.
625          */
626         vqc = &vq->vq_class[p];
627         search.io_timestamp = 0;
628         search.io_offset = vq->vq_last_offset + 1;
629         VERIFY3P(avl_find(&vqc->vqc_queued_tree, &search, &idx), ==, NULL);
630         zio = avl_nearest(&vqc->vqc_queued_tree, idx, AVL_AFTER);
631         if (zio == NULL)
632                 zio = avl_first(&vqc->vqc_queued_tree);
633         ASSERT3U(zio->io_priority, ==, p);

635         aio = vdev_queue_aggregate(vq, zio);
636         if (aio != NULL)
637                 zio = aio;
638         else
639                 vdev_queue_io_remove(vq, zio);

641         /*
642          * If the I/O is or was optional and therefore has no data, we need to
643          * simply discard it. We need to drop the vdev queue's lock to avoid a
644          * deadlock that we could encounter since this I/O will complete
645          * immediately.
646          */
647         if (zio->io_flags & ZIO_FLAG_NODATA) {
648                 mutex_exit(&vq->vq_lock);
649                 zio_vdev_io_bypass(zio);
650                 zio_execute(zio);
651                 mutex_enter(&vq->vq_lock);
652                 goto again;
653         }

655         vdev_queue_pending_add(vq, zio);
```

```
656              vq->vq_last_offset = zio->io_offset;

658              return (zio);
659 }

661 zio_t *
662 vdev_queue_io(zio_t *zio)
663 {
664              vdev_queue_t *vq = &zio->io_vd->vdev_queue;
665              zio_t *nio;

667              if (zio->io_flags & ZIO_FLAG_DONT_QUEUE)
668                      return (zio);

670              /*
671               * Children i/os inherent their parent's priority, which might
672               * not match the child's i/o type.  Fix it up here.
673               */
674              if (zio->io_type == ZIO_TYPE_READ) {
675                      if (zio->io_priority != ZIO_PRIORITY_SYNC_READ &&
676                          zio->io_priority != ZIO_PRIORITY_ASYNC_READ &&
677                          zio->io_priority != ZIO_PRIORITY_SCRUB)
678                              zio->io_priority = ZIO_PRIORITY_ASYNC_READ;
679              } else {
680                      ASSERT(zio->io_type == ZIO_TYPE_WRITE);
681                      if (zio->io_priority != ZIO_PRIORITY_SYNC_WRITE &&
682                          zio->io_priority != ZIO_PRIORITY_ASYNC_WRITE)
683                              zio->io_priority = ZIO_PRIORITY_ASYNC_WRITE;
684              }

686              zio->io_flags |= ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_QUEUE;

688              mutex_enter(&vq->vq_lock);
689              zio->io_timestamp = gethrtime();
690              vdev_queue_io_add(vq, zio);
691              nio = vdev_queue_io_to_issue(vq);
692              mutex_exit(&vq->vq_lock);

694              if (nio == NULL)
695                      return (NULL);

697              if (nio->io_done == vdev_queue_agg_io_done) {
698                      zio_nowait(nio);
699                      return (NULL);
700              }

702              return (nio);
703 }

705 void
706 vdev_queue_io_done(zio_t *zio)
707 {
708              vdev_queue_t *vq = &zio->io_vd->vdev_queue;
709              zio_t *nio;

711              if (zio_injection_enabled)
712                      delay(SEC_TO_TICK(zio_handle_io_delay(zio)));

714              mutex_enter(&vq->vq_lock);

716              vdev_queue_pending_remove(vq, zio);

718              vq->vq_io_complete_ts = gethrtime();

720              while ((nio = vdev_queue_io_to_issue(vq)) != NULL) {
721                      mutex_exit(&vq->vq_lock);
```

```
722                      if (nio->io_done == vdev_queue_agg_io_done) {
723                              zio_nowait(nio);
724                      } else {
725                              zio_vdev_io_reissue(nio);
726                              zio_execute(nio);
727                      }
728                      mutex_enter(&vq->vq_lock);
729              }

731              mutex_exit(&vq->vq_lock);
732 }

734 /*
735  * As these three methods are only used for load calculations we're not
736  * concerned if we get an incorrect value on 32bit platforms due to lack of
737  * vq_lock mutex use here, instead we prefer to keep it lock free for
738  * performance.
739  */
740 int
741 vdev_queue_length(vdev_t *vd)
742 {
743              vdev_queue_t *vq = &vd->vdev_queue;
744              int len = 0;

746              for (zio_priority_t p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++)
747                      len += avl_numnodes(&vq->vq_class[p].vqc_queued_tree);

749              return (len);
750 }

752 uint64_t
753 vdev_queue_last_queued_offset(vdev_t *vd)
754 {
755              return (vd->vdev_queue.vq_last_queued_offset);
756 }

758 void
759 vdev_queue_register_last_queued_offset(vdev_t *vd, zio_t *zio)
760 {
761              vd->vdev_queue.vq_last_queued_offset = zio->io_offset + zio->io_size;
762 }
763 #endif /* ! codereview */
```