```
**********************************************************
   77013 Thu Aug 22 16:14:57 2013
new/usr/src/cmd/mdb/common/modules/zfs/zfs.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 267 /* ARGSUSED */
 268 static int
 269 zfs_params(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
 270 {
 271         /*
 272          * This table can be approximately generated by running:
 273          * egrep "^[a-z0-9_]+ [a-z0-9_]+( =.*)?;" *.c | cut -d ' ' -f 2
 274          */
 275         static const char *params[] = {
 276                 "arc_reduce_dnlc_percent",
 277                 "arc_lotsfree_percent",
 278                 "zfs_dirty_data_max",
 279                 "zfs_dirty_data_sync",
 280                 "zfs_delay_max_ns",
 281                 "zfs_delay_min_dirty_percent",
 282                 "zfs_delay_scale",
 283                 "zfs_vdev_max_active",
 284                 "zfs_vdev_sync_read_min_active",
 285                 "zfs_vdev_sync_read_max_active",
 286                 "zfs_vdev_sync_write_min_active",
 287                 "zfs_vdev_sync_write_max_active",
 288                 "zfs_vdev_async_read_min_active",
 289                 "zfs_vdev_async_read_max_active",
 290                 "zfs_vdev_async_write_min_active",
 291                 "zfs_vdev_async_write_max_active",
 292                 "zfs_vdev_scrub_min_active",
 293                 "zfs_vdev_scrub_max_active",
 294                 "zfs_vdev_async_write_active_min_dirty_percent",
 295                 "zfs_vdev_async_write_active_max_dirty_percent",
 296                 "spa_asize_inflation",
 297                 "zfs_arc_max",
 298                 "zfs_arc_min",
 299                 "arc_shrink_shift",
 300                 "zfs_mdcomp_disable",
 301                 "zfs_prefetch_disable",
 302                 "zfetch_max_streams",
 303                 "zfetch_min_sec_reap",
 304                 "zfetch_block_cap",
 305                 "zfetch_array_rd_sz",
 306                 "zfs_default_bs",
 307                 "zfs_default_ibs",
 308                 "metaslab_aliquot",
 309                 "reference_tracking_enable",
 310                 "reference_history",
 311                 "spa_max_replication_override",
 312                 "spa_mode_global",
 313                 "zfs_flags",
 294                 "zfs_txg_synctime_ms",
 314                 "zfs_txg_timeout",
 296                 "zfs_write_limit_min",
 297                 "zfs_write_limit_max",
 298                 "zfs_write_limit_shift",
 299                 "zfs_write_limit_override",
 300                 "zfs_no_write_throttle",
 315                 "zfs_vdev_cache_max",
 316                 "zfs_vdev_cache_size",
```



```
 317                 "zfs_vdev_cache_bshift",
 318                 "vdev_mirror_shift",
 305                 "zfs_vdev_max_pending",
 306                 "zfs_vdev_min_pending",
 319                 "zfs_scrub_limit",
 320                 "zfs_no_scrub_io",
 321                 "zfs_no_scrub_prefetch",
 310                 "zfs_vdev_time_shift",
 311                 "zfs_vdev_ramp_rate",
 322                 "zfs_vdev_aggregation_limit",
 323                 "fzap_default_block_shift",
 324                 "zfs_immediate_write_sz",
 325                 "zfs_read_chunk_size",
 326                 "zfs_nocacheflush",
 327                 "zil_replay_disable",
 328                 "metaslab_gang_bang",
 329                 "metaslab_df_alloc_threshold",
 330                 "metaslab_df_free_pct",
 331                 "zio_injection_enabled",
 332                 "zvol_immediate_write_sz",
 333         };

 335         for (int i = 0; i < sizeof (params) / sizeof (params[0]); i++) {
 336                 int sz;
 337                 uint64_t val64;
 338                 uint32_t *val32p = (uint32_t *)&val64;

 340                 sz = mdb_readvar(&val64, params[i]);
 341                 if (sz == 4) {
 342                         mdb_printf("%s = 0x%x\n", params[i], *val32p);
 343                 } else if (sz == 8) {
 344                         mdb_printf("%s = 0x%llx\n", params[i], val64);
 345                 } else {
 346                         mdb_warn("variable %s not found", params[i]);
 347                 }
 348         }

 350         return (DCMD_OK);
 351 }
______unchanged_portion_omitted_

1831 /* ARGSUSED */
1832 static int
1833 zio_child_cb(uintptr_t addr, const void *unknown, void *arg)
1834 {
1835         zio_link_t zl;
1836         uintptr_t ziop;
1837         zio_print_args_t *zpa = arg;

1839         if (mdb_vread(&zl, sizeof (zl), addr) == -1) {
1840                 mdb_warn("failed to read zio_link_t at %p", addr);
1841                 return (WALK_ERR);
1842         }

1844         if (zpa->zpa_type == ZIO_WALK_PARENT)
1845                 ziop = (uintptr_t)zl.zl_parent;
1846         else
1847                 ziop = (uintptr_t)zl.zl_child;

1849         return (zio_print_cb(ziop, zpa));
1839         return (zio_print_cb(ziop, arg));
1850 }
______unchanged_portion_omitted_
```

```
**********************************************************
  160678 Thu Aug 22 16:14:59 2013
new/usr/src/cmd/ztest/ztest.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 185 extern uint64_t metaslab_gang_bang;
 186 extern uint64_t metaslab_df_alloc_threshold;
 187 extern uint64_t zfs_deadman_synctime_ms;
 187 extern uint64_t zfs_deadman_synctime;

 189 static ztest_shared_opts_t *ztest_shared_opts;
 190 static ztest_shared_opts_t ztest_opts;

 192 typedef struct ztest_shared_ds {
 193         uint64_t        zd_seq;
 194 } ztest_shared_ds_t;
______unchanged_portion_omitted_

5320 static void *
5321 ztest_deadman_thread(void *arg)
5322 {
5323         ztest_shared_t *zs = arg;
5324         spa_t *spa = ztest_spa;
5325         hrtime_t delta, total = 0;

5327         for (;;) {
5328                 delta = zs->zs_thread_stop - zs->zs_thread_start +
5329                     MSEC2NSEC(zfs_deadman_synctime_ms);
5328                 delta = (zs->zs_thread_stop - zs->zs_thread_start) /
5329                     NANOSEC + zfs_deadman_synctime;

5331                 (void) poll(NULL, 0, (int)NSEC2MSEC(delta));
5331                 (void) poll(NULL, 0, (int)(1000 * delta));

5333                 /*
5334                  * If the pool is suspended then fail immediately. Otherwise,
5335                  * check to see if the pool is making any progress. If
5336                  * vdev_deadman() discovers that there hasn't been any recent
5337                  * I/Os then it will end up aborting the tests.
5338                  */
5339                 if (spa_suspended(spa)) {
5340                         fatal(0, "aborting test after %llu seconds because "
5341                             "pool has transitioned to a suspended state.",
5342                             zfs_deadman_synctime_ms / 1000);
5342                             zfs_deadman_synctime);
5343                         return (NULL);
5344                 }
5345                 vdev_deadman(spa->spa_root_vdev);

5347                 total += zfs_deadman_synctime_ms/1000;
5347                 total += zfs_deadman_synctime;
5348                 (void) printf("ztest has been running for %lld seconds\n",
5349                     total);
5350         }
5351 }
______unchanged_portion_omitted_

6056 int
6057 main(int argc, char **argv)
6058 {
6059         int kills = 0;
```



```
6060         int iters = 0;
6061         int older = 0;
6062         int newer = 0;
6063         ztest_shared_t *zs;
6064         ztest_info_t *zi;
6065         ztest_shared_callstate_t *zc;
6066         char timebuf[100];
6067         char numbuf[6];
6068         spa_t *spa;
6069         char *cmd;
6070         boolean_t hasalt;
6071         char *fd_data_str = getenv("ZTEST_FD_DATA");

6073         (void) setvbuf(stdout, NULL, _IOLBF, 0);

6075         dprintf_setup(&argc, argv);
6076         zfs_deadman_synctime_ms = 300000;
6076         zfs_deadman_synctime = 300;

6078         ztest_fd_rand = open("/dev/urandom", O_RDONLY);
6079         ASSERT3S(ztest_fd_rand, >=, 0);

6081         if (!fd_data_str) {
6082                 process_options(argc, argv);

6084                 setup_data_fd();
6085                 setup_hdr();
6086                 setup_data();
6087                 bcopy(&ztest_opts, ztest_shared_opts,
6088                     sizeof (*ztest_shared_opts));
6089         } else {
6090                 ztest_fd_data = atoi(fd_data_str);
6091                 setup_data();
6092                 bcopy(ztest_shared_opts, &ztest_opts, sizeof (ztest_opts));
6093         }
6094         ASSERT3U(ztest_opts.zo_datasets, ==, ztest_shared_hdr->zh_ds_count);

6096         /* Override location of zpool.cache */
6097         VERIFY3U(asprintf((char **)&spa_config_path, "%s/zpool.cache",
6098             ztest_opts.zo_dir), !=, -1);

6100         ztest_ds = umem_alloc(ztest_opts.zo_datasets * sizeof (ztest_ds_t),
6101             UMEM_NOFAIL);
6102         zs = ztest_shared;

6104         if (fd_data_str) {
6105                 metaslab_gang_bang = ztest_opts.zo_metaslab_gang_bang;
6106                 metaslab_df_alloc_threshold =
6107                     zs->zs_metaslab_df_alloc_threshold;

6109                 if (zs->zs_do_init)
6110                         ztest_run_init();
6111                 else
6112                         ztest_run(zs);
6113                 exit(0);
6114         }

6116         hasalt = (strlen(ztest_opts.zo_alt_ztest) != 0);

6118         if (ztest_opts.zo_verbose >= 1) {
6119                 (void) printf("%llu vdevs, %d datasets, %d threads,"
6120                     " %llu seconds...\n",
6121                     (u_longlong_t)ztest_opts.zo_vdevs,
6122                     ztest_opts.zo_datasets,
6123                     ztest_opts.zo_threads,
6124                     (u_longlong_t)ztest_opts.zo_time);
```

```
6125         }
6126
6127         cmd = umem_alloc(MAXNAMELEN, UMEM_NOFAIL);
6128         (void) strlcpy(cmd, getexecname(), MAXNAMELEN);
6129
6130         zs->zs_do_init = B_TRUE;
6131         if (strlen(ztest_opts.zo_alt_ztest) != 0) {
6132                 if (ztest_opts.zo_verbose >= 1) {
6133                         (void) printf("Executing older ztest for "
6134                             "initialization: %s\n", ztest_opts.zo_alt_ztest);
6135                 }
6136                 VERIFY(!exec_child(ztest_opts.zo_alt_ztest,
6137                     ztest_opts.zo_alt_libpath, B_FALSE, NULL));
6138         } else {
6139                 VERIFY(!exec_child(NULL, NULL, B_FALSE, NULL));
6140         }
6141         zs->zs_do_init = B_FALSE;
6142
6143         zs->zs_proc_start = gethrtime();
6144         zs->zs_proc_stop = zs->zs_proc_start + ztest_opts.zo_time * NANOSEC;
6145
6146         for (int f = 0; f < ZTEST_FUNCS; f++) {
6147                 zi = &ztest_info[f];
6148                 zc = ZTEST_GET_SHARED_CALLSTATE(f);
6149                 if (zs->zs_proc_start + zi->zi_interval[0] > zs->zs_proc_stop)
6150                         zc->zc_next = UINT64_MAX;
6151                 else
6152                         zc->zc_next = zs->zs_proc_start +
6153                             ztest_random(2 * zi->zi_interval[0] + 1);
6154         }
6155
6156         /*
6157          * Run the tests in a loop.  These tests include fault injection
6158          * to verify that self-healing data works, and forced crashes
6159          * to verify that we never lose on-disk consistency.
6160          */
6161         while (gethrtime() < zs->zs_proc_stop) {
6162                 int status;
6163                 boolean_t killed;
6164
6165                 /*
6166                  * Initialize the workload counters for each function.
6167                  */
6168                 for (int f = 0; f < ZTEST_FUNCS; f++) {
6169                         zc = ZTEST_GET_SHARED_CALLSTATE(f);
6170                         zc->zc_count = 0;
6171                         zc->zc_time = 0;
6172                 }
6173
6174                 /* Set the allocation switch size */
6175                 zs->zs_metaslab_df_alloc_threshold =
6176                     ztest_random(zs->zs_metaslab_sz / 4) + 1;
6177
6178                 if (!hasalt || ztest_random(2) == 0) {
6179                         if (hasalt && ztest_opts.zo_verbose >= 1) {
6180                                 (void) printf("Executing newer ztest: %s\n",
6181                                     cmd);
6182                         }
6183                         newer++;
6184                         killed = exec_child(cmd, NULL, B_TRUE, &status);
6185                 } else {
6186                         if (hasalt && ztest_opts.zo_verbose >= 1) {
6187                                 (void) printf("Executing older ztest: %s\n",
6188                                     ztest_opts.zo_alt_ztest);
6189                         }
6190                         older++;
```



```
6191                         killed = exec_child(ztest_opts.zo_alt_ztest,
6192                             ztest_opts.zo_alt_libpath, B_TRUE, &status);
6193                 }
6194
6195                 if (killed)
6196                         kills++;
6197                 iters++;
6198
6199                 if (ztest_opts.zo_verbose >= 1) {
6200                         hrtime_t now = gethrtime();
6201
6202                         now = MIN(now, zs->zs_proc_stop);
6203                         print_time(zs->zs_proc_stop - now, timebuf);
6204                         nicenum(zs->zs_space, numbuf);
6205
6206                         (void) printf("Pass %3d, %8s, %3llu ENOSPC, "
6207                             "%4.1f%% of %5s used, %3.0f%% done, %8s to go\n",
6208                             iters,
6209                             WIFEXITED(status) ? "Complete" : "SIGKILL",
6210                             (u_longlong_t)zs->zs_enospc_count,
6211                             100.0 * zs->zs_alloc / zs->zs_space,
6212                             numbuf,
6213                             100.0 * (now - zs->zs_proc_start) /
6214                             (ztest_opts.zo_time * NANOSEC), timebuf);
6215                 }
6216
6217                 if (ztest_opts.zo_verbose >= 2) {
6218                         (void) printf("\nWorkload summary:\n\n");
6219                         (void) printf("%7s %9s   %s\n",
6220                             "Calls", "Time", "Function");
6221                         (void) printf("%7s %9s   %s\n",
6222                             "-----", "----", "--------");
6223                         for (int f = 0; f < ZTEST_FUNCS; f++) {
6224                                 Dl_info dli;
6225
6226                                 zi = &ztest_info[f];
6227                                 zc = ZTEST_GET_SHARED_CALLSTATE(f);
6228                                 print_time(zc->zc_time, timebuf);
6229                                 (void) dladdr((void *)zi->zi_func, &dli);
6230                                 (void) printf("%7llu %9s   %s\n",
6231                                     (u_longlong_t)zc->zc_count, timebuf,
6232                                     dli.dli_sname);
6233                         }
6234                         (void) printf("\n");
6235                 }
6236
6237                 /*
6238                  * It's possible that we killed a child during a rename test,
6239                  * in which case we'll have a 'ztest_tmp' pool lying around
6240                  * instead of 'ztest'.  Do a blind rename in case this happened.
6241                  */
6242                 kernel_init(FREAD);
6243                 if (spa_open(ztest_opts.zo_pool, &spa, FTAG) == 0) {
6244                         spa_close(spa, FTAG);
6245                 } else {
6246                         char tmpname[MAXNAMELEN];
6247                         kernel_fini();
6248                         kernel_init(FREAD | FWRITE);
6249                         (void) snprintf(tmpname, sizeof (tmpname), "%s_tmp",
6250                             ztest_opts.zo_pool);
6251                         (void) spa_rename(tmpname, ztest_opts.zo_pool);
6252                 }
6253                 kernel_fini();
6254
6255                 ztest_run_zdb(ztest_opts.zo_pool);
6256         }
```

```
6258         if (ztest_opts.zo_verbose >= 1) {
6259                 if (hasalt) {
6260                         (void) printf("%d runs of older ztest: %s\n", older,
6261                             ztest_opts.zo_alt_ztest);
6262                         (void) printf("%d runs of newer ztest: %s\n", newer,
6263                             cmd);
6264                 }
6265                 (void) printf("%d killed, %d completed, %.0f%% kill rate\n",
6266                     kills, iters - kills, (100.0 * kills) / MAX(1, iters));
6267         }

6269         umem_free(cmd, MAXNAMELEN);

6271         return (0);
6272 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1940 Thu Aug 22 16:15:01 2013
new/usr/src/lib/libzpool/common/llib-lzpool
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/* LINTLIBRARY */
/* PROTOLIB1 */

#include <sys/zfs_context.h>
#include <sys/list.h>
#include <sys/list_impl.h>
#include <sys/sysmacros.h>
#include <sys/debug.h>
#include <sys/dmu_traverse.h>
#include <sys/dnode.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_dataset.h>
#include <sys/spa.h>
#include <sys/spa_impl.h>
#include <sys/space_map.h>
#include <sys/vdev.h>
#include <sys/vdev_impl.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/zio_compress.h>
#include <sys/zil.h>
#include <sys/bplist.h>
#include <sys/zfs_znode.h>
#include <sys/arc.h>
#include <sys/dbuf.h>
#include <sys/zio_checksum.h>
#include <sys/ddt.h>
#include <sys/sa.h>
```



```
#include <sys/zfs_sa.h>
#include <sys/zfeature.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_destroy.h>
#include <sys/dsl_userhold.h>

extern uint64_t metaslab_gang_bang;
extern uint64_t metaslab_df_alloc_threshold;
extern boolean_t zfeature_checks_disable;
extern uint64_t zfs_deadman_synctime_ms;
extern uint64_t zfs_deadman_synctime;
```

```
**********************************************************
   18036 Thu Aug 22 16:15:02 2013
new/usr/src/lib/libzpool/common/sys/zfs_context.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_ZFS_CONTEXT_H
#define _SYS_ZFS_CONTEXT_H

#ifdef __cplusplus
extern "C" {
#endif

#define _SYS_MUTEX_H
#define _SYS_RWLOCK_H
#define _SYS_CONDVAR_H
#define _SYS_SYSTM_H
#define _SYS_T_LOCK_H
#define _SYS_VNODE_H
#define _SYS_VFS_H
#define _SYS_SUNDDI_H
#define _SYS_CALLB_H

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdarg.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <synch.h>
#include <thread.h>
#include <assert.h>
#include <alloca.h>
#include <umem.h>
```



```
#include <limits.h>
#include <atomic.h>
#include <dirent.h>
#include <time.h>
#include <procfs.h>
#include <pthread.h>
#include <sys/debug.h>
#include <libsysevent.h>
#include <sys/note.h>
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/sysmacros.h>
#include <sys/bitmap.h>
#include <sys/resource.h>
#include <sys/byteorder.h>
#include <sys/list.h>
#include <sys/uio.h>
#include <sys/zfs_debug.h>
#include <sys/sdt.h>
#include <sys/kstat.h>
#include <sys/u8_textprep.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dev.h>
#include <sys/sunddi.h>
#include <sys/debug.h>
#include "zfs.h"

/*
 * Debugging
 */

/*
 * Note that we are not using the debugging levels.
 */

#define CE_CONT         0       /* continuation         */
#define CE_NOTE         1       /* notice               */
#define CE_WARN         2       /* warning              */
#define CE_PANIC        3       /* panic                */
#define CE_IGNORE       4       /* print nothing        */

/*
 * ZFS debugging
 */

#ifdef ZFS_DEBUG
extern void dprintf_setup(int *argc, char **argv);
#endif /* ZFS_DEBUG */

extern void cmn_err(int, const char *, ...);
extern void vcmn_err(int, const char *, __va_list);
extern void panic(const char *, ...);
extern void vpanic(const char *, __va_list);

#define fm_panic        panic

extern int aok;

/*
 * DTrace SDT probes have different signatures in userland than they do in
 * kernel.  If they're being used in kernel code, re-define them out of
 * existence for their counterparts in libzpool.
 */

#ifdef DTRACE_PROBE
#undef  DTRACE_PROBE
```

```
125 #endif  /* DTRACE_PROBE */
126 #define DTRACE_PROBE(a) \
127         ZFS_PROBE0(#a)

129 #ifdef DTRACE_PROBE1
130 #undef  DTRACE_PROBE1
131 #endif  /* DTRACE_PROBE1 */
132 #define DTRACE_PROBE1(a, b, c) \
133         ZFS_PROBE1(#a, (unsigned long)c)

135 #ifdef DTRACE_PROBE2
136 #undef  DTRACE_PROBE2
137 #endif  /* DTRACE_PROBE2 */
138 #define DTRACE_PROBE2(a, b, c, d, e) \
139         ZFS_PROBE2(#a, (unsigned long)c, (unsigned long)e)

141 #ifdef DTRACE_PROBE3
142 #undef  DTRACE_PROBE3
143 #endif  /* DTRACE_PROBE3 */
144 #define DTRACE_PROBE3(a, b, c, d, e, f, g) \
145         ZFS_PROBE3(#a, (unsigned long)c, (unsigned long)e, (unsigned long)g)

147 #ifdef DTRACE_PROBE4
148 #undef  DTRACE_PROBE4
149 #endif  /* DTRACE_PROBE4 */
150 #define DTRACE_PROBE4(a, b, c, d, e, f, g, h, i) \
151         ZFS_PROBE4(#a, (unsigned long)c, (unsigned long)e, (unsigned long)g, \
152         (unsigned long)i)

154 /*
155  * We use the comma operator so that this macro can be used without much
156  * additional code.  For example, "return (EINVAL);" becomes
157  * "return (SET_ERROR(EINVAL));".  Note that the argument will be evaluated
158  * twice, so it should not have side effects (e.g. something like:
159  * "return (SET_ERROR(log_error(EINVAL, info)));" would log the error twice).
160  */
161 #define SET_ERROR(err) (ZFS_SET_ERROR(err), err)

163 /*
164  * Threads
165  */
166 #define curthread       ((void *)(uintptr_t)thr_self())

168 #define kpreempt(x)     yield()

170 typedef struct kthread kthread_t;

172 #define thread_create(stk, stksize, func, arg, len, pp, state, pri)     \
173         zk_thread_create(func, arg)
174 #define thread_exit() thr_exit(NULL)
175 #define thread_join(t)  panic("libzpool cannot join threads")

177 #define newproc(f, a, cid, pri, ctp, pid)       (ENOSYS)

179 /* in libzpool, p0 exists only to have its address taken */
180 struct proc {
181         uintptr_t       this_is_never_used_dont_dereference_it;
182 };
```

*unchanged_portion_omitted_*

```
**********************************************************
  147634 Thu Aug 22 16:15:03 2013
new/usr/src/uts/common/fs/zfs/arc.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

/*
 * DVA-based Adjustable Replacement Cache
 *
 * While much of the theory of operation used here is
 * based on the self-tuning, low overhead replacement cache
 * presented by Megiddo and Modha at FAST 2003, there are some
 * significant differences:
 *
 * 1. The Megiddo and Modha model assumes any page is evictable.
 * Pages in its cache cannot be "locked" into memory.  This makes
 * the eviction algorithm simple: evict the last page in the list.
 * This also make the performance characteristics easy to reason
 * about.  Our cache is not so simple.  At any given moment, some
 * subset of the blocks in the cache are un-evictable because we
 * have handed out a reference to them.  Blocks are only evictable
 * when there are no external references active.  This makes
 * eviction far more problematic:  we choose to evict the evictable
 * blocks that are the "lowest" in the list.
 *
 * There are times when it is not possible to evict the requested
 * space.  In these circumstances we are unable to adjust the cache
 * size.  To prevent the cache growing unbounded at these times we
 * implement a "cache throttle" that slows the flow of new data
 * into the cache until we can make space available.
 *
 * 2. The Megiddo and Modha model assumes a fixed cache size.
 * Pages are evicted when the cache is full and there is a cache
 * miss.  Our model has a variable sized cache.  It grows with
 * high use, but also tries to react to memory pressure from the
 * operating system: decreasing its size when system memory is
 * tight.
```



```
 *
 * 3. The Megiddo and Modha model assumes a fixed page size. All
 * elements of the cache are therefore exactly the same size.  So
 * when adjusting the cache size following a cache miss, its simply
 * a matter of choosing a single page to evict.  In our model, we
 * have variable sized cache blocks (rangeing from 512 bytes to
 * 128K bytes).  We therefore choose a set of blocks to evict to make
 * space for a cache miss that approximates as closely as possible
 * the space used by the new block.
 *
 * See also:  "ARC: A Self-Tuning, Low Overhead Replacement Cache"
 * by N. Megiddo & D. Modha, FAST 2003
 */

/*
 * The locking model:
 *
 * A new reference to a cache buffer can be obtained in two
 * ways: 1) via a hash table lookup using the DVA as a key,
 * or 2) via one of the ARC lists.  The arc_read() interface
 * uses method 1, while the internal arc algorithms for
 * adjusting the cache use method 2.  We therefore provide two
 * types of locks: 1) the hash table lock array, and 2) the
 * arc list locks.
 *
 * Buffers do not have their own mutexes, rather they rely on the
 * hash table mutexes for the bulk of their protection (i.e. most
 * fields in the arc_buf_hdr_t are protected by these mutexes).
 *
 * buf_hash_find() returns the appropriate mutex (held) when it
 * locates the requested buffer in the hash table.  It returns
 * NULL for the mutex if the buffer was not in the table.
 *
 * buf_hash_remove() expects the appropriate hash mutex to be
 * already held before it is invoked.
 *
 * Each arc state also has a mutex which is used to protect the
 * buffer list associated with the state.  When attempting to
 * obtain a hash table lock while holding an arc list lock you
 * must use: mutex_tryenter() to avoid deadlock.  Also note that
 * the active state mutex must be held before the ghost state mutex.
 *
 * Arc buffers may have an associated eviction callback function.
 * This function will be invoked prior to removing the buffer (e.g.
 * in arc_do_user_evicts()).  Note however that the data associated
 * with the buffer may be evicted prior to the callback.  The callback
 * must be made with *no locks held* (to prevent deadlock).  Additionally,
 * the users of callbacks must ensure that their private data is
 * protected from simultaneous callbacks from arc_buf_evict()
 * and arc_do_user_evicts().
 *
 * Note that the majority of the performance stats are manipulated
 * with atomic operations.
 *
 * The L2ARC uses the l2arc_buflist_mtx global mutex for the following:
 *
 *      - L2ARC buflist creation
 *      - L2ARC buflist eviction
 *      - L2ARC write completion, which walks L2ARC buflists
 *      - ARC header destruction, as it removes from L2ARC buflists
 *      - ARC header release, as it removes from L2ARC buflists
 */

#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/zio_compress.h>
```

```
125 #include <sys/zfs_context.h>
126 #include <sys/arc.h>
127 #include <sys/refcount.h>
128 #include <sys/vdev.h>
129 #include <sys/vdev_impl.h>
130 #include <sys/dsl_pool.h>
131 #ifdef _KERNEL
132 #include <sys/vmsystm.h>
133 #include <vm/anon.h>
134 #include <sys/fs/swapnode.h>
135 #include <sys/dnlc.h>
136 #endif
137 #include <sys/callb.h>
138 #include <sys/kstat.h>
139 #include <zfs_fletcher.h>

141 #ifndef _KERNEL
142 /* set with ZFS_DEBUG=watch, to enable watchpoints on frozen buffers */
143 boolean_t arc_watch = B_FALSE;
144 int arc_procfd;
145 #endif

147 static kmutex_t         arc_reclaim_thr_lock;
148 static kcondvar_t       arc_reclaim_thr_cv;     /* used to signal reclaim thr */
149 static uint8_t          arc_thread_exit;

150 extern int zfs_write_limit_shift;
151 extern uint64_t zfs_write_limit_max;
152 extern kmutex_t zfs_write_limit_lock;

151 #define ARC_REDUCE_DNLC_PERCENT 3
152 uint_t arc_reduce_dnlc_percent = ARC_REDUCE_DNLC_PERCENT;

154 typedef enum arc_reclaim_strategy {
155         ARC_RECLAIM_AGGR,               /* Aggressive reclaim strategy */
156         ARC_RECLAIM_CONS                /* Conservative reclaim strategy */
157 } arc_reclaim_strategy_t;

159 /*
160  * The number of iterations through arc_evict_*() before we
161  * drop & reacquire the lock.
162  */
163 int arc_evict_iterations = 100;

165 /* number of seconds before growing cache again */
166 static int              arc_grow_retry = 60;

168 /* shift of arc_c for calculating both min and max arc_p */
169 static int              arc_p_min_shift = 4;

171 /* log2(fraction of arc to reclaim) */
172 static int              arc_shrink_shift = 5;

174 /*
175  * minimum lifespan of a prefetch block in clock ticks
176  * (initialized in arc_init())
177  */
178 static int              arc_min_prefetch_lifespan;

180 /*
181  * If this percent of memory is free, don't throttle.
182  */
183 int arc_lotsfree_percent = 10;

185 static int arc_dead;
```



```
187 /*
188  * The arc has filled available memory and has now warmed up.
189  */
190 static boolean_t arc_warm;

192 /*
193  * These tunables are for performance analysis.
194  */
195 uint64_t zfs_arc_max;
196 uint64_t zfs_arc_min;
197 uint64_t zfs_arc_meta_limit = 0;
198 int zfs_arc_grow_retry = 0;
199 int zfs_arc_shrink_shift = 0;
200 int zfs_arc_p_min_shift = 0;
201 int zfs_disable_dup_eviction = 0;

203 /*
204  * Note that buffers can be in one of 6 states:
205  *      ARC_anon        - anonymous (discussed below)
206  *      ARC_mru         - recently used, currently cached
207  *      ARC_mru_ghost   - recentely used, no longer in cache
208  *      ARC_mfu         - frequently used, currently cached
209  *      ARC_mfu_ghost   - frequently used, no longer in cache
210  *      ARC_l2c_only    - exists in L2ARC but not other states
211  * When there are no active references to the buffer, they are
212  * are linked onto a list in one of these arc states.  These are
213  * the only buffers that can be evicted or deleted.  Within each
214  * state there are multiple lists, one for meta-data and one for
215  * non-meta-data.  Meta-data (indirect blocks, blocks of dnodes,
216  * etc.) is tracked separately so that it can be managed more
217  * explicitly: favored over data, limited explicitly.
218  *
219  * Anonymous buffers are buffers that are not associated with
220  * a DVA.  These are buffers that hold dirty block copies
221  * before they are written to stable storage.  By definition,
222  * they are "ref'd" and are considered part of arc_mru
223  * that cannot be freed.  Generally, they will aquire a DVA
224  * as they are written and migrate onto the arc_mru list.
225  *
226  * The ARC_l2c_only state is for buffers that are in the second
227  * level ARC but no longer in any of the ARC_m* lists.  The second
228  * level ARC itself may also contain buffers that are in any of
229  * the ARC_m* states - meaning that a buffer can exist in two
230  * places.  The reason for the ARC_l2c_only state is to keep the
231  * buffer header in the hash table, so that reads that hit the
232  * second level ARC benefit from these fast lookups.
233  */

235 typedef struct arc_state {
236         list_t  arcs_list[ARC_BUFC_NUMTYPES];   /* list of evictable buffers */
237         uint64_t arcs_lsize[ARC_BUFC_NUMTYPES]; /* amount of evictable data */
238         uint64_t arcs_size;     /* total amount of data in this state */
239         kmutex_t arcs_mtx;
240 } arc_state_t;
_____unchanged_portion_omitted_

475 typedef struct arc_write_callback arc_write_callback_t;

477 struct arc_write_callback {
478         void            *awcb_private;
479         arc_done_func_t *awcb_ready;
480         arc_done_func_t *awcb_physdone;
481         arc_done_func_t *awcb_done;
482         arc_buf_t       *awcb_buf;
483 };
_____unchanged_portion_omitted_
```

```
1163 /*
1164  * Move the supplied buffer to the indicated state.  The mutex
1165  * for the buffer must be held by the caller.
1166  */
1167 static void
1168 arc_change_state(arc_state_t *new_state, arc_buf_hdr_t *ab, kmutex_t *hash_lock)
1169 {
1170         arc_state_t *old_state = ab->b_state;
1171         int64_t refcnt = refcount_count(&ab->b_refcnt);
1172         uint64_t from_delta, to_delta;

1174         ASSERT(MUTEX_HELD(hash_lock));
1175         ASSERT3P(new_state, !=, old_state);
1166         ASSERT(new_state != old_state);
1176         ASSERT(refcnt == 0 || ab->b_datacnt > 0);
1177         ASSERT(ab->b_datacnt == 0 || !GHOST_STATE(new_state));
1178         ASSERT(ab->b_datacnt <= 1 || old_state != arc_anon);

1180         from_delta = to_delta = ab->b_datacnt * ab->b_size;

1182         /*
1183          * If this buffer is evictable, transfer it from the
1184          * old state list to the new state list.
1185          */
1186         if (refcnt == 0) {
1187                 if (old_state != arc_anon) {
1188                         int use_mutex = !MUTEX_HELD(&old_state->arcs_mtx);
1189                         uint64_t *size = &old_state->arcs_lsize[ab->b_type];

1191                         if (use_mutex)
1192                                 mutex_enter(&old_state->arcs_mtx);

1194                         ASSERT(list_link_active(&ab->b_arc_node));
1195                         list_remove(&old_state->arcs_list[ab->b_type], ab);

1197                         /*
1198                          * If prefetching out of the ghost cache,
1199                          * we will have a non-zero datacnt.
1200                          */
1201                         if (GHOST_STATE(old_state) && ab->b_datacnt == 0) {
1202                                 /* ghost elements have a ghost size */
1203                                 ASSERT(ab->b_buf == NULL);
1204                                 from_delta = ab->b_size;
1205                         }
1206                         ASSERT3U(*size, >=, from_delta);
1207                         atomic_add_64(size, -from_delta);

1209                         if (use_mutex)
1210                                 mutex_exit(&old_state->arcs_mtx);
1211                 }
1212                 if (new_state != arc_anon) {
1213                         int use_mutex = !MUTEX_HELD(&new_state->arcs_mtx);
1214                         uint64_t *size = &new_state->arcs_lsize[ab->b_type];

1216                         if (use_mutex)
1217                                 mutex_enter(&new_state->arcs_mtx);

1219                         list_insert_head(&new_state->arcs_list[ab->b_type], ab);

1221                         /* ghost elements have a ghost size */
1222                         if (GHOST_STATE(new_state)) {
1223                                 ASSERT(ab->b_datacnt == 0);
1224                                 ASSERT(ab->b_buf == NULL);
1225                                 to_delta = ab->b_size;
1226                         }
```



```
1227                         atomic_add_64(size, to_delta);

1229                         if (use_mutex)
1230                                 mutex_exit(&new_state->arcs_mtx);
1231                 }
1232         }

1234         ASSERT(!BUF_EMPTY(ab));
1235         if (new_state == arc_anon && HDR_IN_HASH_TABLE(ab))
1236                 buf_hash_remove(ab);

1238         /* adjust state sizes */
1239         if (to_delta)
1240                 atomic_add_64(&new_state->arcs_size, to_delta);
1241         if (from_delta) {
1242                 ASSERT3U(old_state->arcs_size, >=, from_delta);
1243                 atomic_add_64(&old_state->arcs_size, -from_delta);
1244         }
1245         ab->b_state = new_state;

1247         /* adjust l2arc hdr stats */
1248         if (new_state == arc_l2c_only)
1249                 l2arc_hdr_stat_add();
1250         else if (old_state == arc_l2c_only)
1251                 l2arc_hdr_stat_remove();
1252 }
```

*unchanged_portion_omitted_*

```
1766 /*
1767  * Evict buffers from list until we've removed the specified number of
1768  * bytes.  Move the removed buffers to the appropriate evict state.
1769  * If the recycle flag is set, then attempt to "recycle" a buffer:
1770  * - look for a buffer to evict that is 'bytes' long.
1771  * - return the data block from this buffer rather than freeing it.
1772  * This flag is used by callers that are trying to make space for a
1773  * new buffer in a full arc cache.
1774  *
1775  * This function makes a "best effort".  It skips over any buffers
1776  * it can't get a hash_lock on, and so may not catch all candidates.
1777  * It may also return without evicting as much space as requested.
1778  */
1779 static void *
1780 arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle,
1781     arc_buf_contents_t type)
1782 {
1783         arc_state_t *evicted_state;
1784         uint64_t bytes_evicted = 0, skipped = 0, missed = 0;
1785         arc_buf_hdr_t *ab, *ab_prev = NULL;
1786         list_t *list = &state->arcs_list[type];
1787         kmutex_t *hash_lock;
1788         boolean_t have_lock;
1789         void *stolen = NULL;
1790         arc_buf_hdr_t marker = { 0 };
1791         int count = 0;

1793         ASSERT(state == arc_mru || state == arc_mfu);

1795         evicted_state = (state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;

1797         mutex_enter(&state->arcs_mtx);
1798         mutex_enter(&evicted_state->arcs_mtx);

1800         for (ab = list_tail(list); ab; ab = ab_prev) {
1801                 ab_prev = list_prev(list, ab);
1802                 /* prefetch buffers have a minimum lifespan */
1803                 if (HDR_IO_IN_PROGRESS(ab) ||
```

```
1804                    (spa && ab->b_spa != spa) ||
1805                    (ab->b_flags & (ARC_PREFETCH|ARC_INDIRECT) &&
1806                    ddi_get_lbolt() - ab->b_arc_access <
1807                    arc_min_prefetch_lifespan)) {
1808                        skipped++;
1809                        continue;
1810                }
1811                /* "lookahead" for better eviction candidate */
1812                if (recycle && ab->b_size != bytes &&
1813                    ab_prev && ab_prev->b_size == bytes)
1814                        continue;

1816                /* ignore markers */
1817                if (ab->b_spa == 0)
1818                        continue;

1820                /*
1821                 * It may take a long time to evict all the bufs requested.
1822                 * To avoid blocking all arc activity, periodically drop
1823                 * the arcs_mtx and give other threads a chance to run
1824                 * before reacquiring the lock.
1825                 *
1826                 * If we are looking for a buffer to recycle, we are in
1827                 * the hot code path, so don't sleep.
1828                 */
1829                if (!recycle && count++ > arc_evict_iterations) {
1830                        list_insert_after(list, ab, &marker);
1831                        mutex_exit(&evicted_state->arcs_mtx);
1832                        mutex_exit(&state->arcs_mtx);
1833                        kpreempt(KPREEMPT_SYNC);
1834                        mutex_enter(&state->arcs_mtx);
1835                        mutex_enter(&evicted_state->arcs_mtx);
1836                        ab_prev = list_prev(list, &marker);
1837                        list_remove(list, &marker);
1838                        count = 0;
1839                        continue;
1840                }

1842                hash_lock = HDR_LOCK(ab);
1843                have_lock = MUTEX_HELD(hash_lock);
1844                if (have_lock || mutex_tryenter(hash_lock)) {
1845                        ASSERT0(refcount_count(&ab->b_refcnt));
1846                        ASSERT(ab->b_datacnt > 0);
1847                        while (ab->b_buf) {
1848                                arc_buf_t *buf = ab->b_buf;
1849                                if (!mutex_tryenter(&buf->b_evict_lock)) {
1850                                        missed += 1;
1851                                        break;
1852                                }
1853                                if (buf->b_data) {
1854                                        bytes_evicted += ab->b_size;
1855                                        if (recycle && ab->b_type == type &&
1856                                            ab->b_size == bytes &&
1857                                            !HDR_L2_WRITING(ab)) {
1858                                                stolen = buf->b_data;
1859                                                recycle = FALSE;
1860                                        }
1861                                }
1862                                if (buf->b_efunc) {
1863                                        mutex_enter(&arc_eviction_mtx);
1864                                        arc_buf_destroy(buf,
1865                                            buf->b_data == stolen, FALSE);
1866                                        ab->b_buf = buf->b_next;
1867                                        buf->b_hdr = &arc_eviction_hdr;
1868                                        buf->b_next = arc_eviction_list;
1869                                        arc_eviction_list = buf;
```



```
1870                                        mutex_exit(&arc_eviction_mtx);
1871                                        mutex_exit(&buf->b_evict_lock);
1872                                } else {
1873                                        mutex_exit(&buf->b_evict_lock);
1874                                        arc_buf_destroy(buf,
1875                                            buf->b_data == stolen, TRUE);
1876                                }
1877                        }

1879                        if (ab->b_l2hdr) {
1880                                ARCSTAT_INCR(arcstat_evict_l2_cached,
1881                                    ab->b_size);
1882                        } else {
1883                                if (l2arc_write_eligible(ab->b_spa, ab)) {
1884                                        ARCSTAT_INCR(arcstat_evict_l2_eligible,
1885                                            ab->b_size);
1886                                } else {
1887                                        ARCSTAT_INCR(
1888                                            arcstat_evict_l2_ineligible,
1889                                            ab->b_size);
1890                                }
1891                        }

1893                        if (ab->b_datacnt == 0) {
1894                                arc_change_state(evicted_state, ab, hash_lock);
1895                                ASSERT(HDR_IN_HASH_TABLE(ab));
1896                                ab->b_flags |= ARC_IN_HASH_TABLE;
1897                                ab->b_flags &= ~ARC_BUF_AVAILABLE;
1898                                DTRACE_PROBE1(arc__evict, arc_buf_hdr_t *, ab);
1899                        }
1900                        if (!have_lock)
1901                                mutex_exit(hash_lock);
1902                        if (bytes >= 0 && bytes_evicted >= bytes)
1903                                break;
1904                } else {
1905                        missed += 1;
1906                }
1907        }

1909        mutex_exit(&evicted_state->arcs_mtx);
1910        mutex_exit(&state->arcs_mtx);

1912        if (bytes_evicted < bytes)
1913                dprintf("only evicted %lld bytes from %x",
1914                    (longlong_t)bytes_evicted, state);

1916        if (skipped)
1917                ARCSTAT_INCR(arcstat_evict_skip, skipped);

1919        if (missed)
1920                ARCSTAT_INCR(arcstat_mutex_miss, missed);

1922        /*
1923         * Note: we have just evicted some data into the ghost state,
1924         * potentially putting the ghost size over the desired size.  Rather
1925         * that evicting from the ghost list in this hot code path, leave
1926         * this chore to the arc_reclaim_thread().
1885         * We have just evicted some data into the ghost state, make
1886         * sure we also adjust the ghost state size if necessary.
1927         */
1888        if (arc_no_grow &&
1889            arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size > arc_c) {
1890                int64_t mru_over = arc_anon->arcs_size + arc_mru->arcs_size +
1891                    arc_mru_ghost->arcs_size - arc_c;

1893                if (mru_over > 0 && arc_mru_ghost->arcs_lsize[type] > 0) {
```

```
1894                        int64_t todelete =
1895                            MIN(arc_mru_ghost->arcs_lsize[type], mru_over);
1896                        arc_evict_ghost(arc_mru_ghost, NULL, todelete);
1897                } else if (arc_mfu_ghost->arcs_lsize[type] > 0) {
1898                        int64_t todelete = MIN(arc_mfu_ghost->arcs_lsize[type],
1899                            arc_mru_ghost->arcs_size +
1900                            arc_mfu_ghost->arcs_size - arc_c);
1901                        arc_evict_ghost(arc_mfu_ghost, NULL, todelete);
1902                }
1903        }

1929        return (stolen);
1930 }

1932 /*
1933  * Remove buffers from list until we've removed the specified number of
1934  * bytes.  Destroy the buffers that are removed.
1935  */
1936 static void
1937 arc_evict_ghost(arc_state_t *state, uint64_t spa, int64_t bytes)
1938 {
1939        arc_buf_hdr_t *ab, *ab_prev;
1940        arc_buf_hdr_t marker = { 0 };
1941        list_t *list = &state->arcs_list[ARC_BUFC_DATA];
1942        kmutex_t *hash_lock;
1943        uint64_t bytes_deleted = 0;
1944        uint64_t bufs_skipped = 0;
1945        int count = 0;

1947        ASSERT(GHOST_STATE(state));
1948 top:
1949        mutex_enter(&state->arcs_mtx);
1950        for (ab = list_tail(list); ab; ab = ab_prev) {
1951                ab_prev = list_prev(list, ab);
1952                if (ab->b_type > ARC_BUFC_NUMTYPES)
1953                        panic("invalid ab=%p", (void *)ab);
1954                if (spa && ab->b_spa != spa)
1955                        continue;

1957                /* ignore markers */
1958                if (ab->b_spa == 0)
1959                        continue;

1961                hash_lock = HDR_LOCK(ab);
1962                /* caller may be trying to modify this buffer, skip it */
1963                if (MUTEX_HELD(hash_lock))
1964                        continue;

1966                /*
1967                 * It may take a long time to evict all the bufs requested.
1968                 * To avoid blocking all arc activity, periodically drop
1969                 * the arcs_mtx and give other threads a chance to run
1970                 * before reacquiring the lock.
1971                 */
1972                if (count++ > arc_evict_iterations) {
1973                        list_insert_after(list, ab, &marker);
1974                        mutex_exit(&state->arcs_mtx);
1975                        kpreempt(KPREEMPT_SYNC);
1976                        mutex_enter(&state->arcs_mtx);
1977                        ab_prev = list_prev(list, &marker);
1978                        list_remove(list, &marker);
1979                        count = 0;
1980                        continue;
1981                }
1982                if (mutex_tryenter(hash_lock)) {
1983                        ASSERT(!HDR_IO_IN_PROGRESS(ab));
```



```
1984                        ASSERT(ab->b_buf == NULL);
1985                        ARCSTAT_BUMP(arcstat_deleted);
1986                        bytes_deleted += ab->b_size;

1988                        if (ab->b_l2hdr != NULL) {
1989                                /*
1990                                 * This buffer is cached on the 2nd Level ARC;
1991                                 * don't destroy the header.
1992                                 */
1993                                arc_change_state(arc_l2c_only, ab, hash_lock);
1994                                mutex_exit(hash_lock);
1995                        } else {
1996                                arc_change_state(arc_anon, ab, hash_lock);
1997                                mutex_exit(hash_lock);
1998                                arc_hdr_destroy(ab);
1999                        }

2001                        DTRACE_PROBE1(arc__delete, arc_buf_hdr_t *, ab);
2002                        if (bytes >= 0 && bytes_deleted >= bytes)
2003                                break;
2004                } else if (bytes < 0) {
2005                        /*
2006                         * Insert a list marker and then wait for the
2007                         * hash lock to become available. Once its
2008                         * available, restart from where we left off.
2009                         */
2010                        list_insert_after(list, ab, &marker);
2011                        mutex_exit(&state->arcs_mtx);
2012                        mutex_enter(hash_lock);
2013                        mutex_exit(hash_lock);
2014                        mutex_enter(&state->arcs_mtx);
2015                        ab_prev = list_prev(list, &marker);
2016                        list_remove(list, &marker);
2017                } else {
1973                } else
2018                        bufs_skipped += 1;
2019                }

2021        }
2022        mutex_exit(&state->arcs_mtx);

2024        if (list == &state->arcs_list[ARC_BUFC_DATA] &&
2025            (bytes < 0 || bytes_deleted < bytes)) {
2026                list = &state->arcs_list[ARC_BUFC_METADATA];
2027                goto top;
2028        }

2030        if (bufs_skipped) {
2031                ARCSTAT_INCR(arcstat_mutex_miss, bufs_skipped);
2032                ASSERT(bytes >= 0);
2033        }

2035        if (bytes_deleted < bytes)
2036                dprintf("only deleted %lld bytes from %p",
2037                    (longlong_t)bytes_deleted, state);
2038 }
_____unchanged_portion_omitted_

2854 /*
2855  * "Read" the block at the specified DVA (in bp) via the
2856  * cache.  If the block is found in the cache, invoke the provided
2857  * callback immediately and return.  Note that the 'zio' parameter
2858  * in the callback will be NULL in this case, since no IO was
2859  * required.  If the block is not in the cache pass the read request
2860  * on to the spa with a substitute callback function, so that the
2861  * requested block will be added to the cache.
```

```
 *
 * If a read request arrives for a block that has a read in-progress,
 * either wait for the in-progress read to complete (and return the
 * results); or, if this is a read with a "done" func, add a record
 * to the read to invoke the "done" func when the read completes,
 * and return; or just return.
 *
 * arc_read_done() will invoke all the requested "done" functions
 * for readers of this block.
 */
int
arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, arc_done_func_t *done,
    void *private, zio_priority_t priority, int zio_flags, uint32_t *arc_flags,
    void *private, int priority, int zio_flags, uint32_t *arc_flags,
    const zbookmark_t *zb)
{
        arc_buf_hdr_t *hdr;
        arc_buf_t *buf = NULL;
        kmutex_t *hash_lock;
        zio_t *rzio;
        uint64_t guid = spa_load_guid(spa);

top:
        hdr = buf_hash_find(guid, BP_IDENTITY(bp), BP_PHYSICAL_BIRTH(bp),
            &hash_lock);
        if (hdr && hdr->b_datacnt > 0) {

                *arc_flags |= ARC_CACHED;

                if (HDR_IO_IN_PROGRESS(hdr)) {

                        if (*arc_flags & ARC_WAIT) {
                                cv_wait(&hdr->b_cv, hash_lock);
                                mutex_exit(hash_lock);
                                goto top;
                        }
                        ASSERT(*arc_flags & ARC_NOWAIT);

                        if (done) {
                                arc_callback_t  *acb = NULL;

                                acb = kmem_zalloc(sizeof (arc_callback_t),
                                    KM_SLEEP);
                                acb->acb_done = done;
                                acb->acb_private = private;
                                if (pio != NULL)
                                        acb->acb_zio_dummy = zio_null(pio,
                                            spa, NULL, NULL, NULL, zio_flags);

                                ASSERT(acb->acb_done != NULL);
                                acb->acb_next = hdr->b_acb;
                                hdr->b_acb = acb;
                                add_reference(hdr, hash_lock, private);
                                mutex_exit(hash_lock);
                                return (0);
                        }
                        mutex_exit(hash_lock);
                        return (0);
                }

                ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);

                if (done) {
                        add_reference(hdr, hash_lock, private);
                        /*
                         * If this block is already in use, create a new
```



```
                         * copy of the data so that we will be guaranteed
                         * that arc_release() will always succeed.
                         */
                        buf = hdr->b_buf;
                        ASSERT(buf);
                        ASSERT(buf->b_data);
                        if (HDR_BUF_AVAILABLE(hdr)) {
                                ASSERT(buf->b_efunc == NULL);
                                hdr->b_flags &= ~ARC_BUF_AVAILABLE;
                        } else {
                                buf = arc_buf_clone(buf);
                        }

                } else if (*arc_flags & ARC_PREFETCH &&
                    refcount_count(&hdr->b_refcnt) == 0) {
                        hdr->b_flags |= ARC_PREFETCH;
                }
                DTRACE_PROBE1(arc__hit, arc_buf_hdr_t *, hdr);
                arc_access(hdr, hash_lock);
                if (*arc_flags & ARC_L2CACHE)
                        hdr->b_flags |= ARC_L2CACHE;
                if (*arc_flags & ARC_L2COMPRESS)
                        hdr->b_flags |= ARC_L2COMPRESS;
                mutex_exit(hash_lock);
                ARCSTAT_BUMP(arcstat_hits);
                ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
                    demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
                    data, metadata, hits);

                if (done)
                        done(NULL, buf, private);
        } else {
                uint64_t size = BP_GET_LSIZE(bp);
                arc_callback_t  *acb;
                vdev_t *vd = NULL;
                uint64_t addr = 0;
                boolean_t devw = B_FALSE;

                if (hdr == NULL) {
                        /* this block is not in the cache */
                        arc_buf_hdr_t   *exists;
                        arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
                        buf = arc_buf_alloc(spa, size, private, type);
                        hdr = buf->b_hdr;
                        hdr->b_dva = *BP_IDENTITY(bp);
                        hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
                        hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
                        exists = buf_hash_insert(hdr, &hash_lock);
                        if (exists) {
                                /* somebody beat us to the hash insert */
                                mutex_exit(hash_lock);
                                buf_discard_identity(hdr);
                                (void) arc_buf_remove_ref(buf, private);
                                goto top; /* restart the IO request */
                        }
                        /* if this is a prefetch, we don't have a reference */
                        if (*arc_flags & ARC_PREFETCH) {
                                (void) remove_reference(hdr, hash_lock,
                                    private);
                                hdr->b_flags |= ARC_PREFETCH;
                        }
                        if (*arc_flags & ARC_L2CACHE)
                                hdr->b_flags |= ARC_L2CACHE;
                        if (*arc_flags & ARC_L2COMPRESS)
                                hdr->b_flags |= ARC_L2COMPRESS;
                        if (BP_GET_LEVEL(bp) > 0)
```

```
2993                                    hdr->b_flags |= ARC_INDIRECT;
2994                    } else {
2995                            /* this block is in the ghost cache */
2996                            ASSERT(GHOST_STATE(hdr->b_state));
2997                            ASSERT(!HDR_IO_IN_PROGRESS(hdr));
2998                            ASSERT0(refcount_count(&hdr->b_refcnt));
2999                            ASSERT(hdr->b_buf == NULL);

3001                            /* if this is a prefetch, we don't have a reference */
3002                            if (*arc_flags & ARC_PREFETCH)
3003                                    hdr->b_flags |= ARC_PREFETCH;
3004                            else
3005                                    add_reference(hdr, hash_lock, private);
3006                            if (*arc_flags & ARC_L2CACHE)
3007                                    hdr->b_flags |= ARC_L2CACHE;
3008                            if (*arc_flags & ARC_L2COMPRESS)
3009                                    hdr->b_flags |= ARC_L2COMPRESS;
3010                            buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);
3011                            buf->b_hdr = hdr;
3012                            buf->b_data = NULL;
3013                            buf->b_efunc = NULL;
3014                            buf->b_private = NULL;
3015                            buf->b_next = NULL;
3016                            hdr->b_buf = buf;
3017                            ASSERT(hdr->b_datacnt == 0);
3018                            hdr->b_datacnt = 1;
3019                            arc_get_data_buf(buf);
3020                            arc_access(hdr, hash_lock);
3021                    }

3023                    ASSERT(!GHOST_STATE(hdr->b_state));

3025                    acb = kmem_zalloc(sizeof (arc_callback_t), KM_SLEEP);
3026                    acb->acb_done = done;
3027                    acb->acb_private = private;

3029                    ASSERT(hdr->b_acb == NULL);
3030                    hdr->b_acb = acb;
3031                    hdr->b_flags |= ARC_IO_IN_PROGRESS;

3033                    if (HDR_L2CACHE(hdr) && hdr->b_l2hdr != NULL &&
3034                        (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
3035                            devw = hdr->b_l2hdr->b_dev->l2ad_writing;
3036                            addr = hdr->b_l2hdr->b_daddr;
3037                            /*
3038                             * Lock out device removal.
3039                             */
3040                            if (vdev_is_dead(vd) ||
3041                                !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
3042                                    vd = NULL;
3043                    }

3045                    mutex_exit(hash_lock);

3047                    /*
3048                     * At this point, we have a level 1 cache miss.  Try again in
3049                     * L2ARC if possible.
3050                     */
3051                    ASSERT3U(hdr->b_size, ==, size);
3052                    DTRACE_PROBE4(arc__miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
3053                        uint64_t, size, zbookmark_t *, zb);
3054                    ARCSTAT_BUMP(arcstat_misses);
3055                    ARCSTAT_CONDSTAT(!(hdr->b_flags & ARC_PREFETCH),
3056                        demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
3057                        data, metadata, misses);
```



```
3059                    if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
3060                            /*
3061                             * Read from the L2ARC if the following are true:
3062                             * 1. The L2ARC vdev was previously cached.
3063                             * 2. This buffer still has L2ARC metadata.
3064                             * 3. This buffer isn't currently writing to the L2ARC.
3065                             * 4. The L2ARC entry wasn't evicted, which may
3066                             *    also have invalidated the vdev.
3067                             * 5. This isn't prefetch and l2arc_noprefetch is set.
3068                             */
3069                            if (hdr->b_l2hdr != NULL &&
3070                                !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
3071                                !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
3072                                    l2arc_read_callback_t *cb;

3074                                    DTRACE_PROBE1(l2arc__hit, arc_buf_hdr_t *, hdr);
3075                                    ARCSTAT_BUMP(arcstat_l2_hits);

3077                                    cb = kmem_zalloc(sizeof (l2arc_read_callback_t),
3078                                        KM_SLEEP);
3079                                    cb->l2rcb_buf = buf;
3080                                    cb->l2rcb_spa = spa;
3081                                    cb->l2rcb_bp = *bp;
3082                                    cb->l2rcb_zb = *zb;
3083                                    cb->l2rcb_flags = zio_flags;
3084                                    cb->l2rcb_compress = hdr->b_l2hdr->b_compress;

3086                                    ASSERT(addr >= VDEV_LABEL_START_SIZE &&
3087                                        addr + size < vd->vdev_psize -
3088                                        VDEV_LABEL_END_SIZE);

3090                                    /*
3091                                     * l2arc read.  The SCL_L2ARC lock will be
3092                                     * released by l2arc_read_done().
3093                                     * Issue a null zio if the underlying buffer
3094                                     * was squashed to zero size by compression.
3095                                     */
3096                                    if (hdr->b_l2hdr->b_compress ==
3097                                        ZIO_COMPRESS_EMPTY) {
3098                                            rzio = zio_null(pio, spa, vd,
3099                                                l2arc_read_done, cb,
3100                                                zio_flags | ZIO_FLAG_DONT_CACHE |
3101                                                ZIO_FLAG_CANFAIL |
3102                                                ZIO_FLAG_DONT_PROPAGATE |
3103                                                ZIO_FLAG_DONT_RETRY);
3104                                    } else {
3105                                            rzio = zio_read_phys(pio, vd, addr,
3106                                                hdr->b_l2hdr->b_asize,
3107                                                buf->b_data, ZIO_CHECKSUM_OFF,
3108                                                l2arc_read_done, cb, priority,
3109                                                zio_flags | ZIO_FLAG_DONT_CACHE |
3110                                                ZIO_FLAG_CANFAIL |
3111                                                ZIO_FLAG_DONT_PROPAGATE |
3112                                                ZIO_FLAG_DONT_RETRY, B_FALSE);
3113                                    }
3114                                    DTRACE_PROBE2(l2arc__read, vdev_t *, vd,
3115                                        zio_t *, rzio);
3116                                    ARCSTAT_INCR(arcstat_l2_read_bytes,
3117                                        hdr->b_l2hdr->b_asize);

3119                                    if (*arc_flags & ARC_NOWAIT) {
3120                                            zio_nowait(rzio);
3121                                            return (0);
3122                                    }

3124                                    ASSERT(*arc_flags & ARC_WAIT);
```

```
3125                                if (zio_wait(rzio) == 0)
3126                                        return (0);
3127
3128                                /* l2arc read error; goto zio_read() */
3129                        } else {
3130                                DTRACE_PROBE1(l2arc__miss,
3131                                    arc_buf_hdr_t *, hdr);
3132                                ARCSTAT_BUMP(arcstat_l2_misses);
3133                                if (HDR_L2_WRITING(hdr))
3134                                        ARCSTAT_BUMP(arcstat_l2_rw_clash);
3135                                spa_config_exit(spa, SCL_L2ARC, vd);
3136                        }
3137                } else {
3138                        if (vd != NULL)
3139                                spa_config_exit(spa, SCL_L2ARC, vd);
3140                        if (l2arc_ndev != 0) {
3141                                DTRACE_PROBE1(l2arc__miss,
3142                                    arc_buf_hdr_t *, hdr);
3143                                ARCSTAT_BUMP(arcstat_l2_misses);
3144                        }
3145                }
3146
3147                rzio = zio_read(pio, spa, bp, buf->b_data, size,
3148                    arc_read_done, buf, priority, zio_flags, zb);
3149
3150                if (*arc_flags & ARC_WAIT)
3151                        return (zio_wait(rzio));
3152
3153                ASSERT(*arc_flags & ARC_NOWAIT);
3154                zio_nowait(rzio);
3155        }
3156        return (0);
3157 }
```

*_____unchanged_portion_omitted_*

```
3477 /*
3478  * The SPA calls this callback for each physical write that happens on behalf
3479  * of a logical write.  See the comment in dbuf_write_physdone() for details.
3480  */
3481 static void
3482 arc_write_physdone(zio_t *zio)
3483 {
3484         arc_write_callback_t *cb = zio->io_private;
3485         if (cb->awcb_physdone != NULL)
3486                 cb->awcb_physdone(zio, cb->awcb_buf, cb->awcb_private);
3487 }
3488
3489 static void
3490 arc_write_done(zio_t *zio)
3491 {
3492         arc_write_callback_t *callback = zio->io_private;
3493         arc_buf_t *buf = callback->awcb_buf;
3494         arc_buf_hdr_t *hdr = buf->b_hdr;
3495
3496         ASSERT(hdr->b_acb == NULL);
3497
3498         if (zio->io_error == 0) {
3499                 hdr->b_dva = *BP_IDENTITY(zio->io_bp);
3500                 hdr->b_birth = BP_PHYSICAL_BIRTH(zio->io_bp);
3501                 hdr->b_cksum0 = zio->io_bp->blk_cksum.zc_word[0];
3502         } else {
3503                 ASSERT(BUF_EMPTY(hdr));
3504         }
3505
3506         /*
3507          * If the block to be written was all-zero, we may have
```



```
3508          * compressed it away.  In this case no write was performed
3509          * so there will be no dva/birth/checksum.  The buffer must
3510          * therefore remain anonymous (and uncached).
3511          */
3512         if (!BUF_EMPTY(hdr)) {
3513                 arc_buf_hdr_t *exists;
3514                 kmutex_t *hash_lock;
3515
3516                 ASSERT(zio->io_error == 0);
3517
3518                 arc_cksum_verify(buf);
3519
3520                 exists = buf_hash_insert(hdr, &hash_lock);
3521                 if (exists) {
3522                         /*
3523                          * This can only happen if we overwrite for
3524                          * sync-to-convergence, because we remove
3525                          * buffers from the hash table when we arc_free().
3526                          */
3527                         if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
3528                                 if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
3529                                         panic("bad overwrite, hdr=%p exists=%p",
3530                                             (void *)hdr, (void *)exists);
3531                                 ASSERT(refcount_is_zero(&exists->b_refcnt));
3532                                 arc_change_state(arc_anon, exists, hash_lock);
3533                                 mutex_exit(hash_lock);
3534                                 arc_hdr_destroy(exists);
3535                                 exists = buf_hash_insert(hdr, &hash_lock);
3536                                 ASSERT3P(exists, ==, NULL);
3537                         } else if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
3538                                 /* nopwrite */
3539                                 ASSERT(zio->io_prop.zp_nopwrite);
3540                                 if (!BP_EQUAL(&zio->io_bp_orig, zio->io_bp))
3541                                         panic("bad nopwrite, hdr=%p exists=%p",
3542                                             (void *)hdr, (void *)exists);
3543                         } else {
3544                                 /* Dedup */
3545                                 ASSERT(hdr->b_datacnt == 1);
3546                                 ASSERT(hdr->b_state == arc_anon);
3547                                 ASSERT(BP_GET_DEDUP(zio->io_bp));
3548                                 ASSERT(BP_GET_LEVEL(zio->io_bp) == 0);
3549                         }
3550                 }
3551                 hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
3552                 /* if it's not anon, we are doing a scrub */
3553                 if (!exists && hdr->b_state == arc_anon)
3554                         arc_access(hdr, hash_lock);
3555                 mutex_exit(hash_lock);
3556         } else {
3557                 hdr->b_flags &= ~ARC_IO_IN_PROGRESS;
3558         }
3559
3560         ASSERT(!refcount_is_zero(&hdr->b_refcnt));
3561         callback->awcb_done(zio, buf, callback->awcb_private);
3562
3563         kmem_free(callback, sizeof (arc_write_callback_t));
3564 }
3565
3566 zio_t *
3567 arc_write(zio_t *pio, spa_t *spa, uint64_t txg,
3568     blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, boolean_t l2arc_compress,
3569     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *physdone,
3570     arc_done_func_t *done, void *private, zio_priority_t priority,
3571     int zio_flags, const zbookmark_t *zb)
3511     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *done,
3512     void *private, int priority, int zio_flags, const zbookmark_t *zb)
```

```
3572 {
3573         arc_buf_hdr_t *hdr = buf->b_hdr;
3574         arc_write_callback_t *callback;
3575         zio_t *zio;

3577         ASSERT(ready != NULL);
3578         ASSERT(done != NULL);
3579         ASSERT(!HDR_IO_ERROR(hdr));
3580         ASSERT((hdr->b_flags & ARC_IO_IN_PROGRESS) == 0);
3581         ASSERT(hdr->b_acb == NULL);
3582         if (l2arc)
3583                 hdr->b_flags |= ARC_L2CACHE;
3584         if (l2arc_compress)
3585                 hdr->b_flags |= ARC_L2COMPRESS;
3586         callback = kmem_zalloc(sizeof (arc_write_callback_t), KM_SLEEP);
3587         callback->awcb_ready = ready;
3588         callback->awcb_physdone = physdone;
3589         callback->awcb_done = done;
3590         callback->awcb_private = private;
3591         callback->awcb_buf = buf;

3593         zio = zio_write(pio, spa, txg, bp, buf->b_data, hdr->b_size, zp,
3594             arc_write_ready, arc_write_physdone, arc_write_done, callback,
3595             priority, zio_flags, zb);
3534             arc_write_ready, arc_write_done, callback, priority, zio_flags, zb);

3597         return (zio);
3598 }

3600 static int
3601 arc_memory_throttle(uint64_t reserve, uint64_t txg)
3540 arc_memory_throttle(uint64_t reserve, uint64_t inflight_data, uint64_t txg)
3602 {
3603 #ifdef _KERNEL
3604         uint64_t available_memory = ptob(freemem);
3605         static uint64_t page_load = 0;
3606         static uint64_t last_txg = 0;

3608 #if defined(__i386)
3609         available_memory =
3610             MIN(available_memory, vmem_size(heap_arena, VMEM_FREE));
3611 #endif

3613         if (freemem > physmem * arc_lotsfree_percent / 100)
3551         if (available_memory >= zfs_write_limit_max)
3614                 return (0);

3616         if (txg > last_txg) {
3617                 last_txg = txg;
3618                 page_load = 0;
3619         }
3620         /*
3621          * If we are in pageout, we know that memory is already tight,
3622          * the arc is already going to be evicting, so we just want to
3623          * continue to let page writes occur as quickly as possible.
3624          */
3625         if (curproc == proc_pageout) {
3626                 if (page_load > MAX(ptob(minfree), available_memory) / 4)
3627                         return (SET_ERROR(ERESTART));
3628                 /* Note: reserve is inflated, so we deflate */
3629                 page_load += reserve / 8;
3630                 return (0);
3631         } else if (page_load > 0 && arc_reclaim_needed()) {
3632                 /* memory is low, delay before restarting */
3633                 ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
3634                 return (SET_ERROR(EAGAIN));
```



```
3635         }
3636         page_load = 0;

3576         if (arc_size > arc_c_min) {
3577                 uint64_t evictable_memory =
3578                     arc_mru->arcs_lsize[ARC_BUFC_DATA] +
3579                     arc_mru->arcs_lsize[ARC_BUFC_METADATA] +
3580                     arc_mfu->arcs_lsize[ARC_BUFC_DATA] +
3581                     arc_mfu->arcs_lsize[ARC_BUFC_METADATA];
3582                 available_memory += MIN(evictable_memory, arc_size - arc_c_min);
3583         }

3585         if (inflight_data > available_memory / 4) {
3586                 ARCSTAT_INCR(arcstat_memory_throttle_count, 1);
3587                 return (SET_ERROR(ERESTART));
3588         }
3637 #endif
3638         return (0);
3639 }
______unchanged_portion_omitted_

3648 int
3649 arc_tempreserve_space(uint64_t reserve, uint64_t txg)
3650 {
3651         int error;
3652         uint64_t anon_size;

3606 #ifdef ZFS_DEBUG
3607         /*
3608          * Once in a while, fail for no reason.  Everything should cope.
3609          */
3610         if (spa_get_random(10000) == 0) {
3611                 dprintf("forcing random failure\n");
3612                 return (SET_ERROR(ERESTART));
3613         }
3614 #endif
3654         if (reserve > arc_c/4 && !arc_no_grow)
3655                 arc_c = MIN(arc_c_max, reserve * 4);
3656         if (reserve > arc_c)
3657                 return (SET_ERROR(ENOMEM));

3659         /*
3660          * Don't count loaned bufs as in flight dirty data to prevent long
3661          * network delays from blocking transactions that are ready to be
3662          * assigned to a txg.
3663          */
3664         anon_size = MAX((int64_t)(arc_anon->arcs_size - arc_loaned_bytes), 0);

3666         /*
3667          * Writes will, almost always, require additional memory allocations
3668          * in order to compress/encrypt/etc the data.  We therefore need to
3669          * make sure that there is sufficient available memory for this.
3670          */
3671         error = arc_memory_throttle(reserve, txg);
3672         if (error != 0)
3632         if (error = arc_memory_throttle(reserve, anon_size, txg))
3673                 return (error);

3675         /*
3676          * Throttle writes when the amount of dirty data in the cache
3677          * gets too large.  We try to keep the cache less than half full
3678          * of dirty blocks so that our sync times don't grow too large.
3679          * Note: if two requests come in concurrently, we might let them
3680          * both succeed, when one of them should fail.  Not a huge deal.
3681          */
```

```
3683         if (reserve + arc_tempreserve + anon_size > arc_c / 2 &&
3684             anon_size > arc_c / 4) {
3685                 dprintf("failing, arc_tempreserve=%lluK anon_meta=%lluK "
3686                     "anon_data=%lluK tempreserve=%lluK arc_c=%lluK\n",
3687                     arc_tempreserve>>10,
3688                     arc_anon->arcs_lsize[ARC_BUFC_METADATA]>>10,
3689                     arc_anon->arcs_lsize[ARC_BUFC_DATA]>>10,
3690                     reserve>>10, arc_c>>10);
3691                 return (SET_ERROR(ERESTART));
3692         }
3693         atomic_add_64(&arc_tempreserve, reserve);
3694         return (0);
3695 }

3697 void
3698 arc_init(void)
3699 {
3700         mutex_init(&arc_reclaim_thr_lock, NULL, MUTEX_DEFAULT, NULL);
3701         cv_init(&arc_reclaim_thr_cv, NULL, CV_DEFAULT, NULL);

3703         /* Convert seconds to clock ticks */
3704         arc_min_prefetch_lifespan = 1 * hz;

3706         /* Start out with 1/8 of all memory */
3707         arc_c = physmem * PAGESIZE / 8;

3709 #ifdef _KERNEL
3710         /*
3711          * On architectures where the physical memory can be larger
3712          * than the addressable space (intel in 32-bit mode), we may
3713          * need to limit the cache to 1/8 of VM size.
3714          */
3715         arc_c = MIN(arc_c, vmem_size(heap_arena, VMEM_ALLOC | VMEM_FREE) / 8);
3716 #endif

3718         /* set min cache to 1/32 of all memory, or 64MB, whichever is more */
3719         arc_c_min = MAX(arc_c / 4, 64<<20);
3720         /* set max to 3/4 of all memory, or all but 1GB, whichever is more */
3721         if (arc_c * 8 >= 1<<30)
3722                 arc_c_max = (arc_c * 8) - (1<<30);
3723         else
3724                 arc_c_max = arc_c_min;
3725         arc_c_max = MAX(arc_c * 6, arc_c_max);

3727         /*
3728          * Allow the tunables to override our calculations if they are
3729          * reasonable (ie. over 64MB)
3730          */
3731         if (zfs_arc_max > 64<<20 && zfs_arc_max < physmem * PAGESIZE)
3732                 arc_c_max = zfs_arc_max;
3733         if (zfs_arc_min > 64<<20 && zfs_arc_min <= arc_c_max)
3734                 arc_c_min = zfs_arc_min;

3736         arc_c = arc_c_max;
3737         arc_p = (arc_c >> 1);

3739         /* limit meta-data to 1/4 of the arc capacity */
3740         arc_meta_limit = arc_c_max / 4;

3742         /* Allow the tunable to override if it is reasonable */
3743         if (zfs_arc_meta_limit > 0 && zfs_arc_meta_limit <= arc_c_max)
3744                 arc_meta_limit = zfs_arc_meta_limit;

3746         if (arc_c_min < arc_meta_limit / 2 && zfs_arc_min == 0)
3747                 arc_c_min = arc_meta_limit / 2;
```



```
3749         if (zfs_arc_grow_retry > 0)
3750                 arc_grow_retry = zfs_arc_grow_retry;

3752         if (zfs_arc_shrink_shift > 0)
3753                 arc_shrink_shift = zfs_arc_shrink_shift;

3755         if (zfs_arc_p_min_shift > 0)
3756                 arc_p_min_shift = zfs_arc_p_min_shift;

3758         /* if kmem_flags are set, lets try to use less memory */
3759         if (kmem_debugging())
3760                 arc_c = arc_c / 2;
3761         if (arc_c < arc_c_min)
3762                 arc_c = arc_c_min;

3764         arc_anon = &ARC_anon;
3765         arc_mru = &ARC_mru;
3766         arc_mru_ghost = &ARC_mru_ghost;
3767         arc_mfu = &ARC_mfu;
3768         arc_mfu_ghost = &ARC_mfu_ghost;
3769         arc_l2c_only = &ARC_l2c_only;
3770         arc_size = 0;

3772         mutex_init(&arc_anon->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3773         mutex_init(&arc_mru->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3774         mutex_init(&arc_mru_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3775         mutex_init(&arc_mfu->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3776         mutex_init(&arc_mfu_ghost->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);
3777         mutex_init(&arc_l2c_only->arcs_mtx, NULL, MUTEX_DEFAULT, NULL);

3779         list_create(&arc_mru->arcs_list[ARC_BUFC_METADATA],
3780             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3781         list_create(&arc_mru->arcs_list[ARC_BUFC_DATA],
3782             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3783         list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA],
3784             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3785         list_create(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA],
3786             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3787         list_create(&arc_mfu->arcs_list[ARC_BUFC_METADATA],
3788             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3789         list_create(&arc_mfu->arcs_list[ARC_BUFC_DATA],
3790             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3791         list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA],
3792             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3793         list_create(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA],
3794             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3795         list_create(&arc_l2c_only->arcs_list[ARC_BUFC_METADATA],
3796             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));
3797         list_create(&arc_l2c_only->arcs_list[ARC_BUFC_DATA],
3798             sizeof (arc_buf_hdr_t), offsetof(arc_buf_hdr_t, b_arc_node));

3800         buf_init();

3802         arc_thread_exit = 0;
3803         arc_eviction_list = NULL;
3804         mutex_init(&arc_eviction_mtx, NULL, MUTEX_DEFAULT, NULL);
3805         bzero(&arc_eviction_hdr, sizeof (arc_buf_hdr_t));

3807         arc_ksp = kstat_create("zfs", 0, "arcstats", "misc", KSTAT_TYPE_NAMED,
3808             sizeof (arc_stats) / sizeof (kstat_named_t), KSTAT_FLAG_VIRTUAL);

3810         if (arc_ksp != NULL) {
3811                 arc_ksp->ks_data = &arc_stats;
3812                 kstat_install(arc_ksp);
3813         }
```

```
3815         (void) thread_create(NULL, 0, arc_reclaim_thread, NULL, 0, &p0,
3816             TS_RUN, minclsyspri);

3818         arc_dead = FALSE;
3819         arc_warm = B_FALSE;

3821         /*
3822          * Calculate maximum amount of dirty data per pool.
3823          *
3824          * If it has been set by /etc/system, take that.
3825          * Otherwise, use a percentage of physical memory defined by
3826          * zfs_dirty_data_max_percent (default 10%) with a cap at
3827          * zfs_dirty_data_max_max (default 4GB).
3828          */
3829         if (zfs_dirty_data_max == 0) {
3830                 zfs_dirty_data_max = physmem * PAGESIZE *
3831                     zfs_dirty_data_max_percent / 100;
3832                 zfs_dirty_data_max = MIN(zfs_dirty_data_max,
3833                     zfs_dirty_data_max_max);
3834         }
3781         if (zfs_write_limit_max == 0)
3782                 zfs_write_limit_max = ptob(physmem) >> zfs_write_limit_shift;
3783         else
3784                 zfs_write_limit_shift = 0;
3785         mutex_init(&zfs_write_limit_lock, NULL, MUTEX_DEFAULT, NULL);
3835 }

3837 void
3838 arc_fini(void)
3839 {
3840         mutex_enter(&arc_reclaim_thr_lock);
3841         arc_thread_exit = 1;
3842         while (arc_thread_exit != 0)
3843                 cv_wait(&arc_reclaim_thr_cv, &arc_reclaim_thr_lock);
3844         mutex_exit(&arc_reclaim_thr_lock);

3846         arc_flush(NULL);

3848         arc_dead = TRUE;

3850         if (arc_ksp != NULL) {
3851                 kstat_delete(arc_ksp);
3852                 arc_ksp = NULL;
3853         }

3855         mutex_destroy(&arc_eviction_mtx);
3856         mutex_destroy(&arc_reclaim_thr_lock);
3857         cv_destroy(&arc_reclaim_thr_cv);

3859         list_destroy(&arc_mru->arcs_list[ARC_BUFC_METADATA]);
3860         list_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_METADATA]);
3861         list_destroy(&arc_mfu->arcs_list[ARC_BUFC_METADATA]);
3862         list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_METADATA]);
3863         list_destroy(&arc_mru->arcs_list[ARC_BUFC_DATA]);
3864         list_destroy(&arc_mru_ghost->arcs_list[ARC_BUFC_DATA]);
3865         list_destroy(&arc_mfu->arcs_list[ARC_BUFC_DATA]);
3866         list_destroy(&arc_mfu_ghost->arcs_list[ARC_BUFC_DATA]);

3868         mutex_destroy(&arc_anon->arcs_mtx);
3869         mutex_destroy(&arc_mru->arcs_mtx);
3870         mutex_destroy(&arc_mru_ghost->arcs_mtx);
3871         mutex_destroy(&arc_mfu->arcs_mtx);
3872         mutex_destroy(&arc_mfu_ghost->arcs_mtx);
3873         mutex_destroy(&arc_l2c_only->arcs_mtx);

3826         mutex_destroy(&zfs_write_limit_lock);
```



```
3875         buf_fini();

3877         ASSERT(arc_loaned_bytes == 0);
3878 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   76982 Thu Aug 22 16:15:05 2013
new/usr/src/uts/common/fs/zfs/dbuf.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 803 /*
 804  * Evict (if its unreferenced) or clear (if its referenced) any level-0
 805  * data blocks in the free range, so that any future readers will find
 806  * empty blocks.  Also, if we happen across any level-1 dbufs in the
 807  * range that have not already been marked dirty, mark them dirty so
 808  * they stay in memory.
 809  *
 810  * This is a no-op if the dataset is in the middle of an incremental
 811  * receive; see comment below for details.
 812  */
 813 void
 814 dbuf_free_range(dnode_t *dn, uint64_t start, uint64_t end, dmu_tx_t *tx)
 815 {
 816         dmu_buf_impl_t *db, *db_next;
 817         uint64_t txg = tx->tx_txg;
 818         int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
 819         uint64_t first_l1 = start >> epbs;
 820         uint64_t last_l1 = end >> epbs;

 822         if (end > dn->dn_maxblkid && (end != DMU_SPILL_BLKID)) {
 823                 end = dn->dn_maxblkid;
 824                 last_l1 = end >> epbs;
 825         }
 826         dprintf_dnode(dn, "start=%llu end=%llu\n", start, end);

 828         mutex_enter(&dn->dn_dbufs_mtx);
 829         if (start >= dn->dn_unlisted_l0_blkid * dn->dn_datablksz) {
 830                 /* There can't be any dbufs in this range; no need to search. */
 831                 mutex_exit(&dn->dn_dbufs_mtx);
 832                 return;
 833         } else if (dmu_objset_is_receiving(dn->dn_objset)) {
 834                 /*
 835                  * If we are receiving, we expect there to be no dbufs in
 836                  * the range to be freed, because receive modifies each
 837                  * block at most once, and in offset order.  If this is
 838                  * not the case, it can lead to performance problems,
 839                  * so note that we unexpectedly took the slow path.
 840                  */
 841                 atomic_inc_64(&zfs_free_range_recv_miss);
 842         }

 844         for (db = list_head(&dn->dn_dbufs); db != NULL; db = db_next) {
 844         for (db = list_head(&dn->dn_dbufs); db; db = db_next) {
 845                 db_next = list_next(&dn->dn_dbufs, db);
 846                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);

 848                 if (db->db_level == 1 &&
 849                     db->db_blkid >= first_l1 && db->db_blkid <= last_l1) {
 850                         mutex_enter(&db->db_mtx);
 851                         if (db->db_last_dirty &&
 852                             db->db_last_dirty->dr_txg < txg) {
 853                                 dbuf_add_ref(db, FTAG);
 854                                 mutex_exit(&db->db_mtx);
 855                                 dbuf_will_dirty(db, tx);
 856                                 dbuf_rele(db, FTAG);
 857                         } else {
```



```
 858                                 mutex_exit(&db->db_mtx);
 859                         }
 860                 }

 862                 if (db->db_level != 0)
 863                         continue;
 864                 dprintf_dbuf(db, "found buf %s\n", "");
 865                 if (db->db_blkid < start || db->db_blkid > end)
 866                         continue;

 868                 /* found a level 0 buffer in the range */
 869                 mutex_enter(&db->db_mtx);
 870                 if (dbuf_undirty(db, tx)) {
 871                         /* mutex has been dropped and dbuf destroyed */
 872                         continue;
 873                 }

 875                 if (db->db_state == DB_UNCACHED ||
 876                     db->db_state == DB_NOFILL ||
 877                     db->db_state == DB_EVICTING) {
 878                         ASSERT(db->db.db_data == NULL);
 879                         mutex_exit(&db->db_mtx);
 880                         continue;
 881                 }
 882                 if (db->db_state == DB_READ || db->db_state == DB_FILL) {
 883                         /* will be handled in dbuf_read_done or dbuf_rele */
 884                         db->db_freed_in_flight = TRUE;
 885                         mutex_exit(&db->db_mtx);
 886                         continue;
 887                 }
 888                 if (refcount_count(&db->db_holds) == 0) {
 889                         ASSERT(db->db_buf);
 890                         dbuf_clear(db);
 891                         continue;
 892                 }
 893                 /* The dbuf is referenced */

 895                 if (db->db_last_dirty != NULL) {
 896                         dbuf_dirty_record_t *dr = db->db_last_dirty;

 898                         if (dr->dr_txg == txg) {
 899                                 /*
 900                                  * This buffer is "in-use", re-adjust the file
 901                                  * size to reflect that this buffer may
 902                                  * contain new data when we sync.
 903                                  */
 904                                 if (db->db_blkid != DMU_SPILL_BLKID &&
 905                                     db->db_blkid > dn->dn_maxblkid)
 906                                         dn->dn_maxblkid = db->db_blkid;
 907                                 dbuf_unoverride(dr);
 908                         } else {
 909                                 /*
 910                                  * This dbuf is not dirty in the open context.
 911                                  * Either uncache it (if its not referenced in
 912                                  * the open context) or reset its contents to
 913                                  * empty.
 914                                  */
 915                                 dbuf_fix_old_data(db, txg);
 916                         }
 917                 }
 918                 /* clear the contents if its cached */
 919                 if (db->db_state == DB_CACHED) {
 920                         ASSERT(db->db.db_data != NULL);
 921                         arc_release(db->db_buf, db);
 922                         bzero(db->db.db_data, db->db.db_size);
 923                         arc_buf_freeze(db->db_buf);
```

```
 924                 }

 926                 mutex_exit(&db->db_mtx);
 927         }
 928         mutex_exit(&dn->dn_dbufs_mtx);
 929 }
_____unchanged_portion_omitted_

1028 dbuf_dirty_record_t *
1029 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1030 {
1031         dnode_t *dn;
1032         objset_t *os;
1033         dbuf_dirty_record_t **drp, *dr;
1034         int drop_struct_lock = FALSE;
1035         boolean_t do_free_accounting = B_FALSE;
1036         int txgoff = tx->tx_txg & TXG_MASK;

1038         ASSERT(tx->tx_txg != 0);
1039         ASSERT(!refcount_is_zero(&db->db_holds));
1040         DMU_TX_DIRTY_BUF(tx, db);

1042         DB_DNODE_ENTER(db);
1043         dn = DB_DNODE(db);
1044         /*
1045          * Shouldn't dirty a regular buffer in syncing context.  Private
1046          * objects may be dirtied in syncing context, but only if they
1047          * were already pre-dirtied in open context.
1048          */
1049         ASSERT(!dmu_tx_is_syncing(tx) ||
1050             BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1051             DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1052             dn->dn_objset->os_dsl_dataset == NULL);
1053         /*
1054          * We make this assert for private objects as well, but after we
1055          * check if we're already dirty.  They are allowed to re-dirty
1056          * in syncing context.
1057          */
1058         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1059             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1060             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1062         mutex_enter(&db->db_mtx);
1063         /*
1064          * XXX make this true for indirects too?  The problem is that
1065          * transactions created with dmu_tx_create_assigned() from
1066          * syncing context don't bother holding ahead.
1067          */
1068         ASSERT(db->db_level != 0 ||
1069             db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1070             db->db_state == DB_NOFILL);

1072         mutex_enter(&dn->dn_mtx);
1073         /*
1074          * Don't set dirtyctx to SYNC if we're just modifying this as we
1075          * initialize the objset.
1076          */
1077         if (dn->dn_dirtyctx == DN_UNDIRTIED &&
1078             !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1079                 dn->dn_dirtyctx =
1080                     (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1081                 ASSERT(dn->dn_dirtyctx_firstset == NULL);
1082                 dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1083         }
1084         mutex_exit(&dn->dn_mtx);
```



```
1086         if (db->db_blkid == DMU_SPILL_BLKID)
1087                 dn->dn_have_spill = B_TRUE;

1089         /*
1090          * If this buffer is already dirty, we're done.
1091          */
1092         drp = &db->db_last_dirty;
1093         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1094             db->db.db_object == DMU_META_DNODE_OBJECT);
1095         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1096                 drp = &dr->dr_next;
1097         if (dr && dr->dr_txg == tx->tx_txg) {
1098                 DB_DNODE_EXIT(db);

1100                 if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1101                         /*
1102                          * If this buffer has already been written out,
1103                          * we now need to reset its state.
1104                          */
1105                         dbuf_unoverride(dr);
1106                         if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1107                             db->db_state != DB_NOFILL)
1108                                 arc_buf_thaw(db->db_buf);
1109                 }
1110                 mutex_exit(&db->db_mtx);
1111                 return (dr);
1112         }

1114         /*
1115          * Only valid if not already dirty.
1116          */
1117         ASSERT(dn->dn_object == 0 ||
1118             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1119             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1121         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1122         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1123             dn->dn_phys->dn_nlevels > db->db_level ||
1124             dn->dn_next_nlevels[txgoff] > db->db_level ||
1125             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1126             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);

1128         /*
1129          * We should only be dirtying in syncing context if it's the
1130          * mos or we're initializing the os or it's a special object.
1131          * However, we are allowed to dirty in syncing context provided
1132          * we already dirtied it in open context.  Hence we must make
1133          * this assertion only if we're not already dirty.
1134          */
1135         os = dn->dn_objset;
1136         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1137             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1138         ASSERT(db->db.db_size != 0);

1140         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1142         if (db->db_blkid != DMU_BONUS_BLKID) {
1143                 /*
1144                  * Update the accounting.
1145                  * Note: we delay "free accounting" until after we drop
1146                  * the db_mtx.  This keeps us from grabbing other locks
1147                  * (and possibly deadlocking) in bp_get_dsize() while
1148                  * also holding the db_mtx.
1149                  */
1150                 dnode_willuse_space(dn, db->db.db_size, tx);
1151                 do_free_accounting = dbuf_block_freeable(db);
```

```
1152         }

1154         /*
1155          * If this buffer is dirty in an old transaction group we need
1156          * to make a copy of it so that the changes we make in this
1157          * transaction group won't leak out when we sync the older txg.
1158          */
1159         dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1160         if (db->db_level == 0) {
1161                 void *data_old = db->db_buf;

1163                 if (db->db_state != DB_NOFILL) {
1164                         if (db->db_blkid == DMU_BONUS_BLKID) {
1165                                 dbuf_fix_old_data(db, tx->tx_txg);
1166                                 data_old = db->db.db_data;
1167                         } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1168                                 /*
1169                                  * Release the data buffer from the cache so
1170                                  * that we can modify it without impacting
1171                                  * possible other users of this cached data
1172                                  * block.  Note that indirect blocks and
1173                                  * private objects are not released until the
1174                                  * syncing state (since they are only modified
1175                                  * then).
1176                                  */
1177                                 arc_release(db->db_buf, db);
1178                                 dbuf_fix_old_data(db, tx->tx_txg);
1179                                 data_old = db->db_buf;
1180                         }
1181                         ASSERT(data_old != NULL);
1182                 }
1183                 dr->dt.dl.dr_data = data_old;
1184         } else {
1185                 mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1186                 list_create(&dr->dt.di.dr_children,
1187                     sizeof (dbuf_dirty_record_t),
1188                     offsetof(dbuf_dirty_record_t, dr_dirty_node));
1189         }
1190         if (db->db_blkid != DMU_BONUS_BLKID && os->os_dsl_dataset != NULL)
1191                 dr->dr_accounted = db->db.db_size;
1192         dr->dr_dbuf = db;
1193         dr->dr_txg = tx->tx_txg;
1194         dr->dr_next = *drp;
1195         *drp = dr;

1197         /*
1198          * We could have been freed_in_flight between the dbuf_noread
1199          * and dbuf_dirty.  We win, as though the dbuf_noread() had
1200          * happened after the free.
1201          */
1202         if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1203             db->db_blkid != DMU_SPILL_BLKID) {
1204                 mutex_enter(&dn->dn_mtx);
1205                 dnode_clear_range(dn, db->db_blkid, 1, tx);
1206                 mutex_exit(&dn->dn_mtx);
1207                 db->db_freed_in_flight = FALSE;
1208         }

1210         /*
1211          * This buffer is now part of this txg
1212          */
1213         dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1214         db->db_dirtycnt += 1;
1215         ASSERT3U(db->db_dirtycnt, <=, 3);

1217         mutex_exit(&db->db_mtx);
```



```
1219         if (db->db_blkid == DMU_BONUS_BLKID ||
1220             db->db_blkid == DMU_SPILL_BLKID) {
1221                 mutex_enter(&dn->dn_mtx);
1222                 ASSERT(!list_link_active(&dr->dr_dirty_node));
1223                 list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1224                 mutex_exit(&dn->dn_mtx);
1225                 dnode_setdirty(dn, tx);
1226                 DB_DNODE_EXIT(db);
1227                 return (dr);
1228         } else if (do_free_accounting) {
1229                 blkptr_t *bp = db->db_blkptr;
1230                 int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1231                     bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1232                 /*
1233                  * This is only a guess -- if the dbuf is dirty
1234                  * in a previous txg, we don't know how much
1235                  * space it will use on disk yet.  We should
1236                  * really have the struct_rwlock to access
1237                  * db_blkptr, but since this is just a guess,
1238                  * it's OK if we get an odd answer.
1239                  */
1240                 ddt_prefetch(os->os_spa, bp);
1241                 dnode_willuse_space(dn, -willfree, tx);
1242         }

1244         if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1245                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
1246                 drop_struct_lock = TRUE;
1247         }

1249         if (db->db_level == 0) {
1250                 dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1251                 ASSERT(dn->dn_maxblkid >= db->db_blkid);
1252         }

1254         if (db->db_level+1 < dn->dn_nlevels) {
1255                 dmu_buf_impl_t *parent = db->db_parent;
1256                 dbuf_dirty_record_t *di;
1257                 int parent_held = FALSE;

1259                 if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1260                         int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1262                         parent = dbuf_hold_level(dn, db->db_level+1,
1263                             db->db_blkid >> epbs, FTAG);
1264                         ASSERT(parent != NULL);
1265                         parent_held = TRUE;
1266                 }
1267                 if (drop_struct_lock)
1268                         rw_exit(&dn->dn_struct_rwlock);
1269                 ASSERT3U(db->db_level+1, ==, parent->db_level);
1270                 di = dbuf_dirty(parent, tx);
1271                 if (parent_held)
1272                         dbuf_rele(parent, FTAG);

1274                 mutex_enter(&db->db_mtx);
1275                 /*
1276                  * Since we've dropped the mutex, it's possible that
1277                  * dbuf_undirty() might have changed this out from under us.
1278                  */
1273                 /*  possible race with dbuf_undirty() */
1279                 if (db->db_last_dirty == dr ||
1280                     dn->dn_object == DMU_META_DNODE_OBJECT) {
1281                         mutex_enter(&di->dt.di.dr_mtx);
1282                         ASSERT3U(di->dr_txg, ==, tx->tx_txg);
```

```
1283                        ASSERT(!list_link_active(&dr->dr_dirty_node));
1284                        list_insert_tail(&di->dt.di.dr_children, dr);
1285                        mutex_exit(&di->dt.di.dr_mtx);
1286                        dr->dr_parent = di;
1287                }
1288                mutex_exit(&db->db_mtx);
1289        } else {
1290                ASSERT(db->db_level+1 == dn->dn_nlevels);
1291                ASSERT(db->db_blkid < dn->dn_nblkptr);
1292                ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1293                mutex_enter(&dn->dn_mtx);
1294                ASSERT(!list_link_active(&dr->dr_dirty_node));
1295                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1296                mutex_exit(&dn->dn_mtx);
1297                if (drop_struct_lock)
1298                        rw_exit(&dn->dn_struct_rwlock);
1299        }

1301        dnode_setdirty(dn, tx);
1302        DB_DNODE_EXIT(db);
1303        return (dr);
1304 }

1306 /*
1307  * Undirty a buffer in the transaction group referenced by the given
1308  * transaction.  Return whether this evicted the dbuf.
1309  */
1310 static boolean_t
1311 dbuf_undirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1312 {
1313        dnode_t *dn;
1314        uint64_t txg = tx->tx_txg;
1315        dbuf_dirty_record_t *dr, **drp;

1317        ASSERT(txg != 0);
1318        ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1319        ASSERT0(db->db_level);
1320        ASSERT(MUTEX_HELD(&db->db_mtx));

1322        /*
1323         * If this buffer is not dirty, we're done.
1324         */
1325        for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
1326                if (dr->dr_txg <= txg)
1327                        break;
1328        if (dr == NULL || dr->dr_txg < txg)
1329                return (B_FALSE);
1330        ASSERT(dr->dr_txg == txg);
1331        ASSERT(dr->dr_dbuf == db);

1333        DB_DNODE_ENTER(db);
1334        dn = DB_DNODE(db);

1336        /*
1337         * Note:  This code will probably work even if there are concurrent
1338         * holders, but it is untested in that scenerio, as the ZPL and
1339         * ztest have additional locking (the range locks) that prevents
1340         * that type of concurrent access.
1341         */
1342        ASSERT3U(refcount_count(&db->db_holds), ==, db->db_dirtycnt);

1344        dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1346        ASSERT(db->db.db_size != 0);

1348        /*
```



```
1349         * Any space we accounted for in dp_dirty_* will be cleaned up by
1350         * dsl_pool_sync().  This is relatively rare so the discrepancy
1351         * is not a big deal.
1352         */
1343        /* XXX would be nice to fix up dn_towrite_space[] */

1354        *drp = dr->dr_next;

1356        /*
1357         * Note that there are three places in dbuf_dirty()
1358         * where this dirty record may be put on a list.
1359         * Make sure to do a list_remove corresponding to
1360         * every one of those list_insert calls.
1361         */
1362        if (dr->dr_parent) {
1363                mutex_enter(&dr->dr_parent->dt.di.dr_mtx);
1364                list_remove(&dr->dr_parent->dt.di.dr_children, dr);
1365                mutex_exit(&dr->dr_parent->dt.di.dr_mtx);
1366        } else if (db->db_blkid == DMU_SPILL_BLKID ||
1367            db->db_level+1 == dn->dn_nlevels) {
1368                ASSERT(db->db_blkptr == NULL || db->db_parent == dn->dn_dbuf);
1369                mutex_enter(&dn->dn_mtx);
1370                list_remove(&dn->dn_dirty_records[txg & TXG_MASK], dr);
1371                mutex_exit(&dn->dn_mtx);
1372        }
1373        DB_DNODE_EXIT(db);

1375        if (db->db_state != DB_NOFILL) {
1376                dbuf_unoverride(dr);

1378                ASSERT(db->db_buf != NULL);
1379                ASSERT(dr->dt.dl.dr_data != NULL);
1380                if (dr->dt.dl.dr_data != db->db_buf)
1381                        VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data, db));
1382        }
1383        kmem_free(dr, sizeof (dbuf_dirty_record_t));

1385        ASSERT(db->db_dirtycnt > 0);
1386        db->db_dirtycnt -= 1;

1388        if (refcount_remove(&db->db_holds, (void *)(uintptr_t)txg) == 0) {
1389                arc_buf_t *buf = db->db_buf;

1391                ASSERT(db->db_state == DB_NOFILL || arc_released(buf));
1392                dbuf_set_data(db, NULL);
1393                VERIFY(arc_buf_remove_ref(buf, db));
1394                dbuf_evict(db);
1395                return (B_TRUE);
1396        }

1398        return (B_FALSE);
1399 }
```

*unchanged_portion_omitted_*

```
1530 /*
1531  * "Clear" the contents of this dbuf.  This will mark the dbuf
1532  * EVICTING and clear *most* of its references.  Unfortunately,
1523  * EVICTING and clear *most* of its references.  Unfortunetely,
1533  * when we are not holding the dn_dbufs_mtx, we can't clear the
1534  * entry in the dn_dbufs list.  We have to wait until dbuf_destroy()
1535  * in this case.  For callers from the DMU we will usually see:
1536  *      dbuf_clear()->arc_buf_evict()->dbuf_do_evict()->dbuf_destroy()
1537  * For the arc callback, we will usually see:
1538  *      dbuf_do_evict()->dbuf_clear();dbuf_destroy()
1539  * Sometimes, though, we will get a mix of these two:
1540  *      DMU: dbuf_clear()->arc_buf_evict()
```

```
1541  *      ARC: dbuf_do_evict()->dbuf_destroy()
1542  */
1543 void
1544 dbuf_clear(dmu_buf_impl_t *db)
1545 {
1546         dnode_t *dn;
1547         dmu_buf_impl_t *parent = db->db_parent;
1548         dmu_buf_impl_t *dndb;
1549         int dbuf_gone = FALSE;

1551         ASSERT(MUTEX_HELD(&db->db_mtx));
1552         ASSERT(refcount_is_zero(&db->db_holds));

1554         dbuf_evict_user(db);

1556         if (db->db_state == DB_CACHED) {
1557                 ASSERT(db->db.db_data != NULL);
1558                 if (db->db_blkid == DMU_BONUS_BLKID) {
1559                         zio_buf_free(db->db.db_data, DN_MAX_BONUSLEN);
1560                         arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
1561                 }
1562                 db->db.db_data = NULL;
1563                 db->db_state = DB_UNCACHED;
1564         }

1566         ASSERT(db->db_state == DB_UNCACHED || db->db_state == DB_NOFILL);
1567         ASSERT(db->db_data_pending == NULL);

1569         db->db_state = DB_EVICTING;
1570         db->db_blkptr = NULL;

1572         DB_DNODE_ENTER(db);
1573         dn = DB_DNODE(db);
1574         dndb = dn->dn_dbuf;
1575         if (db->db_blkid != DMU_BONUS_BLKID && MUTEX_HELD(&dn->dn_dbufs_mtx)) {
1576                 list_remove(&dn->dn_dbufs, db);
1577                 (void) atomic_dec_32_nv(&dn->dn_dbufs_count);
1578                 membar_producer();
1579                 DB_DNODE_EXIT(db);
1580                 /*
1581                  * Decrementing the dbuf count means that the hold corresponding
1582                  * to the removed dbuf is no longer discounted in dnode_move(),
1583                  * so the dnode cannot be moved until after we release the hold.
1584                  * The membar_producer() ensures visibility of the decremented
1585                  * value in dnode_move(), since DB_DNODE_EXIT doesn't actually
1586                  * release any lock.
1587                  */
1588                 dnode_rele(dn, db);
1589                 db->db_dnode_handle = NULL;
1590         } else {
1591                 DB_DNODE_EXIT(db);
1592         }

1594         if (db->db_buf)
1595                 dbuf_gone = arc_buf_evict(db->db_buf);

1597         if (!dbuf_gone)
1598                 mutex_exit(&db->db_mtx);

1600         /*
1601          * If this dbuf is referenced from an indirect dbuf,
1602          * decrement the ref count on the indirect dbuf.
1603          */
1604         if (parent && parent != dndb)
1605                 dbuf_rele(parent, db);
1606 }
```

unchanged_portion_omitted_


```
1675 static dmu_buf_impl_t *
1676 dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
1677     dmu_buf_impl_t *parent, blkptr_t *blkptr)
1678 {
1679         objset_t *os = dn->dn_objset;
1680         dmu_buf_impl_t *db, *odb;

1682         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));
1683         ASSERT(dn->dn_type != DMU_OT_NONE);

1685         db = kmem_cache_alloc(dbuf_cache, KM_SLEEP);

1687         db->db_objset = os;
1688         db->db.db_object = dn->dn_object;
1689         db->db_level = level;
1690         db->db_blkid = blkid;
1691         db->db_last_dirty = NULL;
1692         db->db_dirtycnt = 0;
1693         db->db_dnode_handle = dn->dn_handle;
1694         db->db_parent = parent;
1695         db->db_blkptr = blkptr;

1697         db->db_user_ptr = NULL;
1698         db->db_user_data_ptr_ptr = NULL;
1699         db->db_evict_func = NULL;
1700         db->db_immediate_evict = 0;
1701         db->db_freed_in_flight = 0;

1703         if (blkid == DMU_BONUS_BLKID) {
1704                 ASSERT3P(parent, ==, dn->dn_dbuf);
1705                 db->db.db_size = DN_MAX_BONUSLEN -
1706                     (dn->dn_nblkptr-1) * sizeof (blkptr_t);
1707                 ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
1708                 db->db.db_offset = DMU_BONUS_BLKID;
1709                 db->db_state = DB_UNCACHED;
1710                 /* the bonus dbuf is not placed in the hash table */
1711                 arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);
1712                 return (db);
1713         } else if (blkid == DMU_SPILL_BLKID) {
1714                 db->db.db_size = (blkptr != NULL) ?
1715                     BP_GET_LSIZE(blkptr) : SPA_MINBLOCKSIZE;
1716                 db->db.db_offset = 0;
1717         } else {
1718                 int blocksize =
1719                     db->db_level ? 1 << dn->dn_indblkshift : dn->dn_datablksz;
1710                     db->db_level ? 1<<dn->dn_indblkshift :  dn->dn_datablksz;
1720                 db->db.db_size = blocksize;
1721                 db->db.db_offset = db->db_blkid * blocksize;
1722         }

1724         /*
1725          * Hold the dn_dbufs_mtx while we get the new dbuf
1726          * in the hash table *and* added to the dbufs list.
1727          * This prevents a possible deadlock with someone
1728          * trying to look up this dbuf before its added to the
1729          * dn_dbufs list.
1730          */
1731         mutex_enter(&dn->dn_dbufs_mtx);
1732         db->db_state = DB_EVICTING;
1733         if ((odb = dbuf_hash_insert(db)) != NULL) {
1734                 /* someone else inserted it first */
1735                 kmem_cache_free(dbuf_cache, db);
1736                 mutex_exit(&dn->dn_dbufs_mtx);
1737                 return (odb);
1738         }
```

```
1739         list_insert_head(&dn->dn_dbufs, db);
1740         if (db->db_level == 0 && db->db_blkid >=
1741             dn->dn_unlisted_l0_blkid)
1742                 dn->dn_unlisted_l0_blkid = db->db_blkid + 1;
1743         db->db_state = DB_UNCACHED;
1744         mutex_exit(&dn->dn_dbufs_mtx);
1745         arc_space_consume(sizeof (dmu_buf_impl_t), ARC_SPACE_OTHER);

1747         if (parent && parent != dn->dn_dbuf)
1748                 dbuf_add_ref(parent, db);

1750         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1751             refcount_count(&dn->dn_holds) > 0);
1752         (void) refcount_add(&dn->dn_holds, db);
1753         (void) atomic_inc_32_nv(&dn->dn_dbufs_count);

1755         dprintf_dbuf(db, "db=%p\n", db);

1757         return (db);
1758 }
```

*unchanged_portion_omitted_*

```
1827 void
1828 dbuf_prefetch(dnode_t *dn, uint64_t blkid, zio_priority_t prio)
1829 {
1830         dmu_buf_impl_t *db = NULL;
1831         blkptr_t *bp = NULL;

1833         ASSERT(blkid != DMU_BONUS_BLKID);
1834         ASSERT(RW_LOCK_HELD(&dn->dn_struct_rwlock));

1836         if (dnode_block_freed(dn, blkid))
1837                 return;

1839         /* dbuf_find() returns with db_mtx held */
1840         if (db = dbuf_find(dn, 0, blkid)) {
1841                 /*
1842                  * This dbuf is already in the cache.  We assume that
1843                  * it is already CACHED, or else about to be either
1844                  * read or filled.
1845                  */
1846                 mutex_exit(&db->db_mtx);
1847                 return;
1848         }

1850         if (dbuf_findbp(dn, 0, blkid, TRUE, &db, &bp) == 0) {
1851                 if (bp && !BP_IS_HOLE(bp)) {
1843                         int priority = dn->dn_type == DMU_OT_DDT_ZAP ?
1844                             ZIO_PRIORITY_DDT_PREFETCH : ZIO_PRIORITY_ASYNC_READ;
1852                         dsl_dataset_t *ds = dn->dn_objset->os_dsl_dataset;
1853                         uint32_t aflags = ARC_NOWAIT | ARC_PREFETCH;
1854                         zbookmark_t zb;

1856                         SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
1857                             dn->dn_object, 0, blkid);

1859                         (void) arc_read(NULL, dn->dn_objset->os_spa,
1860                             bp, NULL, NULL, prio,
1853                             bp, NULL, NULL, priority,
1861                             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
1862                             &aflags, &zb);
1863                 }
1864                 if (db)
1865                         dbuf_rele(db, NULL);
1866         }
```



```
1867 }
```

*unchanged_portion_omitted_*

```
2541 /*
2542  * The SPA will call this callback several times for each zio - once
2543  * for every physical child i/o (zio->io_phys_children times).  This
2544  * allows the DMU to monitor the progress of each logical i/o.  For example,
2545  * there may be 2 copies of an indirect block, or many fragments of a RAID-Z
2546  * block.  There may be a long delay before all copies/fragments are completed,
2547  * so this callback allows us to retire dirty space gradually, as the physical
2548  * i/os complete.
2549  */
2550 /* ARGSUSED */
2551 static void
2552 dbuf_write_physdone(zio_t *zio, arc_buf_t *buf, void *arg)
2553 {
2554         dmu_buf_impl_t *db = arg;
2555         objset_t *os = db->db_objset;
2556         dsl_pool_t *dp = dmu_objset_pool(os);
2557         dbuf_dirty_record_t *dr;
2558         int delta = 0;

2560         dr = db->db_data_pending;
2561         ASSERT3U(dr->dr_txg, ==, zio->io_txg);

2563         /*
2564          * The callback will be called io_phys_children times.  Retire one
2565          * portion of our dirty space each time we are called.  Any rounding
2566          * error will be cleaned up by dsl_pool_sync()'s call to
2567          * dsl_pool_undirty_space().
2568          */
2569         delta = dr->dr_accounted / zio->io_phys_children;
2570         dsl_pool_undirty_space(dp, delta, zio->io_txg);
2571 }

2573 /* ARGSUSED */
2574 static void
2575 dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
2576 {
2577         dmu_buf_impl_t *db = vdb;
2578         blkptr_t *bp = zio->io_bp;
2579         blkptr_t *bp_orig = &zio->io_bp_orig;
2580         uint64_t txg = zio->io_txg;
2581         dbuf_dirty_record_t **drp, *dr;

2583         ASSERT0(zio->io_error);
2584         ASSERT(db->db_blkptr == bp);

2586         /*
2587          * For nopwrites and rewrites we ensure that the bp matches our
2588          * original and bypass all the accounting.
2589          */
2590         if (zio->io_flags & (ZIO_FLAG_IO_REWRITE | ZIO_FLAG_NOPWRITE)) {
2591                 ASSERT(BP_EQUAL(bp, bp_orig));
2592         } else {
2593                 objset_t *os;
2594                 dsl_dataset_t *ds;
2595                 dmu_tx_t *tx;

2597                 DB_GET_OBJSET(&os, db);
2598                 ds = os->os_dsl_dataset;
2599                 tx = os->os_synctx;

2601                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
2602                 dsl_dataset_block_born(ds, bp, tx);
2603         }
```

```
2605         mutex_enter(&db->db_mtx);

2607         DBUF_VERIFY(db);

2609         drp = &db->db_last_dirty;
2610         while ((dr = *drp) != db->db_data_pending)
2611                 drp = &dr->dr_next;
2612         ASSERT(!list_link_active(&dr->dr_dirty_node));
2613         ASSERT(dr->dr_txg == txg);
2614         ASSERT(dr->dr_dbuf == db);
2615         ASSERT(dr->dr_next == NULL);
2616         *drp = dr->dr_next;

2618 #ifdef ZFS_DEBUG
2619         if (db->db_blkid == DMU_SPILL_BLKID) {
2620                 dnode_t *dn;

2622                 DB_DNODE_ENTER(db);
2623                 dn = DB_DNODE(db);
2624                 ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2625                 ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2626                     db->db_blkptr == &dn->dn_phys->dn_spill);
2627                 DB_DNODE_EXIT(db);
2628         }
2629 #endif

2631         if (db->db_level == 0) {
2632                 ASSERT(db->db_blkid != DMU_BONUS_BLKID);
2633                 ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
2634                 if (db->db_state != DB_NOFILL) {
2635                         if (dr->dt.dl.dr_data != db->db_buf)
2636                                 VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
2637                                     db));
2638                         else if (!arc_released(db->db_buf))
2639                                 arc_set_callback(db->db_buf, dbuf_do_evict, db);
2640                 }
2641         } else {
2642                 dnode_t *dn;

2644                 DB_DNODE_ENTER(db);
2645                 dn = DB_DNODE(db);
2646                 ASSERT(list_head(&dr->dt.di.dr_children) == NULL);
2647                 ASSERT3U(db->db.db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2648                 if (!BP_IS_HOLE(db->db_blkptr)) {
2649                         int epbs =
2650                             dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
2651                         ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
2652                             db->db.db_size);
2653                         ASSERT3U(dn->dn_phys->dn_maxblkid
2654                             >> (db->db_level * epbs), >=, db->db_blkid);
2655                         arc_set_callback(db->db_buf, dbuf_do_evict, db);
2656                 }
2657                 DB_DNODE_EXIT(db);
2658                 mutex_destroy(&dr->dt.di.dr_mtx);
2659                 list_destroy(&dr->dt.di.dr_children);
2660         }
2661         kmem_free(dr, sizeof (dbuf_dirty_record_t));

2663         cv_broadcast(&db->db_changed);
2664         ASSERT(db->db_dirtycnt > 0);
2665         db->db_dirtycnt -= 1;
2666         db->db_data_pending = NULL;

2668         dbuf_rele_and_unlock(db, (void *)(uintptr_t)txg);
2669 }
```

unchanged_portion_omitted_


```
2710 /* Issue I/O to commit a dirty buffer to disk. */
2711 static void
2712 dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
2713 {
2714         dmu_buf_impl_t *db = dr->dr_dbuf;
2715         dnode_t *dn;
2716         objset_t *os;
2717         dmu_buf_impl_t *parent = db->db_parent;
2718         uint64_t txg = tx->tx_txg;
2719         zbookmark_t zb;
2720         zio_prop_t zp;
2721         zio_t *zio;
2722         int wp_flag = 0;

2724         DB_DNODE_ENTER(db);
2725         dn = DB_DNODE(db);
2726         os = dn->dn_objset;

2728         if (db->db_state != DB_NOFILL) {
2729                 if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
2730                         /*
2731                          * Private object buffers are released here rather
2732                          * than in dbuf_dirty() since they are only modified
2733                          * in the syncing context and we don't want the
2734                          * overhead of making multiple copies of the data.
2735                          */
2736                         if (BP_IS_HOLE(db->db_blkptr)) {
2737                                 arc_buf_thaw(data);
2738                         } else {
2739                                 dbuf_release_bp(db);
2740                         }
2741                 }
2742         }

2744         if (parent != dn->dn_dbuf) {
2745                 /* Our parent is an indirect block. */
2746                 /* We have a dirty parent that has been scheduled for write. */
2747                 ASSERT(parent && parent->db_data_pending);
2748                 /* Our parent's buffer is one level closer to the dnode. */
2749                 ASSERT(db->db_level == parent->db_level-1);
2750                 /*
2751                  * We're about to modify our parent's db_data by modifying
2752                  * our block pointer, so the parent must be released.
2753                  */
2754                 ASSERT(arc_released(parent->db_buf));
2755                 zio = parent->db_data_pending->dr_zio;
2756         } else {
2757                 /* Our parent is the dnode itself. */
2758                 ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
2759                     db->db_blkid != DMU_SPILL_BLKID) ||
2760                     (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
2761                 if (db->db_blkid != DMU_SPILL_BLKID)
2762                         ASSERT3P(db->db_blkptr, ==,
2763                             &dn->dn_phys->dn_blkptr[db->db_blkid]);
2764                 zio = dn->dn_zio;
2765         }

2767         ASSERT(db->db_level == 0 || data == db->db_buf);
2768         ASSERT3U(db->db_blkptr->blk_birth, <=, txg);
2769         ASSERT(zio);

2771         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
2772             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
2773             db->db.db_object, db->db_level, db->db_blkid);
```

```
2775         if (db->db_blkid == DMU_SPILL_BLKID)
2776                 wp_flag = WP_SPILL;
2777         wp_flag |= (db->db_state == DB_NOFILL) ? WP_NOFILL : 0;

2779         dmu_write_policy(os, dn, db->db_level, wp_flag, &zp);
2780         DB_DNODE_EXIT(db);

2782         if (db->db_level == 0 && dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
2783                 ASSERT(db->db_state != DB_NOFILL);
2784                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2785                     db->db_blkptr, data->b_data, arc_buf_size(data), &zp,
2786                     dbuf_write_override_ready, NULL, dbuf_write_override_done,
2787                     dr, ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2746                     dbuf_write_override_ready, dbuf_write_override_done, dr,
2747                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2788                 mutex_enter(&db->db_mtx);
2789                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
2790                 zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
2791                     dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
2792                 mutex_exit(&db->db_mtx);
2793         } else if (db->db_state == DB_NOFILL) {
2794                 ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF);
2795                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2796                     db->db_blkptr, NULL, db->db.db_size, &zp,
2797                     dbuf_write_nofill_ready, NULL, dbuf_write_nofill_done, db,
2757                     dbuf_write_nofill_ready, dbuf_write_nofill_done, db,
2798                     ZIO_PRIORITY_ASYNC_WRITE,
2799                     ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
2800         } else {
2801                 ASSERT(arc_released(data));
2802                 dr->dr_zio = arc_write(zio, os->os_spa, txg,
2803                     db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db),
2804                     DBUF_IS_L2COMPRESSIBLE(db), &zp, dbuf_write_ready,
2805                     dbuf_write_physdone, dbuf_write_done, db,
2806                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2765                     dbuf_write_done, db, ZIO_PRIORITY_ASYNC_WRITE,
2766                     ZIO_FLAG_MUSTSUCCEED, &zb);
2807         }
2808 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   44460 Thu Aug 22 16:15:06 2013
new/usr/src/uts/common/fs/zfs/dmu.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 364 /*
 365  * Note: longer-term, we should modify all of the dmu_buf_*() interfaces
 366  * to take a held dnode rather than <os, object> -- the lookup is wasteful,
 367  * and can induce severe lock contention when writing to several files
 368  * whose dnodes are in the same block.
 369  */
 370 static int
 371 dmu_buf_hold_array_by_dnode(dnode_t *dn, uint64_t offset, uint64_t length,
 372     int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp, uint32_t flags)
 373 {
 374         dsl_pool_t *dp = NULL;
 374         dmu_buf_t **dbp;
 375         uint64_t blkid, nblks, i;
 376         uint32_t dbuf_flags;
 377         int err;
 378         zio_t *zio;
 380         hrtime_t start;

 380         ASSERT(length <= DMU_MAX_ACCESS);

 382         dbuf_flags = DB_RF_CANFAIL | DB_RF_NEVERWAIT | DB_RF_HAVESTRUCT;
 383         if (flags & DMU_READ_NO_PREFETCH || length > zfetch_array_rd_sz)
 384                 dbuf_flags |= DB_RF_NOPREFETCH;

 386         rw_enter(&dn->dn_struct_rwlock, RW_READER);
 387         if (dn->dn_datablkshift) {
 388                 int blkshift = dn->dn_datablkshift;
 389                 nblks = (P2ROUNDUP(offset+length, 1ULL<<blkshift) -
 390                     P2ALIGN(offset, 1ULL<<blkshift)) >> blkshift;
 391         } else {
 392                 if (offset + length > dn->dn_datablksz) {
 393                         zfs_panic_recover("zfs: accessing past end of object "
 394                             "%llx/%llx (size=%u access=%llu+%llu)",
 395                             (longlong_t)dn->dn_objset->
 396                             os_dsl_dataset->ds_object,
 397                             (longlong_t)dn->dn_object, dn->dn_datablksz,
 398                             (longlong_t)offset, (longlong_t)length);
 399                         rw_exit(&dn->dn_struct_rwlock);
 400                         return (SET_ERROR(EIO));
 401                 }
 402                 nblks = 1;
 403         }
 404         dbp = kmem_zalloc(sizeof (dmu_buf_t *) * nblks, KM_SLEEP);

 408         if (dn->dn_objset->os_dsl_dataset)
 409                 dp = dn->dn_objset->os_dsl_dataset->ds_dir->dd_pool;
 410         start = gethrtime();
 406         zio = zio_root(dn->dn_objset->os_spa, NULL, NULL, ZIO_FLAG_CANFAIL);
 407         blkid = dbuf_whichblock(dn, offset);
 408         for (i = 0; i < nblks; i++) {
 409                 dmu_buf_impl_t *db = dbuf_hold(dn, blkid+i, tag);
 410                 if (db == NULL) {
 411                         rw_exit(&dn->dn_struct_rwlock);
 412                         dmu_buf_rele_array(dbp, nblks, tag);
 413                         zio_nowait(zio);
 414                         return (SET_ERROR(EIO));
```



```
 415                 }
 416                 /* initiate async i/o */
 417                 if (read) {
 418                         (void) dbuf_read(db, zio, dbuf_flags);
 419                 }
 420                 dbp[i] = &db->db;
 421         }
 422         rw_exit(&dn->dn_struct_rwlock);

 424         /* wait for async i/o */
 425         err = zio_wait(zio);
 431         /* track read overhead when we are in sync context */
 432         if (dp && dsl_pool_sync_context(dp))
 433                 dp->dp_read_overhead += gethrtime() - start;
 426         if (err) {
 427                 dmu_buf_rele_array(dbp, nblks, tag);
 428                 return (err);
 429         }

 431         /* wait for other io to complete */
 432         if (read) {
 433                 for (i = 0; i < nblks; i++) {
 434                         dmu_buf_impl_t *db = (dmu_buf_impl_t *)dbp[i];
 435                         mutex_enter(&db->db_mtx);
 436                         while (db->db_state == DB_READ ||
 437                             db->db_state == DB_FILL)
 438                                 cv_wait(&db->db_changed, &db->db_mtx);
 439                         if (db->db_state == DB_UNCACHED)
 440                                 err = SET_ERROR(EIO);
 441                         mutex_exit(&db->db_mtx);
 442                         if (err) {
 443                                 dmu_buf_rele_array(dbp, nblks, tag);
 444                                 return (err);
 445                         }
 446                 }
 447         }

 449         *numbufsp = nblks;
 450         *dbpp = dbp;
 451         return (0);
 452 }
______unchanged_portion_omitted_

 507 /*
 508  * Issue prefetch i/os for the given blocks.
 509  *
 510  * Note: The assumption is that we *know* these blocks will be needed
 511  * almost immediately.  Therefore, the prefetch i/os will be issued at
 512  * ZIO_PRIORITY_SYNC_READ
 513  *
 514  * Note: indirect blocks and other metadata will be read synchronously,
 515  * causing this function to block if they are not already cached.
 516  */
 517 void
 518 dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset, uint64_t len)
 519 {
 520         dnode_t *dn;
 521         uint64_t blkid;
 522         int nblks, err;
 520         int nblks, i, err;

 524         if (zfs_prefetch_disable)
 525                 return;

 527         if (len == 0) {  /* they're interested in the bonus buffer */
 528                 dn = DMU_META_DNODE(os);
```

```
530                 if (object == 0 || object >= DN_MAX_OBJECT)
531                         return;

533                 rw_enter(&dn->dn_struct_rwlock, RW_READER);
534                 blkid = dbuf_whichblock(dn, object * sizeof (dnode_phys_t));
535                 dbuf_prefetch(dn, blkid, ZIO_PRIORITY_SYNC_READ);
533                 dbuf_prefetch(dn, blkid);
536                 rw_exit(&dn->dn_struct_rwlock);
537                 return;
538         }

540         /*
541          * XXX - Note, if the dnode for the requested object is not
542          * already cached, we will do a *synchronous* read in the
543          * dnode_hold() call.  The same is true for any indirects.
544          */
545         err = dnode_hold(os, object, FTAG, &dn);
546         if (err != 0)
547                 return;

549         rw_enter(&dn->dn_struct_rwlock, RW_READER);
550         if (dn->dn_datablkshift) {
551                 int blkshift = dn->dn_datablkshift;
552                 nblks = (P2ROUNDUP(offset + len, 1 << blkshift) -
553                     P2ALIGN(offset, 1 << blkshift)) >> blkshift;
550                 nblks = (P2ROUNDUP(offset+len, 1<<blkshift) -
551                     P2ALIGN(offset, 1<<blkshift)) >> blkshift;
554         } else {
555                 nblks = (offset < dn->dn_datablksz);
556         }

558         if (nblks != 0) {
559                 blkid = dbuf_whichblock(dn, offset);
560                 for (int i = 0; i < nblks; i++)
561                         dbuf_prefetch(dn, blkid + i, ZIO_PRIORITY_SYNC_READ);
558                 for (i = 0; i < nblks; i++)
559                         dbuf_prefetch(dn, blkid+i);
562         }

564         rw_exit(&dn->dn_struct_rwlock);

566         dnode_rele(dn, FTAG);
567 }
```

_____***unchanged_portion_omitted_***

```
1338 static int
1339 dmu_sync_late_arrival(zio_t *pio, objset_t *os, dmu_sync_cb_t *done, zgd_t *zgd,
1340     zio_prop_t *zp, zbookmark_t *zb)
1341 {
1342         dmu_sync_arg_t *dsa;
1343         dmu_tx_t *tx;

1345         tx = dmu_tx_create(os);
1346         dmu_tx_hold_space(tx, zgd->zgd_db->db_size);
1347         if (dmu_tx_assign(tx, TXG_WAIT) != 0) {
1348                 dmu_tx_abort(tx);
1349                 /* Make zl_get_data do txg_waited_synced() */
1350                 return (SET_ERROR(EIO));
1351         }

1353         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1354         dsa->dsa_dr = NULL;
1355         dsa->dsa_done = done;
1356         dsa->dsa_zgd = zgd;
1357         dsa->dsa_tx = tx;
```



```
1359         zio_nowait(zio_write(pio, os->os_spa, dmu_tx_get_txg(tx), zgd->zgd_bp,
1360             zgd->zgd_db->db_data, zgd->zgd_db->db_size, zp,
1361             dmu_sync_late_arrival_ready, NULL, dmu_sync_late_arrival_done, dsa,
1359             dmu_sync_late_arrival_ready, dmu_sync_late_arrival_done, dsa,
1362             ZIO_PRIORITY_SYNC_WRITE, ZIO_FLAG_CANFAIL, zb));

1364         return (0);
1365 }

1367 /*
1368  * Intent log support: sync the block associated with db to disk.
1369  * N.B. and XXX: the caller is responsible for making sure that the
1370  * data isn't changing while dmu_sync() is writing it.
1371  *
1372  * Return values:
1373  *
1374  *      EEXIST: this txg has already been synced, so there's nothing to do.
1375  *              The caller should not log the write.
1376  *
1377  *      ENOENT: the block was dbuf_free_range()'d, so there's nothing to do.
1378  *              The caller should not log the write.
1379  *
1380  *      EALREADY: this block is already in the process of being synced.
1381  *              The caller should track its progress (somehow).
1382  *
1383  *      EIO: could not do the I/O.
1384  *              The caller should do a txg_wait_synced().
1385  *
1386  *      0: the I/O has been initiated.
1387  *              The caller should log this blkptr in the done callback.
1388  *              It is possible that the I/O will fail, in which case
1389  *              the error will be reported to the done callback and
1390  *              propagated to pio from zio_done().
1391  */
1392 int
1393 dmu_sync(zio_t *pio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd)
1394 {
1395         blkptr_t *bp = zgd->zgd_bp;
1396         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zgd->zgd_db;
1397         objset_t *os = db->db_objset;
1398         dsl_dataset_t *ds = os->os_dsl_dataset;
1399         dbuf_dirty_record_t *dr;
1400         dmu_sync_arg_t *dsa;
1401         zbookmark_t zb;
1402         zio_prop_t zp;
1403         dnode_t *dn;

1405         ASSERT(pio != NULL);
1406         ASSERT(txg != 0);

1408         SET_BOOKMARK(&zb, ds->ds_object,
1409             db->db.db_object, db->db_level, db->db_blkid);

1411         DB_DNODE_ENTER(db);
1412         dn = DB_DNODE(db);
1413         dmu_write_policy(os, dn, db->db_level, WP_DMU_SYNC, &zp);
1414         DB_DNODE_EXIT(db);

1416         /*
1417          * If we're frozen (running ziltest), we always need to generate a bp.
1418          */
1419         if (txg > spa_freeze_txg(os->os_spa))
1420                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));

1422         /*
```

```
1423          * Grabbing db_mtx now provides a barrier between dbuf_sync_leaf()
1424          * and us.  If we determine that this txg is not yet syncing,
1425          * but it begins to sync a moment later, that's OK because the
1426          * sync thread will block in dbuf_sync_leaf() until we drop db_mtx.
1427          */
1428         mutex_enter(&db->db_mtx);

1430         if (txg <= spa_last_synced_txg(os->os_spa)) {
1431                 /*
1432                  * This txg has already synced.  There's nothing to do.
1433                  */
1434                 mutex_exit(&db->db_mtx);
1435                 return (SET_ERROR(EEXIST));
1436         }

1438         if (txg <= spa_syncing_txg(os->os_spa)) {
1439                 /*
1440                  * This txg is currently syncing, so we can't mess with
1441                  * the dirty record anymore; just write a new log block.
1442                  */
1443                 mutex_exit(&db->db_mtx);
1444                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));
1445         }

1447         dr = db->db_last_dirty;
1448         while (dr && dr->dr_txg != txg)
1449                 dr = dr->dr_next;

1451         if (dr == NULL) {
1452                 /*
1453                  * There's no dr for this dbuf, so it must have been freed.
1454                  * There's no need to log writes to freed blocks, so we're done.
1455                  */
1456                 mutex_exit(&db->db_mtx);
1457                 return (SET_ERROR(ENOENT));
1458         }

1460         ASSERT(dr->dr_next == NULL || dr->dr_next->dr_txg < txg);

1462         /*
1463          * Assume the on-disk data is X, the current syncing data is Y,
1464          * and the current in-memory data is Z (currently in dmu_sync).
1465          * X and Z are identical but Y is has been modified. Normally,
1466          * when X and Z are the same we will perform a nopwrite but if Y
1467          * is different we must disable nopwrite since the resulting write
1468          * of Y to disk can free the block containing X. If we allowed a
1469          * nopwrite to occur the block pointing to Z would reference a freed
1470          * block. Since this is a rare case we simplify this by disabling
1471          * nopwrite if the current dmu_sync-ing dbuf has been modified in
1472          * a previous transaction.
1473          */
1474         if (dr->dr_next)
1475                 zp.zp_nopwrite = B_FALSE;

1477         ASSERT(dr->dr_txg == txg);
1478         if (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC ||
1479             dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
1480                 /*
1481                  * We have already issued a sync write for this buffer,
1482                  * or this buffer has already been synced.  It could not
1483                  * have been dirtied since, or we would have cleared the state.
1484                  */
1485                 mutex_exit(&db->db_mtx);
1486                 return (SET_ERROR(EALREADY));
1487         }
```



```
1489         ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
1490         dr->dt.dl.dr_override_state = DR_IN_DMU_SYNC;
1491         mutex_exit(&db->db_mtx);

1493         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1494         dsa->dsa_dr = dr;
1495         dsa->dsa_done = done;
1496         dsa->dsa_zgd = zgd;
1497         dsa->dsa_tx = NULL;

1499         zio_nowait(arc_write(pio, os->os_spa, txg,
1500             bp, dr->dt.dl.dr_data, DBUF_IS_L2CACHEABLE(db),
1501             DBUF_IS_L2COMPRESSIBLE(db), &zp, dmu_sync_ready,
1502             NULL, dmu_sync_done, dsa, ZIO_PRIORITY_SYNC_WRITE,
1503             ZIO_FLAG_CANFAIL, &zb));
1499             DBUF_IS_L2COMPRESSIBLE(db), &zp, dmu_sync_ready, dmu_sync_done,
1500             dsa, ZIO_PRIORITY_SYNC_WRITE, ZIO_FLAG_CANFAIL, &zb));

1505         return (0);
1506 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   44057 Thu Aug 22 16:15:07 2013
new/usr/src/uts/common/fs/zfs/dmu_objset.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 991 /* called from dsl */
 992 void
 993 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
 994 {
 995         int txgoff;
 996         zbookmark_t zb;
 997         zio_prop_t zp;
 998         zio_t *zio;
 999         list_t *list;
1000         list_t *newlist = NULL;
1001         dbuf_dirty_record_t *dr;

1003         dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

1005         ASSERT(dmu_tx_is_syncing(tx));
1006         /* XXX the write_done callback should really give us the tx... */
1007         os->os_synctx = tx;

1009         if (os->os_dsl_dataset == NULL) {
1010                 /*
1011                  * This is the MOS.  If we have upgraded,
1012                  * spa_max_replication() could change, so reset
1013                  * os_copies here.
1014                  */
1015                 os->os_copies = spa_max_replication(os->os_spa);
1016         }

1018         /*
1019          * Create the root block IO
1020          */
1021         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1022             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1023             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1024         arc_release(os->os_phys_buf, &os->os_phys_buf);

1026         dmu_write_policy(os, NULL, 0, 0, &zp);

1028         zio = arc_write(pio, os->os_spa, tx->tx_txg,
1029             os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os),
1030             DMU_OS_IS_L2COMPRESSIBLE(os), &zp, dmu_objset_write_ready,
1031             NULL, dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
1031             dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
1032             ZIO_FLAG_MUSTSUCCEED, &zb);

1034         /*
1035          * Sync special dnodes - the parent IO for the sync is the root block
1036          */
1037         DMU_META_DNODE(os)->dn_zio = zio;
1038         dnode_sync(DMU_META_DNODE(os), tx);

1040         os->os_phys->os_flags = os->os_flags;

1042         if (DMU_USERUSED_DNODE(os) &&
1043             DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
1044                 DMU_USERUSED_DNODE(os)->dn_zio = zio;
1045                 dnode_sync(DMU_USERUSED_DNODE(os), tx);
```



```
1046                 DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
1047                 dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1048         }

1050         txgoff = tx->tx_txg & TXG_MASK;

1052         if (dmu_objset_userused_enabled(os)) {
1053                 newlist = &os->os_synced_dnodes;
1054                 /*
1055                  * We must create the list here because it uses the
1056                  * dn_dirty_link[] of this txg.
1057                  */
1058                 list_create(newlist, sizeof (dnode_t),
1059                     offsetof(dnode_t, dn_dirty_link[txgoff]));
1060         }

1062         dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1063         dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

1065         list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
1066         while (dr = list_head(list)) {
1067                 ASSERT0(dr->dr_dbuf->db_level);
1068                 list_remove(list, dr);
1069                 if (dr->dr_zio)
1070                         zio_nowait(dr->dr_zio);
1071         }
1072         /*
1073          * Free intent log blocks up to this tx.
1074          */
1075         zil_sync(os->os_zil, tx);
1076         os->os_phys->os_zil_header = os->os_zil_header;
1077         zio_nowait(zio);
1078 }
______unchanged_portion_omitted_
```

```
**********************************************************
   44546 Thu Aug 22 16:15:08 2013
new/usr/src/uts/common/fs/zfs/dmu_tx.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/dmu.h>
#include <sys/dmu_impl.h>
#include <sys/dbuf.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h> /* for dsl_dataset_block_freeable() */
#include <sys/dsl_dir.h> /* for dsl_dir_tempreserve_*() */
#include <sys/dsl_pool.h>
#include <sys/zap_impl.h> /* for fzap_default_block_shift */
#include <sys/spa.h>
#include <sys/sa.h>
#include <sys/sa_impl.h>
#include <sys/zfs_context.h>
#include <sys/varargs.h>

typedef void (*dmu_tx_hold_func_t)(dmu_tx_t *tx, struct dnode *dn,
    uint64_t arg1, uint64_t arg2);


dmu_tx_t *
dmu_tx_create_dd(dsl_dir_t *dd)
{
        dmu_tx_t *tx = kmem_zalloc(sizeof (dmu_tx_t), KM_SLEEP);
        tx->tx_dir = dd;
        if (dd != NULL)
                tx->tx_pool = dd->dd_pool;
        list_create(&tx->tx_holds, sizeof (dmu_tx_hold_t),
            offsetof(dmu_tx_hold_t, txh_node));
        list_create(&tx->tx_callbacks, sizeof (dmu_tx_callback_t),
            offsetof(dmu_tx_callback_t, dcb_node));
        tx->tx_start = gethrtime();
#ifdef ZFS_DEBUG
```



```
        refcount_create(&tx->tx_space_written);
        refcount_create(&tx->tx_space_freed);
#endif
        return (tx);
}
```

*unchanged_portion_omitted_*

```
void
dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off, uint64_t len)
{
        dmu_tx_hold_t *txh;
        dnode_t *dn;
        int err;
        zio_t *zio;

        ASSERT(tx->tx_txg == 0);

        txh = dmu_tx_hold_object_impl(tx, tx->tx_objset,
            object, THT_FREE, off, len);
        if (txh == NULL)
                return;
        dn = txh->txh_dnode;
        dmu_tx_count_dnode(txh);

        if (off >= (dn->dn_maxblkid+1) * dn->dn_datablksz)
                return;
        if (len == DMU_OBJECT_END)
                len = (dn->dn_maxblkid+1) * dn->dn_datablksz - off;

        dmu_tx_count_dnode(txh);

        /*
         * For i/o error checking, we read the first and last level-0
         * blocks if they are not aligned, and all the level-1 blocks.
         *
         * Note:  dbuf_free_range() assumes that we have not instantiated
         * any level-0 dbufs that will be completely freed.  Therefore we must
         * exercise care to not read or count the first and last blocks
         * if they are blocksize-aligned.
         */
        if (dn->dn_datablkshift == 0) {
                if (off != 0 || len < dn->dn_datablksz)
                        dmu_tx_count_write(txh, off, len);
        } else {
                /* first block will be modified if it is not aligned */
                if (!IS_P2ALIGNED(off, 1 << dn->dn_datablkshift))
                        dmu_tx_count_write(txh, off, 1);
                /* last block will be modified if it is not aligned */
                if (!IS_P2ALIGNED(off + len, 1 << dn->dn_datablkshift))
                        dmu_tx_count_write(txh, off+len, 1);
        }

        /*
         * Check level-1 blocks.
         */
        if (dn->dn_nlevels > 1) {
                int shift = dn->dn_datablkshift + dn->dn_indblkshift -
                    SPA_BLKPTRSHIFT;
                uint64_t start = off >> shift;
                uint64_t end = (off + len) >> shift;

                ASSERT(dn->dn_datablkshift != 0);
                ASSERT(dn->dn_indblkshift != 0);

                zio = zio_root(tx->tx_pool->dp_spa,
                    NULL, NULL, ZIO_FLAG_CANFAIL);
```

```
644                 for (uint64_t i = start; i <= end; i++) {
645                         uint64_t ibyte = i << shift;
646                         err = dnode_next_offset(dn, 0, &ibyte, 2, 1, 0);
647                         i = ibyte >> shift;
648                         if (err == ESRCH)
649                                 break;
650                         if (err) {
651                                 tx->tx_err = err;
652                                 return;
653                         }

655                         err = dmu_tx_check_ioerr(zio, dn, 1, i);
656                         if (err) {
657                                 tx->tx_err = err;
658                                 return;
659                         }
660                 }
661                 err = zio_wait(zio);
662                 if (err) {
663                         tx->tx_err = err;
664                         return;
665                 }
666         }

668         dmu_tx_count_free(txh, off, len);
669 }
```

*unchanged_portion_omitted_*

```
913 #endif

915 /*
916  * If we can't do 10 iops, something is wrong.  Let us go ahead
917  * and hit zfs_dirty_data_max.
918  */
919 hrtime_t zfs_delay_max_ns = MSEC2NSEC(100);
920 int zfs_delay_resolution_ns = 100 * 1000; /* 100 microseconds */

922 /*
923  * We delay transactions when we've determined that the backend storage
924  * isn't able to accommodate the rate of incoming writes.
925  *
926  * If there is already a transaction waiting, we delay relative to when
927  * that transaction finishes waiting.  This way the calculated min_time
928  * is independent of the number of threads concurrently executing
929  * transactions.
930  *
931  * If we are the only waiter, wait relative to when the transaction
932  * started, rather than the current time.  This credits the transaction for
933  * "time already served", e.g. reading indirect blocks.
934  *
935  * The minimum time for a transaction to take is calculated as:
936  *     min_time = scale * (dirty - min) / (max - dirty)
937  *     min_time is then capped at zfs_delay_max_ns.
938  *
939  * The delay has two degrees of freedom that can be adjusted via tunables.
940  * The percentage of dirty data at which we start to delay is defined by
941  * zfs_delay_min_dirty_percent. This should typically be at or above
942  * zfs_vdev_async_write_active_max_dirty_percent so that we only start to
943  * delay after writing at full speed has failed to keep up with the incoming
944  * write rate. The scale of the curve is defined by zfs_delay_scale. Roughly
945  * speaking, this variable determines the amount of delay at the midpoint of
946  * the curve.
947  *
948  * delay
949  *  10ms +-------------------------------------------------------------*+
950  *       |                                                             *|
951  *   9ms +                                                             *+
```



```
952  *       |                                                             *|
953  *   8ms +                                                             *+
954  *       |                                                            * |
955  *   7ms +                                                            * +
956  *       |                                                            * |
957  *   6ms +                                                            * +
958  *       |                                                            * |
959  *   5ms +                                                           *  +
960  *       |                                                           *  |
961  *   4ms +                                                           *  +
962  *       |                                                           *  |
963  *   3ms +                                                          *   +
964  *       |                                                          *   |
965  *   2ms +                                                (midpoint) *    +
966  *       |                                                     |    **     |
967  *   1ms +                                                     v ***       +
968  *       |             zfs_delay_scale ---------->     ********          |
969  *     0 +-----------------------------------------*********--------------+
970  *       0%                    <- zfs_dirty_data_max ->               100%
971  *
972  * Note that since the delay is added to the outstanding time remaining on the
973  * most recent transaction, the delay is effectively the inverse of IOPS.
974  * Here the midpoint of 500us translates to 2000 IOPS. The shape of the curve
975  * was chosen such that small changes in the amount of accumulated dirty data
976  * in the first 3/4 of the curve yield relatively small differences in the
977  * amount of delay.
978  *
979  * The effects can be easier to understand when the amount of delay is
980  * represented on a log scale:
981  *
982  * delay
983  * 100ms +-------------------------------------------------------------++
984  *       +                                                              +
985  *       |                                                              |
986  *       +                                                             *+
987  *  10ms +                                                             *+
988  *       +                                                            ** +
989  *       |                                                (midpoint)  **  |
990  *       +                                                     |    **    +
991  *   1ms +                                                     v ****     +
992  *       +             zfs_delay_scale ---------->        *****         +
993  *       |                                             ****             |
994  *       +                                         ****                 +
995  * 100us +                                       **                     +
996  *       +                                      *                       +
997  *       |                                     *                        |
998  *       +                                    *                         +
999  *  10us +                                    *                         +
1000 *       +                                                              +
1001 *       |                                                              |
1002 *       +                                                              +
1003 *       +--------------------------------------------------------------+
1004 *       0%                    <- zfs_dirty_data_max ->               100%
1005 *
1006 * Note here that only as the amount of dirty data approaches its limit does
1007 * the delay start to increase rapidly. The goal of a properly tuned system
1008 * should be to keep the amount of dirty data out of that range by first
1009 * ensuring that the appropriate limits are set for the I/O scheduler to reach
1010 * optimal throughput on the backend storage, and then by changing the value
1011 * of zfs_delay_scale to increase the steepness of the curve.
1012 */
1013 static void
1014 dmu_tx_delay(dmu_tx_t *tx, uint64_t dirty)
1015 {
1016         dsl_pool_t *dp = tx->tx_pool;
1017         uint64_t delay_min_bytes =
```

```
1018             zfs_dirty_data_max * zfs_delay_min_dirty_percent / 100;
1019         hrtime_t wakeup, min_tx_time, now;

1021         if (dirty <= delay_min_bytes)
1022                 return;

1024         /*
1025          * The caller has already waited until we are under the max.
1026          * We make them pass us the amount of dirty data so we don't
1027          * have to handle the case of it being >= the max, which could
1028          * cause a divide-by-zero if it's == the max.
1029          */
1030         ASSERT3U(dirty, <, zfs_dirty_data_max);

1032         now = gethrtime();
1033         min_tx_time = zfs_delay_scale *
1034             (dirty - delay_min_bytes) / (zfs_dirty_data_max - dirty);
1035         if (now > tx->tx_start + min_tx_time)
1036                 return;

1038         min_tx_time = MIN(min_tx_time, zfs_delay_max_ns);

1040         DTRACE_PROBE3(delay__mintime, dmu_tx_t *, tx, uint64_t, dirty,
1041             uint64_t, min_tx_time);

1043         mutex_enter(&dp->dp_lock);
1044         wakeup = MAX(tx->tx_start + min_tx_time,
1045             dp->dp_last_wakeup + min_tx_time);
1046         dp->dp_last_wakeup = wakeup;
1047         mutex_exit(&dp->dp_lock);

1049 #ifdef _KERNEL
1050         mutex_enter(&curthread->t_delay_lock);
1051         while (cv_timedwait_hires(&curthread->t_delay_cv,
1052             &curthread->t_delay_lock, wakeup, zfs_delay_resolution_ns,
1053             CALLOUT_FLAG_ABSOLUTE | CALLOUT_FLAG_ROUNDUP) > 0)
1054                 continue;
1055         mutex_exit(&curthread->t_delay_lock);
1056 #else
1057         hrtime_t delta = wakeup - gethrtime();
1058         struct timespec ts;
1059         ts.tv_sec = delta / NANOSEC;
1060         ts.tv_nsec = delta % NANOSEC;
1061         (void) nanosleep(&ts, NULL);
1062 #endif
1063 }

1065 static int
1066 dmu_tx_try_assign(dmu_tx_t *tx, txg_how_t txg_how)
1067 {
1068         dmu_tx_hold_t *txh;
1069         spa_t *spa = tx->tx_pool->dp_spa;
1070         uint64_t memory, asize, fsize, usize;
1071         uint64_t towrite, tofree, tooverwrite, tounref, tohold, fudge;

1073         ASSERT0(tx->tx_txg);

1075         if (tx->tx_err)
1076                 return (tx->tx_err);

1078         if (spa_suspended(spa)) {
1079                 /*
1080                  * If the user has indicated a blocking failure mode
1081                  * then return ERESTART which will block in dmu_tx_wait().
1082                  * Otherwise, return EIO so that an error can get
1083                  * propagated back to the VOP calls.
```



```
1084                  *
1085                  * Note that we always honor the txg_how flag regardless
1086                  * of the failuremode setting.
1087                  */
1088                 if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_CONTINUE &&
1089                     txg_how != TXG_WAIT)
1090                         return (SET_ERROR(EIO));

1092                 return (SET_ERROR(ERESTART));
1093         }

1095         if (!tx->tx_waited &&
1096             dsl_pool_need_dirty_delay(tx->tx_pool)) {
1097                 tx->tx_wait_dirty = B_TRUE;
1098                 return (SET_ERROR(ERESTART));
1099         }

1101         tx->tx_txg = txg_hold_open(tx->tx_pool, &tx->tx_txgh);
1102         tx->tx_needassign_txh = NULL;

1104         /*
1105          * NB: No error returns are allowed after txg_hold_open, but
1106          * before processing the dnode holds, due to the
1107          * dmu_tx_unassign() logic.
1108          */

1110         towrite = tofree = tooverwrite = tounref = tohold = fudge = 0;
1111         for (txh = list_head(&tx->tx_holds); txh;
1112             txh = list_next(&tx->tx_holds, txh)) {
1113                 dnode_t *dn = txh->txh_dnode;
1114                 if (dn != NULL) {
1115                         mutex_enter(&dn->dn_mtx);
1116                         if (dn->dn_assigned_txg == tx->tx_txg - 1) {
1117                                 mutex_exit(&dn->dn_mtx);
1118                                 tx->tx_needassign_txh = txh;
1119                                 return (SET_ERROR(ERESTART));
1120                         }
1121                         if (dn->dn_assigned_txg == 0)
1122                                 dn->dn_assigned_txg = tx->tx_txg;
1123                         ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
1124                         (void) refcount_add(&dn->dn_tx_holds, tx);
1125                         mutex_exit(&dn->dn_mtx);
1126                 }
1127                 towrite += txh->txh_space_towrite;
1128                 tofree += txh->txh_space_tofree;
1129                 tooverwrite += txh->txh_space_tooverwrite;
1130                 tounref += txh->txh_space_tounref;
1131                 tohold += txh->txh_memory_tohold;
1132                 fudge += txh->txh_fudge;
1133         }

1135         /*
1136          * If a snapshot has been taken since we made our estimates,
1137          * assume that we won't be able to free or overwrite anything.
1138          */
1139         if (tx->tx_objset &&
1140             dsl_dataset_prev_snap_txg(tx->tx_objset->os_dsl_dataset) >
1141             tx->tx_lastsnap_txg) {
1142                 towrite += tooverwrite;
1143                 tooverwrite = tofree = 0;
1144         }

1146         /* needed allocation: worst-case estimate of write space */
1147         asize = spa_get_asize(tx->tx_pool->dp_spa, towrite + tooverwrite);
1148         /* freed space estimate: worst-case overwrite + free estimate */
1149         fsize = spa_get_asize(tx->tx_pool->dp_spa, tooverwrite) + tofree;
```

```
1150         /* convert unrefd space to worst-case estimate */
1151         usize = spa_get_asize(tx->tx_pool->dp_spa, tounref);
1152         /* calculate memory footprint estimate */
1153         memory = towrite + tooverwrite + tohold;

1155 #ifdef ZFS_DEBUG
1156         /*
1157          * Add in 'tohold' to account for our dirty holds on this memory
1158          * XXX - the "fudge" factor is to account for skipped blocks that
1159          * we missed because dnode_next_offset() misses in-core-only blocks.
1160          */
1161         tx->tx_space_towrite = asize +
1162             spa_get_asize(tx->tx_pool->dp_spa, tohold + fudge);
1163         tx->tx_space_tofree = tofree;
1164         tx->tx_space_tooverwrite = tooverwrite;
1165         tx->tx_space_tounref = tounref;
1166 #endif

1168         if (tx->tx_dir && asize != 0) {
1169                 int err = dsl_dir_tempreserve_space(tx->tx_dir, memory,
1170                     asize, fsize, usize, &tx->tx_tempreserve_cookie, tx);
1171                 if (err)
1172                         return (err);
1173         }

1175         return (0);
1176 }
______unchanged_portion_omitted_

1214 /*
1215  * Assign tx to a transaction group.  txg_how can be one of:
1216  *
1217  * (1)  TXG_WAIT.  If the current open txg is full, waits until there's
1218  *      a new one.  This should be used when you're not holding locks.
1219  *      It will only fail if we're truly out of space (or over quota).
1220  *
1221  * (2)  TXG_NOWAIT.  If we can't assign into the current open txg without
1222  *      blocking, returns immediately with ERESTART.  This should be used
1223  *      whenever you're holding locks.  On an ERESTART error, the caller
1224  *      should drop locks, do a dmu_tx_wait(tx), and try again.
1225  *
1226  * (3)  TXG_WAITED.  Like TXG_NOWAIT, but indicates that dmu_tx_wait()
1227  *      has already been called on behalf of this operation (though
1228  *      most likely on a different tx).
1229  */
1230 int
1231 dmu_tx_assign(dmu_tx_t *tx, txg_how_t txg_how)
1232 {
1233         int err;

1235         ASSERT(tx->tx_txg == 0);
1236         ASSERT(txg_how == TXG_WAIT || txg_how == TXG_NOWAIT ||
1237             txg_how == TXG_WAITED);
1075         ASSERT(txg_how == TXG_WAIT || txg_how == TXG_NOWAIT);
1238         ASSERT(!dsl_pool_sync_context(tx->tx_pool));

1240         /* If we might wait, we must not hold the config lock. */
1241         ASSERT(txg_how != TXG_WAIT || !dsl_pool_config_held(tx->tx_pool));

1243         if (txg_how == TXG_WAITED)
1244                 tx->tx_waited = B_TRUE;

1246         while ((err = dmu_tx_try_assign(tx, txg_how)) != 0) {
1247                 dmu_tx_unassign(tx);

1249                 if (err != ERESTART || txg_how != TXG_WAIT)
```



```
1250                         return (err);

1252                 dmu_tx_wait(tx);
1253         }

1255         txg_rele_to_quiesce(&tx->tx_txgh);

1257         return (0);
1258 }

1260 void
1261 dmu_tx_wait(dmu_tx_t *tx)
1262 {
1263         spa_t *spa = tx->tx_pool->dp_spa;
1264         dsl_pool_t *dp = tx->tx_pool;

1266         ASSERT(tx->tx_txg == 0);
1267         ASSERT(!dsl_pool_config_held(tx->tx_pool));

1269         if (tx->tx_wait_dirty) {
1270                 /*
1271                  * dmu_tx_try_assign() has determined that we need to wait
1272                  * because we've consumed much or all of the dirty buffer
1273                  * space.
1104          * It's possible that the pool has become active after this thread
1105          * has tried to obtain a tx. If that's the case then his
1106          * tx_lasttried_txg would not have been assigned.
1274                  */
1275                 mutex_enter(&dp->dp_lock);
1276                 while (dp->dp_dirty_total >= zfs_dirty_data_max)
1277                         cv_wait(&dp->dp_spaceavail_cv, &dp->dp_lock);
1278                 uint64_t dirty = dp->dp_dirty_total;
1279                 mutex_exit(&dp->dp_lock);

1281                 dmu_tx_delay(tx, dirty);

1283                 tx->tx_wait_dirty = B_FALSE;

1285                 /*
1286                  * Note: setting tx_waited only has effect if the caller
1287                  * used TX_WAIT.  Otherwise they are going to destroy
1288                  * this tx and try again.  The common case, zfs_write(),
1289                  * uses TX_WAIT.
1290                  */
1291                 tx->tx_waited = B_TRUE;
1292         } else if (spa_suspended(spa) || tx->tx_lasttried_txg == 0) {
1293                 /*
1294                  * If the pool is suspended we need to wait until it
1295                  * is resumed.  Note that it's possible that the pool
1296                  * has become active after this thread has tried to
1297                  * obtain a tx.  If that's the case then tx_lasttried_txg
1298                  * would not have been set.
1299                  */
1300                 txg_wait_synced(dp, spa_last_synced_txg(spa) + 1);
1108         if (spa_suspended(spa) || tx->tx_lasttried_txg == 0) {
1109                 txg_wait_synced(tx->tx_pool, spa_last_synced_txg(spa) + 1);
1301         } else if (tx->tx_needassign_txh) {
1302                 /*
1303                  * A dnode is assigned to the quiescing txg.  Wait for its
1304                  * transaction to complete.
1305                  */
1306                 dnode_t *dn = tx->tx_needassign_txh->txh_dnode;

1308                 mutex_enter(&dn->dn_mtx);
1309                 while (dn->dn_assigned_txg == tx->tx_lasttried_txg - 1)
1310                         cv_wait(&dn->dn_notxholds, &dn->dn_mtx);
```

```
1311                 mutex_exit(&dn->dn_mtx);
1312                 tx->tx_needassign_txh = NULL;
1313         } else {
1314                 txg_wait_open(tx->tx_pool, tx->tx_lasttried_txg + 1);
1315         }
1316 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   19232 Thu Aug 22 16:15:10 2013
new/usr/src/uts/common/fs/zfs/dmu_zfetch.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dnode.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_zfetch.h>
#include <sys/dmu.h>
#include <sys/dbuf.h>
#include <sys/kstat.h>

/*
 * I'm against tune-ables, but these should probably exist as tweakable globals
 * until we can get this working the way we want it to.
 */

int zfs_prefetch_disable = 0;

/* max # of streams per zfetch */
uint32_t        zfetch_max_streams = 8;
/* min time before stream reclaim */
uint32_t        zfetch_min_sec_reap = 2;
/* max number of blocks to fetch at a time */
uint32_t        zfetch_block_cap = 256;
/* number of bytes in a array_read at which we stop prefetching (1Mb) */
uint64_t        zfetch_array_rd_sz = 1024 * 1024;

/* forward decls for static routines */
static boolean_t        dmu_zfetch_colinear(zfetch_t *, zstream_t *);
static void             dmu_zfetch_dofetch(zfetch_t *, zstream_t *);
static uint64_t         dmu_zfetch_fetch(dnode_t *, uint64_t, uint64_t);
static uint64_t         dmu_zfetch_fetchsz(dnode_t *, uint64_t, uint64_t);
```



```
static boolean_t        dmu_zfetch_find(zfetch_t *, zstream_t *, int);
static int              dmu_zfetch_stream_insert(zfetch_t *, zstream_t *);
static zstream_t        *dmu_zfetch_stream_reclaim(zfetch_t *);
static void             dmu_zfetch_stream_remove(zfetch_t *, zstream_t *);
static int              dmu_zfetch_streams_equal(zstream_t *, zstream_t *);

typedef struct zfetch_stats {
        kstat_named_t zfetchstat_hits;
        kstat_named_t zfetchstat_misses;
        kstat_named_t zfetchstat_colinear_hits;
        kstat_named_t zfetchstat_colinear_misses;
        kstat_named_t zfetchstat_stride_hits;
        kstat_named_t zfetchstat_stride_misses;
        kstat_named_t zfetchstat_reclaim_successes;
        kstat_named_t zfetchstat_reclaim_failures;
        kstat_named_t zfetchstat_stream_resets;
        kstat_named_t zfetchstat_stream_noresets;
        kstat_named_t zfetchstat_bogus_streams;
} zfetch_stats_t;
_____unchanged_portion_omitted_

/*
 * This function computes the actual size, in blocks, that can be prefetched,
 * and fetches it.
 */
static uint64_t
dmu_zfetch_fetch(dnode_t *dn, uint64_t blkid, uint64_t nblks)
{
        uint64_t        fetchsz;
        uint64_t        i;

        fetchsz = dmu_zfetch_fetchsz(dn, blkid, nblks);

        for (i = 0; i < fetchsz; i++) {
                dbuf_prefetch(dn, blkid + i, ZIO_PRIORITY_ASYNC_READ);
                dbuf_prefetch(dn, blkid + i);
        }

        return (fetchsz);
}
_____unchanged_portion_omitted_
```

```
**********************************************************
   56694 Thu Aug 22 16:15:11 2013
new/usr/src/uts/common/fs/zfs/dnode.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1790 /*
1791  * Call when we think we're going to write/free space in open context to track
1792  * the amount of memory in use by the currently open txg.
1791  * Call when we think we're going to write/free space in open context.
1792  * Be conservative (ie. OK to write less than this or free more than
1793  * this, but don't write more or free less).
1793  */
1794 void
1795 dnode_willuse_space(dnode_t *dn, int64_t space, dmu_tx_t *tx)
1796 {
1797         objset_t *os = dn->dn_objset;
1798         dsl_dataset_t *ds = os->os_dsl_dataset;
1799         int64_t aspace = spa_get_asize(os->os_spa, space);

1801         if (ds != NULL) {
1802                 dsl_dir_willuse_space(ds->ds_dir, aspace, tx);
1803                 dsl_pool_dirty_space(dmu_tx_pool(tx), space, tx);
1804         }
1801         if (space > 0)
1802                 space = spa_get_asize(os->os_spa, space);

1806         dmu_tx_willuse_space(tx, aspace);
1804         if (ds)
1805                 dsl_dir_willuse_space(ds->ds_dir, space, tx);

1807         dmu_tx_willuse_space(tx, space);
1807 }
______unchanged_portion_omitted_
```

```
**********************************************************
   35695 Thu Aug 22 16:15:12 2013
new/usr/src/uts/common/fs/zfs/dsl_dir.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 585 struct tempreserve {
 586         list_node_t tr_node;
 587         dsl_pool_t *tr_dp;
 587         dsl_dir_t *tr_ds;
 588         uint64_t tr_size;
 589 };
______unchanged_portion_omitted_

 706 /*
 707  * Reserve space in this dsl_dir, to be used in this tx's txg.
 708  * After the space has been dirtied (and dsl_dir_willuse_space()
 709  * has been called), the reservation should be canceled, using
 710  * dsl_dir_tempreserve_clear().
 711  */
 712 int
 713 dsl_dir_tempreserve_space(dsl_dir_t *dd, uint64_t lsize, uint64_t asize,
 714     uint64_t fsize, uint64_t usize, void **tr_cookiep, dmu_tx_t *tx)
 715 {
 716         int err;
 717         list_t *tr_list;

 719         if (asize == 0) {
 720                 *tr_cookiep = NULL;
 721                 return (0);
 722         }

 724         tr_list = kmem_alloc(sizeof (list_t), KM_SLEEP);
 725         list_create(tr_list, sizeof (struct tempreserve),
 726             offsetof(struct tempreserve, tr_node));
 727         ASSERT3S(asize, >, 0);
 728         ASSERT3S(fsize, >=, 0);

 730         err = arc_tempreserve_space(lsize, tx->tx_txg);
 731         if (err == 0) {
 732                 struct tempreserve *tr;

 734                 tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
 735                 tr->tr_size = lsize;
 736                 list_insert_tail(tr_list, tr);

 739                 err = dsl_pool_tempreserve_space(dd->dd_pool, asize, tx);
 737         } else {
 738                 if (err == EAGAIN) {
 739                         /*
 740                          * If arc_memory_throttle() detected that pageout
 741                          * is running and we are low on memory, we delay new
 742                          * non-pageout transactions to give pageout an
 743                          * advantage.
 744                          *
 745                          * It is unfortunate to be delaying while the caller's
 746                          * locks are held.
 747                          */
 748                         txg_delay(dd->dd_pool, tx->tx_txg,
 749                             MSEC2NSEC(10), MSEC2NSEC(10));
 750                         err = SET_ERROR(ERESTART);
 751                 }
```



```
 746                 dsl_pool_memory_pressure(dd->dd_pool);
 752         }

 754         if (err == 0) {
 750                 struct tempreserve *tr;

 752                 tr = kmem_zalloc(sizeof (struct tempreserve), KM_SLEEP);
 753                 tr->tr_dp = dd->dd_pool;
 754                 tr->tr_size = asize;
 755                 list_insert_tail(tr_list, tr);

 755                 err = dsl_dir_tempreserve_impl(dd, asize, fsize >= asize,
 756                     FALSE, asize > usize, tr_list, tx, TRUE);
 757         }

 759         if (err != 0)
 760                 dsl_dir_tempreserve_clear(tr_list, tx);
 761         else
 762                 *tr_cookiep = tr_list;

 764         return (err);
 765 }

 767 /*
 768  * Clear a temporary reservation that we previously made with
 769  * dsl_dir_tempreserve_space().
 770  */
 771 void
 772 dsl_dir_tempreserve_clear(void *tr_cookie, dmu_tx_t *tx)
 773 {
 774         int txgidx = tx->tx_txg & TXG_MASK;
 775         list_t *tr_list = tr_cookie;
 776         struct tempreserve *tr;

 778         ASSERT3U(tx->tx_txg, !=, 0);

 780         if (tr_cookie == NULL)
 781                 return;

 783         while ((tr = list_head(tr_list)) != NULL) {
 784                 if (tr->tr_ds) {
 785         while (tr = list_head(tr_list)) {
 786                 if (tr->tr_dp) {
 787                         dsl_pool_tempreserve_clear(tr->tr_dp, tr->tr_size, tx);
 788                 } else if (tr->tr_ds) {
 785                         mutex_enter(&tr->tr_ds->dd_lock);
 786                         ASSERT3U(tr->tr_ds->dd_tempreserved[txgidx], >=,
 787                             tr->tr_size);
 788                         tr->tr_ds->dd_tempreserved[txgidx] -= tr->tr_size;
 789                         mutex_exit(&tr->tr_ds->dd_lock);
 790                 } else {
 791                         arc_tempreserve_clear(tr->tr_size);
 792                 }
 793                 list_remove(tr_list, tr);
 794                 kmem_free(tr, sizeof (struct tempreserve));
 795         }

 797         kmem_free(tr_list, sizeof (list_t));
 798 }

 800 /*
 801  * This should be called from open context when we think we're going to write
 802  * or free space, for example when dirtying data. Be conservative; it's okay
 803  * to write less space or free more, but we don't want to write more or free
 804  * less than the amount specified.
 805  */
```

```
806 void
807 dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
804 static void
805 dsl_dir_willuse_space_impl(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
808 {
809         int64_t parent_space;
810         uint64_t est_used;

812         mutex_enter(&dd->dd_lock);
813         if (space > 0)
814                 dd->dd_space_towrite[tx->tx_txg & TXG_MASK] += space;

816         est_used = dsl_dir_space_towrite(dd) + dd->dd_phys->dd_used_bytes;
817         parent_space = parent_delta(dd, est_used, space);
818         mutex_exit(&dd->dd_lock);

820         /* Make sure that we clean up dd_space_to* */
821         dsl_dir_dirty(dd, tx);

823         /* XXX this is potentially expensive and unnecessary... */
824         if (parent_space && dd->dd_parent)
825                 dsl_dir_willuse_space(dd->dd_parent, parent_space, tx);
823                 dsl_dir_willuse_space_impl(dd->dd_parent, parent_space, tx);
826 }

826 /*
827  * Call in open context when we think we're going to write/free space,
828  * eg. when dirtying data.  Be conservative (ie. OK to write less than
829  * this or free more than this, but don't write more or free less).
830  */
831 void
832 dsl_dir_willuse_space(dsl_dir_t *dd, int64_t space, dmu_tx_t *tx)
833 {
834         dsl_pool_willuse_space(dd->dd_pool, space, tx);
835         dsl_dir_willuse_space_impl(dd, space, tx);
836 }

828 /* call from syncing context when we actually write/free space for this dd */
829 void
830 dsl_dir_diduse_space(dsl_dir_t *dd, dd_used_t type,
831     int64_t used, int64_t compressed, int64_t uncompressed, dmu_tx_t *tx)
832 {
833         int64_t accounted_delta;

835         /*
836          * dsl_dataset_set_refreservation_sync_impl() calls this with
837          * dd_lock held, so that it can atomically update
838          * ds->ds_reserved and the dsl_dir accounting, so that
839          * dsl_dataset_check_quota() can see dataset and dir accounting
840          * consistently.
841          */
842         boolean_t needlock = !MUTEX_HELD(&dd->dd_lock);

844         ASSERT(dmu_tx_is_syncing(tx));
845         ASSERT(type < DD_USED_NUM);

847         dmu_buf_will_dirty(dd->dd_dbuf, tx);

849         if (needlock)
850                 mutex_enter(&dd->dd_lock);
851         accounted_delta = parent_delta(dd, dd->dd_phys->dd_used_bytes, used);
852         ASSERT(used >= 0 || dd->dd_phys->dd_used_bytes >= -used);
853         ASSERT(compressed >= 0 ||
854             dd->dd_phys->dd_compressed_bytes >= -compressed);
855         ASSERT(uncompressed >= 0 ||
856             dd->dd_phys->dd_uncompressed_bytes >= -uncompressed);
```



```
857         dd->dd_phys->dd_used_bytes += used;
858         dd->dd_phys->dd_uncompressed_bytes += uncompressed;
859         dd->dd_phys->dd_compressed_bytes += compressed;

861         if (dd->dd_phys->dd_flags & DD_FLAG_USED_BREAKDOWN) {
862                 ASSERT(used > 0 ||
863                     dd->dd_phys->dd_used_breakdown[type] >= -used);
864                 dd->dd_phys->dd_used_breakdown[type] += used;
865 #ifdef DEBUG
866                 dd_used_t t;
867                 uint64_t u = 0;
868                 for (t = 0; t < DD_USED_NUM; t++)
869                         u += dd->dd_phys->dd_used_breakdown[t];
870                 ASSERT3U(u, ==, dd->dd_phys->dd_used_bytes);
871 #endif
872         }
873         if (needlock)
874                 mutex_exit(&dd->dd_lock);

876         if (dd->dd_parent != NULL) {
877                 dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
878                     accounted_delta, compressed, uncompressed, tx);
879                 dsl_dir_transfer_space(dd->dd_parent,
880                     used - accounted_delta,
881                     DD_USED_CHILD_RSRV, DD_USED_CHILD, tx);
882         }
883 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30648 Thu Aug 22 16:15:13 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

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

#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_scan.h>
#include <sys/dnode.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/arc.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/zfs_context.h>
#include <sys/fs/zfs.h>
#include <sys/zfs_znode.h>
#include <sys/spa_impl.h>
#include <sys/dsl_deadlist.h>
#include <sys/bptree.h>
#include <sys/zfeature.h>
#include <sys/zil_impl.h>
#include <sys/dsl_userhold.h>

/*
 * ZFS Write Throttle
 * ------------------
 *
 * ZFS must limit the rate of incoming writes to the rate at which it is able
 * to sync data modifications to the backend storage. Throttling by too much
 * creates an artificial limit; throttling by too little can only be sustained
 * for short periods and would lead to highly lumpy performance. On a per-pool
 * basis, ZFS tracks the amount of modified (dirty) data. As operations change
 * data, the amount of dirty data increases; as ZFS syncs out data, the amount
```



```
 * of dirty data decreases. When the amount of dirty data exceeds a
 * predetermined threshold further modifications are blocked until the amount
 * of dirty data decreases (as data is synced out).
 *
 * The limit on dirty data is tunable, and should be adjusted according to
 * both the IO capacity and available memory of the system. The larger the
 * window, the more ZFS is able to aggregate and amortize metadata (and data)
 * changes. However, memory is a limited resource, and allowing for more dirty
 * data comes at the cost of keeping other useful data in memory (for example
 * ZFS data cached by the ARC).
 *
 * Implementation
 *
 * As buffers are modified dsl_pool_willuse_space() increments both the per-
 * txg (dp_dirty_pertxg[]) and poolwide (dp_dirty_total) accounting of
 * dirty space used; dsl_pool_dirty_space() decrements those values as data
 * is synced out from dsl_pool_sync(). While only the poolwide value is
 * relevant, the per-txg value is useful for debugging. The tunable
 * zfs_dirty_data_max determines the dirty space limit. Once that value is
 * exceeded, new writes are halted until space frees up.
 *
 * The zfs_dirty_data_sync tunable dictates the threshold at which we
 * ensure that there is a txg syncing (see the comment in txg.c for a full
 * description of transaction group stages).
 *
 * The IO scheduler uses both the dirty space limit and current amount of
 * dirty data as inputs. Those values affect the number of concurrent IOs ZFS
 * issues. See the comment in vdev_queue.c for details of the IO scheduler.
 *
 * The delay is also calculated based on the amount of dirty data.  See the
 * comment above dmu_tx_delay() for details.
 */
int zfs_no_write_throttle = 0;
int zfs_write_limit_shift = 3;                  /* 1/8th of physical memory */
int zfs_txg_synctime_ms = 1000;         /* target millisecs to sync a txg */

/*
 * zfs_dirty_data_max will be set to zfs_dirty_data_max_percent% of all memory,
 * capped at zfs_dirty_data_max_max.  It can also be overridden in /etc/system.
 */
uint64_t zfs_dirty_data_max;
uint64_t zfs_dirty_data_max_max = 4ULL * 1024 * 1024 * 1024;
int zfs_dirty_data_max_percent = 10;
uint64_t zfs_write_limit_min = 32 << 20;        /* min write limit is 32MB */
uint64_t zfs_write_limit_max = 0;               /* max data payload per txg */
uint64_t zfs_write_limit_inflated = 0;
uint64_t zfs_write_limit_override = 0;

/*
 * If there is at least this much dirty data, push out a txg.
 */
uint64_t zfs_dirty_data_sync = 64 * 1024 * 1024;
kmutex_t zfs_write_limit_lock;

/*
 * Once there is this amount of dirty data, the dmu_tx_delay() will kick in
 * and delay each transaction.
 * This value should be >= zfs_vdev_async_write_active_max_dirty_percent.
 */
int zfs_delay_min_dirty_percent = 60;
static pgcnt_t old_physmem = 0;

/*
 * This controls how quickly the delay approaches infinity.
 * Larger values cause it to delay less for a given amount of dirty data.
 * Therefore larger values will cause there to be more dirty data for a
```

```
 * given throughput.
 *
 * For the smoothest delay, this value should be about 1 billion divided
 * by the maximum number of operations per second.  This will smoothly
 * handle between 10x and 1/10th this number.
 *
 * Note: zfs_delay_scale * zfs_dirty_data_max must be < 2^64, due to the
 * multiply in dmu_tx_delay().
 */
uint64_t zfs_delay_scale = 1000 * 1000 * 1000 / 2000;


/*
 * XXX someday maybe turn these into #defines, and you have to tune it on a
 * per-pool basis using zfs.conf.
 */


hrtime_t zfs_throttle_delay = MSEC2NSEC(10);
hrtime_t zfs_throttle_resolution = MSEC2NSEC(10);

int
dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **ddp)
{
        uint64_t obj;
        int err;

        err = zap_lookup(dp->dp_meta_objset,
            dp->dp_root_dir->dd_phys->dd_child_dir_zapobj,
            name, sizeof (obj), 1, &obj);
        if (err)
                return (err);

        return (dsl_dir_hold_obj(dp, obj, name, dp, ddp));
}

static dsl_pool_t *
dsl_pool_open_impl(spa_t *spa, uint64_t txg)
{
        dsl_pool_t *dp;
        blkptr_t *bp = spa_get_rootblkptr(spa);

        dp = kmem_zalloc(sizeof (dsl_pool_t), KM_SLEEP);
        dp->dp_spa = spa;
        dp->dp_meta_rootbp = *bp;
        rrw_init(&dp->dp_config_rwlock, B_TRUE);
        dp->dp_write_limit = zfs_write_limit_min;
        txg_init(dp, txg);

        txg_list_create(&dp->dp_dirty_datasets,
            offsetof(dsl_dataset_t, ds_dirty_link));
        txg_list_create(&dp->dp_dirty_zilogs,
            offsetof(zilog_t, zl_dirty_link));
        txg_list_create(&dp->dp_dirty_dirs,
            offsetof(dsl_dir_t, dd_dirty_link));
        txg_list_create(&dp->dp_sync_tasks,
            offsetof(dsl_sync_task_t, dst_node));

        mutex_init(&dp->dp_lock, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&dp->dp_spaceavail_cv, NULL, CV_DEFAULT, NULL);

        dp->dp_vnrele_taskq = taskq_create("zfs_vn_rele_taskq", 1, minclsyspri,
            1, 4, 0);

        return (dp);
}
```

*unchanged_portion_omitted_*


```
void
dsl_pool_close(dsl_pool_t *dp)
{
        /* drop our references from dsl_pool_open() */

        /*
         * Drop our references from dsl_pool_open().
         *
         * Since we held the origin_snap from "syncing" context (which
         * includes pool-opening context), it actually only got a "ref"
         * and not a hold, so just drop that here.
         */
        if (dp->dp_origin_snap)
                dsl_dataset_rele(dp->dp_origin_snap, dp);
        if (dp->dp_mos_dir)
                dsl_dir_rele(dp->dp_mos_dir, dp);
        if (dp->dp_free_dir)
                dsl_dir_rele(dp->dp_free_dir, dp);
        if (dp->dp_root_dir)
                dsl_dir_rele(dp->dp_root_dir, dp);

        bpobj_close(&dp->dp_free_bpobj);

        /* undo the dmu_objset_open_impl(mos) from dsl_pool_open() */
        if (dp->dp_meta_objset)
                dmu_objset_evict(dp->dp_meta_objset);

        txg_list_destroy(&dp->dp_dirty_datasets);
        txg_list_destroy(&dp->dp_dirty_zilogs);
        txg_list_destroy(&dp->dp_sync_tasks);
        txg_list_destroy(&dp->dp_dirty_dirs);

        arc_flush(dp->dp_spa);
        txg_fini(dp);
        dsl_scan_fini(dp);
        rrw_destroy(&dp->dp_config_rwlock);
        mutex_destroy(&dp->dp_lock);
        taskq_destroy(dp->dp_vnrele_taskq);
        if (dp->dp_blkstats)
                kmem_free(dp->dp_blkstats, sizeof (zfs_all_blkstats_t));
        kmem_free(dp, sizeof (dsl_pool_t));
}
```

*unchanged_portion_omitted_*

```
static void
dsl_pool_sync_mos(dsl_pool_t *dp, dmu_tx_t *tx)
{
        zio_t *zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
        dmu_objset_sync(dp->dp_meta_objset, zio, tx);
        VERIFY0(zio_wait(zio));
        dprintf_bp(&dp->dp_meta_rootbp, "meta objset rootbp is %s", "");
        spa_set_rootblkptr(dp->dp_spa, &dp->dp_meta_rootbp);
}

static void
dsl_pool_dirty_delta(dsl_pool_t *dp, int64_t delta)
{
        ASSERT(MUTEX_HELD(&dp->dp_lock));

        if (delta < 0)
                ASSERT3U(-delta, <=, dp->dp_dirty_total);

        dp->dp_dirty_total += delta;

        /*
```

```
442          * Note: we signal even when increasing dp_dirty_total.
443          * This ensures forward progress -- each thread wakes the next waiter.
444          */
445         if (dp->dp_dirty_total <= zfs_dirty_data_max)
446                 cv_signal(&dp->dp_spaceavail_cv);
447 }
448
449 void
450 dsl_pool_sync(dsl_pool_t *dp, uint64_t txg)
451 {
452         zio_t *zio;
453         dmu_tx_t *tx;
454         dsl_dir_t *dd;
455         dsl_dataset_t *ds;
456         objset_t *mos = dp->dp_meta_objset;
457         hrtime_t start, write_time;
458         uint64_t data_written;
459         int err;
460         list_t synced_datasets;
461
462         list_create(&synced_datasets, sizeof (dsl_dataset_t),
463             offsetof(dsl_dataset_t, ds_synced_link));
464
465         tx = dmu_tx_create_assigned(dp, txg);
466
467         /*
468          * Write out all dirty blocks of dirty datasets.
469          * We need to copy dp_space_towrite() before doing
470          * dsl_sync_task_sync(), because
471          * dsl_dataset_snapshot_reserve_space() will increase
472          * dp_space_towrite but not actually write anything.
473          */
474         data_written = dp->dp_space_towrite[txg & TXG_MASK];
475
476         tx = dmu_tx_create_assigned(dp, txg);
477
478         dp->dp_read_overhead = 0;
479         start = gethrtime();
480
481         zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
482         while ((ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) != NULL) {
483         while (ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) {
484                 /*
485                  * We must not sync any non-MOS datasets twice, because
486                  * we may have taken a snapshot of them.  However, we
487                  * may sync newly-created datasets on pass 2.
488                  */
489                 ASSERT(!list_link_active(&ds->ds_synced_link));
490                 list_insert_tail(&synced_datasets, ds);
491                 dsl_dataset_sync(ds, zio, tx);
492         }
493         VERIFY0(zio_wait(zio));
494         DTRACE_PROBE(pool_sync__1setup);
495         err = zio_wait(zio);
496
497         /*
498          * We have written all of the accounted dirty data, so our
499          * dp_space_towrite should now be zero.  However, some seldom-used
500          * code paths do not adhere to this (e.g. dbuf_undirty(), also
501          * rounding error in dbuf_write_physdone).
502          * Shore up the accounting of any dirtied space now.
503          */
504         dsl_pool_undirty_space(dp, dp->dp_dirty_pertxg[txg & TXG_MASK], txg);
505         write_time = gethrtime() - start;
506         ASSERT(err == 0);
507         DTRACE_PROBE(pool_sync__2rootzio);
```



```
508         /*
509          * After the data blocks have been written (ensured by the zio_wait()
510          * above), update the user/group space accounting.
511          */
512         for (ds = list_head(&synced_datasets); ds != NULL;
513             ds = list_next(&synced_datasets, ds)) {
514         for (ds = list_head(&synced_datasets); ds;
515             ds = list_next(&synced_datasets, ds))
516                 dmu_objset_do_userquota_updates(ds->ds_objset, tx);
517         }
518
519         /*
520          * Sync the datasets again to push out the changes due to
521          * userspace updates.  This must be done before we process the
522          * sync tasks, so that any snapshots will have the correct
523          * user accounting information (and we won't get confused
524          * about which blocks are part of the snapshot).
525          */
526         zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
527         while ((ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) != NULL) {
528         while (ds = txg_list_remove(&dp->dp_dirty_datasets, txg)) {
529                 ASSERT(list_link_active(&ds->ds_synced_link));
530                 dmu_buf_rele(ds->ds_dbuf, ds);
531                 dsl_dataset_sync(ds, zio, tx);
532         }
533         VERIFY0(zio_wait(zio));
534         err = zio_wait(zio);
535
536         /*
537          * Now that the datasets have been completely synced, we can
538          * clean up our in-memory structures accumulated while syncing:
539          *
540          *  - move dead blocks from the pending deadlist to the on-disk deadlist
541          *  - release hold from dsl_dataset_dirty()
542          */
543         while ((ds = list_remove_head(&synced_datasets)) != NULL) {
544         while (ds = list_remove_head(&synced_datasets)) {
545                 objset_t *os = ds->ds_objset;
546                 bplist_iterate(&ds->ds_pending_deadlist,
547                     deadlist_enqueue_cb, &ds->ds_deadlist, tx);
548                 ASSERT(!dmu_objset_is_dirty(os, txg));
549                 dmu_buf_rele(ds->ds_dbuf, ds);
550         }
551         while ((dd = txg_list_remove(&dp->dp_dirty_dirs, txg)) != NULL) {
552
553         start = gethrtime();
554         while (dd = txg_list_remove(&dp->dp_dirty_dirs, txg))
555                 dsl_dir_sync(dd, tx);
556         }
557         write_time += gethrtime() - start;
558
559         /*
560          * The MOS's space is accounted for in the pool/$MOS
561          * (dp_mos_dir).  We can't modify the mos while we're syncing
562          * it, so we remember the deltas and apply them here.
563          */
564         if (dp->dp_mos_used_delta != 0 || dp->dp_mos_compressed_delta != 0 ||
565             dp->dp_mos_uncompressed_delta != 0) {
566                 dsl_dir_diduse_space(dp->dp_mos_dir, DD_USED_HEAD,
567                     dp->dp_mos_used_delta,
568                     dp->dp_mos_compressed_delta,
569                     dp->dp_mos_uncompressed_delta, tx);
570                 dp->dp_mos_used_delta = 0;
571                 dp->dp_mos_compressed_delta = 0;
572                 dp->dp_mos_uncompressed_delta = 0;
```

```
545         }

455         start = gethrtime();
547         if (list_head(&mos->os_dirty_dnodes[txg & TXG_MASK]) != NULL ||
548             list_head(&mos->os_free_dnodes[txg & TXG_MASK]) != NULL) {
549                 dsl_pool_sync_mos(dp, tx);
458                 zio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
459                 dmu_objset_sync(mos, zio, tx);
460                 err = zio_wait(zio);
461                 ASSERT(err == 0);
462                 dprintf_bp(&dp->dp_meta_rootbp, "meta objset rootbp is %s", "");
463                 spa_set_rootblkptr(dp->dp_spa, &dp->dp_meta_rootbp);
550         }
465         write_time += gethrtime() - start;
466         DTRACE_PROBE2(pool_sync__4io, hrtime_t, write_time,
467             hrtime_t, dp->dp_read_overhead);
468         write_time -= dp->dp_read_overhead;

552         /*
553          * If we modify a dataset in the same txg that we want to destroy it,
554          * its dsl_dir's dd_dbuf will be dirty, and thus have a hold on it.
555          * dsl_dir_destroy_check() will fail if there are unexpected holds.
556          * Therefore, we want to sync the MOS (thus syncing the dd_dbuf
557          * and clearing the hold on it) before we process the sync_tasks.
558          * The MOS data dirtied by the sync_tasks will be synced on the next
559          * pass.
560          */
479         DTRACE_PROBE(pool_sync__3task);
561         if (!txg_list_empty(&dp->dp_sync_tasks, txg)) {
562                 dsl_sync_task_t *dst;
563                 /*
564                  * No more sync tasks should have been added while we
565                  * were syncing.
566                  */
567                 ASSERT3U(spa_sync_pass(dp->dp_spa), ==, 1);
568                 while ((dst = txg_list_remove(&dp->dp_sync_tasks, txg)) != NULL)
486                 ASSERT(spa_sync_pass(dp->dp_spa) == 1);
487                 while (dst = txg_list_remove(&dp->dp_sync_tasks, txg))
569                         dsl_sync_task_sync(dst, tx);
570         }

572         dmu_tx_commit(tx);

574         DTRACE_PROBE2(dsl_pool_sync__done, dsl_pool_t *dp, dp, uint64_t, txg);
493         dp->dp_space_towrite[txg & TXG_MASK] = 0;
494         ASSERT(dp->dp_tempreserved[txg & TXG_MASK] == 0);

496         /*
497          * If the write limit max has not been explicitly set, set it
498          * to a fraction of available physical memory (default 1/8th).
499          * Note that we must inflate the limit because the spa
500          * inflates write sizes to account for data replication.
501          * Check this each sync phase to catch changing memory size.
502          */
503         if (physmem != old_physmem && zfs_write_limit_shift) {
504                 mutex_enter(&zfs_write_limit_lock);
505                 old_physmem = physmem;
506                 zfs_write_limit_max = ptob(physmem) >> zfs_write_limit_shift;
507                 zfs_write_limit_inflated = MAX(zfs_write_limit_min,
508                     spa_get_asize(dp->dp_spa, zfs_write_limit_max));
509                 mutex_exit(&zfs_write_limit_lock);
510         }

512         /*
513          * Attempt to keep the sync time consistent by adjusting the
514          * amount of write traffic allowed into each transaction group.
```



```
515          * Weight the throughput calculation towards the current value:
516          *      thru = 3/4 old_thru + 1/4 new_thru
517          *
518          * Note: write_time is in nanosecs while dp_throughput is expressed in
519          * bytes per millisecond.
520          */
521         ASSERT(zfs_write_limit_min > 0);
522         if (data_written > zfs_write_limit_min / 8 &&
523             write_time > MSEC2NSEC(1)) {
524                 uint64_t throughput = data_written / NSEC2MSEC(write_time);

526                 if (dp->dp_throughput)
527                         dp->dp_throughput = throughput / 4 +
528                             3 * dp->dp_throughput / 4;
529                 else
530                         dp->dp_throughput = throughput;
531                 dp->dp_write_limit = MIN(zfs_write_limit_inflated,
532                     MAX(zfs_write_limit_min,
533                     dp->dp_throughput * zfs_txg_synctime_ms));
534         }
575 }

577 void
578 dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg)
579 {
580         zilog_t *zilog;
541         dsl_dataset_t *ds;

582         while (zilog = txg_list_remove(&dp->dp_dirty_zilogs, txg)) {
583                 dsl_dataset_t *ds = dmu_objset_ds(zilog->zl_os);
544                 ds = dmu_objset_ds(zilog->zl_os);
584                 zil_clean(zilog, txg);
585                 ASSERT(!dmu_objset_is_dirty(zilog->zl_os, txg));
586                 dmu_buf_rele(ds->ds_dbuf, zilog);
587         }
588         ASSERT(!dmu_objset_is_dirty(dp->dp_meta_objset, txg));
589 }
_____unchanged_portion_omitted_

625 boolean_t
626 dsl_pool_need_dirty_delay(dsl_pool_t *dp)
586 int
587 dsl_pool_tempreserve_space(dsl_pool_t *dp, uint64_t space, dmu_tx_t *tx)
627 {
628         uint64_t delay_min_bytes =
629             zfs_dirty_data_max * zfs_delay_min_dirty_percent / 100;
630         boolean_t rv;
589         uint64_t reserved = 0;
590         uint64_t write_limit = (zfs_write_limit_override ?
591             zfs_write_limit_override : dp->dp_write_limit);

632         mutex_enter(&dp->dp_lock);
633         if (dp->dp_dirty_total > zfs_dirty_data_sync)
634                 txg_kick(dp);
635         rv = (dp->dp_dirty_total > delay_min_bytes);
636         mutex_exit(&dp->dp_lock);
637         return (rv);
593         if (zfs_no_write_throttle) {
594                 atomic_add_64(&dp->dp_tempreserved[tx->tx_txg & TXG_MASK],
595                     space);
596                 return (0);
597         }

599         /*
600          * Check to see if we have exceeded the maximum allowed IO for
601          * this transaction group.  We can do this without locks since
```

```
602          * a little slop here is ok.  Note that we do the reserved check
603          * with only half the requested reserve: this is because the
604          * reserve requests are worst-case, and we really don't want to
605          * throttle based off of worst-case estimates.
606          */
607         if (write_limit > 0) {
608                 reserved = dp->dp_space_towrite[tx->tx_txg & TXG_MASK]
609                     + dp->dp_tempreserved[tx->tx_txg & TXG_MASK] / 2;

611                 if (reserved && reserved > write_limit)
612                         return (SET_ERROR(ERESTART));
613         }

615         atomic_add_64(&dp->dp_tempreserved[tx->tx_txg & TXG_MASK], space);

617         /*
618          * If this transaction group is over 7/8ths capacity, delay
619          * the caller 1 clock tick.  This will slow down the "fill"
620          * rate until the sync process can catch up with us.
621          */
622         if (reserved && reserved > (write_limit - (write_limit >> 3))) {
623                 txg_delay(dp, tx->tx_txg, zfs_throttle_delay,
624                     zfs_throttle_resolution);
625         }

627         return (0);
638 }

640 void
641 dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx)
631 dsl_pool_tempreserve_clear(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx)
642 {
643         if (space > 0) {
644                 mutex_enter(&dp->dp_lock);
645                 dp->dp_dirty_pertxg[tx->tx_txg & TXG_MASK] += space;
646                 dsl_pool_dirty_delta(dp, space);
647                 mutex_exit(&dp->dp_lock);
648         }
633         ASSERT(dp->dp_tempreserved[tx->tx_txg & TXG_MASK] >= space);
634         atomic_add_64(&dp->dp_tempreserved[tx->tx_txg & TXG_MASK], -space);
649 }

651 void
652 dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg) {
653         ASSERT3S(space, >=, 0);
654         if (space == 0)
638 dsl_pool_memory_pressure(dsl_pool_t *dp)
639 {
640         uint64_t space_inuse = 0;
641         int i;

643         if (dp->dp_write_limit == zfs_write_limit_min)
655                 return;
656         mutex_enter(&dp->dp_lock);
657         if (dp->dp_dirty_pertxg[txg & TXG_MASK] < space) {
658                 /* XXX writing something we didn't dirty? */
659                 space = dp->dp_dirty_pertxg[txg & TXG_MASK];

646         for (i = 0; i < TXG_SIZE; i++) {
647                 space_inuse += dp->dp_space_towrite[i];
648                 space_inuse += dp->dp_tempreserved[i];
660         }
661         ASSERT3U(dp->dp_dirty_pertxg[txg & TXG_MASK], >=, space);
662         dp->dp_dirty_pertxg[txg & TXG_MASK] -= space;
663         ASSERT3U(dp->dp_dirty_total, >=, space);
664         dsl_pool_dirty_delta(dp, -space);
```



```
650         dp->dp_write_limit = MAX(zfs_write_limit_min,
651             MIN(dp->dp_write_limit, space_inuse / 4));
652 }

654 void
655 dsl_pool_willuse_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx)
656 {
657         if (space > 0) {
658                 mutex_enter(&dp->dp_lock);
659                 dp->dp_space_towrite[tx->tx_txg & TXG_MASK] += space;
665         mutex_exit(&dp->dp_lock);
661         }
666 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   50364 Thu Aug 22 16:15:14 2013
new/usr/src/uts/common/fs/zfs/dsl_scan.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1620 static int
1621 dsl_scan_scrub_cb(dsl_pool_t *dp,
1622     const blkptr_t *bp, const zbookmark_t *zb)
1623 {
1624         dsl_scan_t *scn = dp->dp_scan;
1625         size_t size = BP_GET_PSIZE(bp);
1626         spa_t *spa = dp->dp_spa;
1627         uint64_t phys_birth = BP_PHYSICAL_BIRTH(bp);
1628         boolean_t needs_io;
1629         int zio_flags = ZIO_FLAG_SCAN_THREAD | ZIO_FLAG_RAW | ZIO_FLAG_CANFAIL;
1630         int zio_priority;
1630         int scan_delay = 0;

1632         if (phys_birth <= scn->scn_phys.scn_min_txg ||
1633             phys_birth >= scn->scn_phys.scn_max_txg)
1634                 return (0);

1636         count_block(dp->dp_blkstats, bp);

1638         ASSERT(DSL_SCAN_IS_SCRUB_RESILVER(scn));
1639         if (scn->scn_phys.scn_func == POOL_SCAN_SCRUB) {
1640                 zio_flags |= ZIO_FLAG_SCRUB;
1642                 zio_priority = ZIO_PRIORITY_SCRUB;
1641                 needs_io = B_TRUE;
1642                 scan_delay = zfs_scrub_delay;
1643         } else {
1644                 ASSERT3U(scn->scn_phys.scn_func, ==, POOL_SCAN_RESILVER);
1645                 zio_flags |= ZIO_FLAG_RESILVER;
1648                 zio_priority = ZIO_PRIORITY_RESILVER;
1646                 needs_io = B_FALSE;
1647                 scan_delay = zfs_resilver_delay;
1648         }

1650         /* If it's an intent log block, failure is expected. */
1651         if (zb->zb_level == ZB_ZIL_LEVEL)
1652                 zio_flags |= ZIO_FLAG_SPECULATIVE;

1654         for (int d = 0; d < BP_GET_NDVAS(bp); d++) {
1655                 vdev_t *vd = vdev_lookup_top(spa,
1656                     DVA_GET_VDEV(&bp->blk_dva[d]));

1658                 /*
1659                  * Keep track of how much data we've examined so that
1660                  * zpool(1M) status can make useful progress reports.
1661                  */
1662                 scn->scn_phys.scn_examined += DVA_GET_ASIZE(&bp->blk_dva[d]);
1663                 spa->spa_scan_pass_exam += DVA_GET_ASIZE(&bp->blk_dva[d]);

1665                 /* if it's a resilver, this may not be in the target range */
1666                 if (!needs_io) {
1667                         if (DVA_GET_GANG(&bp->blk_dva[d])) {
1668                                 /*
1669                                  * Gang members may be spread across multiple
1670                                  * vdevs, so the best estimate we have is the
1671                                  * scrub range, which has already been checked.
1672                                  * XXX -- it would be better to change our
```



```
1673                                  * allocation policy to ensure that all
1674                                  * gang members reside on the same vdev.
1675                                  */
1676                                 needs_io = B_TRUE;
1677                         } else {
1678                                 needs_io = vdev_dtl_contains(vd, DTL_PARTIAL,
1679                                     phys_birth, 1);
1680                         }
1681                 }
1682         }

1684         if (needs_io && !zfs_no_scrub_io) {
1685                 vdev_t *rvd = spa->spa_root_vdev;
1686                 uint64_t maxinflight = rvd->vdev_children * zfs_top_maxinflight;
1687                 void *data = zio_data_buf_alloc(size);

1689                 mutex_enter(&spa->spa_scrub_lock);
1690                 while (spa->spa_scrub_inflight >= maxinflight)
1691                         cv_wait(&spa->spa_scrub_io_cv, &spa->spa_scrub_lock);
1692                 spa->spa_scrub_inflight++;
1693                 mutex_exit(&spa->spa_scrub_lock);

1695                 /*
1696                  * If we're seeing recent (zfs_scan_idle) "important" I/Os
1697                  * then throttle our workload to limit the impact of a scan.
1698                  */
1699                 if (ddi_get_lbolt64() - spa->spa_last_io <= zfs_scan_idle)
1700                         delay(scan_delay);

1702                 zio_nowait(zio_read(NULL, spa, bp, data, size,
1703                     dsl_scan_scrub_done, NULL, ZIO_PRIORITY_SCRUB,
1706                     dsl_scan_scrub_done, NULL, zio_priority,
1704                     zio_flags, zb));
1705         }

1707         /* do not relocate this block */
1708         return (0);
1709 }
______unchanged_portion_omitted_
```

```
**********************************************************
  175934 Thu Aug 22 16:15:16 2013
new/usr/src/uts/common/fs/zfs/spa.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 * SPA: Storage Pool Allocator
 *
 * This file contains all the routines used when modifying on-disk SPA state.
 * This includes opening, importing, destroying, exporting a pool, and syncing a
 * pool.
 */

#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/ddt.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/metaslab_impl.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_objset.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
```



```
#include <sys/dsl_synctask.h>
#include <sys/fs/zfs.h>
#include <sys/arc.h>
#include <sys/callb.h>
#include <sys/systeminfo.h>
#include <sys/spa_boot.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_scan.h>
#include <sys/zfeature.h>
#include <sys/dsl_destroy.h>

#ifdef  _KERNEL
#include <sys/bootprops.h>
#include <sys/callb.h>
#include <sys/cpupart.h>
#include <sys/pool.h>
#include <sys/sysdc.h>
#include <sys/zone.h>
#endif  /* _KERNEL */

#include "zfs_prop.h"
#include "zfs_comutil.h"

/*
 * The interval, in seconds, at which failed configuration cache file writes
 * should be retried.
 */
static int zfs_ccw_retry_interval = 300;

typedef enum zti_modes {
        ZTI_MODE_FIXED,                 /* value is # of threads (min 1) */
        ZTI_MODE_ONLINE_PERCENT,        /* value is % of online CPUs */
        ZTI_MODE_BATCH,                 /* cpu-intensive; value is ignored */
        ZTI_MODE_NULL,                  /* don't create a taskq */
        ZTI_NMODES
} zti_modes_t;

#define ZTI_P(n, q)     { ZTI_MODE_FIXED, (n), (q) }
#define ZTI_PCT(n)      { ZTI_MODE_ONLINE_PERCENT, (n), 1 }
#define ZTI_BATCH       { ZTI_MODE_BATCH, 0, 1 }
#define ZTI_NULL        { ZTI_MODE_NULL, 0, 0 }

#define ZTI_N(n)        ZTI_P(n, 1)
#define ZTI_ONE         ZTI_N(1)

typedef struct zio_taskq_info {
        zti_modes_t zti_mode;
        uint_t zti_value;
        uint_t zti_count;
} zio_taskq_info_t;
```

*_____unchanged_portion_omitted_*

```
static void spa_sync_version(void *arg, dmu_tx_t *tx);
static void spa_sync_props(void *arg, dmu_tx_t *tx);
static boolean_t spa_has_active_shared_spare(spa_t *spa);
static int spa_load_impl(spa_t *spa, uint64_t, nvlist_t *config,
    spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
    char **ereport);
static void spa_vdev_resilver_done(spa_t *spa);

uint_t          zio_taskq_batch_pct = 75;       /* 1 thread per cpu in pset */
uint_t          zio_taskq_batch_pct = 100;      /* 1 thread per cpu in pset */
id_t            zio_taskq_psrset_bind = PS_NONE;
boolean_t       zio_taskq_sysdc = B_TRUE;       /* use SDC scheduling class */
uint_t          zio_taskq_basedc = 80;          /* base duty cycle */
```

```
152 boolean_t       spa_create_process = B_TRUE;    /* no process ==> no sysdc */
153 extern int      zfs_sync_pass_deferred_free;

155 /*
156  * This (illegal) pool name is used when temporarily importing a spa_t in order
157  * to get the vdev stats associated with the imported devices.
158  */
159 #define TRYIMPORT_NAME  "$import"

161 /*
162  * ==========================================================================
163  * SPA properties routines
164  * ==========================================================================
165  */

167 /*
168  * Add a (source=src, propname=propval) list to an nvlist.
169  */
170 static void
171 spa_prop_add_list(nvlist_t *nvl, zpool_prop_t prop, char *strval,
172     uint64_t intval, zprop_source_t src)
173 {
174         const char *propname = zpool_prop_to_name(prop);
175         nvlist_t *propval;

177         VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
178         VERIFY(nvlist_add_uint64(propval, ZPROP_SOURCE, src) == 0);

180         if (strval != NULL)
181                 VERIFY(nvlist_add_string(propval, ZPROP_VALUE, strval) == 0);
182         else
183                 VERIFY(nvlist_add_uint64(propval, ZPROP_VALUE, intval) == 0);

185         VERIFY(nvlist_add_nvlist(nvl, propname, propval) == 0);
186         nvlist_free(propval);
187 }
```

*unchanged_portion_omitted_*

```
820 static void
821 spa_taskqs_init(spa_t *spa, zio_type_t t, zio_taskq_type_t q)
822 {
823         const zio_taskq_info_t *ztip = &zio_taskqs[t][q];
824         enum zti_modes mode = ztip->zti_mode;
825         uint_t value = ztip->zti_value;
826         uint_t count = ztip->zti_count;
827         spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
828         char name[32];
829         uint_t flags = 0;
830         boolean_t batch = B_FALSE;

832         if (mode == ZTI_MODE_NULL) {
833                 tqs->stqs_count = 0;
834                 tqs->stqs_taskq = NULL;
835                 return;
836         }

838         ASSERT3U(count, >, 0);

840         tqs->stqs_count = count;
841         tqs->stqs_taskq = kmem_alloc(count * sizeof (taskq_t *), KM_SLEEP);

845         for (uint_t i = 0; i < count; i++) {
846                 taskq_t *tq;

843         switch (mode) {
844         case ZTI_MODE_FIXED:
```



```
845                 ASSERT3U(value, >=, 1);
846                 value = MAX(value, 1);
847                 break;

849         case ZTI_MODE_BATCH:
850                 batch = B_TRUE;
851                 flags |= TASKQ_THREADS_CPU_PCT;
852                 value = zio_taskq_batch_pct;
853                 break;

860                 case ZTI_MODE_ONLINE_PERCENT:
861                         flags |= TASKQ_THREADS_CPU_PCT;
862                         break;

855         default:
856                 panic("unrecognized mode for %s_%s taskq (%u:%u) in "
857                     "spa_activate()",
858                     zio_type_name[t], zio_taskq_types[q], mode, value);
859                 break;
860         }

862         for (uint_t i = 0; i < count; i++) {
863                 taskq_t *tq;

865                 if (count > 1) {
866                         (void) snprintf(name, sizeof (name), "%s_%s_%u",
867                             zio_type_name[t], zio_taskq_types[q], i);
868                 } else {
869                         (void) snprintf(name, sizeof (name), "%s_%s",
870                             zio_type_name[t], zio_taskq_types[q]);
871                 }

873                 if (zio_taskq_sysdc && spa->spa_proc != &p0) {
874                         if (batch)
875                                 flags |= TASKQ_DC_BATCH;

877                         tq = taskq_create_sysdc(name, value, 50, INT_MAX,
878                             spa->spa_proc, zio_taskq_basedc, flags);
879                 } else {
880                         pri_t pri = maxclsyspri;
881                         /*
882                          * The write issue taskq can be extremely CPU
883                          * intensive.  Run it at slightly lower priority
884                          * than the other taskqs.
885                          */
886                         if (t == ZIO_TYPE_WRITE && q == ZIO_TASKQ_ISSUE)
887                                 pri--;

889                         tq = taskq_create_proc(name, value, pri, 50,
886                         tq = taskq_create_proc(name, value, maxclsyspri, 50,
890                             INT_MAX, spa->spa_proc, flags);
891                 }

893                 tqs->stqs_taskq[i] = tq;
894         }
895 }
```

*unchanged_portion_omitted_*

```
5743 /*
5744  * Note: this simple function is not inlined to make it easier to dtrace the
5745  * amount of time spent syncing frees.
5746  */
5747 static void
5748 spa_sync_frees(spa_t *spa, bplist_t *bpl, dmu_tx_t *tx)
5749 {
5750         zio_t *zio = zio_root(spa, NULL, NULL, 0);
```

```
5751         bplist_iterate(bpl, spa_free_sync_cb, zio, tx);
5752         VERIFY(zio_wait(zio) == 0);
5753 }

5755 /*
5756  * Note: this simple function is not inlined to make it easier to dtrace the
5757  * amount of time spent syncing deferred frees.
5758  */
5759 static void
5760 spa_sync_deferred_frees(spa_t *spa, dmu_tx_t *tx)
5761 {
5762         zio_t *zio = zio_root(spa, NULL, NULL, 0);
5763         VERIFY3U(bpobj_iterate(&spa->spa_deferred_bpobj,
5764             spa_free_sync_cb, zio, tx), ==, 0);
5765         VERIFY0(zio_wait(zio));
5766 }


5769 static void
5770 spa_sync_nvlist(spa_t *spa, uint64_t obj, nvlist_t *nv, dmu_tx_t *tx)
5771 {
5772         char *packed = NULL;
5773         size_t bufsize;
5774         size_t nvsize = 0;
5775         dmu_buf_t *db;

5777         VERIFY(nvlist_size(nv, &nvsize, NV_ENCODE_XDR) == 0);

5779         /*
5780          * Write full (SPA_CONFIG_BLOCKSIZE) blocks of configuration
5781          * information.  This avoids the dbuf_will_dirty() path and
5782          * saves us a pre-read to get data we don't actually care about.
5783          */
5784         bufsize = P2ROUNDUP((uint64_t)nvsize, SPA_CONFIG_BLOCKSIZE);
5785         packed = kmem_alloc(bufsize, KM_SLEEP);

5787         VERIFY(nvlist_pack(nv, &packed, &nvsize, NV_ENCODE_XDR,
5788             KM_SLEEP) == 0);
5789         bzero(packed + nvsize, bufsize - nvsize);

5791         dmu_write(spa->spa_meta_objset, obj, 0, bufsize, packed, tx);

5793         kmem_free(packed, bufsize);

5795         VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset, obj, FTAG, &db));
5796         dmu_buf_will_dirty(db, tx);
5797         *(uint64_t *)db->db_data = nvsize;
5798         dmu_buf_rele(db, FTAG);
5799 }
```

*_____unchanged_portion_omitted_*

```
6086 /*
6087  * Sync the specified transaction group.  New blocks may be dirtied as
6088  * part of the process, so we iterate until it converges.
6089  */
6090 void
6091 spa_sync(spa_t *spa, uint64_t txg)
6092 {
6093         dsl_pool_t *dp = spa->spa_dsl_pool;
6094         objset_t *mos = spa->spa_meta_objset;
6066         bpobj_t *defer_bpo = &spa->spa_deferred_bpobj;
6095         bplist_t *free_bpl = &spa->spa_free_bplist[txg & TXG_MASK];
6096         vdev_t *rvd = spa->spa_root_vdev;
6097         vdev_t *vd;
6098         dmu_tx_t *tx;
6099         int error;
```



```
6101         VERIFY(spa_writeable(spa));

6103         /*
6104          * Lock out configuration changes.
6105          */
6106         spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);

6108         spa->spa_syncing_txg = txg;
6109         spa->spa_sync_pass = 0;

6111         /*
6112          * If there are any pending vdev state changes, convert them
6113          * into config changes that go out with this transaction group.
6114          */
6115         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
6116         while (list_head(&spa->spa_state_dirty_list) != NULL) {
6117                 /*
6118                  * We need the write lock here because, for aux vdevs,
6119                  * calling vdev_config_dirty() modifies sav_config.
6120                  * This is ugly and will become unnecessary when we
6121                  * eliminate the aux vdev wart by integrating all vdevs
6122                  * into the root vdev tree.
6123                  */
6124                 spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
6125                 spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_WRITER);
6126                 while ((vd = list_head(&spa->spa_state_dirty_list)) != NULL) {
6127                         vdev_state_clean(vd);
6128                         vdev_config_dirty(vd);
6129                 }
6130                 spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
6131                 spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
6132         }
6133         spa_config_exit(spa, SCL_STATE, FTAG);

6135         tx = dmu_tx_create_assigned(dp, txg);

6137         spa->spa_sync_starttime = gethrtime();
6138         VERIFY(cyclic_reprogram(spa->spa_deadman_cycid,
6139             spa->spa_sync_starttime + spa->spa_deadman_synctime));

6141         /*
6142          * If we are upgrading to SPA_VERSION_RAIDZ_DEFLATE this txg,
6143          * set spa_deflate if we have no raid-z vdevs.
6144          */
6145         if (spa->spa_ubsync.ub_version < SPA_VERSION_RAIDZ_DEFLATE &&
6146             spa->spa_uberblock.ub_version >= SPA_VERSION_RAIDZ_DEFLATE) {
6147                 int i;

6149                 for (i = 0; i < rvd->vdev_children; i++) {
6150                         vd = rvd->vdev_child[i];
6151                         if (vd->vdev_deflate_ratio != SPA_MINBLOCKSIZE)
6152                                 break;
6153                 }
6154                 if (i == rvd->vdev_children) {
6155                         spa->spa_deflate = TRUE;
6156                         VERIFY(0 == zap_add(spa->spa_meta_objset,
6157                             DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
6158                             sizeof (uint64_t), 1, &spa->spa_deflate, tx));
6159                 }
6160         }

6162         /*
6163          * If anything has changed in this txg, or if someone is waiting
6164          * for this txg to sync (eg, spa_vdev_remove()), push the
6165          * deferred frees from the previous txg.  If not, leave them
```

```
6166          * alone so that we don't generate work on an otherwise idle
6167          * system.
6168          */
6169         if (!txg_list_empty(&dp->dp_dirty_datasets, txg) ||
6170             !txg_list_empty(&dp->dp_dirty_dirs, txg) ||
6171             !txg_list_empty(&dp->dp_sync_tasks, txg) ||
6172             ((dsl_scan_active(dp->dp_scan) ||
6173             txg_sync_waiting(dp)) && !spa_shutting_down(spa))) {
6174                 spa_sync_deferred_frees(spa, tx);
6146                 zio_t *zio = zio_root(spa, NULL, NULL, 0);
6147                 VERIFY3U(bpobj_iterate(defer_bpo,
6148                     spa_free_sync_cb, zio, tx), ==, 0);
6149                 VERIFY0(zio_wait(zio));
6175         }

6177         /*
6178          * Iterate to convergence.
6179          */
6180         do {
6181                 int pass = ++spa->spa_sync_pass;

6183                 spa_sync_config_object(spa, tx);
6184                 spa_sync_aux_dev(spa, &spa->spa_spares, tx,
6185                     ZPOOL_CONFIG_SPARES, DMU_POOL_SPARES);
6186                 spa_sync_aux_dev(spa, &spa->spa_l2cache, tx,
6187                     ZPOOL_CONFIG_L2CACHE, DMU_POOL_L2CACHE);
6188                 spa_errlog_sync(spa, txg);
6189                 dsl_pool_sync(dp, txg);

6191                 if (pass < zfs_sync_pass_deferred_free) {
6192                         spa_sync_frees(spa, free_bpl, tx);
6167                         zio_t *zio = zio_root(spa, NULL, NULL, 0);
6168                         bplist_iterate(free_bpl, spa_free_sync_cb,
6169                             zio, tx);
6170                         VERIFY(zio_wait(zio) == 0);
6193                 } else {
6194                         bplist_iterate(free_bpl, bpobj_enqueue_cb,
6195                             &spa->spa_deferred_bpobj, tx);
6173                             defer_bpo, tx);
6196                 }

6198                 ddt_sync(spa, txg);
6199                 dsl_scan_sync(dp, tx);

6201                 while (vd = txg_list_remove(&spa->spa_vdev_txg_list, txg))
6202                         vdev_sync(vd, txg);

6204                 if (pass == 1)
6205                         spa_sync_upgrades(spa, tx);

6207         } while (dmu_objset_is_dirty(mos, txg));

6209         /*
6210          * Rewrite the vdev configuration (which includes the uberblock)
6211          * to commit the transaction group.
6212          *
6213          * If there are no dirty vdevs, we sync the uberblock to a few
6214          * random top-level vdevs that are known to be visible in the
6215          * config cache (see spa_vdev_add() for a complete description).
6216          * If there *are* dirty vdevs, sync the uberblock to all vdevs.
6217          */
6218         for (;;) {
6219                 /*
6220                  * We hold SCL_STATE to prevent vdev open/close/etc.
6221                  * while we're attempting to write the vdev labels.
6222                  */
```



```
6223                 spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);

6225                 if (list_is_empty(&spa->spa_config_dirty_list)) {
6226                         vdev_t *svd[SPA_DVAS_PER_BP];
6227                         int svdcount = 0;
6228                         int children = rvd->vdev_children;
6229                         int c0 = spa_get_random(children);

6231                         for (int c = 0; c < children; c++) {
6232                                 vd = rvd->vdev_child[(c0 + c) % children];
6233                                 if (vd->vdev_ms_array == 0 || vd->vdev_islog)
6234                                         continue;
6235                                 svd[svdcount++] = vd;
6236                                 if (svdcount == SPA_DVAS_PER_BP)
6237                                         break;
6238                         }
6239                         error = vdev_config_sync(svd, svdcount, txg, B_FALSE);
6240                         if (error != 0)
6241                                 error = vdev_config_sync(svd, svdcount, txg,
6242                                     B_TRUE);
6243                 } else {
6244                         error = vdev_config_sync(rvd->vdev_child,
6245                             rvd->vdev_children, txg, B_FALSE);
6246                         if (error != 0)
6247                                 error = vdev_config_sync(rvd->vdev_child,
6248                                     rvd->vdev_children, txg, B_TRUE);
6249                 }

6251                 if (error == 0)
6252                         spa->spa_last_synced_guid = rvd->vdev_guid;

6254                 spa_config_exit(spa, SCL_STATE, FTAG);

6256                 if (error == 0)
6257                         break;
6258                 zio_suspend(spa, NULL);
6259                 zio_resume_wait(spa);
6260         }
6261         dmu_tx_commit(tx);

6263         VERIFY(cyclic_reprogram(spa->spa_deadman_cycid, CY_INFINITY));

6265         /*
6266          * Clear the dirty config list.
6267          */
6268         while ((vd = list_head(&spa->spa_config_dirty_list)) != NULL)
6269                 vdev_config_clean(vd);

6271         /*
6272          * Now that the new config has synced transactionally,
6273          * let it become visible to the config cache.
6274          */
6275         if (spa->spa_config_syncing != NULL) {
6276                 spa_config_set(spa, spa->spa_config_syncing);
6277                 spa->spa_config_txg = txg;
6278                 spa->spa_config_syncing = NULL;
6279         }

6281         spa->spa_ubsync = spa->spa_uberblock;

6283         dsl_pool_sync_done(dp, txg);

6285         /*
6286          * Update usable space statistics.
6287          */
6288         while (vd = txg_list_remove(&spa->spa_vdev_txg_list, TXG_CLEAN(txg)))
```

```
6289                 vdev_sync_done(vd, txg);

6291         spa_update_dspace(spa);

6293         /*
6294          * It had better be the case that we didn't dirty anything
6295          * since vdev_config_sync().
6296          */
6297         ASSERT(txg_list_empty(&dp->dp_dirty_datasets, txg));
6298         ASSERT(txg_list_empty(&dp->dp_dirty_dirs, txg));
6299         ASSERT(txg_list_empty(&spa->spa_vdev_txg_list, txg));

6301         spa->spa_sync_pass = 0;

6303         spa_config_exit(spa, SCL_CONFIG, FTAG);

6305         spa_handle_ignored_writes(spa);

6307         /*
6308          * If any async tasks have been requested, kick them off.
6309          */
6310         spa_async_dispatch(spa);
6311 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   46056 Thu Aug 22 16:15:18 2013
new/usr/src/uts/common/fs/zfs/spa_misc.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2013 by Delphix. All rights reserved.
  24  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  25  */

  27 #include <sys/zfs_context.h>
  28 #include <sys/spa_impl.h>
  29 #include <sys/spa_boot.h>
  30 #include <sys/zio.h>
  31 #include <sys/zio_checksum.h>
  32 #include <sys/zio_compress.h>
  33 #include <sys/dmu.h>
  34 #include <sys/dmu_tx.h>
  35 #include <sys/zap.h>
  36 #include <sys/zil.h>
  37 #include <sys/vdev_impl.h>
  38 #include <sys/metaslab.h>
  39 #include <sys/uberblock_impl.h>
  40 #include <sys/txg.h>
  41 #include <sys/avl.h>
  42 #include <sys/unique.h>
  43 #include <sys/dsl_pool.h>
  44 #include <sys/dsl_dir.h>
  45 #include <sys/dsl_prop.h>
  46 #include <sys/dsl_scan.h>
  47 #include <sys/fs/zfs.h>
  48 #include <sys/metaslab_impl.h>
  49 #include <sys/arc.h>
  50 #include <sys/ddt.h>
  51 #include "zfs_prop.h"
  52 #include "zfeature_common.h"

  54 /*
  55  * SPA locking
  56  *
  57  * There are four basic locks for managing spa_t structures:
  58  *
```



```
  59  * spa_namespace_lock (global mutex)
  60  *
  61  *      This lock must be acquired to do any of the following:
  62  *
  63  *              - Lookup a spa_t by name
  64  *              - Add or remove a spa_t from the namespace
  65  *              - Increase spa_refcount from non-zero
  66  *              - Check if spa_refcount is zero
  67  *              - Rename a spa_t
  68  *              - add/remove/attach/detach devices
  69  *              - Held for the duration of create/destroy/import/export
  70  *
  71  *      It does not need to handle recursion.  A create or destroy may
  72  *      reference objects (files or zvols) in other pools, but by
  73  *      definition they must have an existing reference, and will never need
  74  *      to lookup a spa_t by name.
  75  *
  76  * spa_refcount (per-spa refcount_t protected by mutex)
  77  *
  78  *      This reference count keep track of any active users of the spa_t.  The
  79  *      spa_t cannot be destroyed or freed while this is non-zero.  Internally,
  80  *      the refcount is never really 'zero' - opening a pool implicitly keeps
  81  *      some references in the DMU.  Internally we check against spa_minref, but
  82  *      present the image of a zero/non-zero value to consumers.
  83  *
  84  * spa_config_lock[] (per-spa array of rwlocks)
  85  *
  86  *      This protects the spa_t from config changes, and must be held in
  87  *      the following circumstances:
  88  *
  89  *              - RW_READER to perform I/O to the spa
  90  *              - RW_WRITER to change the vdev config
  91  *
  92  * The locking order is fairly straightforward:
  93  *
  94  *              spa_namespace_lock      ->      spa_refcount
  95  *
  96  *      The namespace lock must be acquired to increase the refcount from 0
  97  *      or to check if it is zero.
  98  *
  99  *              spa_refcount            ->      spa_config_lock[]
 100  *
 101  *      There must be at least one valid reference on the spa_t to acquire
 102  *      the config lock.
 103  *
 104  *              spa_namespace_lock      ->      spa_config_lock[]
 105  *
 106  *      The namespace lock must always be taken before the config lock.
 107  *
 108  *
 109  * The spa_namespace_lock can be acquired directly and is globally visible.
 110  *
 111  * The namespace is manipulated using the following functions, all of which
 112  * require the spa_namespace_lock to be held.
 113  *
 114  *      spa_lookup()            Lookup a spa_t by name.
 115  *
 116  *      spa_add()               Create a new spa_t in the namespace.
 117  *
 118  *      spa_remove()            Remove a spa_t from the namespace.  This also
 119  *                              frees up any memory associated with the spa_t.
 120  *
 121  *      spa_next()              Returns the next spa_t in the system, or the
 122  *                              first if NULL is passed.
 123  *
 124  *      spa_evict_all()         Shutdown and remove all spa_t structures in
```

```
125  *                              the system.
126  *
127  *      spa_guid_exists()       Determine whether a pool/device guid exists.
128  *
129  * The spa_refcount is manipulated using the following functions:
130  *
131  *      spa_open_ref()          Adds a reference to the given spa_t.  Must be
132  *                              called with spa_namespace_lock held if the
133  *                              refcount is currently zero.
134  *
135  *      spa_close()             Remove a reference from the spa_t.  This will
136  *                              not free the spa_t or remove it from the
137  *                              namespace.  No locking is required.
138  *
139  *      spa_refcount_zero()     Returns true if the refcount is currently
140  *                              zero.  Must be called with spa_namespace_lock
141  *                              held.
142  *
143  * The spa_config_lock[] is an array of rwlocks, ordered as follows:
144  * SCL_CONFIG > SCL_STATE > SCL_ALLOC > SCL_ZIO > SCL_FREE > SCL_VDEV.
145  * spa_config_lock[] is manipulated with spa_config_{enter,exit,held}().
146  *
147  * To read the configuration, it suffices to hold one of these locks as reader.
148  * To modify the configuration, you must hold all locks as writer.  To modify
149  * vdev state without altering the vdev tree's topology (e.g. online/offline),
150  * you must hold SCL_STATE and SCL_ZIO as writer.
151  *
152  * We use these distinct config locks to avoid recursive lock entry.
153  * For example, spa_sync() (which holds SCL_CONFIG as reader) induces
154  * block allocations (SCL_ALLOC), which may require reading space maps
155  * from disk (dmu_read() -> zio_read() -> SCL_ZIO).
156  *
157  * The spa config locks cannot be normal rwlocks because we need the
158  * ability to hand off ownership.  For example, SCL_ZIO is acquired
159  * by the issuing thread and later released by an interrupt thread.
160  * They do, however, obey the usual write-wanted semantics to prevent
161  * writer (i.e. system administrator) starvation.
162  *
163  * The lock acquisition rules are as follows:
164  *
165  * SCL_CONFIG
166  *      Protects changes to the vdev tree topology, such as vdev
167  *      add/remove/attach/detach.  Protects the dirty config list
168  *      (spa_config_dirty_list) and the set of spares and l2arc devices.
169  *
170  * SCL_STATE
171  *      Protects changes to pool state and vdev state, such as vdev
172  *      online/offline/fault/degrade/clear.  Protects the dirty state list
173  *      (spa_state_dirty_list) and global pool state (spa_state).
174  *
175  * SCL_ALLOC
176  *      Protects changes to metaslab groups and classes.
177  *      Held as reader by metaslab_alloc() and metaslab_claim().
178  *
179  * SCL_ZIO
180  *      Held by bp-level zios (those which have no io_vd upon entry)
181  *      to prevent changes to the vdev tree.  The bp-level zio implicitly
182  *      protects all of its vdev child zios, which do not hold SCL_ZIO.
183  *
184  * SCL_FREE
185  *      Protects changes to metaslab groups and classes.
186  *      Held as reader by metaslab_free().  SCL_FREE is distinct from
187  *      SCL_ALLOC, and lower than SCL_ZIO, so that we can safely free
188  *      blocks in zio_done() while another i/o that holds either
189  *      SCL_ALLOC or SCL_ZIO is waiting for this i/o to complete.
190  *
```



```
191  * SCL_VDEV
192  *      Held as reader to prevent changes to the vdev tree during trivial
193  *      inquiries such as bp_get_dsize().  SCL_VDEV is distinct from the
194  *      other locks, and lower than all of them, to ensure that it's safe
195  *      to acquire regardless of caller context.
196  *
197  * In addition, the following rules apply:
198  *
199  * (a)  spa_props_lock protects pool properties, spa_config and spa_config_list.
200  *      The lock ordering is SCL_CONFIG > spa_props_lock.
201  *
202  * (b)  I/O operations on leaf vdevs.  For any zio operation that takes
203  *      an explicit vdev_t argument -- such as zio_ioctl(), zio_read_phys(),
204  *      or zio_write_phys() -- the caller must ensure that the config cannot
205  *      cannot change in the interim, and that the vdev cannot be reopened.
206  *      SCL_STATE as reader suffices for both.
207  *
208  * The vdev configuration is protected by spa_vdev_enter() / spa_vdev_exit().
209  *
210  *      spa_vdev_enter()        Acquire the namespace lock and the config lock
211  *                              for writing.
212  *
213  *      spa_vdev_exit()         Release the config lock, wait for all I/O
214  *                              to complete, sync the updated configs to the
215  *                              cache, and release the namespace lock.
216  *
217  * vdev state is protected by spa_vdev_state_enter() / spa_vdev_state_exit().
218  * Like spa_vdev_enter/exit, these are convenience wrappers -- the actual
219  * locking is, always, based on spa_namespace_lock and spa_config_lock[].
220  *
221  * spa_rename() is also implemented within this file since it requires
222  * manipulation of the namespace.
223  */

225 static avl_tree_t spa_namespace_avl;
226 kmutex_t spa_namespace_lock;
227 static kcondvar_t spa_namespace_cv;
228 static int spa_active_count;
229 int spa_max_replication_override = SPA_DVAS_PER_BP;

231 static kmutex_t spa_spare_lock;
232 static avl_tree_t spa_spare_avl;
233 static kmutex_t spa_l2cache_lock;
234 static avl_tree_t spa_l2cache_avl;

236 kmem_cache_t *spa_buffer_pool;
237 int spa_mode_global;

239 #ifdef ZFS_DEBUG
240 /* Everything except dprintf and spa is on by default in debug builds */
241 int zfs_flags = ~(ZFS_DEBUG_DPRINTF | ZFS_DEBUG_SPA);
242 #else
243 int zfs_flags = 0;
244 #endif

246 /*
247  * zfs_recover can be set to nonzero to attempt to recover from
248  * otherwise-fatal errors, typically caused by on-disk corruption.  When
249  * set, calls to zfs_panic_recover() will turn into warning messages.
250  */
251 int zfs_recover = 0;

253 /*
254  * Expiration time in milliseconds. This value has two meanings. First it is
255  * used to determine when the spa_deadman() logic should fire. By default the
256  * spa_deadman() will fire if spa_sync() has not completed in 1000 seconds.
```

```
 257  * Secondly, the value determines if an I/O is considered "hung". Any I/O that
 258  * has not completed in zfs_deadman_synctime_ms is considered "hung" resulting
 259  * in a system panic.
 260  */
 261 uint64_t zfs_deadman_synctime_ms = 1000000ULL;
 253 extern int zfs_txg_synctime_ms;

 263 /*
 264  * Check time in milliseconds. This defines the frequency at which we check
 265  * for hung I/O.
 256  * Expiration time in units of zfs_txg_synctime_ms. This value has two
 257  * meanings. First it is used to determine when the spa_deadman logic
 258  * should fire. By default the spa_deadman will fire if spa_sync has
 259  * not completed in 1000 * zfs_txg_synctime_ms (i.e. 1000 seconds).
 260  * Secondly, the value determines if an I/O is considered "hung".
 261  * Any I/O that has not completed in zfs_deadman_synctime is considered
 262  * "hung" resulting in a system panic.
 266  */
 267 uint64_t zfs_deadman_checktime_ms = 5000ULL;
 264 uint64_t zfs_deadman_synctime = 1000ULL;

 269 /*
 270  * Override the zfs deadman behavior via /etc/system. By default the
 271  * deadman is enabled except on VMware and sparc deployments.
 272  */
 273 int zfs_deadman_enabled = -1;

 275 /*
 276  * The worst case is single-sector max-parity RAID-Z blocks, in which
 277  * case the space requirement is exactly (VDEV_RAIDZ_MAXPARITY + 1)
 278  * times the size; so just assume that.  Add to this the fact that
 279  * we can have up to 3 DVAs per bp, and one more factor of 2 because
 280  * the block may be dittoed with up to 3 DVAs by ddt_sync().  All together,
 281  * the worst case is:
 282  *     (VDEV_RAIDZ_MAXPARITY + 1) * SPA_DVAS_PER_BP * 2 == 24
 283  */
 284 int spa_asize_inflation = 24;

 286 /*
 287  * ==========================================================================
 288  * SPA config locking
 289  * ==========================================================================
 290  */
 291 static void
 292 spa_config_lock_init(spa_t *spa)
 293 {
 294         for (int i = 0; i < SCL_LOCKS; i++) {
 295                 spa_config_lock_t *scl = &spa->spa_config_lock[i];
 296                 mutex_init(&scl->scl_lock, NULL, MUTEX_DEFAULT, NULL);
 297                 cv_init(&scl->scl_cv, NULL, CV_DEFAULT, NULL);
 298                 refcount_create_untracked(&scl->scl_count);
 299                 scl->scl_writer = NULL;
 300                 scl->scl_write_wanted = 0;
 301         }
 302 }
_____unchanged_portion_omitted_

 467 /*
 468  * Create an uninitialized spa_t with the given name.  Requires
 469  * spa_namespace_lock.  The caller must ensure that the spa_t doesn't already
 470  * exist by calling spa_lookup() first.
 471  */
 472 spa_t *
 473 spa_add(const char *name, nvlist_t *config, const char *altroot)
 474 {
 475         spa_t *spa;
```



```
 476         spa_config_dirent_t *dp;
 477         cyc_handler_t hdlr;
 478         cyc_time_t when;

 480         ASSERT(MUTEX_HELD(&spa_namespace_lock));

 482         spa = kmem_zalloc(sizeof (spa_t), KM_SLEEP);

 484         mutex_init(&spa->spa_async_lock, NULL, MUTEX_DEFAULT, NULL);
 485         mutex_init(&spa->spa_errlist_lock, NULL, MUTEX_DEFAULT, NULL);
 486         mutex_init(&spa->spa_errlog_lock, NULL, MUTEX_DEFAULT, NULL);
 487         mutex_init(&spa->spa_history_lock, NULL, MUTEX_DEFAULT, NULL);
 488         mutex_init(&spa->spa_proc_lock, NULL, MUTEX_DEFAULT, NULL);
 489         mutex_init(&spa->spa_props_lock, NULL, MUTEX_DEFAULT, NULL);
 490         mutex_init(&spa->spa_scrub_lock, NULL, MUTEX_DEFAULT, NULL);
 491         mutex_init(&spa->spa_suspend_lock, NULL, MUTEX_DEFAULT, NULL);
 492         mutex_init(&spa->spa_vdev_top_lock, NULL, MUTEX_DEFAULT, NULL);
 493         mutex_init(&spa->spa_iokstat_lock, NULL, MUTEX_DEFAULT, NULL);

 495         cv_init(&spa->spa_async_cv, NULL, CV_DEFAULT, NULL);
 496         cv_init(&spa->spa_proc_cv, NULL, CV_DEFAULT, NULL);
 497         cv_init(&spa->spa_scrub_io_cv, NULL, CV_DEFAULT, NULL);
 498         cv_init(&spa->spa_suspend_cv, NULL, CV_DEFAULT, NULL);

 500         for (int t = 0; t < TXG_SIZE; t++)
 501                 bplist_create(&spa->spa_free_bplist[t]);

 503         (void) strlcpy(spa->spa_name, name, sizeof (spa->spa_name));
 504         spa->spa_state = POOL_STATE_UNINITIALIZED;
 505         spa->spa_freeze_txg = UINT64_MAX;
 506         spa->spa_final_txg = UINT64_MAX;
 507         spa->spa_load_max_txg = UINT64_MAX;
 508         spa->spa_proc = &p0;
 509         spa->spa_proc_state = SPA_PROC_NONE;

 511         hdlr.cyh_func = spa_deadman;
 512         hdlr.cyh_arg = spa;
 513         hdlr.cyh_level = CY_LOW_LEVEL;

 515         spa->spa_deadman_synctime = MSEC2NSEC(zfs_deadman_synctime_ms);
 502         spa->spa_deadman_synctime = MSEC2NSEC(zfs_deadman_synctime *
 503             zfs_txg_synctime_ms);

 517         /*
 518          * This determines how often we need to check for hung I/Os after
 519          * the cyclic has already fired. Since checking for hung I/Os is
 520          * an expensive operation we don't want to check too frequently.
 521          * Instead wait for 5 seconds before checking again.
 509          * Instead wait for 5 synctimes before checking again.
 522          */
 523         when.cyt_interval = MSEC2NSEC(zfs_deadman_checktime_ms);
 511         when.cyt_interval = MSEC2NSEC(5 * zfs_txg_synctime_ms);
 524         when.cyt_when = CY_INFINITY;
 525         mutex_enter(&cpu_lock);
 526         spa->spa_deadman_cycid = cyclic_add(&hdlr, &when);
 527         mutex_exit(&cpu_lock);

 529         refcount_create(&spa->spa_refcount);
 530         spa_config_lock_init(spa);

 532         avl_add(&spa_namespace_avl, spa);

 534         /*
 535          * Set the alternate root, if there is one.
 536          */
 537         if (altroot) {
```

```
538                 spa->spa_root = spa_strdup(altroot);
539                 spa_active_count++;
540         }

542         /*
543          * Every pool starts with the default cachefile
544          */
545         list_create(&spa->spa_config_list, sizeof (spa_config_dirent_t),
546             offsetof(spa_config_dirent_t, scd_link));

548         dp = kmem_zalloc(sizeof (spa_config_dirent_t), KM_SLEEP);
549         dp->scd_path = altroot ? NULL : spa_strdup(spa_config_path);
550         list_insert_head(&spa->spa_config_list, dp);

552         VERIFY(nvlist_alloc(&spa->spa_load_info, NV_UNIQUE_NAME,
553             KM_SLEEP) == 0);

555         if (config != NULL) {
556                 nvlist_t *features;

558                 if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_FEATURES_FOR_READ,
559                     &features) == 0) {
560                         VERIFY(nvlist_dup(features, &spa->spa_label_features,
561                             0) == 0);
562                 }

564                 VERIFY(nvlist_dup(config, &spa->spa_config, 0) == 0);
565         }

567         if (spa->spa_label_features == NULL) {
568                 VERIFY(nvlist_alloc(&spa->spa_label_features, NV_UNIQUE_NAME,
569                     KM_SLEEP) == 0);
570         }

572         spa->spa_iokstat = kstat_create("zfs", 0, name,
573             "disk", KSTAT_TYPE_IO, 1, 0);
574         if (spa->spa_iokstat) {
575                 spa->spa_iokstat->ks_lock = &spa->spa_iokstat_lock;
576                 kstat_install(spa->spa_iokstat);
577         }

579         spa->spa_debug = ((zfs_flags & ZFS_DEBUG_SPA) != 0);

581         return (spa);
582 }
```

_____***unchanged_portion_omitted_***

```
1510 /* ARGSUSED */
1511 uint64_t
1512 spa_get_asize(spa_t *spa, uint64_t lsize)
1513 {
1514         return (lsize * spa_asize_inflation);
1502         /*
1503          * The worst case is single-sector max-parity RAID-Z blocks, in which
1504          * case the space requirement is exactly (VDEV_RAIDZ_MAXPARITY + 1)
1505          * times the size; so just assume that.  Add to this the fact that
1506          * we can have up to 3 DVAs per bp, and one more factor of 2 because
1507          * the block may be dittoed with up to 3 DVAs by ddt_sync().
1508          */
1509         return (lsize * (VDEV_RAIDZ_MAXPARITY + 1) * SPA_DVAS_PER_BP * 2);
1515 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4550 Thu Aug 22 16:15:19 2013
new/usr/src/uts/common/fs/zfs/sys/arc.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  84 void arc_space_consume(uint64_t space, arc_space_type_t type);
  85 void arc_space_return(uint64_t space, arc_space_type_t type);
  86 void *arc_data_buf_alloc(uint64_t space);
  87 void arc_data_buf_free(void *buf, uint64_t space);
  88 arc_buf_t *arc_buf_alloc(spa_t *spa, int size, void *tag,
  89     arc_buf_contents_t type);
  90 arc_buf_t *arc_loan_buf(spa_t *spa, int size);
  91 void arc_return_buf(arc_buf_t *buf, void *tag);
  92 void arc_loan_inuse_buf(arc_buf_t *buf, void *tag);
  93 void arc_buf_add_ref(arc_buf_t *buf, void *tag);
  94 boolean_t arc_buf_remove_ref(arc_buf_t *buf, void *tag);
  95 int arc_buf_size(arc_buf_t *buf);
  96 void arc_release(arc_buf_t *buf, void *tag);
  97 int arc_released(arc_buf_t *buf);
  98 int arc_has_callback(arc_buf_t *buf);
  99 void arc_buf_freeze(arc_buf_t *buf);
 100 void arc_buf_thaw(arc_buf_t *buf);
 101 boolean_t arc_buf_eviction_needed(arc_buf_t *buf);
 102 #ifdef ZFS_DEBUG
 103 int arc_referenced(arc_buf_t *buf);
 104 #endif

 106 int arc_read(zio_t *pio, spa_t *spa, const blkptr_t *bp,
 107     arc_done_func_t *done, void *private, zio_priority_t priority, int flags,
 107     arc_done_func_t *done, void *private, int priority, int flags,
 108     uint32_t *arc_flags, const zbookmark_t *zb);
 109 zio_t *arc_write(zio_t *pio, spa_t *spa, uint64_t txg,
 110     blkptr_t *bp, arc_buf_t *buf, boolean_t l2arc, boolean_t l2arc_compress,
 111     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *physdone,
 112     arc_done_func_t *done, void *private, zio_priority_t priority,
 113     int zio_flags, const zbookmark_t *zb);
 111     const zio_prop_t *zp, arc_done_func_t *ready, arc_done_func_t *done,
 112     void *private, int priority, int zio_flags, const zbookmark_t *zb);
 114 void arc_freed(spa_t *spa, const blkptr_t *bp);

 116 void arc_set_callback(arc_buf_t *buf, arc_evict_func_t *func, void *private);
 117 int arc_buf_evict(arc_buf_t *buf);

 119 void arc_flush(spa_t *spa);
 120 void arc_tempreserve_clear(uint64_t reserve);
 121 int arc_tempreserve_space(uint64_t reserve, uint64_t txg);

 123 void arc_init(void);
 124 void arc_fini(void);

 126 /*
 127  * Level 2 ARC
 128  */

 130 void l2arc_add_vdev(spa_t *spa, vdev_t *vd);
 131 void l2arc_remove_vdev(vdev_t *vd);
 132 boolean_t l2arc_vdev_present(vdev_t *vd);
 133 void l2arc_init(void);
 134 void l2arc_fini(void);
 135 void l2arc_start(void);
 136 void l2arc_stop(void);
```



```
 138 #ifndef _KERNEL
 139 extern boolean_t arc_watch;
 140 extern int arc_procfd;
 141 #endif

 143 #ifdef  __cplusplus
 144 }
______unchanged_portion_omitted_
```

```
**********************************************************
   10290 Thu Aug 22 16:15:20 2013
new/usr/src/uts/common/fs/zfs/sys/dbuf.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

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

#ifndef _SYS_DBUF_H
#define _SYS_DBUF_H

#include <sys/dmu.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/zfs_context.h>
#include <sys/refcount.h>
#include <sys/zrlock.h>

#ifdef __cplusplus
extern "C" {
#endif

#define IN_DMU_SYNC 2

/*
 * define flags for dbuf_read
 */

#define DB_RF_MUST_SUCCEED      (1 << 0)
#define DB_RF_CANFAIL           (1 << 1)
#define DB_RF_HAVESTRUCT        (1 << 2)
#define DB_RF_NOPREFETCH        (1 << 3)
#define DB_RF_NEVERWAIT         (1 << 4)
#define DB_RF_CACHED            (1 << 5)

/*
 * The simplified state transition diagram for dbufs looks like:
```



```
 *
 *              +----> READ ----+
 *              |               |
 *              |               V
 *  (alloc)-->UNCACHED       CACHED-->EVICTING-->(free)
 *              |               ^        ^
 *              |               |        |
 *              +----> FILL ----+        |
 *              |                        |
 *              |                        |
 *              +--------> NOFILL -------+
 */
typedef enum dbuf_states {
        DB_UNCACHED,
        DB_FILL,
        DB_NOFILL,
        DB_READ,
        DB_CACHED,
        DB_EVICTING
} dbuf_states_t;
```

*unchanged_portion_omitted_*

```
typedef struct dbuf_dirty_record {
        /* link on our parents dirty list */
        list_node_t dr_dirty_node;

        /* transaction group this data will sync in */
        uint64_t dr_txg;

        /* zio of outstanding write IO */
        zio_t *dr_zio;

        /* pointer back to our dbuf */
        struct dmu_buf_impl *dr_dbuf;

        /* pointer to next dirty record */
        struct dbuf_dirty_record *dr_next;

        /* pointer to parent dirty record */
        struct dbuf_dirty_record *dr_parent;

        /* How much space was changed to dsl_pool_dirty_space() for this? */
        unsigned int dr_accounted;

        union dirty_types {
                struct dirty_indirect {

                        /* protect access to list */
                        kmutex_t dr_mtx;

                        /* Our list of dirty children */
                        list_t dr_children;
                } di;
                struct dirty_leaf {

                        /*
                         * dr_data is set when we dirty the buffer
                         * so that we can retain the pointer even if it
                         * gets COW'd in a subsequent transaction group.
                         */
                        arc_buf_t *dr_data;
                        blkptr_t dr_overridden_by;
                        override_states_t dr_override_state;
                        uint8_t dr_copies;
                        boolean_t dr_nopwrite;
                } dl;
```

```
 140         } dt;
 141 } dbuf_dirty_record_t;
_____unchanged_portion_omitted_


 245 uint64_t dbuf_whichblock(struct dnode *di, uint64_t offset);

 247 dmu_buf_impl_t *dbuf_create_tlib(struct dnode *dn, char *data);
 248 void dbuf_create_bonus(struct dnode *dn);
 249 int dbuf_spill_set_blksz(dmu_buf_t *db, uint64_t blksz, dmu_tx_t *tx);
 250 void dbuf_spill_hold(struct dnode *dn, dmu_buf_impl_t **dbp, void *tag);

 252 void dbuf_rm_spill(struct dnode *dn, dmu_tx_t *tx);

 254 dmu_buf_impl_t *dbuf_hold(struct dnode *dn, uint64_t blkid, void *tag);
 255 dmu_buf_impl_t *dbuf_hold_level(struct dnode *dn, int level, uint64_t blkid,
 256     void *tag);
 257 int dbuf_hold_impl(struct dnode *dn, uint8_t level, uint64_t blkid, int create,
 258     void *tag, dmu_buf_impl_t **dbp);

 260 void dbuf_prefetch(struct dnode *dn, uint64_t blkid, zio_priority_t prio);
 257 void dbuf_prefetch(struct dnode *dn, uint64_t blkid);

 262 void dbuf_add_ref(dmu_buf_impl_t *db, void *tag);
 263 uint64_t dbuf_refcount(dmu_buf_impl_t *db);

 265 void dbuf_rele(dmu_buf_impl_t *db, void *tag);
 266 void dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag);

 268 dmu_buf_impl_t *dbuf_find(struct dnode *dn, uint8_t level, uint64_t blkid);

 270 int dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags);
 271 void dbuf_will_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
 272 void dbuf_fill_done(dmu_buf_impl_t *db, dmu_tx_t *tx);
 273 void dmu_buf_will_not_fill(dmu_buf_t *db, dmu_tx_t *tx);
 274 void dmu_buf_will_fill(dmu_buf_t *db, dmu_tx_t *tx);
 275 void dmu_buf_fill_done(dmu_buf_t *db, dmu_tx_t *tx);
 276 void dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx);
 277 dbuf_dirty_record_t *dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
 278 arc_buf_t *dbuf_loan_arcbuf(dmu_buf_impl_t *db);

 280 void dbuf_clear(dmu_buf_impl_t *db);
 281 void dbuf_evict(dmu_buf_impl_t *db);

 283 void dbuf_setdirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
 284 void dbuf_unoverride(dbuf_dirty_record_t *dr);
 285 void dbuf_sync_list(list_t *list, dmu_tx_t *tx);
 286 void dbuf_release_bp(dmu_buf_impl_t *db);

 288 void dbuf_free_range(struct dnode *dn, uint64_t start, uint64_t end,
 289     struct dmu_tx *);

 291 void dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx);

 293 #define DB_DNODE(_db)           ((_db)->db_dnode_handle->dnh_dnode)
 294 #define DB_DNODE_LOCK(_db)      ((_db)->db_dnode_handle->dnh_zrlock)
 295 #define DB_DNODE_ENTER(_db)     (zrl_add(&DB_DNODE_LOCK(_db)))
 296 #define DB_DNODE_EXIT(_db)      (zrl_remove(&DB_DNODE_LOCK(_db)))
 297 #define DB_DNODE_HELD(_db)      (!zrl_is_zero(&DB_DNODE_LOCK(_db)))
 298 #define DB_GET_SPA(_spa_p, _db) {               \
 299         dnode_t *__dn;                         \
 300         DB_DNODE_ENTER(_db);                   \
 301         __dn = DB_DNODE(_db);                  \
 302         *(_spa_p) = __dn->dn_objset->os_spa;   \
 303         DB_DNODE_EXIT(_db);                    \
 304 }
_____unchanged_portion_omitted_
```

****
28910 Thu Aug 22 16:15:21 2013
new/usr/src/uts/common/fs/zfs/sys/dmu.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****

_____***unchanged_portion_omitted_***

```
 220 typedef enum txg_how {
 221         TXG_WAIT = 1,
 222         TXG_NOWAIT,
 223         TXG_WAITED,
 224 } txg_how_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4298 Thu Aug 22 16:15:23 2013
new/usr/src/uts/common/fs/zfs/sys/dmu_tx.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_DMU_TX_H
#define _SYS_DMU_TX_H

#include <sys/inttypes.h>
#include <sys/dmu.h>
#include <sys/txg.h>
#include <sys/refcount.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dmu_buf_impl;
struct dmu_tx_hold;
struct dnode_link;
struct dsl_pool;
struct dnode;
struct dsl_dir;

struct dmu_tx {
        /*
         * No synchronization is needed because a tx can only be handled
         * by one thread.
         */
        list_t tx_holds; /* list of dmu_tx_hold_t */
        objset_t *tx_objset;
        struct dsl_dir *tx_dir;
        struct dsl_pool *tx_pool;
        uint64_t tx_txg;
```



```
        uint64_t tx_lastsnap_txg;
        uint64_t tx_lasttried_txg;
        txg_handle_t tx_txgh;
        void *tx_tempreserve_cookie;
        struct dmu_tx_hold *tx_needassign_txh;

        /* list of dmu_tx_callback_t on this dmu_tx */
        list_t tx_callbacks;

        /* placeholder for syncing context, doesn't need specific holds */
        boolean_t tx_anyobj;

        /* has this transaction already been delayed? */
        boolean_t tx_waited;

        /* time this transaction was created */
        hrtime_t tx_start;

        /* need to wait for sufficient dirty space */
        boolean_t tx_wait_dirty;

        list_t tx_callbacks; /* list of dmu_tx_callback_t on this dmu_tx */
        uint8_t tx_anyobj;
        int tx_err;
#ifdef ZFS_DEBUG
        uint64_t tx_space_towrite;
        uint64_t tx_space_tofree;
        uint64_t tx_space_tooverwrite;
        uint64_t tx_space_tounref;
        refcount_t tx_space_written;
        refcount_t tx_space_freed;
#endif
};
______unchanged_portion_omitted_
```

****
5499 Thu Aug 22 16:15:24 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dir.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****

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

#ifndef _SYS_DSL_DIR_H
#define _SYS_DSL_DIR_H

#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/refcount.h>
#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_dataset;

typedef enum dd_used {
        DD_USED_HEAD,
        DD_USED_SNAP,
        DD_USED_CHILD,
        DD_USED_CHILD_RSRV,
        DD_USED_REFRSRV,
        DD_USED_NUM
} dd_used_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5397 Thu Aug 22 16:15:25 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_pool.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

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

#ifndef _SYS_DSL_POOL_H
#define _SYS_DSL_POOL_H

#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/txg_impl.h>
#include <sys/zfs_context.h>
#include <sys/zio.h>
#include <sys/dnode.h>
#include <sys/ddt.h>
#include <sys/arc.h>
#include <sys/bpobj.h>
#include <sys/bptree.h>
#include <sys/rrwlock.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct objset;
struct dsl_dir;
struct dsl_dataset;
struct dsl_pool;
struct dmu_tx;
struct dsl_scan;

extern uint64_t zfs_dirty_data_max;
extern uint64_t zfs_dirty_data_max_max;
extern uint64_t zfs_dirty_data_sync;
extern int zfs_dirty_data_max_percent;
extern int zfs_delay_min_dirty_percent;
extern uint64_t zfs_delay_scale;
```



```
/* These macros are for indexing into the zfs_all_blkstats_t. */
#define DMU_OT_DEFERRED DMU_OT_NONE
#define DMU_OT_OTHER    DMU_OT_NUMTYPES /* place holder for DMU_OT() types */
#define DMU_OT_TOTAL    (DMU_OT_NUMTYPES + 1)

typedef struct zfs_blkstat {
        uint64_t        zb_count;
        uint64_t        zb_asize;
        uint64_t        zb_lsize;
        uint64_t        zb_psize;
        uint64_t        zb_gangs;
        uint64_t        zb_ditto_2_of_2_samevdev;
        uint64_t        zb_ditto_2_of_3_samevdev;
        uint64_t        zb_ditto_3_of_3_samevdev;
} zfs_blkstat_t;
______unchanged_portion_omitted_


typedef struct dsl_pool {
        /* Immutable */
        spa_t *dp_spa;
        struct objset *dp_meta_objset;
        struct dsl_dir *dp_root_dir;
        struct dsl_dir *dp_mos_dir;
        struct dsl_dir *dp_free_dir;
        struct dsl_dataset *dp_origin_snap;
        uint64_t dp_root_dir_obj;
        struct taskq *dp_vnrele_taskq;

        /* No lock needed - sync context only */
        blkptr_t dp_meta_rootbp;
        hrtime_t dp_read_overhead;
        uint64_t dp_throughput; /* bytes per millisec */
        uint64_t dp_write_limit;
        uint64_t dp_tmp_userrefs_obj;
        bpobj_t dp_free_bpobj;
        uint64_t dp_bptree_obj;
        uint64_t dp_empty_bpobj;

        struct dsl_scan *dp_scan;

        /* Uses dp_lock */
        kmutex_t dp_lock;
        kcondvar_t dp_spaceavail_cv;
        uint64_t dp_dirty_pertxg[TXG_SIZE];
        uint64_t dp_dirty_total;
        uint64_t dp_space_towrite[TXG_SIZE];
        uint64_t dp_tempreserved[TXG_SIZE];
        uint64_t dp_mos_used_delta;
        uint64_t dp_mos_compressed_delta;
        uint64_t dp_mos_uncompressed_delta;

        /*
         * Time of most recently scheduled (furthest in the future)
         * wakeup for delayed transactions.
         */
        hrtime_t dp_last_wakeup;

        /* Has its own locking */
        tx_state_t dp_tx;
        txg_list_t dp_dirty_datasets;
        txg_list_t dp_dirty_zilogs;
        txg_list_t dp_dirty_dirs;
        txg_list_t dp_sync_tasks;
```

```
122         /*
123          * Protects administrative changes (properties, namespace)
124          *
125          * It is only held for write in syncing context.  Therefore
126          * syncing context does not need to ever have it for read, since
127          * nobody else could possibly have it for write.
128          */
129         rrwlock_t dp_config_rwlock;

131         zfs_all_blkstats_t *dp_blkstats;
132 } dsl_pool_t;

134 int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
135 int dsl_pool_open(dsl_pool_t *dp);
136 void dsl_pool_close(dsl_pool_t *dp);
137 dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
138 void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
139 void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
140 int dsl_pool_sync_context(dsl_pool_t *dp);
141 uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
142 uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
143 void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
144 void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
132 int dsl_pool_tempreserve_space(dsl_pool_t *dp, uint64_t space, dmu_tx_t *tx);
133 void dsl_pool_tempreserve_clear(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
134 void dsl_pool_memory_pressure(dsl_pool_t *dp);
135 void dsl_pool_willuse_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
145 void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
146 void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
147     const blkptr_t *bpp);
148 void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
149 void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
150 void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
151 void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
152     int64_t used, int64_t comp, int64_t uncomp);
153 void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
154 void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
155 boolean_t dsl_pool_config_held(dsl_pool_t *dp);
156 boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

158 taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

160 int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
161     const char *tag, uint64_t now, dmu_tx_t *tx);
162 int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
163     const char *tag, dmu_tx_t *tx);
164 void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
165 int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
166 int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
167 void dsl_pool_rele(dsl_pool_t *dp, void *tag);

169 #ifdef  __cplusplus
170 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    8420 Thu Aug 22 16:15:26 2013
new/usr/src/uts/common/fs/zfs/sys/sa_impl.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_SA_IMPL_H
#define _SYS_SA_IMPL_H

#include <sys/dmu.h>
#include <sys/refcount.h>
#include <sys/list.h>

/*
 * Array of known attributes and their
 * various characteristics.
 */
typedef struct sa_attr_table {
        sa_attr_type_t  sa_attr;
        uint8_t sa_registered;
        uint16_t sa_length;
        sa_bswap_type_t sa_byteswap;
        char *sa_name;
} sa_attr_table_t;
______unchanged_portion_omitted_

/*
 * header for all bonus and spill buffers.
 *
 * The header has a fixed portion with a variable number
 * of "lengths" depending on the number of variable sized
 * attributes which are determined by the "layout number"
 * attribues which are determined by the "layout number"
 */

#define SA_MAGIC        0x2F505A  /* ZFS SA */
typedef struct sa_hdr_phys {
        uint32_t sa_magic;
```



```
        /* BEGIN CSTYLED */
        /*
         * Encoded with hdrsize and layout number as follows:
         * 16      10       0
         * +--------+-------+
         * | hdrsz  |layout |
         * +--------+-------+
         *
         * Bits 0-10 are the layout number
         * Bits 11-16 are the size of the header.
         * The hdrsize is the number * 8
         *
         * For example.
         * hdrsz of 1 ==> 8 byte header
         *          2 ==> 16 byte header
         *
         */
        /* END CSTYLED */
        uint16_t sa_layout_info;
        uint16_t sa_lengths[1]; /* optional sizes for variable length attrs */
        /* ... Data follows the lengths.  */
} sa_hdr_phys_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   11077 Thu Aug 22 16:15:27 2013
new/usr/src/uts/common/fs/zfs/sys/spa_impl.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _SYS_SPA_IMPL_H
#define _SYS_SPA_IMPL_H

#include <sys/spa.h>
#include <sys/vdev.h>
#include <sys/metaslab.h>
#include <sys/dmu.h>
#include <sys/dsl_pool.h>
#include <sys/uberblock_impl.h>
#include <sys/zfs_context.h>
#include <sys/avl.h>
#include <sys/refcount.h>
#include <sys/bplist.h>
#include <sys/bpobj.h>

#ifdef  __cplusplus
extern "C" {
#endif

typedef struct spa_error_entry {
        zbookmark_t     se_bookmark;
        char            *se_name;
        avl_node_t      se_avl;
} spa_error_entry_t;
```

*unchanged_portion_omitted_*

```
struct spa {
        /*
         * Fields protected by spa_namespace_lock.
         */
        char            spa_name[MAXNAMELEN];   /* pool name */
```



```
        char            *spa_comment;           /* comment */
        avl_node_t      spa_avl;                /* node in spa_namespace_avl */
        nvlist_t        *spa_config;            /* last synced config */
        nvlist_t        *spa_config_syncing;    /* currently syncing config */
        nvlist_t        *spa_config_splitting;  /* config for splitting */
        nvlist_t        *spa_load_info;         /* info and errors from load */
        uint64_t        spa_config_txg;         /* txg of last config change */
        int             spa_sync_pass;          /* iterate-to-convergence */
        pool_state_t    spa_state;              /* pool state */
        int             spa_inject_ref;         /* injection references */
        uint8_t         spa_sync_on;            /* sync threads are running */
        spa_load_state_t spa_load_state;        /* current load operation */
        uint64_t        spa_import_flags;       /* import specific flags */
        spa_taskqs_t    spa_zio_taskq[ZIO_TYPES][ZIO_TASKQ_TYPES];
        dsl_pool_t      *spa_dsl_pool;
        boolean_t       spa_is_initializing;    /* true while opening pool */
        metaslab_class_t *spa_normal_class;     /* normal data class */
        metaslab_class_t *spa_log_class;        /* intent log data class */
        uint64_t        spa_first_txg;          /* first txg after spa_open() */
        uint64_t        spa_final_txg;          /* txg of export/destroy */
        uint64_t        spa_freeze_txg;         /* freeze pool at this txg */
        uint64_t        spa_load_max_txg;       /* best initial ub_txg */
        uint64_t        spa_claim_max_txg;      /* highest claimed birth txg */
        timespec_t      spa_loaded_ts;          /* 1st successful open time */
        objset_t        *spa_meta_objset;       /* copy of dp->dp_meta_objset */
        txg_list_t      spa_vdev_txg_list;      /* per-txg dirty vdev list */
        vdev_t          *spa_root_vdev;         /* top-level vdev container */
        uint64_t        spa_config_guid;        /* config pool guid */
        uint64_t        spa_load_guid;          /* spa_load initialized guid */
        uint64_t        spa_last_synced_guid;   /* last synced guid */
        list_t          spa_config_dirty_list;  /* vdevs with dirty config */
        list_t          spa_state_dirty_list;   /* vdevs with dirty state */
        spa_aux_vdev_t  spa_spares;             /* hot spares */
        spa_aux_vdev_t  spa_l2cache;            /* L2ARC cache devices */
        nvlist_t        *spa_label_features;    /* Features for reading MOS */
        uint64_t        spa_config_object;      /* MOS object for pool config */
        uint64_t        spa_config_generation;  /* config generation number */
        uint64_t        spa_syncing_txg;        /* txg currently syncing */
        bpobj_t         spa_deferred_bpobj;     /* deferred-free bplist */
        bplist_t        spa_free_bplist[TXG_SIZE]; /* bplist of stuff to free */
        uberblock_t     spa_ubsync;             /* last synced uberblock */
        uberblock_t     spa_uberblock;          /* current uberblock */
        boolean_t       spa_extreme_rewind;     /* rewind past deferred frees */
        uint64_t        spa_last_io;            /* lbolt of last non-scan I/O */
        kmutex_t        spa_scrub_lock;         /* resilver/scrub lock */
        uint64_t        spa_scrub_inflight;     /* in-flight scrub I/Os */
        kcondvar_t      spa_scrub_io_cv;        /* scrub I/O completion */
        uint8_t         spa_scrub_active;       /* active or suspended? */
        uint8_t         spa_scrub_type;         /* type of scrub we're doing */
        uint8_t         spa_scrub_finished;     /* indicator to rotate logs */
        uint8_t         spa_scrub_started;      /* started since last boot */
        uint8_t         spa_scrub_reopen;       /* scrub doing vdev_reopen */
        uint64_t        spa_scan_pass_start;    /* start time per pass/reboot */
        uint64_t        spa_scan_pass_exam;     /* examined bytes per pass */
        kmutex_t        spa_async_lock;         /* protect async state */
        kthread_t       *spa_async_thread;      /* thread doing async task */
        int             spa_async_suspended;    /* async tasks suspended */
        kcondvar_t      spa_async_cv;           /* wait for thread_exit() */
        uint16_t        spa_async_tasks;        /* async task mask */
        char            *spa_root;              /* alternate root directory */
        uint64_t        spa_ena;                /* spa-wide ereport ENA */
        int             spa_last_open_failed;   /* error if last open failed */
        uint64_t        spa_last_ubsync_txg;    /* "best" uberblock txg */
        uint64_t        spa_last_ubsync_txg_ts; /* timestamp from that ub */
        uint64_t        spa_load_txg;           /* ub txg that loaded */
        uint64_t        spa_load_txg_ts;        /* timestamp from that ub */
```

```
186         uint64_t        spa_load_meta_errors;   /* verify metadata err count */
187         uint64_t        spa_load_data_errors;   /* verify data err count */
188         uint64_t        spa_verify_min_txg;     /* start txg of verify scrub */
189         kmutex_t        spa_errlog_lock;        /* error log lock */
190         uint64_t        spa_errlog_last;        /* last error log object */
191         uint64_t        spa_errlog_scrub;       /* scrub error log object */
192         kmutex_t        spa_errlist_lock;       /* error list/ereport lock */
193         avl_tree_t      spa_errlist_last;       /* last error list */
194         avl_tree_t      spa_errlist_scrub;      /* scrub error list */
195         uint64_t        spa_deflate;            /* should we deflate? */
196         uint64_t        spa_history;            /* history object */
197         kmutex_t        spa_history_lock;       /* history lock */
198         vdev_t          *spa_pending_vdev;      /* pending vdev additions */
199         kmutex_t        spa_props_lock;         /* property lock */
200         uint64_t        spa_pool_props_object;  /* object for properties */
201         uint64_t        spa_bootfs;             /* default boot filesystem */
202         uint64_t        spa_failmode;           /* failure mode for the pool */
203         uint64_t        spa_delegation;         /* delegation on/off */
204         list_t          spa_config_list;        /* previous cache file(s) */
205         zio_t           *spa_async_zio_root;    /* root of all async I/O */
206         zio_t           *spa_suspend_zio_root;  /* root of all suspended I/O */
207         kmutex_t        spa_suspend_lock;       /* protects suspend_zio_root */
208         kcondvar_t      spa_suspend_cv;         /* notification of resume */
209         uint8_t         spa_suspended;          /* pool is suspended */
210         uint8_t         spa_claiming;           /* pool is doing zil_claim() */
211         boolean_t       spa_debug;              /* debug enabled? */
212         boolean_t       spa_is_root;            /* pool is root */
213         int             spa_minref;             /* num refs when first opened */
214         int             spa_mode;               /* FREAD | FWRITE */
215         spa_log_state_t spa_log_state;          /* log state */
216         uint64_t        spa_autoexpand;         /* lun expansion on/off */
217         ddt_t           *spa_ddt[ZIO_CHECKSUM_FUNCTIONS]; /* in-core DDTs */
218         uint64_t        spa_ddt_stat_object;    /* DDT statistics */
219         uint64_t        spa_dedup_ditto;        /* dedup ditto threshold */
220         uint64_t        spa_dedup_checksum;     /* default dedup checksum */
221         uint64_t        spa_dspace;             /* dspace in normal class */
222         kmutex_t        spa_vdev_top_lock;      /* dueling offline/remove */
223         kmutex_t        spa_proc_lock;          /* protects spa_proc* */
224         kcondvar_t      spa_proc_cv;            /* spa_proc_state transitions */
225         spa_proc_state_t spa_proc_state;        /* see definition */
226         struct proc     *spa_proc;              /* "zpool-poolname" process */
227         uint64_t        spa_did;                /* if procp != p0, did of t1 */
228         boolean_t       spa_autoreplace;        /* autoreplace set in open */
229         int             spa_vdev_locks;         /* locks grabbed */
230         uint64_t        spa_creation_version;   /* version at pool creation */
231         uint64_t        spa_prev_software_version; /* See ub_software_version */
232         uint64_t        spa_feat_for_write_obj; /* required to write to pool */
233         uint64_t        spa_feat_for_read_obj;  /* required to read from pool */
234         uint64_t        spa_feat_desc_obj;      /* Feature descriptions */
235         cyclic_id_t     spa_deadman_cycid;      /* cyclic id */
236         uint64_t        spa_deadman_calls;      /* number of deadman calls */
237         hrtime_t        spa_sync_starttime;     /* starting time fo spa_sync */
237         uint64_t        spa_sync_starttime;     /* starting time fo spa_sync */
238         uint64_t        spa_deadman_synctime;   /* deadman expiration timer */

240         /*
241          * spa_iokstat_lock protects spa_iokstat and
242          * spa_queue_stats[].
243          */
244         kmutex_t        spa_iokstat_lock;
239         uint64_t        spa_iokstat_lock;       /* protects spa_iokstat_* */
245         struct kstat    *spa_iokstat;           /* kstat of io to this pool */
246         struct {
247                 int spa_active;
248                 int spa_queued;
249         } spa_queue_stats[ZIO_PRIORITY_NUM_QUEUEABLE];
```



```
251         hrtime_t        spa_ccw_fail_time;      /* Conf cache write fail time */

253         /*
254          * spa_refcount & spa_config_lock must be the last elements
255          * because refcount_t changes size based on compilation options.
256          * In order for the MDB module to function correctly, the other
257          * fields must remain in the same location.
258          */
259         spa_config_lock_t spa_config_lock[SCL_LOCKS]; /* config changes */
260         refcount_t      spa_refcount;           /* number of opens */
261 };
_____unchanged_portion_omitted_
```

```
**********************************************************
    4086 Thu Aug 22 16:15:28 2013
new/usr/src/uts/common/fs/zfs/sys/txg.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_TXG_H
#define _SYS_TXG_H

#include <sys/spa.h>
#include <sys/zfs_context.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define TXG_CONCURRENT_STATES   3       /* open, quiescing, syncing     */
#define TXG_SIZE                4               /* next power of 2      */
#define TXG_MASK                (TXG_SIZE - 1)  /* mask for size        */
#define TXG_INITIAL             TXG_SIZE        /* initial txg          */
#define TXG_IDX                 (txg & TXG_MASK)

/* Number of txgs worth of frees we defer adding to in-core spacemaps */
#define TXG_DEFER_SIZE          2

typedef struct tx_cpu tx_cpu_t;

typedef struct txg_handle {
        tx_cpu_t        *th_cpu;
        uint64_t        th_txg;
} txg_handle_t;
_____unchanged_portion_omitted_

struct dsl_pool;
```



```
extern void txg_init(struct dsl_pool *dp, uint64_t txg);
extern void txg_fini(struct dsl_pool *dp);
extern void txg_sync_start(struct dsl_pool *dp);
extern void txg_sync_stop(struct dsl_pool *dp);
extern uint64_t txg_hold_open(struct dsl_pool *dp, txg_handle_t *txghp);
extern void txg_rele_to_quiesce(txg_handle_t *txghp);
extern void txg_rele_to_sync(txg_handle_t *txghp);
extern void txg_register_callbacks(txg_handle_t *txghp, list_t *tx_callbacks);

extern void txg_delay(struct dsl_pool *dp, uint64_t txg, hrtime_t delta,
    hrtime_t resolution);
extern void txg_kick(struct dsl_pool *dp);

/*
 * Wait until the given transaction group has finished syncing.
 * Try to make this happen as soon as possible (eg. kick off any
 * necessary syncs immediately).  If txg==0, wait for the currently open
 * txg to finish syncing.
 */
extern void txg_wait_synced(struct dsl_pool *dp, uint64_t txg);

/*
 * Wait until the given transaction group, or one after it, is
 * the open transaction group.  Try to make this happen as soon
 * as possible (eg. kick off any necessary syncs immediately).
 * If txg == 0, wait for the next open txg.
 */
extern void txg_wait_open(struct dsl_pool *dp, uint64_t txg);

/*
 * Returns TRUE if we are "backed up" waiting for the syncing
 * transaction to complete; otherwise returns FALSE.
 */
extern boolean_t txg_stalled(struct dsl_pool *dp);

/* returns TRUE if someone is waiting for the next txg to sync */
extern boolean_t txg_sync_waiting(struct dsl_pool *dp);

/*
 * Per-txg object lists.
 */

#define TXG_CLEAN(txg)  ((txg) - 1)

extern void txg_list_create(txg_list_t *tl, size_t offset);
extern void txg_list_destroy(txg_list_t *tl);
extern boolean_t txg_list_empty(txg_list_t *tl, uint64_t txg);
extern boolean_t txg_list_add(txg_list_t *tl, void *p, uint64_t txg);
extern boolean_t txg_list_add_tail(txg_list_t *tl, void *p, uint64_t txg);
extern void *txg_list_remove(txg_list_t *tl, uint64_t txg);
extern void *txg_list_remove_this(txg_list_t *tl, void *p, uint64_t txg);
extern boolean_t txg_list_member(txg_list_t *tl, void *p, uint64_t txg);
extern void *txg_list_head(txg_list_t *tl, uint64_t txg);
extern void *txg_list_next(txg_list_t *tl, void *p, uint64_t txg);

#ifdef  __cplusplus
}
#endif
_____unchanged_portion_omitted_
```

****
4819 Thu Aug 22 16:15:29 2013
new/usr/src/uts/common/fs/zfs/sys/txg_impl.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

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

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#ifndef _SYS_TXG_IMPL_H
#define _SYS_TXG_IMPL_H

#include <sys/spa.h>
#include <sys/txg.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * The tx_cpu structure is a per-cpu structure that is used to track
 * the number of active transaction holds (tc_count). As transactions
 * are assigned into a transaction group the appropriate tc_count is
 * incremented to indicate that there are pending changes that have yet
 * to quiesce. Consumers evenutally call txg_rele_to_sync() to decrement
 * the tc_count. A transaction group is not considered quiesced until all
 * tx_cpu structures have reached a tc_count of zero.
 *
 * This structure is a per-cpu structure by design. Updates to this structure
 * are frequent and concurrent. Having a single structure would result in
 * heavy lock contention so a per-cpu design was implemented. With the fanned
 * out mutex design, consumers only need to lock the mutex associated with
 * thread's cpu.
 *
 * The tx_cpu contains two locks, the tc_lock and tc_open_lock.
 * The tc_lock is used to protect all members of the tx_cpu structure with
 * the exception of the tc_open_lock. This lock should only be held for a
```



```
 * short period of time, typically when updating the value of tc_count.
 *
 * The tc_open_lock protects the tx_open_txg member of the tx_state structure.
 * This lock is used to ensure that transactions are only assigned into
 * the current open transaction group. In order to move the current open
 * transaction group to the quiesce phase, the txg_quiesce thread must
 * grab all tc_open_locks, increment the tx_open_txg, and drop the locks.
 * The tc_open_lock is held until the transaction is assigned into the
 * transaction group. Typically, this is a short operation but if throttling
 * is occuring it may be held for longer periods of time.
 */
struct tx_cpu {
        kmutex_t        tc_open_lock;   /* protects tx_open_txg */
        kmutex_t        tc_lock;        /* protects the rest of this struct */
        kcondvar_t      tc_cv[TXG_SIZE];
        uint64_t        tc_count[TXG_SIZE];     /* tx hold count on each txg */
        list_t          tc_callbacks[TXG_SIZE]; /* commit cb list */
        char            tc_pad[8];              /* pad to fill 3 cache lines */
};

/*
 * The tx_state structure maintains the state information about the different
 * stages of the pool's transcation groups. A per pool tx_state structure
 * is used to track this information. The tx_state structure also points to
 * an array of tx_cpu structures (described above). Although the tx_sync_lock
 * is used to protect the members of this structure, it is not used to
 * protect the tx_open_txg. Instead a special lock in the tx_cpu structure
 * is used. Readers of tx_open_txg must grab the per-cpu tc_open_lock.
 * Any thread wishing to update tx_open_txg must grab the tc_open_lock on
 * every cpu (see txg_quiesce()).
 */
typedef struct tx_state {
        tx_cpu_t        *tx_cpu;        /* protects access to tx_open_txg */
        kmutex_t        tx_sync_lock;   /* protects the rest of this struct */

        uint64_t        tx_open_txg;    /* currently open txg id */
        uint64_t        tx_quiesced_txg; /* quiesced txg waiting for sync */
        uint64_t        tx_syncing_txg; /* currently syncing txg id */
        uint64_t        tx_synced_txg;  /* last synced txg id */

        hrtime_t        tx_open_time;   /* start time of tx_open_txg */

        uint64_t        tx_sync_txg_waiting; /* txg we're waiting to sync */
        uint64_t        tx_quiesce_txg_waiting; /* txg we're waiting to open */

        kcondvar_t      tx_sync_more_cv;
        kcondvar_t      tx_sync_done_cv;
        kcondvar_t      tx_quiesce_more_cv;
        kcondvar_t      tx_quiesce_done_cv;
        kcondvar_t      tx_timeout_cv;
        kcondvar_t      tx_exit_cv;     /* wait for all threads to exit */

        uint8_t         tx_threads;     /* number of threads */
        uint8_t         tx_exiting;     /* set when we're exiting */

        kthread_t       *tx_sync_thread;
        kthread_t       *tx_quiesce_thread;

        taskq_t         *tx_commit_cb_taskq; /* commit callback taskq */
} tx_state_t;
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   11711 Thu Aug 22 16:15:31 2013
new/usr/src/uts/common/fs/zfs/sys/vdev_impl.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

typedef struct vdev_queue_class {
        uint32_t        vqc_active;

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

struct vdev_queue {
        vdev_t          *vq_vdev;
        vdev_queue_class_t vq_class[ZIO_PRIORITY_NUM_QUEUEABLE];
        avl_tree_t      vq_active_tree;
        uint64_t        vq_last_offset;
        hrtime_t        vq_io_complete_ts; /* time last i/o completed */
        avl_tree_t      vq_deadline_tree;
        avl_tree_t      vq_read_tree;
        avl_tree_t      vq_write_tree;
        avl_tree_t      vq_pending_tree;
        hrtime_t        vq_io_complete_ts;
        kmutex_t        vq_lock;
};
______unchanged_portion_omitted_
```

****
2135 Thu Aug 22 16:15:32 2013
new/usr/src/uts/common/fs/zfs/sys/zfs_context.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#ifndef _SYS_ZFS_CONTEXT_H
#define _SYS_ZFS_CONTEXT_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/note.h>
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/atomic.h>
#include <sys/sysmacros.h>
#include <sys/bitmap.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/taskq.h>
#include <sys/taskq_impl.h>
#include <sys/buf.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cpuvar.h>
#include <sys/kobj.h>
#include <sys/conf.h>
#include <sys/disp.h>
#include <sys/debug.h>
#include <sys/random.h>
#include <sys/byteorder.h>
```



```
#include <sys/systm.h>
#include <sys/list.h>
#include <sys/uio.h>
#include <sys/dirent.h>
#include <sys/time.h>
#include <vm/seg_kmem.h>
#include <sys/zone.h>
#include <sys/uio.h>
#include <sys/zfs_debug.h>
#include <sys/sysevent.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dev.h>
#include <sys/fm/util.h>
#include <sys/sunddi.h>
#include <sys/cyclic.h>
#include <sys/disp.h>
#include <sys/callo.h>

#define CPU_SEQID       (CPU->cpu_seqid)

#ifdef  __cplusplus
}
```

*unchanged_portion_omitted_*

****
17668 Thu Aug 22 16:15:33 2013
new/usr/src/uts/common/fs/zfs/sys/zio.h
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

#ifndef _ZIO_H
#define _ZIO_H

#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/fs/zfs.h>
#include <sys/zio_impl.h>

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Embedded checksum
 */
#define ZEC_MAGIC       0x210da7ab10c7a11ULL

typedef struct zio_eck {
        uint64_t        zec_magic;      /* for validation, endianness   */
        zio_cksum_t     zec_cksum;      /* 256-bit checksum             */
} zio_eck_t;
```

*unchanged_portion_omitted_*

```
/* N.B. when altering this value, also change BOOTFS_COMPRESS_VALID below */
#define ZIO_COMPRESS_ON_VALUE   ZIO_COMPRESS_LZJB
#define ZIO_COMPRESS_DEFAULT    ZIO_COMPRESS_OFF
```



```
#define BOOTFS_COMPRESS_VALID(compress)                 \
        ((compress) == ZIO_COMPRESS_LZJB ||             \
        (compress) == ZIO_COMPRESS_LZ4 ||               \
        ((compress) == ZIO_COMPRESS_ON &&               \
        ZIO_COMPRESS_ON_VALUE == ZIO_COMPRESS_LZJB) ||  \
        (compress) == ZIO_COMPRESS_OFF)

#define ZIO_FAILURE_MODE_WAIT           0
#define ZIO_FAILURE_MODE_CONTINUE       1
#define ZIO_FAILURE_MODE_PANIC          2

typedef enum zio_priority {
        ZIO_PRIORITY_SYNC_READ,
        ZIO_PRIORITY_SYNC_WRITE,        /* ZIL */
        ZIO_PRIORITY_ASYNC_READ,        /* prefetch */
        ZIO_PRIORITY_ASYNC_WRITE,       /* spa_sync() */
        ZIO_PRIORITY_SCRUB,             /* asynchronous scrub/resilver reads */
        ZIO_PRIORITY_NUM_QUEUEABLE,
#define ZIO_PRIORITY_NOW                (zio_priority_table[0])
#define ZIO_PRIORITY_SYNC_READ          (zio_priority_table[1])
#define ZIO_PRIORITY_SYNC_WRITE         (zio_priority_table[2])
#define ZIO_PRIORITY_LOG_WRITE          (zio_priority_table[3])
#define ZIO_PRIORITY_CACHE_FILL         (zio_priority_table[4])
#define ZIO_PRIORITY_AGG                (zio_priority_table[5])
#define ZIO_PRIORITY_FREE               (zio_priority_table[6])
#define ZIO_PRIORITY_ASYNC_WRITE        (zio_priority_table[7])
#define ZIO_PRIORITY_ASYNC_READ         (zio_priority_table[8])
#define ZIO_PRIORITY_RESILVER           (zio_priority_table[9])
#define ZIO_PRIORITY_SCRUB              (zio_priority_table[10])
#define ZIO_PRIORITY_DDT_PREFETCH       (zio_priority_table[11])
#define ZIO_PRIORITY_TABLE_SIZE         12

        ZIO_PRIORITY_NOW                /* non-queued i/os (e.g. free) */
} zio_priority_t;

#define ZIO_PIPELINE_CONTINUE           0x100
#define ZIO_PIPELINE_STOP               0x101

enum zio_flag {
        /*
         * Flags inherited by gang, ddt, and vdev children,
         * and that must be equal for two zios to aggregate
         */
        ZIO_FLAG_DONT_AGGREGATE = 1 << 0,
        ZIO_FLAG_IO_REPAIR      = 1 << 1,
        ZIO_FLAG_SELF_HEAL      = 1 << 2,
        ZIO_FLAG_RESILVER       = 1 << 3,
        ZIO_FLAG_SCRUB          = 1 << 4,
        ZIO_FLAG_SCAN_THREAD    = 1 << 5,

#define ZIO_FLAG_AGG_INHERIT    (ZIO_FLAG_CANFAIL - 1)

        /*
         * Flags inherited by ddt, gang, and vdev children.
         */
        ZIO_FLAG_CANFAIL        = 1 << 6,       /* must be first for INHERIT */
        ZIO_FLAG_SPECULATIVE    = 1 << 7,
        ZIO_FLAG_CONFIG_WRITER  = 1 << 8,
        ZIO_FLAG_DONT_RETRY     = 1 << 9,
        ZIO_FLAG_DONT_CACHE     = 1 << 10,
        ZIO_FLAG_NODATA         = 1 << 11,
        ZIO_FLAG_INDUCE_DAMAGE  = 1 << 12,

#define ZIO_FLAG_DDT_INHERIT    (ZIO_FLAG_IO_RETRY - 1)
#define ZIO_FLAG_GANG_INHERIT   (ZIO_FLAG_IO_RETRY - 1)
```

```
171         /*
172          * Flags inherited by vdev children.
173          */
174         ZIO_FLAG_IO_RETRY       = 1 << 13,      /* must be first for INHERIT */
175         ZIO_FLAG_PROBE          = 1 << 14,
176         ZIO_FLAG_TRYHARD        = 1 << 15,
177         ZIO_FLAG_OPTIONAL       = 1 << 16,

179 #define ZIO_FLAG_VDEV_INHERIT   (ZIO_FLAG_DONT_QUEUE - 1)

181         /*
182          * Flags not inherited by any children.
183          */
184         ZIO_FLAG_DONT_QUEUE     = 1 << 17,      /* must be first for INHERIT */
185         ZIO_FLAG_DONT_PROPAGATE = 1 << 18,
186         ZIO_FLAG_IO_BYPASS      = 1 << 19,
187         ZIO_FLAG_IO_REWRITE     = 1 << 20,
188         ZIO_FLAG_RAW            = 1 << 21,
189         ZIO_FLAG_GANG_CHILD     = 1 << 22,
190         ZIO_FLAG_DDT_CHILD      = 1 << 23,
191         ZIO_FLAG_GODFATHER      = 1 << 24,
192         ZIO_FLAG_NOPWRITE       = 1 << 25,
193         ZIO_FLAG_REEXECUTED     = 1 << 26,
194         ZIO_FLAG_DELEGATED      = 1 << 27,
195 };
_____unchanged_portion_omitted_

225 /*
226  * We'll take the unused errnos, 'EBADE' and 'EBADR' (from the Convergent
227  * graveyard) to indicate checksum errors and fragmentation.
228  */
229 #define ECKSUM  EBADE
230 #define EFRAGS  EBADR

232 typedef void zio_done_func_t(zio_t *zio);

234 extern const char *zio_type_name[ZIO_TYPES];
236 extern uint8_t zio_priority_table[ZIO_PRIORITY_TABLE_SIZE];
237 extern char *zio_type_name[ZIO_TYPES];

236 /*
237  * A bookmark is a four-tuple <objset, object, level, blkid> that uniquely
238  * identifies any block in the pool.  By convention, the meta-objset (MOS)
239  * is objset 0, and the meta-dnode is object 0.  This covers all blocks
240  * except root blocks and ZIL blocks, which are defined as follows:
241  *
242  * Root blocks (objset_phys_t) are object 0, level -1:  <objset, 0, -1, 0>.
243  * ZIL blocks are bookmarked <objset, 0, -2, blkid == ZIL sequence number>.
244  * dmu_sync()ed ZIL data blocks are bookmarked <objset, object, -2, blkid>.
245  *
246  * Note: this structure is called a bookmark because its original purpose
247  * was to remember where to resume a pool-wide traverse.
248  *
249  * Note: this structure is passed between userland and the kernel.
250  * Therefore it must not change size or alignment between 32/64 bit
251  * compilation options.
252  */
253 typedef struct zbookmark {
254         uint64_t        zb_objset;
255         uint64_t        zb_object;
256         int64_t         zb_level;
257         uint64_t        zb_blkid;
258 } zbookmark_t;
_____unchanged_portion_omitted_

367 struct zio {
```



```
368         /* Core information about this I/O */
369         zbookmark_t     io_bookmark;
370         zio_prop_t      io_prop;
371         zio_type_t      io_type;
372         enum zio_child  io_child_type;
373         int             io_cmd;
374         zio_priority_t  io_priority;
377         uint8_t         io_priority;
375         uint8_t         io_reexecute;
376         uint8_t         io_state[ZIO_WAIT_TYPES];
377         uint64_t        io_txg;
378         spa_t           *io_spa;
379         blkptr_t        *io_bp;
380         blkptr_t        *io_bp_override;
381         blkptr_t        io_bp_copy;
382         list_t          io_parent_list;
383         list_t          io_child_list;
384         zio_link_t      *io_walk_link;
385         zio_t           *io_logical;
386         zio_transform_t *io_transform_stack;

388         /* Callback info */
389         zio_done_func_t *io_ready;
390         zio_done_func_t *io_physdone;
391         zio_done_func_t *io_done;
392         void            *io_private;
393         int64_t         io_prev_space_delta;    /* DMU private */
394         blkptr_t        io_bp_orig;

396         /* Data represented by this I/O */
397         void            *io_data;
398         void            *io_orig_data;
399         uint64_t        io_size;
400         uint64_t        io_orig_size;

402         /* Stuff for the vdev stack */
403         vdev_t          *io_vd;
404         void            *io_vsd;
405         const zio_vsd_ops_t *io_vsd_ops;

407         uint64_t        io_offset;
410         uint64_t        io_deadline;
408         hrtime_t        io_timestamp;
409         avl_node_t      io_queue_node;
412         avl_node_t      io_offset_node;
413         avl_node_t      io_deadline_node;
414         avl_tree_t      *io_vdev_tree;

411         /* Internal pipeline state */
412         enum zio_flag   io_flags;
413         enum zio_stage  io_stage;
414         enum zio_stage  io_pipeline;
415         enum zio_flag   io_orig_flags;
416         enum zio_stage  io_orig_stage;
417         enum zio_stage  io_orig_pipeline;
418         int             io_error;
419         int             io_child_error[ZIO_CHILD_TYPES];
420         uint64_t        io_children[ZIO_CHILD_TYPES][ZIO_WAIT_TYPES];
421         uint64_t        io_child_count;
422         uint64_t        io_phys_children;
423         uint64_t        io_parent_count;
424         uint64_t        *io_stall;
425         zio_t           *io_gang_leader;
426         zio_gang_node_t *io_gang_tree;
427         void            *io_executor;
428         void            *io_waiter;
```

```
        kmutex_t        io_lock;
        kcondvar_t      io_cv;

        /* FMA state */
        zio_cksum_report_t *io_cksum_report;
        uint64_t        io_ena;

        /* Taskq dispatching state */
        taskq_ent_t     io_tqent;
};

extern zio_t *zio_null(zio_t *pio, spa_t *spa, vdev_t *vd,
    zio_done_func_t *done, void *private, enum zio_flag flags);

extern zio_t *zio_root(spa_t *spa,
    zio_done_func_t *done, void *private, enum zio_flag flags);

extern zio_t *zio_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, void *data,
    uint64_t size, zio_done_func_t *done, void *private,
    zio_priority_t priority, enum zio_flag flags, const zbookmark_t *zb);
    int priority, enum zio_flag flags, const zbookmark_t *zb);

extern zio_t *zio_write(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
    void *data, uint64_t size, const zio_prop_t *zp,
    zio_done_func_t *ready, zio_done_func_t *physdone, zio_done_func_t *done,
    void *private,
    zio_priority_t priority, enum zio_flag flags, const zbookmark_t *zb);
    zio_done_func_t *ready, zio_done_func_t *done, void *private,
    int priority, enum zio_flag flags, const zbookmark_t *zb);

extern zio_t *zio_rewrite(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
    void *data, uint64_t size, zio_done_func_t *done, void *private,
    zio_priority_t priority, enum zio_flag flags, zbookmark_t *zb);
    int priority, enum zio_flag flags, zbookmark_t *zb);

extern void zio_write_override(zio_t *zio, blkptr_t *bp, int copies,
    boolean_t nopwrite);

extern void zio_free(spa_t *spa, uint64_t txg, const blkptr_t *bp);

extern zio_t *zio_claim(zio_t *pio, spa_t *spa, uint64_t txg,
    const blkptr_t *bp,
    zio_done_func_t *done, void *private, enum zio_flag flags);

extern zio_t *zio_ioctl(zio_t *pio, spa_t *spa, vdev_t *vd, int cmd,
    zio_done_func_t *done, void *private, enum zio_flag flags);
    zio_done_func_t *done, void *private, int priority, enum zio_flag flags);

extern zio_t *zio_read_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
    uint64_t size, void *data, int checksum,
    zio_done_func_t *done, void *private, zio_priority_t priority,
    enum zio_flag flags, boolean_t labels);
    zio_done_func_t *done, void *private, int priority, enum zio_flag flags,
    boolean_t labels);

extern zio_t *zio_write_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
    uint64_t size, void *data, int checksum,
    zio_done_func_t *done, void *private, zio_priority_t priority,
    enum zio_flag flags, boolean_t labels);
    zio_done_func_t *done, void *private, int priority, enum zio_flag flags,
    boolean_t labels);

extern zio_t *zio_free_sync(zio_t *pio, spa_t *spa, uint64_t txg,
    const blkptr_t *bp, enum zio_flag flags);

extern int zio_alloc_zil(spa_t *spa, uint64_t txg, blkptr_t *new_bp,
```



```
    blkptr_t *old_bp, uint64_t size, boolean_t use_slog);
extern void zio_free_zil(spa_t *spa, uint64_t txg, blkptr_t *bp);
extern void zio_flush(zio_t *zio, vdev_t *vd);
extern void zio_shrink(zio_t *zio, uint64_t size);

extern int zio_wait(zio_t *zio);
extern void zio_nowait(zio_t *zio);
extern void zio_execute(zio_t *zio);
extern void zio_interrupt(zio_t *zio);

extern zio_t *zio_walk_parents(zio_t *cio);
extern zio_t *zio_walk_children(zio_t *pio);
extern zio_t *zio_unique_parent(zio_t *cio);
extern void zio_add_child(zio_t *pio, zio_t *cio);

extern void *zio_buf_alloc(size_t size);
extern void zio_buf_free(void *buf, size_t size);
extern void *zio_data_buf_alloc(size_t size);
extern void zio_data_buf_free(void *buf, size_t size);

extern void zio_resubmit_stage_async(void *);

extern zio_t *zio_vdev_child_io(zio_t *zio, blkptr_t *bp, vdev_t *vd,
    uint64_t offset, void *data, uint64_t size, int type,
    zio_priority_t priority, enum zio_flag flags,
    zio_done_func_t *done, void *private);
    uint64_t offset, void *data, uint64_t size, int type, int priority,
    enum zio_flag flags, zio_done_func_t *done, void *private);

extern zio_t *zio_vdev_delegated_io(vdev_t *vd, uint64_t offset,
    void *data, uint64_t size, int type, zio_priority_t priority,
    void *data, uint64_t size, int type, int priority,
    enum zio_flag flags, zio_done_func_t *done, void *private);

extern void zio_vdev_io_bypass(zio_t *zio);
extern void zio_vdev_io_reissue(zio_t *zio);
extern void zio_vdev_io_redone(zio_t *zio);

extern void zio_checksum_verified(zio_t *zio);
extern int zio_worst_error(int e1, int e2);

extern enum zio_checksum zio_checksum_select(enum zio_checksum child,
    enum zio_checksum parent);
extern enum zio_checksum zio_checksum_dedup_select(spa_t *spa,
    enum zio_checksum child, enum zio_checksum parent);
extern enum zio_compress zio_compress_select(enum zio_compress child,
    enum zio_compress parent);

extern void zio_suspend(spa_t *spa, zio_t *zio);
extern int zio_resume(spa_t *spa);
extern void zio_resume_wait(spa_t *spa);

/*
 * Initial setup and teardown.
 */
extern void zio_init(void);
extern void zio_fini(void);

/*
 * Fault injection
 */
struct zinject_record;
extern uint32_t zio_injection_enabled;
extern int zio_inject_fault(char *name, int flags, int *id,
    struct zinject_record *record);
extern int zio_inject_list_next(int *id, char *name, size_t buflen,
```

```
549     struct zinject_record *record);
550 extern int zio_clear_fault(int id);
551 extern void zio_handle_panic_injection(spa_t *spa, char *tag, uint64_t type);
552 extern int zio_handle_fault_injection(zio_t *zio, int error);
553 extern int zio_handle_device_injection(vdev_t *vd, zio_t *zio, int error);
554 extern int zio_handle_label_injection(zio_t *zio, int error);
555 extern void zio_handle_ignored_writes(zio_t *zio);
556 extern uint64_t zio_handle_io_delay(zio_t *zio);

558 /*
559  * Checksum ereport functions
560  */
561 extern void zfs_ereport_start_checksum(spa_t *spa, vdev_t *vd, struct zio *zio,
562     uint64_t offset, uint64_t length, void *arg, struct zio_bad_cksum *info);
563 extern void zfs_ereport_finish_checksum(zio_cksum_report_t *report,
564     const void *good_data, const void *bad_data, boolean_t drop_if_identical);

566 extern void zfs_ereport_send_interim_checksum(zio_cksum_report_t *report);
567 extern void zfs_ereport_free_checksum(zio_cksum_report_t *report);

569 /* If we have the good data in hand, this function can be used */
570 extern void zfs_ereport_post_checksum(spa_t *spa, vdev_t *vd,
571     struct zio *zio, uint64_t offset, uint64_t length,
572     const void *good_data, const void *bad_data, struct zio_bad_cksum *info);

574 /* Called from spa_sync(), but primarily an injection handler */
575 extern void spa_handle_ignored_writes(spa_t *spa);

577 /* zbookmark functions */
578 boolean_t zbookmark_is_before(const struct dnode_phys *dnp,
579     const zbookmark_t *zb1, const zbookmark_t *zb2);

581 #ifdef  __cplusplus
582 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   22802 Thu Aug 22 16:15:34 2013
new/usr/src/uts/common/fs/zfs/txg.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Portions Copyright 2011 Martin Matuska
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/txg_impl.h>
#include <sys/dmu_impl.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_scan.h>
#include <sys/callb.h>

/*
 * ZFS Transaction Groups
 * ----------------------
 *
 * ZFS transaction groups are, as the name implies, groups of transactions
 * that act on persistent state. ZFS asserts consistency at the granularity of
 * these transaction groups. Each successive transaction group (txg) is
 * assigned a 64-bit consecutive identifier. There are three active
 * transaction group states: open, quiescing, or syncing. At any given time,
 * there may be an active txg associated with each state; each active txg may
 * either be processing, or blocked waiting to enter the next state. There may
 * be up to three active txgs, and there is always a txg in the open state
 * (though it may be blocked waiting to enter the quiescing state). In broad
 * strokes, transactions -- operations that change in-memory structures -- are
 * accepted into the txg in the open state, and are completed while the txg is
 * in the open or quiescing states. The accumulated changes are written to
 * disk in the syncing state.
 *
 * Open
 *
 * When a new txg becomes active, it first enters the open state. New
 * transactions -- updates to in-memory structures -- are assigned to the
```



```
 * currently open txg. There is always a txg in the open state so that ZFS can
 * accept new changes (though the txg may refuse new changes if it has hit
 * some limit). ZFS advances the open txg to the next state for a variety of
 * reasons such as it hitting a time or size threshold, or the execution of an
 * administrative action that must be completed in the syncing state.
 *
 * Quiescing
 *
 * After a txg exits the open state, it enters the quiescing state. The
 * quiescing state is intended to provide a buffer between accepting new
 * transactions in the open state and writing them out to stable storage in
 * the syncing state. While quiescing, transactions can continue their
 * operation without delaying either of the other states. Typically, a txg is
 * in the quiescing state very briefly since the operations are bounded by
 * software latencies rather than, say, slower I/O latencies. After all
 * transactions complete, the txg is ready to enter the next state.
 *
 * Syncing
 *
 * In the syncing state, the in-memory state built up during the open and (to
 * a lesser degree) the quiescing states is written to stable storage. The
 * process of writing out modified data can, in turn modify more data. For
 * example when we write new blocks, we need to allocate space for them; those
 * allocations modify metadata (space maps)... which themselves must be
 * written to stable storage. During the sync state, ZFS iterates, writing out
 * data until it converges and all in-memory changes have been written out.
 * The first such pass is the largest as it encompasses all the modified user
 * data (as opposed to filesystem metadata). Subsequent passes typically have
 * far less data to write as they consist exclusively of filesystem metadata.
 *
 * To ensure convergence, after a certain number of passes ZFS begins
 * overwriting locations on stable storage that had been allocated earlier in
 * the syncing state (and subsequently freed). ZFS usually allocates new
 * blocks to optimize for large, continuous, writes. For the syncing state to
 * converge however it must complete a pass where no new blocks are allocated
 * since each allocation requires a modification of persistent metadata.
 * Further, to hasten convergence, after a prescribed number of passes, ZFS
 * also defers frees, and stops compressing.
 *
 * In addition to writing out user data, we must also execute synctasks during
 * the syncing context. A synctask is the mechanism by which some
 * administrative activities work such as creating and destroying snapshots or
 * datasets. Note that when a synctask is initiated it enters the open txg,
 * and ZFS then pushes that txg as quickly as possible to completion of the
 * syncing state in order to reduce the latency of the administrative
 * activity. To complete the syncing state, ZFS writes out a new uberblock,
 * the root of the tree of blocks that comprise all state stored on the ZFS
 * pool. Finally, if there is a quiesced txg waiting, we signal that it can
 * now transition to the syncing state.
 */

static void txg_sync_thread(dsl_pool_t *dp);
static void txg_quiesce_thread(dsl_pool_t *dp);

int zfs_txg_timeout = 5;        /* max seconds worth of delta per txg */

/*
 * Prepare the txg subsystem.
 */
void
txg_init(dsl_pool_t *dp, uint64_t txg)
{
        tx_state_t *tx = &dp->dp_tx;
        int c;
        bzero(tx, sizeof (tx_state_t));
```

```
123         tx->tx_cpu = kmem_zalloc(max_ncpus * sizeof (tx_cpu_t), KM_SLEEP);

125         for (c = 0; c < max_ncpus; c++) {
126                 int i;

128                 mutex_init(&tx->tx_cpu[c].tc_lock, NULL, MUTEX_DEFAULT, NULL);
129                 mutex_init(&tx->tx_cpu[c].tc_open_lock, NULL, MUTEX_DEFAULT,
130                     NULL);
131                 for (i = 0; i < TXG_SIZE; i++) {
132                         cv_init(&tx->tx_cpu[c].tc_cv[i], NULL, CV_DEFAULT,
133                             NULL);
134                         list_create(&tx->tx_cpu[c].tc_callbacks[i],
135                             sizeof (dmu_tx_callback_t),
136                             offsetof(dmu_tx_callback_t, dcb_node));
137                 }
138         }

140         mutex_init(&tx->tx_sync_lock, NULL, MUTEX_DEFAULT, NULL);

142         cv_init(&tx->tx_sync_more_cv, NULL, CV_DEFAULT, NULL);
143         cv_init(&tx->tx_sync_done_cv, NULL, CV_DEFAULT, NULL);
144         cv_init(&tx->tx_quiesce_more_cv, NULL, CV_DEFAULT, NULL);
145         cv_init(&tx->tx_quiesce_done_cv, NULL, CV_DEFAULT, NULL);
146         cv_init(&tx->tx_exit_cv, NULL, CV_DEFAULT, NULL);

148         tx->tx_open_txg = txg;
149 }
```

unchanged_portion_omitted_

```
346 /*
347  * Blocks until all transactions in the group are committed.
348  *
349  * On return, the transaction group has reached a stable state in which it can
350  * then be passed off to the syncing context.
351  */
352 static void
353 txg_quiesce(dsl_pool_t *dp, uint64_t txg)
354 {
355         tx_state_t *tx = &dp->dp_tx;
356         int g = txg & TXG_MASK;
357         int c;

359         /*
360          * Grab all tc_open_locks so nobody else can get into this txg.
361          */
362         for (c = 0; c < max_ncpus; c++)
363                 mutex_enter(&tx->tx_cpu[c].tc_open_lock);

365         ASSERT(txg == tx->tx_open_txg);
366         tx->tx_open_txg++;
367         tx->tx_open_time = gethrtime();

369         DTRACE_PROBE2(txg__quiescing, dsl_pool_t *, dp, uint64_t, txg);
370         DTRACE_PROBE2(txg__opened, dsl_pool_t *, dp, uint64_t, tx->tx_open_txg);

372         /*
373          * Now that we've incremented tx_open_txg, we can let threads
374          * enter the next transaction group.
375          */
376         for (c = 0; c < max_ncpus; c++)
377                 mutex_exit(&tx->tx_cpu[c].tc_open_lock);

379         /*
380          * Quiesce the transaction group by waiting for everyone to txg_exit().
381          */
382         for (c = 0; c < max_ncpus; c++) {
```



```
383                 tx_cpu_t *tc = &tx->tx_cpu[c];
384                 mutex_enter(&tc->tc_lock);
385                 while (tc->tc_count[g] != 0)
386                         cv_wait(&tc->tc_cv[g], &tc->tc_lock);
387                 mutex_exit(&tc->tc_lock);
388         }
389 }
```

unchanged_portion_omitted_

```
446 static void
447 txg_sync_thread(dsl_pool_t *dp)
448 {
449         spa_t *spa = dp->dp_spa;
450         tx_state_t *tx = &dp->dp_tx;
451         callb_cpr_t cpr;
452         uint64_t start, delta;

454         txg_thread_enter(tx, &cpr);

456         start = delta = 0;
457         for (;;) {
458                 uint64_t timeout = zfs_txg_timeout * hz;
459                 uint64_t timer;
457                 uint64_t timer, timeout = zfs_txg_timeout * hz;
460                 uint64_t txg;

462                 /*
463                  * We sync when we're scanning, there's someone waiting
464                  * on us, or the quiesce thread has handed off a txg to
465                  * us, or we have reached our timeout.
466                  */
467                 timer = (delta >= timeout ? 0 : timeout - delta);
468                 while (!dsl_scan_active(dp->dp_scan) &&
469                     !tx->tx_exiting && timer > 0 &&
470                     tx->tx_synced_txg >= tx->tx_sync_txg_waiting &&
471                     tx->tx_quiesced_txg == 0 &&
472                     dp->dp_dirty_total < zfs_dirty_data_sync) {
469                     tx->tx_quiesced_txg == 0) {
473                         dprintf("waiting; tx_synced=%llu waiting=%llu dp=%p\n",
474                             tx->tx_synced_txg, tx->tx_sync_txg_waiting, dp);
475                         txg_thread_wait(tx, &cpr, &tx->tx_sync_more_cv, timer);
476                         delta = ddi_get_lbolt() - start;
477                         timer = (delta > timeout ? 0 : timeout - delta);
478                 }

480                 /*
481                  * Wait until the quiesce thread hands off a txg to us,
482                  * prompting it to do so if necessary.
483                  */
484                 while (!tx->tx_exiting && tx->tx_quiesced_txg == 0) {
485                         if (tx->tx_quiesce_txg_waiting < tx->tx_open_txg+1)
486                                 tx->tx_quiesce_txg_waiting = tx->tx_open_txg+1;
487                         cv_broadcast(&tx->tx_quiesce_more_cv);
488                         txg_thread_wait(tx, &cpr, &tx->tx_quiesce_done_cv, 0);
489                 }

491                 if (tx->tx_exiting)
492                         txg_thread_exit(tx, &cpr, &tx->tx_sync_thread);

494                 /*
495                  * Consume the quiesced txg which has been handed off to
496                  * us.  This may cause the quiescing thread to now be
497                  * able to quiesce another txg, so we must signal it.
498                  */
499                 txg = tx->tx_quiesced_txg;
500                 tx->tx_quiesced_txg = 0;
```

```
501                 tx->tx_syncing_txg = txg;
502                 DTRACE_PROBE2(txg__syncing, dsl_pool_t *, dp, uint64_t, txg);
503                 cv_broadcast(&tx->tx_quiesce_more_cv);
505                 dprintf("txg=%llu quiesce_txg=%llu sync_txg=%llu\n",
506                     txg, tx->tx_quiesce_txg_waiting, tx->tx_sync_txg_waiting);
507                 mutex_exit(&tx->tx_sync_lock);
509                 start = ddi_get_lbolt();
510                 spa_sync(spa, txg);
511                 delta = ddi_get_lbolt() - start;
513                 mutex_enter(&tx->tx_sync_lock);
514                 tx->tx_synced_txg = txg;
515                 tx->tx_syncing_txg = 0;
516                 DTRACE_PROBE2(txg__synced, dsl_pool_t *, dp, uint64_t, txg);
517                 cv_broadcast(&tx->tx_sync_done_cv);
519                 /*
520                  * Dispatch commit callbacks to worker threads.
521                  */
522                 txg_dispatch_callbacks(dp, txg);
523         }
524 }
```

*unchanged_portion_omitted_*

```
627 void
628 txg_wait_open(dsl_pool_t *dp, uint64_t txg)
629 {
630         tx_state_t *tx = &dp->dp_tx;
632         ASSERT(!dsl_pool_config_held(dp));
634         mutex_enter(&tx->tx_sync_lock);
635         ASSERT(tx->tx_threads == 2);
636         if (txg == 0)
637                 txg = tx->tx_open_txg + 1;
638         if (tx->tx_quiesce_txg_waiting < txg)
639                 tx->tx_quiesce_txg_waiting = txg;
640         dprintf("txg=%llu quiesce_txg=%llu sync_txg=%llu\n",
641             txg, tx->tx_quiesce_txg_waiting, tx->tx_sync_txg_waiting);
642         while (tx->tx_open_txg < txg) {
643                 cv_broadcast(&tx->tx_quiesce_more_cv);
644                 cv_wait(&tx->tx_quiesce_done_cv, &tx->tx_sync_lock);
645         }
646         mutex_exit(&tx->tx_sync_lock);
647 }
649 /*
650  * If there isn't a txg syncing or in the pipeline, push another txg through
651  * the pipeline by queiscing the open txg.
652  */
653 void
654 txg_kick(dsl_pool_t *dp)
655 {
656         tx_state_t *tx = &dp->dp_tx;
658         ASSERT(!dsl_pool_config_held(dp));
660         mutex_enter(&tx->tx_sync_lock);
661         if (tx->tx_syncing_txg == 0 &&
662             tx->tx_quiesce_txg_waiting <= tx->tx_open_txg &&
663             tx->tx_sync_txg_waiting <= tx->tx_synced_txg &&
664             tx->tx_quiesced_txg <= tx->tx_synced_txg) {
665                 tx->tx_quiesce_txg_waiting = tx->tx_open_txg + 1;
666                 cv_broadcast(&tx->tx_quiesce_more_cv);
```



```
667         }
668         mutex_exit(&tx->tx_sync_lock);
669 }
```

*unchanged_portion_omitted_*

****
88665 Thu Aug 22 16:15:35 2013
new/usr/src/uts/common/fs/zfs/vdev.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****
_____***unchanged_portion_omitted_***

```
3247 void
3248 vdev_deadman(vdev_t *vd)
3249 {
3250         for (int c = 0; c < vd->vdev_children; c++) {
3251                 vdev_t *cvd = vd->vdev_child[c];

3253                 vdev_deadman(cvd);
3254         }

3256         if (vd->vdev_ops->vdev_op_leaf) {
3257                 vdev_queue_t *vq = &vd->vdev_queue;

3259                 mutex_enter(&vq->vq_lock);
3260                 if (avl_numnodes(&vq->vq_active_tree) > 0) {
3260                 if (avl_numnodes(&vq->vq_pending_tree) > 0) {
3261                         spa_t *spa = vd->vdev_spa;
3262                         zio_t *fio;
3263                         uint64_t delta;

3265                         /*
3266                          * Look at the head of all the pending queues,
3267                          * if any I/O has been outstanding for longer than
3268                          * the spa_deadman_synctime we panic the system.
3269                          */
3270                         fio = avl_first(&vq->vq_active_tree);
3270                         fio = avl_first(&vq->vq_pending_tree);
3271                         delta = gethrtime() - fio->io_timestamp;
3272                         if (delta > spa_deadman_synctime(spa)) {
3273                                 zfs_dbgmsg("SLOW IO: zio timestamp %lluns, "
3274                                     "delta %lluns, last io %lluns",
3275                                     fio->io_timestamp, delta,
3276                                     vq->vq_io_complete_ts);
3277                                 fm_panic("I/O to pool '%s' appears to be "
3278                                     "hung.", spa_name(spa));
3279                         }
3280                 }
3281                 mutex_exit(&vq->vq_lock);
3282         }
3283 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11413 Thu Aug 22 16:15:36 2013
new/usr/src/uts/common/fs/zfs/vdev_cache.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 250 /*
 251  * Read data from the cache.  Returns 0 on cache hit, errno on a miss.
 252  */
 253 int
 254 vdev_cache_read(zio_t *zio)
 255 {
 256         vdev_cache_t *vc = &zio->io_vd->vdev_cache;
 257         vdev_cache_entry_t *ve, ve_search;
 258         uint64_t cache_offset = P2ALIGN(zio->io_offset, VCBS);
 259         uint64_t cache_phase = P2PHASE(zio->io_offset, VCBS);
 260         zio_t *fio;

 262         ASSERT(zio->io_type == ZIO_TYPE_READ);

 264         if (zio->io_flags & ZIO_FLAG_DONT_CACHE)
 265                 return (SET_ERROR(EINVAL));

 267         if (zio->io_size > zfs_vdev_cache_max)
 268                 return (SET_ERROR(EOVERFLOW));

 270         /*
 271          * If the I/O straddles two or more cache blocks, don't cache it.
 272          */
 273         if (P2BOUNDARY(zio->io_offset, zio->io_size, VCBS))
 274                 return (SET_ERROR(EXDEV));

 276         ASSERT(cache_phase + zio->io_size <= VCBS);

 278         mutex_enter(&vc->vc_lock);

 280         ve_search.ve_offset = cache_offset;
 281         ve = avl_find(&vc->vc_offset_tree, &ve_search, NULL);

 283         if (ve != NULL) {
 284                 if (ve->ve_missed_update) {
 285                         mutex_exit(&vc->vc_lock);
 286                         return (SET_ERROR(ESTALE));
 287                 }

 289                 if ((fio = ve->ve_fill_io) != NULL) {
 290                         zio_vdev_io_bypass(zio);
 291                         zio_add_child(zio, fio);
 292                         mutex_exit(&vc->vc_lock);
 293                         VDCSTAT_BUMP(vdc_stat_delegations);
 294                         return (0);
 295                 }

 297                 vdev_cache_hit(vc, ve, zio);
 298                 zio_vdev_io_bypass(zio);

 300                 mutex_exit(&vc->vc_lock);
 301                 VDCSTAT_BUMP(vdc_stat_hits);
 302                 return (0);
 303         }

 305         ve = vdev_cache_allocate(zio);
```



```
 307         if (ve == NULL) {
 308                 mutex_exit(&vc->vc_lock);
 309                 return (SET_ERROR(ENOMEM));
 310         }

 312         fio = zio_vdev_delegated_io(zio->io_vd, cache_offset,
 313             ve->ve_data, VCBS, ZIO_TYPE_READ, ZIO_PRIORITY_NOW,
 313             ve->ve_data, VCBS, ZIO_TYPE_READ, ZIO_PRIORITY_CACHE_FILL,
 314             ZIO_FLAG_DONT_CACHE, vdev_cache_fill, ve);

 316         ve->ve_fill_io = fio;
 317         zio_vdev_io_bypass(zio);
 318         zio_add_child(zio, fio);

 320         mutex_exit(&vc->vc_lock);
 321         zio_nowait(fio);
 322         VDCSTAT_BUMP(vdc_stat_misses);

 324         return (0);
 325 }
______unchanged_portion_omitted_
```

```
**********************************************************
   12213 Thu Aug 22 16:15:38 2013
new/usr/src/uts/common/fs/zfs/vdev_mirror.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 331 static void
 332 vdev_mirror_io_done(zio_t *zio)
 333 {
 334         mirror_map_t *mm = zio->io_vsd;
 335         mirror_child_t *mc;
 336         int c;
 337         int good_copies = 0;
 338         int unexpected_errors = 0;

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

 343                 if (mc->mc_error) {
 344                         if (!mc->mc_skipped)
 345                                 unexpected_errors++;
 346                 } else if (mc->mc_tried) {
 347                         good_copies++;
 348                 }
 349         }

 351         if (zio->io_type == ZIO_TYPE_WRITE) {
 352                 /*
 353                  * XXX -- for now, treat partial writes as success.
 354                  *
 355                  * Now that we support write reallocation, it would be better
 356                  * to treat partial failure as real failure unless there are
 357                  * no non-degraded top-level vdevs left, and not update DTLs
 358                  * if we intend to reallocate.
 359                  */
 360                 /* XXPOLICY */
 361                 if (good_copies != mm->mm_children) {
 362                         /*
 363                          * Always require at least one good copy.
 364                          *
 365                          * For ditto blocks (io_vd == NULL), require
 366                          * all copies to be good.
 367                          *
 368                          * XXX -- for replacing vdevs, there's no great answer.
 369                          * If the old device is really dead, we may not even
 370                          * be able to access it -- so we only want to
 371                          * require good writes to the new device.  But if
 372                          * the new device turns out to be flaky, we want
 373                          * to be able to detach it -- which requires all
 374                          * writes to the old device to have succeeded.
 375                          */
 376                         if (good_copies == 0 || zio->io_vd == NULL)
 377                                 zio->io_error = vdev_mirror_worst_error(mm);
 378                 }
 379                 return;
 380         }

 382         ASSERT(zio->io_type == ZIO_TYPE_READ);

 384         /*
 385          * If we don't have a good copy yet, keep trying other children.
 386          */
```



```
 387         /* XXPOLICY */
 388         if (good_copies == 0 && (c = vdev_mirror_child_select(zio)) != -1) {
 389                 ASSERT(c >= 0 && c < mm->mm_children);
 390                 mc = &mm->mm_child[c];
 391                 zio_vdev_io_redone(zio);
 392                 zio_nowait(zio_vdev_child_io(zio, zio->io_bp,
 393                     mc->mc_vd, mc->mc_offset, zio->io_data, zio->io_size,
 394                     ZIO_TYPE_READ, zio->io_priority, 0,
 395                     vdev_mirror_child_done, mc));
 396                 return;
 397         }

 399         /* XXPOLICY */
 400         if (good_copies == 0) {
 401                 zio->io_error = vdev_mirror_worst_error(mm);
 402                 ASSERT(zio->io_error != 0);
 403         }

 405         if (good_copies && spa_writeable(zio->io_spa) &&
 406             (unexpected_errors ||
 407             (zio->io_flags & ZIO_FLAG_RESILVER) ||
 408             ((zio->io_flags & ZIO_FLAG_SCRUB) && mm->mm_replacing))) {
 409                 /*
 410                  * Use the good data we have in hand to repair damaged children.
 411                  */
 412                 for (c = 0; c < mm->mm_children; c++) {
 413                         /*
 414                          * Don't rewrite known good children.
 415                          * Not only is it unnecessary, it could
 416                          * actually be harmful: if the system lost
 417                          * power while rewriting the only good copy,
 418                          * there would be no good copies left!
 419                          */
 420                         mc = &mm->mm_child[c];

 422                         if (mc->mc_error == 0) {
 423                                 if (mc->mc_tried)
 424                                         continue;
 425                                 if (!(zio->io_flags & ZIO_FLAG_SCRUB) &&
 426                                     !vdev_dtl_contains(mc->mc_vd, DTL_PARTIAL,
 427                                     zio->io_txg, 1))
 428                                         continue;
 429                                 mc->mc_error = SET_ERROR(ESTALE);
 430                         }

 432                         zio_nowait(zio_vdev_child_io(zio, zio->io_bp,
 433                             mc->mc_vd, mc->mc_offset,
 434                             zio->io_data, zio->io_size,
 435                             ZIO_TYPE_WRITE, ZIO_PRIORITY_ASYNC_WRITE,
 435                             ZIO_TYPE_WRITE, zio->io_priority,
 436                             ZIO_FLAG_IO_REPAIR | (unexpected_errors ?
 437                             ZIO_FLAG_SELF_HEAL : 0), NULL, NULL));
 438                 }
 439         }
 440 }
______unchanged_portion_omitted_
```

```
**********************************************************
   22931 Thu Aug 22 16:15:39 2013
new/usr/src/uts/common/fs/zfs/vdev_queue.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/vdev_impl.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/avl.h>
#include <sys/dsl_pool.h>

/*
 * ZFS I/O Scheduler
 * ---------------
 *
 * ZFS issues I/O operations to leaf vdevs to satisfy and complete zios.  The
 * I/O scheduler determines when and in what order those operations are
 * issued.  The I/O scheduler divides operations into five I/O classes
 * prioritized in the following order: sync read, sync write, async read,
 * async write, and scrub/resilver.  Each queue defines the minimum and
 * maximum number of concurrent operations that may be issued to the device.
 * In addition, the device has an aggregate maximum. Note that the sum of the
 * per-queue minimums must not exceed the aggregate maximum, and if the
 * aggregate maximum is equal to or greater than the sum of the per-queue
 * maximums, the per-queue minimum has no effect.
 *
 * For many physical devices, throughput increases with the number of
 * concurrent operations, but latency typically suffers. Further, physical
 * devices typically have a limit at which more concurrent operations have no
 * effect on throughput or can actually cause it to decrease.
 *
 * The scheduler selects the next operation to issue by first looking for an
```



```
 * I/O class whose minimum has not been satisfied. Once all are satisfied and
 * the aggregate maximum has not been hit, the scheduler looks for classes
 * whose maximum has not been satisfied. Iteration through the I/O classes is
 * done in the order specified above. No further operations are issued if the
 * aggregate maximum number of concurrent operations has been hit or if there
 * are no operations queued for an I/O class that has not hit its maximum.
 * Every time an i/o is queued or an operation completes, the I/O scheduler
 * looks for new operations to issue.
 *
 * All I/O classes have a fixed maximum number of outstanding operations
 * except for the async write class. Asynchronous writes represent the data
 * that is committed to stable storage during the syncing stage for
 * transaction groups (see txg.c). Transaction groups enter the syncing state
 * periodically so the number of queued async writes will quickly burst up and
 * then bleed down to zero. Rather than servicing them as quickly as possible,
 * the I/O scheduler changes the maximum number of active async write i/os
 * according to the amount of dirty data in the pool (see dsl_pool.c). Since
 * both throughput and latency typically increase with the number of
 * concurrent operations issued to physical devices, reducing the burstiness
 * in the number of concurrent operations also stabilizes the response time of
 * operations from other -- and in particular synchronous -- queues. In broad
 * strokes, the I/O scheduler will issue more concurrent operations from the
 * async write queue as there's more dirty data in the pool.
 *
 * Async Writes
 *
 * The number of concurrent operations issued for the async write I/O class
 * follows a piece-wise linear function defined by a few adjustable points.
 *
 *        |                   o---------| <-- zfs_vdev_async_write_max_active
 *   ^    |                  /^         |
 *   |    |                 / |         |
 * active |                /  |         |
 *  I/O   |               /   |         |
 * count  |              /    |         |
 *        |             /     |         |
 *        |------------o      |         | <-- zfs_vdev_async_write_min_active
 *       0|___________^______|_________|
 *        0%          |      |       100% of zfs_dirty_data_max
 *                    |      |
 *                    |      `-- zfs_vdev_async_write_active_max_dirty_percent
 *                    `--------- zfs_vdev_async_write_active_min_dirty_percent
 *
 * Until the amount of dirty data exceeds a minimum percentage of the dirty
 * data allowed in the pool, the I/O scheduler will limit the number of
 * concurrent operations to the minimum. As that threshold is crossed, the
 * number of concurrent operations issued increases linearly to the maximum at
 * the specified maximum percentage of the dirty data allowed in the pool.
 *
 * Ideally, the amount of dirty data on a busy pool will stay in the sloped
 * part of the function between zfs_vdev_async_write_active_min_dirty_percent
 * and zfs_vdev_async_write_active_max_dirty_percent. If it exceeds the
 * maximum percentage, this indicates that the rate of incoming data is
 * greater than the rate that the backend storage can handle. In this case, we
 * must further throttle incoming writes (see dmu_tx_delay() for details).
 * These tunables are for performance analysis.
 */

/*
 * The maximum number of i/os active to each device.  Ideally, this will be >=
 * the sum of each queue's max_active.  It must be at least the sum of each
 * queue's min_active.
 */
uint32_t zfs_vdev_max_active = 1000;
/* The maximum number of I/Os concurrently pending to each device. */
int zfs_vdev_max_pending = 10;
```

```
122 /*
123  * Per-queue limits on the number of i/os active to each device.  If the
124  * sum of the queue's max_active is < zfs_vdev_max_active, then the
125  * min_active comes into play.  We will send min_active from each queue,
126  * and then select from queues in the order defined by zio_priority_t.
127  *
128  * In general, smaller max_active's will lead to lower latency of synchronous
129  * operations.  Larger max_active's may lead to higher overall throughput,
130  * depending on underlying storage.
131  *
132  * The ratio of the queues' max_actives determines the balance of performance
133  * between reads, writes, and scrubs.  E.g., increasing
134  * zfs_vdev_scrub_max_active will cause the scrub or resilver to complete
135  * more quickly, but reads and writes to have higher latency and lower
136  * throughput.
 44  * The initial number of I/Os pending to each device, before it starts ramping
 45  * up to zfs_vdev_max_pending.
137  */
138 uint32_t zfs_vdev_sync_read_min_active = 10;
139 uint32_t zfs_vdev_sync_read_max_active = 10;
140 uint32_t zfs_vdev_sync_write_min_active = 10;
141 uint32_t zfs_vdev_sync_write_max_active = 10;
142 uint32_t zfs_vdev_async_read_min_active = 1;
143 uint32_t zfs_vdev_async_read_max_active = 3;
144 uint32_t zfs_vdev_async_write_min_active = 1;
145 uint32_t zfs_vdev_async_write_max_active = 10;
146 uint32_t zfs_vdev_scrub_min_active = 1;
147 uint32_t zfs_vdev_scrub_max_active = 2;
 47 int zfs_vdev_min_pending = 4;

149 /*
150  * When the pool has less than zfs_vdev_async_write_active_min_dirty_percent
151  * dirty data, use zfs_vdev_async_write_min_active.  When it has more than
152  * zfs_vdev_async_write_active_max_dirty_percent, use
153  * zfs_vdev_async_write_max_active. The value is linearly interpolated
154  * between min and max.
 50  * The deadlines are grouped into buckets based on zfs_vdev_time_shift:
 51  * deadline = pri + gethrtime() >> time_shift)
155  */
156 int zfs_vdev_async_write_active_min_dirty_percent = 30;
157 int zfs_vdev_async_write_active_max_dirty_percent = 60;
 53 int zfs_vdev_time_shift = 29; /* each bucket is 0.537 seconds */

 55 /* exponential I/O issue ramp-up rate */
 56 int zfs_vdev_ramp_rate = 2;

159 /*
160  * To reduce IOPs, we aggregate small adjacent I/Os into one large I/O.
161  * For read I/Os, we also aggregate across small adjacency gaps; for writes
162  * we include spans of optional I/Os to aid aggregation at the disk even when
163  * they aren't able to help us aggregate at this level.
164  */
165 int zfs_vdev_aggregation_limit = SPA_MAXBLOCKSIZE;
166 int zfs_vdev_read_gap_limit = 32 << 10;
167 int zfs_vdev_write_gap_limit = 4 << 10;

 68 /*
 69  * Virtual device vector for disk I/O scheduling.
 70  */
169 int
170 vdev_queue_offset_compare(const void *x1, const void *x2)
 72 vdev_queue_deadline_compare(const void *x1, const void *x2)
171 {
172         const zio_t *z1 = x1;
173         const zio_t *z2 = x2;
```



```
 77         if (z1->io_deadline < z2->io_deadline)
 78                 return (-1);
 79         if (z1->io_deadline > z2->io_deadline)
 80                 return (1);

175         if (z1->io_offset < z2->io_offset)
176                 return (-1);
177         if (z1->io_offset > z2->io_offset)
178                 return (1);

180         if (z1 < z2)
181                 return (-1);
182         if (z1 > z2)
183                 return (1);

185         return (0);
186 }

188 int
189 vdev_queue_timestamp_compare(const void *x1, const void *x2)
 96 vdev_queue_offset_compare(const void *x1, const void *x2)
190 {
191         const zio_t *z1 = x1;
192         const zio_t *z2 = x2;

194         if (z1->io_timestamp < z2->io_timestamp)
101         if (z1->io_offset < z2->io_offset)
195                 return (-1);
196         if (z1->io_timestamp > z2->io_timestamp)
103         if (z1->io_offset > z2->io_offset)
197                 return (1);

199         if (z1 < z2)
200                 return (-1);
201         if (z1 > z2)
202                 return (1);

204         return (0);
205 }

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

212         mutex_init(&vq->vq_lock, NULL, MUTEX_DEFAULT, NULL);
213         vq->vq_vdev = vd;

215         avl_create(&vq->vq_active_tree, vdev_queue_offset_compare,
216             sizeof (zio_t), offsetof(struct zio, io_queue_node));
121         avl_create(&vq->vq_deadline_tree, vdev_queue_deadline_compare,
122             sizeof (zio_t), offsetof(struct zio, io_deadline_node));

218         for (zio_priority_t p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++) {
219                 /*
220                  * The synchronous i/o queues are FIFO rather than LBA ordered.
221                  * This provides more consistent latency for these i/os, and
222                  * they tend to not be tightly clustered anyway so there is
223                  * little to no throughput loss.
224                  */
225                 boolean_t fifo = (p == ZIO_PRIORITY_SYNC_READ ||
226                     p == ZIO_PRIORITY_SYNC_WRITE);
227                 avl_create(&vq->vq_class[p].vqc_queued_tree,
228                     fifo ? vdev_queue_timestamp_compare :
229                     vdev_queue_offset_compare,
```

```
230                     sizeof (zio_t), offsetof(struct zio, io_queue_node));
231         }
124         avl_create(&vq->vq_read_tree, vdev_queue_offset_compare,
125             sizeof (zio_t), offsetof(struct zio, io_offset_node));

127         avl_create(&vq->vq_write_tree, vdev_queue_offset_compare,
128             sizeof (zio_t), offsetof(struct zio, io_offset_node));

130         avl_create(&vq->vq_pending_tree, vdev_queue_offset_compare,
131             sizeof (zio_t), offsetof(struct zio, io_offset_node));
232 }

234 void
235 vdev_queue_fini(vdev_t *vd)
236 {
237         vdev_queue_t *vq = &vd->vdev_queue;

239         for (zio_priority_t p = 0; p < ZIO_PRIORITY_NUM_QUEUEABLE; p++)
240                 avl_destroy(&vq->vq_class[p].vqc_queued_tree);
241         avl_destroy(&vq->vq_active_tree);
139         avl_destroy(&vq->vq_deadline_tree);
140         avl_destroy(&vq->vq_read_tree);
141         avl_destroy(&vq->vq_write_tree);
142         avl_destroy(&vq->vq_pending_tree);

243         mutex_destroy(&vq->vq_lock);
244 }

246 static void
247 vdev_queue_io_add(vdev_queue_t *vq, zio_t *zio)
248 {
249         spa_t *spa = zio->io_spa;
250         ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
251         avl_add(&vq->vq_class[zio->io_priority].vqc_queued_tree, zio);
151         avl_add(&vq->vq_deadline_tree, zio);
152         avl_add(zio->io_vdev_tree, zio);

154         if (spa->spa_iokstat != NULL) {
253         mutex_enter(&spa->spa_iokstat_lock);
254         spa->spa_queue_stats[zio->io_priority].spa_queued++;
255         if (spa->spa_iokstat != NULL)
256                 kstat_waitq_enter(spa->spa_iokstat->ks_data);
257         mutex_exit(&spa->spa_iokstat_lock);
158         }
258 }

260 static void
261 vdev_queue_io_remove(vdev_queue_t *vq, zio_t *zio)
262 {
263         spa_t *spa = zio->io_spa;
264         ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
265         avl_remove(&vq->vq_class[zio->io_priority].vqc_queued_tree, zio);
165         avl_remove(&vq->vq_deadline_tree, zio);
166         avl_remove(zio->io_vdev_tree, zio);

168         if (spa->spa_iokstat != NULL) {
267         mutex_enter(&spa->spa_iokstat_lock);
268         ASSERT3U(spa->spa_queue_stats[zio->io_priority].spa_queued, >, 0);
269         spa->spa_queue_stats[zio->io_priority].spa_queued--;
270         if (spa->spa_iokstat != NULL)
271                 kstat_waitq_exit(spa->spa_iokstat->ks_data);
272         mutex_exit(&spa->spa_iokstat_lock);
172         }
273 }

275 static void
```



```
276 vdev_queue_pending_add(vdev_queue_t *vq, zio_t *zio)
277 {
278         spa_t *spa = zio->io_spa;
279         ASSERT(MUTEX_HELD(&vq->vq_lock));
280         ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
281         vq->vq_class[zio->io_priority].vqc_active++;
282         avl_add(&vq->vq_active_tree, zio);

179         avl_add(&vq->vq_pending_tree, zio);
180         if (spa->spa_iokstat != NULL) {
284         mutex_enter(&spa->spa_iokstat_lock);
285         spa->spa_queue_stats[zio->io_priority].spa_active++;
286         if (spa->spa_iokstat != NULL)
287                 kstat_runq_enter(spa->spa_iokstat->ks_data);
288         mutex_exit(&spa->spa_iokstat_lock);
184         }
289 }

291 static void
292 vdev_queue_pending_remove(vdev_queue_t *vq, zio_t *zio)
293 {
294         spa_t *spa = zio->io_spa;
295         ASSERT(MUTEX_HELD(&vq->vq_lock));
296         ASSERT3U(zio->io_priority, <, ZIO_PRIORITY_NUM_QUEUEABLE);
297         vq->vq_class[zio->io_priority].vqc_active--;
298         avl_remove(&vq->vq_active_tree, zio);

300         mutex_enter(&spa->spa_iokstat_lock);
301         ASSERT3U(spa->spa_queue_stats[zio->io_priority].spa_active, >, 0);
302         spa->spa_queue_stats[zio->io_priority].spa_active--;
191         avl_remove(&vq->vq_pending_tree, zio);
303         if (spa->spa_iokstat != NULL) {
304                 kstat_io_t *ksio = spa->spa_iokstat->ks_data;

195                 mutex_enter(&spa->spa_iokstat_lock);
306                 kstat_runq_exit(spa->spa_iokstat->ks_data);
307                 if (zio->io_type == ZIO_TYPE_READ) {
308                         ksio->reads++;
309                         ksio->nread += zio->io_size;
310                 } else if (zio->io_type == ZIO_TYPE_WRITE) {
311                         ksio->writes++;
312                         ksio->nwritten += zio->io_size;
313                 }
314         }
315         mutex_exit(&spa->spa_iokstat_lock);
205         }
316 }

318 static void
319 vdev_queue_agg_io_done(zio_t *aio)
320 {
321         if (aio->io_type == ZIO_TYPE_READ) {
322                 zio_t *pio;
323                 while ((pio = zio_walk_parents(aio)) != NULL) {

213         while ((pio = zio_walk_parents(aio)) != NULL)
214                 if (aio->io_type == ZIO_TYPE_READ)
324                         bcopy((char *)aio->io_data + (pio->io_offset -
325                             aio->io_offset), pio->io_data, pio->io_size);
326                 }
327         }

329         zio_buf_free(aio->io_data, aio->io_size);
330 }

332 static int
```

```
vdev_queue_class_min_active(zio_priority_t p)
{
        switch (p) {
        case ZIO_PRIORITY_SYNC_READ:
                return (zfs_vdev_sync_read_min_active);
        case ZIO_PRIORITY_SYNC_WRITE:
                return (zfs_vdev_sync_write_min_active);
        case ZIO_PRIORITY_ASYNC_READ:
                return (zfs_vdev_async_read_min_active);
        case ZIO_PRIORITY_ASYNC_WRITE:
                return (zfs_vdev_async_write_min_active);
        case ZIO_PRIORITY_SCRUB:
                return (zfs_vdev_scrub_min_active);
        default:
                panic("invalid priority %u", p);
                return (0);
        }
}

static int
vdev_queue_max_async_writes(uint64_t dirty)
{
        int writes;
        uint64_t min_bytes = zfs_dirty_data_max *
            zfs_vdev_async_write_active_min_dirty_percent / 100;
        uint64_t max_bytes = zfs_dirty_data_max *
            zfs_vdev_async_write_active_max_dirty_percent / 100;

        if (dirty < min_bytes)
                return (zfs_vdev_async_write_min_active);
        if (dirty > max_bytes)
                return (zfs_vdev_async_write_max_active);

        /*
         * linear interpolation:
         * slope = (max_writes - min_writes) / (max_bytes - min_bytes)
         * move right by min_bytes
         * move up by min_writes
         */
        writes = (dirty - min_bytes) *
            (zfs_vdev_async_write_max_active -
            zfs_vdev_async_write_min_active) /
            (max_bytes - min_bytes) +
            zfs_vdev_async_write_min_active;
        ASSERT3U(writes, >=, zfs_vdev_async_write_min_active);
        ASSERT3U(writes, <=, zfs_vdev_async_write_max_active);
        return (writes);
}

static int
vdev_queue_class_max_active(spa_t *spa, zio_priority_t p)
{
        switch (p) {
        case ZIO_PRIORITY_SYNC_READ:
                return (zfs_vdev_sync_read_max_active);
        case ZIO_PRIORITY_SYNC_WRITE:
                return (zfs_vdev_sync_write_max_active);
        case ZIO_PRIORITY_ASYNC_READ:
                return (zfs_vdev_async_read_max_active);
        case ZIO_PRIORITY_ASYNC_WRITE:
                return (vdev_queue_max_async_writes(
                    spa->spa_dsl_pool->dp_dirty_total));
        case ZIO_PRIORITY_SCRUB:
                return (zfs_vdev_scrub_max_active);
        default:
                panic("invalid priority %u", p);
```



```
                return (0);
        }
}

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

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

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

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

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

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

static zio_t *
vdev_queue_aggregate(vdev_queue_t *vq, zio_t *zio)
vdev_queue_io_to_issue(vdev_queue_t *vq, uint64_t pending_limit)
{
        zio_t *first, *last, *aio, *dio, *mandatory, *nio;
        uint64_t maxgap = 0;
        uint64_t size;
        boolean_t stretch = B_FALSE;
        vdev_queue_class_t *vqc = &vq->vq_class[zio->io_priority];
        avl_tree_t *t = &vqc->vqc_queued_tree;
        enum zio_flag flags = zio->io_flags & ZIO_FLAG_AGG_INHERIT;
        zio_t *fio, *lio, *aio, *dio, *nio, *mio;
        avl_tree_t *t;
        int flags;
        uint64_t maxspan = zfs_vdev_aggregation_limit;
        uint64_t maxgap;
        int stretch;
```

```
459         if (zio->io_flags & ZIO_FLAG_DONT_AGGREGATE)
460                 return (NULL);
240 again:
241         ASSERT(MUTEX_HELD(&vq->vq_lock));

462         /*
463          * The synchronous i/o queues are not sorted by LBA, so we can't
464          * find adjacent i/os.  These i/os tend to not be tightly clustered,
465          * or too large to aggregate, so this has little impact on performance.
466          */
467         if (zio->io_priority == ZIO_PRIORITY_SYNC_READ ||
468             zio->io_priority == ZIO_PRIORITY_SYNC_WRITE)
243         if (avl_numnodes(&vq->vq_pending_tree) >= pending_limit ||
244             avl_numnodes(&vq->vq_deadline_tree) == 0)
469                 return (NULL);

471         first = last = zio;
247         fio = lio = avl_first(&vq->vq_deadline_tree);

473         if (zio->io_type == ZIO_TYPE_READ)
474                 maxgap = zfs_vdev_read_gap_limit;
249         t = fio->io_vdev_tree;
250         flags = fio->io_flags & ZIO_FLAG_AGG_INHERIT;
251         maxgap = (t == &vq->vq_read_tree) ? zfs_vdev_read_gap_limit : 0;

253         if (!(flags & ZIO_FLAG_DONT_AGGREGATE)) {
476                 /*
477                  * We can aggregate I/Os that are sufficiently adjacent and of
478                  * the same flavor, as expressed by the AGG_INHERIT flags.
479                  * The latter requirement is necessary so that certain
480                  * attributes of the I/O, such as whether it's a normal I/O
481                  * or a scrub/resilver, can be preserved in the aggregate.
482                  * We can include optional I/Os, but don't allow them
483                  * to begin a range as they add no benefit in that situation.
484                  */

486                 /*
487                  * We keep track of the last non-optional I/O.
488                  */
489                 mandatory = (first->io_flags & ZIO_FLAG_OPTIONAL) ? NULL : first;
267                 mio = (fio->io_flags & ZIO_FLAG_OPTIONAL) ? NULL : fio;

491                 /*
492                  * Walk backwards through sufficiently contiguous I/Os
493                  * recording the last non-option I/O.
494                  */
495                 while ((dio = AVL_PREV(t, first)) != NULL &&
273                     while ((dio = AVL_PREV(t, fio)) != NULL &&
496                     (dio->io_flags & ZIO_FLAG_AGG_INHERIT) == flags &&
497                     IO_SPAN(dio, last) <= zfs_vdev_aggregation_limit &&
498                     IO_GAP(dio, first) <= maxgap) {
499                         first = dio;
500                         if (mandatory == NULL && !(first->io_flags & ZIO_FLAG_OPTIONAL))
501                                 mandatory = first;
275                         IO_SPAN(dio, lio) <= maxspan &&
276                         IO_GAP(dio, fio) <= maxgap) {
277                             fio = dio;
278                             if (mio == NULL && !(fio->io_flags & ZIO_FLAG_OPTIONAL))
279                                     mio = fio;
502         }

504         /*
505          * Skip any initial optional I/Os.
506          */
507         while ((first->io_flags & ZIO_FLAG_OPTIONAL) && first != last) {
```



```
508                 first = AVL_NEXT(t, first);
509                 ASSERT(first != NULL);
285                 while ((fio->io_flags & ZIO_FLAG_OPTIONAL) && fio != lio) {
286                         fio = AVL_NEXT(t, fio);
287                         ASSERT(fio != NULL);
510         }

512         /*
513          * Walk forward through sufficiently contiguous I/Os.
514          */
515         while ((dio = AVL_NEXT(t, last)) != NULL &&
293                 while ((dio = AVL_NEXT(t, lio)) != NULL &&
516             (dio->io_flags & ZIO_FLAG_AGG_INHERIT) == flags &&
517             IO_SPAN(first, dio) <= zfs_vdev_aggregation_limit &&
518             IO_GAP(last, dio) <= maxgap) {
519                 last = dio;
520                 if (!(last->io_flags & ZIO_FLAG_OPTIONAL))
521                         mandatory = last;
295                 IO_SPAN(fio, dio) <= maxspan &&
296                 IO_GAP(lio, dio) <= maxgap) {
297                     lio = dio;
298                     if (!(lio->io_flags & ZIO_FLAG_OPTIONAL))
299                             mio = lio;
522         }

524         /*
525          * Now that we've established the range of the I/O aggregation
526          * we must decide what to do with trailing optional I/Os.
527          * For reads, there's nothing to do. While we are unable to
528          * aggregate further, it's possible that a trailing optional
529          * I/O would allow the underlying device to aggregate with
530          * subsequent I/Os. We must therefore determine if the next
531          * non-optional I/O is close enough to make aggregation
532          * worthwhile.
533          */
534         if (zio->io_type == ZIO_TYPE_WRITE && mandatory != NULL) {
535                 zio_t *nio = last;
312                 stretch = B_FALSE;
313                 if (t != &vq->vq_read_tree && mio != NULL) {
314                         nio = lio;
536                 while ((dio = AVL_NEXT(t, nio)) != NULL &&
537                     IO_GAP(nio, dio) == 0 &&
538                     IO_GAP(mandatory, dio) <= zfs_vdev_write_gap_limit) {
317                         IO_GAP(mio, dio) <= zfs_vdev_write_gap_limit) {
539                         nio = dio;
540                         if (!(nio->io_flags & ZIO_FLAG_OPTIONAL)) {
541                                 stretch = B_TRUE;
542                                 break;
543                         }
544                 }
545         }

547         if (stretch) {
548                 /* This may be a no-op. */
549                 dio = AVL_NEXT(t, last);
328                     VERIFY((dio = AVL_NEXT(t, lio)) != NULL);
550                 dio->io_flags &= ~ZIO_FLAG_OPTIONAL;
551         } else {
552                 while (last != mandatory && last != first) {
553                         ASSERT(last->io_flags & ZIO_FLAG_OPTIONAL);
554                         last = AVL_PREV(t, last);
555                         ASSERT(last != NULL);
331                         while (lio != mio && lio != fio) {
332                                 ASSERT(lio->io_flags & ZIO_FLAG_OPTIONAL);
333                                 lio = AVL_PREV(t, lio);
334                                 ASSERT(lio != NULL);
```

```
556                 }
557         }
337         }

559         if (first == last)
560                 return (NULL);
339         if (fio != lio) {
340                 uint64_t size = IO_SPAN(fio, lio);
341                 ASSERT(size <= zfs_vdev_aggregation_limit);

562         size = IO_SPAN(first, last);
563         ASSERT3U(size, <=, zfs_vdev_aggregation_limit);

565         aio = zio_vdev_delegated_io(first->io_vd, first->io_offset,
566             zio_buf_alloc(size), size, first->io_type, zio->io_priority,
343                 aio = zio_vdev_delegated_io(fio->io_vd, fio->io_offset,
344                     zio_buf_alloc(size), size, fio->io_type, ZIO_PRIORITY_AGG,
567             flags | ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_QUEUE,
568             vdev_queue_agg_io_done, NULL);
569         aio->io_timestamp = first->io_timestamp;
347                 aio->io_timestamp = fio->io_timestamp;

571         nio = first;
349                 nio = fio;
572         do {
573                 dio = nio;
574                 nio = AVL_NEXT(t, dio);
575                 ASSERT3U(dio->io_type, ==, aio->io_type);
353                         ASSERT(dio->io_type == aio->io_type);
354                         ASSERT(dio->io_vdev_tree == t);

577                 if (dio->io_flags & ZIO_FLAG_NODATA) {
578                         ASSERT3U(dio->io_type, ==, ZIO_TYPE_WRITE);
357                                 ASSERT(dio->io_type == ZIO_TYPE_WRITE);
579                         bzero((char *)aio->io_data + (dio->io_offset -
580                             aio->io_offset), dio->io_size);
581                 } else if (dio->io_type == ZIO_TYPE_WRITE) {
582                         bcopy(dio->io_data, (char *)aio->io_data +
583                             (dio->io_offset - aio->io_offset),
584                             dio->io_size);
585                 }

587                 zio_add_child(dio, aio);
588                 vdev_queue_io_remove(vq, dio);
589                 zio_vdev_io_bypass(dio);
590                 zio_execute(dio);
591         } while (dio != last);
370                 } while (dio != lio);

593         return (aio);
594 }
372                 vdev_queue_pending_add(vq, aio);

596 static zio_t *
597 vdev_queue_io_to_issue(vdev_queue_t *vq)
598 {
599         zio_t *zio, *aio;
600         zio_priority_t p;
601         avl_index_t idx;
602         vdev_queue_class_t *vqc;
603         zio_t search;

605 again:
606         ASSERT(MUTEX_HELD(&vq->vq_lock));

608         p = vdev_queue_class_to_issue(vq);
```



```
610         if (p == ZIO_PRIORITY_NUM_QUEUEABLE) {
611                 /* No eligible queued i/os */
612                 return (NULL);
374                 return (aio);
613         }

615         /*
616          * For LBA-ordered queues (async / scrub), issue the i/o which follows
617          * the most recently issued i/o in LBA (offset) order.
618          *
619          * For FIFO queues (sync), issue the i/o with the lowest timestamp.
620          */
621         vqc = &vq->vq_class[p];
622         search.io_timestamp = 0;
623         search.io_offset = vq->vq_last_offset + 1;
624         VERIFY3P(avl_find(&vqc->vqc_queued_tree, &search, &idx), ==, NULL);
625         zio = avl_nearest(&vqc->vqc_queued_tree, idx, AVL_AFTER);
626         if (zio == NULL)
627                 zio = avl_first(&vqc->vqc_queued_tree);
628         ASSERT3U(zio->io_priority, ==, p);
377         ASSERT(fio->io_vdev_tree == t);
378         vdev_queue_io_remove(vq, fio);

630         aio = vdev_queue_aggregate(vq, zio);
631         if (aio != NULL)
632                 zio = aio;
633         else
634                 vdev_queue_io_remove(vq, zio);

636         /*
637          * If the I/O is or was optional and therefore has no data, we need to
638          * simply discard it. We need to drop the vdev queue's lock to avoid a
639          * deadlock that we could encounter since this I/O will complete
640          * immediately.
641          */
642         if (zio->io_flags & ZIO_FLAG_NODATA) {
386         if (fio->io_flags & ZIO_FLAG_NODATA) {
643                 mutex_exit(&vq->vq_lock);
644                 zio_vdev_io_bypass(zio);
645                 zio_execute(zio);
388                 zio_vdev_io_bypass(fio);
389                 zio_execute(fio);
646                 mutex_enter(&vq->vq_lock);
647                 goto again;
648         }

650         vdev_queue_pending_add(vq, zio);
651         vq->vq_last_offset = zio->io_offset;
394         vdev_queue_pending_add(vq, fio);

653         return (zio);
396         return (fio);
654 }

656 zio_t *
657 vdev_queue_io(zio_t *zio)
658 {
659         vdev_queue_t *vq = &zio->io_vd->vdev_queue;
660         zio_t *nio;

405         ASSERT(zio->io_type == ZIO_TYPE_READ || zio->io_type == ZIO_TYPE_WRITE);

662         if (zio->io_flags & ZIO_FLAG_DONT_QUEUE)
663                 return (zio);
```

```
665         /*
666          * Children i/os inherent their parent's priority, which might
667          * not match the child's i/o type.  Fix it up here.
668          */
669         if (zio->io_type == ZIO_TYPE_READ) {
670                 if (zio->io_priority != ZIO_PRIORITY_SYNC_READ &&
671                     zio->io_priority != ZIO_PRIORITY_ASYNC_READ &&
672                     zio->io_priority != ZIO_PRIORITY_SCRUB)
673                         zio->io_priority = ZIO_PRIORITY_ASYNC_READ;
674         } else {
675                 ASSERT(zio->io_type == ZIO_TYPE_WRITE);
676                 if (zio->io_priority != ZIO_PRIORITY_SYNC_WRITE &&
677                     zio->io_priority != ZIO_PRIORITY_ASYNC_WRITE)
678                         zio->io_priority = ZIO_PRIORITY_ASYNC_WRITE;
679         }

681         zio->io_flags |= ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_QUEUE;

412         if (zio->io_type == ZIO_TYPE_READ)
413                 zio->io_vdev_tree = &vq->vq_read_tree;
414         else
415                 zio->io_vdev_tree = &vq->vq_write_tree;

683         mutex_enter(&vq->vq_lock);

684         zio->io_timestamp = gethrtime();
420         zio->io_deadline = (zio->io_timestamp >> zfs_vdev_time_shift) +
421             zio->io_priority;

685         vdev_queue_io_add(vq, zio);
686         nio = vdev_queue_io_to_issue(vq);

425         nio = vdev_queue_io_to_issue(vq, zfs_vdev_min_pending);

687         mutex_exit(&vq->vq_lock);

689         if (nio == NULL)
690                 return (NULL);

692         if (nio->io_done == vdev_queue_agg_io_done) {
693                 zio_nowait(nio);
694                 return (NULL);
695         }

697         return (nio);
698 }

700 void
701 vdev_queue_io_done(zio_t *zio)
702 {
703         vdev_queue_t *vq = &zio->io_vd->vdev_queue;
704         zio_t *nio;

706         if (zio_injection_enabled)
707                 delay(SEC_TO_TICK(zio_handle_io_delay(zio)));

709         mutex_enter(&vq->vq_lock);

711         vdev_queue_pending_remove(vq, zio);

713         vq->vq_io_complete_ts = gethrtime();

715         while ((nio = vdev_queue_io_to_issue(vq)) != NULL) {
454         for (int i = 0; i < zfs_vdev_ramp_rate; i++) {
455                 zio_t *nio = vdev_queue_io_to_issue(vq, zfs_vdev_max_pending);
456                 if (nio == NULL)
```



```
457                         break;
716                 mutex_exit(&vq->vq_lock);
717                 if (nio->io_done == vdev_queue_agg_io_done) {
718                         zio_nowait(nio);
719                 } else {
720                         zio_vdev_io_reissue(nio);
721                         zio_execute(nio);
722                 }
723                 mutex_enter(&vq->vq_lock);
724         }

726         mutex_exit(&vq->vq_lock);
727 }
______unchanged_portion_omitted_
```

****
64399 Thu Aug 22 16:15:40 2013
new/usr/src/uts/common/fs/zfs/vdev_raidz.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
****
*_____unchanged_portion_omitted_*

```
1907 /*
1908  * Complete an IO operation on a RAIDZ VDev
1909  *
1910  * Outline:
1911  * - For write operations:
1912  *   1. Check for errors on the child IOs.
1913  *   2. Return, setting an error code if too few child VDevs were written
1914  *      to reconstruct the data later.  Note that partial writes are
1915  *      considered successful if they can be reconstructed at all.
1916  * - For read operations:
1917  *   1. Check for errors on the child IOs.
1918  *   2. If data errors occurred:
1919  *      a. Try to reassemble the data from the parity available.
1920  *      b. If we haven't yet read the parity drives, read them now.
1921  *      c. If all parity drives have been read but the data still doesn't
1922  *         reassemble with a correct checksum, then try combinatorial
1923  *         reconstruction.
1924  *      d. If that doesn't work, return an error.
1925  *   3. If there were unexpected errors or this is a resilver operation,
1926  *      rewrite the vdevs that had errors.
1927  */
1928 static void
1929 vdev_raidz_io_done(zio_t *zio)
1930 {
1931         vdev_t *vd = zio->io_vd;
1932         vdev_t *cvd;
1933         raidz_map_t *rm = zio->io_vsd;
1934         raidz_col_t *rc;
1935         int unexpected_errors = 0;
1936         int parity_errors = 0;
1937         int parity_untried = 0;
1938         int data_errors = 0;
1939         int total_errors = 0;
1940         int n, c;
1941         int tgts[VDEV_RAIDZ_MAXPARITY];
1942         int code;

1944         ASSERT(zio->io_bp != NULL);  /* XXX need to add code to enforce this */

1946         ASSERT(rm->rm_missingparity <= rm->rm_firstdatacol);
1947         ASSERT(rm->rm_missingdata <= rm->rm_cols - rm->rm_firstdatacol);

1949         for (c = 0; c < rm->rm_cols; c++) {
1950                 rc = &rm->rm_col[c];

1952                 if (rc->rc_error) {
1953                         ASSERT(rc->rc_error != ECKSUM); /* child has no bp */

1955                         if (c < rm->rm_firstdatacol)
1956                                 parity_errors++;
1957                         else
1958                                 data_errors++;

1960                         if (!rc->rc_skipped)
1961                                 unexpected_errors++;
```



```
1963                         total_errors++;
1964                 } else if (c < rm->rm_firstdatacol && !rc->rc_tried) {
1965                         parity_untried++;
1966                 }
1967         }

1969         if (zio->io_type == ZIO_TYPE_WRITE) {
1970                 /*
1971                  * XXX -- for now, treat partial writes as a success.
1972                  * (If we couldn't write enough columns to reconstruct
1973                  * the data, the I/O failed.  Otherwise, good enough.)
1974                  *
1975                  * Now that we support write reallocation, it would be better
1976                  * to treat partial failure as real failure unless there are
1977                  * no non-degraded top-level vdevs left, and not update DTLs
1978                  * if we intend to reallocate.
1979                  */
1980                 /* XXPOLICY */
1981                 if (total_errors > rm->rm_firstdatacol)
1982                         zio->io_error = vdev_raidz_worst_error(rm);

1984                 return;
1985         }

1987         ASSERT(zio->io_type == ZIO_TYPE_READ);
1988         /*
1989          * There are three potential phases for a read:
1990          *      1. produce valid data from the columns read
1991          *      2. read all disks and try again
1992          *      3. perform combinatorial reconstruction
1993          *
1994          * Each phase is progressively both more expensive and less likely to
1995          * occur. If we encounter more errors than we can repair or all phases
1996          * fail, we have no choice but to return an error.
1997          */

1999         /*
2000          * If the number of errors we saw was correctable -- less than or equal
2001          * to the number of parity disks read -- attempt to produce data that
2002          * has a valid checksum. Naturally, this case applies in the absence of
2003          * any errors.
2004          */
2005         if (total_errors <= rm->rm_firstdatacol - parity_untried) {
2006                 if (data_errors == 0) {
2007                         if (raidz_checksum_verify(zio) == 0) {
2008                                 /*
2009                                  * If we read parity information (unnecessarily
2010                                  * as it happens since no reconstruction was
2011                                  * needed) regenerate and verify the parity.
2012                                  * We also regenerate parity when resilvering
2013                                  * so we can write it out to the failed device
2014                                  * later.
2015                                  */
2016                                 if (parity_errors + parity_untried <
2017                                     rm->rm_firstdatacol ||
2018                                     (zio->io_flags & ZIO_FLAG_RESILVER)) {
2019                                         n = raidz_parity_verify(zio, rm);
2020                                         unexpected_errors += n;
2021                                         ASSERT(parity_errors + n <=
2022                                             rm->rm_firstdatacol);
2023                                 }
2024                                 goto done;
2025                         }
2026                 } else {
2027                         /*
2028                          * We either attempt to read all the parity columns or
```

```
2029                         * none of them. If we didn't try to read parity, we
2030                         * wouldn't be here in the correctable case. There must
2031                         * also have been fewer parity errors than parity
2032                         * columns or, again, we wouldn't be in this code path.
2033                         */
2034                        ASSERT(parity_untried == 0);
2035                        ASSERT(parity_errors < rm->rm_firstdatacol);

2037                        /*
2038                         * Identify the data columns that reported an error.
2039                         */
2040                        n = 0;
2041                        for (c = rm->rm_firstdatacol; c < rm->rm_cols; c++) {
2042                                rc = &rm->rm_col[c];
2043                                if (rc->rc_error != 0) {
2044                                        ASSERT(n < VDEV_RAIDZ_MAXPARITY);
2045                                        tgts[n++] = c;
2046                                }
2047                        }

2049                        ASSERT(rm->rm_firstdatacol >= n);

2051                        code = vdev_raidz_reconstruct(rm, tgts, n);

2053                        if (raidz_checksum_verify(zio) == 0) {
2054                                atomic_inc_64(&raidz_corrected[code]);

2056                                /*
2057                                 * If we read more parity disks than were used
2058                                 * for reconstruction, confirm that the other
2059                                 * parity disks produced correct data. This
2060                                 * routine is suboptimal in that it regenerates
2061                                 * the parity that we already used in addition
2062                                 * to the parity that we're attempting to
2063                                 * verify, but this should be a relatively
2064                                 * uncommon case, and can be optimized if it
2065                                 * becomes a problem. Note that we regenerate
2066                                 * parity when resilvering so we can write it
2067                                 * out to failed devices later.
2068                                 */
2069                                if (parity_errors < rm->rm_firstdatacol - n ||
2070                                    (zio->io_flags & ZIO_FLAG_RESILVER)) {
2071                                        n = raidz_parity_verify(zio, rm);
2072                                        unexpected_errors += n;
2073                                        ASSERT(parity_errors + n <=
2074                                            rm->rm_firstdatacol);
2075                                }

2077                                goto done;
2078                        }
2079                }
2080        }

2082        /*
2083         * This isn't a typical situation -- either we got a read error or
2084         * a child silently returned bad data. Read every block so we can
2085         * try again with as much data and parity as we can track down. If
2086         * we've already been through once before, all children will be marked
2087         * as tried so we'll proceed to combinatorial reconstruction.
2088         */
2089        unexpected_errors = 1;
2090        rm->rm_missingdata = 0;
2091        rm->rm_missingparity = 0;

2093        for (c = 0; c < rm->rm_cols; c++) {
2094                if (rm->rm_col[c].rc_tried)
```



```
2095                        continue;

2097                zio_vdev_io_redone(zio);
2098                do {
2099                        rc = &rm->rm_col[c];
2100                        if (rc->rc_tried)
2101                                continue;
2102                        zio_nowait(zio_vdev_child_io(zio, NULL,
2103                            vd->vdev_child[rc->rc_devidx],
2104                            rc->rc_offset, rc->rc_data, rc->rc_size,
2105                            zio->io_type, zio->io_priority, 0,
2106                            vdev_raidz_child_done, rc));
2107                } while (++c < rm->rm_cols);

2109                return;
2110        }

2112        /*
2113         * At this point we've attempted to reconstruct the data given the
2114         * errors we detected, and we've attempted to read all columns. There
2115         * must, therefore, be one or more additional problems -- silent errors
2116         * resulting in invalid data rather than explicit I/O errors resulting
2117         * in absent data. We check if there is enough additional data to
2118         * possibly reconstruct the data and then perform combinatorial
2119         * reconstruction over all possible combinations. If that fails,
2120         * we're cooked.
2121         */
2122        if (total_errors > rm->rm_firstdatacol) {
2123                zio->io_error = vdev_raidz_worst_error(rm);

2125        } else if (total_errors < rm->rm_firstdatacol &&
2126            (code = vdev_raidz_combrec(zio, total_errors, data_errors)) != 0) {
2127                /*
2128                 * If we didn't use all the available parity for the
2129                 * combinatorial reconstruction, verify that the remaining
2130                 * parity is correct.
2131                 */
2132                if (code != (1 << rm->rm_firstdatacol) - 1)
2133                        (void) raidz_parity_verify(zio, rm);
2134        } else {
2135                /*
2136                 * We're here because either:
2137                 *
2138                 *      total_errors == rm_first_datacol, or
2139                 *      vdev_raidz_combrec() failed
2140                 *
2141                 * In either case, there is enough bad data to prevent
2142                 * reconstruction.
2143                 *
2144                 * Start checksum ereports for all children which haven't
2145                 * failed, and the IO wasn't speculative.
2146                 */
2147                zio->io_error = SET_ERROR(ECKSUM);

2149                if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
2150                        for (c = 0; c < rm->rm_cols; c++) {
2151                                rc = &rm->rm_col[c];
2152                                if (rc->rc_error == 0) {
2153                                        zio_bad_cksum_t zbc;
2154                                        zbc.zbc_has_cksum = 0;
2155                                        zbc.zbc_injected =
2156                                            rm->rm_ecksuminjected;

2158                                        zfs_ereport_start_checksum(
2159                                            zio->io_spa,
2160                                            vd->vdev_child[rc->rc_devidx],
```

```
2161                                            zio, rc->rc_offset, rc->rc_size,
2162                                            (void *)(uintptr_t)c, &zbc);
2163                                }
2164                        }
2165                }
2166        }

2168 done:
2169        zio_checksum_verified(zio);

2171        if (zio->io_error == 0 && spa_writeable(zio->io_spa) &&
2172            (unexpected_errors || (zio->io_flags & ZIO_FLAG_RESILVER))) {
2173                /*
2174                 * Use the good data we have in hand to repair damaged children.
2175                 */
2176                for (c = 0; c < rm->rm_cols; c++) {
2177                        rc = &rm->rm_col[c];
2178                        cvd = vd->vdev_child[rc->rc_devidx];

2180                        if (rc->rc_error == 0)
2181                                continue;

2183                        zio_nowait(zio_vdev_child_io(zio, NULL, cvd,
2184                            rc->rc_offset, rc->rc_data, rc->rc_size,
2185                            ZIO_TYPE_WRITE, ZIO_PRIORITY_ASYNC_WRITE,
2185                            ZIO_TYPE_WRITE, zio->io_priority,
2186                            ZIO_FLAG_IO_REPAIR | (unexpected_errors ?
2187                            ZIO_FLAG_SELF_HEAL : 0), NULL, NULL));
2188                }
2189        }
2190 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  131629 Thu Aug 22 16:15:41 2013
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

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

/* Portions Copyright 2007 Jeremy Teo */
/* Portions Copyright 2010 Robert Milkowski */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/resource.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/kmem.h>
#include <sys/taskq.h>
#include <sys/uio.h>
#include <sys/vmsystm.h>
#include <sys/atomic.h>
#include <sys/vm.h>
#include <vm/seg_vn.h>
#include <vm/pvn.h>
#include <vm/as.h>
#include <vm/kpm.h>
#include <vm/seg_kpm.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/cmn_err.h>
#include <sys/errno.h>
#include <sys/unistd.h>
#include <sys/zfs_dir.h>
#include <sys/zfs_acl.h>
```



```
#include <sys/zfs_ioctl.h>
#include <sys/fs/zfs.h>
#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/dbuf.h>
#include <sys/zap.h>
#include <sys/sa.h>
#include <sys/dirent.h>
#include <sys/policy.h>
#include <sys/sunddi.h>
#include <sys/filio.h>
#include <sys/sid.h>
#include "fs/fs_subr.h"
#include <sys/zfs_ctldir.h>
#include <sys/zfs_fuid.h>
#include <sys/zfs_sa.h>
#include <sys/dnlc.h>
#include <sys/zfs_rlock.h>
#include <sys/extdirent.h>
#include <sys/kidmap.h>
#include <sys/cred.h>
#include <sys/attr.h>

/*
 * Programming rules.
 *
 * Each vnode op performs some logical unit of work.  To do this, the ZPL must
 * properly lock its in-core state, create a DMU transaction, do the work,
 * record this work in the intent log (ZIL), commit the DMU transaction,
 * and wait for the intent log to commit if it is a synchronous operation.
 * Moreover, the vnode ops must work in both normal and log replay context.
 * The ordering of events is important to avoid deadlocks and references
 * to freed memory.  The example below illustrates the following Big Rules:
 *
 *  (1) A check must be made in each zfs thread for a mounted file system.
 *      This is done avoiding races using ZFS_ENTER(zfsvfs).
 *      A ZFS_EXIT(zfsvfs) is needed before all returns.  Any znodes
 *      must be checked with ZFS_VERIFY_ZP(zp).  Both of these macros
 *      can return EIO from the calling function.
 *
 *  (2) VN_RELE() should always be the last thing except for zil_commit()
 *      (if necessary) and ZFS_EXIT(). This is for 3 reasons:
 *      First, if it's the last reference, the vnode/znode
 *      can be freed, so the zp may point to freed memory.  Second, the last
 *      reference will call zfs_zinactive(), which may induce a lot of work --
 *      pushing cached pages (which acquires range locks) and syncing out
 *      cached atime changes.  Third, zfs_zinactive() may require a new tx,
 *      which could deadlock the system if you were already holding one.
 *      If you must call VN_RELE() within a tx then use VN_RELE_ASYNC().
 *
 *  (3) All range locks must be grabbed before calling dmu_tx_assign(),
 *      as they can span dmu_tx_assign() calls.
 *
 *  (4) Always pass TXG_NOWAIT as the second argument to dmu_tx_assign().
 *      This is critical because we don't want to block while holding locks.
 *      Note, in particular, that if a lock is sometimes acquired before
 *      the tx assigns, and sometimes after (e.g. z_lock), then failing to
 *      use a non-blocking assign can deadlock the system.  The scenario:
 *
 *      Thread A has grabbed a lock before calling dmu_tx_assign().
 *      Thread B is in an already-assigned tx, and blocks for this lock.
 *      Thread A calls dmu_tx_assign(TXG_WAIT) and blocks in txg_wait_open()
 *      forever, because the previous txg can't quiesce until B's tx commits.
 *
```

```
 125  *      If dmu_tx_assign() returns ERESTART and zfsvfs->z_assign is TXG_NOWAIT,
 126  *      then drop all locks, call dmu_tx_wait(), and try again.  On subsequent
 127  *      calls to dmu_tx_assign(), pass TXG_WAITED rather than TXG_NOWAIT,
 128  *      to indicate that this operation has already called dmu_tx_wait().
 129  *      This will ensure that we don't retry forever, waiting a short bit
 130  *      each time.
 126  *      then drop all locks, call dmu_tx_wait(), and try again.
 131  *
 132  *  (5) If the operation succeeded, generate the intent log entry for it
 133  *      before dropping locks.  This ensures that the ordering of events
 134  *      in the intent log matches the order in which they actually occurred.
 135  *      During ZIL replay the zfs_log_* functions will update the sequence
 136  *      number to indicate the zil transaction has replayed.
 137  *
 138  *  (6) At the end of each vnode op, the DMU tx must always commit,
 139  *      regardless of whether there were any errors.
 140  *
 141  *  (7) After dropping all locks, invoke zil_commit(zilog, foid)
 142  *      to ensure that synchronous semantics are provided when necessary.
 143  *
 144  * In general, this is how things should be ordered in each vnode op:
 145  *
 146  *      ZFS_ENTER(zfsvfs);              // exit if unmounted
 147  * top:
 148  *      zfs_dirent_lock(&dl, ...)       // lock directory entry (may VN_HOLD())
 149  *      rw_enter(...);                  // grab any other locks you need
 150  *      tx = dmu_tx_create(...);        // get DMU tx
 151  *      dmu_tx_hold_*();                // hold each object you might modify
 152  *      error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
 148  *      error = dmu_tx_assign(tx, TXG_NOWAIT);  // try to assign
 153  *      if (error) {
 154  *              rw_exit(...);           // drop locks
 155  *              zfs_dirent_unlock(dl);  // unlock directory entry
 156  *              VN_RELE(...);           // release held vnodes
 157  *              if (error == ERESTART) {
 158  *                      waited = B_TRUE;
 159  *                      dmu_tx_wait(tx);
 160  *                      dmu_tx_abort(tx);
 161  *                      goto top;
 162  *              }
 163  *              dmu_tx_abort(tx);       // abort DMU tx
 164  *              ZFS_EXIT(zfsvfs);       // finished in zfs
 165  *              return (error);         // really out of space
 166  *      }
 167  *      error = do_real_work();         // do whatever this VOP does
 168  *      if (error == 0)
 169  *              zfs_log_*(...);         // on success, make ZIL entry
 170  *      dmu_tx_commit(tx);              // commit DMU tx -- error or not
 171  *      rw_exit(...);                   // drop locks
 172  *      zfs_dirent_unlock(dl);          // unlock directory entry
 173  *      VN_RELE(...);                   // release held vnodes
 174  *      zil_commit(zilog, foid);        // synchronous when necessary
 175  *      ZFS_EXIT(zfsvfs);               // finished in zfs
 176  *      return (error);                 // done, report error
 177  */

 179 /* ARGSUSED */
 180 static int
 181 zfs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
 182 {
 183         znode_t *zp = VTOZ(*vpp);
 184         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

 186         ZFS_ENTER(zfsvfs);
 187         ZFS_VERIFY_ZP(zp);
```



```
 189         if ((flag & FWRITE) && (zp->z_pflags & ZFS_APPENDONLY) &&
 190             ((flag & FAPPEND) == 0)) {
 191                 ZFS_EXIT(zfsvfs);
 192                 return (SET_ERROR(EPERM));
 193         }

 195         if (!zfs_has_ctldir(zp) && zp->z_zfsvfs->z_vscan &&
 196             ZTOV(zp)->v_type == VREG &&
 197             !(zp->z_pflags & ZFS_AV_QUARANTINED) && zp->z_size > 0) {
 198                 if (fs_vscan(*vpp, cr, 0) != 0) {
 199                         ZFS_EXIT(zfsvfs);
 200                         return (SET_ERROR(EACCES));
 201                 }
 202         }

 204         /* Keep a count of the synchronous opens in the znode */
 205         if (flag & (FSYNC | FDSYNC))
 206                 atomic_inc_32(&zp->z_sync_cnt);

 208         ZFS_EXIT(zfsvfs);
 209         return (0);
 210 }
_____unchanged_portion_omitted_

1280 /*
1281  * Attempt to create a new entry in a directory.  If the entry
1282  * already exists, truncate the file if permissible, else return
1283  * an error.  Return the vp of the created or trunc'd file.
1284  *
1285  *      IN:     dvp     - vnode of directory to put new file entry in.
1286  *              name    - name of new file entry.
1287  *              vap     - attributes of new file.
1288  *              excl    - flag indicating exclusive or non-exclusive mode.
1289  *              mode    - mode to open file with.
1290  *              cr      - credentials of caller.
1291  *              flag    - large file flag [UNUSED].
1292  *              ct      - caller context
1293  *              vsecp   - ACL to be set
1294  *
1295  *      OUT:    vpp     - vnode of created or trunc'd entry.
1296  *
1297  *      RETURN: 0 on success, error code on failure.
1298  *
1299  * Timestamps:
1300  *      dvp - ctime|mtime updated if new entry created
1301  *       vp - ctime|mtime always, atime if new
1302  */

1304 /* ARGSUSED */
1305 static int
1306 zfs_create(vnode_t *dvp, char *name, vattr_t *vap, vcexcl_t excl,
1307     int mode, vnode_t **vpp, cred_t *cr, int flag, caller_context_t *ct,
1308     vsecattr_t *vsecp)
1309 {
1310         znode_t         *zp, *dzp = VTOZ(dvp);
1311         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1312         zilog_t         *zilog;
1313         objset_t        *os;
1314         zfs_dirlock_t   *dl;
1315         dmu_tx_t        *tx;
1316         int             error;
1317         ksid_t          *ksid;
1318         uid_t           uid;
1319         gid_t           gid = crgetgid(cr);
1320         zfs_acl_ids_t   acl_ids;
1321         boolean_t       fuid_dirtied;
```

```
1322         boolean_t       have_acl = B_FALSE;
1323         boolean_t       waited = B_FALSE;

1325         /*
1326          * If we have an ephemeral id, ACL, or XVATTR then
1327          * make sure file system is at proper version
1328          */

1330         ksid = crgetsid(cr, KSID_OWNER);
1331         if (ksid)
1332                 uid = ksid_getid(ksid);
1333         else
1334                 uid = crgetuid(cr);

1336         if (zfsvfs->z_use_fuids == B_FALSE &&
1337             (vsecp || (vap->va_mask & AT_XVATTR) ||
1338             IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1339                 return (SET_ERROR(EINVAL));

1341         ZFS_ENTER(zfsvfs);
1342         ZFS_VERIFY_ZP(dzp);
1343         os = zfsvfs->z_os;
1344         zilog = zfsvfs->z_log;

1346         if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
1347             NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1348                 ZFS_EXIT(zfsvfs);
1349                 return (SET_ERROR(EILSEQ));
1350         }

1352         if (vap->va_mask & AT_XVATTR) {
1353                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
1354                     crgetuid(cr), cr, vap->va_type)) != 0) {
1355                         ZFS_EXIT(zfsvfs);
1356                         return (error);
1357                 }
1358         }
1359 top:
1360         *vpp = NULL;

1362         if ((vap->va_mode & VSVTX) && secpolicy_vnode_stky_modify(cr))
1363                 vap->va_mode &= ~VSVTX;

1365         if (*name == '\0') {
1366                 /*
1367                  * Null component name refers to the directory itself.
1368                  */
1369                 VN_HOLD(dvp);
1370                 zp = dzp;
1371                 dl = NULL;
1372                 error = 0;
1373         } else {
1374                 /* possible VN_HOLD(zp) */
1375                 int zflg = 0;

1377                 if (flag & FIGNORECASE)
1378                         zflg |= ZCILOOK;

1380                 error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1381                     NULL, NULL);
1382                 if (error) {
1383                         if (have_acl)
1384                                 zfs_acl_ids_free(&acl_ids);
1385                         if (strcmp(name, "..") == 0)
1386                                 error = SET_ERROR(EISDIR);
1387                         ZFS_EXIT(zfsvfs);
```



```
1388                         return (error);
1389                 }
1390         }

1392         if (zp == NULL) {
1393                 uint64_t txtype;

1395                 /*
1396                  * Create a new file object and update the directory
1397                  * to reference it.
1398                  */
1399                 if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
1400                         if (have_acl)
1401                                 zfs_acl_ids_free(&acl_ids);
1402                         goto out;
1403                 }

1405                 /*
1406                  * We only support the creation of regular files in
1407                  * extended attribute directories.
1408                  */

1410                 if ((dzp->z_pflags & ZFS_XATTR) &&
1411                     (vap->va_type != VREG)) {
1412                         if (have_acl)
1413                                 zfs_acl_ids_free(&acl_ids);
1414                         error = SET_ERROR(EINVAL);
1415                         goto out;
1416                 }

1418                 if (!have_acl && (error = zfs_acl_ids_create(dzp, 0, vap,
1419                     cr, vsecp, &acl_ids)) != 0)
1420                         goto out;
1421                 have_acl = B_TRUE;

1423                 if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
1424                         zfs_acl_ids_free(&acl_ids);
1425                         error = SET_ERROR(EDQUOT);
1426                         goto out;
1427                 }

1429                 tx = dmu_tx_create(os);

1431                 dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
1432                     ZFS_SA_BASE_ATTR_SIZE);

1434                 fuid_dirtied = zfsvfs->z_fuid_dirty;
1435                 if (fuid_dirtied)
1436                         zfs_fuid_txhold(zfsvfs, tx);
1437                 dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
1438                 dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
1439                 if (!zfsvfs->z_use_sa &&
1440                     acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1441                         dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
1442                             0, acl_ids.z_aclp->z_acl_bytes);
1443                 }
1444                 error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
1438                 error = dmu_tx_assign(tx, TXG_NOWAIT);
1445                 if (error) {
1446                         zfs_dirent_unlock(dl);
1447                         if (error == ERESTART) {
1448                                 waited = B_TRUE;
1449                                 dmu_tx_wait(tx);
1450                                 dmu_tx_abort(tx);
1451                                 goto top;
1452                         }
```

```
1453                        zfs_acl_ids_free(&acl_ids);
1454                        dmu_tx_abort(tx);
1455                        ZFS_EXIT(zfsvfs);
1456                        return (error);
1457                }
1458                zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

1460                if (fuid_dirtied)
1461                        zfs_fuid_sync(zfsvfs, tx);

1463                (void) zfs_link_create(dl, zp, tx, ZNEW);
1464                txtype = zfs_log_create_txtype(Z_FILE, vsecp, vap);
1465                if (flag & FIGNORECASE)
1466                        txtype |= TX_CI;
1467                zfs_log_create(zilog, tx, txtype, dzp, zp, name,
1468                    vsecp, acl_ids.z_fuidp, vap);
1469                zfs_acl_ids_free(&acl_ids);
1470                dmu_tx_commit(tx);
1471        } else {
1472                int aflags = (flag & FAPPEND) ? V_APPEND : 0;

1474                if (have_acl)
1475                        zfs_acl_ids_free(&acl_ids);
1476                have_acl = B_FALSE;

1478                /*
1479                 * A directory entry already exists for this name.
1480                 */
1481                /*
1482                 * Can't truncate an existing file if in exclusive mode.
1483                 */
1484                if (excl == EXCL) {
1485                        error = SET_ERROR(EEXIST);
1486                        goto out;
1487                }
1488                /*
1489                 * Can't open a directory for writing.
1490                 */
1491                if ((ZTOV(zp)->v_type == VDIR) && (mode & S_IWRITE)) {
1492                        error = SET_ERROR(EISDIR);
1493                        goto out;
1494                }
1495                /*
1496                 * Verify requested access to file.
1497                 */
1498                if (mode && (error = zfs_zaccess_rwx(zp, mode, aflags, cr))) {
1499                        goto out;
1500                }

1502                mutex_enter(&dzp->z_lock);
1503                dzp->z_seq++;
1504                mutex_exit(&dzp->z_lock);

1506                /*
1507                 * Truncate regular files if requested.
1508                 */
1509                if ((ZTOV(zp)->v_type == VREG) &&
1510                    (vap->va_mask & AT_SIZE) && (vap->va_size == 0)) {
1511                        /* we can't hold any locks when calling zfs_freesp() */
1512                        zfs_dirent_unlock(dl);
1513                        dl = NULL;
1514                        error = zfs_freesp(zp, 0, 0, mode, TRUE);
1515                        if (error == 0) {
1516                                vnevent_create(ZTOV(zp), ct);
1517                        }
1518                }
```



```
1519        }
1520 out:

1522        if (dl)
1523                zfs_dirent_unlock(dl);

1525        if (error) {
1526                if (zp)
1527                        VN_RELE(ZTOV(zp));
1528        } else {
1529                *vpp = ZTOV(zp);
1530                error = specvp_check(vpp, cr);
1531        }

1533        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1534                zil_commit(zilog, 0);

1536        ZFS_EXIT(zfsvfs);
1537        return (error);
1538 }

1540 /*
1541  * Remove an entry from a directory.
1542  *
1543  *      IN:     dvp     - vnode of directory to remove entry from.
1544  *              name    - name of entry to remove.
1545  *              cr      - credentials of caller.
1546  *              ct      - caller context
1547  *              flags   - case flags
1548  *
1549  *      RETURN: 0 on success, error code on failure.
1550  *
1551  * Timestamps:
1552  *      dvp - ctime|mtime
1553  *       vp - ctime (if nlink > 0)
1554  */

1556 uint64_t null_xattr = 0;

1558 /*ARGSUSED*/
1559 static int
1560 zfs_remove(vnode_t *dvp, char *name, cred_t *cr, caller_context_t *ct,
1561     int flags)
1562 {
1563        znode_t         *zp, *dzp = VTOZ(dvp);
1564        znode_t         *xzp;
1565        vnode_t         *vp;
1566        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1567        zilog_t         *zilog;
1568        uint64_t        acl_obj, xattr_obj;
1569        uint64_t        xattr_obj_unlinked = 0;
1570        uint64_t        obj = 0;
1571        zfs_dirlock_t   *dl;
1572        dmu_tx_t        *tx;
1573        boolean_t       may_delete_now, delete_now = FALSE;
1574        boolean_t       unlinked, toobig = FALSE;
1575        uint64_t        txtype;
1576        pathname_t      *realnmp = NULL;
1577        pathname_t      realnm;
1578        int             error;
1579        int             zflg = ZEXISTS;
1580        boolean_t       waited = B_FALSE;

1582        ZFS_ENTER(zfsvfs);
1583        ZFS_VERIFY_ZP(dzp);
1584        zilog = zfsvfs->z_log;
```

```
1586         if (flags & FIGNORECASE) {
1587                 zflg |= ZCILOOK;
1588                 pn_alloc(&realnm);
1589                 realnmp = &realnm;
1590         }

1592 top:
1593         xattr_obj = 0;
1594         xzp = NULL;
1595         /*
1596          * Attempt to lock directory; fail if entry doesn't exist.
1597          */
1598         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1599             NULL, realnmp)) {
1600                 if (realnmp)
1601                         pn_free(realnmp);
1602                 ZFS_EXIT(zfsvfs);
1603                 return (error);
1604         }

1606         vp = ZTOV(zp);

1608         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
1609                 goto out;
1610         }

1612         /*
1613          * Need to use rmdir for removing directories.
1614          */
1615         if (vp->v_type == VDIR) {
1616                 error = SET_ERROR(EPERM);
1617                 goto out;
1618         }

1620         vnevent_remove(vp, dvp, name, ct);

1622         if (realnmp)
1623                 dnlc_remove(dvp, realnmp->pn_buf);
1624         else
1625                 dnlc_remove(dvp, name);

1627         mutex_enter(&vp->v_lock);
1628         may_delete_now = vp->v_count == 1 && !vn_has_cached_data(vp);
1629         mutex_exit(&vp->v_lock);

1631         /*
1632          * We may delete the znode now, or we may put it in the unlinked set;
1633          * it depends on whether we're the last link, and on whether there are
1634          * other holds on the vnode.  So we dmu_tx_hold() the right things to
1635          * allow for either case.
1636          */
1637         obj = zp->z_id;
1638         tx = dmu_tx_create(zfsvfs->z_os);
1639         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
1640         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1641         zfs_sa_upgrade_txholds(tx, zp);
1642         zfs_sa_upgrade_txholds(tx, dzp);
1643         if (may_delete_now) {
1644                 toobig =
1645                     zp->z_size > zp->z_blksz * DMU_MAX_DELETEBLKCNT;
1646                 /* if the file is too big, only hold_free a token amount */
1647                 dmu_tx_hold_free(tx, zp->z_id, 0,
1648                     (toobig ? DMU_MAX_ACCESS : DMU_OBJECT_END));
1649         }
```



```
1651         /* are there any extended attributes? */
1652         error = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1653             &xattr_obj, sizeof (xattr_obj));
1654         if (error == 0 && xattr_obj) {
1655                 error = zfs_zget(zfsvfs, xattr_obj, &xzp);
1656                 ASSERT0(error);
1657                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
1658                 dmu_tx_hold_sa(tx, xzp->z_sa_hdl, B_FALSE);
1659         }

1661         mutex_enter(&zp->z_lock);
1662         if ((acl_obj = zfs_external_acl(zp)) != 0 && may_delete_now)
1663                 dmu_tx_hold_free(tx, acl_obj, 0, DMU_OBJECT_END);
1664         mutex_exit(&zp->z_lock);

1666         /* charge as an update -- would be nice not to charge at all */
1667         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);

1669         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
1661         error = dmu_tx_assign(tx, TXG_NOWAIT);
1670         if (error) {
1671                 zfs_dirent_unlock(dl);
1672                 VN_RELE(vp);
1673                 if (xzp)
1674                         VN_RELE(ZTOV(xzp));
1675                 if (error == ERESTART) {
1676                         waited = B_TRUE;
1677                         dmu_tx_wait(tx);
1678                         dmu_tx_abort(tx);
1679                         goto top;
1680                 }
1681                 if (realnmp)
1682                         pn_free(realnmp);
1683                 dmu_tx_abort(tx);
1684                 ZFS_EXIT(zfsvfs);
1685                 return (error);
1686         }

1688         /*
1689          * Remove the directory entry.
1690          */
1691         error = zfs_link_destroy(dl, zp, tx, zflg, &unlinked);

1693         if (error) {
1694                 dmu_tx_commit(tx);
1695                 goto out;
1696         }

1698         if (unlinked) {

1700                 /*
1701                  * Hold z_lock so that we can make sure that the ACL obj
1702                  * hasn't changed.  Could have been deleted due to
1703                  * zfs_sa_upgrade().
1704                  */
1705                 mutex_enter(&zp->z_lock);
1706                 mutex_enter(&vp->v_lock);
1707                 (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1708                     &xattr_obj_unlinked, sizeof (xattr_obj_unlinked));
1709                 delete_now = may_delete_now && !toobig &&
1710                     vp->v_count == 1 && !vn_has_cached_data(vp) &&
1711                     xattr_obj == xattr_obj_unlinked && zfs_external_acl(zp) ==
1712                     acl_obj;
1713                 mutex_exit(&vp->v_lock);
1714         }
```

```
1716         if (delete_now) {
1717                 if (xattr_obj_unlinked) {
1718                         ASSERT3U(xzp->z_links, ==, 2);
1719                         mutex_enter(&xzp->z_lock);
1720                         xzp->z_unlinked = 1;
1721                         xzp->z_links = 0;
1722                         error = sa_update(xzp->z_sa_hdl, SA_ZPL_LINKS(zfsvfs),
1723                             &xzp->z_links, sizeof (xzp->z_links), tx);
1724                         ASSERT3U(error,  ==,  0);
1725                         mutex_exit(&xzp->z_lock);
1726                         zfs_unlinked_add(xzp, tx);

1728                         if (zp->z_is_sa)
1729                                 error = sa_remove(zp->z_sa_hdl,
1730                                     SA_ZPL_XATTR(zfsvfs), tx);
1731                         else
1732                                 error = sa_update(zp->z_sa_hdl,
1733                                     SA_ZPL_XATTR(zfsvfs), &null_xattr,
1734                                     sizeof (uint64_t), tx);
1735                         ASSERT0(error);
1736                 }
1737                 mutex_enter(&vp->v_lock);
1738                 vp->v_count--;
1739                 ASSERT0(vp->v_count);
1740                 mutex_exit(&vp->v_lock);
1741                 mutex_exit(&zp->z_lock);
1742                 zfs_znode_delete(zp, tx);
1743         } else if (unlinked) {
1744                 mutex_exit(&zp->z_lock);
1745                 zfs_unlinked_add(zp, tx);
1746         }

1748         txtype = TX_REMOVE;
1749         if (flags & FIGNORECASE)
1750                 txtype |= TX_CI;
1751         zfs_log_remove(zilog, tx, txtype, dzp, name, obj);

1753         dmu_tx_commit(tx);
1754 out:
1755         if (realnmp)
1756                 pn_free(realnmp);

1758         zfs_dirent_unlock(dl);

1760         if (!delete_now)
1761                 VN_RELE(vp);
1762         if (xzp)
1763                 VN_RELE(ZTOV(xzp));

1765         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1766                 zil_commit(zilog, 0);

1768         ZFS_EXIT(zfsvfs);
1769         return (error);
1770 }

1772 /*
1773  * Create a new directory and insert it into dvp using the name
1774  * provided.  Return a pointer to the inserted directory.
1775  *
1776  *      IN:     dvp     - vnode of directory to add subdir to.
1777  *              dirname - name of new directory.
1778  *              vap     - attributes of new directory.
1779  *              cr      - credentials of caller.
1780  *              ct      - caller context
1781  *              flags   - case flags
```



```
1782  *              vsecp   - ACL to be set
1783  *
1784  *      OUT:    vpp     - vnode of created directory.
1785  *
1786  *      RETURN: 0 on success, error code on failure.
1787  *
1788  * Timestamps:
1789  *      dvp - ctime|mtime updated
1790  *       vp - ctime|mtime|atime updated
1791  */
1792 /*ARGSUSED*/
1793 static int
1794 zfs_mkdir(vnode_t *dvp, char *dirname, vattr_t *vap, vnode_t **vpp, cred_t *cr,
1795     caller_context_t *ct, int flags, vsecattr_t *vsecp)
1796 {
1797         znode_t         *zp, *dzp = VTOZ(dvp);
1798         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1799         zilog_t         *zilog;
1800         zfs_dirlock_t   *dl;
1801         uint64_t        txtype;
1802         dmu_tx_t        *tx;
1803         int             error;
1804         int             zf = ZNEW;
1805         ksid_t          *ksid;
1806         uid_t           uid;
1807         gid_t           gid = crgetgid(cr);
1808         zfs_acl_ids_t   acl_ids;
1809         boolean_t       fuid_dirtied;
1810         boolean_t       waited = B_FALSE;

1812         ASSERT(vap->va_type == VDIR);

1814         /*
1815          * If we have an ephemeral id, ACL, or XVATTR then
1816          * make sure file system is at proper version
1817          */

1819         ksid = crgetsid(cr, KSID_OWNER);
1820         if (ksid)
1821                 uid = ksid_getid(ksid);
1822         else
1823                 uid = crgetuid(cr);
1824         if (zfsvfs->z_use_fuids == B_FALSE &&
1825             (vsecp || (vap->va_mask & AT_XVATTR) ||
1826             IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1827                 return (SET_ERROR(EINVAL));

1829         ZFS_ENTER(zfsvfs);
1830         ZFS_VERIFY_ZP(dzp);
1831         zilog = zfsvfs->z_log;

1833         if (dzp->z_pflags & ZFS_XATTR) {
1834                 ZFS_EXIT(zfsvfs);
1835                 return (SET_ERROR(EINVAL));
1836         }

1838         if (zfsvfs->z_utf8 && u8_validate(dirname,
1839             strlen(dirname), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1840                 ZFS_EXIT(zfsvfs);
1841                 return (SET_ERROR(EILSEQ));
1842         }
1843         if (flags & FIGNORECASE)
1844                 zf |= ZCILOOK;

1846         if (vap->va_mask & AT_XVATTR) {
1847                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
```

```
1848                     crgetuid(cr), cr, vap->va_type)) != 0) {
1849                         ZFS_EXIT(zfsvfs);
1850                         return (error);
1851                 }
1852         }
1853
1854         if ((error = zfs_acl_ids_create(dzp, 0, vap, cr,
1855             vsecp, &acl_ids)) != 0) {
1856                 ZFS_EXIT(zfsvfs);
1857                 return (error);
1858         }
1859         /*
1860          * First make sure the new directory doesn't exist.
1861          *
1862          * Existence is checked first to make sure we don't return
1863          * EACCES instead of EEXIST which can cause some applications
1864          * to fail.
1865          */
1866 top:
1867         *vpp = NULL;
1868
1869         if (error = zfs_dirent_lock(&dl, dzp, dirname, &zp, zf,
1870             NULL, NULL)) {
1871                 zfs_acl_ids_free(&acl_ids);
1872                 ZFS_EXIT(zfsvfs);
1873                 return (error);
1874         }
1875
1876         if (error = zfs_zaccess(dzp, ACE_ADD_SUBDIRECTORY, 0, B_FALSE, cr)) {
1877                 zfs_acl_ids_free(&acl_ids);
1878                 zfs_dirent_unlock(dl);
1879                 ZFS_EXIT(zfsvfs);
1880                 return (error);
1881         }
1882
1883         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
1884                 zfs_acl_ids_free(&acl_ids);
1885                 zfs_dirent_unlock(dl);
1886                 ZFS_EXIT(zfsvfs);
1887                 return (SET_ERROR(EDQUOT));
1888         }
1889
1890         /*
1891          * Add a new entry to the directory.
1892          */
1893         tx = dmu_tx_create(zfsvfs->z_os);
1894         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, dirname);
1895         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
1896         fuid_dirtied = zfsvfs->z_fuid_dirty;
1897         if (fuid_dirtied)
1898                 zfs_fuid_txhold(zfsvfs, tx);
1899         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1900                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
1901                     acl_ids.z_aclp->z_acl_bytes);
1902         }
1903
1904         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
1905             ZFS_SA_BASE_ATTR_SIZE);
1906
1907         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
1897         error = dmu_tx_assign(tx, TXG_NOWAIT);
1908         if (error) {
1909                 zfs_dirent_unlock(dl);
1910                 if (error == ERESTART) {
1911                         waited = B_TRUE;
1912                         dmu_tx_wait(tx);
```



```
1913                         dmu_tx_abort(tx);
1914                         goto top;
1915                 }
1916                 zfs_acl_ids_free(&acl_ids);
1917                 dmu_tx_abort(tx);
1918                 ZFS_EXIT(zfsvfs);
1919                 return (error);
1920         }
1921
1922         /*
1923          * Create new node.
1924          */
1925         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);
1926
1927         if (fuid_dirtied)
1928                 zfs_fuid_sync(zfsvfs, tx);
1929
1930         /*
1931          * Now put new name in parent dir.
1932          */
1933         (void) zfs_link_create(dl, zp, tx, ZNEW);
1934
1935         *vpp = ZTOV(zp);
1936
1937         txtype = zfs_log_create_txtype(Z_DIR, vsecp, vap);
1938         if (flags & FIGNORECASE)
1939                 txtype |= TX_CI;
1940         zfs_log_create(zilog, tx, txtype, dzp, zp, dirname, vsecp,
1941             acl_ids.z_fuidp, vap);
1942
1943         zfs_acl_ids_free(&acl_ids);
1944
1945         dmu_tx_commit(tx);
1946
1947         zfs_dirent_unlock(dl);
1948
1949         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1950                 zil_commit(zilog, 0);
1951
1952         ZFS_EXIT(zfsvfs);
1953         return (0);
1954 }
1955
1956 /*
1957  * Remove a directory subdir entry.  If the current working
1958  * directory is the same as the subdir to be removed, the
1959  * remove will fail.
1960  *
1961  *      IN:     dvp     - vnode of directory to remove from.
1962  *              name    - name of directory to be removed.
1963  *              cwd     - vnode of current working directory.
1964  *              cr      - credentials of caller.
1965  *              ct      - caller context
1966  *              flags   - case flags
1967  *
1968  *      RETURN: 0 on success, error code on failure.
1969  *
1970  * Timestamps:
1971  *      dvp - ctime|mtime updated
1972  */
1973 /*ARGSUSED*/
1974 static int
1975 zfs_rmdir(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
1976     caller_context_t *ct, int flags)
1977 {
1978         znode_t         *dzp = VTOZ(dvp);
```

```
1979         znode_t         *zp;
1980         vnode_t         *vp;
1981         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1982         zilog_t         *zilog;
1983         zfs_dirlock_t   *dl;
1984         dmu_tx_t        *tx;
1985         int             error;
1986         int             zflg = ZEXISTS;
1987         boolean_t       waited = B_FALSE;

1989         ZFS_ENTER(zfsvfs);
1990         ZFS_VERIFY_ZP(dzp);
1991         zilog = zfsvfs->z_log;

1993         if (flags & FIGNORECASE)
1994                 zflg |= ZCILOOK;
1995 top:
1996         zp = NULL;

1998         /*
1999          * Attempt to lock directory; fail if entry doesn't exist.
2000          */
2001         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
2002             NULL, NULL)) {
2003                 ZFS_EXIT(zfsvfs);
2004                 return (error);
2005         }

2007         vp = ZTOV(zp);

2009         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
2010                 goto out;
2011         }

2013         if (vp->v_type != VDIR) {
2014                 error = SET_ERROR(ENOTDIR);
2015                 goto out;
2016         }

2018         if (vp == cwd) {
2019                 error = SET_ERROR(EINVAL);
2020                 goto out;
2021         }

2023         vnevent_rmdir(vp, dvp, name, ct);

2025         /*
2026          * Grab a lock on the directory to make sure that noone is
2027          * trying to add (or lookup) entries while we are removing it.
2028          */
2029         rw_enter(&zp->z_name_lock, RW_WRITER);

2031         /*
2032          * Grab a lock on the parent pointer to make sure we play well
2033          * with the treewalk and directory rename code.
2034          */
2035         rw_enter(&zp->z_parent_lock, RW_WRITER);

2037         tx = dmu_tx_create(zfsvfs->z_os);
2038         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
2039         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
2040         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
2041         zfs_sa_upgrade_txholds(tx, zp);
2042         zfs_sa_upgrade_txholds(tx, dzp);
2043         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
2031         error = dmu_tx_assign(tx, TXG_NOWAIT);
```



```
2044         if (error) {
2045                 rw_exit(&zp->z_parent_lock);
2046                 rw_exit(&zp->z_name_lock);
2047                 zfs_dirent_unlock(dl);
2048                 VN_RELE(vp);
2049                 if (error == ERESTART) {
2050                         waited = B_TRUE;
2051                         dmu_tx_wait(tx);
2052                         dmu_tx_abort(tx);
2053                         goto top;
2054                 }
2055                 dmu_tx_abort(tx);
2056                 ZFS_EXIT(zfsvfs);
2057                 return (error);
2058         }

2060         error = zfs_link_destroy(dl, zp, tx, zflg, NULL);

2062         if (error == 0) {
2063                 uint64_t txtype = TX_RMDIR;
2064                 if (flags & FIGNORECASE)
2065                         txtype |= TX_CI;
2066                 zfs_log_remove(zilog, tx, txtype, dzp, name, ZFS_NO_OBJECT);
2067         }

2069         dmu_tx_commit(tx);

2071         rw_exit(&zp->z_parent_lock);
2072         rw_exit(&zp->z_name_lock);
2073 out:
2074         zfs_dirent_unlock(dl);

2076         VN_RELE(vp);

2078         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
2079                 zil_commit(zilog, 0);

2081         ZFS_EXIT(zfsvfs);
2082         return (error);
2083 }
_____unchanged_portion_omitted_

3345 /*
3346  * Move an entry from the provided source directory to the target
3347  * directory.  Change the entry name as indicated.
3348  *
3349  *      IN:     sdvp    - Source directory containing the "old entry".
3350  *              snm     - Old entry name.
3351  *              tdvp    - Target directory to contain the "new entry".
3352  *              tnm     - New entry name.
3353  *              cr      - credentials of caller.
3354  *              ct      - caller context
3355  *              flags   - case flags
3356  *
3357  *      RETURN: 0 on success, error code on failure.
3358  *
3359  * Timestamps:
3360  *      sdvp,tdvp - ctime|mtime updated
3361  */
3362 /*ARGSUSED*/
3363 static int
3364 zfs_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm, cred_t *cr,
3365     caller_context_t *ct, int flags)
3366 {
3367         znode_t         *tdzp, *szp, *tzp;
3368         znode_t         *sdzp = VTOZ(sdvp);
```

```
3369         zfsvfs_t        *zfsvfs = sdzp->z_zfsvfs;
3370         zilog_t         *zilog;
3371         vnode_t         *realvp;
3372         zfs_dirlock_t   *sdl, *tdl;
3373         dmu_tx_t        *tx;
3374         zfs_zlock_t     *zl;
3375         int             cmp, serr, terr;
3376         int             error = 0;
3377         int             zflg = 0;
3378         boolean_t       waited = B_FALSE;

3380         ZFS_ENTER(zfsvfs);
3381         ZFS_VERIFY_ZP(sdzp);
3382         zilog = zfsvfs->z_log;

3384         /*
3385          * Make sure we have the real vp for the target directory.
3386          */
3387         if (VOP_REALVP(tdvp, &realvp, ct) == 0)
3388                 tdvp = realvp;

3390         tdzp = VTOZ(tdvp);
3391         ZFS_VERIFY_ZP(tdzp);

3393         /*
3394          * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
3395          * ctldir appear to have the same v_vfsp.
3396          */
3397         if (tdzp->z_zfsvfs != zfsvfs || zfsctl_is_node(tdvp)) {
3398                 ZFS_EXIT(zfsvfs);
3399                 return (SET_ERROR(EXDEV));
3400         }

3402         if (zfsvfs->z_utf8 && u8_validate(tnm,
3403             strlen(tnm), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3404                 ZFS_EXIT(zfsvfs);
3405                 return (SET_ERROR(EILSEQ));
3406         }

3408         if (flags & FIGNORECASE)
3409                 zflg |= ZCILOOK;

3411 top:
3412         szp = NULL;
3413         tzp = NULL;
3414         zl = NULL;

3416         /*
3417          * This is to prevent the creation of links into attribute space
3418          * by renaming a linked file into/outof an attribute directory.
3419          * See the comment in zfs_link() for why this is considered bad.
3420          */
3421         if ((tdzp->z_pflags & ZFS_XATTR) != (sdzp->z_pflags & ZFS_XATTR)) {
3422                 ZFS_EXIT(zfsvfs);
3423                 return (SET_ERROR(EINVAL));
3424         }

3426         /*
3427          * Lock source and target directory entries.  To prevent deadlock,
3428          * a lock ordering must be defined.  We lock the directory with
3429          * the smallest object id first, or if it's a tie, the one with
3430          * the lexically first name.
3431          */
3432         if (sdzp->z_id < tdzp->z_id) {
3433                 cmp = -1;
3434         } else if (sdzp->z_id > tdzp->z_id) {
```



```
3435                 cmp = 1;
3436         } else {
3437                 /*
3438                  * First compare the two name arguments without
3439                  * considering any case folding.
3440                  */
3441                 int nofold = (zfsvfs->z_norm & ~U8_TEXTPREP_TOUPPER);

3443                 cmp = u8_strcmp(snm, tnm, 0, nofold, U8_UNICODE_LATEST, &error);
3444                 ASSERT(error == 0 || !zfsvfs->z_utf8);
3445                 if (cmp == 0) {
3446                         /*
3447                          * POSIX: "If the old argument and the new argument
3448                          * both refer to links to the same existing file,
3449                          * the rename() function shall return successfully
3450                          * and perform no other action."
3451                          */
3452                         ZFS_EXIT(zfsvfs);
3453                         return (0);
3454                 }
3455                 /*
3456                  * If the file system is case-folding, then we may
3457                  * have some more checking to do.  A case-folding file
3458                  * system is either supporting mixed case sensitivity
3459                  * access or is completely case-insensitive.  Note
3460                  * that the file system is always case preserving.
3461                  *
3462                  * In mixed sensitivity mode case sensitive behavior
3463                  * is the default.  FIGNORECASE must be used to
3464                  * explicitly request case insensitive behavior.
3465                  *
3466                  * If the source and target names provided differ only
3467                  * by case (e.g., a request to rename 'tim' to 'Tim'),
3468                  * we will treat this as a special case in the
3469                  * case-insensitive mode: as long as the source name
3470                  * is an exact match, we will allow this to proceed as
3471                  * a name-change request.
3472                  */
3473                 if ((zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
3474                     (zfsvfs->z_case == ZFS_CASE_MIXED &&
3475                     flags & FIGNORECASE)) &&
3476                     u8_strcmp(snm, tnm, 0, zfsvfs->z_norm, U8_UNICODE_LATEST,
3477                     &error) == 0) {
3478                         /*
3479                          * case preserving rename request, require exact
3480                          * name matches
3481                          */
3482                         zflg |= ZCIEXACT;
3483                         zflg &= ~ZCILOOK;
3484                 }
3485         }

3487         /*
3488          * If the source and destination directories are the same, we should
3489          * grab the z_name_lock of that directory only once.
3490          */
3491         if (sdzp == tdzp) {
3492                 zflg |= ZHAVELOCK;
3493                 rw_enter(&sdzp->z_name_lock, RW_READER);
3494         }

3496         if (cmp < 0) {
3497                 serr = zfs_dirent_lock(&sdl, sdzp, snm, &szp,
3498                     ZEXISTS | zflg, NULL, NULL);
3499                 terr = zfs_dirent_lock(&tdl,
3500                     tdzp, tnm, &tzp, ZRENAMING | zflg, NULL, NULL);
```

```
3501         } else {
3502                 terr = zfs_dirent_lock(&tdl,
3503                     tdzp, tnm, &tzp, zflg, NULL, NULL);
3504                 serr = zfs_dirent_lock(&sdl,
3505                     sdzp, snm, &szp, ZEXISTS | ZRENAMING | zflg,
3506                     NULL, NULL);
3507         }

3509         if (serr) {
3510                 /*
3511                  * Source entry invalid or not there.
3512                  */
3513                 if (!terr) {
3514                         zfs_dirent_unlock(tdl);
3515                         if (tzp)
3516                                 VN_RELE(ZTOV(tzp));
3517                 }

3519                 if (sdzp == tdzp)
3520                         rw_exit(&sdzp->z_name_lock);

3522                 if (strcmp(snm, "..") == 0)
3523                         serr = SET_ERROR(EINVAL);
3524                 ZFS_EXIT(zfsvfs);
3525                 return (serr);
3526         }
3527         if (terr) {
3528                 zfs_dirent_unlock(sdl);
3529                 VN_RELE(ZTOV(szp));

3531                 if (sdzp == tdzp)
3532                         rw_exit(&sdzp->z_name_lock);

3534                 if (strcmp(tnm, "..") == 0)
3535                         terr = SET_ERROR(EINVAL);
3536                 ZFS_EXIT(zfsvfs);
3537                 return (terr);
3538         }

3540         /*
3541          * Must have write access at the source to remove the old entry
3542          * and write access at the target to create the new entry.
3543          * Note that if target and source are the same, this can be
3544          * done in a single check.
3545          */

3547         if (error = zfs_zaccess_rename(sdzp, szp, tdzp, tzp, cr))
3548                 goto out;

3550         if (ZTOV(szp)->v_type == VDIR) {
3551                 /*
3552                  * Check to make sure rename is valid.
3553                  * Can't do a move like this: /usr/a/b to /usr/a/b/c/d
3554                  */
3555                 if (error = zfs_rename_lock(szp, tdzp, sdzp, &zl))
3556                         goto out;
3557         }

3559         /*
3560          * Does target exist?
3561          */
3562         if (tzp) {
3563                 /*
3564                  * Source and target must be the same type.
3565                  */
3566                 if (ZTOV(szp)->v_type == VDIR) {
```



```
3567                         if (ZTOV(tzp)->v_type != VDIR) {
3568                                 error = SET_ERROR(ENOTDIR);
3569                                 goto out;
3570                         }
3571                 } else {
3572                         if (ZTOV(tzp)->v_type == VDIR) {
3573                                 error = SET_ERROR(EISDIR);
3574                                 goto out;
3575                         }
3576                 }
3577                 /*
3578                  * POSIX dictates that when the source and target
3579                  * entries refer to the same file object, rename
3580                  * must do nothing and exit without error.
3581                  */
3582                 if (szp->z_id == tzp->z_id) {
3583                         error = 0;
3584                         goto out;
3585                 }
3586         }

3588         vnevent_rename_src(ZTOV(szp), sdvp, snm, ct);
3589         if (tzp)
3590                 vnevent_rename_dest(ZTOV(tzp), tdvp, tnm, ct);

3592         /*
3593          * notify the target directory if it is not the same
3594          * as source directory.
3595          */
3596         if (tdvp != sdvp) {
3597                 vnevent_rename_dest_dir(tdvp, ct);
3598         }

3600         tx = dmu_tx_create(zfsvfs->z_os);
3601         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3602         dmu_tx_hold_sa(tx, sdzp->z_sa_hdl, B_FALSE);
3603         dmu_tx_hold_zap(tx, sdzp->z_id, FALSE, snm);
3604         dmu_tx_hold_zap(tx, tdzp->z_id, TRUE, tnm);
3605         if (sdzp != tdzp) {
3606                 dmu_tx_hold_sa(tx, tdzp->z_sa_hdl, B_FALSE);
3607                 zfs_sa_upgrade_txholds(tx, tdzp);
3608         }
3609         if (tzp) {
3610                 dmu_tx_hold_sa(tx, tzp->z_sa_hdl, B_FALSE);
3611                 zfs_sa_upgrade_txholds(tx, tzp);
3612         }

3614         zfs_sa_upgrade_txholds(tx, szp);
3615         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
3616         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3602         error = dmu_tx_assign(tx, TXG_NOWAIT);
3617         if (error) {
3618                 if (zl != NULL)
3619                         zfs_rename_unlock(&zl);
3620                 zfs_dirent_unlock(sdl);
3621                 zfs_dirent_unlock(tdl);

3623                 if (sdzp == tdzp)
3624                         rw_exit(&sdzp->z_name_lock);

3626                 VN_RELE(ZTOV(szp));
3627                 if (tzp)
3628                         VN_RELE(ZTOV(tzp));
3629                 if (error == ERESTART) {
3630                         waited = B_TRUE;
3631                         dmu_tx_wait(tx);
```

```
3632                        dmu_tx_abort(tx);
3633                        goto top;
3634                }
3635                dmu_tx_abort(tx);
3636                ZFS_EXIT(zfsvfs);
3637                return (error);
3638        }

3640        if (tzp)        /* Attempt to remove the existing target */
3641                error = zfs_link_destroy(tdl, tzp, tx, zflg, NULL);

3643        if (error == 0) {
3644                error = zfs_link_create(tdl, szp, tx, ZRENAMING);
3645                if (error == 0) {
3646                        szp->z_pflags |= ZFS_AV_MODIFIED;

3648                        error = sa_update(szp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
3649                            (void *)&szp->z_pflags, sizeof (uint64_t), tx);
3650                        ASSERT0(error);

3652                        error = zfs_link_destroy(sdl, szp, tx, ZRENAMING, NULL);
3653                        if (error == 0) {
3654                                zfs_log_rename(zilog, tx, TX_RENAME |
3655                                    (flags & FIGNORECASE ? TX_CI : 0), sdzp,
3656                                    sdl->dl_name, tdzp, tdl->dl_name, szp);

3658                                /*
3659                                 * Update path information for the target vnode
3660                                 */
3661                                vn_renamepath(tdvp, ZTOV(szp), tnm,
3662                                    strlen(tnm));
3663                        } else {
3664                                /*
3665                                 * At this point, we have successfully created
3666                                 * the target name, but have failed to remove
3667                                 * the source name.  Since the create was done
3668                                 * with the ZRENAMING flag, there are
3669                                 * complications; for one, the link count is
3670                                 * wrong.  The easiest way to deal with this
3671                                 * is to remove the newly created target, and
3672                                 * return the original error.  This must
3673                                 * succeed; fortunately, it is very unlikely to
3674                                 * fail, since we just created it.
3675                                 */
3676                                VERIFY3U(zfs_link_destroy(tdl, szp, tx,
3677                                    ZRENAMING, NULL), ==, 0);
3678                        }
3679                }
3680        }

3682        dmu_tx_commit(tx);
3683 out:
3684        if (zl != NULL)
3685                zfs_rename_unlock(&zl);

3687        zfs_dirent_unlock(sdl);
3688        zfs_dirent_unlock(tdl);

3690        if (sdzp == tdzp)
3691                rw_exit(&sdzp->z_name_lock);


3694        VN_RELE(ZTOV(szp));
3695        if (tzp)
3696                VN_RELE(ZTOV(tzp));
```



```
3698        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3699                zil_commit(zilog, 0);

3701        ZFS_EXIT(zfsvfs);
3702        return (error);
3703 }

3705 /*
3706  * Insert the indicated symbolic reference entry into the directory.
3707  *
3708  *      IN:     dvp     - Directory to contain new symbolic link.
3709  *              link    - Name for new symlink entry.
3710  *              vap     - Attributes of new entry.
3711  *              cr      - credentials of caller.
3712  *              ct      - caller context
3713  *              flags   - case flags
3714  *
3715  *      RETURN: 0 on success, error code on failure.
3716  *
3717  * Timestamps:
3718  *      dvp - ctime|mtime updated
3719  */
3720 /*ARGSUSED*/
3721 static int
3722 zfs_symlink(vnode_t *dvp, char *name, vattr_t *vap, char *link, cred_t *cr,
3723     caller_context_t *ct, int flags)
3724 {
3725        znode_t         *zp, *dzp = VTOZ(dvp);
3726        zfs_dirlock_t   *dl;
3727        dmu_tx_t        *tx;
3728        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3729        zilog_t         *zilog;
3730        uint64_t        len = strlen(link);
3731        int             error;
3732        int             zflg = ZNEW;
3733        zfs_acl_ids_t   acl_ids;
3734        boolean_t       fuid_dirtied;
3735        uint64_t        txtype = TX_SYMLINK;
3736        boolean_t       waited = B_FALSE;

3738        ASSERT(vap->va_type == VLNK);

3740        ZFS_ENTER(zfsvfs);
3741        ZFS_VERIFY_ZP(dzp);
3742        zilog = zfsvfs->z_log;

3744        if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
3745            NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3746                ZFS_EXIT(zfsvfs);
3747                return (SET_ERROR(EILSEQ));
3748        }
3749        if (flags & FIGNORECASE)
3750                zflg |= ZCILOOK;

3752        if (len > MAXPATHLEN) {
3753                ZFS_EXIT(zfsvfs);
3754                return (SET_ERROR(ENAMETOOLONG));
3755        }

3757        if ((error = zfs_acl_ids_create(dzp, 0,
3758            vap, cr, NULL, &acl_ids)) != 0) {
3759                ZFS_EXIT(zfsvfs);
3760                return (error);
3761        }
3762 top:
3763        /*
```

```
3764          * Attempt to lock directory; fail if entry already exists.
3765          */
3766         error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg, NULL, NULL);
3767         if (error) {
3768                 zfs_acl_ids_free(&acl_ids);
3769                 ZFS_EXIT(zfsvfs);
3770                 return (error);
3771         }

3773         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
3774                 zfs_acl_ids_free(&acl_ids);
3775                 zfs_dirent_unlock(dl);
3776                 ZFS_EXIT(zfsvfs);
3777                 return (error);
3778         }

3780         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
3781                 zfs_acl_ids_free(&acl_ids);
3782                 zfs_dirent_unlock(dl);
3783                 ZFS_EXIT(zfsvfs);
3784                 return (SET_ERROR(EDQUOT));
3785         }
3786         tx = dmu_tx_create(zfsvfs->z_os);
3787         fuid_dirtied = zfsvfs->z_fuid_dirty;
3788         dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0, MAX(1, len));
3789         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
3790         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
3791             ZFS_SA_BASE_ATTR_SIZE + len);
3792         dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
3793         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3794                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
3795                     acl_ids.z_aclp->z_acl_bytes);
3796         }
3797         if (fuid_dirtied)
3798                 zfs_fuid_txhold(zfsvfs, tx);
3799         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3783         error = dmu_tx_assign(tx, TXG_NOWAIT);
3800         if (error) {
3801                 zfs_dirent_unlock(dl);
3802                 if (error == ERESTART) {
3803                         waited = B_TRUE;
3804                         dmu_tx_wait(tx);
3805                         dmu_tx_abort(tx);
3806                         goto top;
3807                 }
3808                 zfs_acl_ids_free(&acl_ids);
3809                 dmu_tx_abort(tx);
3810                 ZFS_EXIT(zfsvfs);
3811                 return (error);
3812         }

3814         /*
3815          * Create a new object for the symlink.
3816          * for version 4 ZPL datsets the symlink will be an SA attribute
3817          */
3818         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

3820         if (fuid_dirtied)
3821                 zfs_fuid_sync(zfsvfs, tx);

3823         mutex_enter(&zp->z_lock);
3824         if (zp->z_is_sa)
3825                 error = sa_update(zp->z_sa_hdl, SA_ZPL_SYMLINK(zfsvfs),
3826                     link, len, tx);
3827         else
3828                 zfs_sa_symlink(zp, link, len, tx);
```



```
3829         mutex_exit(&zp->z_lock);

3831         zp->z_size = len;
3832         (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
3833             &zp->z_size, sizeof (zp->z_size), tx);
3834         /*
3835          * Insert the new object into the directory.
3836          */
3837         (void) zfs_link_create(dl, zp, tx, ZNEW);

3839         if (flags & FIGNORECASE)
3840                 txtype |= TX_CI;
3841         zfs_log_symlink(zilog, tx, txtype, dzp, zp, name, link);

3843         zfs_acl_ids_free(&acl_ids);

3845         dmu_tx_commit(tx);

3847         zfs_dirent_unlock(dl);

3849         VN_RELE(ZTOV(zp));

3851         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3852                 zil_commit(zilog, 0);

3854         ZFS_EXIT(zfsvfs);
3855         return (error);
3856 }
_____unchanged_portion_omitted_

3899 /*
3900  * Insert a new entry into directory tdvp referencing svp.
3901  *
3902  *      IN:     tdvp    - Directory to contain new entry.
3903  *              svp     - vnode of new entry.
3904  *              name    - name of new entry.
3905  *              cr      - credentials of caller.
3906  *              ct      - caller context
3907  *
3908  *      RETURN: 0 on success, error code on failure.
3909  *
3910  * Timestamps:
3911  *      tdvp - ctime|mtime updated
3912  *       svp - ctime updated
3913  */
3914 /* ARGSUSED */
3915 static int
3916 zfs_link(vnode_t *tdvp, vnode_t *svp, char *name, cred_t *cr,
3917     caller_context_t *ct, int flags)
3918 {
3919         znode_t         *dzp = VTOZ(tdvp);
3920         znode_t         *tzp, *szp;
3921         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3922         zilog_t         *zilog;
3923         zfs_dirlock_t   *dl;
3924         dmu_tx_t        *tx;
3925         vnode_t         *realvp;
3926         int             error;
3927         int             zf = ZNEW;
3928         uint64_t        parent;
3929         uid_t           owner;
3930         boolean_t       waited = B_FALSE;

3932         ASSERT(tdvp->v_type == VDIR);

3934         ZFS_ENTER(zfsvfs);
```

```
3935         ZFS_VERIFY_ZP(dzp);
3936         zilog = zfsvfs->z_log;

3938         if (VOP_REALVP(svp, &realvp, ct) == 0)
3939                 svp = realvp;

3941         /*
3942          * POSIX dictates that we return EPERM here.
3943          * Better choices include ENOTSUP or EISDIR.
3944          */
3945         if (svp->v_type == VDIR) {
3946                 ZFS_EXIT(zfsvfs);
3947                 return (SET_ERROR(EPERM));
3948         }

3950         szp = VTOZ(svp);
3951         ZFS_VERIFY_ZP(szp);

3953         /*
3954          * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
3955          * ctldir appear to have the same v_vfsp.
3956          */
3957         if (szp->z_zfsvfs != zfsvfs || zfsctl_is_node(svp)) {
3958                 ZFS_EXIT(zfsvfs);
3959                 return (SET_ERROR(EXDEV));
3960         }

3962         /* Prevent links to .zfs/shares files */

3964         if ((error = sa_lookup(szp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
3965             &parent, sizeof (uint64_t))) != 0) {
3966                 ZFS_EXIT(zfsvfs);
3967                 return (error);
3968         }
3969         if (parent == zfsvfs->z_shares_dir) {
3970                 ZFS_EXIT(zfsvfs);
3971                 return (SET_ERROR(EPERM));
3972         }

3974         if (zfsvfs->z_utf8 && u8_validate(name,
3975             strlen(name), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3976                 ZFS_EXIT(zfsvfs);
3977                 return (SET_ERROR(EILSEQ));
3978         }
3979         if (flags & FIGNORECASE)
3980                 zf |= ZCILOOK;

3982         /*
3983          * We do not support links between attributes and non-attributes
3984          * because of the potential security risk of creating links
3985          * into "normal" file space in order to circumvent restrictions
3986          * imposed in attribute space.
3987          */
3988         if ((szp->z_pflags & ZFS_XATTR) != (dzp->z_pflags & ZFS_XATTR)) {
3989                 ZFS_EXIT(zfsvfs);
3990                 return (SET_ERROR(EINVAL));
3991         }


3994         owner = zfs_fuid_map_id(zfsvfs, szp->z_uid, cr, ZFS_OWNER);
3995         if (owner != crgetuid(cr) && secpolicy_basic_link(cr) != 0) {
3996                 ZFS_EXIT(zfsvfs);
3997                 return (SET_ERROR(EPERM));
3998         }

4000         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
```



```
4001                 ZFS_EXIT(zfsvfs);
4002                 return (error);
4003         }

4005 top:
4006         /*
4007          * Attempt to lock directory; fail if entry already exists.
4008          */
4009         error = zfs_dirent_lock(&dl, dzp, name, &tzp, zf, NULL, NULL);
4010         if (error) {
4011                 ZFS_EXIT(zfsvfs);
4012                 return (error);
4013         }

4015         tx = dmu_tx_create(zfsvfs->z_os);
4016         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
4017         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
4018         zfs_sa_upgrade_txholds(tx, szp);
4019         zfs_sa_upgrade_txholds(tx, dzp);
4020         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
4002         error = dmu_tx_assign(tx, TXG_NOWAIT);
4021         if (error) {
4022                 zfs_dirent_unlock(dl);
4023                 if (error == ERESTART) {
4024                         waited = B_TRUE;
4025                         dmu_tx_wait(tx);
4026                         dmu_tx_abort(tx);
4027                         goto top;
4028                 }
4029                 dmu_tx_abort(tx);
4030                 ZFS_EXIT(zfsvfs);
4031                 return (error);
4032         }

4034         error = zfs_link_create(dl, szp, tx, 0);

4036         if (error == 0) {
4037                 uint64_t txtype = TX_LINK;
4038                 if (flags & FIGNORECASE)
4039                         txtype |= TX_CI;
4040                 zfs_log_link(zilog, tx, txtype, dzp, szp, name);
4041         }

4043         dmu_tx_commit(tx);

4045         zfs_dirent_unlock(dl);

4047         if (error == 0) {
4048                 vnevent_link(svp, ct);
4049         }

4051         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4052                 zil_commit(zilog, 0);

4054         ZFS_EXIT(zfsvfs);
4055         return (error);
4056 }
```

*_____unchanged_portion_omitted_*

```
*****************************************************
   57658 Thu Aug 22 16:15:43 2013
new/usr/src/uts/common/fs/zfs/zil.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
*****************************************************
______unchanged_portion_omitted_

 856 /*
 857  * Initialize the io for a log block.
 858  */
 859 static void
 860 zil_lwb_write_init(zilog_t *zilog, lwb_t *lwb)
 861 {
 862         zbookmark_t zb;

 864         SET_BOOKMARK(&zb, lwb->lwb_blk.blk_cksum.zc_word[ZIL_ZC_OBJSET],
 865             ZB_ZIL_OBJECT, ZB_ZIL_LEVEL,
 866             lwb->lwb_blk.blk_cksum.zc_word[ZIL_ZC_SEQ]);

 868         if (zilog->zl_root_zio == NULL) {
 869                 zilog->zl_root_zio = zio_root(zilog->zl_spa, NULL, NULL,
 870                     ZIO_FLAG_CANFAIL);
 871         }
 872         if (lwb->lwb_zio == NULL) {
 873                 lwb->lwb_zio = zio_rewrite(zilog->zl_root_zio, zilog->zl_spa,
 874                     0, &lwb->lwb_blk, lwb->lwb_buf, BP_GET_LSIZE(&lwb->lwb_blk),
 875                     zil_lwb_write_done, lwb, ZIO_PRIORITY_SYNC_WRITE,
 875                     zil_lwb_write_done, lwb, ZIO_PRIORITY_LOG_WRITE,
 876                     ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE, &zb);
 877         }
 878 }
______unchanged_portion_omitted_
```

```
**********************************************************
   90288 Thu Aug 22 16:15:44 2013
new/usr/src/uts/common/fs/zfs/zio.c
4045 zfs write throttle & i/o scheduler performance work
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/spa_impl.h>
#include <sys/vdev_impl.h>
#include <sys/zio_impl.h>
#include <sys/zio_compress.h>
#include <sys/zio_checksum.h>
#include <sys/dmu_objset.h>
#include <sys/arc.h>
#include <sys/ddt.h>

/*
 * ==========================================================================
 * I/O priority table
 * ==========================================================================
 */
uint8_t zio_priority_table[ZIO_PRIORITY_TABLE_SIZE] = {
        0,      /* ZIO_PRIORITY_NOW             */
        0,      /* ZIO_PRIORITY_SYNC_READ       */
        0,      /* ZIO_PRIORITY_SYNC_WRITE      */
        0,      /* ZIO_PRIORITY_LOG_WRITE       */
        1,      /* ZIO_PRIORITY_CACHE_FILL      */
        1,      /* ZIO_PRIORITY_AGG             */
        4,      /* ZIO_PRIORITY_FREE            */
        4,      /* ZIO_PRIORITY_ASYNC_WRITE     */
        6,      /* ZIO_PRIORITY_ASYNC_READ      */
        10,     /* ZIO_PRIORITY_RESILVER        */
        20,     /* ZIO_PRIORITY_SCRUB           */
        2,      /* ZIO_PRIORITY_DDT_PREFETCH    */
};
```



```
/*
 * ==========================================================================
 * I/O type descriptions
 * ==========================================================================
 */
const char *zio_type_name[ZIO_TYPES] = {
char *zio_type_name[ZIO_TYPES] = {
        "zio_null", "zio_read", "zio_write", "zio_free", "zio_claim",
        "zio_ioctl"
};
_____unchanged_portion_omitted_

static void
zio_notify_parent(zio_t *pio, zio_t *zio, enum zio_wait_type wait)
{
        uint64_t *countp = &pio->io_children[zio->io_child_type][wait];
        int *errorp = &pio->io_child_error[zio->io_child_type];

        mutex_enter(&pio->io_lock);
        if (zio->io_error && !(zio->io_flags & ZIO_FLAG_DONT_PROPAGATE))
                *errorp = zio_worst_error(*errorp, zio->io_error);
        pio->io_reexecute |= zio->io_reexecute;
        ASSERT3U(*countp, >, 0);

        (*countp)--;

        if (*countp == 0 && pio->io_stall == countp) {
        if (--*countp == 0 && pio->io_stall == countp) {
                pio->io_stall = NULL;
                mutex_exit(&pio->io_lock);
                zio_execute(pio);
        } else {
                mutex_exit(&pio->io_lock);
        }
}
_____unchanged_portion_omitted_

/*
 * ==========================================================================
 * Create the various types of I/O (read, write, free, etc)
 * ==========================================================================
 */
static zio_t *
zio_create(zio_t *pio, spa_t *spa, uint64_t txg, const blkptr_t *bp,
    void *data, uint64_t size, zio_done_func_t *done, void *private,
    zio_type_t type, zio_priority_t priority, enum zio_flag flags,
    zio_type_t type, int priority, enum zio_flag flags,
    vdev_t *vd, uint64_t offset, const zbookmark_t *zb,
    enum zio_stage stage, enum zio_stage pipeline)
{
        zio_t *zio;

        ASSERT3U(size, <=, SPA_MAXBLOCKSIZE);
        ASSERT(P2PHASE(size, SPA_MINBLOCKSIZE) == 0);
        ASSERT(P2PHASE(offset, SPA_MINBLOCKSIZE) == 0);

        ASSERT(!vd || spa_config_held(spa, SCL_STATE_ALL, RW_READER));
        ASSERT(!bp || !(flags & ZIO_FLAG_CONFIG_WRITER));
        ASSERT(vd || stage == ZIO_STAGE_OPEN);

        zio = kmem_cache_alloc(zio_cache, KM_SLEEP);
        bzero(zio, sizeof (zio_t));

        mutex_init(&zio->io_lock, NULL, MUTEX_DEFAULT, NULL);
        cv_init(&zio->io_cv, NULL, CV_DEFAULT, NULL);
```

```
516         list_create(&zio->io_parent_list, sizeof (zio_link_t),
517             offsetof(zio_link_t, zl_parent_node));
518         list_create(&zio->io_child_list, sizeof (zio_link_t),
519             offsetof(zio_link_t, zl_child_node));

521         if (vd != NULL)
522                 zio->io_child_type = ZIO_CHILD_VDEV;
523         else if (flags & ZIO_FLAG_GANG_CHILD)
524                 zio->io_child_type = ZIO_CHILD_GANG;
525         else if (flags & ZIO_FLAG_DDT_CHILD)
526                 zio->io_child_type = ZIO_CHILD_DDT;
527         else
528                 zio->io_child_type = ZIO_CHILD_LOGICAL;

530         if (bp != NULL) {
531                 zio->io_bp = (blkptr_t *)bp;
532                 zio->io_bp_copy = *bp;
533                 zio->io_bp_orig = *bp;
534                 if (type != ZIO_TYPE_WRITE ||
535                     zio->io_child_type == ZIO_CHILD_DDT)
536                         zio->io_bp = &zio->io_bp_copy;  /* so caller can free */
537                 if (zio->io_child_type == ZIO_CHILD_LOGICAL)
538                         zio->io_logical = zio;
539                 if (zio->io_child_type > ZIO_CHILD_GANG && BP_IS_GANG(bp))
540                         pipeline |= ZIO_GANG_STAGES;
541         }

543         zio->io_spa = spa;
544         zio->io_txg = txg;
545         zio->io_done = done;
546         zio->io_private = private;
547         zio->io_type = type;
548         zio->io_priority = priority;
549         zio->io_vd = vd;
550         zio->io_offset = offset;
551         zio->io_orig_data = zio->io_data = data;
552         zio->io_orig_size = zio->io_size = size;
553         zio->io_orig_flags = zio->io_flags = flags;
554         zio->io_orig_stage = zio->io_stage = stage;
555         zio->io_orig_pipeline = zio->io_pipeline = pipeline;

557         zio->io_state[ZIO_WAIT_READY] = (stage >= ZIO_STAGE_READY);
558         zio->io_state[ZIO_WAIT_DONE] = (stage >= ZIO_STAGE_DONE);

560         if (zb != NULL)
561                 zio->io_bookmark = *zb;

563         if (pio != NULL) {
564                 if (zio->io_logical == NULL)
565                         zio->io_logical = pio->io_logical;
566                 if (zio->io_child_type == ZIO_CHILD_GANG)
567                         zio->io_gang_leader = pio->io_gang_leader;
568                 zio_add_child(pio, zio);
569         }

571         return (zio);
572 }
```

*unchanged_portion_omitted_*

```
603 zio_t *
604 zio_read(zio_t *pio, spa_t *spa, const blkptr_t *bp,
605     void *data, uint64_t size, zio_done_func_t *done, void *private,
606     zio_priority_t priority, enum zio_flag flags, const zbookmark_t *zb)
623     int priority, enum zio_flag flags, const zbookmark_t *zb)
607 {
```



```
608         zio_t *zio;

610         zio = zio_create(pio, spa, BP_PHYSICAL_BIRTH(bp), bp,
611             data, size, done, private,
612             ZIO_TYPE_READ, priority, flags, NULL, 0, zb,
613             ZIO_STAGE_OPEN, (flags & ZIO_FLAG_DDT_CHILD) ?
614             ZIO_DDT_CHILD_READ_PIPELINE : ZIO_READ_PIPELINE);

616         return (zio);
617 }

619 zio_t *
620 zio_write(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
621     void *data, uint64_t size, const zio_prop_t *zp,
622     zio_done_func_t *ready, zio_done_func_t *physdone, zio_done_func_t *done,
623     void *private,
624     zio_priority_t priority, enum zio_flag flags, const zbookmark_t *zb)
639     zio_done_func_t *ready, zio_done_func_t *done, void *private,
640     int priority, enum zio_flag flags, const zbookmark_t *zb)
625 {
626         zio_t *zio;

628         ASSERT(zp->zp_checksum >= ZIO_CHECKSUM_OFF &&
629             zp->zp_checksum < ZIO_CHECKSUM_FUNCTIONS &&
630             zp->zp_compress >= ZIO_COMPRESS_OFF &&
631             zp->zp_compress < ZIO_COMPRESS_FUNCTIONS &&
632             DMU_OT_IS_VALID(zp->zp_type) &&
633             zp->zp_level < 32 &&
634             zp->zp_copies > 0 &&
635             zp->zp_copies <= spa_max_replication(spa));

637         zio = zio_create(pio, spa, txg, bp, data, size, done, private,
638             ZIO_TYPE_WRITE, priority, flags, NULL, 0, zb,
639             ZIO_STAGE_OPEN, (flags & ZIO_FLAG_DDT_CHILD) ?
640             ZIO_DDT_CHILD_WRITE_PIPELINE : ZIO_WRITE_PIPELINE);

642         zio->io_ready = ready;
643         zio->io_physdone = physdone;
644         zio->io_prop = *zp;

646         return (zio);
647 }

649 zio_t *
650 zio_rewrite(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp, void *data,
651     uint64_t size, zio_done_func_t *done, void *private,
652     zio_priority_t priority, enum zio_flag flags, zbookmark_t *zb)
666     uint64_t size, zio_done_func_t *done, void *private, int priority,
667     enum zio_flag flags, zbookmark_t *zb)
653 {
654         zio_t *zio;

656         zio = zio_create(pio, spa, txg, bp, data, size, done, private,
657             ZIO_TYPE_WRITE, priority, flags, NULL, 0, zb,
658             ZIO_STAGE_OPEN, ZIO_REWRITE_PIPELINE);

660         return (zio);
661 }
```

*unchanged_portion_omitted_*

```
702 zio_t *
703 zio_free_sync(zio_t *pio, spa_t *spa, uint64_t txg, const blkptr_t *bp,
704     enum zio_flag flags)
705 {
706         zio_t *zio;
707         enum zio_stage stage = ZIO_FREE_PIPELINE;
```

```
709         dprintf_bp(bp, "freeing in txg %llu, pass %u",
710             (longlong_t)txg, spa->spa_sync_pass);

712         ASSERT(!BP_IS_HOLE(bp));
713         ASSERT(spa_syncing_txg(spa) == txg);
714         ASSERT(spa_sync_pass(spa) < zfs_sync_pass_deferred_free);

716         metaslab_check_free(spa, bp);
717         arc_freed(spa, bp);

719         /*
720          * GANG and DEDUP blocks can induce a read (for the gang block header,
721          * or the DDT), so issue them asynchronously so that this thread is
722          * not tied up.
723          */
724         if (BP_IS_GANG(bp) || BP_GET_DEDUP(bp))
725                 stage |= ZIO_STAGE_ISSUE_ASYNC;

727         zio = zio_create(pio, spa, txg, bp, NULL, BP_GET_PSIZE(bp),
728             NULL, NULL, ZIO_TYPE_FREE, ZIO_PRIORITY_NOW, flags,
743             NULL, NULL, ZIO_TYPE_FREE, ZIO_PRIORITY_FREE, flags,
729             NULL, 0, NULL, ZIO_STAGE_OPEN, stage);


732         return (zio);
733 }
```

*unchanged_portion_omitted_*

```
764 zio_t *
765 zio_ioctl(zio_t *pio, spa_t *spa, vdev_t *vd, int cmd,
766     zio_done_func_t *done, void *private, enum zio_flag flags)
781     zio_done_func_t *done, void *private, int priority, enum zio_flag flags)
767 {
768         zio_t *zio;
769         int c;

771         if (vd->vdev_children == 0) {
772                 zio = zio_create(pio, spa, 0, NULL, NULL, 0, done, private,
773                     ZIO_TYPE_IOCTL, ZIO_PRIORITY_NOW, flags, vd, 0, NULL,
788                     ZIO_TYPE_IOCTL, priority, flags, vd, 0, NULL,
774                     ZIO_STAGE_OPEN, ZIO_IOCTL_PIPELINE);

776                 zio->io_cmd = cmd;
777         } else {
778                 zio = zio_null(pio, spa, NULL, NULL, NULL, flags);

780                 for (c = 0; c < vd->vdev_children; c++)
781                         zio_nowait(zio_ioctl(zio, spa, vd->vdev_child[c], cmd,
782                             done, private, flags));
797                             done, private, priority, flags));
783         }

785         return (zio);
786 }

788 zio_t *
789 zio_read_phys(zio_t *pio, vdev_t *vd, uint64_t offset, uint64_t size,
790     void *data, int checksum, zio_done_func_t *done, void *private,
791     zio_priority_t priority, enum zio_flag flags, boolean_t labels)
806     int priority, enum zio_flag flags, boolean_t labels)
792 {
793         zio_t *zio;

795         ASSERT(vd->vdev_children == 0);
796         ASSERT(!labels || offset + size <= VDEV_LABEL_START_SIZE ||
```



```
797             offset >= vd->vdev_psize - VDEV_LABEL_END_SIZE);
798         ASSERT3U(offset + size, <=, vd->vdev_psize);

800         zio = zio_create(pio, vd->vdev_spa, 0, NULL, data, size, done, private,
801             ZIO_TYPE_READ, priority, flags, vd, offset, NULL,
802             ZIO_STAGE_OPEN, ZIO_READ_PHYS_PIPELINE);

804         zio->io_prop.zp_checksum = checksum;

806         return (zio);
807 }

809 zio_t *
810 zio_write_phys(zio_t *pio, vdev_t *vd, uint64_t offset, uint64_t size,
811     void *data, int checksum, zio_done_func_t *done, void *private,
812     zio_priority_t priority, enum zio_flag flags, boolean_t labels)
827     int priority, enum zio_flag flags, boolean_t labels)
813 {
814         zio_t *zio;

816         ASSERT(vd->vdev_children == 0);
817         ASSERT(!labels || offset + size <= VDEV_LABEL_START_SIZE ||
818             offset >= vd->vdev_psize - VDEV_LABEL_END_SIZE);
819         ASSERT3U(offset + size, <=, vd->vdev_psize);

821         zio = zio_create(pio, vd->vdev_spa, 0, NULL, data, size, done, private,
822             ZIO_TYPE_WRITE, priority, flags, vd, offset, NULL,
823             ZIO_STAGE_OPEN, ZIO_WRITE_PHYS_PIPELINE);

825         zio->io_prop.zp_checksum = checksum;

827         if (zio_checksum_table[checksum].ci_eck) {
828                 /*
829                  * zec checksums are necessarily destructive -- they modify
830                  * the end of the write buffer to hold the verifier/checksum.
831                  * Therefore, we must make a local copy in case the data is
832                  * being written to multiple places in parallel.
833                  */
834                 void *wbuf = zio_buf_alloc(size);
835                 bcopy(data, wbuf, size);
836                 zio_push_transform(zio, wbuf, size, size, NULL);
837         }

839         return (zio);
840 }

842 /*
843  * Create a child I/O to do some work for us.
844  */
845 zio_t *
846 zio_vdev_child_io(zio_t *pio, blkptr_t *bp, vdev_t *vd, uint64_t offset,
847         void *data, uint64_t size, int type, zio_priority_t priority,
848         enum zio_flag flags, zio_done_func_t *done, void *private)
862         void *data, uint64_t size, int type, int priority, enum zio_flag flags,
863         zio_done_func_t *done, void *private)
849 {
850         enum zio_stage pipeline = ZIO_VDEV_CHILD_PIPELINE;
851         zio_t *zio;

853         ASSERT(vd->vdev_parent ==
854             (pio->io_vd ? pio->io_vd : pio->io_spa->spa_root_vdev));

856         if (type == ZIO_TYPE_READ && bp != NULL) {
857                 /*
858                  * If we have the bp, then the child should perform the
859                  * checksum and the parent need not.  This pushes error
```

```
 860                  * detection as close to the leaves as possible and
 861                  * eliminates redundant checksums in the interior nodes.
 862                  */
 863                 pipeline |= ZIO_STAGE_CHECKSUM_VERIFY;
 864                 pio->io_pipeline &= ~ZIO_STAGE_CHECKSUM_VERIFY;
 865         }

 867         if (vd->vdev_children == 0)
 868                 offset += VDEV_LABEL_START_SIZE;

 870         flags |= ZIO_VDEV_CHILD_FLAGS(pio) | ZIO_FLAG_DONT_PROPAGATE;

 872         /*
 873          * If we've decided to do a repair, the write is not speculative --
 874          * even if the original read was.
 875          */
 876         if (flags & ZIO_FLAG_IO_REPAIR)
 877                 flags &= ~ZIO_FLAG_SPECULATIVE;

 879         zio = zio_create(pio, pio->io_spa, pio->io_txg, bp, data, size,
 880             done, private, type, priority, flags, vd, offset, &pio->io_bookmark,
 881             ZIO_STAGE_VDEV_IO_START >> 1, pipeline);

 883         zio->io_physdone = pio->io_physdone;
 884         if (vd->vdev_ops->vdev_op_leaf && zio->io_logical != NULL)
 885                 zio->io_logical->io_phys_children++;

 887         return (zio);
 888 }

 890 zio_t *
 891 zio_vdev_delegated_io(vdev_t *vd, uint64_t offset, void *data, uint64_t size,
 892         int type, zio_priority_t priority, enum zio_flag flags,
 903         int type, int priority, enum zio_flag flags,
 893         zio_done_func_t *done, void *private)
 894 {
 895         zio_t *zio;

 897         ASSERT(vd->vdev_ops->vdev_op_leaf);

 899         zio = zio_create(NULL, vd->vdev_spa, 0, NULL,
 900             data, size, done, private, type, priority,
 901             flags | ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_RETRY | ZIO_FLAG_DELEGATED,
 912             flags | ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_RETRY,
 902             vd, offset, NULL,
 903             ZIO_STAGE_VDEV_IO_START >> 1, ZIO_VDEV_CHILD_PIPELINE);

 905         return (zio);
 906 }

 908 void
 909 zio_flush(zio_t *zio, vdev_t *vd)
 910 {
 911         zio_nowait(zio_ioctl(zio, zio->io_spa, vd, DKIOCFLUSHWRITECACHE,
 912             NULL, NULL,
 923             NULL, NULL, ZIO_PRIORITY_NOW,
 913             ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY));
 914 }
_____unchanged_portion_omitted_

1752 static int
1753 zio_write_gang_block(zio_t *pio)
1754 {
1755         spa_t *spa = pio->io_spa;
1756         blkptr_t *bp = pio->io_bp;
1757         zio_t *gio = pio->io_gang_leader;
```



```
1758         zio_t *zio;
1759         zio_gang_node_t *gn, **gnpp;
1760         zio_gbh_phys_t *gbh;
1761         uint64_t txg = pio->io_txg;
1762         uint64_t resid = pio->io_size;
1763         uint64_t lsize;
1764         int copies = gio->io_prop.zp_copies;
1765         int gbh_copies = MIN(copies + 1, spa_max_replication(spa));
1766         zio_prop_t zp;
1767         int error;

1769         error = metaslab_alloc(spa, spa_normal_class(spa), SPA_GANGBLOCKSIZE,
1770             bp, gbh_copies, txg, pio == gio ? NULL : gio->io_bp,
1771             METASLAB_HINTBP_FAVOR | METASLAB_GANG_HEADER);
1772         if (error) {
1773                 pio->io_error = error;
1774                 return (ZIO_PIPELINE_CONTINUE);
1775         }

1777         if (pio == gio) {
1778                 gnpp = &gio->io_gang_tree;
1779         } else {
1780                 gnpp = pio->io_private;
1781                 ASSERT(pio->io_ready == zio_write_gang_member_ready);
1782         }

1784         gn = zio_gang_node_alloc(gnpp);
1785         gbh = gn->gn_gbh;
1786         bzero(gbh, SPA_GANGBLOCKSIZE);

1788         /*
1789          * Create the gang header.
1790          */
1791         zio = zio_rewrite(pio, spa, txg, bp, gbh, SPA_GANGBLOCKSIZE, NULL, NULL,
1792             pio->io_priority, ZIO_GANG_CHILD_FLAGS(pio), &pio->io_bookmark);

1794         /*
1795          * Create and nowait the gang children.
1796          */
1797         for (int g = 0; resid != 0; resid -= lsize, g++) {
1798                 lsize = P2ROUNDUP(resid / (SPA_GBH_NBLKPTRS - g),
1799                     SPA_MINBLOCKSIZE);
1800                 ASSERT(lsize >= SPA_MINBLOCKSIZE && lsize <= resid);

1802                 zp.zp_checksum = gio->io_prop.zp_checksum;
1803                 zp.zp_compress = ZIO_COMPRESS_OFF;
1804                 zp.zp_type = DMU_OT_NONE;
1805                 zp.zp_level = 0;
1806                 zp.zp_copies = gio->io_prop.zp_copies;
1807                 zp.zp_dedup = B_FALSE;
1808                 zp.zp_dedup_verify = B_FALSE;
1809                 zp.zp_nopwrite = B_FALSE;

1811                 zio_nowait(zio_write(zio, spa, txg, &gbh->zg_blkptr[g],
1812                     (char *)pio->io_data + (pio->io_size - resid), lsize, &zp,
1813                     zio_write_gang_member_ready, NULL, NULL, &gn->gn_child[g],
1824                     zio_write_gang_member_ready, NULL, &gn->gn_child[g],
1814                     pio->io_priority, ZIO_GANG_CHILD_FLAGS(pio),
1815                     &pio->io_bookmark));
1816         }

1818         /*
1819          * Set pio's pipeline to just wait for zio to finish.
1820          */
1821         pio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
```

```
1823         zio_nowait(zio);

1825         return (ZIO_PIPELINE_CONTINUE);
1826 }
_____unchanged_portion_omitted_

2120 static int
2121 zio_ddt_write(zio_t *zio)
2122 {
2123         spa_t *spa = zio->io_spa;
2124         blkptr_t *bp = zio->io_bp;
2125         uint64_t txg = zio->io_txg;
2126         zio_prop_t *zp = &zio->io_prop;
2127         int p = zp->zp_copies;
2128         int ditto_copies;
2129         zio_t *cio = NULL;
2130         zio_t *dio = NULL;
2131         ddt_t *ddt = ddt_select(spa, bp);
2132         ddt_entry_t *dde;
2133         ddt_phys_t *ddp;

2135         ASSERT(BP_GET_DEDUP(bp));
2136         ASSERT(BP_GET_CHECKSUM(bp) == zp->zp_checksum);
2137         ASSERT(BP_IS_HOLE(bp) || zio->io_bp_override);

2139         ddt_enter(ddt);
2140         dde = ddt_lookup(ddt, bp, B_TRUE);
2141         ddp = &dde->dde_phys[p];

2143         if (zp->zp_dedup_verify && zio_ddt_collision(zio, ddt, dde)) {
2144                 /*
2145                  * If we're using a weak checksum, upgrade to a strong checksum
2146                  * and try again.  If we're already using a strong checksum,
2147                  * we can't resolve it, so just convert to an ordinary write.
2148                  * (And automatically e-mail a paper to Nature?)
2149                  */
2150                 if (!zio_checksum_table[zp->zp_checksum].ci_dedup) {
2151                         zp->zp_checksum = spa_dedup_checksum(spa);
2152                         zio_pop_transforms(zio);
2153                         zio->io_stage = ZIO_STAGE_OPEN;
2154                         BP_ZERO(bp);
2155                 } else {
2156                         zp->zp_dedup = B_FALSE;
2157                 }
2158                 zio->io_pipeline = ZIO_WRITE_PIPELINE;
2159                 ddt_exit(ddt);
2160                 return (ZIO_PIPELINE_CONTINUE);
2161         }

2163         ditto_copies = ddt_ditto_copies_needed(ddt, dde, ddp);
2164         ASSERT(ditto_copies < SPA_DVAS_PER_BP);

2166         if (ditto_copies > ddt_ditto_copies_present(dde) &&
2167             dde->dde_lead_zio[DDT_PHYS_DITTO] == NULL) {
2168                 zio_prop_t czp = *zp;

2170                 czp.zp_copies = ditto_copies;

2172                 /*
2173                  * If we arrived here with an override bp, we won't have run
2174                  * the transform stack, so we won't have the data we need to
2175                  * generate a child i/o.  So, toss the override bp and restart.
2176                  * This is safe, because using the override bp is just an
2177                  * optimization; and it's rare, so the cost doesn't matter.
2178                  */
2179                 if (zio->io_bp_override) {
```



```
2180                         zio_pop_transforms(zio);
2181                         zio->io_stage = ZIO_STAGE_OPEN;
2182                         zio->io_pipeline = ZIO_WRITE_PIPELINE;
2183                         zio->io_bp_override = NULL;
2184                         BP_ZERO(bp);
2185                         ddt_exit(ddt);
2186                         return (ZIO_PIPELINE_CONTINUE);
2187                 }

2189                 dio = zio_write(zio, spa, txg, bp, zio->io_orig_data,
2190                     zio->io_orig_size, &czp, NULL, NULL,
2201                     zio->io_orig_size, &czp, NULL,
2191                     zio_ddt_ditto_write_done, dde, zio->io_priority,
2192                     ZIO_DDT_CHILD_FLAGS(zio), &zio->io_bookmark);

2194                 zio_push_transform(dio, zio->io_data, zio->io_size, 0, NULL);
2195                 dde->dde_lead_zio[DDT_PHYS_DITTO] = dio;
2196         }

2198         if (ddp->ddp_phys_birth != 0 || dde->dde_lead_zio[p] != NULL) {
2199                 if (ddp->ddp_phys_birth != 0)
2200                         ddt_bp_fill(ddp, bp, txg);
2201                 if (dde->dde_lead_zio[p] != NULL)
2202                         zio_add_child(zio, dde->dde_lead_zio[p]);
2203                 else
2204                         ddt_phys_addref(ddp);
2205         } else if (zio->io_bp_override) {
2206                 ASSERT(bp->blk_birth == txg);
2207                 ASSERT(BP_EQUAL(bp, zio->io_bp_override));
2208                 ddt_phys_fill(ddp, bp);
2209                 ddt_phys_addref(ddp);
2210         } else {
2211                 cio = zio_write(zio, spa, txg, bp, zio->io_orig_data,
2212                     zio->io_orig_size, zp, zio_ddt_child_write_ready, NULL,
2223                     zio->io_orig_size, zp, zio_ddt_child_write_ready,
2213                     zio_ddt_child_write_done, dde, zio->io_priority,
2214                     ZIO_DDT_CHILD_FLAGS(zio), &zio->io_bookmark);

2216                 zio_push_transform(cio, zio->io_data, zio->io_size, 0, NULL);
2217                 dde->dde_lead_zio[p] = cio;
2218         }

2220         ddt_exit(ddt);

2222         if (cio)
2223                 zio_nowait(cio);
2224         if (dio)
2225                 zio_nowait(dio);

2227         return (ZIO_PIPELINE_CONTINUE);
2228 }
_____unchanged_portion_omitted_

2571 static int
2572 zio_vdev_io_assess(zio_t *zio)
2573 {
2574         vdev_t *vd = zio->io_vd;

2576         if (zio_wait_for_children(zio, ZIO_CHILD_VDEV, ZIO_WAIT_DONE))
2577                 return (ZIO_PIPELINE_STOP);

2579         if (vd == NULL && !(zio->io_flags & ZIO_FLAG_CONFIG_WRITER))
2580                 spa_config_exit(zio->io_spa, SCL_ZIO, zio);

2582         if (zio->io_vsd != NULL) {
2583                 zio->io_vsd_ops->vsd_free(zio);
```

```
2584                 zio->io_vsd = NULL;
2585         }

2587         if (zio_injection_enabled && zio->io_error == 0)
2588                 zio->io_error = zio_handle_fault_injection(zio, EIO);

2590         /*
2591          * If the I/O failed, determine whether we should attempt to retry it.
2592          *
2593          * On retry, we cut in line in the issue queue, since we don't want
2594          * compression/checksumming/etc. work to prevent our (cheap) IO reissue.
2595          */
2596         if (zio->io_error && vd == NULL &&
2597             !(zio->io_flags & (ZIO_FLAG_DONT_RETRY | ZIO_FLAG_IO_RETRY))) {
2598                 ASSERT(!(zio->io_flags & ZIO_FLAG_DONT_QUEUE)); /* not a leaf */
2599                 ASSERT(!(zio->io_flags & ZIO_FLAG_IO_BYPASS));  /* not a leaf */
2600                 zio->io_error = 0;
2601                 zio->io_flags |= ZIO_FLAG_IO_RETRY |
2602                     ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE;
2603                 zio->io_stage = ZIO_STAGE_VDEV_IO_START >> 1;
2604                 zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE,
2605                     zio_requeue_io_start_cut_in_line);
2606                 return (ZIO_PIPELINE_STOP);
2607         }

2609         /*
2610          * If we got an error on a leaf device, convert it to ENXIO
2611          * if the device is not accessible at all.
2612          */
2613         if (zio->io_error && vd != NULL && vd->vdev_ops->vdev_op_leaf &&
2614             !vdev_accessible(vd, zio))
2615                 zio->io_error = SET_ERROR(ENXIO);

2617         /*
2618          * If we can't write to an interior vdev (mirror or RAID-Z),
2619          * set vdev_cant_write so that we stop trying to allocate from it.
2620          */
2621         if (zio->io_error == ENXIO && zio->io_type == ZIO_TYPE_WRITE &&
2622             vd != NULL && !vd->vdev_ops->vdev_op_leaf) {
2623                 vd->vdev_cant_write = B_TRUE;
2624         }

2626         if (zio->io_error)
2627                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;

2629         if (vd != NULL && vd->vdev_ops->vdev_op_leaf &&
2630             zio->io_physdone != NULL) {
2631                 ASSERT(!(zio->io_flags & ZIO_FLAG_DELEGATED));
2632                 ASSERT(zio->io_child_type == ZIO_CHILD_VDEV);
2633                 zio->io_physdone(zio->io_logical);
2634         }

2636         return (ZIO_PIPELINE_CONTINUE);
2637 }
```

*_____unchanged_portion_omitted_*