```
**********************************************************
   45230 Thu Jun 21 14:13:52 2012
new/usr/src/uts/common/fs/zfs/dmu_send.c
**********************************************************
______unchanged_portion_omitted_

  74 static int
  75 dump_free(dmu_sendarg_t *dsp, uint64_t object, uint64_t offset,
  76     uint64_t length)
  77 {
  78         struct drr_free *drrf = &(dsp->dsa_drr->drr_u.drr_free);

  80         if (length != -1ULL && offset + length < offset)
  81                 length = -1ULL;

  83 #endif /* ! codereview */
  84         /*
  85          * If there is a pending op, but it's not PENDING_FREE, push it out,
  86          * since free block aggregation can only be done for blocks of the
  87          * same type (i.e., DRR_FREE records can only be aggregated with
  88          * other DRR_FREE records.  DRR_FREEOBJECTS records can only be
  89          * aggregated with other DRR_FREEOBJECTS records.
  90          */
  91         if (dsp->dsa_pending_op != PENDING_NONE &&
  92             dsp->dsa_pending_op != PENDING_FREE) {
  93                 if (dump_bytes(dsp, dsp->dsa_drr,
  94                     sizeof (dmu_replay_record_t)) != 0)
  95                         return (EINTR);
  96                 dsp->dsa_pending_op = PENDING_NONE;
  97         }

  99         if (dsp->dsa_pending_op == PENDING_FREE) {
 100                 /*
 101                  * There should never be a PENDING_FREE if length is -1
 102                  * (because dump_dnode is the only place where this
 103                  * function is called with a -1, and only after flushing
 104                  * any pending record).
 105                  */
 106                 ASSERT(length != -1ULL);
 107                 /*
 108                  * Check to see whether this free block can be aggregated
 109                  * with pending one.
 110                  */
 111                 if (drrf->drr_object == object && drrf->drr_offset +
 112                     drrf->drr_length == offset) {
 113                         drrf->drr_length += length;
 114                         return (0);
 115                 } else {
 116                         /* not a continuation.  Push out pending record */
 117                         if (dump_bytes(dsp, dsp->dsa_drr,
 118                             sizeof (dmu_replay_record_t)) != 0)
 119                                 return (EINTR);
 120                         dsp->dsa_pending_op = PENDING_NONE;
 121                 }
 122         }
 123         /* create a FREE record and make it pending */
 124         bzero(dsp->dsa_drr, sizeof (dmu_replay_record_t));
 125         dsp->dsa_drr->drr_type = DRR_FREE;
 126         drrf->drr_object = object;
 127         drrf->drr_offset = offset;
 128         drrf->drr_length = length;
 129         drrf->drr_toguid = dsp->dsa_toguid;
 130         if (length == -1ULL) {
 131                 if (dump_bytes(dsp, dsp->dsa_drr,
 132                     sizeof (dmu_replay_record_t)) != 0)
 133                         return (EINTR);
```



```
 134         } else {
 135                 dsp->dsa_pending_op = PENDING_FREE;
 136         }

 138         return (0);
 139 }

 141 static int
 142 dump_data(dmu_sendarg_t *dsp, dmu_object_type_t type,
 143     uint64_t object, uint64_t offset, int blksz, const blkptr_t *bp, void *data)
 144 {
 145         struct drr_write *drrw = &(dsp->dsa_drr->drr_u.drr_write);


 148         /*
 149          * If there is any kind of pending aggregation (currently either
 150          * a grouping of free objects or free blocks), push it out to
 151          * the stream, since aggregation can't be done across operations
 152          * of different types.
 153          */
 154         if (dsp->dsa_pending_op != PENDING_NONE) {
 155                 if (dump_bytes(dsp, dsp->dsa_drr,
 156                     sizeof (dmu_replay_record_t)) != 0)
 157                         return (EINTR);
 158                 dsp->dsa_pending_op = PENDING_NONE;
 159         }
 160         /* write a DATA record */
 161         bzero(dsp->dsa_drr, sizeof (dmu_replay_record_t));
 162         dsp->dsa_drr->drr_type = DRR_WRITE;
 163         drrw->drr_object = object;
 164         drrw->drr_type = type;
 165         drrw->drr_offset = offset;
 166         drrw->drr_length = blksz;
 167         drrw->drr_toguid = dsp->dsa_toguid;
 168         drrw->drr_checksumtype = BP_GET_CHECKSUM(bp);
 169         if (zio_checksum_table[drrw->drr_checksumtype].ci_dedup)
 170                 drrw->drr_checksumflags |= DRR_CHECKSUM_DEDUP;
 171         DDK_SET_LSIZE(&drrw->drr_key, BP_GET_LSIZE(bp));
 172         DDK_SET_PSIZE(&drrw->drr_key, BP_GET_PSIZE(bp));
 173         DDK_SET_COMPRESS(&drrw->drr_key, BP_GET_COMPRESS(bp));
 174         drrw->drr_key.ddk_cksum = bp->blk_cksum;

 176         if (dump_bytes(dsp, dsp->dsa_drr, sizeof (dmu_replay_record_t)) != 0)
 177                 return (EINTR);
 178         if (dump_bytes(dsp, data, blksz) != 0)
 179                 return (EINTR);
 180         return (0);
 181 }

 183 static int
 184 dump_spill(dmu_sendarg_t *dsp, uint64_t object, int blksz, void *data)
 185 {
 186         struct drr_spill *drrs = &(dsp->dsa_drr->drr_u.drr_spill);

 188         if (dsp->dsa_pending_op != PENDING_NONE) {
 189                 if (dump_bytes(dsp, dsp->dsa_drr,
 190                     sizeof (dmu_replay_record_t)) != 0)
 191                         return (EINTR);
 192                 dsp->dsa_pending_op = PENDING_NONE;
 193         }

 195         /* write a SPILL record */
 196         bzero(dsp->dsa_drr, sizeof (dmu_replay_record_t));
 197         dsp->dsa_drr->drr_type = DRR_SPILL;
 198         drrs->drr_object = object;
 199         drrs->drr_length = blksz;
```

```
200         drrs->drr_toguid = dsp->dsa_toguid;
201
202         if (dump_bytes(dsp, dsp->dsa_drr, sizeof (dmu_replay_record_t)))
203                 return (EINTR);
204         if (dump_bytes(dsp, data, blksz))
205                 return (EINTR);
206         return (0);
207 }
208
209 static int
210 dump_freeobjects(dmu_sendarg_t *dsp, uint64_t firstobj, uint64_t numobjs)
211 {
212         struct drr_freeobjects *drrfo = &(dsp->dsa_drr->drr_u.drr_freeobjects);
213
214         /*
215          * If there is a pending op, but it's not PENDING_FREEOBJECTS,
216          * push it out, since free block aggregation can only be done for
217          * blocks of the same type (i.e., DRR_FREE records can only be
218          * aggregated with other DRR_FREE records.  DRR_FREEOBJECTS records
219          * can only be aggregated with other DRR_FREEOBJECTS records.
220          */
221         if (dsp->dsa_pending_op != PENDING_NONE &&
222             dsp->dsa_pending_op != PENDING_FREEOBJECTS) {
223                 if (dump_bytes(dsp, dsp->dsa_drr,
224                     sizeof (dmu_replay_record_t)) != 0)
225                         return (EINTR);
226                 dsp->dsa_pending_op = PENDING_NONE;
227         }
228         if (dsp->dsa_pending_op == PENDING_FREEOBJECTS) {
229                 /*
230                  * See whether this free object array can be aggregated
231                  * with pending one
232                  */
233                 if (drrfo->drr_firstobj + drrfo->drr_numobjs == firstobj) {
234                         drrfo->drr_numobjs += numobjs;
235                         return (0);
236                 } else {
237                         /* can't be aggregated.  Push out pending record */
238                         if (dump_bytes(dsp, dsp->dsa_drr,
239                             sizeof (dmu_replay_record_t)) != 0)
240                                 return (EINTR);
241                         dsp->dsa_pending_op = PENDING_NONE;
242                 }
243         }
244
245         /* write a FREEOBJECTS record */
246         bzero(dsp->dsa_drr, sizeof (dmu_replay_record_t));
247         dsp->dsa_drr->drr_type = DRR_FREEOBJECTS;
248         drrfo->drr_firstobj = firstobj;
249         drrfo->drr_numobjs = numobjs;
250         drrfo->drr_toguid = dsp->dsa_toguid;
251
252         dsp->dsa_pending_op = PENDING_FREEOBJECTS;
253
254         return (0);
255 }
256
257 static int
258 dump_dnode(dmu_sendarg_t *dsp, uint64_t object, dnode_phys_t *dnp)
259 {
260         struct drr_object *drro = &(dsp->dsa_drr->drr_u.drr_object);
261
262         if (dnp == NULL || dnp->dn_type == DMU_OT_NONE)
263                 return (dump_freeobjects(dsp, object, 1));
264
265         if (dsp->dsa_pending_op != PENDING_NONE) {
```



```
266                 if (dump_bytes(dsp, dsp->dsa_drr,
267                     sizeof (dmu_replay_record_t)) != 0)
268                         return (EINTR);
269                 dsp->dsa_pending_op = PENDING_NONE;
270         }
271
272         /* write an OBJECT record */
273         bzero(dsp->dsa_drr, sizeof (dmu_replay_record_t));
274         dsp->dsa_drr->drr_type = DRR_OBJECT;
275         drro->drr_object = object;
276         drro->drr_type = dnp->dn_type;
277         drro->drr_bonustype = dnp->dn_bonustype;
278         drro->drr_blksz = dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT;
279         drro->drr_bonuslen = dnp->dn_bonuslen;
280         drro->drr_checksumtype = dnp->dn_checksum;
281         drro->drr_compress = dnp->dn_compress;
282         drro->drr_toguid = dsp->dsa_toguid;
283
284         if (dump_bytes(dsp, dsp->dsa_drr, sizeof (dmu_replay_record_t)) != 0)
285                 return (EINTR);
286
287         if (dump_bytes(dsp, DN_BONUS(dnp), P2ROUNDUP(dnp->dn_bonuslen, 8)) != 0)
288                 return (EINTR);
289
290         /* free anything past the end of the file */
291         if (dump_free(dsp, object, (dnp->dn_maxblkid + 1) *
292             (dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT), -1ULL))
293                 return (EINTR);
294         if (dsp->dsa_err)
295                 return (EINTR);
296         return (0);
297 }
298
299 #define BP_SPAN(dnp, level) \
300         (((uint64_t)dnp->dn_datablkszsec) << (SPA_MINBLOCKSHIFT + \
301         (level) * (dnp->dn_indblkshift - SPA_BLKPTRSHIFT)))
302
303 /* ARGSUSED */
304 static int
305 backup_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf,
306     const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
307 {
308         dmu_sendarg_t *dsp = arg;
309         dmu_object_type_t type = bp ? BP_GET_TYPE(bp) : DMU_OT_NONE;
310         int err = 0;
311
312         if (issig(JUSTLOOKING) && issig(FORREAL))
313                 return (EINTR);
314
315         if (zb->zb_object != DMU_META_DNODE_OBJECT &&
316             DMU_OBJECT_IS_SPECIAL(zb->zb_object)) {
317                 return (0);
318         } else if (bp == NULL && zb->zb_object == DMU_META_DNODE_OBJECT) {
319                 uint64_t span = BP_SPAN(dnp, zb->zb_level);
320                 uint64_t dnobj = (zb->zb_blkid * span) >> DNODE_SHIFT;
321                 err = dump_freeobjects(dsp, dnobj, span >> DNODE_SHIFT);
322         } else if (bp == NULL) {
323                 uint64_t span = BP_SPAN(dnp, zb->zb_level);
324                 err = dump_free(dsp, zb->zb_object, zb->zb_blkid * span, span);
325         } else if (zb->zb_level > 0 || type == DMU_OT_OBJSET) {
326                 return (0);
327         } else if (type == DMU_OT_DNODE) {
328                 dnode_phys_t *blk;
329                 int i;
330                 int blksz = BP_GET_LSIZE(bp);
331                 uint32_t aflags = ARC_WAIT;
```

```
332                 arc_buf_t *abuf;
333
334                 if (dsl_read(NULL, spa, bp, pbuf,
335                     arc_getbuf_func, &abuf, ZIO_PRIORITY_ASYNC_READ,
336                     ZIO_FLAG_CANFAIL, &aflags, zb) != 0)
337                         return (EIO);
338
339                 blk = abuf->b_data;
340                 for (i = 0; i < blksz >> DNODE_SHIFT; i++) {
341                         uint64_t dnobj = (zb->zb_blkid <<
342                             (DNODE_BLOCK_SHIFT - DNODE_SHIFT)) + i;
343                         err = dump_dnode(dsp, dnobj, blk+i);
344                         if (err)
345                                 break;
346                 }
347                 (void) arc_buf_remove_ref(abuf, &abuf);
348         } else if (type == DMU_OT_SA) {
349                 uint32_t aflags = ARC_WAIT;
350                 arc_buf_t *abuf;
351                 int blksz = BP_GET_LSIZE(bp);
352
353                 if (arc_read_nolock(NULL, spa, bp,
354                     arc_getbuf_func, &abuf, ZIO_PRIORITY_ASYNC_READ,
355                     ZIO_FLAG_CANFAIL, &aflags, zb) != 0)
356                         return (EIO);
357
358                 err = dump_spill(dsp, zb->zb_object, blksz, abuf->b_data);
359                 (void) arc_buf_remove_ref(abuf, &abuf);
360         } else { /* it's a level-0 block of a regular object */
361                 uint32_t aflags = ARC_WAIT;
362                 arc_buf_t *abuf;
363                 int blksz = BP_GET_LSIZE(bp);
364
365                 if (dsl_read(NULL, spa, bp, pbuf,
366                     arc_getbuf_func, &abuf, ZIO_PRIORITY_ASYNC_READ,
367                     ZIO_FLAG_CANFAIL, &aflags, zb) != 0) {
368                         if (zfs_send_corrupt_data) {
369                                 /* Send a block filled with 0x"zfs badd bloc" */
370                                 abuf = arc_buf_alloc(spa, blksz, &abuf,
371                                     ARC_BUFC_DATA);
372                                 uint64_t *ptr;
373                                 for (ptr = abuf->b_data;
374                                     (char *)ptr < (char *)abuf->b_data + blksz;
375                                     ptr++)
376                                         *ptr = 0x2f5baddb10c;
377                         } else {
378                                 return (EIO);
379                         }
380                 }
381
382                 err = dump_data(dsp, type, zb->zb_object, zb->zb_blkid * blksz,
383                     blksz, bp, abuf->b_data);
384                 (void) arc_buf_remove_ref(abuf, &abuf);
385         }
386
387         ASSERT(err == 0 || err == EINTR);
388         return (err);
389 }
390
391 int
392 dmu_send(objset_t *tosnap, objset_t *fromsnap, boolean_t fromorigin,
393     int outfd, vnode_t *vp, offset_t *off)
394 {
395         dsl_dataset_t *ds = tosnap->os_dsl_dataset;
396         dsl_dataset_t *fromds = fromsnap ? fromsnap->os_dsl_dataset : NULL;
397         dmu_replay_record_t *drr;
```



```
398         dmu_sendarg_t *dsp;
399         int err;
400         uint64_t fromtxg = 0;
401
402         /* tosnap must be a snapshot */
403         if (ds->ds_phys->ds_next_snap_obj == 0)
404                 return (EINVAL);
405
406         /* fromsnap must be an earlier snapshot from the same fs as tosnap */
407         if (fromds && (ds->ds_dir != fromds->ds_dir ||
408             fromds->ds_phys->ds_creation_txg >= ds->ds_phys->ds_creation_txg))
409                 return (EXDEV);
410
411         if (fromorigin) {
412                 dsl_pool_t *dp = ds->ds_dir->dd_pool;
413
414                 if (fromsnap)
415                         return (EINVAL);
416
417                 if (dsl_dir_is_clone(ds->ds_dir)) {
418                         rw_enter(&dp->dp_config_rwlock, RW_READER);
419                         err = dsl_dataset_hold_obj(dp,
420                             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &fromds);
421                         rw_exit(&dp->dp_config_rwlock);
422                         if (err)
423                                 return (err);
424                 } else {
425                         fromorigin = B_FALSE;
426                 }
427         }
428
429
430         drr = kmem_zalloc(sizeof (dmu_replay_record_t), KM_SLEEP);
431         drr->drr_type = DRR_BEGIN;
432         drr->drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
433         DMU_SET_STREAM_HDRTYPE(drr->drr_u.drr_begin.drr_versioninfo,
434             DMU_SUBSTREAM);
435
436 #ifdef _KERNEL
437         if (dmu_objset_type(tosnap) == DMU_OST_ZFS) {
438                 uint64_t version;
439                 if (zfs_get_zplprop(tosnap, ZFS_PROP_VERSION, &version) != 0) {
440                         kmem_free(drr, sizeof (dmu_replay_record_t));
441                         return (EINVAL);
442                 }
443                 if (version == ZPL_VERSION_SA) {
444                         DMU_SET_FEATUREFLAGS(
445                             drr->drr_u.drr_begin.drr_versioninfo,
446                             DMU_BACKUP_FEATURE_SA_SPILL);
447                 }
448         }
449 #endif
450
451         drr->drr_u.drr_begin.drr_creation_time =
452             ds->ds_phys->ds_creation_time;
453         drr->drr_u.drr_begin.drr_type = tosnap->os_phys->os_type;
454         if (fromorigin)
455                 drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CLONE;
456         drr->drr_u.drr_begin.drr_toguid = ds->ds_phys->ds_guid;
457         if (ds->ds_phys->ds_flags & DS_FLAG_CI_DATASET)
458                 drr->drr_u.drr_begin.drr_flags |= DRR_FLAG_CI_DATA;
459
460         if (fromds)
461                 drr->drr_u.drr_begin.drr_fromguid = fromds->ds_phys->ds_guid;
462         dsl_dataset_name(ds, drr->drr_u.drr_begin.drr_toname);
```

```
464         if (fromds)
465                 fromtxg = fromds->ds_phys->ds_creation_txg;
466         if (fromorigin)
467                 dsl_dataset_rele(fromds, FTAG);
468
469         dsp = kmem_zalloc(sizeof (dmu_sendarg_t), KM_SLEEP);
470
471         dsp->dsa_drr = drr;
472         dsp->dsa_vp = vp;
473         dsp->dsa_outfd = outfd;
474         dsp->dsa_proc = curproc;
475         dsp->dsa_os = tosnap;
476         dsp->dsa_off = off;
477         dsp->dsa_toguid = ds->ds_phys->ds_guid;
478         ZIO_SET_CHECKSUM(&dsp->dsa_zc, 0, 0, 0, 0);
479         dsp->dsa_pending_op = PENDING_NONE;
480
481         mutex_enter(&ds->ds_sendstream_lock);
482         list_insert_head(&ds->ds_sendstreams, dsp);
483         mutex_exit(&ds->ds_sendstream_lock);
484
485         if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
486                 err = dsp->dsa_err;
487                 goto out;
488         }
489
490         err = traverse_dataset(ds, fromtxg, TRAVERSE_PRE | TRAVERSE_PREFETCH,
491             backup_cb, dsp);
492
493         if (dsp->dsa_pending_op != PENDING_NONE)
494                 if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0)
495                         err = EINTR;
496
497         if (err) {
498                 if (err == EINTR && dsp->dsa_err)
499                         err = dsp->dsa_err;
500                 goto out;
501         }
502
503         bzero(drr, sizeof (dmu_replay_record_t));
504         drr->drr_type = DRR_END;
505         drr->drr_u.drr_end.drr_checksum = dsp->dsa_zc;
506         drr->drr_u.drr_end.drr_toguid = dsp->dsa_toguid;
507
508         if (dump_bytes(dsp, drr, sizeof (dmu_replay_record_t)) != 0) {
509                 err = dsp->dsa_err;
510                 goto out;
511         }
512
513 out:
514         mutex_enter(&ds->ds_sendstream_lock);
515         list_remove(&ds->ds_sendstreams, dsp);
516         mutex_exit(&ds->ds_sendstream_lock);
517
518         kmem_free(drr, sizeof (dmu_replay_record_t));
519         kmem_free(dsp, sizeof (dmu_sendarg_t));
520
521         return (err);
522 }
523
524 int
525 dmu_send_estimate(objset_t *tosnap, objset_t *fromsnap, boolean_t fromorigin,
526     uint64_t *sizep)
527 {
528         dsl_dataset_t *ds = tosnap->os_dsl_dataset;
529         dsl_dataset_t *fromds = fromsnap ? fromsnap->os_dsl_dataset : NULL;
```



```
530         dsl_pool_t *dp = ds->ds_dir->dd_pool;
531         int err;
532         uint64_t size;
533
534         /* tosnap must be a snapshot */
535         if (ds->ds_phys->ds_next_snap_obj == 0)
536                 return (EINVAL);
537
538         /* fromsnap must be an earlier snapshot from the same fs as tosnap */
539         if (fromds && (ds->ds_dir != fromds->ds_dir ||
540             fromds->ds_phys->ds_creation_txg >= ds->ds_phys->ds_creation_txg))
541                 return (EXDEV);
542
543         if (fromorigin) {
544                 if (fromsnap)
545                         return (EINVAL);
546
547                 if (dsl_dir_is_clone(ds->ds_dir)) {
548                         rw_enter(&dp->dp_config_rwlock, RW_READER);
549                         err = dsl_dataset_hold_obj(dp,
550                             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &fromds);
551                         rw_exit(&dp->dp_config_rwlock);
552                         if (err)
553                                 return (err);
554                 } else {
555                         fromorigin = B_FALSE;
556                 }
557         }
558
559         /* Get uncompressed size estimate of changed data. */
560         if (fromds == NULL) {
561                 size = ds->ds_phys->ds_uncompressed_bytes;
562         } else {
563                 uint64_t used, comp;
564                 err = dsl_dataset_space_written(fromds, ds,
565                     &used, &comp, &size);
566                 if (fromorigin)
567                         dsl_dataset_rele(fromds, FTAG);
568                 if (err)
569                         return (err);
570         }
571
572         /*
573          * Assume that space (both on-disk and in-stream) is dominated by
574          * data.  We will adjust for indirect blocks and the copies property,
575          * but ignore per-object space used (eg, dnodes and DRR_OBJECT records).
576          */
577
578         /*
579          * Subtract out approximate space used by indirect blocks.
580          * Assume most space is used by data blocks (non-indirect, non-dnode).
581          * Assume all blocks are recordsize.  Assume ditto blocks and
582          * internal fragmentation counter out compression.
583          *
584          * Therefore, space used by indirect blocks is sizeof(blkptr_t) per
585          * block, which we observe in practice.
586          */
587         uint64_t recordsize;
588         rw_enter(&dp->dp_config_rwlock, RW_READER);
589         err = dsl_prop_get_ds(ds, "recordsize",
590             sizeof (recordsize), 1, &recordsize, NULL);
591         rw_exit(&dp->dp_config_rwlock);
592         if (err)
593                 return (err);
594         size -= size / recordsize * sizeof (blkptr_t);
```

```
596         /* Add in the space for the record associated with each block. */
597         size += size / recordsize * sizeof (dmu_replay_record_t);

599         *sizep = size;

601         return (0);
602 }

604 struct recvbeginsyncarg {
605         const char *tofs;
606         const char *tosnap;
607         dsl_dataset_t *origin;
608         uint64_t fromguid;
609         dmu_objset_type_t type;
610         void *tag;
611         boolean_t force;
612         uint64_t dsflags;
613         char clonelastname[MAXNAMELEN];
614         dsl_dataset_t *ds; /* the ds to recv into; returned from the syncfunc */
615         cred_t *cr;
616 };

618 /* ARGSUSED */
619 static int
620 recv_new_check(void *arg1, void *arg2, dmu_tx_t *tx)
621 {
622         dsl_dir_t *dd = arg1;
623         struct recvbeginsyncarg *rbsa = arg2;
624         objset_t *mos = dd->dd_pool->dp_meta_objset;
625         uint64_t val;
626         int err;

628         err = zap_lookup(mos, dd->dd_phys->dd_child_dir_zapobj,
629             strrchr(rbsa->tofs, '/') + 1, sizeof (uint64_t), 1, &val);

631         if (err != ENOENT)
632                 return (err ? err : EEXIST);

634         if (rbsa->origin) {
635                 /* make sure it's a snap in the same pool */
636                 if (rbsa->origin->ds_dir->dd_pool != dd->dd_pool)
637                         return (EXDEV);
638                 if (!dsl_dataset_is_snapshot(rbsa->origin))
639                         return (EINVAL);
640                 if (rbsa->origin->ds_phys->ds_guid != rbsa->fromguid)
641                         return (ENODEV);
642         }

644         return (0);
645 }

647 static void
648 recv_new_sync(void *arg1, void *arg2, dmu_tx_t *tx)
649 {
650         dsl_dir_t *dd = arg1;
651         struct recvbeginsyncarg *rbsa = arg2;
652         uint64_t flags = DS_FLAG_INCONSISTENT | rbsa->dsflags;
653         uint64_t dsobj;

655         /* Create and open new dataset. */
656         dsobj = dsl_dataset_create_sync(dd, strrchr(rbsa->tofs, '/') + 1,
657             rbsa->origin, flags, rbsa->cr, tx);
658         VERIFY(0 == dsl_dataset_own_obj(dd->dd_pool, dsobj,
659             B_TRUE, dmu_recv_tag, &rbsa->ds));

661         if (rbsa->origin == NULL) {
```



```
662                 (void) dmu_objset_create_impl(dd->dd_pool->dp_spa,
663                     rbsa->ds, &rbsa->ds->ds_phys->ds_bp, rbsa->type, tx);
664         }

666         spa_history_log_internal(LOG_DS_REPLAY_FULL_SYNC,
667             dd->dd_pool->dp_spa, tx, "dataset = %lld", dsobj);
668 }

670 /* ARGSUSED */
671 static int
672 recv_existing_check(void *arg1, void *arg2, dmu_tx_t *tx)
673 {
674         dsl_dataset_t *ds = arg1;
675         struct recvbeginsyncarg *rbsa = arg2;
676         int err;
677         uint64_t val;

679         /* must not have any changes since most recent snapshot */
680         if (!rbsa->force && dsl_dataset_modified_since_lastsnap(ds))
681                 return (ETXTBSY);

683         /* new snapshot name must not exist */
684         err = zap_lookup(ds->ds_dir->dd_pool->dp_meta_objset,
685             ds->ds_phys->ds_snapnames_zapobj, rbsa->tosnap, 8, 1, &val);
686         if (err == 0)
687                 return (EEXIST);
688         if (err != ENOENT)
689                 return (err);

691         if (rbsa->fromguid) {
692                 /* if incremental, most recent snapshot must match fromguid */
693                 if (ds->ds_prev == NULL)
694                         return (ENODEV);

696                 /*
697                  * most recent snapshot must match fromguid, or there are no
698                  * changes since the fromguid one
699                  */
700                 if (ds->ds_prev->ds_phys->ds_guid != rbsa->fromguid) {
701                         uint64_t birth = ds->ds_prev->ds_phys->ds_bp.blk_birth;
702                         uint64_t obj = ds->ds_prev->ds_phys->ds_prev_snap_obj;
703                         while (obj != 0) {
704                                 dsl_dataset_t *snap;
705                                 err = dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
706                                     obj, FTAG, &snap);
707                                 if (err)
708                                         return (ENODEV);
709                                 if (snap->ds_phys->ds_creation_txg < birth) {
710                                         dsl_dataset_rele(snap, FTAG);
711                                         return (ENODEV);
712                                 }
713                                 if (snap->ds_phys->ds_guid == rbsa->fromguid) {
714                                         dsl_dataset_rele(snap, FTAG);
715                                         break; /* it's ok */
716                                 }
717                                 obj = snap->ds_phys->ds_prev_snap_obj;
718                                 dsl_dataset_rele(snap, FTAG);
719                         }
720                         if (obj == 0)
721                                 return (ENODEV);
722                 }
723         } else {
724                 /* if full, most recent snapshot must be $ORIGIN */
725                 if (ds->ds_phys->ds_prev_snap_txg >= TXG_INITIAL)
726                         return (ENODEV);
727         }
```

```
729         /* temporary clone name must not exist */
730         err = zap_lookup(ds->ds_dir->dd_pool->dp_meta_objset,
731             ds->ds_dir->dd_phys->dd_child_dir_zapobj,
732             rbsa->clonelastname, 8, 1, &val);
733         if (err == 0)
734                 return (EEXIST);
735         if (err != ENOENT)
736                 return (err);
737
738         return (0);
739 }
740
741 /* ARGSUSED */
742 static void
743 recv_existing_sync(void *arg1, void *arg2, dmu_tx_t *tx)
744 {
745         dsl_dataset_t *ohds = arg1;
746         struct recvbeginsyncarg *rbsa = arg2;
747         dsl_pool_t *dp = ohds->ds_dir->dd_pool;
748         dsl_dataset_t *cds;
749         uint64_t flags = DS_FLAG_INCONSISTENT | rbsa->dsflags;
750         uint64_t dsobj;
751
752         /* create and open the temporary clone */
753         dsobj = dsl_dataset_create_sync(ohds->ds_dir, rbsa->clonelastname,
754             ohds->ds_prev, flags, rbsa->cr, tx);
755         VERIFY(0 == dsl_dataset_own_obj(dp, dsobj, B_TRUE, dmu_recv_tag, &cds));
756
757         /*
758          * If we actually created a non-clone, we need to create the
759          * objset in our new dataset.
760          */
761         if (BP_IS_HOLE(dsl_dataset_get_blkptr(cds))) {
762                 (void) dmu_objset_create_impl(dp->dp_spa,
763                     cds, dsl_dataset_get_blkptr(cds), rbsa->type, tx);
764         }
765
766         rbsa->ds = cds;
767
768         spa_history_log_internal(LOG_DS_REPLAY_INC_SYNC,
769             dp->dp_spa, tx, "dataset = %lld", dsobj);
770 }
771
772 static boolean_t
773 dmu_recv_verify_features(dsl_dataset_t *ds, struct drr_begin *drrb)
774 {
775         int featureflags;
776
777         featureflags = DMU_GET_FEATUREFLAGS(drrb->drr_versioninfo);
778
779         /* Verify pool version supports SA if SA_SPILL feature set */
780         return ((featureflags & DMU_BACKUP_FEATURE_SA_SPILL) &&
781             (spa_version(dsl_dataset_get_spa(ds)) < SPA_VERSION_SA));
782 }
783
784 /*
785  * NB: callers *MUST* call dmu_recv_stream() if dmu_recv_begin()
786  * succeeds; otherwise we will leak the holds on the datasets.
787  */
788 int
789 dmu_recv_begin(char *tofs, char *tosnap, char *top_ds, struct drr_begin *drrb,
790     boolean_t force, objset_t *origin, dmu_recv_cookie_t *drc)
791 {
792         int err = 0;
793         boolean_t byteswap;
```



```
794         struct recvbeginsyncarg rbsa = { 0 };
795         uint64_t versioninfo;
796         int flags;
797         dsl_dataset_t *ds;
798
799         if (drrb->drr_magic == DMU_BACKUP_MAGIC)
800                 byteswap = FALSE;
801         else if (drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC))
802                 byteswap = TRUE;
803         else
804                 return (EINVAL);
805
806         rbsa.tofs = tofs;
807         rbsa.tosnap = tosnap;
808         rbsa.origin = origin ? origin->os_dsl_dataset : NULL;
809         rbsa.fromguid = drrb->drr_fromguid;
810         rbsa.type = drrb->drr_type;
811         rbsa.tag = FTAG;
812         rbsa.dsflags = 0;
813         rbsa.cr = CRED();
814         versioninfo = drrb->drr_versioninfo;
815         flags = drrb->drr_flags;
816
817         if (byteswap) {
818                 rbsa.type = BSWAP_32(rbsa.type);
819                 rbsa.fromguid = BSWAP_64(rbsa.fromguid);
820                 versioninfo = BSWAP_64(versioninfo);
821                 flags = BSWAP_32(flags);
822         }
823
824         if (DMU_GET_STREAM_HDRTYPE(versioninfo) == DMU_COMPOUNDSTREAM ||
825             rbsa.type >= DMU_OST_NUMTYPES ||
826             ((flags & DRR_FLAG_CLONE) && origin == NULL))
827                 return (EINVAL);
828
829         if (flags & DRR_FLAG_CI_DATA)
830                 rbsa.dsflags = DS_FLAG_CI_DATASET;
831
832         bzero(drc, sizeof (dmu_recv_cookie_t));
833         drc->drc_drrb = drrb;
834         drc->drc_tosnap = tosnap;
835         drc->drc_top_ds = top_ds;
836         drc->drc_force = force;
837
838         /*
839          * Process the begin in syncing context.
840          */
841
842         /* open the dataset we are logically receiving into */
843         err = dsl_dataset_hold(tofs, dmu_recv_tag, &ds);
844         if (err == 0) {
845                 if (dmu_recv_verify_features(ds, drrb)) {
846                         dsl_dataset_rele(ds, dmu_recv_tag);
847                         return (ENOTSUP);
848                 }
849                 /* target fs already exists; recv into temp clone */
850
851                 /* Can't recv a clone into an existing fs */
852                 if (flags & DRR_FLAG_CLONE) {
853                         dsl_dataset_rele(ds, dmu_recv_tag);
854                         return (EINVAL);
855                 }
856
857                 /* must not have an incremental recv already in progress */
858                 if (!mutex_tryenter(&ds->ds_recvlock)) {
859                         dsl_dataset_rele(ds, dmu_recv_tag);
```

```
860                         return (EBUSY);
861                 }

863                 /* tmp clone name is: tofs/%tosnap" */
864                 (void) snprintf(rbsa.clonelastname, sizeof (rbsa.clonelastname),
865                     "%%%s", tosnap);
866                 rbsa.force = force;
867                 err = dsl_sync_task_do(ds->ds_dir->dd_pool,
868                     recv_existing_check, recv_existing_sync, ds, &rbsa, 5);
869                 if (err) {
870                         mutex_exit(&ds->ds_recvlock);
871                         dsl_dataset_rele(ds, dmu_recv_tag);
872                         return (err);
873                 }
874                 drc->drc_logical_ds = ds;
875                 drc->drc_real_ds = rbsa.ds;
876         } else if (err == ENOENT) {
877                 /* target fs does not exist; must be a full backup or clone */
878                 char *cp;

880                 /*
881                  * If it's a non-clone incremental, we are missing the
882                  * target fs, so fail the recv.
883                  */
884                 if (rbsa.fromguid && !(flags & DRR_FLAG_CLONE))
885                         return (ENOENT);

887                 /* Open the parent of tofs */
888                 cp = strrchr(tofs, '/');
889                 *cp = '\0';
890                 err = dsl_dataset_hold(tofs, FTAG, &ds);
891                 *cp = '/';
892                 if (err)
893                         return (err);

895                 if (dmu_recv_verify_features(ds, drrb)) {
896                         dsl_dataset_rele(ds, FTAG);
897                         return (ENOTSUP);
898                 }

900                 err = dsl_sync_task_do(ds->ds_dir->dd_pool,
901                     recv_new_check, recv_new_sync, ds->ds_dir, &rbsa, 5);
902                 dsl_dataset_rele(ds, FTAG);
903                 if (err)
904                         return (err);
905                 drc->drc_logical_ds = drc->drc_real_ds = rbsa.ds;
906                 drc->drc_newfs = B_TRUE;
907         }

909         return (err);
910 }

912 struct restorearg {
913         int err;
914         int byteswap;
915         vnode_t *vp;
916         char *buf;
917         uint64_t voff;
918         int bufsize; /* amount of memory allocated for buf */
919         zio_cksum_t cksum;
920         avl_tree_t *guid_to_ds_map;
921 };

923 typedef struct guid_map_entry {
924         uint64_t        guid;
925         dsl_dataset_t   *gme_ds;
```



```
926         avl_node_t      avlnode;
927 } guid_map_entry_t;

929 static int
930 guid_compare(const void *arg1, const void *arg2)
931 {
932         const guid_map_entry_t *gmep1 = arg1;
933         const guid_map_entry_t *gmep2 = arg2;

935         if (gmep1->guid < gmep2->guid)
936                 return (-1);
937         else if (gmep1->guid > gmep2->guid)
938                 return (1);
939         return (0);
940 }

942 static void
943 free_guid_map_onexit(void *arg)
944 {
945         avl_tree_t *ca = arg;
946         void *cookie = NULL;
947         guid_map_entry_t *gmep;

949         while ((gmep = avl_destroy_nodes(ca, &cookie)) != NULL) {
950                 dsl_dataset_rele(gmep->gme_ds, ca);
951                 kmem_free(gmep, sizeof (guid_map_entry_t));
952         }
953         avl_destroy(ca);
954         kmem_free(ca, sizeof (avl_tree_t));
955 }

957 static void *
958 restore_read(struct restorearg *ra, int len)
959 {
960         void *rv;
961         int done = 0;

963         /* some things will require 8-byte alignment, so everything must */
964         ASSERT3U(len % 8, ==, 0);

966         while (done < len) {
967                 ssize_t resid;

969                 ra->err = vn_rdwr(UIO_READ, ra->vp,
970                     (caddr_t)ra->buf + done, len - done,
971                     ra->voff, UIO_SYSSPACE, FAPPEND,
972                     RLIM64_INFINITY, CRED(), &resid);

974                 if (resid == len - done)
975                         ra->err = EINVAL;
976                 ra->voff += len - done - resid;
977                 done = len - resid;
978                 if (ra->err)
979                         return (NULL);
980         }

982         ASSERT3U(done, ==, len);
983         rv = ra->buf;
984         if (ra->byteswap)
985                 fletcher_4_incremental_byteswap(rv, len, &ra->cksum);
986         else
987                 fletcher_4_incremental_native(rv, len, &ra->cksum);
988         return (rv);
989 }

991 static void
```

```
backup_byteswap(dmu_replay_record_t *drr)
{
#define DO64(X) (drr->drr_u.X = BSWAP_64(drr->drr_u.X))
#define DO32(X) (drr->drr_u.X = BSWAP_32(drr->drr_u.X))
        drr->drr_type = BSWAP_32(drr->drr_type);
        drr->drr_payloadlen = BSWAP_32(drr->drr_payloadlen);
        switch (drr->drr_type) {
        case DRR_BEGIN:
                DO64(drr_begin.drr_magic);
                DO64(drr_begin.drr_versioninfo);
                DO64(drr_begin.drr_creation_time);
                DO32(drr_begin.drr_type);
                DO32(drr_begin.drr_flags);
                DO64(drr_begin.drr_toguid);
                DO64(drr_begin.drr_fromguid);
                break;
        case DRR_OBJECT:
                DO64(drr_object.drr_object);
                /* DO64(drr_object.drr_allocation_txg); */
                DO32(drr_object.drr_type);
                DO32(drr_object.drr_bonustype);
                DO32(drr_object.drr_blksz);
                DO32(drr_object.drr_bonuslen);
                DO64(drr_object.drr_toguid);
                break;
        case DRR_FREEOBJECTS:
                DO64(drr_freeobjects.drr_firstobj);
                DO64(drr_freeobjects.drr_numobjs);
                DO64(drr_freeobjects.drr_toguid);
                break;
        case DRR_WRITE:
                DO64(drr_write.drr_object);
                DO32(drr_write.drr_type);
                DO64(drr_write.drr_offset);
                DO64(drr_write.drr_length);
                DO64(drr_write.drr_toguid);
                DO64(drr_write.drr_key.ddk_cksum.zc_word[0]);
                DO64(drr_write.drr_key.ddk_cksum.zc_word[1]);
                DO64(drr_write.drr_key.ddk_cksum.zc_word[2]);
                DO64(drr_write.drr_key.ddk_cksum.zc_word[3]);
                DO64(drr_write.drr_key.ddk_prop);
                break;
        case DRR_WRITE_BYREF:
                DO64(drr_write_byref.drr_object);
                DO64(drr_write_byref.drr_offset);
                DO64(drr_write_byref.drr_length);
                DO64(drr_write_byref.drr_toguid);
                DO64(drr_write_byref.drr_refguid);
                DO64(drr_write_byref.drr_refobject);
                DO64(drr_write_byref.drr_refoffset);
                DO64(drr_write_byref.drr_key.ddk_cksum.zc_word[0]);
                DO64(drr_write_byref.drr_key.ddk_cksum.zc_word[1]);
                DO64(drr_write_byref.drr_key.ddk_cksum.zc_word[2]);
                DO64(drr_write_byref.drr_key.ddk_cksum.zc_word[3]);
                DO64(drr_write_byref.drr_key.ddk_prop);
                break;
        case DRR_FREE:
                DO64(drr_free.drr_object);
                DO64(drr_free.drr_offset);
                DO64(drr_free.drr_length);
                DO64(drr_free.drr_toguid);
                break;
        case DRR_SPILL:
                DO64(drr_spill.drr_object);
                DO64(drr_spill.drr_length);
                DO64(drr_spill.drr_toguid);
```



```
                break;
        case DRR_END:
                DO64(drr_end.drr_checksum.zc_word[0]);
                DO64(drr_end.drr_checksum.zc_word[1]);
                DO64(drr_end.drr_checksum.zc_word[2]);
                DO64(drr_end.drr_checksum.zc_word[3]);
                DO64(drr_end.drr_toguid);
                break;
        }
#undef DO64
#undef DO32
}

static int
restore_object(struct restorearg *ra, objset_t *os, struct drr_object *drro)
{
        int err;
        dmu_tx_t *tx;
        void *data = NULL;

        if (drro->drr_type == DMU_OT_NONE ||
            !DMU_OT_IS_VALID(drro->drr_type) ||
            !DMU_OT_IS_VALID(drro->drr_bonustype) ||
            drro->drr_checksumtype >= ZIO_CHECKSUM_FUNCTIONS ||
            drro->drr_compress >= ZIO_COMPRESS_FUNCTIONS ||
            P2PHASE(drro->drr_blksz, SPA_MINBLOCKSIZE) ||
            drro->drr_blksz < SPA_MINBLOCKSIZE ||
            drro->drr_blksz > SPA_MAXBLOCKSIZE ||
            drro->drr_bonuslen > DN_MAX_BONUSLEN) {
                return (EINVAL);
        }

        err = dmu_object_info(os, drro->drr_object, NULL);

        if (err != 0 && err != ENOENT)
                return (EINVAL);

        if (drro->drr_bonuslen) {
                data = restore_read(ra, P2ROUNDUP(drro->drr_bonuslen, 8));
                if (ra->err)
                        return (ra->err);
        }

        if (err == ENOENT) {
                /* currently free, want to be allocated */
                tx = dmu_tx_create(os);
                dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);
                err = dmu_tx_assign(tx, TXG_WAIT);
                if (err) {
                        dmu_tx_abort(tx);
                        return (err);
                }
                err = dmu_object_claim(os, drro->drr_object,
                    drro->drr_type, drro->drr_blksz,
                    drro->drr_bonustype, drro->drr_bonuslen, tx);
                dmu_tx_commit(tx);
        } else {
                /* currently allocated, want to be allocated */
                err = dmu_object_reclaim(os, drro->drr_object,
                    drro->drr_type, drro->drr_blksz,
                    drro->drr_bonustype, drro->drr_bonuslen);
        }
        if (err) {
                return (EINVAL);
        }
```

```
1124         tx = dmu_tx_create(os);
1125         dmu_tx_hold_bonus(tx, drro->drr_object);
1126         err = dmu_tx_assign(tx, TXG_WAIT);
1127         if (err) {
1128                 dmu_tx_abort(tx);
1129                 return (err);
1130         }
1131
1132         dmu_object_set_checksum(os, drro->drr_object, drro->drr_checksumtype,
1133             tx);
1134         dmu_object_set_compress(os, drro->drr_object, drro->drr_compress, tx);
1135
1136         if (data != NULL) {
1137                 dmu_buf_t *db;
1138
1139                 VERIFY(0 == dmu_bonus_hold(os, drro->drr_object, FTAG, &db));
1140                 dmu_buf_will_dirty(db, tx);
1141
1142                 ASSERT3U(db->db_size, >=, drro->drr_bonuslen);
1143                 bcopy(data, db->db_data, drro->drr_bonuslen);
1144                 if (ra->byteswap) {
1145                         dmu_object_byteswap_t byteswap =
1146                             DMU_OT_BYTESWAP(drro->drr_bonustype);
1147                         dmu_ot_byteswap[byteswap].ob_func(db->db_data,
1148                             drro->drr_bonuslen);
1149                 }
1150                 dmu_buf_rele(db, FTAG);
1151         }
1152         dmu_tx_commit(tx);
1153         return (0);
1154 }
1155
1156 /* ARGSUSED */
1157 static int
1158 restore_freeobjects(struct restorearg *ra, objset_t *os,
1159     struct drr_freeobjects *drrfo)
1160 {
1161         uint64_t obj;
1162
1163         if (drrfo->drr_firstobj + drrfo->drr_numobjs < drrfo->drr_firstobj)
1164                 return (EINVAL);
1165
1166         for (obj = drrfo->drr_firstobj;
1167             obj < drrfo->drr_firstobj + drrfo->drr_numobjs;
1168             (void) dmu_object_next(os, &obj, FALSE, 0)) {
1169                 int err;
1170
1171                 if (dmu_object_info(os, obj, NULL) != 0)
1172                         continue;
1173
1174                 err = dmu_free_object(os, obj);
1175                 if (err)
1176                         return (err);
1177         }
1178         return (0);
1179 }
1180
1181 static int
1182 restore_write(struct restorearg *ra, objset_t *os,
1183     struct drr_write *drrw)
1184 {
1185         dmu_tx_t *tx;
1186         void *data;
1187         int err;
1188
1189         if (drrw->drr_offset + drrw->drr_length < drrw->drr_offset ||
```



```
1190             !DMU_OT_IS_VALID(drrw->drr_type))
1191                 return (EINVAL);
1192
1193         data = restore_read(ra, drrw->drr_length);
1194         if (data == NULL)
1195                 return (ra->err);
1196
1197         if (dmu_object_info(os, drrw->drr_object, NULL) != 0)
1198                 return (EINVAL);
1199
1200         tx = dmu_tx_create(os);
1201
1202         dmu_tx_hold_write(tx, drrw->drr_object,
1203             drrw->drr_offset, drrw->drr_length);
1204         err = dmu_tx_assign(tx, TXG_WAIT);
1205         if (err) {
1206                 dmu_tx_abort(tx);
1207                 return (err);
1208         }
1209         if (ra->byteswap) {
1210                 dmu_object_byteswap_t byteswap =
1211                     DMU_OT_BYTESWAP(drrw->drr_type);
1212                 dmu_ot_byteswap[byteswap].ob_func(data, drrw->drr_length);
1213         }
1214         dmu_write(os, drrw->drr_object,
1215             drrw->drr_offset, drrw->drr_length, data, tx);
1216         dmu_tx_commit(tx);
1217         return (0);
1218 }
1219
1220 /*
1221  * Handle a DRR_WRITE_BYREF record.  This record is used in dedup'ed
1222  * streams to refer to a copy of the data that is already on the
1223  * system because it came in earlier in the stream.  This function
1224  * finds the earlier copy of the data, and uses that copy instead of
1225  * data from the stream to fulfill this write.
1226  */
1227 static int
1228 restore_write_byref(struct restorearg *ra, objset_t *os,
1229     struct drr_write_byref *drrwbr)
1230 {
1231         dmu_tx_t *tx;
1232         int err;
1233         guid_map_entry_t gmesrch;
1234         guid_map_entry_t *gmep;
1235         avl_index_t     where;
1236         objset_t *ref_os = NULL;
1237         dmu_buf_t *dbp;
1238
1239         if (drrwbr->drr_offset + drrwbr->drr_length < drrwbr->drr_offset)
1240                 return (EINVAL);
1241
1242         /*
1243          * If the GUID of the referenced dataset is different from the
1244          * GUID of the target dataset, find the referenced dataset.
1245          */
1246         if (drrwbr->drr_toguid != drrwbr->drr_refguid) {
1247                 gmesrch.guid = drrwbr->drr_refguid;
1248                 if ((gmep = avl_find(ra->guid_to_ds_map, &gmesrch,
1249                     &where)) == NULL) {
1250                         return (EINVAL);
1251                 }
1252                 if (dmu_objset_from_ds(gmep->gme_ds, &ref_os))
1253                         return (EINVAL);
1254         } else {
1255                 ref_os = os;
```

```
1256         }

1258         if (err = dmu_buf_hold(ref_os, drrwbr->drr_refobject,
1259             drrwbr->drr_refoffset, FTAG, &dbp, DMU_READ_PREFETCH))
1260                 return (err);

1262         tx = dmu_tx_create(os);

1264         dmu_tx_hold_write(tx, drrwbr->drr_object,
1265             drrwbr->drr_offset, drrwbr->drr_length);
1266         err = dmu_tx_assign(tx, TXG_WAIT);
1267         if (err) {
1268                 dmu_tx_abort(tx);
1269                 return (err);
1270         }
1271         dmu_write(os, drrwbr->drr_object,
1272             drrwbr->drr_offset, drrwbr->drr_length, dbp->db_data, tx);
1273         dmu_buf_rele(dbp, FTAG);
1274         dmu_tx_commit(tx);
1275         return (0);
1276 }

1278 static int
1279 restore_spill(struct restorearg *ra, objset_t *os, struct drr_spill *drrs)
1280 {
1281         dmu_tx_t *tx;
1282         void *data;
1283         dmu_buf_t *db, *db_spill;
1284         int err;

1286         if (drrs->drr_length < SPA_MINBLOCKSIZE ||
1287             drrs->drr_length > SPA_MAXBLOCKSIZE)
1288                 return (EINVAL);

1290         data = restore_read(ra, drrs->drr_length);
1291         if (data == NULL)
1292                 return (ra->err);

1294         if (dmu_object_info(os, drrs->drr_object, NULL) != 0)
1295                 return (EINVAL);

1297         VERIFY(0 == dmu_bonus_hold(os, drrs->drr_object, FTAG, &db));
1298         if ((err = dmu_spill_hold_by_bonus(db, FTAG, &db_spill)) != 0) {
1299                 dmu_buf_rele(db, FTAG);
1300                 return (err);
1301         }

1303         tx = dmu_tx_create(os);

1305         dmu_tx_hold_spill(tx, db->db_object);

1307         err = dmu_tx_assign(tx, TXG_WAIT);
1308         if (err) {
1309                 dmu_buf_rele(db, FTAG);
1310                 dmu_buf_rele(db_spill, FTAG);
1311                 dmu_tx_abort(tx);
1312                 return (err);
1313         }
1314         dmu_buf_will_dirty(db_spill, tx);

1316         if (db_spill->db_size < drrs->drr_length)
1317                 VERIFY(0 == dbuf_spill_set_blksz(db_spill,
1318                     drrs->drr_length, tx));
1319         bcopy(data, db_spill->db_data, drrs->drr_length);

1321         dmu_buf_rele(db, FTAG);
```



```
1322         dmu_buf_rele(db_spill, FTAG);

1324         dmu_tx_commit(tx);
1325         return (0);
1326 }

1328 /* ARGSUSED */
1329 static int
1330 restore_free(struct restorearg *ra, objset_t *os,
1331     struct drr_free *drrf)
1332 {
1333         int err;

1335         if (drrf->drr_length != -1ULL &&
1336             drrf->drr_offset + drrf->drr_length < drrf->drr_offset)
1337                 return (EINVAL);

1339         if (dmu_object_info(os, drrf->drr_object, NULL) != 0)
1340                 return (EINVAL);

1342         err = dmu_free_long_range(os, drrf->drr_object,
1343             drrf->drr_offset, drrf->drr_length);
1344         return (err);
1345 }

1347 /*
1348  * NB: callers *must* call dmu_recv_end() if this succeeds.
1349  */
1350 int
1351 dmu_recv_stream(dmu_recv_cookie_t *drc, vnode_t *vp, offset_t *voffp,
1352     int cleanup_fd, uint64_t *action_handlep)
1353 {
1354         struct restorearg ra = { 0 };
1355         dmu_replay_record_t *drr;
1356         objset_t *os;
1357         zio_cksum_t pcksum;
1358         int featureflags;

1360         if (drc->drc_drrb->drr_magic == BSWAP_64(DMU_BACKUP_MAGIC))
1361                 ra.byteswap = TRUE;

1363         {
1364                 /* compute checksum of drr_begin record */
1365                 dmu_replay_record_t *drr;
1366                 drr = kmem_zalloc(sizeof (dmu_replay_record_t), KM_SLEEP);

1368                 drr->drr_type = DRR_BEGIN;
1369                 drr->drr_u.drr_begin = *drc->drc_drrb;
1370                 if (ra.byteswap) {
1371                         fletcher_4_incremental_byteswap(drr,
1372                             sizeof (dmu_replay_record_t), &ra.cksum);
1373                 } else {
1374                         fletcher_4_incremental_native(drr,
1375                             sizeof (dmu_replay_record_t), &ra.cksum);
1376                 }
1377                 kmem_free(drr, sizeof (dmu_replay_record_t));
1378         }

1380         if (ra.byteswap) {
1381                 struct drr_begin *drrb = drc->drc_drrb;
1382                 drrb->drr_magic = BSWAP_64(drrb->drr_magic);
1383                 drrb->drr_versioninfo = BSWAP_64(drrb->drr_versioninfo);
1384                 drrb->drr_creation_time = BSWAP_64(drrb->drr_creation_time);
1385                 drrb->drr_type = BSWAP_32(drrb->drr_type);
1386                 drrb->drr_toguid = BSWAP_64(drrb->drr_toguid);
1387                 drrb->drr_fromguid = BSWAP_64(drrb->drr_fromguid);
```

```
1388         }

1390         ra.vp = vp;
1391         ra.voff = *voffp;
1392         ra.bufsize = 1<<20;
1393         ra.buf = kmem_alloc(ra.bufsize, KM_SLEEP);

1395         /* these were verified in dmu_recv_begin */
1396         ASSERT(DMU_GET_STREAM_HDRTYPE(drc->drc_drrb->drr_versioninfo) ==
1397             DMU_SUBSTREAM);
1398         ASSERT(drc->drc_drrb->drr_type < DMU_OST_NUMTYPES);

1400         /*
1401          * Open the objset we are modifying.
1402          */
1403         VERIFY(dmu_objset_from_ds(drc->drc_real_ds, &os) == 0);

1405         ASSERT(drc->drc_real_ds->ds_phys->ds_flags & DS_FLAG_INCONSISTENT);

1407         featureflags = DMU_GET_FEATUREFLAGS(drc->drc_drrb->drr_versioninfo);

1409         /* if this stream is dedup'ed, set up the avl tree for guid mapping */
1410         if (featureflags & DMU_BACKUP_FEATURE_DEDUP) {
1411                 minor_t minor;

1413                 if (cleanup_fd == -1) {
1414                         ra.err = EBADF;
1415                         goto out;
1416                 }
1417                 ra.err = zfs_onexit_fd_hold(cleanup_fd, &minor);
1418                 if (ra.err) {
1419                         cleanup_fd = -1;
1420                         goto out;
1421                 }

1423                 if (*action_handlep == 0) {
1424                         ra.guid_to_ds_map =
1425                             kmem_alloc(sizeof (avl_tree_t), KM_SLEEP);
1426                         avl_create(ra.guid_to_ds_map, guid_compare,
1427                             sizeof (guid_map_entry_t),
1428                             offsetof(guid_map_entry_t, avlnode));
1429                         ra.err = zfs_onexit_add_cb(minor,
1430                             free_guid_map_onexit, ra.guid_to_ds_map,
1431                             action_handlep);
1432                         if (ra.err)
1433                                 goto out;
1434                 } else {
1435                         ra.err = zfs_onexit_cb_data(minor, *action_handlep,
1436                             (void **)&ra.guid_to_ds_map);
1437                         if (ra.err)
1438                                 goto out;
1439                 }

1441                 drc->drc_guid_to_ds_map = ra.guid_to_ds_map;
1442         }

1444         /*
1445          * Read records and process them.
1446          */
1447         pcksum = ra.cksum;
1448         while (ra.err == 0 &&
1449             NULL != (drr = restore_read(&ra, sizeof (*drr)))) {
1450                 if (issig(JUSTLOOKING) && issig(FORREAL)) {
1451                         ra.err = EINTR;
1452                         goto out;
1453                 }
```



```
1455                 if (ra.byteswap)
1456                         backup_byteswap(drr);

1458                 switch (drr->drr_type) {
1459                 case DRR_OBJECT:
1460                 {
1461                         /*
1462                          * We need to make a copy of the record header,
1463                          * because restore_{object,write} may need to
1464                          * restore_read(), which will invalidate drr.
1465                          */
1466                         struct drr_object drro = drr->drr_u.drr_object;
1467                         ra.err = restore_object(&ra, os, &drro);
1468                         break;
1469                 }
1470                 case DRR_FREEOBJECTS:
1471                 {
1472                         struct drr_freeobjects drrfo =
1473                             drr->drr_u.drr_freeobjects;
1474                         ra.err = restore_freeobjects(&ra, os, &drrfo);
1475                         break;
1476                 }
1477                 case DRR_WRITE:
1478                 {
1479                         struct drr_write drrw = drr->drr_u.drr_write;
1480                         ra.err = restore_write(&ra, os, &drrw);
1481                         break;
1482                 }
1483                 case DRR_WRITE_BYREF:
1484                 {
1485                         struct drr_write_byref drrwbr =
1486                             drr->drr_u.drr_write_byref;
1487                         ra.err = restore_write_byref(&ra, os, &drrwbr);
1488                         break;
1489                 }
1490                 case DRR_FREE:
1491                 {
1492                         struct drr_free drrf = drr->drr_u.drr_free;
1493                         ra.err = restore_free(&ra, os, &drrf);
1494                         break;
1495                 }
1496                 case DRR_END:
1497                 {
1498                         struct drr_end drre = drr->drr_u.drr_end;
1499                         /*
1500                          * We compare against the *previous* checksum
1501                          * value, because the stored checksum is of
1502                          * everything before the DRR_END record.
1503                          */
1504                         if (!ZIO_CHECKSUM_EQUAL(drre.drr_checksum, pcksum))
1505                                 ra.err = ECKSUM;
1506                         goto out;
1507                 }
1508                 case DRR_SPILL:
1509                 {
1510                         struct drr_spill drrs = drr->drr_u.drr_spill;
1511                         ra.err = restore_spill(&ra, os, &drrs);
1512                         break;
1513                 }
1514                 default:
1515                         ra.err = EINVAL;
1516                         goto out;
1517                 }
1518                 pcksum = ra.cksum;
1519         }
```

```
1520         ASSERT(ra.err != 0);

1522 out:
1523         if ((featureflags & DMU_BACKUP_FEATURE_DEDUP) && (cleanup_fd != -1))
1524                 zfs_onexit_fd_rele(cleanup_fd);

1526         if (ra.err != 0) {
1527                 /*
1528                  * destroy what we created, so we don't leave it in the
1529                  * inconsistent restoring state.
1530                  */
1531                 txg_wait_synced(drc->drc_real_ds->ds_dir->dd_pool, 0);

1533                 (void) dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag,
1534                     B_FALSE);
1535                 if (drc->drc_real_ds != drc->drc_logical_ds) {
1536                         mutex_exit(&drc->drc_logical_ds->ds_recvlock);
1537                         dsl_dataset_rele(drc->drc_logical_ds, dmu_recv_tag);
1538                 }
1539         }

1541         kmem_free(ra.buf, ra.bufsize);
1542         *voffp = ra.voff;
1543         return (ra.err);
1544 }

1546 struct recvendsyncarg {
1547         char *tosnap;
1548         uint64_t creation_time;
1549         uint64_t toguid;
1550 };

1552 static int
1553 recv_end_check(void *arg1, void *arg2, dmu_tx_t *tx)
1554 {
1555         dsl_dataset_t *ds = arg1;
1556         struct recvendsyncarg *resa = arg2;

1558         return (dsl_dataset_snapshot_check(ds, resa->tosnap, tx));
1559 }

1561 static void
1562 recv_end_sync(void *arg1, void *arg2, dmu_tx_t *tx)
1563 {
1564         dsl_dataset_t *ds = arg1;
1565         struct recvendsyncarg *resa = arg2;

1567         dsl_dataset_snapshot_sync(ds, resa->tosnap, tx);

1569         /* set snapshot's creation time and guid */
1570         dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
1571         ds->ds_prev->ds_phys->ds_creation_time = resa->creation_time;
1572         ds->ds_prev->ds_phys->ds_guid = resa->toguid;
1573         ds->ds_prev->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;

1575         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1576         ds->ds_phys->ds_flags &= ~DS_FLAG_INCONSISTENT;
1577 }

1579 static int
1580 add_ds_to_guidmap(avl_tree_t *guid_map, dsl_dataset_t *ds)
1581 {
1582         dsl_pool_t *dp = ds->ds_dir->dd_pool;
1583         uint64_t snapobj = ds->ds_phys->ds_prev_snap_obj;
1584         dsl_dataset_t *snapds;
1585         guid_map_entry_t *gmep;
```



```
1586         int err;

1588         ASSERT(guid_map != NULL);

1590         rw_enter(&dp->dp_config_rwlock, RW_READER);
1591         err = dsl_dataset_hold_obj(dp, snapobj, guid_map, &snapds);
1592         if (err == 0) {
1593                 gmep = kmem_alloc(sizeof (guid_map_entry_t), KM_SLEEP);
1594                 gmep->guid = snapds->ds_phys->ds_guid;
1595                 gmep->gme_ds = snapds;
1596                 avl_add(guid_map, gmep);
1597         }

1599         rw_exit(&dp->dp_config_rwlock);
1600         return (err);
1601 }

1603 static int
1604 dmu_recv_existing_end(dmu_recv_cookie_t *drc)
1605 {
1606         struct recvendsyncarg resa;
1607         dsl_dataset_t *ds = drc->drc_logical_ds;
1608         int err, myerr;

1610         /*
1611          * XXX hack; seems the ds is still dirty and dsl_pool_zil_clean()
1612          * expects it to have a ds_user_ptr (and zil), but clone_swap()
1613          * can close it.
1614          */
1615         txg_wait_synced(ds->ds_dir->dd_pool, 0);

1617         if (dsl_dataset_tryown(ds, FALSE, dmu_recv_tag)) {
1618                 err = dsl_dataset_clone_swap(drc->drc_real_ds, ds,
1619                     drc->drc_force);
1620                 if (err)
1621                         goto out;
1622         } else {
1623                 mutex_exit(&ds->ds_recvlock);
1624                 dsl_dataset_rele(ds, dmu_recv_tag);
1625                 (void) dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag,
1626                     B_FALSE);
1627                 return (EBUSY);
1628         }

1630         resa.creation_time = drc->drc_drrb->drr_creation_time;
1631         resa.toguid = drc->drc_drrb->drr_toguid;
1632         resa.tosnap = drc->drc_tosnap;

1634         err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1635             recv_end_check, recv_end_sync, ds, &resa, 3);
1636         if (err) {
1637                 /* swap back */
1638                 (void) dsl_dataset_clone_swap(drc->drc_real_ds, ds, B_TRUE);
1639         }

1641 out:
1642         mutex_exit(&ds->ds_recvlock);
1643         if (err == 0 && drc->drc_guid_to_ds_map != NULL)
1644                 (void) add_ds_to_guidmap(drc->drc_guid_to_ds_map, ds);
1645         dsl_dataset_disown(ds, dmu_recv_tag);
1646         myerr = dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag, B_FALSE);
1647         ASSERT3U(myerr, ==, 0);
1648         return (err);
1649 }

1651 static int
```

```
1652 dmu_recv_new_end(dmu_recv_cookie_t *drc)
1653 {
1654         struct recvendsyncarg resa;
1655         dsl_dataset_t *ds = drc->drc_logical_ds;
1656         int err;

1658         /*
1659          * XXX hack; seems the ds is still dirty and dsl_pool_zil_clean()
1660          * expects it to have a ds_user_ptr (and zil), but clone_swap()
1661          * can close it.
1662          */
1663         txg_wait_synced(ds->ds_dir->dd_pool, 0);

1665         resa.creation_time = drc->drc_drrb->drr_creation_time;
1666         resa.toguid = drc->drc_drrb->drr_toguid;
1667         resa.tosnap = drc->drc_tosnap;

1669         err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1670             recv_end_check, recv_end_sync, ds, &resa, 3);
1671         if (err) {
1672                 /* clean up the fs we just recv'd into */
1673                 (void) dsl_dataset_destroy(ds, dmu_recv_tag, B_FALSE);
1674         } else {
1675                 if (drc->drc_guid_to_ds_map != NULL)
1676                         (void) add_ds_to_guidmap(drc->drc_guid_to_ds_map, ds);
1677                 /* release the hold from dmu_recv_begin */
1678                 dsl_dataset_disown(ds, dmu_recv_tag);
1679         }
1680         return (err);
1681 }

1683 int
1684 dmu_recv_end(dmu_recv_cookie_t *drc)
1685 {
1686         if (drc->drc_logical_ds != drc->drc_real_ds)
1687                 return (dmu_recv_existing_end(drc));
1688         else
1689                 return (dmu_recv_new_end(drc));
1690 }
```