```
**********************************************************
    8478 Fri May 24 00:51:01 2013
new/usr/src/cmd/ndmpd/ndmp/ndmpd_chkpnt.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************
______unchanged_portion_omitted_

/*
 * Put a hold on snapshot
 */
int
snapshot_hold(char *volname, char *snapname, char *jname, boolean_t recursive)
{
        zfs_handle_t *zhp;
        char *p;

        if ((zhp = zfs_open(zlibh, volname, ZFS_TYPE_DATASET)) == 0) {
                NDMP_LOG(LOG_ERR, "Cannot open volume %s.", volname);
                return (-1);
        }

        if (cleanup_fd == -1 && (cleanup_fd = open(ZFS_DEV,
            O_RDWR|O_EXCL)) < 0) {
                NDMP_LOG(LOG_ERR, "Cannot open dev %d", errno);
                zfs_close(zhp);
                return (-1);
        }

        p = strchr(snapname, '@') + 1;
        if (zfs_hold(zhp, p, jname, recursive, cleanup_fd) != 0) {
        if (zfs_hold(zhp, p, jname, recursive, B_FALSE, cleanup_fd) != 0) {
                NDMP_LOG(LOG_ERR, "Cannot hold snapshot %s", p);
                zfs_close(zhp);
                return (-1);
        }
        zfs_close(zhp);
        return (0);
}
______unchanged_portion_omitted_
```

****
  161499 Fri May 24 00:51:01 2013
new/usr/src/cmd/zfs/zfs_main.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
****
______*unchanged_portion_omitted_*

```
static int
zfs_do_hold_rele_impl(int argc, char **argv, boolean_t holding)
{
        int errors = 0;
        int i;
        const char *tag;
        boolean_t recursive = B_FALSE;
        const char *opts = holding ? "rt" : "r";
        int c;

        /* check options */
        while ((c = getopt(argc, argv, opts)) != -1) {
                switch (c) {
                case 'r':
                        recursive = B_TRUE;
                        break;
                case '?':
                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
                            optopt);
                        usage(B_FALSE);
                }
        }

        argc -= optind;
        argv += optind;

        /* check number of arguments */
        if (argc < 2)
                usage(B_FALSE);

        tag = argv[0];
        --argc;
        ++argv;

        if (holding && tag[0] == '.') {
                /* tags starting with '.' are reserved for libzfs */
                (void) fprintf(stderr, gettext("tag may not start with '.'\n"));
                usage(B_FALSE);
        }

        for (i = 0; i < argc; ++i) {
                zfs_handle_t *zhp;
                char parent[ZFS_MAXNAMELEN];
                const char *delim;
                char *path = argv[i];

                delim = strchr(path, '@');
                if (delim == NULL) {
                        (void) fprintf(stderr,
                            gettext("'%s' is not a snapshot\n"), path);
                        ++errors;
                        continue;
                }
                (void) strncpy(parent, path, delim - path);
                parent[delim - path] = '\0';

                zhp = zfs_open(g_zfs, parent,
                    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME);
```

```
                if (zhp == NULL) {
                        ++errors;
                        continue;
                }
                if (holding) {
                        if (zfs_hold(zhp, delim+1, tag, recursive, -1) != 0)
                        if (zfs_hold(zhp, delim+1, tag, recursive,
                            B_FALSE, -1) != 0)
                                ++errors;
                } else {
                        if (zfs_release(zhp, delim+1, tag, recursive) != 0)
                                ++errors;
                }
                zfs_close(zhp);
        }

        return (errors != 0);
}
```
______*unchanged_portion_omitted_*

****
   26918 Fri May 24 00:51:01 2013
new/usr/src/lib/libzfs/common/libzfs.h
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
****

______***unchanged_portion_omitted_***

```
typedef boolean_t (snapfilter_cb_t)(zfs_handle_t *, void *);

extern int zfs_send(zfs_handle_t *, const char *, const char *,
    sendflags_t *, int, snapfilter_cb_t, void *, nvlist_t **);

extern int zfs_promote(zfs_handle_t *);
extern int zfs_hold(zfs_handle_t *, const char *, const char *,
    boolean_t, int);
extern int zfs_hold_nvl(zfs_handle_t *, int, nvlist_t *);
    boolean_t, boolean_t, int);
extern int zfs_release(zfs_handle_t *, const char *, const char *, boolean_t);
extern int zfs_get_holds(zfs_handle_t *, nvlist_t **);
extern uint64_t zvol_volsize_to_reservation(uint64_t, nvlist_t *);

typedef int (*zfs_userspace_cb_t)(void *arg, const char *domain,
    uid_t rid, uint64_t space);

extern int zfs_userspace(zfs_handle_t *, zfs_userquota_prop_t,
    zfs_userspace_cb_t, void *);

extern int zfs_get_fsacl(zfs_handle_t *, nvlist_t **);
extern int zfs_set_fsacl(zfs_handle_t *, boolean_t, nvlist_t *);

typedef struct recvflags {
        /* print informational messages (ie, -v was specified) */
        boolean_t verbose;

        /* the destination is a prefix, not the exact fs (ie, -d) */
        boolean_t isprefix;

        /*
         * Only the tail of the sent snapshot path is appended to the
         * destination to determine the received snapshot name (ie, -e).
         */
        boolean_t istail;

        /* do not actually do the recv, just check if it would work (ie, -n) */
        boolean_t dryrun;

        /* rollback/destroy filesystems as necessary (eg, -F) */
        boolean_t force;

        /* set "canmount=off" on all modified filesystems */
        boolean_t canmountoff;

        /* byteswap flag is used internally; callers need not specify */
        boolean_t byteswap;

        /* do not mount file systems as they are extracted (private) */
        boolean_t nomount;
} recvflags_t;
```

______***unchanged_portion_omitted_***

```
**********************************************************
  110822 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs/common/libzfs_dataset.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************
______unchanged_portion_omitted_

static int
zfs_hold_one(zfs_handle_t *zhp, void *arg)
{
        struct holdarg *ha = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

        (void) snprintf(name, sizeof (name),
            "%s@%s", zhp->zfs_name, ha->snapname);

        if (lzc_exists(name))
        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                fnvlist_add_string(ha->nvl, name, ha->tag);
                zfs_close(szhp);
        }

        if (ha->recursive)
                rv = zfs_iter_filesystems(zhp, zfs_hold_one, ha);
        zfs_close(zhp);
        return (rv);
}

int
zfs_hold(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive, int cleanup_fd)
    boolean_t recursive, boolean_t enoent_ok, int cleanup_fd)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_hold_one(zfs_handle_dup(zhp), &ha);
        ret = zfs_hold_nvl(zhp, cleanup_fd, ha.nvl);
        ret = lzc_hold(ha.nvl, cleanup_fd, &errors);
        fnvlist_free(ha.nvl);

        return (ret);
}

int
zfs_hold_nvl(zfs_handle_t *zhp, int cleanup_fd, nvlist_t *holds)
{
        int ret;
        nvlist_t *errors;
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char errbuf[1024];
        nvpair_t *elem;

        errors = NULL;
```

```
        ret = lzc_hold(holds, cleanup_fd, &errors);

        if (ret == 0) {
                /* There may be errors even in the success case. */
                fnvlist_free(errors);
        if (ret == 0)
                return (0);
        }
#endif /* ! codereview */

        if (nvlist_next_nvpair(errors, NULL) == NULL) {
                /* no hold-specific errors */
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot hold"));
                switch (ret) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error(hdl, ret, errbuf);
                }
        }

        for (elem = nvlist_next_nvpair(errors, NULL);
            elem != NULL;
            elem = nvlist_next_nvpair(errors, elem)) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot hold snapshot '%s'"), nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case E2BIG:
                        /*
                         * Temporary tags wind up having the ds object id
                         * prepended. So even if we passed the length check
                         * above, it's still possible for the tag to wind
                         * up being slightly too long.
                         */
                        (void) zfs_error(hdl, EZFS_TAGTOOLONG, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case EEXIST:
                        (void) zfs_error(hdl, EZFS_REFTAG_HOLD, errbuf);
                        break;
                case ENOENT:
                        if (enoent_ok)
                                return (ENOENT);
                        /* FALLTHROUGH */
                default:
                        (void) zfs_standard_error(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

struct releasearg {
        nvlist_t *nvl;
        const char *snapname;
        const char *tag;
        boolean_t recursive;
};

static int
zfs_release_one(zfs_handle_t *zhp, void *arg)
{
        struct holdarg *ha = arg;
        zfs_handle_t *szhp;
        char name[ZFS_MAXNAMELEN];
        int rv = 0;

        (void) snprintf(name, sizeof (name),
            "%s@%s", zhp->zfs_name, ha->snapname);

        if (lzc_exists(name)) {
        szhp = make_dataset_handle(zhp->zfs_hdl, name);
        if (szhp) {
                nvlist_t *holds = fnvlist_alloc();
                fnvlist_add_boolean(holds, ha->tag);
                fnvlist_add_nvlist(ha->nvl, name, holds);
                fnvlist_free(holds);
                zfs_close(szhp);
        }

        if (ha->recursive)
                rv = zfs_iter_filesystems(zhp, zfs_release_one, ha);
        zfs_close(zhp);
        return (rv);
}

int
zfs_release(zfs_handle_t *zhp, const char *snapname, const char *tag,
    boolean_t recursive)
{
        int ret;
        struct holdarg ha;
        nvlist_t *errors;
        nvpair_t *elem;
        libzfs_handle_t *hdl = zhp->zfs_hdl;

        ha.nvl = fnvlist_alloc();
        ha.snapname = snapname;
        ha.tag = tag;
        ha.recursive = recursive;
        (void) zfs_release_one(zfs_handle_dup(zhp), &ha);
        errors = NULL;
#endif /* ! codereview */
        ret = lzc_release(ha.nvl, &errors);
        fnvlist_free(ha.nvl);

        if (ret == 0) {
                /* There may be errors even in the success case. */
                fnvlist_free(errors);
        if (ret == 0)
                return (0);
        }
#endif /* ! codereview */

        if (nvlist_next_nvpair(errors, NULL) == NULL) {
                /* no hold-specific errors */
                char errbuf[1024];

                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "cannot release"));
```

```
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl, errno, errbuf);
                }
        }

        for (elem = nvlist_next_nvpair(errors, NULL);
            elem != NULL;
            elem = nvlist_next_nvpair(errors, elem)) {
                char errbuf[1024];

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN,
                    "cannot release hold from snapshot '%s'"),
                    nvpair_name(elem));
                switch (fnvpair_value_int32(elem)) {
                case ESRCH:
                        (void) zfs_error(hdl, EZFS_REFTAG_RELE, errbuf);
                        break;
                case EINVAL:
                        (void) zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                default:
                        (void) zfs_standard_error_fmt(hdl,
                            fnvpair_value_int32(elem), errbuf);
                }
        }

        fnvlist_free(errors);
        return (ret);
}

int
zfs_get_fsacl(zfs_handle_t *zhp, nvlist_t **nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        int nvsz = 2048;
        void *nvbuf;
        int err = 0;
        char errbuf[1024];

        assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
            zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

tryagain:

        nvbuf = malloc(nvsz);
        if (nvbuf == NULL) {
                err = (zfs_error(hdl, EZFS_NOMEM, strerror(errno)));
                goto out;
        }

        zc.zc_nvlist_dst_size = nvsz;
        zc.zc_nvlist_dst = (uintptr_t)nvbuf;

        (void) strlcpy(zc.zc_name, zhp->zfs_name, ZFS_MAXNAMELEN);

        if (ioctl(hdl->libzfs_fd, ZFS_IOC_GET_FSACL, &zc) != 0) {
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get permissions on '%s'"),
                    zc.zc_name);
                switch (errno) {
                case ENOMEM:
                        free(nvbuf);
                        nvsz = zc.zc_nvlist_dst_size;
                        goto tryagain;

                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        } else {
                /* success */
                int rc = nvlist_unpack(nvbuf, zc.zc_nvlist_dst_size, nvl, 0);
                if (rc) {
                        (void) snprintf(errbuf, sizeof (errbuf), dgettext(
                            TEXT_DOMAIN, "cannot get permissions on '%s'"),
                            zc.zc_name);
                        err = zfs_standard_error_fmt(hdl, rc, errbuf);
                }
        }

        free(nvbuf);
out:
        return (err);
}

int
zfs_set_fsacl(zfs_handle_t *zhp, boolean_t un, nvlist_t *nvl)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        char *nvbuf;
        char errbuf[1024];
        size_t nvsz;
        int err;

        assert(zhp->zfs_type == ZFS_TYPE_VOLUME ||
            zhp->zfs_type == ZFS_TYPE_FILESYSTEM);

        err = nvlist_size(nvl, &nvsz, NV_ENCODE_NATIVE);
        assert(err == 0);

        nvbuf = malloc(nvsz);

        err = nvlist_pack(nvl, &nvbuf, &nvsz, NV_ENCODE_NATIVE, 0);
        assert(err == 0);

        zc.zc_nvlist_src_size = nvsz;
        zc.zc_nvlist_src = (uintptr_t)nvbuf;
        zc.zc_perm_action = un;

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));

        if (zfs_ioctl(hdl, ZFS_IOC_SET_FSACL, &zc) != 0) {
```

```
                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot set permissions on '%s'"),
                    zc.zc_name);
                switch (errno) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }

        free(nvbuf);

        return (err);
}

int
zfs_get_holds(zfs_handle_t *zhp, nvlist_t **nvl)
{
        int err;
        char errbuf[1024];

        err = lzc_get_holds(zhp->zfs_name, nvl);

        if (err != 0) {
                libzfs_handle_t *hdl = zhp->zfs_hdl;

                (void) snprintf(errbuf, sizeof (errbuf),
                    dgettext(TEXT_DOMAIN, "cannot get holds for '%s'"),
                    zhp->zfs_name);
                switch (err) {
                case ENOTSUP:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "pool must be upgraded"));
                        err = zfs_error(hdl, EZFS_BADVERSION, errbuf);
                        break;
                case EINVAL:
                        err = zfs_error(hdl, EZFS_BADTYPE, errbuf);
                        break;
                case ENOENT:
                        err = zfs_error(hdl, EZFS_NOENT, errbuf);
                        break;
                default:
                        err = zfs_standard_error_fmt(hdl, errno, errbuf);
                        break;
                }
        }

        return (err);
}

uint64_t
zvol_volsize_to_reservation(uint64_t volsize, nvlist_t *props)
{
        uint64_t numdb;
        uint64_t nblocks, volblocksize;
        int ncopies;
        char *strval;

        if (nvlist_lookup_string(props,
            zfs_prop_to_name(ZFS_PROP_COPIES), &strval) == 0)
                ncopies = atoi(strval);
        else
                ncopies = 1;
        if (nvlist_lookup_uint64(props,
            zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE),
            &volblocksize) != 0)
                volblocksize = ZVOL_DEFAULT_BLOCKSIZE;
        nblocks = volsize/volblocksize;
        /* start with metadnode L0-L6 */
        numdb = 7;
        /* calculate number of indirects */
        while (nblocks > 1) {
                nblocks += DNODES_PER_LEVEL - 1;
                nblocks /= DNODES_PER_LEVEL;
                numdb += nblocks;
        }
        numdb *= MIN(SPA_DVAS_PER_BP, ncopies + 1);
        volsize *= ncopies;
        /*
         * this is exactly DN_MAX_INDBLKSHIFT when metadata isn't
         * compressed, but in practice they compress down to about
         * 1100 bytes
         */
        numdb *= 1ULL << DN_MAX_INDBLKSHIFT;
        volsize += numdb;
        return (volsize);
}
```

```
***********************************************************
   84475 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs/common/libzfs_sendrecv.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
***********************************************************
______unchanged_portion_omitted_

/*
 * Routines specific to "zfs send"
 */
typedef struct send_dump_data {
        /* these are all just the short snapname (the part after the @) */
        const char *fromsnap;
        const char *tosnap;
        char prevsnap[ZFS_MAXNAMELEN];
        uint64_t prevsnap_obj;
        boolean_t seenfrom, seento, replicate, doall, fromorigin;
        boolean_t verbose, dryrun, parsable, progress;
        int outfd;
        boolean_t err;
        nvlist_t *fss;
        nvlist_t *snapholds;
#endif /* ! codereview */
        avl_tree_t *fsavl;
        snapfilter_cb_t *filter_cb;
        void *filter_cb_arg;
        nvlist_t *debugnv;
        char holdtag[ZFS_MAXNAMELEN];
        int cleanup_fd;
        uint64_t size;
} send_dump_data_t;

static int
estimate_ioctl(zfs_handle_t *zhp, uint64_t fromsnap_obj,
    boolean_t fromorigin, uint64_t *sizep)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
        assert(fromsnap_obj == 0 || !fromorigin);

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
        zc.zc_obj = fromorigin;
        zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
        zc.zc_fromobj = fromsnap_obj;
        zc.zc_guid = 1;  /* estimate flag */

        if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
                char errbuf[1024];
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "warning: cannot estimate space for '%s'"), zhp->zfs_name);

                switch (errno) {
                case EXDEV:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "not an earlier snapshot from the same fs"));
                        return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

                case ENOENT:
                        if (zfs_dataset_exists(hdl, zc.zc_name,
                            ZFS_TYPE_SNAPSHOT)) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "incremental source (@%s) does not exist"),
                                    zc.zc_value);
```

```
                        }
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

                case EDQUOT:
                case EFBIG:
                case EIO:
                case ENOLINK:
                case ENOSPC:
                case ENOSTR:
                case ENXIO:
                case EPIPE:
                case ERANGE:
                case EFAULT:
                case EROFS:
                        zfs_error_aux(hdl, strerror(errno));
                        return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

                default:
                        return (zfs_standard_error(hdl, errno, errbuf));
                }
        }

        *sizep = zc.zc_objset_type;

        return (0);
}

/*
 * Dumps a backup of the given snapshot (incremental from fromsnap if it's not
 * NULL) to the file descriptor specified by outfd.
 */
static int
dump_ioctl(zfs_handle_t *zhp, const char *fromsnap, uint64_t fromsnap_obj,
    boolean_t fromorigin, int outfd, nvlist_t *debugnv)
{
        zfs_cmd_t zc = { 0 };
        libzfs_handle_t *hdl = zhp->zfs_hdl;
        nvlist_t *thisdbg;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);
        assert(fromsnap_obj == 0 || !fromorigin);

        (void) strlcpy(zc.zc_name, zhp->zfs_name, sizeof (zc.zc_name));
        zc.zc_cookie = outfd;
        zc.zc_obj = fromorigin;
        zc.zc_sendobj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
        zc.zc_fromobj = fromsnap_obj;

        VERIFY(0 == nvlist_alloc(&thisdbg, NV_UNIQUE_NAME, 0));
        if (fromsnap && fromsnap[0] != '\0') {
                VERIFY(0 == nvlist_add_string(thisdbg,
                    "fromsnap", fromsnap));
        }

        if (zfs_ioctl(zhp->zfs_hdl, ZFS_IOC_SEND, &zc) != 0) {
                char errbuf[1024];
                (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
                    "warning: cannot send '%s'"), zhp->zfs_name);

                VERIFY(0 == nvlist_add_uint64(thisdbg, "error", errno));
                if (debugnv) {
                        VERIFY(0 == nvlist_add_nvlist(debugnv,
                            zhp->zfs_name, thisdbg));
                }
                nvlist_free(thisdbg);
                switch (errno) {
                case EXDEV:
                        zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                            "not an earlier snapshot from the same fs"));
                        return (zfs_error(hdl, EZFS_CROSSTARGET, errbuf));

                case ENOENT:
                        if (zfs_dataset_exists(hdl, zc.zc_name,
                            ZFS_TYPE_SNAPSHOT)) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "incremental source (@%s) does not exist"),
                                    zc.zc_value);
                        }
                        return (zfs_error(hdl, EZFS_NOENT, errbuf));

                case EDQUOT:
                case EFBIG:
                case EIO:
                case ENOLINK:
                case ENOSPC:
                case ENOSTR:
                case ENXIO:
                case EPIPE:
                case ERANGE:
                case EFAULT:
                case EROFS:
                        zfs_error_aux(hdl, strerror(errno));
                        return (zfs_error(hdl, EZFS_BADBACKUP, errbuf));

                default:
                        return (zfs_standard_error(hdl, errno, errbuf));
                }
        }

        if (debugnv)
                VERIFY(0 == nvlist_add_nvlist(debugnv, zhp->zfs_name, thisdbg));
        nvlist_free(thisdbg);

        return (0);
}

static void
gather_holds(zfs_handle_t *zhp, send_dump_data_t *sdd)
static int
hold_for_send(zfs_handle_t *zhp, send_dump_data_t *sdd)
{
        zfs_handle_t *pzhp;
        int error = 0;
        char *thissnap;

        assert(zhp->zfs_type == ZFS_TYPE_SNAPSHOT);

        if (sdd->dryrun)
                return (0);

        /*
         * zfs_send() only sets snapholds for sends that need them,
         * zfs_send() only opens a cleanup_fd for sends that need it,
         * e.g. replication and doall.
         */
        if (sdd->snapholds == NULL)
                return;
        if (sdd->cleanup_fd == -1)
                return (0);

        thissnap = strchr(zhp->zfs_name, '@') + 1;
```

```
        *(thissnap - 1) = '\0';
        pzhp = zfs_open(zhp->zfs_hdl, zhp->zfs_name, ZFS_TYPE_DATASET);
        *(thissnap - 1) = '@';

        /*
         * It's OK if the parent no longer exists.  The send code will
         * handle that error.
         */
        if (pzhp) {
                error = zfs_hold(pzhp, thissnap, sdd->holdtag,
                    B_FALSE, B_TRUE, sdd->cleanup_fd);
                zfs_close(pzhp);
        }

        fnvlist_add_string(sdd->snapholds, zhp->zfs_name, sdd->holdtag);
        return (error);
}
______unchanged_portion_omitted_

static int
dump_snapshot(zfs_handle_t *zhp, void *arg)
{
        send_dump_data_t *sdd = arg;
        progress_arg_t pa = { 0 };
        pthread_t tid;

        char *thissnap;
        int err;
        boolean_t isfromsnap, istosnap, fromorigin;
        boolean_t exclude = B_FALSE;

        err = 0;
#endif /* ! codereview */
        thissnap = strchr(zhp->zfs_name, '@') + 1;
        isfromsnap = (sdd->fromsnap != NULL &&
            strcmp(sdd->fromsnap, thissnap) == 0);

        if (!sdd->seenfrom && isfromsnap) {
                gather_holds(zhp, sdd);
                err = hold_for_send(zhp, sdd);
                if (err == 0) {
                sdd->seenfrom = B_TRUE;
                (void) strcpy(sdd->prevsnap, thissnap);
                sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
                        sdd->prevsnap_obj = zfs_prop_get_int(zhp,
                            ZFS_PROP_OBJSETID);
                } else if (err == ENOENT) {
                        err = 0;
                }
                zfs_close(zhp);
                return (0);
                return (err);
        }

        if (sdd->seento || !sdd->seenfrom) {
                zfs_close(zhp);
                return (0);
        }

        istosnap = (strcmp(sdd->tosnap, thissnap) == 0);
        if (istosnap)
                sdd->seento = B_TRUE;

        if (!sdd->doall && !isfromsnap && !istosnap) {
                if (sdd->replicate) {
                        char *snapname;
```

```
			nvlist_t *snapprops;
			/*
			 * Filter out all intermediate snapshots except origin
			 * snapshots needed to replicate clones.
			 */
			nvlist_t *nvfs = fsavl_find(sdd->fsavl,
			    zhp->zfs_dmustats.dds_guid, &snapname);

			VERIFY(0 == nvlist_lookup_nvlist(nvfs,
			    "snapprops", &snapprops));
			VERIFY(0 == nvlist_lookup_nvlist(snapprops,
			    thissnap, &snapprops));
			exclude = !nvlist_exists(snapprops, "is_clone_origin");
		} else {
			exclude = B_TRUE;
		}
	}

	/*
	 * If a filter function exists, call it to determine whether
	 * this snapshot will be sent.
	 */
	if (exclude || (sdd->filter_cb != NULL &&
	    sdd->filter_cb(zhp, sdd->filter_cb_arg) == B_FALSE)) {
		/*
		 * This snapshot is filtered out.  Don't send it, and don't
		 * set prevsnap_obj, so it will be as if this snapshot didn't
		 * exist, and the next accepted snapshot will be sent as
		 * an incremental from the last accepted one, or as the
		 * first (and full) snapshot in the case of a replication,
		 * non-incremental send.
		 */
		zfs_close(zhp);
		return (0);
	}

	gather_holds(zhp, sdd);
	err = hold_for_send(zhp, sdd);
	if (err) {
		if (err == ENOENT)
			err = 0;
		zfs_close(zhp);
		return (err);
	}

	fromorigin = sdd->prevsnap[0] == '\0' &&
	    (sdd->fromorigin || sdd->replicate);

	if (sdd->verbose) {
		uint64_t size;
		err = estimate_ioctl(zhp, sdd->prevsnap_obj,
		    fromorigin, &size);

		if (sdd->parsable) {
			if (sdd->prevsnap[0] != '\0') {
				(void) fprintf(stderr, "incremental\t%s\t%s",
				    sdd->prevsnap, zhp->zfs_name);
			} else {
				(void) fprintf(stderr, "full\t%s",
				    zhp->zfs_name);
			}
		} else {
			(void) fprintf(stderr, dgettext(TEXT_DOMAIN,
			    "send from @%s to %s"),
			    sdd->prevsnap, zhp->zfs_name);
		}
```

```
		if (err == 0) {
			if (sdd->parsable) {
				(void) fprintf(stderr, "\t%llu\n",
				    (longlong_t)size);
			} else {
				char buf[16];
				zfs_nicenum(size, buf, sizeof (buf));
				(void) fprintf(stderr, dgettext(TEXT_DOMAIN,
				    " estimated size is %s\n"), buf);
			}
			sdd->size += size;
		} else {
			(void) fprintf(stderr, "\n");
		}
	}

	if (!sdd->dryrun) {
		/*
		 * If progress reporting is requested, spawn a new thread to
		 * poll ZFS_IOC_SEND_PROGRESS at a regular interval.
		 */
		if (sdd->progress) {
			pa.pa_zhp = zhp;
			pa.pa_fd = sdd->outfd;
			pa.pa_parsable = sdd->parsable;

			if (err = pthread_create(&tid, NULL,
			    send_progress_thread, &pa)) {
				zfs_close(zhp);
				return (err);
			}
		}

		err = dump_ioctl(zhp, sdd->prevsnap, sdd->prevsnap_obj,
		    fromorigin, sdd->outfd, sdd->debugnv);

		if (sdd->progress) {
			(void) pthread_cancel(tid);
			(void) pthread_join(tid, NULL);
		}
	}

	(void) strcpy(sdd->prevsnap, thissnap);
	sdd->prevsnap_obj = zfs_prop_get_int(zhp, ZFS_PROP_OBJSETID);
	zfs_close(zhp);
	return (err);
}
______unchanged_portion_omitted_

/*
 * Generate a send stream for the dataset identified by the argument zhp.
 *
 * The content of the send stream is the snapshot identified by
 * 'tosnap'.  Incremental streams are requested in two ways:
 *     - from the snapshot identified by "fromsnap" (if non-null) or
 *     - from the origin of the dataset identified by zhp, which must
 *       be a clone.  In this case, "fromsnap" is null and "fromorigin"
 *       is TRUE.
 *
 * The send stream is recursive (i.e. dumps a hierarchy of snapshots) and
 * uses a special header (with a hdrtype field of DMU_COMPOUNDSTREAM)
 * if "replicate" is set.  If "doall" is set, dump all the intermediate
 * snapshots. The DMU_COMPOUNDSTREAM header is used in the "doall"
 * case too. If "props" is set, send properties.
 */
int
```

```
zfs_send(zfs_handle_t *zhp, const char *fromsnap, const char *tosnap,
    sendflags_t *flags, int outfd, snapfilter_cb_t filter_func,
    void *cb_arg, nvlist_t **debugnvp)
{
        char errbuf[1024];
        send_dump_data_t sdd = { 0 };
        int err = 0;
        nvlist_t *fss = NULL;
        avl_tree_t *fsavl = NULL;
        static uint64_t holdseq;
        int spa_version;
        pthread_t tid = 0;
        pthread_t tid;
        int pipefd[2];
        dedup_arg_t dda = { 0 };
        int featureflags = 0;

        (void) snprintf(errbuf, sizeof (errbuf), dgettext(TEXT_DOMAIN,
            "cannot send '%s'"), zhp->zfs_name);

        if (fromsnap && fromsnap[0] == '\0') {
                zfs_error_aux(zhp->zfs_hdl, dgettext(TEXT_DOMAIN,
                    "zero-length incremental source"));
                return (zfs_error(zhp->zfs_hdl, EZFS_NOENT, errbuf));
        }

        if (zhp->zfs_type == ZFS_TYPE_FILESYSTEM) {
                uint64_t version;
                version = zfs_prop_get_int(zhp, ZFS_PROP_VERSION);
                if (version >= ZPL_VERSION_SA) {
                        featureflags |= DMU_BACKUP_FEATURE_SA_SPILL;
                }
        }

        if (flags->dedup && !flags->dryrun) {
                featureflags |= (DMU_BACKUP_FEATURE_DEDUP |
                    DMU_BACKUP_FEATURE_DEDUPPROPS);
                if (err = pipe(pipefd)) {
                        zfs_error_aux(zhp->zfs_hdl, strerror(errno));
                        return (zfs_error(zhp->zfs_hdl, EZFS_PIPEFAILED,
                            errbuf));
                }
                dda.outputfd = outfd;
                dda.inputfd = pipefd[1];
                dda.dedup_hdl = zhp->zfs_hdl;
                if (err = pthread_create(&tid, NULL, cksummer, &dda)) {
                        (void) close(pipefd[0]);
                        (void) close(pipefd[1]);
                        zfs_error_aux(zhp->zfs_hdl, strerror(errno));
                        return (zfs_error(zhp->zfs_hdl,
                            EZFS_THREADCREATEFAILED, errbuf));
                }
        }

        if (flags->replicate || flags->doall || flags->props) {
                dmu_replay_record_t drr = { 0 };
                char *packbuf = NULL;
                size_t buflen = 0;
                zio_cksum_t zc = { 0 };

                if (flags->replicate || flags->props) {
                        nvlist_t *hdrnv;

                        VERIFY(0 == nvlist_alloc(&hdrnv, NV_UNIQUE_NAME, 0));
                        if (fromsnap) {
                                VERIFY(0 == nvlist_add_string(hdrnv,
```

```
                                    "fromsnap", fromsnap));
                        }
                        VERIFY(0 == nvlist_add_string(hdrnv, "tosnap", tosnap));
                        if (!flags->replicate) {
                                VERIFY(0 == nvlist_add_boolean(hdrnv,
                                    "not_recursive"));
                        }

                        err = gather_nvlist(zhp->zfs_hdl, zhp->zfs_name,
                            fromsnap, tosnap, flags->replicate, &fss, &fsavl);
                        if (err)
                                goto err_out;
                        VERIFY(0 == nvlist_add_nvlist(hdrnv, "fss", fss));
                        err = nvlist_pack(hdrnv, &packbuf, &buflen,
                            NV_ENCODE_XDR, 0);
                        if (debugnvp)
                                *debugnvp = hdrnv;
                        else
                                nvlist_free(hdrnv);
                        if (err)
                        if (err) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                goto stderr_out;
                }
                }

                if (!flags->dryrun) {
                        /* write first begin record */
                        drr.drr_type = DRR_BEGIN;
                        drr.drr_u.drr_begin.drr_magic = DMU_BACKUP_MAGIC;
                        DMU_SET_STREAM_HDRTYPE(drr.drr_u.drr_begin.
                            drr_versioninfo, DMU_COMPOUNDSTREAM);
                        DMU_SET_FEATUREFLAGS(drr.drr_u.drr_begin.
                            drr_versioninfo, featureflags);
                        (void) snprintf(drr.drr_u.drr_begin.drr_toname,
                            sizeof (drr.drr_u.drr_begin.drr_toname),
                            "%s@%s", zhp->zfs_name, tosnap);
                        drr.drr_payloadlen = buflen;
                        err = cksum_and_write(&drr, sizeof (drr), &zc, outfd);

                        /* write header nvlist */
                        if (err != -1 && packbuf != NULL) {
                                err = cksum_and_write(packbuf, buflen, &zc,
                                    outfd);
                        }
                        free(packbuf);
                        if (err == -1) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                err = errno;
                                goto stderr_out;
                        }

                        /* write end record */
                        bzero(&drr, sizeof (drr));
                        drr.drr_type = DRR_END;
                        drr.drr_u.drr_end.drr_checksum = zc;
                        err = write(outfd, &drr, sizeof (drr));
                        if (err == -1) {
                                fsavl_destroy(fsavl);
                                nvlist_free(fss);
                                err = errno;
                                goto stderr_out;
                        }
                        err = 0;
                }
        }

        /* dump each stream */
        sdd.fromsnap = fromsnap;
        sdd.tosnap = tosnap;
        if (tid != 0)
        if (flags->dedup)
                sdd.outfd = pipefd[0];
        else
                sdd.outfd = outfd;
        sdd.replicate = flags->replicate;
        sdd.doall = flags->doall;
        sdd.fromorigin = flags->fromorigin;
        sdd.fss = fss;
        sdd.fsavl = fsavl;
        sdd.verbose = flags->verbose;
        sdd.parsable = flags->parsable;
        sdd.progress = flags->progress;
        sdd.dryrun = flags->dryrun;
        sdd.filter_cb = filter_func;
        sdd.filter_cb_arg = cb_arg;
        if (debugnvp)
                sdd.debugnv = *debugnvp;

        /*
         * Some flags require that we place user holds on the datasets that are
         * being sent so they don't get destroyed during the send. We can skip
         * this step if the pool is imported read-only since the datasets cannot
         * be destroyed.
         */
        if (!flags->dryrun && !zpool_get_prop_int(zfs_get_pool_handle(zhp),
            ZPOOL_PROP_READONLY, NULL) &&
            zfs_spa_version(zhp, &spa_version) == 0 &&
            spa_version >= SPA_VERSION_USERREFS &&
            (flags->doall || flags->replicate)) {
                ++holdseq;
                (void) snprintf(sdd.holdtag, sizeof (sdd.holdtag),
                    ".send-%d-%llu", getpid(), (u_longlong_t)holdseq);
                sdd.cleanup_fd = open(ZFS_DEV, O_RDWR|O_EXCL);
                if (sdd.cleanup_fd < 0) {
                        err = errno;
                        goto stderr_out;
                }
                sdd.snapholds = fnvlist_alloc();
#endif /* ! codereview */
        } else {
                sdd.cleanup_fd = -1;
                sdd.snapholds = NULL;
#endif /* ! codereview */
        }
        if (flags->verbose || sdd.snapholds != NULL) {
        if (flags->verbose) {
                /*
                 * Do a verbose no-op dry run to get all the verbose output
                 * or to gather snapshot hold's before generating any data,
                 * then do a non-verbose real run to generate the streams.
                 * before generating any data.  Then do a non-verbose real
                 * run to generate the streams.
                 */
                sdd.dryrun = B_TRUE;
                err = dump_filesystems(zhp, &sdd);

                if (err != 0)
                        goto stderr_out;
```

```
                if (flags->verbose) {
                sdd.dryrun = flags->dryrun;
                sdd.verbose = B_FALSE;
                        if (flags->parsable) {
                                (void) fprintf(stderr, "size\t%llu\n",
                                    (longlong_t)sdd.size);
                        } else {
                                char buf[16];
                                zfs_nicenum(sdd.size, buf, sizeof (buf));
                                (void) fprintf(stderr, dgettext(TEXT_DOMAIN,
                                    "total estimated size is %s\n"), buf);
                        }
                }

                /* Ensure no snaps found is treated as an error. */
                if (!sdd.seento) {
                        err = ENOENT;
                        goto err_out;
                }

                /* Skip the second run if dryrun was requested. */
                if (flags->dryrun)
                        goto err_out;

                if (sdd.snapholds != NULL) {
                        err = zfs_hold_nvl(zhp, sdd.cleanup_fd, sdd.snapholds);
                        if (err != 0)
                                goto stderr_out;
                        fnvlist_free(sdd.snapholds);
                        sdd.snapholds = NULL;
                }

                sdd.dryrun = B_FALSE;
                sdd.verbose = B_FALSE;
        }

#endif /* ! codereview */
        err = dump_filesystems(zhp, &sdd);
        fsavl_destroy(fsavl);
        nvlist_free(fss);

        /* Ensure no snaps found is treated as an error. */
        if (err == 0 && !sdd.seento)
                err = ENOENT;

        if (tid != 0) {
                if (err != 0)
                        (void) pthread_cancel(tid);
                (void) pthread_join(tid, NULL);
        if (flags->dedup) {
                (void) close(pipefd[0]);
                (void) pthread_join(tid, NULL);
        }

        if (sdd.cleanup_fd != -1) {
                VERIFY(0 == close(sdd.cleanup_fd));
                sdd.cleanup_fd = -1;
        }

        if (!flags->dryrun && (flags->replicate || flags->doall ||
            flags->props)) {
                /*
                 * write final end record.  NB: want to do this even if
                 * there was some error, because it might not be totally
                 * failed.
```

```
		 */
		dmu_replay_record_t drr = { 0 };
		drr.drr_type = DRR_END;
		if (write(outfd, &drr, sizeof (drr)) == -1) {
			return (zfs_standard_error(zhp->zfs_hdl,
			    errno, errbuf));
		}
	}

	return (err || sdd.err);

stderr_out:
	err = zfs_standard_error(zhp->zfs_hdl, err, errbuf);
err_out:
	fsavl_destroy(fsavl);
	nvlist_free(fss);
	fnvlist_free(sdd.snapholds);

#endif /* ! codereview */
	if (sdd.cleanup_fd != -1)
		VERIFY(0 == close(sdd.cleanup_fd));
	if (tid != 0) {
	if (flags->dedup) {
		(void) pthread_cancel(tid);
		(void) pthread_join(tid, NULL);
		(void) close(pipefd[0]);
	}
	return (err);
}
```
______***unchanged_portion_omitted_***

****
   17044 Fri May 24 00:51:02 2013
new/usr/src/lib/libzfs_core/common/libzfs_core.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
****

______unchanged_portion_omitted_

```
/*
 * Destroys snapshots.
 *
 * The keys in the snaps nvlist are the snapshots to be destroyed.
 * They must all be in the same pool.
 *
 * Snapshots that do not exist will be silently ignored.
 *
 * If 'defer' is not set, and a snapshot has user holds or clones, the
 * destroy operation will fail and none of the snapshots will be
 * destroyed.
 *
 * If 'defer' is set, and a snapshot has user holds or clones, it will be
 * marked for deferred destruction, and will be destroyed when the last hold
 * or clone is removed/destroyed.
 *
 * The return value will be ENOENT if none of the snapshots existed.
 *
#endif /* ! codereview */
 * The return value will be 0 if all snapshots were destroyed (or marked for
 * later destruction if 'defer' is set) or didn't exist to begin with and
 * at least one snapshot was destroyed.
 * later destruction if 'defer' is set) or didn't exist to begin with.
 *
 * Otherwise the return value will be the errno of a (unspecified) snapshot
 * that failed, no snapshots will be destroyed, and the errlist will have an
 * entry for each snapshot that failed.  The value in the errlist will be
 * the (int32) error code.
 */
int
lzc_destroy_snaps(nvlist_t *snaps, boolean_t defer, nvlist_t **errlist)
{
        nvpair_t *elem;
        nvlist_t *args;
        int error;
        char pool[MAXNAMELEN];

        /* determine the pool name */
        elem = nvlist_next_nvpair(snaps, NULL);
        if (elem == NULL)
                return (0);
        (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
        pool[strcspn(pool, "/@")] = '\0';

        args = fnvlist_alloc();
        fnvlist_add_nvlist(args, "snaps", snaps);
        if (defer)
                fnvlist_add_boolean(args, "defer");

        error = lzc_ioctl(ZFS_IOC_DESTROY_SNAPS, pool, args, errlist);
        nvlist_free(args);

        return (error);

}
```

______unchanged_portion_omitted_

```
/*
```

```
 * Create "user holds" on snapshots.  If there is a hold on a snapshot,
 * the snapshot can not be destroyed.  (However, it can be marked for deletion
 * by lzc_destroy_snaps(defer=B_TRUE).)
 *
 * The keys in the nvlist are snapshot names.
 * The snapshots must all be in the same pool.
 * The value is the name of the hold (string type).
 *
 * If cleanup_fd is not -1, it must be the result of open("/dev/zfs", O_EXCL).
 * In this case, when the cleanup_fd is closed (including on process
 * termination), the holds will be released.  If the system is shut down
 * uncleanly, the holds will be released when the pool is next opened
 * or imported.
 *
 * Holds for snapshots which don't exist will be skipped and have an entry
 * added to errlist, but will not cause an overall failure, except in the
 * case that all holds where skipped.
 *
 * The return value will be ENOENT if none of the snapshots for the requested
 * holds existed.
 *
 * The return value will be 0 if the nvl holds was empty or all holds, for
 * snapshots that existed, were succesfully created and at least one hold
 * was created.
 *
 * Otherwise the return value will be the errno of a (unspecified) hold that
 * failed and no holds will be created.
 *
 * In all cases the errlist will have an entry for each hold that failed
 * (name = snapshot), with its value being the error code (int32).
 * The return value will be 0 if all holds were created. Otherwise the return
 * value will be the errno of a (unspecified) hold that failed, no holds will
 * be created, and the errlist will have an entry for each hold that
 * failed (name = snapshot).  The value in the errlist will be the error
 * code (int32).
 */
int
lzc_hold(nvlist_t *holds, int cleanup_fd, nvlist_t **errlist)
{
        char pool[MAXNAMELEN];
        nvlist_t *args;
        nvpair_t *elem;
        int error;

        /* determine the pool name */
        elem = nvlist_next_nvpair(holds, NULL);
        if (elem == NULL)
                return (0);
        (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
        pool[strcspn(pool, "/@")] = '\0';

        args = fnvlist_alloc();
        fnvlist_add_nvlist(args, "holds", holds);
        if (cleanup_fd != -1)
                fnvlist_add_int32(args, "cleanup_fd", cleanup_fd);

        error = lzc_ioctl(ZFS_IOC_HOLD, pool, args, errlist);
        nvlist_free(args);
        return (error);
}

/*
 * Release "user holds" on snapshots.  If the snapshot has been marked for
 * deferred destroy (by lzc_destroy_snaps(defer=B_TRUE)), it does not have
 * any clones, and all the user holds are removed, then the snapshot will be
 * destroyed.
 *
 * The keys in the nvlist are snapshot names.
 * The snapshots must all be in the same pool.
 * The value is a nvlist whose keys are the holds to remove.
 *
 * Holds which failed to release because they didn't exist will have an entry
 * added to errlist, but will not cause an overall failure, except in the
 * case that all releases where skipped.
 *
 * The return value will be ENOENT if none of the specified holds existed.
 *
 * The return value will be 0 if the nvl holds was empty or all holds, that
 * existed, were succesfully removed and at least one hold was removed.
 *
 * Otherwise the return value will be the errno of a (unspecified) hold that
 * failed to release and no holds will be released.
 *
 * In all cases the errlist will have an entry for each hold that failed to
 * to release.
 * The return value will be 0 if all holds were removed.
 * Otherwise the return value will be the errno of a (unspecified) release
 * that failed, no holds will be released, and the errlist will have an
 * entry for each snapshot that has failed releases (name = snapshot).
 * The value in the errlist will be the error code (int32) of a failed release.
 */
int
lzc_release(nvlist_t *holds, nvlist_t **errlist)
{
        char pool[MAXNAMELEN];
        nvpair_t *elem;

        /* determine the pool name */
        elem = nvlist_next_nvpair(holds, NULL);
        if (elem == NULL)
                return (0);
        (void) strlcpy(pool, nvpair_name(elem), sizeof (pool));
        pool[strcspn(pool, "/@")] = '\0';

        return (lzc_ioctl(ZFS_IOC_RELEASE, pool, holds, errlist));
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   29735 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************
______unchanged_portion_omitted_

/*
 * Walk through the pool-wide zap object of temporary snapshot user holds
 * and release them.
 */
void
dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp)
{
        zap_attribute_t za;
        zap_cursor_t zc;
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj = dp->dp_tmp_userrefs_obj;
        nvlist_t *holds;
#endif /* ! codereview */

        if (zapobj == 0)
                return;
        ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);

        holds = fnvlist_alloc();

#endif /* ! codereview */
        for (zap_cursor_init(&zc, mos, zapobj);
            zap_cursor_retrieve(&zc, &za) == 0;
            zap_cursor_advance(&zc)) {
                char *htag;
                uint64_t dsobj;
                nvlist_t *tags;
#endif /* ! codereview */

                htag = strchr(za.za_name, '-');
                *htag = '\0';
                ++htag;
                if (nvlist_lookup_nvlist(holds, za.za_name, &tags) != 0) {
                        tags = fnvlist_alloc();
                        fnvlist_add_boolean(tags, htag);
                        fnvlist_add_nvlist(holds, za.za_name, tags);
                        fnvlist_free(tags);
                } else {
                        fnvlist_add_boolean(tags, htag);
                }
                dsobj = strtonum(za.za_name, NULL);
                dsl_dataset_user_release_tmp(dp, dsobj, htag);
        }
        dsl_dataset_user_release_tmp(dp, holds);
        fnvlist_free(holds);
#endif /* ! codereview */
        zap_cursor_fini(&zc);
}

/*
 * Create the pool-wide zap object for storing temporary snapshot holds.
 */
void
dsl_pool_user_hold_create_obj(dsl_pool_t *dp, dmu_tx_t *tx)
{
        objset_t *mos = dp->dp_meta_objset;

        ASSERT(dp->dp_tmp_userrefs_obj == 0);
```

```
        ASSERT(dmu_tx_is_syncing(tx));

        dp->dp_tmp_userrefs_obj = zap_create_link(mos, DMU_OT_USERREFS,
            DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_TMP_USERREFS, tx);
}

static int
dsl_pool_user_hold_rele_impl(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, uint64_t now, dmu_tx_t *tx, boolean_t holding)
{
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj = dp->dp_tmp_userrefs_obj;
        char *name;
        int error;

        ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
        ASSERT(dmu_tx_is_syncing(tx));

        /*
         * If the pool was created prior to SPA_VERSION_USERREFS, the
         * zap object for temporary holds might not exist yet.
         */
        if (zapobj == 0) {
                if (holding) {
                        dsl_pool_user_hold_create_obj(dp, tx);
                        zapobj = dp->dp_tmp_userrefs_obj;
                } else {
                        return (SET_ERROR(ENOENT));
                }
        }

        name = kmem_asprintf("%llx-%s", (u_longlong_t)dsobj, tag);
        if (holding)
                error = zap_add(mos, zapobj, name, 8, 1, &now, tx);
        else
                error = zap_remove(mos, zapobj, name, tx);
        strfree(name);

        return (error);
}

/*
 * Add a temporary hold for the given dataset object and tag.
 */
int
dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
    uint64_t now, dmu_tx_t *tx)
{
        return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, now, tx, B_TRUE));
}

/*
 * Release a temporary hold for the given dataset object and tag.
 */
int
dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj, const char *tag,
    dmu_tx_t *tx)
{
        return (dsl_pool_user_hold_rele_impl(dp, dsobj, tag, NULL,
            tx, B_FALSE));
}

/*
 * DSL Pool Configuration Lock
 *
 * The dp_config_rwlock protects against changes to DSL state (e.g. dataset
 * creation / destruction / rename / property setting).  It must be held for
 * read to hold a dataset or dsl_dir.  I.e. you must call
 * dsl_pool_config_enter() or dsl_pool_hold() before calling
 * dsl_{dataset,dir}_hold{_obj}.  In most circumstances, the dp_config_rwlock
 * must be held continuously until all datasets and dsl_dirs are released.
 *
 * The only exception to this rule is that if a "long hold" is placed on
 * a dataset, then the dp_config_rwlock may be dropped while the dataset
 * is still held.  The long hold will prevent the dataset from being
 * destroyed -- the destroy will fail with EBUSY.  A long hold can be
 * obtained by calling dsl_dataset_long_hold(), or by "owning" a dataset
 * (by calling dsl_{dataset,objset}_{try}own{_obj}).
 *
 * Legitimate long-holders (including owners) should be long-running, cancelable
 * tasks that should cause "zfs destroy" to fail.  This includes DMU
 * consumers (i.e. a ZPL filesystem being mounted or ZVOL being open),
 * "zfs send", and "zfs diff".  There are several other long-holders whose
 * uses are suboptimal (e.g. "zfs promote", and zil_suspend()).
 *
 * The usual formula for long-holding would be:
 * dsl_pool_hold()
 * dsl_dataset_hold()
 * ... perform checks ...
 * dsl_dataset_long_hold()
 * dsl_pool_rele()
 * ... perform long-running task ...
 * dsl_dataset_long_rele()
 * dsl_dataset_rele()
 *
 * Note that when the long hold is released, the dataset is still held but
 * the pool is not held.  The dataset may change arbitrarily during this time
 * (e.g. it could be destroyed).  Therefore you shouldn't do anything to the
 * dataset except release it.
 *
 * User-initiated operations (e.g. ioctls, zfs_ioc_*()) are either read-only
 * or modifying operations.
 *
 * Modifying operations should generally use dsl_sync_task().  The synctask
 * infrastructure enforces proper locking strategy with respect to the
 * dp_config_rwlock.  See the comment above dsl_sync_task() for details.
 *
 * Read-only operations will manually hold the pool, then the dataset, obtain
 * information from the dataset, then release the pool and dataset.
 * dmu_objset_{hold,rele}() are convenience routines that also do the pool
 * hold/rele.
 */

int
dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp)
{
        spa_t *spa;
        int error;

        error = spa_open(name, &spa, tag);
        if (error == 0) {
                *dp = spa_get_dsl(spa);
                dsl_pool_config_enter(*dp, tag);
        }
        return (error);
}

void
dsl_pool_rele(dsl_pool_t *dp, void *tag)
{
        dsl_pool_config_exit(dp, tag);
        spa_close(dp->dp_spa, tag);
```

```
}

void
dsl_pool_config_enter(dsl_pool_t *dp, void *tag)
{
        /*
         * We use a "reentrant" reader-writer lock, but not reentrantly.
         *
         * The rrwlock can (with the track_all flag) track all reading threads,
         * which is very useful for debugging which code path failed to release
         * the lock, and for verifying that the *current* thread does hold
         * the lock.
         *
         * (Unlike a rwlock, which knows that N threads hold it for
         * read, but not *which* threads, so rw_held(RW_READER) returns TRUE
         * if any thread holds it for read, even if this thread doesn't).
         */
        ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
        rrw_enter(&dp->dp_config_rwlock, RW_READER, tag);
}

void
dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
{
        rrw_exit(&dp->dp_config_rwlock, tag);
}

boolean_t
dsl_pool_config_held(dsl_pool_t *dp)
{
        return (RRW_LOCK_HELD(&dp->dp_config_rwlock));
}
```

```
**********************************************************
   17754 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/dsl_userhold.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************
/*
 * 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.
 */

#include <sys/zfs_context.h>
#include <sys/dsl_userhold.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_destroy.h>
#include <sys/dsl_synctask.h>
#include <sys/dmu_tx.h>
#include <sys/zfs_onexit.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dir.h>
#include <sys/zfs_ioctl.h>
#include <sys/zap.h>

typedef struct dsl_dataset_user_hold_arg {
        spa_t *dduha_spa;
#endif /* ! codereview */
        nvlist_t *dduha_holds;
        nvlist_t *dduha_chkholds;
        nvlist_t *dduha_tmpholds;
#endif /* ! codereview */
        nvlist_t *dduha_errlist;
        minor_t dduha_minor;
} dsl_dataset_user_hold_arg_t;

/*
 * If you add new checks here, you may need to add additional checks to the
 * "temporary" case in snapshot_check() in dmu_objset.c.
 */
int
dsl_dataset_user_hold_check_one(dsl_dataset_t *ds, const char *htag,
    boolean_t temphold, dmu_tx_t *tx)
{
        dsl_pool_t *dp = dmu_tx_pool(tx);
        objset_t *mos = dp->dp_meta_objset;
        int error = 0;
```

```
        ASSERT(RRW_READ_HELD(&dp->dp_config_rwlock));

#endif /* ! codereview */
        if (strlen(htag) > MAXNAMELEN)
                return (E2BIG);
        /* Tempholds have a more restricted length */
        if (temphold && strlen(htag) + MAX_TAG_PREFIX_LEN >= MAXNAMELEN)
                return (E2BIG);

        /* tags must be unique (if ds already exists) */
        if (ds != NULL && ds->ds_phys->ds_userrefs_obj != 0) {
        if (ds != NULL) {
                mutex_enter(&ds->ds_lock);
                if (ds->ds_phys->ds_userrefs_obj != 0) {
                uint64_t value;

#endif /* ! codereview */
                error = zap_lookup(mos, ds->ds_phys->ds_userrefs_obj,
                    htag, 8, 1, &value);
                if (error == 0)
                        error = SET_ERROR(EEXIST);
                else if (error == ENOENT)
                        error = 0;
        }
                mutex_exit(&ds->ds_lock);
        }

        return (error);
}

static int
dsl_dataset_user_hold_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_hold_arg_t *dduha = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;
        int rv = 0;

        if (spa_version(dp->dp_spa) < SPA_VERSION_USERREFS)
                return (SET_ERROR(ENOTSUP));

        if (!dmu_tx_is_syncing(tx))
                return (0);

#endif /* ! codereview */
        for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
                dsl_dataset_t *ds;
#endif /* ! codereview */
                int error = 0;
                char *htag, *name;
                dsl_dataset_t *ds;
                char *htag;

                /* must be a snapshot */
                name = nvpair_name(pair);
                if (strchr(name, '@') == NULL)
                if (strchr(nvpair_name(pair), '@') == NULL)
                        error = SET_ERROR(EINVAL);

                if (error == 0)
                        error = nvpair_value_string(pair, &htag);

                if (error == 0)
                        error = dsl_dataset_hold(dp, name, FTAG, &ds);
                if (error == 0) {
                        error = dsl_dataset_hold(dp,
                            nvpair_name(pair), FTAG, &ds);
                }
                if (error == 0) {
                        error = dsl_dataset_user_hold_check_one(ds, htag,
                            dduha->dduha_minor != 0, tx);
                        dsl_dataset_rele(ds, FTAG);
                }

                if (error == 0) {
                        fnvlist_add_string(dduha->dduha_chkholds, name, htag);
                } else {
                        /*
                         * We register ENOENT errors so they can be correctly
                         * reported if needed, such as when all holds fail.
                         */
                        fnvlist_add_int32(dduha->dduha_errlist, name, error);
                        if (error != ENOENT)
                                return (error);
                if (error != 0) {
                        rv = error;
                        fnvlist_add_int32(dduha->dduha_errlist,
                            nvpair_name(pair), error);
                }
        }

        /* Return ENOENT if no holds would be created. */
        if (nvlist_next_nvpair(dduha->dduha_chkholds, NULL) == NULL)
                return (ENOENT);

        return (0);
        return (rv);
}


static void
dsl_dataset_user_hold_sync_one_impl(nvlist_t *tmpholds, dsl_dataset_t *ds,
    const char *htag, minor_t minor, uint64_t now, dmu_tx_t *tx)
void
dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
    minor_t minor, uint64_t now, dmu_tx_t *tx)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj;

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

        mutex_enter(&ds->ds_lock);
        if (ds->ds_phys->ds_userrefs_obj == 0) {
                /*
                 * This is the first user hold for this dataset.  Create
                 * the userrefs zap object.
                 */
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                zapobj = ds->ds_phys->ds_userrefs_obj =
                    zap_create(mos, DMU_OT_USERREFS, DMU_OT_NONE, 0, tx);
        } else {
                zapobj = ds->ds_phys->ds_userrefs_obj;
        }
        ds->ds_userrefs++;
        mutex_exit(&ds->ds_lock);

        VERIFY0(zap_add(mos, zapobj, htag, 8, 1, &now, tx));
```

```
        if (minor != 0) {
                char name[MAXNAMELEN];
                nvlist_t *tags;

#endif /* ! codereview */
                VERIFY0(dsl_pool_user_hold(dp, ds->ds_object,
                    htag, now, tx));
                (void) snprintf(name, sizeof(name), "%llx",
                    (u_longlong_t)ds->ds_object);

                if (nvlist_lookup_nvlist(tmpholds, name, &tags) != 0) {
                        tags = fnvlist_alloc();
                        fnvlist_add_boolean(tags, htag);
                        fnvlist_add_nvlist(tmpholds, name, tags);
                        fnvlist_free(tags);
                } else {
                        fnvlist_add_boolean(tags, htag);
                }
                dsl_register_onexit_hold_cleanup(ds, htag, minor);
        }

        spa_history_log_internal_ds(ds, "hold", tx,
            "tag=%s temp=%d refs=%llu",
            htag, minor != 0, ds->ds_userrefs);
}

typedef struct zfs_hold_cleanup_arg {
        char zhca_spaname[MAXNAMELEN];
        uint64_t zhca_spa_load_guid;
        nvlist_t *zhca_holds;
} zfs_hold_cleanup_arg_t;

static void
dsl_dataset_user_release_onexit(void *arg)
{
        zfs_hold_cleanup_arg_t *ca = (zfs_hold_cleanup_arg_t *)arg;
        spa_t *spa;
        int error;

        error = spa_open(ca->zhca_spaname, &spa, FTAG);
        if (error != 0) {
                zfs_dbgmsg("couldn't release holds on pool=%s "
                    "because pool is no longer loaded",
                    ca->zhca_spaname);
                return;
        }
        if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
                zfs_dbgmsg("couldn't release holds on pool=%s "
                    "because pool is no longer loaded (guid doesn't match)",
                    ca->zhca_spaname);
                spa_close(spa, FTAG);
                return;
        }

        (void) dsl_dataset_user_release_tmp(spa_get_dsl(spa), ca->zhca_holds);
        fnvlist_free(ca->zhca_holds);
        kmem_free(ca, sizeof(zfs_hold_cleanup_arg_t));
        spa_close(spa, FTAG);
}

static void
dsl_register_onexit_hold_cleanup(spa_t *spa, nvlist_t *holds, minor_t minor)
{
        zfs_hold_cleanup_arg_t *ca;

        if (minor == 0 || nvlist_next_nvpair(holds, NULL) == NULL) {
                fnvlist_free(holds);
                return;
        }

        ASSERT(spa != NULL);
        ca = kmem_alloc(sizeof (*ca), KM_SLEEP);

        (void) strlcpy(ca->zhca_spaname, spa_name(spa),
            sizeof (ca->zhca_spaname));
        ca->zhca_spa_load_guid = spa_load_guid(spa);
        ca->zhca_holds = holds;
        VERIFY0(zfs_onexit_add_cb(minor,
            dsl_dataset_user_release_onexit, ca, NULL));
}

void
dsl_dataset_user_hold_sync_one(dsl_dataset_t *ds, const char *htag,
    minor_t minor, uint64_t now, dmu_tx_t *tx)
{
        nvlist_t *tmpholds;

        tmpholds = fnvlist_alloc();

        dsl_dataset_user_hold_sync_one_impl(tmpholds, ds, htag, minor, now, tx);
        dsl_register_onexit_hold_cleanup(dsl_dataset_get_spa(ds), tmpholds,
            minor);
}

#endif /* ! codereview */
static void
dsl_dataset_user_hold_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_hold_arg_t *dduha = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;
        uint64_t now = gethrestime_sec();

        for (pair = nvlist_next_nvpair(dduha->dduha_chkholds, NULL);
            pair != NULL;
            pair = nvlist_next_nvpair(dduha->dduha_chkholds, pair)) {
        for (pair = nvlist_next_nvpair(dduha->dduha_holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(dduha->dduha_holds, pair)) {
                dsl_dataset_t *ds;

#endif /* ! codereview */
                VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
                dsl_dataset_user_hold_sync_one_impl(dduha->dduha_tmpholds, ds,
                    fnvpair_value_string(pair), dduha->dduha_minor, now, tx);
                dsl_dataset_user_hold_sync_one(ds, fnvpair_value_string(pair),
                    dduha->dduha_minor, now, tx);
                dsl_dataset_rele(ds, FTAG);
        }
        dduha->dduha_spa = dp->dp_spa;
#endif /* ! codereview */
}

/*
 * The full semantics of this function are described in the comment above
 * lzc_hold().
 *
 * To summarize:
#endif /* ! codereview */
 * holds is nvl of snapname -> holdname
 * errlist will be filled in with snapname -> error
 * if cleanup_minor is not 0, the holds will be temporary, cleaned up
 * when the process exits.
```

```
 *
 * The snaphosts must all be in the same pool.
 *
 * Holds for snapshots that don't exist will be skipped.
 *
 * If none of the snapshots for requested holds exist then ENOENT will be
 * returned.
 *
 * If cleanup_minor is not 0, the holds will be temporary, which will be cleaned
 * up when the process exits.
 *
 * On success all the holds, for snapshots that existed, will be created and 0
 * will be returned.
 *
 * On failure no holds will be created, the errlist will be filled in,
 * and an errno will returned.
 *
 * In all cases the errlist will contain entries for holds where the snapshot
 * didn't exist.
 * if any fails, all will fail.
 */
int
dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor, nvlist_t *errlist)
{
        dsl_dataset_user_hold_arg_t dduha;
        nvpair_t *pair;
        int ret;
#endif /* ! codereview */

        pair = nvlist_next_nvpair(holds, NULL);
        if (pair == NULL)
                return (0);

        dduha.dduha_spa = NULL;
#endif /* ! codereview */
        dduha.dduha_holds = holds;
        dduha.dduha_chkholds = fnvlist_alloc();
        dduha.dduha_tmpholds = fnvlist_alloc();
#endif /* ! codereview */
        dduha.dduha_errlist = errlist;
        dduha.dduha_minor = cleanup_minor;

        ret = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
            dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds));

        /* dsl_register_onexit_hold_cleanup() always frees the passed holds. */
        dsl_register_onexit_hold_cleanup(dduha.dduha_spa, dduha.dduha_tmpholds,
            cleanup_minor);
        fnvlist_free(dduha.dduha_chkholds);

        return (ret);
        return (dsl_sync_task(nvpair_name(pair), dsl_dataset_user_hold_check,
            dsl_dataset_user_hold_sync, &dduha, fnvlist_num_pairs(holds)));
}

typedef int (dsl_holdfunc_t)(dsl_pool_t *dp, const char *name, void *tag,
    dsl_dataset_t **dsp);

#endif /* ! codereview */
typedef struct dsl_dataset_user_release_arg {
        dsl_holdfunc_t *ddura_holdfunc;
#endif /* ! codereview */
        nvlist_t *ddura_holds;
        nvlist_t *ddura_todelete;
        nvlist_t *ddura_errlist;
        nvlist_t *ddura_chkholds;
```

```
#endif /* ! codereview */
} dsl_dataset_user_release_arg_t;

/* Place a dataset hold on the snapshot identified by passed dsobj string */
static int
dsl_dataset_hold_obj_string(dsl_pool_t *dp, const char *dsobj, void *tag,
    dsl_dataset_t **dsp)
{
        return dsl_dataset_hold_obj(dp, strtonum(dsobj, NULL), tag, dsp);
}

#endif /* ! codereview */
static int
dsl_dataset_user_release_check_one(dsl_dataset_user_release_arg_t *ddura,
    dsl_dataset_t *ds, nvlist_t *holds, const char *name)
dsl_dataset_user_release_check_one(dsl_dataset_t *ds,
    nvlist_t *holds, boolean_t *todelete)
{
        uint64_t zapobj;
        nvpair_t *pair;
        nvlist_t *holds_found;
#endif /* ! codereview */
        objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
        int ret, numholds;
        int error;
        int numholds = 0;

        *todelete = B_FALSE;

        if (!dsl_dataset_is_snapshot(ds))
                return (SET_ERROR(EINVAL));

        zapobj = ds->ds_phys->ds_userrefs_obj;
        if (zapobj == 0)
                return (SET_ERROR(ESRCH));

        ret = 0;
        numholds = 0;
        holds_found = fnvlist_alloc();

#endif /* ! codereview */
        for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(holds, pair)) {
                /* Make sure the hold exists */
                uint64_t tmp;
                int error;
                const char *name;

                name = nvpair_name(pair);
                error = zap_lookup(mos, zapobj, name, 8, 1, &tmp);

                /* Non-existent holds aren’t always an error. */
                error = zap_lookup(mos, zapobj, nvpair_name(pair), 8, 1, &tmp);
                if (error == ENOENT)
                        continue;

                if (error != 0) {
                        fnvlist_free(holds_found);
                        error = SET_ERROR(ESRCH);
                if (error != 0)
                        return (error);
                }

                fnvlist_add_boolean(holds_found, name);
#endif /* ! codereview */
                numholds++;
```

```
        }

        if (DS_IS_DEFER_DESTROY(ds) && ds->ds_phys->ds_num_children == 1 &&
            ds->ds_userrefs == numholds) {
                /* we need to destroy the snapshot as well */
                if (dsl_dataset_long_held(ds)) {
                        fnvlist_free(holds_found);

                if (dsl_dataset_long_held(ds))
                        return (SET_ERROR(EBUSY));
                *todelete = B_TRUE;
                }
                fnvlist_add_boolean(ddura->ddura_todelete, name);
        }

        if (numholds == 0)
                ret = ENOENT;
        else
                fnvlist_add_nvlist(ddura->ddura_chkholds, name, holds_found);
        fnvlist_free(holds_found);

        return (ret);
        return (0);
}

static int
dsl_dataset_user_release_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_release_arg_t *ddura;
        dsl_holdfunc_t *holdfunc;
        dsl_pool_t *dp;
        dsl_dataset_user_release_arg_t *ddura = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;
        int rv = 0;

        if (!dmu_tx_is_syncing(tx))
                return (0);

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

        dp = dmu_tx_pool(tx);
        ddura = (dsl_dataset_user_release_arg_t *)arg;
        holdfunc = ddura->ddura_holdfunc;

#endif /* ! codereview */
        for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
                const char *name;
                const char *name = nvpair_name(pair);
                int error;
                dsl_dataset_t *ds;
                nvlist_t *holds;

                name = nvpair_name(pair);
#endif /* ! codereview */
                error = nvpair_value_nvlist(pair, &holds);
                if (error != 0)
                        error = (SET_ERROR(EINVAL));
                if (error == 0)
                        error = holdfunc(dp, name, FTAG, &ds);
                        return (SET_ERROR(EINVAL));

                error = dsl_dataset_hold(dp, name, FTAG, &ds);
                if (error == 0) {
                        error = dsl_dataset_user_release_check_one(ddura, ds,
                            holds, name);
                        boolean_t deleteme;
                        error = dsl_dataset_user_release_check_one(ds,
                            holds, &deleteme);
                        if (error == 0 && deleteme) {
                                fnvlist_add_boolean(ddura->ddura_todelete,
                                    name);
                        }
                        dsl_dataset_rele(ds, FTAG);
                }
                if (error != 0) {
                        if (ddura->ddura_errlist != NULL) {
                                fnvlist_add_int32(ddura->ddura_errlist, name,
                                    error);
                                fnvlist_add_int32(ddura->ddura_errlist,
                                    name, error);
                        }
                        /* Non-existent holds aren't always an error. */
                        if (error != ENOENT)
                                return (error);
                        rv = error;
                }
        }

        /*
         * Return ENOENT if none of the holds existed avoiding the overhead
         * of a sync.
         */
        if (nvlist_next_nvpair(ddura->ddura_chkholds, NULL) == NULL)
                return (ENOENT);

        return (0);
        return (rv);
}

static void
dsl_dataset_user_release_sync_one(dsl_dataset_user_release_arg_t *ddura,
    dsl_dataset_t *ds, nvlist_t *holds, dmu_tx_t *tx)
dsl_dataset_user_release_sync_one(dsl_dataset_t *ds, nvlist_t *holds,
    dmu_tx_t *tx)
{
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        objset_t *mos = dp->dp_meta_objset;
        uint64_t zapobj;
        int error;
        nvpair_t *pair;

        for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(holds, pair)) {
                uint64_t zapobj;
                int error;
                const char *name;

                name = nvpair_name(pair);

                /* Remove temporary hold if one exists. */
                error = dsl_pool_user_release(dp, ds->ds_object, name, tx);
                ds->ds_userrefs--;
                error = dsl_pool_user_release(dp, ds->ds_object,
                    nvpair_name(pair), tx);
                VERIFY(error == 0 || error == ENOENT);

                /* Remove user hold if one exists. */
#endif /* ! codereview */
                zapobj = ds->ds_phys->ds_userrefs_obj;
                error = zap_remove(mos, zapobj, name, tx);
```

```
                if (error == ENOENT)
                        continue;
                VERIFY0(error);

                /* Only if we removed a hold do we decrement ds_userrefs. */
                ds->ds_userrefs--;
                VERIFY0(zap_remove(mos, zapobj, nvpair_name(pair), tx));

                spa_history_log_internal_ds(ds, "release", tx,
                    "tag=%s refs=%lld", nvpair_name(pair),
                    (longlong_t)ds->ds_userrefs);
        }
}

static void
dsl_dataset_user_release_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_release_arg_t *ddura = arg;
        dsl_holdfunc_t *holdfunc = ddura->ddura_holdfunc;
#endif /* ! codereview */
        dsl_pool_t *dp = dmu_tx_pool(tx);
        nvpair_t *pair;

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

        for (pair = nvlist_next_nvpair(ddura->ddura_chkholds, NULL);
            pair != NULL; pair = nvlist_next_nvpair(ddura->ddura_chkholds,
            pair)) {
        for (pair = nvlist_next_nvpair(ddura->ddura_holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(ddura->ddura_holds, pair)) {
                dsl_dataset_t *ds;
                const char *name;
#endif /* ! codereview */

                name = nvpair_name(pair);
                VERIFY0(holdfunc(dp, name, FTAG, &ds));

                dsl_dataset_user_release_sync_one(ddura, ds,
                VERIFY0(dsl_dataset_hold(dp, nvpair_name(pair), FTAG, &ds));
                dsl_dataset_user_release_sync_one(ds,
                    fnvpair_value_nvlist(pair), tx);
                if (nvlist_exists(ddura->ddura_todelete, name)) {
                if (nvlist_exists(ddura->ddura_todelete,
                    nvpair_name(pair))) {
                        ASSERT(ds->ds_userrefs == 0 &&
                            ds->ds_phys->ds_num_children == 1 &&
                            DS_IS_DEFER_DESTROY(ds));
                        dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
                }
                dsl_dataset_rele(ds, FTAG);
        }
}

/*
 * The full semantics of this function are described in the comment above
 * lzc_release().
 *
 * To summarize:
 * Releases holds specified in the nvl holds.
 *
#endif /* ! codereview */
 * holds is nvl of snapname -> { holdname, ... }
 * errlist will be filled in with snapname -> error
 *
 * If tmpdp is not NULL the names for holds should be the dsobj's of snapshots,
 * otherwise they should be the names of shapshots.
 *
 * As a release may cause snapshots to be destroyed this trys to ensure they
 * aren't mounted.
 *
 * The release of non-existent holds are skipped.
 *
 * At least one hold must have been released for the this function to succeed
 * and return 0.
 * if any fails, all will fail.
 */
static int
dsl_dataset_user_release_impl(nvlist_t *holds, nvlist_t *errlist,
    dsl_pool_t *tmpdp)
int
dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
{
        dsl_dataset_user_release_arg_t ddura;
        nvpair_t *pair;
        char *pool;
#endif /* ! codereview */
        int error;

        pair = nvlist_next_nvpair(holds, NULL);
        if (pair == NULL)
                return (0);

#ifdef _KERNEL
        /*
         * The release may cause snapshots to be destroyed; make sure they
         * are not mounted.
         */
        if (tmpdp != NULL) {
                /* Temporary holds are specified by dsobj string. */
                ddura.ddura_holdfunc = dsl_dataset_hold_obj_string;
                pool = spa_name(tmpdp->dp_spa);
        ddura.ddura_holds = holds;
        ddura.ddura_errlist = errlist;
        ddura.ddura_todelete = fnvlist_alloc();

                dsl_pool_config_enter(tmpdp, FTAG);
                for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
                    pair = nvlist_next_nvpair(holds, pair)) {
        error = dsl_sync_task(nvpair_name(pair), dsl_dataset_user_release_check,
            dsl_dataset_user_release_sync, &ddura, fnvlist_num_pairs(holds));
        fnvlist_free(ddura.ddura_todelete);
        return (error);
}

typedef struct dsl_dataset_user_release_tmp_arg {
        uint64_t ddurta_dsobj;
        nvlist_t *ddurta_holds;
        boolean_t ddurta_deleteme;
} dsl_dataset_user_release_tmp_arg_t;

static int
dsl_dataset_user_release_tmp_check(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
                        dsl_dataset_t *ds;
        int error;

                        error = dsl_dataset_hold_obj_string(tmpdp,
                            nvpair_name(pair), FTAG, &ds);
        if (!dmu_tx_is_syncing(tx))
                return (0);
```

```
        error = dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds);
        if (error)
                return (error);

        error = dsl_dataset_user_release_check_one(ds,
            ddurta->ddurta_holds, &ddurta->ddurta_deleteme);
        dsl_dataset_rele(ds, FTAG);
        return (error);
}

static void
dsl_dataset_user_release_tmp_sync(void *arg, dmu_tx_t *tx)
{
        dsl_dataset_user_release_tmp_arg_t *ddurta = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dataset_t *ds;

        VERIFY0(dsl_dataset_hold_obj(dp, ddurta->ddurta_dsobj, FTAG, &ds));
        dsl_dataset_user_release_sync_one(ds, ddurta->ddurta_holds, tx);
        if (ddurta->ddurta_deleteme) {
                ASSERT(ds->ds_userrefs == 0 &&
                    ds->ds_phys->ds_num_children == 1 &&
                    DS_IS_DEFER_DESTROY(ds));
                dsl_destroy_snapshot_sync_impl(ds, B_FALSE, tx);
        }
        dsl_dataset_rele(ds, FTAG);
}

/*
 * Called at spa_load time to release a stale temporary user hold.
 * Also called by the onexit code.
 */
void
dsl_dataset_user_release_tmp(dsl_pool_t *dp, uint64_t dsobj, const char *htag)
{
        dsl_dataset_user_release_tmp_arg_t ddurta;
        dsl_dataset_t *ds;
        int error;

#ifdef _KERNEL
        /* Make sure it is not mounted. */
        dsl_pool_config_enter(dp, FTAG);
        error = dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds);
                        if (error == 0) {
                                char name[MAXNAMELEN];
                                dsl_dataset_name(ds, name);
                                dsl_dataset_rele(ds, FTAG);
                dsl_pool_config_exit(dp, FTAG);
                                zfs_unmount_snap(name);
                        }
                }
                dsl_pool_config_exit(tmpdp, FTAG);
#endif /* ! codereview */
        } else {
                /* Non-temporary holds are specified by name. */
                ddura.ddura_holdfunc = dsl_dataset_hold;
                pool = nvpair_name(pair);

                for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
                    pair = nvlist_next_nvpair(holds, pair)) {
                        zfs_unmount_snap(nvpair_name(pair));
                }
                dsl_pool_config_exit(dp, FTAG);
        }
#endif
```

```
        ddura.ddura_holds = holds;
        ddura.ddura_errlist = errlist;
        ddura.ddura_todelete = fnvlist_alloc();
        ddura.ddura_chkholds = fnvlist_alloc();

        error = dsl_sync_task(pool, dsl_dataset_user_release_check,
            dsl_dataset_user_release_sync, &ddura,
            fnvlist_num_pairs(holds));
        fnvlist_free(ddura.ddura_todelete);
        fnvlist_free(ddura.ddura_chkholds);

        return (error);
        ddurta.ddurta_dsobj = dsobj;
        ddurta.ddurta_holds = fnvlist_alloc();
        fnvlist_add_boolean(ddurta.ddurta_holds, htag);

        (void) dsl_sync_task(spa_name(dp->dp_spa),
            dsl_dataset_user_release_tmp_check,
            dsl_dataset_user_release_tmp_sync, &ddurta, 1);
        fnvlist_free(ddurta.ddurta_holds);
}

/*
 * holds is nvl of snapname -> { holdname, ... }
 * errlist will be filled in with snapname -> error
 */
int
dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist)
typedef struct zfs_hold_cleanup_arg {
        char zhca_spaname[MAXNAMELEN];
        uint64_t zhca_spa_load_guid;
        uint64_t zhca_dsobj;
        char zhca_htag[MAXNAMELEN];
} zfs_hold_cleanup_arg_t;

static void
dsl_dataset_user_release_onexit(void *arg)
{
        return dsl_dataset_user_release_impl(holds, errlist, NULL);
        zfs_hold_cleanup_arg_t *ca = arg;
        spa_t *spa;
        int error;

        error = spa_open(ca->zhca_spaname, &spa, FTAG);
        if (error != 0) {
                zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
                    "because pool is no longer loaded",
                    ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
                return;
        }
        if (spa_load_guid(spa) != ca->zhca_spa_load_guid) {
                zfs_dbgmsg("couldn't release hold on pool=%s ds=%llu tag=%s "
                    "because pool is no longer loaded (guid doesn't match)",
                    ca->zhca_spaname, ca->zhca_dsobj, ca->zhca_htag);
                spa_close(spa, FTAG);
                return;
        }

        dsl_dataset_user_release_tmp(spa_get_dsl(spa),
            ca->zhca_dsobj, ca->zhca_htag);
        kmem_free(ca, sizeof (zfs_hold_cleanup_arg_t));
        spa_close(spa, FTAG);
}

/*
```

```
 * holds is nvl of snapdsobj -> { holdname, ... }
 */
#endif /* ! codereview */
void
dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds)
dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
    minor_t minor)
{
        ASSERT(dp != NULL);
        (void) dsl_dataset_user_release_impl(holds, NULL, dp);
        zfs_hold_cleanup_arg_t *ca = kmem_alloc(sizeof (*ca), KM_SLEEP);
        spa_t *spa = dsl_dataset_get_spa(ds);
        (void) strlcpy(ca->zhca_spaname, spa_name(spa),
            sizeof (ca->zhca_spaname));
        ca->zhca_spa_load_guid = spa_load_guid(spa);
        ca->zhca_dsobj = ds->ds_object;
        (void) strlcpy(ca->zhca_htag, htag, sizeof (ca->zhca_htag));
        VERIFY0(zfs_onexit_add_cb(minor,
            dsl_dataset_user_release_onexit, ca, NULL));
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   10147 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************
______unchanged_portion_omitted_

/*
 * The max length of a temporary tag prefix is the number of hex digits
 * required to express UINT64_MAX plus one for the hyphen.
 */
#define MAX_TAG_PREFIX_LEN      17

#define dsl_dataset_is_snapshot(ds) \
        ((ds)->ds_phys->ds_num_children != 0)

#define DS_UNIQUE_IS_ACCURATE(ds)       \
        (((ds)->ds_phys->ds_flags & DS_FLAG_UNIQUE_ACCURATE) != 0)

int dsl_dataset_hold(struct dsl_pool *dp, const char *name, void *tag,
    dsl_dataset_t **dsp);
int dsl_dataset_hold_obj(struct dsl_pool *dp, uint64_t dsobj, void *tag,
    dsl_dataset_t **);
void dsl_dataset_rele(dsl_dataset_t *ds, void *tag);
int dsl_dataset_own(struct dsl_pool *dp, const char *name,
    void *tag, dsl_dataset_t **dsp);
int dsl_dataset_own_obj(struct dsl_pool *dp, uint64_t dsobj,
    void *tag, dsl_dataset_t **dsp);
void dsl_dataset_disown(dsl_dataset_t *ds, void *tag);
void dsl_dataset_name(dsl_dataset_t *ds, char *name);
boolean_t dsl_dataset_tryown(dsl_dataset_t *ds, void *tag);
void dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
    minor_t minor);
uint64_t dsl_dataset_create_sync(dsl_dir_t *pds, const char *lastname,
    dsl_dataset_t *origin, uint64_t flags, cred_t *, dmu_tx_t *);
uint64_t dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
    uint64_t flags, dmu_tx_t *tx);
int dsl_dataset_snapshot(nvlist_t *snaps, nvlist_t *props, nvlist_t *errors);
int dsl_dataset_promote(const char *name, char *conflsnap);
int dsl_dataset_clone_swap(dsl_dataset_t *clone, dsl_dataset_t *origin_head,
    boolean_t force);
int dsl_dataset_rename_snapshot(const char *fsname,
    const char *oldsnapname, const char *newsnapname, boolean_t recursive);
int dsl_dataset_snapshot_tmp(const char *fsname, const char *snapname,
    minor_t cleanup_minor, const char *htag);

blkptr_t *dsl_dataset_get_blkptr(dsl_dataset_t *ds);
void dsl_dataset_set_blkptr(dsl_dataset_t *ds, blkptr_t *bp, dmu_tx_t *tx);

spa_t *dsl_dataset_get_spa(dsl_dataset_t *ds);

boolean_t dsl_dataset_modified_since_lastsnap(dsl_dataset_t *ds);

void dsl_dataset_sync(dsl_dataset_t *os, zio_t *zio, dmu_tx_t *tx);

void dsl_dataset_block_born(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx);
int dsl_dataset_block_kill(dsl_dataset_t *ds, const blkptr_t *bp,
    dmu_tx_t *tx, boolean_t async);
boolean_t dsl_dataset_block_freeable(dsl_dataset_t *ds, const blkptr_t *bp,
    uint64_t blk_birth);
uint64_t dsl_dataset_prev_snap_txg(dsl_dataset_t *ds);

void dsl_dataset_dirty(dsl_dataset_t *ds, dmu_tx_t *tx);
void dsl_dataset_stats(dsl_dataset_t *os, nvlist_t *nv);
```

```
void dsl_dataset_fast_stat(dsl_dataset_t *ds, dmu_objset_stats_t *stat);
void dsl_dataset_space(dsl_dataset_t *ds,
    uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);
uint64_t dsl_dataset_fsid_guid(dsl_dataset_t *ds);
int dsl_dataset_space_written(dsl_dataset_t *oldsnap, dsl_dataset_t *new,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
int dsl_dataset_space_wouldfree(dsl_dataset_t *firstsnap, dsl_dataset_t *last,
    uint64_t *usedp, uint64_t *compp, uint64_t *uncompp);
boolean_t dsl_dataset_is_dirty(dsl_dataset_t *ds);

int dsl_dsobj_to_dsname(char *pname, uint64_t obj, char *buf);

int dsl_dataset_check_quota(dsl_dataset_t *ds, boolean_t check_quota,
    uint64_t asize, uint64_t inflight, uint64_t *used,
    uint64_t *ref_rsrv);
int dsl_dataset_set_refquota(const char *dsname, zprop_source_t source,
    uint64_t quota);
int dsl_dataset_set_refreservation(const char *dsname, zprop_source_t source,
    uint64_t reservation);

boolean_t dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier);
void dsl_dataset_long_hold(dsl_dataset_t *ds, void *tag);
void dsl_dataset_long_rele(dsl_dataset_t *ds, void *tag);
boolean_t dsl_dataset_long_held(dsl_dataset_t *ds);

int dsl_dataset_clone_swap_check_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, boolean_t force);
void dsl_dataset_clone_swap_sync_impl(dsl_dataset_t *clone,
    dsl_dataset_t *origin_head, dmu_tx_t *tx);
int dsl_dataset_snapshot_check_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx);
void dsl_dataset_snapshot_sync_impl(dsl_dataset_t *ds, const char *snapname,
    dmu_tx_t *tx);

void dsl_dataset_remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj,
    dmu_tx_t *tx);
void dsl_dataset_recalc_head_uniq(dsl_dataset_t *ds);
int dsl_dataset_get_snapname(dsl_dataset_t *ds);
int dsl_dataset_snap_lookup(dsl_dataset_t *ds, const char *name,
    uint64_t *value);
int dsl_dataset_snap_remove(dsl_dataset_t *ds, const char *name, dmu_tx_t *tx);
void dsl_dataset_set_refreservation_sync_impl(dsl_dataset_t *ds,
    zprop_source_t source, uint64_t value, dmu_tx_t *tx);
int dsl_dataset_rollback(const char *fsname);

#ifdef ZFS_DEBUG
#define dprintf_ds(ds, fmt, ...) do { \
        if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
        char *__ds_name = kmem_alloc(MAXNAMELEN, KM_SLEEP); \
        dsl_dataset_name(ds, __ds_name); \
        dprintf("ds=%s " fmt, __ds_name, __VA_ARGS__); \
        kmem_free(__ds_name, MAXNAMELEN); \
        } \
_NOTE(CONSTCOND) } while (0)
#else
#define dprintf_ds(dd, fmt, ...)
#endif

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
    1826 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_userhold.h
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
**********************************************************

/*
 * 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) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_DSL_USERHOLD_H
#define _SYS_DSL_USERHOLD_H

#include <sys/nvpair.h>
#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_pool;
struct dsl_dataset;
struct dmu_tx;

int dsl_dataset_user_hold(nvlist_t *holds, minor_t cleanup_minor,
    nvlist_t *errlist);
int dsl_dataset_user_release(nvlist_t *holds, nvlist_t *errlist);
int dsl_dataset_get_holds(const char *dsname, nvlist_t *nvl);
void dsl_dataset_user_release_tmp(struct dsl_pool *dp, nvlist_t *holds);
void dsl_dataset_user_release_tmp(struct dsl_pool *dp, uint64_t dsobj,
    const char *htag);
int dsl_dataset_user_hold_check_one(struct dsl_dataset *ds, const char *htag,
    boolean_t temphold, struct dmu_tx *tx);
void dsl_dataset_user_hold_sync_one(struct dsl_dataset *ds, const char *htag,
    minor_t minor, uint64_t now, struct dmu_tx *tx);

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

****
143884 Fri May 24 00:51:03 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3740 Poor ZFS send / receive performance due to snapshot hold / release processi
Submitted by: Steven Hartland <steven.hartland@multiplay.co.uk>
****
______***unchanged_portion_omitted_***

```
/*
 * innvl: {
 *     snapname -> { holdname, ... }
 *     ...
 * }
 *
 * outnvl: {
 *     snapname -> error value (int32)
 *     ...
 * }
 */
/* ARGSUSED */
static int
zfs_ioc_release(const char *pool, nvlist_t *holds, nvlist_t *errlist)
{
        nvpair_t *pair;

        /*
         * The release may cause the snapshot to be destroyed; make sure it
         * is not mounted.
         */
        for (pair = nvlist_next_nvpair(holds, NULL); pair != NULL;
            pair = nvlist_next_nvpair(holds, pair))
                zfs_unmount_snap(nvpair_name(pair));

        return (dsl_dataset_user_release(holds, errlist));
}
```
______***unchanged_portion_omitted_***