ZFS On-Disk Data Walk (Or: Where's My Data)

OpenSolaris Developer Conference,
June 25-27, 2008 Prague

Max Bruning
Bruning Systems, LLC

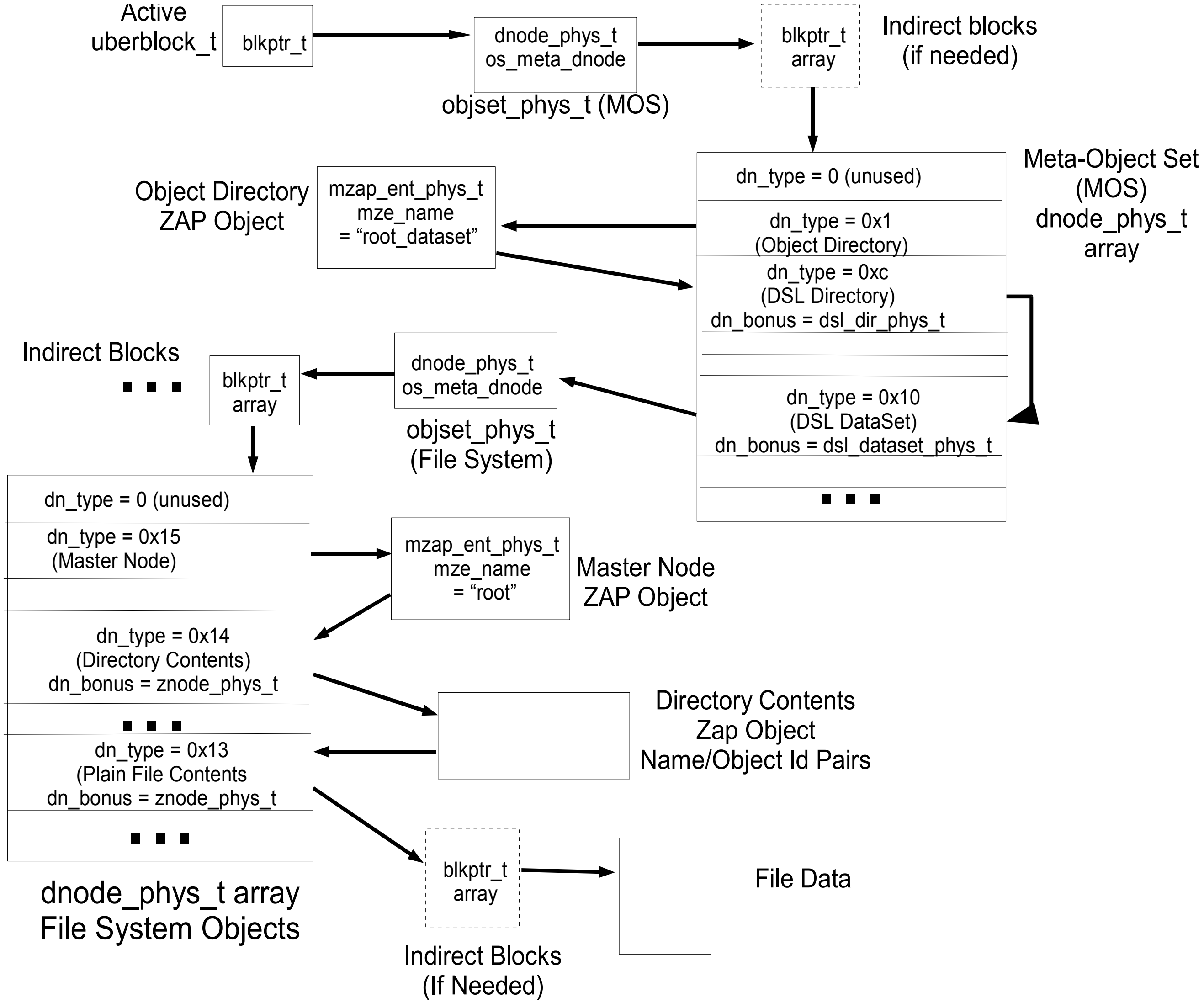
Topic Outline

- Overview of On-Disk Data Structures
- Using zdb(1M) and mdb(1) to examine data on a zfs file system
- Mirrors, Child DataSets (Multiple File Systems in a Pool), Snapshots, etc.
- Conclusions and Future Work
- References

Overview of On-Disk Data Structures

- uberblock_t – Starting point for all file systems in a pool
- blkptr_t – Contains description, location, and size of blocks in a zfs file system
- dnode_phys_t – Meta data used to describe all objects in the file system
- ZAP Objects – Block(s) containing name/value pairs
- Bonus Buffer – Field in dnode_phys_t used to contain additional information about meta data

ZFS On-Disk Layout – The Big Picture



Active
uberblock_t

blkptr_t

dnode_phys_t
os_meta_dnode

objset_phys_t (MOS)

blkptr_t
array

Indirect blocks
(if needed)

Object Directory
ZAP Object

mzap_ent_phys_t
mze_name
= “root_dataset”

dn_type = 0 (unused)

dn_type = 0x1
(Object Directory)

dn_type = 0xc
(DSL Directory)
dn_bonus = dsl_dir_phys_t

dn_type = 0x10
(DSL DataSet)
dn_bonus = dsl_dataset_phys_t

■ ■ ■

Meta-Object Set
(MOS)
dnode_phys_t
array

Indirect Blocks
■ ■ ■

blkptr_t
array

dnode_phys_t
os_meta_dnode

objset_phys_t
(File System)

dn_type = 0 (unused)

dn_type = 0x15
(Master Node)

dn_type = 0x14
(Directory Contents)
dn_bonus = znode_phys_t

■ ■ ■

dn_type = 0x13
(Plain File Contents
dn_bonus = znode_phys_t

■ ■ ■

dnode_phys_t array
File System Objects

mzap_ent_phys_t
mze_name
= “root”

Master Node
ZAP Object

Directory Contents
Zap Object
Name/Object Id Pairs

blkptr_t
array

Indirect Blocks
(If Needed)

File Data

Modifications for mdb and zdb

- Use modified mdb and zdb to examine the metadata
  - Mdb Modifications
    - Load kernel CTF so that “::print type” works with binary files
    - Add rawzfs.so dmod so “::blkptr” works with binary files
  - Zdb Modifications
    - Add decompression options for “zdb -R ...”
      - # zdb -R pool:vdev:location:psize:d,lzjb,lsize
      - “d,lzjb,lsize” not currently supported
      - “lzjb” can be any supported compression mechanism
      - Lsize is size after decompression, psize is physical size on disk
- Modifications can be obtained here ???

Data Walk Overview

- Start with uberblock_t and find data for a given file on disk
- Alternate between zdb(1M) and mdb(1) to retrieve the (decompressed) metadata and display it
- Refer to the “Big Picture” diagram on slide 4 so you don't get lost...

Step 1: Create a file with known data and display active uberblock_t

```
# cp /usr/dict/words /zfs_fs/words   <-- /zfs_fs is a zfs file system
#

# zdb -uuu zfs_fs  <-- Display active uberblock_t
Uberblock

        magic = 0000000000bab10c
        version = 10
        txg = 19148
        guid_sum = 17219723339164464949
        timestamp = 1203801884 UTC = Sat Feb 23 22:24:44 2008
        rootbp = [L0 DMU objset] 400L/200P DVA[0]=<0:4e00:200>
 DVA[1]=<0:1c0004e00:200> DVA[2]=<0:380001200:200>
 fletcher4 lzjb LE contiguous birth=19148 fill=18
 cksum=89aca5d29:38d271ef883:be5570b26779:1af282de579a51
#
```

Step 2: Decompress/display objset_phys_t for MetaObject Set (MOS)

```
# zdb -R zfs_fs:0:4e00:200:d,lzjb,400 2> /tmp/metadnode
Found vdev: /dev/dsk/c4t0d0p1
#
# mdb /tmp/metadnode
> 0::print -a -t zfs`objset_phys_t
{
    0 dnode_phys_t os_meta_dnode = {
        0 uint8_t dn_type = 0xa          <-- DMU_OT_DNODE
        1 uint8_t dn_indblkshift = 0xe
        2 uint8_t dn_nlevels = 0x1       <-- no indirect blocks
        3 uint8_t dn_nblkptr = 0x3       <-- 3 copies in the blkptr_t
 ... <--output omitted
        40 blkptr_t [1] dn_blkptr = [    <-- blkptr is at address 0x40 in t
                   <-- output omitted
```

Step 3: Display blkptr_t for MOS

```
> 40::blkptr  <-- 0x40 is location of blkptr_t in /tmp/metadnode
DVA[0]: vdev_id 0 / 5000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[0]: :0:5000:800:d
DVA[1]: vdev_id 0 / 1c0005000
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[1]: :0:1c0005000:800:d
DVA[2]: vdev_id 0 / 380003800
DVA[2]:       GANG: FALSE  GRID:  0000  ASIZE: 80000000000
DVA[2]: :0:380003800:800:d
LSIZE:  4000                              PSIZE: 800
ENDIAN: LITTLE                                      TYPE:  DMU dnode
BIRTH:  4acc              LEVEL: 0     FILL:  1100000000
CKFUNC: fletcher4                             COMP:  lzjb
CKSUM:  8348a7aa95:8a9a1c0eb664:5b0e32bd611ac7:
   2d691acc5e1f456f
> $q
```

Step 4: Retrieve MOS (dnode_phys_t Array)

```
# zdb -R zfs_fs:0:5000:800:d,lzjb,4000 2> /tmp/mos
Found vdev: /dev/dsk/c4t0d0p1
#
```

Note that there may be indirect blocks between the objset_phys_t and the MOS. Here, the Level field from the ::blkptr output is 0, so no indirect blocks.

Step 5: Display MOS dnode_phys_t Array

```
# mdb /tmp/mos
> ::sizeof zfs`dnode_phys_t   <-- how large is a dnode_phys_t?
sizeof (zfs`dnode_phys_t) = 0x200
> 4000%200=K        <-- how many dnode_phys_t are there in
the block?
                20

> 0,20::print -a -t zfs`dnode_phys_t <-- dump the 32
dnode_phys_t
{
    0 uint8_t dn_type = 0   <-- DMU_OT_NONE (not in use)
...  <-- output truncated
}
{
    200 uint8_t dn_type = 0x1  <--
DMU_OT_OBJECT_DIRECTORY
```

Step 6: Display the Object Directory blkptr_t

```
> 240::blkptr
DVA[0]: vdev_id 0 / 200
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:200:200:d
DVA[1]: vdev_id 0 / 1c0000200
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c0000200:200:d
DVA[2]: vdev_id 0 / 380000000
DVA[2]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[2]: :0:380000000:200:d
LSIZE:  200                              PSIZE: 200
ENDIAN: LITTLE                                      TYPE:  object directory
BIRTH:  4                   LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                            COMP:  uncompressed
CKSUM:  5a6f58679:1cf035fcb09:528ae171d3a8:
    aabf09f1373ae
```

Step 7: Display Object Directory

```
# zdb -R zfs_fs:0:200:200:r 2> /tmp/objdir
Found vdev: /dev/dsk/c4t0d0p1
# mdb /tmp/objdir
> 0/J
0:              8000000000000003 <-- this is a microzap

> 0::print -a -t zfs`mzap_phys_t  <-- print an mzap_phys_t
{
    0 uint64_t mz_block_type = 0x8000000000000003
...<-- output omitted
    40 mzap_ent_phys_t [1] mz_chunk = [
        {
            40 uint64_t mze_value = 0x2  <-- object id in MOS
            48 uint32_t mze_cd = 0
            4c uint16_t mze_pad = 0
            4e char [50] mze_name = [ "root_dataset" ]
        }
```

Step 8: Display root_dataset dnode_phys_t

```
# mdb /tmp/mos <-- from step 5
> 400::print zfs`dnode_phys_t  <-- object id 2 is 2*0x200 bytes arra
{
    400 uint8_t dn_type = 0xc   <-- DMU_OT_DSL_DIR
 ... <-- output omitted
    404 uint8_t dn_bonustype = 0xc
... <-- output omitted
    40a uint16_t dn_bonuslen = 0x100
... <-- output omitted
    440 blkptr_t [1] dn_blkptr = [
        {
                440 uint64_t [2] dva_word = [ 0, 0 ]  <-- blkptr not use
            }
... <-- output omitted
    ]
    4c0 uint8_t [320] dn_bonus = [ 0x34, 0x9c, 0xb9, 0x47, 0, 0, 0,
```

Step 9: Display root_dataset Bonus Buffer

```
> 4c0::print -a -t zfs`dsl_dir_phys_t
{
    4c0 uint64_t dd_creation_time = 0x47b99c34
    4c8 uint64_t dd_head_dataset_obj = 0x5
 ... <-- output omitted
}
>
```

Step 10: Display Dataset dnode_phys_t

```
> 5*200::print -t -a zfs`dnode_phys_t  <-- object id 5 in mos
{
    a00 uint8_t dn_type = 0x10  <-- DMU_OT_DSL_DATASET
... <-- output omitted
    a04 uint8_t dn_bonustype = 0x10  <-- DMU_OT_DSL_DATASET
... <-- output omitted
    a0a uint16_t dn_bonuslen = 0x140
... <-- output omitted
    a40 blkptr_t [1] dn_blkptr = [
        {
            a40 dva_t [3] blk_dva = [
                {
                    a40 uint64_t [2] dva_word = [ 0, 0 ]  <-- blkptr not us
                }
... <-- output omitted
    ]
    ac0 uint8_t [320] dn_bonus = [ 0x2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ]
```

Step 11: Display dsl_dataset_phys_t Bonus Buffer

```
> ac0::print -a -t zfs`dsl_dataset_phys_t
{
    ac0 uint64_t ds_dir_obj = 0x2
 ... <-- output omitted
    b40 blkptr_t ds_bp = {
        b40 dva_t [3] blk_dva = [
            {
                b40 uint64_t [2] dva_word = [ 0x1, 0x26 ]
            }
            {
                b50 uint64_t [2] dva_word = [ 0x1, 0xe00026 ]
            }
            {
                b60 uint64_t [2] dva_word = [ 0, 0 ]
            }
        ]
```

Step 12: Display root dataset blkptr_t

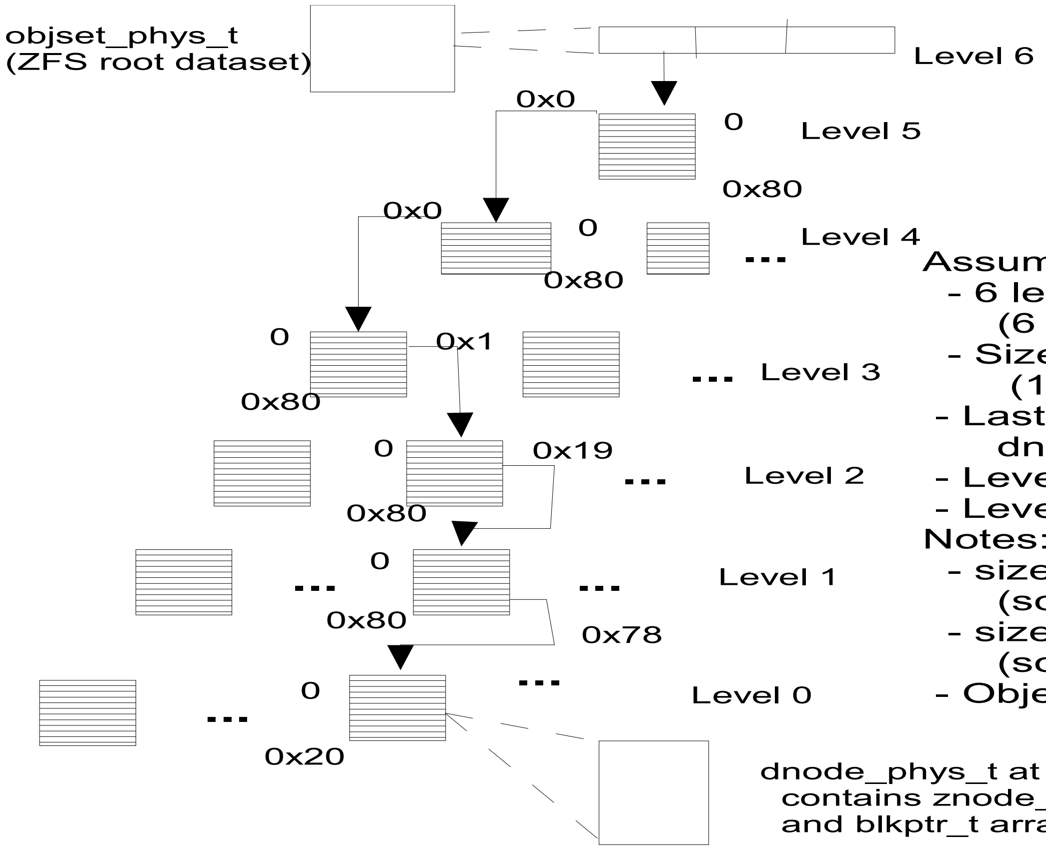
```
> b40::blkptr
DVA[0]: vdev_id 0 / 4c00
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[0]: :0:4c00:200:d
DVA[1]: vdev_id 0 / 1c0004c00
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
DVA[1]: :0:1c0004c00:200:d
LSIZE:  400                         PSIZE: 200
ENDIAN: LITTLE                               TYPE:  DMU objset
BIRTH:  4acc               LEVEL: 0     FILL:  1500000000
CKFUNC: fletcher4                        COMP:  lzjb
CKSUM:  a9a691571:477e2285748:f44e24c94c3d:23418ff35bb2
> $q
# zdb -R zfs_fs:0:4c00:200:d,lzjb,400 2> /tmp/root_dataset_metadnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

Step 13: Display root dataset objset_phys_t

```
# mdb /tmp/root_dataset_metadnode
> 0::print -a -t zfs`objset_phys_t
{
    0 dnode_phys_t os_meta_dnode = {
        0 uint8_t dn_type = 0xa  <-- DMU_OT_DNODE
        1 uint8_t dn_indblkshift = 0xe
        2 uint8_t dn_nlevels = 0x7  <-- levels of indirection
... <-- output omitted
        40 blkptr_t [1] dn_blkptr = [
            {
                40 dva_t [3] blk_dva = [
                    {
                        40 uint64_t [2] dva_word = [ 0x2, 0x24 ]
 ... <-- output omitted
```

Step 14: Walk Indirect Blocks to get File System dnode_phys_t Array

- The blkptr_t from the previous step shows 7 levels of indirection
- This step involves the following sequence to get to Level 0, a dnode_phys_t array
  - Zdb -R zfs_fs:0:offset:psize:d,lzjb,lsize 2> /tmp/file
  - Mdb /tmp/file
    - 0::blkptr
  - And repeat until Level 0 is shown
- For details, see the paper at:



dnode_phys_t at index 0x1d
contains znode_phys_t in bonus buffer
and blkptr_t array to locate data of file

Assumptions:
- 6 levels of Indirect blocks
  (6 is maximum)
- Sizes of all blocks = 16k
  (128K is maximum)
- Last level are blocks containing
  dnode_phys_t structs
- Level 6 through 1 are indirect blocks
- Level 0 are dnode_phys_t structs

Notes:
- sizeof (blkptr_t) = 0x80
  (so 0x80 blkptr_t in 1 block)
- sizeof (dnode_phys_t) = 0x200
  (so 0x20 dnode_phys_t in 1 block)
- Objects are not drawn to scale

Example – Find dnode_phys_t for object id = 0x99f1d

Level 0 index = 0x1d (0x99f1d & 0x1f)
Level 1 index = 0x78 ((0x99f1d>>5)&0x7f)
Level 2 index = 0x19 ((0x991fd>>0xc)&0x7f)
Level 3 index = 0x1 ((0x99f1d>>0x13)&0x7f)
Level 4 index = 0 ((0x99f1d>>0x1a)&0x7f)
Level 5 index = 0 ((0x99f1d>>0x21)&0x7f)
Level 6 index = 0, 1, or 2 (normally, there are 3 copies of the data)

Note: With larger block sizes (for instance 128k), shifts and masks change accordingly. Correct shift and mask values are left as an exercise for the reader.

Step 15: End of Indirect Blocks

```
# mdb /tmp/blkptr1
> 0::blkptr
DVA[0]: vdev_id 0 / 3600
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: a0000000000
DVA[0]: :0:3600:a00:d
DVA[1]: vdev_id 0 / 1c0003600
DVA[1]:       GANG: FALSE  GRID:  0000  ASIZE: a0000000000
DVA[1]: :0:1c0003600:a00:d
LSIZE:  4000                            PSIZE: a00
ENDIAN: LITTLE                                    TYPE:  DMU dnode
BIRTH:  4acc              LEVEL: 0     FILL:  1400000000
CKFUNC: fletcher4                           COMP:  lzjb
CKSUM:  ad27cf4b34:ecbe6f671f03:c231a96352c039:7726f4dc30
> $q
# zdb -R zfs_fs:0:3600:a00:d,lzjb,4000 2> /tmp/dnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

Step 16: Display dnode_phys_t Array of File System Objects

```
# mdb /tmp/dnode
> 0,20::print -a -t zfs`dnode_phys_t
{
    0 uint8_t dn_type = 0  <-- first entry not used
... <-- output omitted
{
    200 uint8_t dn_type = 0x15  <-- DMU_OT_MASTER_NODE
    201 uint8_t dn_indblkshift = 0xe
... <-- output omitted
    240 blkptr_t [1] dn_blkptr = [
        {
            240 dva_t [3] blk_dva = [
                {
                    240 uint64_t [2] dva_word = [ 0x1, 0x63 ]
                }
    <-- output omited
```

Step 17: Master Node dnode_phys_t

```
> 240::blkptr
DVA[0]: vdev_id 0 / c600
...
LSIZE:  200                          PSIZE: 200
ENDIAN: LITTLE                                TYPE:  ZFS master node
BIRTH:  6             LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                      COMP:  uncompressed
CKSUM:  264216abd:f8ce291b17:347052edfaa6:79edede0bbfd
>

# zdb -R laciedisk:c4t0d0p1:c600:200:r 2> /tmp/zfs_master_node
Found vdev: /dev/dsk/c4t0d0p1
#
```

Step 18: Master Node ZAP Object

```
# mdb /tmp/zfs_master_node
> 0/J
0:              8000000000000003 <-- this is a microzap
> 0::print -t -a zfs`mzap_phys_t
{
    0 uint64_t mz_block_type = 0x8000000000000003

... <-- output omitted

c0::print -a -t zfs`mzap_ent_phys_t
{
    c0 uint64_t mze_value = 0x3  <-- the object id for the
root directory of the fs
... <-- output omitted
    ce char [50] mze_name = [ "ROOT" ]
}
```

Step 19: Root Directory dnode_phys_t

```
# mdb /tmp/dnode
> 3*200::print -a -t zfs`dnode_phys_t
{
    600 uint8_t dn_type = 0x14  <-- DMU_OT_DIRECTORY_CON
... <-- output omitted
    604 uint8_t dn_bonustype = 0x11  <-- DMU_OT_ZNODE (fron
... <-- output omitted
    640 blkptr_t [1] dn_blkptr = [
        {
            640 dva_t [3] blk_dva = [
                {
                    640 uint64_t [2] dva_word = [ 0x1, 0x51088 ]
                }
... <-- output omitted
>
```

Step 20: Root Directory znode_phys_t

```
> 6c0::print -a -t zfs`znode_phys_t  <-- 0x6c0 is offset of bonus b
{
    6c0 uint64_t [2] zp_atime = [ 0x47c08f1b, 0x31e41b57 ]
... <-- output omitted, ownership, other time stamps, size, etc.
}
> 6c0/Y
0x6c0:          2008 Feb 23 22:24:43    <-- when the dir was last ac
> 640::blkptr  <-- offset of blkptr in root directory dnode
DVA[0]: vdev_id 0 / a211000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 20000000000
...
LSIZE:  600                              PSIZE: 200
ENDIAN: LITTLE                                      TYPE:  ZFS directory
BIRTH:  46b6                LEVEL: 0     FILL:  100000000
CKFUNC: fletcher4                             COMP:  lzjb
CKSUM:  172e5bed24:7e94f1c3dba:179eeebee7275:325d97fad
> $q
```

Step 21: Root Directory Contents

```
# mdb /tmp/zfs_root_directory
> 0::print -a -t zfs`mzap_phys_t
{
    0 uint64_t mz_block_type = 0x8000000000000003
 <-- output omitted
    40 mzap_ent_phys_t [1] mz_chunk = [
        {
            40 uint64_t mze_value = 0x8000000000000004  <-- the ob
            ...
            4e char [50] mze_name = [ "foo" ]  <-- file in the root directo
    ]
}
... <-- output omitted
{
    440 uint64_t mze_value = 0x8000000000000015  <-- ls -i shows
    ...
    44e char [50] mze_name = [ "words" ]  <-- here's the file we wan
```

Step 22: Plain File dnode_phys_t

```
# mdb /tmp/dnode
> 15*200::print -a -t zfs`dnode_phys_t
{
    2a00 uint8_t dn_type = 0x13  <-- DMU_OT_PLAIN_FILE_CON
    2a01 uint8_t dn_indblkshift = 0xe
    2a02 uint8_t dn_nlevels = 0x2  <-- one layer of indirect blocks
    2a03 uint8_t dn_nblkptr = 0x1
    2a04 uint8_t dn_bonustype = 0x11  <-- DMU_OT_ZNODE ("w
... <- output omitted
    2a40 blkptr_t [1] dn_blkptr = [
        {
            2a40 dva_t [3] blk_dva = [
                {
                    2a40 uint64_t [2] dva_word = [ 0x2, 0x51054 ]
                }
... <-- output omitted
}
```

Step 23: Indirect Block for File Data

```
# zdb -R laciedisk:c4t0d0p1:a20a800:400:d,lzjb,4000 2> /tmp/indir
Found vdev: /dev/dsk/c4t0d0p1
#
# mdb /tmp/indirect
> 0::blkptr
DVA[0]: vdev_id 0 / a220000
DVA[0]:       GANG: FALSE  GRID:  0000  ASIZE: 2000000000000
DVA[0]: :0:a220000:20000:d
LSIZE:  20000                           PSIZE: 20000
ENDIAN: LITTLE                                   TYPE:  ZFS plain file
BIRTH:  46b6               LEVEL: 0     FILL:  100000000
CKFUNC: fletcher2                          COMP:  uncompressed
CKSUM:  281ad9d864b9dc57:79fe4143faf3e2b7:
5e064a117c12a92e:5b788125e084c6b2
>
```

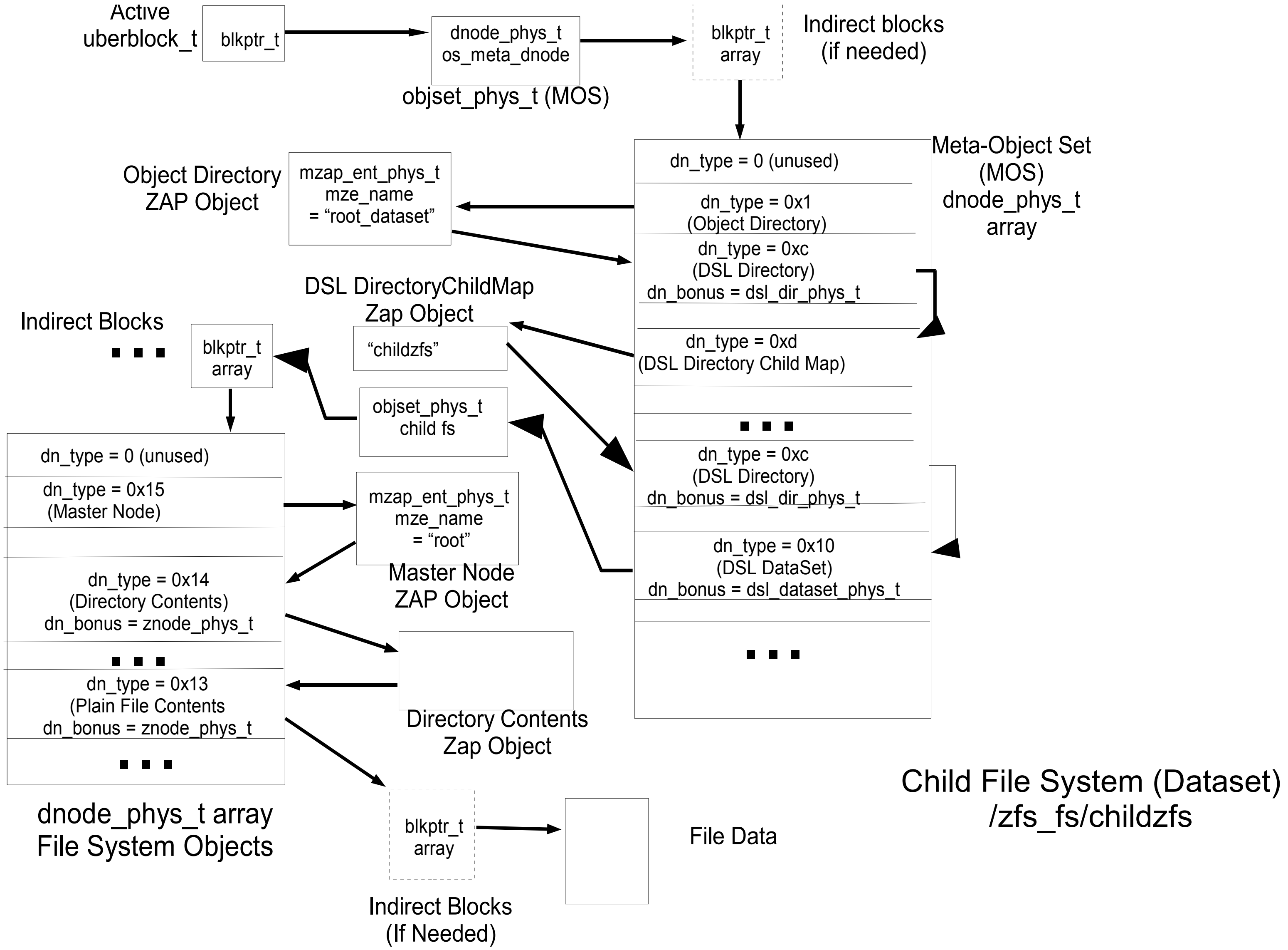
Step 24: The File Data

```
# zdb -R laciedisk:c4t0d0p1:a220000:20000:r
Found vdev: /dev/dsk/c4t0d0p1
10th
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
a
AAA
AAAS
Aarhus
```

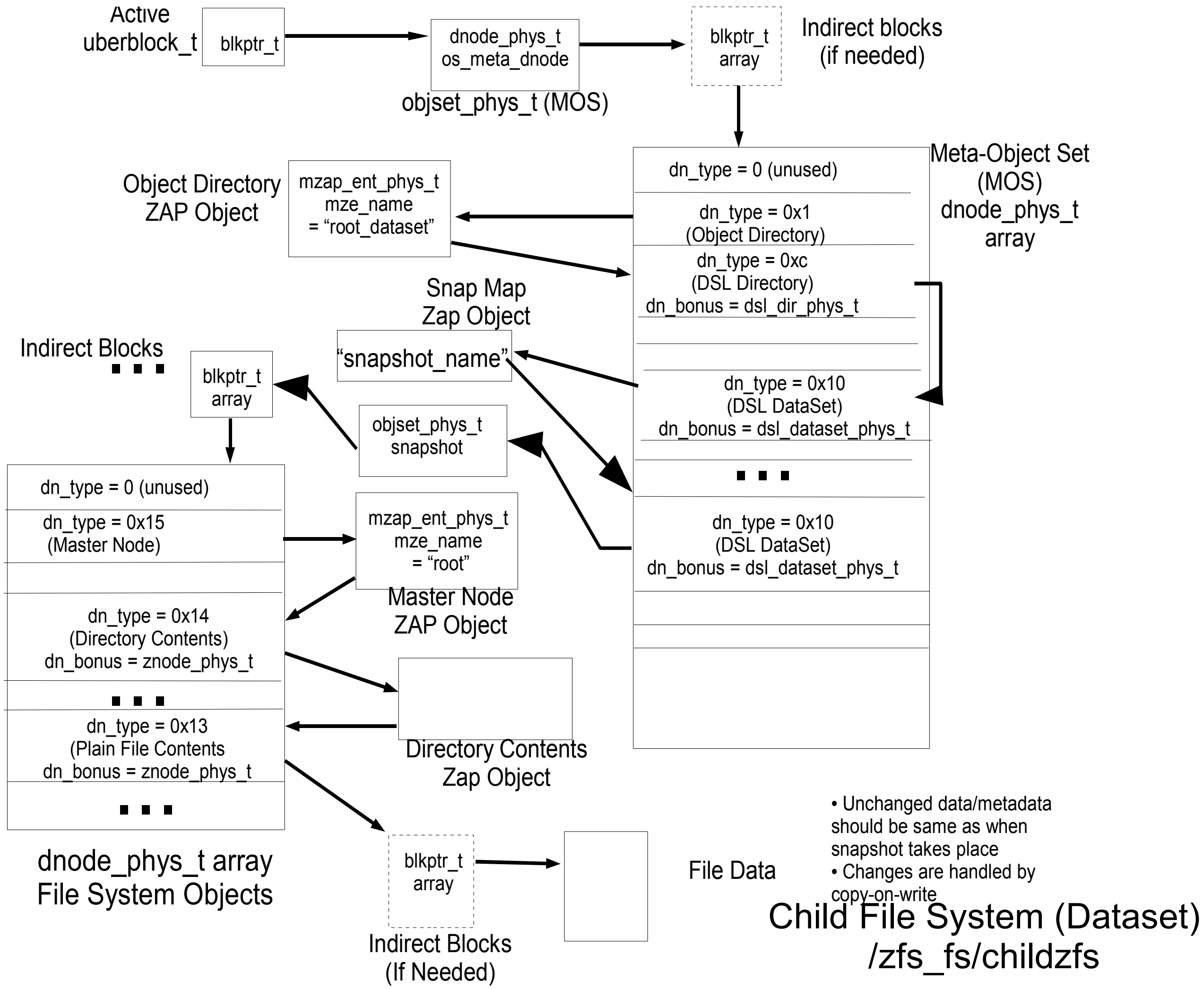
Mirrors

- Mirrors are copies
  - All meta data and the data itself is copied on all vdevs containing the mirror
  - Can use zdb -R *poolname:0.0:offset:size* or *zdb -R poolname:0.1:offset:size*
    - To get either side of a 2-way mirror

ZFS On-Disk Layout – Child Dataset



ZFS On-Disk Layout – Snapshot



Active uberblock_t

blkptr_t

dnode_phys_t
os_meta_dnode

objset_phys_t (MOS)

blkptr_t
array

Indirect blocks
(if needed)

Meta-Object Set (MOS)
dnode_phys_t array

dn_type = 0 (unused)

dn_type = 0x1
(Object Directory)

dn_type = 0xc
(DSL Directory)
dn_bonus = dsl_dir_phys_t

dn_type = 0x10
(DSL DataSet)
dn_bonus = dsl_dataset_phys_t

. . .

dn_type = 0x10
(DSL DataSet)
dn_bonus = dsl_dataset_phys_t

Object Directory
ZAP Object

mzap_ent_phys_t
mze_name
= “root_dataset”

Snap Map
Zap Object

“snapshot_name”

objset_phys_t
snapshot

Indirect Blocks
. . .

blkptr_t
array

dn_type = 0 (unused)

dn_type = 0x15
(Master Node)

dn_type = 0x14
(Directory Contents)
dn_bonus = znode_phys_t

. . .

dn_type = 0x13
(Plain File Contents
dn_bonus = znode_phys_t

. . .

dnode_phys_t array
File System Objects

mzap_ent_phys_t
mze_name
= “root”

Master Node
ZAP Object

Directory Contents
Zap Object

blkptr_t
array

Indirect Blocks
(If Needed)

File Data

- Unchanged data/metadata should be same as when snapshot takes place
- Changes are handled by copy-on-write

Child File System (Dataset)
/zfs_fs/childzfs

Conclusions

- Tools are a bit clumsy
  - Zdb gives too much or too little, and is not interactive
    - (But it tells you everything)
    - You need to understand the layout before much of the zdb output is useful
  - Mdb would be great if it supported something like:

    address::dprint -d *decompression* -l *logical_size type*
  - The ability to use kernel CTF information on binary data makes this much simpler
    - Works with UFS to examine on-disk superblock, cylinder group blocks, inodes, etc.
    - Should work with other file systems that have CTF
- Partly because of compression, much of the metadata is read with 1 or 2 reads

Future Work

- Change mdb to allow loading of specific CTF
  - ::loadctf [-k] [module_name]
- Do the walk for the data structures of an open file

References

- ZFS On-Disk Specification paper at: http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf
- Source Code