NFSv4 M. Eisler
Internet-Draft NetApp
Intended status: Standards Track October 18, 2010
Expires: April 21, 2011

Storage De-Duplication Awareness and Sub-File Caching in NFS
draft-eisler-nfsv4-pnfs-dedupe-01.txt

Abstract

This Internet-Draft describes a means to add awareness of de-duplication storage to NFS in order to save resources on NFS client and to reduce bandwidth for servicing READ and WRITE operations. The means presented leads to a second benefit of providing sub-file, block-granular caching.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 21, 2011.

Copyright Notice

Table of Contents

1. Introduction and Motivation

De-duplication is an emerging trend in the data storage. De-duplication means that two files that have common content derive that content from a common location on the same storage device. As a result, the total storage used is less than the total length of each file. De-duplication is also called folding.

Some file systems have the capability to avoid allocation of storage space when the value of each byte in a contiguous range is zero. Such a range of a file in such a file system is called a "hole", and a file with one or more holes is called a "sparse" file. Sparse files represent a trivial form of de-duplication since the value of every hole of X bytes in length is the common.

De-duplication is accomplished in several ways including,

- Hierarchical de-duplication, where one file is derived from another, usually by one file starting of as copy of another, but zero, or nearly zero bytes of data are actually copied or moved. Instead, the two files share common blocks of data storage. An example is a snapshot, where a snapshot is made of a file system, such that the snapshot and active file system are equal at the time snapshot is taken, and share the same data storage, and thus are effectively copies that involve zero or near zero movement of data. As the source file system changes, the number of shared blocks of data storage reduces. A variation of this is a writable snapshot (aka clone) which is taken of a file system. In this variation as the source and cloned file systems each change, there are fewer shared blocks.

- In-line de-duplication, where a storage access protocol initiator (e.g. an NFS client) creates content via write operations, and the target of the storage access protocol checks if the content being written is duplicated some where else on the target’s storage. If so, the data is not written, but instead the logical content refers to the duplicate.

- Background de-duplication, where a background task on the storage access protocol target scans for duplicate blocks, and frees all but one of the duplicates, mapping the pointers to the now free blocks to the remaining duplicate.

The use of de-duplicated storage does not require changes to the NFS protocol. However if the NFS client is caching content from an NFS server that provides access to de-duplicated files, without changes to the protocol, inefficient use of the resources like memory and network bandwidth will result. E.g., two files of length 1024 bytes

are exactly the same and are de-duplicated. The client reads, and caches the first file. A process on the client requests to read the second file. If the client were aware the second file was a duplicate of the first, it would not have read the second file, nor would it have to cache the second file. A classic use case is hypervisors, which switch between multiple guest operating systems on a single physical computer. If each of these guest operating systems were cloned from a single source, or if each guest was installed from the same operating system installation image, then much of the data of each guest might be highly de-duplicated. De-duplication awareness is consistent with the typical reasons for deploying a hypervisor: reducing costs by reducing utilization of memory, computer cycles, and network.

Sub-file caching is most useful when two conditions are met:

- Multiple NFS clients need to access the same file.
- At least one client is modifying the same file, provided this client updates a relatively small subset of the file.

Under these two conditions many situations can occur where whole file caching, as enabled by NFSv4 delegations, at best provides no benefit and at worst presents a drawback. Examples include:

- One client frequently updates range X of a file, and another client frequently reads range Y of a file where X and Y do not overlap. With whole file delegations, each client enters a cycle of obtain a delegation, process a recall, perform a READ or WRITE to the server, with delegations providing no benefit, and thus resources being unnecessarily consumed on the client and server.
- Two clients randomly read and write different ranges of the same file, and for a sufficiently large file, the probability that they need the to access overlapping ranges is very small. Again, with whole file delegations, the clients are locked in the same cycle as above.

This document describes a method by which NFSv4.1 clients can be aware of de-duplicated storage for optimizing READ requests. As proposed, optimization of READ requests not require a new minor version of NFSv4. Instead, it requires several new layout types, and thus uses the pNFS protocol [2]. The approach presented here for de-duplication awareness is easily extended to support sub-file caching at arbitrary granularities and for abitrary sets of byte ranges of a file.

This document also describes a method by which NFSv4.x clients can

optimize WRITE requests. The method does require a minor version of NFS.

The XDR description is provided in this document in a way that makes it simple for the reader to extract into a ready to compile form. The reader can feed this document into the following shell script to produce the machine readable XDR description of the de-duplication layout:

```
#!/bin/sh
grep "^  *///" | sed 's?^  *///  ??' | sed 's?^.*///??'
```

I.e. if the above script is stored in a file called "extract.sh", and this document is in a file called "spec.txt", then the reader can do:

```
 sh extract.sh < spec.txt > dd.x
```

The effect of the script is to remove leading white space from each line of the specification, plus a sentinel sequence of "///".

2. Terminology

- Source file, the file that contains the de-duplicated data.
- Target file, the file the client has opened.
- Block, the smallest unit of de-duplication or caching that the server is willing to support.
- Slab, a byte range that refers to lists of other byte ranges that contain de-duplicated data (either in whole, or part). A slab can refer to a lists of smaller slabs, or lists of blocks.
- Regular file: An object of file type NF4REG or NF4NAMEDATTR.

3. De-Duplication

3.1. Scope of De-Duplication

This document only de-duplicates the data contents of regular files. Everything else is considered metadata, and de-duplication of metadata is not considered in this document. [[Comment.1: Some metadata, including the contents of directories and symbolic links, as well as attributes (e.g. ACLs) are practical to de-duplicate, but not at the granularity of fixed sized blocks. A future revision of

this document might address de-duplication of metadata.]]

De-duplication awareness of regular file content in NFS has two aspects:

- Optimizing READ requests. Here the goal is to avoid reading a pattern of data the client might already have cached.

- Optimizing WRITE requests. Here the goal is to avoid writing a pattern of data the server might already have elsewhere, such that the pattern can be de-duplicated.

3.2. READ Optimization via De-Duplication and pNFS

Providing awareness of de-duplication to clients needs to be practical. If the data structures the server provides to the client are not compact, or require expensive processing and/or network bandwidth, then de-duplication awareness is not practical. The approach presented in this document uses leaf bitmaps to indicate whether a byte range of a file has been de-duplicated, and if so from what offset of what file. Since the granularity of de-duplication will vary by implementation, and by file, the NFS server has the option of providing indirect bitmaps that refer to bitmaps of finer grained byte ranges. An indirect bitmap can refer to another indirect bitmap or a leaf bitmap.

As noted in Section 1, de-duplication can be the result of hierarchical, inline, or background processes. This document presents an approach to providing awareness of de-duplication allows servers to optimize for any approach.

NFSv4.1 introduces pNFS, which allows clients to access data from multiple storage devices. This means that the NFS server is distributed across a set of nodes on a network. Such a server might be capable of de-duplication among the server’s nodes. The de-duplication awareness feature will allow servers to present awareness of cross-node de-duplication to NFS clients.

3.2.1. The Definition of De-Duplication Layouts

3.2.1.1. Name of De-Duplication Striping Layout Type

There are multiple de-duplication layout types, in order to support multiple levels of indirection plus a leaf level. Since the maximum sized file in pNFS is 2^64 - 1 bytes, a total of 63 levels of indirection are provided.

There are two sets of de-duplication layout types.

- Within the first set, the name of the top-level de-duplication layout type is LAYOUT4_DEDUP_TOP. The names of the remaining de-duplication layout types are in this set LAYOUT4_DEDUP_LEVEL_<xx>, where <xx> is a two digit decimal number that ranges between 02 and 64. The server MUST NOT return LAYOUT4_DEDUP_LEVEL_<xx> in the response to a GETATTR request for the fs_layout_type attribute.

- Within the second set, the name of the top-level de-duplication layout type is LAYOUT4_DEDUP_ROC_TOP. The names of the remaining de-duplication layout types are in this set LAYOUT4_DEDUP_ROC_LEVEL_<xx>, where <xx> is a two digit decimal number that ranges between 02 and 64. The server MUST NOT return LAYOUT4_DEDUP_LEVEL_<xx> in the response to a GETATTR request for the fs_layout_type attribute.

3.2.1.2. Value of De-Duplication Striping Layout Type

See Section 7.

3.2.1.3. Definition of the da_addr_body Field of the device_addr4 Data Type

```
///  %#include "nfs4_prot.h"
///
///  %/* Encoded in the da_addr_body field. */
///
///  union dd_layout_addr switch (bool ddla_simple) {
///    case TRUE:
///      multipath_list4 ddla_simple_addr;
///    case FALSE:
///      layouttype4     ddla_complex_addr;
///  };
```

Figure 1

The device address is only used in leaf layouts, and even then, only when cross server-node de-duplication is in effect. There are two types of device addresses, a simple network address, with zero or more alternate addresses for multipathing, or a complex address which is the value of another layout type. The value of ddla_complex_addr.ddldp_ltype MUST NOT be LAYOUT4_DEDUP_TOP or any of LAYOUT4_DEDUP_LEVEL_<xx>.

3.2.1.4. Definition of the loh_body Field of the layouthint4 Data Type

```
///  enum dd_layout_hint_care4 {
///
///         DD4_CARE_STRIPE_UNIT_SIZE    = 0x040,
///         DD4_CARE_STRIPE_UNIT_ALIGN   = 0x100
///  };
///  %
///  %/* Encoded in the loh_body field of type layouthint4: */
///  %
///  struct dd_layouthint4 {
///         uint32_t       ddlh_care;
///         length4        ddlh_stripe_unit_size;
///         length4        ddlh_stripe_unit_align;
///  };
```

Figure 2

The layout-type specific content for the LAYOUT4_DEDUP_TOP layout type is composed of three fields. The first field, ddlh_care, is a set of flags indicating which values of the hint the client cares about. If DD4_CARE_STRIPE_UNIT_SIZE is set, then the client indicates in the second field, preferred unit of granularity for de-duplication in bytes. If DD4_CARE_STRIPE_UNIT_ALIGN is set, then the client indicates in the third field, the preferred minimum alignment de-duplicated units. For example, if the client specifies ddlh_stripe_unit_size as 1024, and ddlh_stripe_unit_align as 128, then if two files have in common content a string of bytes that is 1024 bytes long, and the string is at offset zero in the first file, and offset 1024 + 128 = 1152 in the second file, then the client would like the server to de-duplicate the common 1024 byte string. Note that the leaf layouts returned by the server are unable to indicate byte ranges that are not whole multiples of the unit size the server uses, so if the server accepts a layout hint with ddlh_stripe_unit_align less than ddlh_stripe_unit_size, it will report units that are equal to ddlh_stripe_unit_align. If the client specifies a value in ddlh_stripe_unit_align that is greater than the value of ddlh_stripe_unit_size, the server will ignore the ddlh_stripe_unit_align hint.

3.2.1.5. Definition of the loc_body Field of the layout_content4 Data Type

```
///  %/*
///  %/* How the bits of each element
///  % * of ddll_blockmap are split up
///  % */
///  const DDLL4_BLKMAP_MASK_ACTIVE      = 0x8000000000000000;
///
///  %/* The remain bits follow DDLL4_BITS_* */
///  const DDLL4_BLKMAP_MASK_PARTITIONED = 0x7FFFFFFFFFFFFFFF;
///
///  %/* These constants index into ddll_bmap_partition */
///  const DDLL4_BITS_FOR_DEVID_IDX   = 0;
///  const DDLL4_BITS_FOR_FH_IDX      = 1;
///  const DDLL4_BITS_FOR_BLK_NUM_IDX = 2;
///
///  struct dd_layout_leaf4 {
///    length4   ddll_block_size;
///
///  % /* ddll_blockmap_partition[0-2] MUST add up to 63 */
///
///    opaque    ddll_blockmap_partition[4];
///    verifier4 ddll_fhsuffix;
///    nfs_fh4   ddll_fhlist<>;
///    uint64_t  ddll_change_attr<>;
///    deviceid4 ddll_devlist<>;
///    uint64_t  ddll_blockmap<>;
///  };
///
///  struct dd_layout_indirect4 {
///    length4     ddli_slab_size;
///    layouttype4 ddli_next_level;
///    bitmap4     ddli_bitmap;
///  };
///
///  union dd_layout4_u switch (bool ddl_is_leaf) {
///    case TRUE:
///      dd_layout_leaf4     ddl_leaf;
///    case FALSE:
///      dd_layout_indirect4 ddl_indirect;
///  };
///  struct dd_layout4 {
///    offset4      ddl_firstoff;
///    offset4      ddl_lastoff;
///    dd_layout4_u ddl_u;
///  };
```

Figure 3

The first fields further bound the layout.

- ddl_firstoff, the first offset in the file that the layout has de-duplication information for. The relationship between the lo_offset field of the layout4 data type that envelops the de-duplication layout and ddl_firstoff is that ddl_firstoff MUST be greater than or equal to lo_offset. If ddl_firstoff is not equal to lo_offset, then this means that the byte range from lo_offset through ddl_firstoff - 1 inclusive either has not been de-duplicated or the server has decided to not provide the information. The value of the field ddl_firstoff MUST be a whole multiple of ddli_slab_size or ddll_block_size.

- ddl_lastoff, the last offset in the file that the layout has de-duplication information for. Field ddl_lastoff MUST be greater than or equal to ddl_firstoff. Field ddl_lastoff MUST be less than or equal to lo_offset + lo_length - 1. If the difference between ddl_lastoff and lo_offset + lo_length - 1 exceeds zero, then this means that byte range from offset ddl_lastoff + 1 through lo_offset + lo_length - 1 inclusive either has not be been de-duplicated or the server has decided to not provide the information. The value of the ddl_lastoff + 1 MUST be a whole multiple of ddli_slab_size or ddll_block_size, even if this means ddl_lastoff goes beyond the end of file.

The remainder of the de-duplication layout is either a leaf layout or an indirect layout.

An indirect layout consists of,

- ddli_slab_size is the length, in bytes of each slab represented by the ddli_bitmap bitmap array.

- ddli_next_level is the layout type the NFS client MUST use when using LAYOUTGET to get finer grained de-duplication information about the de-duplication of one or more slabs. This field SHOULD be one of LAYOUT4_DEDUP_LEVEL_<xx>. The use of ddli_next_level provides a hint to the server for what slab or block size to use on the next level of de-duplication.

- ddli_bitmap is a bitmap. If bit N is set in ddli_bitmap, then this means that slab N has de-duplicated content. Each bit respects a byte range (a slab) of size ddli_slab_size, such that ddl_firstoff is the start of the first slab (slab zero, relative to ddl_firstoff). Slab N represents the byte range ddl_firstoff + N * ddli_slab_size to ddl_firstoff + (N + 1) * ddli_slab_size - 1,

inclusive. The field ddli_bitmap is an array of elements each consisting of a 32 bit unsigned integer. The number of elements in ddli_bitmap MUST be greater than or equal to ((((ddl_lastoff - ddl_firstoff) + 1) / ddli_slab_size) / 32) rounded up to the next whole number.

A leaf layout consists of,

- ddll_block_size is the length, in bytes of each slab represented by the ddll_blockmap array.

- ddll_blockmap_partition is an array of bytes, the first three of which are inspected by the client. This array indicates how each element of ddll_blockmap is partitioned.

- ddll_fhlist is an array of zero or more filehandles. Each element of ddll_blockmap can correspond to a filehandle in ddll_fhlist. Each filehandle represents a source file that has a de-duplicated block that it shares with the target file. If the array is of zero length, then the source file for all de-duplicated blocks is the target file.

- ddll_fhsuffix MUST be appended to each filehandle in ddll_fhlist that the client uses for READ or LAYOUTGET operations. This allows the server to detect if the client is using an invalid layout.

- ddll_change_attr is an array of zero or more change attributes. If the value of the layout type is between LAYOUT4_DEDUP_TOP and LAYOUT4_DEDUP_LEVEL_64, inclusive, then the length of ddll_change_attr MUST be greater than or equal to 1. If the value of the layout type is between LAYOUT4_DEDUP_ROC_TOP and LAYOUT4_CACHE_LEVEL_64, inclusive, then the length of ddll_change_attr MUST be zero.

- If ddll_change_attr is not zero in length, then each element corresponds an element in ddll_fhlist with the same position in the array. I.e. ddll_change_attr[i] is the change attribute for the source file identified by ddll_fhlist[i]. If the array is of zero length, then for each byte range represented by an element of ddl_blockmap that has DDLL4_BLKMAP_MASK_ACTIVE set, the server promises to recall the layout of the byte range before the data on the range mapped from the source file (represented by an element of ddl_fhlist) is changed and before data on range of the target file changed. If the ddll_fhlist array is of zero length, and the ddll_change_attr array has one element, then ddll_change_attr[0] is the change attribute for the source file, which also happens to be the target file.

- ddll_devlist is an array of zero or more device IDs, for the purpose of enabling cross-node de-duplication. Each element of ddll_blockmap can correspond to a device ID in ddll_devlist. Each device ID represents a device that has a source file with a de-duplicated block. The device ID is always for a LAYOUT4_DEDUP_TOP device, and can either map to a network address of an MDS, or a non-de-duplication layout type. The device ID will map to an MDS network address if the source file has not been striped. Otherwise, the device ID will be the layout type used for striping the file. By providing the layout type, the client does not have to send a GETATTR request on the source file for fs_layout_type attribute.

- ddll_blockmap is an array of elements, each a 64 bit unsigned integer. Each element corresponds to a block of size ddll_block_size. E.g., the first element, ddll_blockmap[0] corresponds to the byte range, ddl_firstoff through ddl_firstoff + ddll_block_size - 1 inclusive.

  * If ddll_blockmap[i] & DDLL4_BLKMAP_MASK_ACTIVE is non-zero, then this element corresponds to a block that is de-duplicated. Otherwise, the element does not correspond to a de-duplicated block, and the rest of the element is undefined.

  * The mask ddll_blockmap[i] & DDLL4_BLKMAP_MASK_PARTITIONED represents a bit field that is partitioned according to the content of ddll_blockmap_partition.

    The element ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] indicates how many bits at the start of the bit field are for indexing into the ddll_devlist array. The number of elements in ddll_devlist MUST be less than or equal to 2^ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX]. If ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] is zero, then this means that the blocks of the source file come from the same MDS as the target file.

    The element ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] indicates how many bits in the middle of the bit field are for indexing into the ddll_fhlist array. The number of elements in ddll_fhlist MUST ne less than or equal to 2^ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX]. If ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] is zero, this means that the source file is the same as the target file in every element of ddll_blockmap_partition.

    The element ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] indicates how many bits at the end of the bit field correspond

to an absolute block number into the source file. The absolute offset is calculated by computing the product of ddll_block_size and the absolute block number. If ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] is zero, then this means the absolute block number of the source is the same as the absolute block number of the target.

The dynamic partitioning of the ddll_blockmap element allows for several optimizations. If the de-duplication in the range identified by the layout is due to hierarchical de-duplication, then there is no need for a block number, so ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] will be zero. If there is no cross node de-duplication in the range then ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] will be zero. If all the de-duplication in the range is confined to the target file, i.e. the duplicate blocks were only in the target file and no other file, then ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] will be zero.

An outline for an algorithm for processing a read() system call when the potential for de-duplicated data exists follows. This algorithm illustrates how the layout is interpreted. In this algorithm, we assume that the client always starts with a layout that spans the entire file.

```
/*
 * Returns a vector call "result" of elements
 * containing key / value pairs of ((offset,
 * length), (status, source_mds, source_fh,
 * source_offset)).
 */

dedupe_read(read_offset, read_length, target_fh,
    layout4 logr_layout[]) {

  if (number of elements in logr_layout == zero) {
    result[(read_offset, read_length)] =
        NO_DEDUP_AVAILABLE;

    return result;
  }

  for i from the end of logr_layout to start {
    if (logr_layout[i].lo_offset > read_offset) {
      continue;
    }
```

```
      /* check for range split across segments */
      if (logr_layout[i].lo_length <
          read_length) {

        read_offset_A = read_offset;
        read_length_A = logr_layout[i].lo_length;
        read_offset_B = logr_layout[i+1].lo_offset;
        read_length_B = read_length -
          read_length_A;

        result[(read_offset_A, read_length_A)] =
          dedupe_read(read_offset_A, read_length_A,
          target_fh, logr_layout);

        result[(read_offset_B, read_length_B)] =
          dedupe_read(read_offset_B, read_length_B,
          target_fh, logr_layout);

        return result;
      }

      /*
       * If requested offset exceeds last offset of this layout
       * segment, then we have no de-dupe opportunity.
       */
      if (read_offset > ddl_lastoff) {
        result[(read_offset, read_length)] =
          NO_DEDUP_AVAILABLE;
        return result;
      }

      last_offset = read_offset + read_length - 1;

      if (last_offset > ddl_lastoff) {
        /* we cannot de-dupe the entire range */

        result[(ddl_lastoff + 1, last_offset -
          ddl_lastoff)] = NO_DEDUP_AVAILABLE;
        last_offset = ddl_lastoff;
      }
      if (read_offset < ddl_firstoff) {
        /* we cannot de-dupe the entire range */

        result[(read_offset, ddl_firstoff -
          read_offset)] = NO_DEDUP_AVAILABLE;
        read_offset = ddl_firstoff;
      }
```

```
      if (ddl_is_leaf == FALSE) {
        /*
         * Indirect layout. See if the slabs that correspond
         * to the affected range are de-duplicated.
         */

        let trunc_read_off = read_offset truncated
          to next lowest multiple of
          ddli_slab_size;

        let round_last_off = (last_offset rounded
          to next highest multiple of
          ddli_slab_size) - 1;

        first_bit = trunc_read_off /
          ddli_slab_size;
        last_bit =
          (round_last_off + 1) / ddli_slab_size;

        for (j = first_bit; j++; j <= last_bit) {
          k = j / 32;
          l = j mod 32;
          bit = l << 1;

          if (j == first_bit) {
            read_offset_A = read_offset;
            read_length_A = trunc_read_off +
              ddli_slab_size - read_offset;

          } else {
            read_offset_A = ddl_firstoff + (j *
              ddli_slab_size);
            read_length_A = ddli_slab_size;
          }

          if ((ddli_bitmap[k] & bit) == 1) {
            next_layout_off = j * ddli_slab_size +
              trunc_read_off;

            next_layout_length = ddli_slab_size;
            next_layout_type = ddli_next_level;

            if (client does not have layout for
                (next_layout_off,
                next_layout_length, and
                ddli_next_level) {

               send a LAYOUTGET request;
```

```
            }
            let logr_layout_A = logr_layout array
                of layout for (next_layout_off,
                next_layout_length,
                next_layout_type);

            result[(read_offset_A, read_length_A)]
              = dedupe_read(read_offset_A,
              read_length_A, target_fh,
              logr_layout_A);

          } else {
            result[(read_offset_A, read_length_A)]
              = NO_DEDUP_AVAILABLE;

          }
        }
      } else {
        /* process a leaf layout */

        /*
         * determine the masks for block number, filehandle index, and
         * device ID index.
         */
        let trunc_read_off = read_offset truncated
          to next lowest multiple of
          ddll_block_size;

        let round_last_off = (last_offset rounded
          to next highest multiple of
          ddll_block_size) - 1;

        bits_for_blknum = ddll_blockmap_partition
          [DDLL4_BITS_FOR_BLK_NUM_IDX];

        mask_for_blknum = 0;
        for (j = 0; j < bits_for_blknum; j++) {
          mask_for_blknum = (mask_for_blknum
            << 1) | 1;
        }

        bits_for_fh = ddll_blockmap_partition
          [DDLL4_BITS_FOR_FH_IDX];

        mask_for_fh = 0;
        for (j = 0; j < bits_for_fh; j++) {
          mask_for_fh = (mask_for_blknum <<
            1) | 1;
```

```
        }

        mask_for_fh = mask_for_fh <<
          bits_for_blknum;

        bits_for_dev = ddll_blockmap_partition
          [DDLL4_BITS_FOR_DEVID_IDX];

        mask_for_dev = 0;
        for (j = 0; j < bits_for_dev; j++) {
          mask_for_dev = (mask_for_dev << 1)
            | 1;
        }
        mask_for_dev = mask_for_dev <<
          (bits_for_blknum + mask_for_fh);

        if ((bits_for_blknum + bits_for_fh +
            bits_for_dev) != 63) {

          result[(read_offset, read_length)] =
            CORRUPT_LAYOUT;

          return result;
        }

        first_block = trunc_read_off /
          ddll_block_size;
        last_block = (round_last_off + 1) /
          ddll_block_size;
        slopoff = read_offset - trunc_read_off;
        sloplen = round_last_off - last_offset;

        read_offset_A = trunc_read_off;

        for (j = first_block; j++, read_offset_A +=
            ddll_block_size; j <= last_block) {

          if (ddll_blockmap[j] &
              DDLL4_BLKMAP_MASK_ACTIVE) {

            blockmap = ddll_blockmap[j] &
              DDLL4_BLKMAP_MASK_PARTITIONED;

            source_length = ddll_block_size;
            source_change = 0;
            source_dev = 0;

            if (mask_for_blknum == 0) {
```

```
            source_offset = ddl_firstoff + j *
              ddll_block_size;
          } else {
            source_offset = (blockmap &
              mask_for_blknum) * ddll_block_size;
          }

          if (j == first_block) {
            source_offset += slopoff;
            read_offset_B = read_offset;
          } else {
            read_offset_B = read_offset_A;
          }

          if (j == last_block) {
            source_length -= sloplen;
          }

          if (mask_for_fh == 0) {
            source_fh = target_fh;

            if (number of elements in
                ddll_change_attr > 0) {
              source_change = ddll_change_attr[0];
            }
          } else {
            fhidx = (blockmap & mask_for_fh) >>
              bits_for_blknum;
            source_fh = ddll_fhlist[fhidx];
            if (number of elements in
                ddll_change_attr > 0) {
              source_change =
                ddll_change_attr[fhidx];
            }
          }
          read_source_fh = source_fh concatenated
            with ddll_fhsuffix;
          source_ltype = 0;
          source_mds = MDS of target_fh;
          if (mask_for_dev != 0) {
            devidx = (blockmap & mask_for_dev) >>
              bits_for_blknum;
            source_dev = ddll_devlist[devidx];

            if (client does not have device
                address for source_dev) {
              send a GETDEVICEINFO
                (LAYOUT4_DEDUP_TOP, source_dev);
```

```
                }

                if (ddla_simple from GETDEVICEINFO is
                    TRUE) {
                  let source_mds be an element of
                    ddla_simple_addr;
                } else {
                  source_ltype = ddldp_ltype;

                  if (client does not have layout for
                      (source_mds, source_fh,
                      source_ltype, source_offset,
                      source_length)) {

                    send a LAYOUTGET request for
                      (read_source_fh, source_ltype,
                      source_dev, source_offset,
                      source_length) to target_fh's
                      MDS;

                    cache LAYOUTGET result;
                  }

                  if (client still does not have
                      layout for (source_mds, source_fh,
                      source_ltype, source_offset,
                      source_length)) {
                    source_ltype = 0;
                  } else {
                    let source_layout = the layout
                      from cache;
                  }
                }
              }

              if (source_change == 0 || client has
                  delegation on source_fh) {

                if ({source_fh, source_mds,
                    source_offset, source_length} in
                    cache) {

                  result[(read_offset_B,
                    source_length)] =

                    (SATISFY_READ_FROM_CACHE,
                    source_mds, source_fh,
                    source_offset;)
```

```
                } else {
                  if (source_ltype == 0) {
                    if (read_source_fh not yet open)
                    {
                      send an OPEN request for
                        read_source_fh;
                    }
                    send a { PUTFH read_source_fh,
                      READ source_offset,
                      source_length } request to
                      source_mds;

                    enter results in cache;

                  } else {
                    read from read_source_fh,
                      source_offset, source_length
                      according to source_layout;

                    enter results in cache;
                  }
                  result[(read_offset_B,
                    source_length)] =
                    (SATISFY_READ_FROM_CACHE,
                    source_mds, source_fh,
                    source_offset);

                }
              } else {
                if ({source_mds, source_fh,
                    source_offset, source_length} in
                    cache) {

                  send a { PUTFH source_fh, GETATTR
                    change } request to source_mds;

                  if (change attribute ==
                      source_change) {

                    result[(read_offset_B,
                      source_length)] =
                      (SATISFY_READ_FROM_CACHE,
                      source_mds, source_fh,
                      source_offset);

                  } else {
                    result[(read_offset_B,
                      source_length)] =
```

```
                    (STALE_DEDUP_LAYOUT,
                    source_mds, source_fh,
                    source_offset);

                }
              }
            }
          }
        }
      }
      return result;
    }

    /* should never get here */
    result[(read_offset, read_length)] =
      CORRUPT_LAYOUT;

    return result;
  }
```

Figure 4

There is a trade off between resources (space and time) used for providing de-duplication layouts (especially leaf layouts) and resources for redundant caching of de-duplicated storage. E.g., if a client has to descend through 52 levels of caching to avoid caching a single 4096 byte block twice, then it is not cost effective for the server to return a layout. On the other hand, if 99% of a file is using de-duplicated storage, then having a complete block map for a one gigabyte file, or at least the parts of the file the client wants to cache, is more effective than redundantly caching nearly one gigabyte of storage.

3.2.1.6. Definition of the lou_body Field of the layoutupdate4 Data Type

```
///  %/*
///  % * LAYOUT4_DEDUP_TOP or any of LAYOUT4_DEDUP_LEVEL_<xx>.
///  % * Encoded in the lou_body field of type layoutupdate4:
///  % *      Nothing. lou_body is a zero length array of octets.
///  % */
///  %
```

Figure 5

The LAYOUT4_DEDUP_TOP and LAYOUT4_DEDUP_LEVEL_<xx> layout types have no content for lou_body filed of the layoutupdate4 data type.

3.2.1.7. Storage Access Protocols

The LAYOUT4_DEDUP_TOP and LAYOUT4_DEDUP_LEVEL_<xx> layout types use NFSv4.1 operations (and potentially, operations of higher minor versions of NFSv4, subject to the definition of a minor version of NFSv4) to access de-duplicated data. The de-duplication layout types do not affect access to storage devices. Thus a client might be able to obtain both a de-duplication layout type and a non-de-duplication layout type (e.g., LAYOUT4_NFSV4_1_FILES, LAYOUT4_OSD2_OBJECTS, or LAYOUT4_BLOCK_VOLUME) on the same regular file.

3.2.1.8. Revocation of Layouts

Servers MAY revoke de-duplication layouts. A client using a de-duplication layout SHOULD check if the change attribute of the source file has changed. The use of the ddll_fhsuffix will prevent clients using revoked de-duplication layouts from using potentially stale information. Attempts to use filehandles with the value of ddll_fhsuffix appended, will result in NFS4ERR_STALE.

3.2.1.9. Recovery

[[Comment.2: it is likely this section will follow that of the files layout type specified in the NFSv4.1 specification.]]

3.2.1.9.1. Failure and Restart of Client

TBD

3.2.1.9.2. Failure and Restart of Server

TBD

3.2.1.9.3. Failure and Restart of Storage Device

TBD

3.2.2. Negotiation

A pNFS client sends a GETATTR request for the fs_layout_type attribute to see if the LAYOUT4_DEDUP_TOP layout type is supported.

3.2.3. Operational Recommendation for Deployment

Deploy the de-duplication layouts when it a significant fraction of data storage is de-duplicated.

3.3. WRITE Optimization When De-Duplication Is Present

There are two goals

- Avoid a WRITE of a pattern if client knows that server has stored that pattern somewhere else besides the combination of target file and byte range. the server

- Even if the client does not know if the pattern is stored somewhere, provide a hint to the server that allows it to quickly determine if the pattern is present.

Accomplishing the former merely requires an operation that refers the server to a byte of a file it has stored. One way to is to leverage the proposed COPY operation [3]. Accomplishing the latter can be done by the client providing checksums of byte range it would like to avoid writing. However, to do so would require that client and server agree on checksum algorithm, which has the practical problem that clients and servers with pre-existing de-duplication features are likely to not agree on the checksum algorithm. For this reason, this version of the document does not pursue the second goal.

One caveat using COPY to achieve the first goal (avoiding a WRITE when the client knows the server has stored the pattern elsewhere) is that there is a window between the time the client has cached a byte range of the source file and the time the server receives the COPY request. The use of a de-duplication layout that guarantees a recall before the relevant byte range of the source file is changed. Note that this guarantee is only present if ddll_change_attr is of zero length. The client requires a way to force the server to return such de-duplication layouts. When the client requests the top level de-duplication layout with a type equal to LAYOUT4_DEDUP_TOP | LAYOUT4_DEDUP_RECALL_ON_CHANGE. The value of LAYOUT4_DEDUP_RECALL_ON_CHANGE is mask with one bit set:

```
///  const LAYOUT4_DEDUP_RECALL_ON_CHANGE = 0x40;
```

Figure 6

4. Sub-File Caching

Sub-file caching is built using the concepts and data structures defined in Section 3.2, which introduces a set of layout types that allow customers to optimize READ operations when the NFS client and server support de-duplication. Sub-file caching provides a subset of the functionality defined by the LAYOUT4_DEDUP_ROC_TOP layout type (and layout types LAYOUT4_DEDUP_ROC_LEVEL_02 through

LAYOUT4_DEDUP4_ROC_LEVEL_64 inclusive). The primary similarity is that a sub-file cache leaf layout provides a guarantee that if a block is mapped in the bitmap, then the server will recall a layout covering that block before allowing the block to be modified. The primary difference is that sub-file cache leaf layout does not have de-duplication references.

4.1. Value of the Sub-File Caching Layout Type

See Section 7.

4.2. Sub-File Caching Indirect Layouts

Indirect layouts for sub-file caching have the same format and data types as indirect layouts for de-duplication.

4.3. Sub-File Caching Leaf Layouts

Leaf layouts for sub-file caching have the same format and data types as indirect layouts for de-duplication. However, there are the following restrictions:

- The value of ddll_blockmap_partition[DDLL4_BITS_FOR_DEVID_IDX] MUST be zero.
- The value of ddll_blockmap_partition[DDLL4_BITS_FOR_FH_IDX] MUST be zero.
- The value of ddll_blockmap_partition[DDLL4_BITS_FOR_BLK_NUM_IDX] MUST be 63.
- The length of ddll_fhlist MUST be zero.
- The length of ddll_change_attr MUST be zero.
- The length of ddll_devlist MUST be zero.

The effect of the length of ddll_change_attr being of zero length is that server will recall the layout of a block before allowing that block to be modified. Except for the restriction that ddll_change_attr is of zero length, the effect of the above restrictions is to disable de-duplication when using the sub-file caching layout types. If client wants both sub-file caching and de-duplication awareness, it can request the LAYOUT4_DEDUP_ROC_TOP layout type.

Note that the client can safely cache a block of file only if block's corresponding element in the ddll_blockmap array has the

DDLL4_BLKMAP_MASK_ACTIVE bit set. The rest of the bits of the element of ddll_blockmap MUST be equal to the array index of the element.

5. Acknowledgements

Thanks to Pranoop Erasani, Arthur Lent, and Dave Noveck for validating the strategy described in this document.

6. Security Considerations

If an ACCESS operation by the principal on the source file would fail, then the server has take care when processing requests for de-duplication layouts of the target file. If the server is unable to perform access control at the granularity of the a byte-range, then the server MUST NOT allow the principal to read the source file. A related concern is that if the server can provide per-byte-range access, then the server will need to allow an OPEN operation of the source file by the principal. The server will need to reject READ operations for the non-de-duplicated data. The reader should adjust the algorithm in Figure 4 accordingly.

7. IANA Considerations

This specification requires 196 additions to the Layout Types registry described in Section 22.4 of [2]. Each added entry has five fields. The first entry is:

1. Name of layout type: LAYOUT4_DEDUP_TOP.

2. Value of layout type: TBD1. [[Comment.3: Note to IANA. Assign LAYOUT4_DEDUP_TOP a value that is a whole multiple of 64.]]

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The second through 64th additions to the Layout Types registry each have the following form, where <xx> is a decimal number between 02 and 64, inclusive:

1. Name of layout type: LAYOUT4_DEDUP_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 + LAYOUT4_DEDUP_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 65th entry is:

1. Name of layout type: LAYOUT4_DEDUP_ROC_TOP

2. Value of layout type: The value assigned to LAYOUT4_DEDUP_TOP logically ORed with LAYOUT4_DEDUP_RECALL_ON_CHANGE.

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 66th through 128th additions to the Layout Types registry each have the following form, where <xx> is a decimal number between 2 and 64, inclusive:

1. Name of layout type: LAYOUT4_DEDUP_ROC_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 + LAYOUT4_DEDUP_ROC_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 129th entry is:

1. Name of layout type: LAYOUT4_CACHE_TOP

2. Value of layout type: The value assigned to LAYOUT4_DEDUP_TOP + 2 * LAYOUT4_DEDUP_RECALL_ON_CHANGE.

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

The 130th through 192nd additions to the Layout Types registry each have the following form, where <xx> is a decimal number between 2 and 64, inclusive:

1. Name of layout type: LAYOUT4_CACHE_LEVEL_<xx>.

2. Value of layout type: The result of the expression: <xx> - 1 + LAYOUT4_CACHE_TOP.

3. Standards Track RFC that describes this layout: RFCTBD65, which is the RFC of this document.

4. How the RFC Introduces the specification: L.

5. Minor versions of NFSv4 that can use the layout type: 1.

8. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

[2] Shepler, S., Eisler, M., and D. Noveck, "NFS Version 4 Minor Version 1", RFC RFC5661, Jan 2010.

[3] Lentini, J., Eisler, M., and D. Kenchammana, "NFS Version 4 Minor Version 1", draft-lentini-nfsv4-server-side-copy-05.txt (work in progress), Jul 2010.

Author's Address

Mike Eisler
NetApp
5765 Chase Point Circle
Colorado Springs, CO 80919
US

Phone: +1-719-599-9026
Email: mike@eisler.com