April 4, 2014
PARC

CCNx 1.0 Streaming Storage

Marc Mosko[1]*

Abstract
In CCN, it may be desirable to cache many large content objects in fast memory. This paper describes a way to create a fast in-memory index and an efficient streaming disk store.

Keywords
Content Centric Networks – Router Design

[1] *Palo Alto Research Center*
***Corresponding author**: marc.mosko@parc.com

Contents

Introduction

In CCN, it may be desirable to cache many large content objects in fast memory. This paper describes a way to create a fast in-memory index and an efficient streaming disk store. A fast CCN router could process 10s of millions of content objects per second. For example, a 100 Gbps line card processing 64 Kbyte objects could process up nearly 200,000 objects per second. Therefore, a fast, efficient caching mechanism is needed.

In CCNx Streaming Storage (CCNxSS), which uses exact matching on a Similarity Hash and Forwarding Hash from CCNxHF, the SH/FH combination is hashed down to a smaller index in to physical RAM. Each index entry contains a disk number, a block number, and the number of blocks used for that SH/FH cache entry. The entry may also indicate if the line is valid or not cached, such as by using a special value for the disk. Figure 1 depicts a simplified system.

Hashing the SH/FH down to the index entry should be done with a collision-resistant hash function, so an attacker cannot easily cause the system to overwrite content. Reads are always verified against the actual SH/FH. One possible function could be to use the lower bits of CRC-64-ECMA-182, which is very fast to calculate and may be available in hardware. One could also use other schemes such as FVN-1a or other hardware supported hash functions.

For an Interest, if the index entry is valid, the system then reads a block from disk, such as a 4KB block, and verifies that the block entry matches the SH/FH. If it does, it returns the data, which is a well-formatted CCNxHF packet. If it does not match, the index entry should be invalidated and the interest forwarded as normal.

For a Content Object, if the index entry is invalid, the system looks up the current Tail pointer. The Tail pointer indicates the next block to use, which includes the disk number and 4KB block on that disk. The index entry is updated to the Tail, the Count is updated to the number of 4KB blocks used by the index entry, and the Tail is moved forward appropriately. The Content Object is then written to the indicated location, including a Disk Block header. When writing to multiple consecutive blocks for one Content Object, the "Num" field of the header indicates the sequence.

The Tail pointer is incremented modulo the disk size, so when the end of the disk is reached (in blocks), it wraps around to the first block. In a system with multiple disks, the wrap around could include the disk unit number.

If the system keeps popularity statistics for content objects, it may reserve a disk or part of a disk for popular content. When an object becomes popular, it is migrated to that area, which is not over-written by the circular Tail pointer. A more sophisticated replacement scheme is used in those areas.

The system may use different block sizes than 4KB. For CCNxHF objects which may be up to 64 KB, 4KB is a reasonable number. A single object may be written over 16 consecutive entries, requiring a 4-bit "cnt" field in the index.

The system uses in-memory index for speed. A variant may use a disk-based index that is page swapped to RAM.

The in-memory index needs to be large enough to cover all 4KB blocks in the system. For example, a system with up to 1TB of disk storage requires 256 million index entries. At 40 bits each, the index requires 10.24 GB of RAM. For a system with 16 TB, it requires 160 GB of RAM.

A typical disk might achieve up to 150 Mbyte/sec of sustained throughput. To store 12.5 Gbyte/sec of data, the data must be written over 100 disks in parallel. Using SSD disks, this number might be reduced to 50 disks in parallel. The Seagate 3.5" Video Streaming hard drives, for example, sustain 146 MB/s for reads and writes [ST4000VM000].

The Tail pointer would increment over disks first, then blocks on disks as the slower loop. 100 disk of 250 GB each has a total storage size of 25TB.

As the Tail pointer cycles around, the system must first

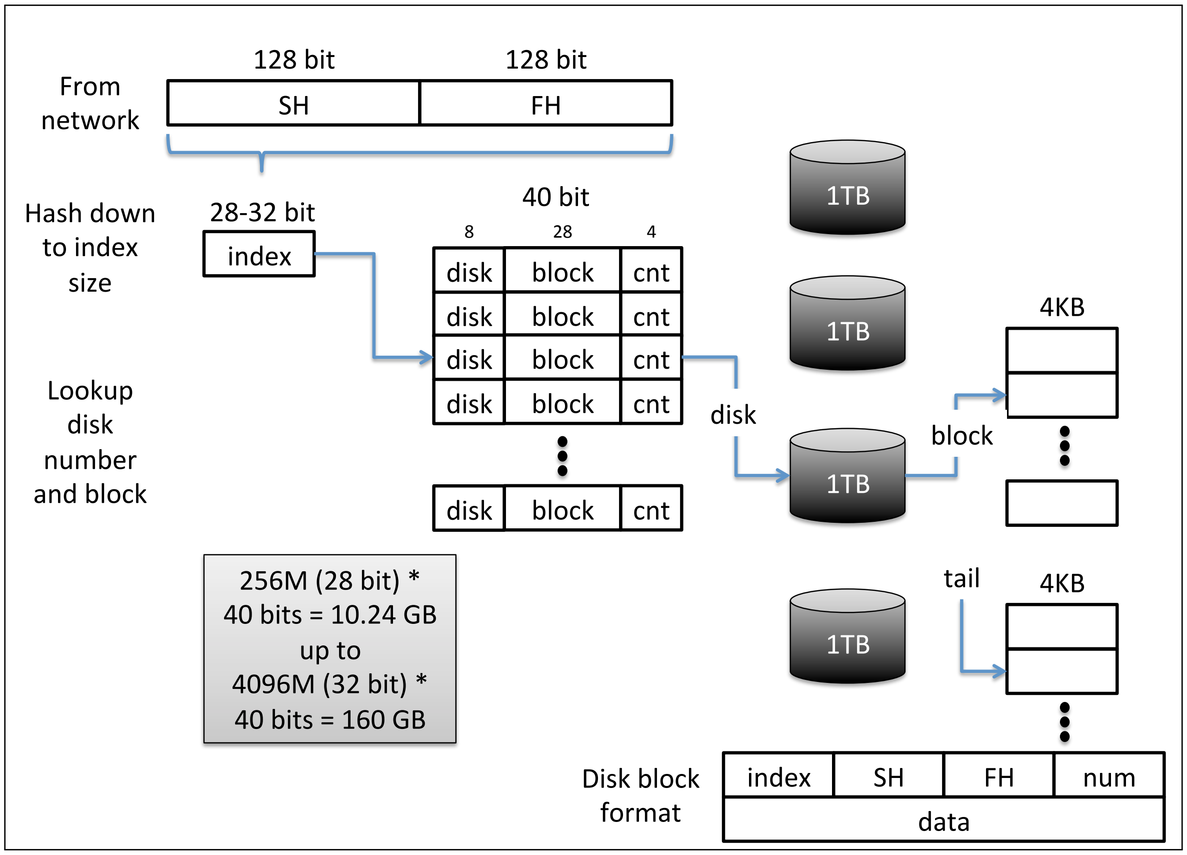

Figure 1. Simplified Architecture

read a block. If the Disk Block header indicates the 4K block is used by an index entry, and it is "num" 0, the system invalidates that index entry before writing the new index entry. It must do this for each block written for a contiguous Content Object. If the Tail pointer does not use a deterministic pattern, then the check must be done for all "num" not just 0.

1. Detailed Algorithm

Let there be N disk drives of 250 GB each. If physical hard drives are larger than 250 GB, partition them in to virtual disks as 250 GB, each assigned its own disk number. The disk numbers should be interleaved so consecutive numbers address different physical disks. Let the block size be 4KB, so there are 64 million index entries per disk, requiring 26 bits in the index. Let each index line be: 26-bit block, 4-bit count, 10-bit disk, a 15-bit "operation table index", and a 1-bit "read valid" flag (total 56 bits). This allows 64M blocks/disk, 511 disks (0x1FF means invalid entry), and up to 16 4K blocks per content object.

Let the physical memory for the index be P bytes. The index_size = floor(P * 8 / 56). For example, with 96GB of RAM, there can be 13.7 billion index entries, or 12 billion entries if the table is 64-bit word aligned. For the example system operating at 200,000 objects/second (at 64 KB/object), this is around 60,000 seconds (16 hours) of storage. We need 34-bits of index, for example extracted from a CRC-64-ECMA-180 digest.

Based on disk latency, there needs to be an operation buffer to hold Interests and Content Objects. This in-memory buffer could be, for example, a circular buffer or a free-list, holding pointers to packet buffer memory. The operation buffer needs to store a pointer to the packet plus an index number (e.g. 34-bits), plus a singly-linked list pointer to a previous entry if multiple operations are waiting on the same disk block. We assume that the operation buffer is contiguous memory and may be indexed by the 15-bit "operation table index".

Figure 2 shows the architecture of the system considered here. It uses a 34-bit index to the in-memory table, for 17,179,869,184 table entries. Each entry is, at minimum 56 bits, though an actual system may store them in 64-bit words. Using 64-bit words, the table would take 128 GB of RAM. In addition, there is the Operation Buffer memory, consisting of 68-bit entries. Each entry contains a 34-bit index to the in-memory table, a 15-bit pointer to the packet buffer (allowing 32,768 in-process packets), a 15-bit pointer to the next Operation Buffer waiting for the same DMA operation, and the 4-bit sequence number of the SH/FH blocks. Each 4K disk block starts with a fixed header, pointing to the in-memory table index, the stored SH/FH, and the sequence number of

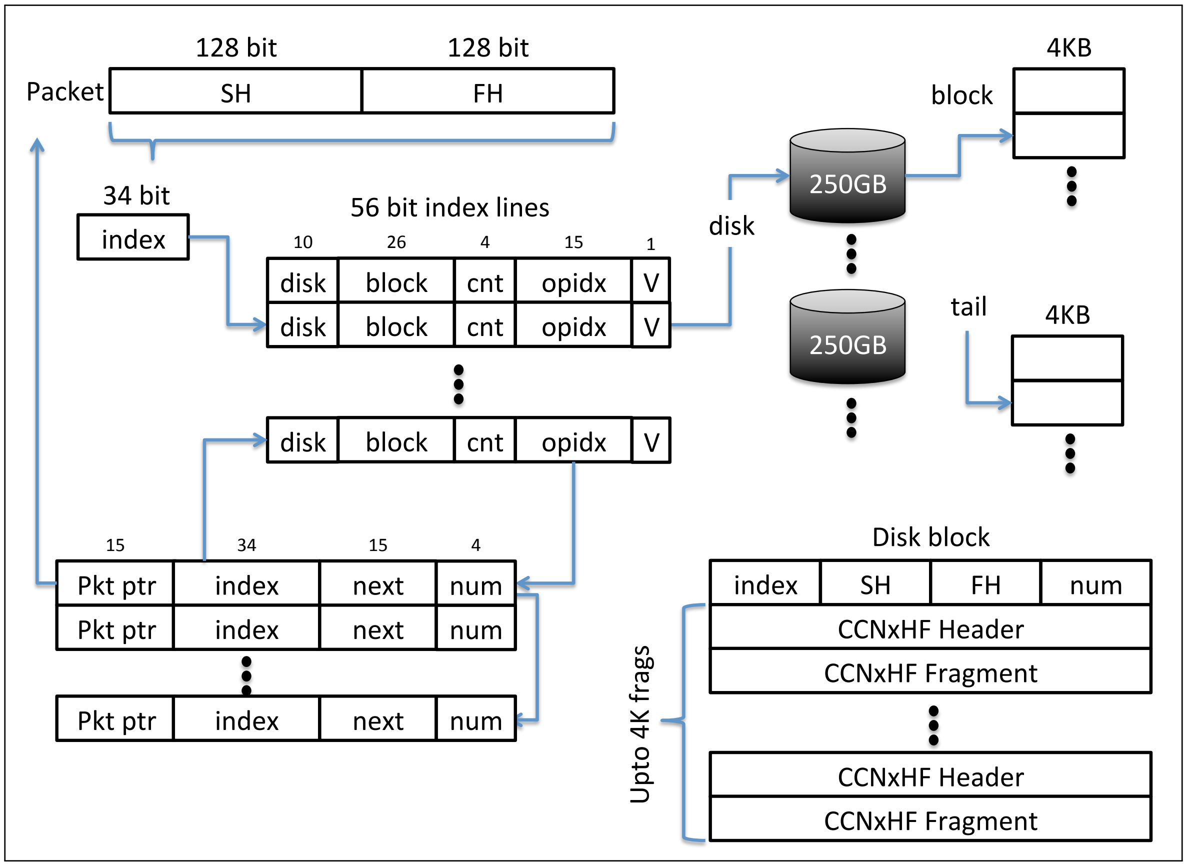

Figure 2. Detailed Architecture

the block for the SH/FH entry. Following the header are the CCNxHF fragments, written in the received order. The description below assumes that no fragment is larger than 4KB minus the disk block header size. A modest modification to the algorithm allows CCNxHF fragments need to span disk blocks.

To facilitate asynchronous DMA disk reads, the DMA buffer contains a pointer to the in-memory index (34-bits), which contains a pointer to the pending operation buffers. Because we use a 64-bit word aligned index entry, this scheme uses otherwise unused bits. In a more compact representation, such as 40-bits (9-bit disk, 26-bit block, 4-bit count, 1-bit valid), it may make more sense to have the DMA buffer point to the first pending operation, and the smaller operation table point to the 34-bit index table. This alternative scheme requires that additional operations waiting for the same block not be added to the head of the list.

First time Initialize

1. Allocate the in-memory index, setting all "disk" numbers to 0x1FF to indicate an invalid entry.

2. Set the "pending table index" to 0xFFFF to indicate invalid.

3. Set the Tail pointer to disk 0, block 0.

Initialize after clean shutdown

1. Read the in-memory index from disk cache.

2. Set Tail pointer from disk cache.

Initialize after dirty shutdown

1. Build in-memory index from "popular" disk cache, ignoring all other objects.

2. Set the Tail pointer to disk 0, block 0.

Receive Interest

1. Receive a pointer to the Interest in packet buffer memory.

2. Allocate an operation buffer and put the pointer in the operation buffer.

3. Hash the SH/FH to the index.

4. If the disk number is 0x1FF, it is a cache miss. Forward the interest as normal.

5. If the "V" flag is not set, we do not have all fragments of a Content Object, so forward the interest as normal.

6. If the operation table index is not 0x7FFF, there is already a pending operation. Discard the interest (when the previous operation completes, it will satisfy this interest) and free the operation buffer.

7. Schedule the block read.

   (a) For the first block of the object:

      i. Put the index in the operation buffer.
      ii. Put the operation buffer index in the "operation table index"
      iii. Point the "next" pointer to itself to indicate it is the last buffer in the chain.
      iv. Set the operation buffer num to "0".
      v. Setup the receive DMA block with a pointer to operation buffer.
      vi. Schedule the disk read

   (b) For the remaining "cnt-1" blocks of the object

      i. As above, but allocate a new operation buffer for each block and add it to the chain. Set the index and pkt ptr as in the first operation buffer. Set the "num" to the sequence being read.
      ii. Schedule a disk read.

Receive a Content Object

1. Receive a pointer to the Object in packet buffer memory. This may be a fragment of the object, not the entire object, but it will have a complete SH/FH.

2. Allocate an operation buffer and put the pointer in the operation buffer.

3. Hash the SH/FH to the index.

4. If the "V" flag is set, we already have all fragments, so discard the object.

5. If the disk number is 0x3FF, it is a cache miss

   (a) Determine the total Object length from the CCNx TLV headers. For CCNxHF, estimate a worst case from the total fragment count times the fragment MTU. Determine how many 4KB blocks, this requires and set the "cnt" field.
   (b) If there are not "cnt" consecutive blocks available where the Tail pointer is (its at the end of a physical disk), advance the pointer until there are consecutive blocks.
   (c) Setup a DMA read buffer, pointing to the index.
   (d) In the operation buffer, set "num" to 0.
   (e) Schedule a disk read of the first 4KB block.

6. Otherwise, the content object (or at least part of it) is in the cache, so schedule a disk read for the block.

   (a) If the opindx is 0x7FFF, fill in the opidx with this operation buffer, then schedule a disk read for block 0 of the series.
   (b) If the opindx is not 0x7FFF, there are other pending write operations (V is not set). Set the "num" to 0 in the operation buffer and tail insert to the list. This may result in multiple operation buffer lines with different packet pointers waiting for a read on the same block.

When a DMA completes

1. Lookup the index from the DMA block.

2. Iterate through all the operation buffer entries pointed to by the linked list.

3. If it is an Interest and "num" matches the block read:

   (a) Verify the index line is valid and "V" is set and the SH/FH match. If so, return the disk data as content object fragment(s). The fragments will be returned in the order written, so they may be out of order. Remove the operation buffer from the list.
   (b) Otherwise, it is a miss. Invalidate the index line and forward the interest as normal.
   (c) If performing popularity measures, increment the block's popularity and possibly migrate it to a "popular" disk block.

4. If it is a Content Object and "num" matches the block read (there may be several entries that match "num", do this for each):

   (a) If the SH/FH does not match the new index entry

      i. The content is being over-written. This should only happen when for block "0" of a new content object write process.
      ii. Look up the old index pointed to by the disk block.
      iii. If there are pending operations (there should only be read operations), then re-write the new index line to use the current Tail pointer (as above) and try again.
      iv. If there are no pending operations, invalidate the old index entry, then continue as if they matched.

   (b) Look through the CCNxHF headers.

      i. If the fragment pointed to by "pkt ptr" exists in the list, it is a duplicate. Drop the packet and remove the list entry.

        ii. If the fragment is not in the list, and there are more blocks, schedule a read for the next block and update the "num" field. If there are no more blocks, append the "pkt ptr" packet to one of the blocks with enough space for it. Update the "num" to that.

    (c) After matching all Content Object entries for the original "num", schedule disk writes for any updated blocks.

5. If it is a Content Object and "num" matches the block written

    (a) Remove the operation buffer line, free the packet.

    (b) If it is the last fragment of the content object, set the "V" flag.

2. Conclusion

CCNx Streaming Storage offers an in-memory index to large content stores. The system is designed to hide disk latency by spreading Content Objects over multiple disks, so it can scale to higher and higher network throughput. It also offers the ability to store some high-value content on a different disk set than a promiscuously written object cache. CCNx Streaming Storage is also aware of CCNx fragmentation and will only serve fragmented content objects once all fragments are in the cache.