ESE532: System-on-a-Chip Architecture

Day 16: October 23, 2019

Deduplication and Compression Project

Today

- Motivation
- Project
- Content-Defined Chunking
- Hashing / Deduplication
- LZW Compression

Message

- Can reduce data size by identifying and reducing redundancy
- Can
  - spend computation and data storage
  - to reduce communication traffic

Problem

- Always want more
  - Bandwidth
  - Storage space
- Carry data with me (phone, laptop)
- Backup laptop, phone data
  - Maybe over limited bandwidth links
- Never delete data
- Download movies, books, datasets
- Make most use of space, bw given

Opportunity

- Significant redundant content in our raw data streams (data storage)
- **More formally**:
  - Information content < raw data
- Reduce the data we need to send or store by identifying redundancies

Example

- Two identical files
  - Different parts of my file systems
- Don't store separate copies
  - Store one
  - And the other says "same as the first file"
    - e.g. keep a pointer

Why Identical?

- Eniac file system (common file server)
  - Multiple students have copies of assignment(s)
  - Snapshots (.snapshot)
    - Has copies of your directory an hour ago, days ago, weeks ago
      - …but most of that data hasn't changed

Broadening

- History file systems
  - snapshot, Apple Time Machine
- Version Control (git, svn)
- Manually keep copies
- Download different software release versions
  - With many common files

Cloud Data Storage

- E.g. Drop Box, Google Drive, Apple Cloud
- Saves data for large class of people
  - Want to only store one copy of each
- Synchronize with local copy on phone/laptop
  - Only want to send one copy on update
  - Only want to send changes
    - Data not already known on other side
    - (or, send that data compactly by just naming it)

Functional Placement

- At file server or USB drive
  - Deduplicate/compress data as stored
- In client
  - Dedup/compress to send to server
- In data center network
  - Dedup/compress data to send between server
- Network infrastructure
  - Dedup/compress from central to regional server

Optimizing the Bottleneck

- Saving data (transmitted, stored)
- By spending compute cycles
  - And storage database
- When communication (storage) is the bottleneck
  - We're willing to spend computation to better utilize the bottleneck resource

Project

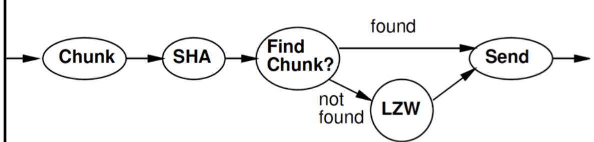
Project

- Perform deduplication/compression at network speeds (1Gb/s, 10Gb/s)
- Use “chunks” instead of files
- Turn a raw/uncompressed data stream into one that exploits
  - Duplicate chunks
  - Redundancies within chunks

Project Context

- File server input link from network
  - Compress data before sending to disk
  - (or USB link from computer, compress before store to flash)
- Network link in data center or infrastructure
  - Compress data that goes over network

Project Task



Motivation

- Can we afford to simply compare every incoming file with all the files we’ve already sent?

Preclass 1

- How many comparisons per input byte?

```
#define MAX_FILE_SIZE 4096
#define MAX_KNOWN_FILES (1024*1024)
#define -1
int  find_file(char file[MAX_FILE_SIZE],int flen, char **known_files) {
   for(int i=0;i<MAX_KNOWN_FILES;i++) {
     bool match=true;
     for (int j=0;j<flen;j++) match=(match && (file[j]==known_files[i][j]));
     if (match) return(i);
     }
   return(NO_MATCH);
}
```


Requirements?

- Can we afford to simply compare every incoming file with all the files we’ve already sent?
- Data coming in at 1 GB/s
- Processor (or datapath) running at 1GHz
- How many comparisons needed per cycle with preclass 1 solution?

Alternate Strategy

- Is there something we can compute on the input file that will let us
  - Know if a file is definitely not equivalent
    - So not worth checking every byte
  - Find the duplicate directly?

Alternatives

- How about
  - Look at size of file?
  - Look at 10 characters at fixed spots in the files?
- Could do better?
  - Could do something where changing any single character might be detected?

Exploring Alternatives

- What if we xor'ed together every byte in the file?
- What if we took sum of every word (group of 4 bytes) in the file?

Fingerprint, checksum, digest

- Compute a function on all the bytes in the file → **digest**
- Bins files into separate classes by the digest
  - Only need to check those
- As increase bits in digest
  - Make likelihood of two files having same digest smaller
- If can arrange for digests to essentially be unique – like a fingerprint

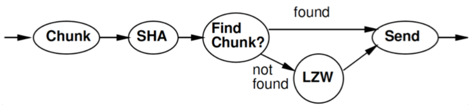
Hash

- A finite digest (fixed number of bits) computed on a potentially large collection of data (like a file)
- Ideally uniformly random digests
  - each hash value equally likely
- Use as building block for grouping and matching

Refined Strategy

- Keep a map of hash digests to files on the system
- On new file,
  - Compute hash digest on file
  - Only compare file contents against files with the same hash
- If hash is perfect with 20b, how does this reduce the number of files need to compare?

Content-Defined Chunking



Files or chunks?

- Why might files be the wrong granularity for identifying duplicates?

Blocks

- We regularly cut files into fixed-sized blocks
  - Disk sectors or blocks
  - inodes in File systems
- We could look for duplicates in blocks
- Why might fixed-sized blocks not be right division for deduplication?

Preclass 2 and 3

- How much duplication opportunity in
  - Preclass 2 blocks?
  - Preclass 3 chunks?
- Why chunks able to do better?

Common File Modifications

- Add a line of text
- Remove a line of text
- Fix a typo
- Rewrite a paragraph
- Trim or compose a video sequence

Content-Define Chunking

- Would like to re-align pieces around unchanged/common sequences
  - Around the content
- Break up larger thing (file) into pieces based on features of content
  - Hence ”content-defined”

Chunks

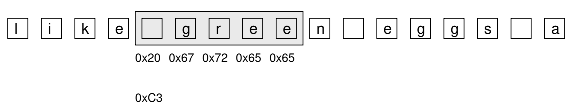
- Pieces of some larger file (data stream)
- Variable size
  - Over a limited range
- Discretion in how formed / divided

Chunk Creation

- How do we identify chunks?

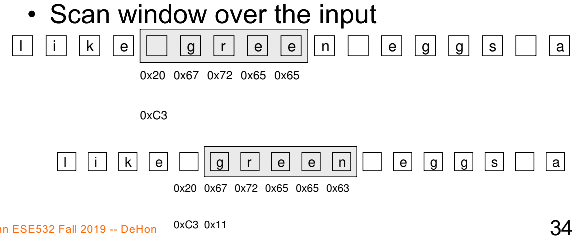
Hashes and Chunk Creation

- Compute a hash on a window of values
  - Window: sequence of N-bytes
  - Like window filter



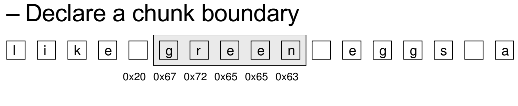
Hashes and Chunk Creation

- Compute a hash on a window of values
  - Window: sequence of N-bytes
  - Like window filter
- Scan window over the input



Hashes and Chunk Creation

- Compute a hash on a window of values
  - Window: sequence of N-bytes
  - Like window filter
- Scan window over the input
- When hash has some special value (like 0 or 0x11)
  - Declare a chunk boundary



Hashes as Chunk Cut Points

- What does this do?
- Guarantees that each chunk begins (or ends) at some fixed hash
- For a particular substring that matches the target hash
  - Always occurs at beginning (or end) of chunk
- If have a large body of repeated text
  - Will synchronize cuts at the same points based on the content

Chunk Size

- Assume hash is uniformly random
- The likelihood of each window having a particular value is the same
- So, if hash has a range of N, the probability of a particular window having the magic "cut" value is 1/N
- …making the average chunk size N
- So, we engineer chunk size by selecting the range of the hash we use
  - E.g. 12b hash for 2^{12} = 4KB chunks

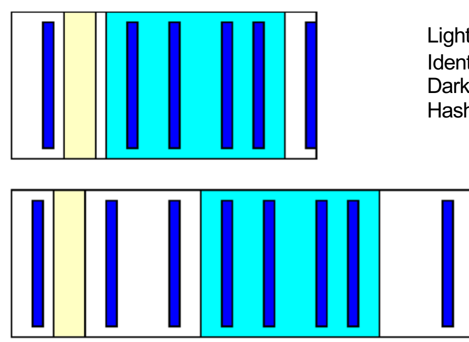
Chunking Design

- Raises questions
  - How big should chunks be?
    - Apply maximum and minimum size beyond content definition?
  - How big should hash window be?
- Discuss
  - What forces drive larger chunks, smaller?
    - How do large chunks help compression? Hurt?

Example Text

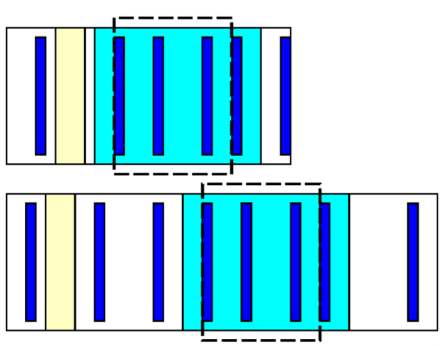
- Consider beginning of repeated block of text.
- This stuff has already been seen.
- But, we are only matching on something that has a hash of zero.
- Maybe this line has a hash of zero.
- But, our repeated text is before and after the magic window with the matched hash value.

Example Data Stream



Light blue Identical.
Dark blue Hash=0.

Example Data Stream



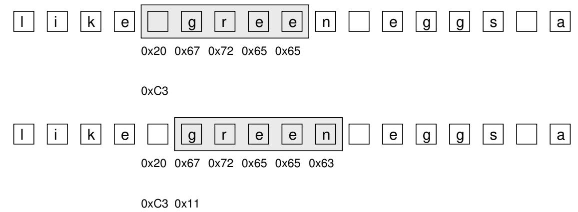
Chunk Size

- Large chunks
  - Increase potential compression
    - ChunkSize/ChunkAddressBits
  - Decrease
    - Probability of finding whole chunk
    - Fraction of repeated content included completely inside chunks

Rolling Hash

- A Windowed hash that can be computed incrementally
- Hash(a[x+0],a[x+1],…a[x+W-1])= G(Hash(a[x-1],a[x+0],…a[x+W-2])) - F(a[x-1])+F(A[x+W-1])
- i.e., hash computation is associative
- (+,- used abstractly here, could be in some other domain than modulo arithmetic)

Rolling Hash



- hash (gree) = 0x20+0x67+0x72+0x65+0x65
- hash (green) = 0x67+0x72+0x65+0x65+0x6e
- hash(green) = hash(gree)-0x20+0x6e

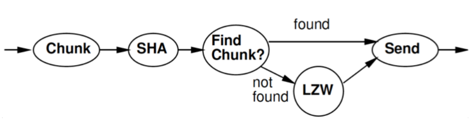
Rabin Fingerprinting

- Particular scheme for *rolling hash* due to Michael Rabin based on polynomial over a finite field
- Commonly used for this chunking application

Content-Defined Chunking

- Compute rolling hash (Rabin Fingerprint) on input stream
- At points where hash value goes to 0, create a new chunk

Hashing Deduplication



Hashes for Equality

- We can also (separately) take the hash signature of an entire chunk
- The longer we make the hash, the lower the likelihood two *different* chunks will have the same hash
- If hash is perfectly uniform,
  - N-bit hash, two chunks have a 2^{-N} chance of having the same hash.

Deduplicate

- Compute chunk hash
- Use chunk hash to lookup known chunks
  – Data already have on disk
  – Data already sent to destination, so destination will know
- If lookup yields a chunk with same hash
  – Check if actually equal (maybe)
- If chunks equal
  – Send (or save) pointer to existing chunk

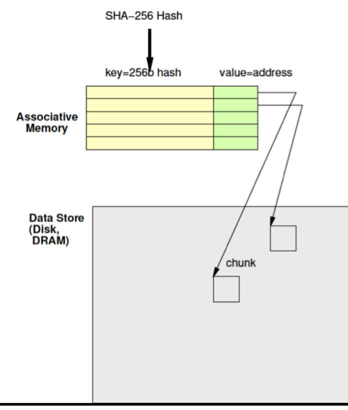
Engineering Hash

- How much DRAM on Ultra96?
- How many 1KB chunks on a 1TB file system?
- Potential hash values for 256b hash?

Deduplicate

- Compute chunk hash
- Use chunk hash to **lookup** known chunks
  – Data already have on disk
  – Data already sent to destination, so destination will know
- If lookup yields a chunk with same hash
  – Check if actually equal (maybe)
- How large of a memory do you need to hold the table of all 256b hash results?
- How relate to Ultra96 DRAM capacity?

Deduplication Architecture



Associative Memory

- Maps from a key to a value
- Key not necessarily dense
  – Contrast simple RAM
- Talk about options to implement next week

Secure Hash

- We regularly use digest signatures to identify if a file has been tampered with
- Again, hashes are same, mean data might be the same
- For security, we would like additional property
  – not easy to make the anti-tamper signature match

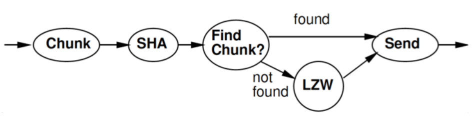
Cryptographic Hash

- One-way functions
- Easy to compute the hash
- Hard to invert
  - Ideally, only way to get back to input data is by brute force – try all possible inputs
- Key: someone cannot change the content (add a backdoor to code) and then change some further to get hash signature to match original

SHA-256

- Standard secure hash with a 256b hash digest signature
- Heavily analyzed
- Heavily used
  - TLS, SSL, PGP, Bitcoin, …

LZW Compression



Preclass 4, 5, 6

- Message?
- Bits in unencoded (decoded) message?
- Bits for encoded message?

Idea

- Use data already sent as the dictionary
  - Give short names to things in dictionary
  - Don't need to pre-arrange dictionary
  - Adapt to common phrases/idioms in a particular document

Encoding

- Greedy simplification
  - Encode by successively selecting the longest match between the head of the remaining string to send and the current window

Algorithm Concept

- While data to send
  - Find largest match in window of data sent
  - If length too small (length=1)
    - Send character
  - Else
    - Send <x,y> = <match-pos,length>
  - Add data encoded into sent window

Preclass 7

- How many comparisons per invocation?

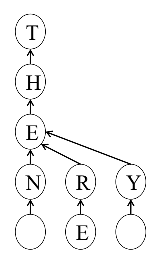
```
#define DICT_SIZE 4096
#define LENGTH 256
// clen<=LENGTH
int longest_match(char dict[DICT_SIZE], char candidate[LENGTH], int clen) {
  int best_len=0; best_loc=-1;
  for (int i=0;i<DICT_SIZE-clen;i++) {
    j=0;
    while((candidate[j]==dict[i+j]) & (j<clen)) j++;
    if (j>best_len) {best_len=j; best_loc=i;}
    }
  return((best_loc<<8)|best_len);
}
```


Idea

- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?
- If prefix same, why check redundantly?
  - Store things with common prefix together
  - Share prefix among substrings
  - Represent all strings as prefix tree
- Follow prefix trees with fixed work per input character

Tree Example

- THEN AND THERE, THEY STOOD…



Tree Algorithm

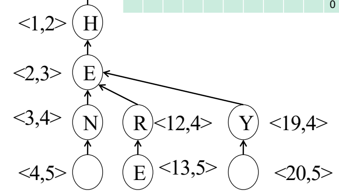
Tree Root for each character

- Follow tree according to input until no more match
- Send <name of last tree node>
  - An <x,y> pair
- Extend tree with new character
- Start over with this character

Tree Example

- Label with <lastpos,len> pair
- THEN AND THERE, THEY STOOD…

T	H	E	N		A	N	D		T	H	E	R	E	,		T	H	E	Y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19



(only showing Root for 'T')

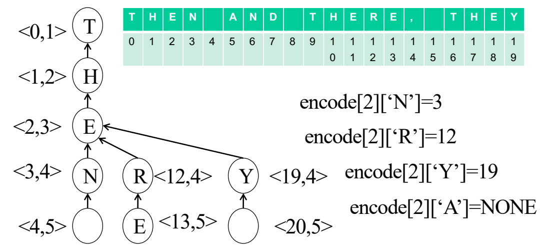
Large Memory Implementation

- int encode[SIZE][256];
- Name tree node by position in chunk
  - lastpos
- c is a character
- Encode[lastpos][c] holds the next tree node that extends tree node lastpos by c
  - Or NONE if there is no such tree node

Tree Example

(only showing Root for 'T')

- Label with <lastpos,len> pair
- THEN AND THERE, THEY STOOD…



Memory Tree Algorithm

```
curr – pointer into input chunk
// follow tree
y=0; x=0;
while(encode[x][input[curr+y]]!=NONE)
    x=encode[x][input[curr+y]]; y++;
If (y>0)
  send <x,y>
else
  send input[curr+y]
encode[x][input[curr+y]]=curr+y
curr=curr+y
```


Complexity

- How much work per character to encode?

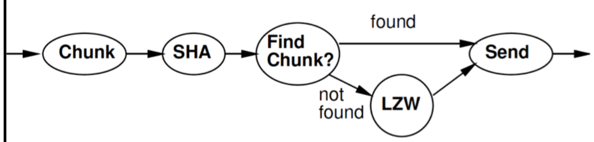
Compact Memory

- int encode[SIZE][256];
- How many entries in this table are not NONE?

Compact Memory

- int encode[SIZE][256];
- Table is very sparse
- Store as associative memory
  - At most SIZE entries
- Look at how to implement associative memories next time

Project Task



Big Ideas

- Can reduce data size by identifying and reducing redundancy
- Can spend computation and data storage to reduce communication traffic

Admin

- HW7 due Friday
- Project assignment out
- Reading for Monday online
- First project milestone due next Friday
  - Including teaming
  - Teams of 3