Data Compression in Solid State Storage

John Fryar
jfryar@ctp-llc.com
Acknowledgements

This presentation would not have been possible without the counsel, hard work and graciousness of the following individuals and/or organizations:

Raymond Savarda

Sandgate Technologies
The opinions expressed herein are those of the author and do not necessarily represent those of any other organization or individual unless specifically cited.

A thorough attempt to acknowledge all sources has been made. That said, we’re all human…
Learning Objectives

At the conclusion of this tutorial the audience will have been exposed to:

• The different types of Data Compression
• Common Data Compression Algorithms
• The Deflate/Inflate (GZIP/GUNZIP) algorithms in detail
• Implementation Options (Software/Hardware)
• Impacts of design parameters in Performance
• SSD benefits and challenges
• Resources for Further Study
Agenda

• Background, Definitions, & Context
• Data Compression Overview
• Data Compression Algorithm Survey
• Deflate/Inflate (GZIP/GUNZIP) in depth
• Software Implementations
• HW Implementations
• Tradeoffs & Advanced Topics
• SSD Benefits and Challenges
• Conclusions
# Definitions

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open System</td>
<td>A system which will compress data for use by other entities. I.E. the compressed data will exit the system</td>
<td>Must strictly adhere to standards on compress / decompress algorithms. Interoperability among vendors mandated for Open Systems.</td>
</tr>
<tr>
<td>Closed System</td>
<td>A system which utilizes compressed data internally but does not expose compressed data to the outside world</td>
<td>Can support a limited, optimized subset of standard. Also allows custom algorithms. No Interoperability req’d.</td>
</tr>
<tr>
<td>Symmetric System</td>
<td>Compress and Decompress throughputs are similar</td>
<td>Example – 40 Gb/s Ethernet Connection.</td>
</tr>
<tr>
<td>Asymmetric System</td>
<td>Compress and Decompress throughputs are dissimilar</td>
<td>Asymmetric can be workload balance or throughput differences</td>
</tr>
</tbody>
</table>
## Definitions

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured Data</td>
<td>Data which can be grouped into records of similar type and organized into a database (typically in Row &amp; Column format)</td>
<td>Can included metadata about unstructured data</td>
</tr>
<tr>
<td>Unstructured Data</td>
<td>Data which does not fit into the structured classification</td>
<td>You know it when you see it…</td>
</tr>
</tbody>
</table>
| Corpra          | Example Datasets used to verify and compare algorithms and their implementations | Examples: Calgary Corpus, Canterbury Corpus, etc.  
Note: Other datasets also exist that are used for this purpose (TPC-H and TPC-R for example) |
## Definitions

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>er</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>Within the context of LZ77 compression – a byte of data not part of a matched string</td>
<td>Substitutes 8 bits with 9 i.e. “a” becomes 0,a</td>
</tr>
<tr>
<td>Length, Distance</td>
<td>Within the context of LZ77 compression, the representation of a string 3 – 258 bytes long which matches a previous string in the history</td>
<td>Replaces the string with 24 bits of Offset and Distance back into the history i.e.: 1,L,D</td>
</tr>
<tr>
<td>Algorithm</td>
<td>A strictly defined procedure (well in a perfect world…) to implement a particular function.</td>
<td>Different methods of implementing algorithms possible for different use cases</td>
</tr>
<tr>
<td>Alphabet</td>
<td>The total set of possible members of a group</td>
<td>A-Z = alphabet of 26 “Literals = alphabet of 256</td>
</tr>
</tbody>
</table>
Compression Approaches

Compression is the elimination of redundancy in data in a reversible manner, increasing entropy and reducing the size of the data.

Compression can be lossless or lossy:

- **Lossy** - In some applications, it is acceptable to “lose” a small amount of input information during compression / decompression in exchange for higher compression ratios
  - Examples are Video or Audio, where eyes or ears will seldom detect some reasonable loss of detail

- **Lossless** – In many application, no data loss is acceptable, and in general this means lossless algorithms compress less than lossy algorithms.
  - Examples are financial data, medical records, user databases, etc.; where any data loss means data corruption and cannot be tolerated.
Lossless Compression Techniques

- **Run Length Encoding** – Replaces long strings of identical data with two symbols representing a symbol and length. Works well for long runs of repeating data:
  - i.e. 000000011111111 replaced with 0,7;1,8
- **Dictionary Coder** – Substitutes matched strings with references to a “dictionary”. The dictionary can be fixed or variable. A file can serve as its own dictionary.
  - Can Emulate Run Length Encoding (LZ77 for example)
- **Huffman Encoding** – Creating variable length symbols based on frequency of usage that replace a fixed length alphabet.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZ1 (LZ77)</td>
<td>Initial algorithms published in Lempel-Ziv’s classic 1977 &amp; 1978 papers:</td>
</tr>
<tr>
<td>LZ2 (LZ78)</td>
<td></td>
</tr>
<tr>
<td>LZS: Lempel-Ziv Stac</td>
<td>Sliding Window of fixed 2K Size + Static Huffman Encoder. Simple fast, popular</td>
</tr>
<tr>
<td>LZW: Lempel-Ziv Welch</td>
<td>Improvement on LZ78</td>
</tr>
<tr>
<td>LZMA: Lempel-Ziv Markhov Algorithm</td>
<td>Used in 7 Zip</td>
</tr>
</tbody>
</table>

Plus: LZO, LZRW, LZJB, LZWL, LZX, LZ4.....
Agenda

- Background, Definitions, & Context
- Data Compression Overview
- Data Compression Algorithm Survey
- Deflate/Inflate (GZIP/GUNZIP) in depth
- Software Implementations
- HW Implementations
- Tradeoffs & Advanced Topics
- SSD Benefits and Challenges
- Conclusions
Deflate Algorithm:

The Deflate Algorithm combines LZ77 & Huffman Encoding into a popular lossless data compression algorithm.

- **LZ77**: Duplicate String Search and Replacement
  - Up to 32KB History Window
  - Minimum 3 byte string, max 258 byte

- **Huffman**: Bit Decimation
  - Static Huffman: Default code table presumed by encoder and decoder
  - Dynamic Huffman: Optimized code table constructed and stored/transmitted within Deflate Block

- Deflate is the algorithm that is used in the popular GZIP and Zlib formats.
Deflate Processing

- LZ77 Processing (String Search & Replacement)
- Huffman Encoding (Bit Reduction)

Data Context, Characteristics, etc.
GZIP, Zlib, and Deflate are interrelated but separate standards and often used interchangeably in the vernacular. This is inaccurate, and can cause issues...

<table>
<thead>
<tr>
<th>Name</th>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deflate / Inflate</td>
<td>RFC-1951</td>
<td>The fundamental compression / decompression algorithm. Can be used without GZIP / Zlib.</td>
</tr>
<tr>
<td>GZIP</td>
<td>RFC-1950</td>
<td>File format which supports multiple algorithms – although only Deflate has been used to date: Wraps deflate output with header / trailer Optional flags allow detailed metadata to be inserted if so desired.</td>
</tr>
<tr>
<td>ZLIB</td>
<td>RFC-1952</td>
<td>Streaming format which supports multiple algorithms – although only Deflate has been specified and used to date: Wraps Deflate output with lightweight header/trailer</td>
</tr>
</tbody>
</table>
Deflate Characteristics

- Non Recursive - Cannot compress already compressed data for additional Compression Ratio: Data will typically expand after first pass.

- Tremendous flexibility in Compressing Data
  - Window Size
  - Output Block Size
  - Maximum Max Length
  - Implementation options for LZ77 Search Engine
    - Hash Based (HW and SW)
    - Systolic Array Based (HW Only)
  - Huffman Encoding
    - Static or Dynamic

Santa Clara, CA
August 2013
Deflate in Closed Systems

Tremendous Additional Flexibility in Closed Systems (Typical of Flash Use Cases) Note: No longer standards based Deflate – proceed with care!

- Custom Optimized Static Huffman Trees (Based on optimizations of known data characteristics)
- Metadata Additions for housekeeping / error detection/correction
  - CRC’s, Hashes, etc.
- Custom Fixed Dictionaries ()
  - Extreme Example: Calgary Corpus = 14 files:
  - Could be represented as a 4 bit dictionary (with 2 spare bits …)
RFC 1950 – Deflate Header Format

Input File / Stream

Compressed Data Stream

Deflate Block 1
Deflate Block 2
Deflate Block 3
... Deflate SubBlock n

Field Description Decoder

<table>
<thead>
<tr>
<th>BF</th>
<th>BType</th>
<th>Compressed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>Block Final</td>
<td>0 – More Blocks Follow</td>
</tr>
<tr>
<td>BF</td>
<td>Block Final</td>
<td>1 – Last Block of Data Stream</td>
</tr>
<tr>
<td>Btype</td>
<td>Block Type</td>
<td>00 – Raw/Stored Mode</td>
</tr>
<tr>
<td>Btype</td>
<td>Block Type</td>
<td>01 – Static Huffman Coding</td>
</tr>
<tr>
<td>Btype</td>
<td>Block Type</td>
<td>10 – Dynamic Huffman Coding</td>
</tr>
<tr>
<td>Btype</td>
<td>Block Type</td>
<td>11 – Reserved, Not Used</td>
</tr>
</tbody>
</table>

Size (2K – 4K typical)
RFC 1950 – Deflate Header Format

Compressed Data Stream

- Deflate Block 1
- Deflate Block 2
- Deflate Block 3
- Deflate Block n

Input File / Stream

<table>
<thead>
<tr>
<th>0</th>
<th>00</th>
<th>PAD [4:0]</th>
<th>LEN [15:0]</th>
<th>NLEN [15:0]</th>
<th>LEN bytes of Literal Data (64K Max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td>LZ77 Static Huffman Encoded Data</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Huffman Code Table</td>
<td></td>
<td></td>
<td>LZ77 Dynamic Huffman Encoded Data</td>
</tr>
</tbody>
</table>
## RFC 1950 – ZLIB Header Format

![Diagram of ZLIB Header Format]

### Field | Description | Decoder
--- | --- | ---
CM | Compression Method | x8 = Deflate; xff = Reserved; Others – Not Assigned
CINFO | Compression Information | For CM = 0x8:
0x8 – xFF: Not Allowed
0x7: - x0: 32K – 256B History Window
For CM Not = x8: Not Defined
FLG [7:6] = Compression Level
DICTID | Dictionary ID | Identification for optional pre-defined Dictionary. See RFC-1950 for Details
ADLER-32 | Checksum | ALDER-32 Checksum over uncompressed data (Before Deflate Processing)
## RFC 1952 – GZIP Header Format

### GZIP Header Format

- **ID1 & ID2**: ID Flags
  - ID1 = 0x1F; ID2 = 0x8B: GZIP; All others undefined
- **CM**: Compression Method
  - CM = 0x8: Deflate; CM = 0x7- 0x0: Reserved
- **FLG**: Flags
  - If FLG [n] = 1:
    - FLG [0] FTEXT (“Probably” Compressed ASCII Text)
    - FLG [1] FHCRC (Optional CRC-16 for Header Present)
    - FLG [2] FEXTRA (Optional Extra Fields Present)
    - FLG [3] FNAME (Filename Present)
    - FLG [4] FCOMMENT (Comments Present)
    - FLG [7:5]: Reserved
- **MTIME**: Modification Time
  - Creation / Modification time in Unix Format
- **XFL**: Extra Flags: For CM = 0x8
  - XFL =2; Max Compression; XFL = 4: Fastest Compression
- **OS**: Operating System
  - OS [8:0] (See RFC 1952 for list)
- **CRC**: CRC-32
  - CRC-32 of Uncompressed Data (Before Deflate Processing)
- **ISIZE**: Input Size
  - Size of original input data modulo $2^{32}$

---

**GZIP Trailer**

**SANTA CLARA, CA**

**August 2013**
GZIP Functional Block Diagram

Applies to HW or SW Implementation
LZ77 is a data compression method first described by Lempel & Ziv in 1977.

It uses a moving “window” of the last N bytes of the data that has been processed, and for the subsequent bytes it then searches for the longest match it can make in that earlier history.

The minimum match length is 3, so even if there is no current match a sequence of 3 new bytes is typically the minimum being searched for.
LZ77 Compression - operation

- If no match \( \geq 3 \) characters is found, the 1\(^{st}\) byte of the string is output as a literal byte, the window start and end are adjusted by one, and the next input byte is appended to the end of the 2 remaining bytes and a new search commences.

- If a 3 byte match IS found, new bytes are one-at-a-time added to the end of the match string, and searches are made to determine if the new longer string also has at least one match in the window.

- If a byte is added and there is no new match for the longer string, the previous matched string is emitted as a windows offset, length pair instead of the literal bytes.
In the Gzip/ZLIB variant of LZ77, an output literal byte occupies 9 bits, and a matched string up to 258 bytes occupies 24 bits.

Thus the worst-case result for just the LZ77 in this case is $1/8 = 12\%$ growth, vs. a best case of $24/258*8 = 98.8\%$ reduction in size.
### LZ77 Compression – Example

<table>
<thead>
<tr>
<th>Step</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0,c</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,0,4</td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0,b</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,5,3</td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>14</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>15</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>16</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>17</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td>1,0,5</td>
</tr>
<tr>
<td>19</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td>0,b</td>
</tr>
</tbody>
</table>

Santa Clara, CA  
August 2013
Lazy Match

- Usually when a match string is broken LZ77 will immediately output a L,D pair and begin a new string. This is a Greedy Match.
- Often a better Compression Ratio will occur if the search engine waits to see if a better match will occur later. This is a Lazy Match.
- Lazy Match waits can repeat for more than one byte.
# Greedy vs. Lazy Match

<table>
<thead>
<tr>
<th>Step</th>
<th>Input String Index</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>2</td>
<td>a b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>3</td>
<td>a b c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a b c d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, b</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a b c d b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a b c d b c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a b c d b c d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a b c d b c d e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1, 1, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0, a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Santa Clara, CA
August 2013

153 160
4.38%
## Greedy vs. Lazy Match Example

<table>
<thead>
<tr>
<th>Input String Index</th>
<th>10 11 12 13 14 15 16 17 18 19 20</th>
<th>#</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c</td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>a b c d</td>
<td></td>
<td></td>
<td>0,b</td>
</tr>
<tr>
<td>a b c d b</td>
<td></td>
<td></td>
<td>0,c</td>
</tr>
<tr>
<td>a b c d b c</td>
<td></td>
<td></td>
<td>0,d</td>
</tr>
<tr>
<td>a b c d b c d</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e</td>
<td></td>
<td></td>
<td>Lwait</td>
</tr>
<tr>
<td>a b c d b c d e b</td>
<td></td>
<td></td>
<td>1,1,3</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,e</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,b</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>1,2,7</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,a</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>0,d</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>138</td>
</tr>
<tr>
<td>a b c d b c d e b a</td>
<td></td>
<td></td>
<td>13.75%</td>
</tr>
</tbody>
</table>
Huffman encoding was invented by a MIT student in 1951, as part of a class assignment (David A Huffman).

- It uses uniquely-encoded binary bit strings to encode data.

- It encodes the most-frequently occurring characters to the shortest strings, and less-frequent characters in longer strings.

- It is only required to instantiate as many bit strings as you have characters, so if the characters to be encoded are “sparse”, the number of bits used will be minimal.

- The bit strings are constructed in such a manner that it is guaranteed when you decode the resulting string, it is always lossless and can be reconstructed from the tree.
Huffman Encoding (1)

Example:

Initial String: ACBCABABAABDB
Frequency Table: A(4); B(3); C(2); D(1)

Huffman Binary Tree

```
A
  B
    C
      D
```

Encoded String: 0 110 10 110 0 10 0 0 111 10

19 Bits w/ Huffman vs. 80 Bits ASCII
Deflate Huffman Encoding

- Complex and Efficient
- Three Alphabets
  - Literals (256 possible values of a byte)
  - Match Length (3-258)
  - Distance (1 – 32,768)
- Encodes 288 +32 (320 Total) Symbols
  - 288 Literals, Match Lengths, Overhead
  - 32 Distance Codes (Separate Tree)
- NOTE: RFC1951 is 15 pages.
  - Boilerplate, References, etc. Several Pages
  - Hashing – a couple of paragraphs
  - Huffman Encoding – over 7 pages…
Agenda

• Background, Definitions, & Context
• Data Compression Overview
• Data Compression Algorithm Survey
• Deflate/Inflate (GZIP/GUNZIP) in depth
  • Software Implementations
  • HW Implementations
• Tradeoffs & Advanced Topics
• SSD Benefits and Challenges
• Conclusions
Implementation Approaches:

Software: Dedicated Programs
• Intel Architecture Processors
• RISC Processors (ARM, PPC, MIPS)
• Others (GPUs, DSP, etc.)

Hardware:
• Dedicated Accelerators
• Task Specific Custom CPUs

Hybrid:
• ISA Enhancements: (e.g. HUFF $temp)
• Algorithm Partitioning (SW + Accelerators)
## Implantation Tradeoffs:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Gb/s capability of a particular implementation. Aggregate – total throughput</td>
<td>Single Stream throughput is most challenging: I.E. – 12 Gb/s single stream is MUCH harder than 12 1 Gb/s streams.</td>
</tr>
<tr>
<td></td>
<td>Thread – throughput of single Deflate Stream</td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>Delay from start to finish of compression / decompression</td>
<td>Larger History Windows, Dynamic Huffman increase latency</td>
</tr>
<tr>
<td>Compression Ratio</td>
<td>Reduction in size of original file. E.g. a 3:1 ratio means the final result</td>
<td>Larger History Windows, Dynamic Huffman increase compression ratio</td>
</tr>
<tr>
<td></td>
<td>is 1/3 the size of the starting block or file</td>
<td></td>
</tr>
<tr>
<td>Resource Utilization</td>
<td>Die Area, CPU Cores, Memory, Power are all resources utilized by H/W or SW</td>
<td>Faster = more power &amp; area Tighter CR = more power &amp; area</td>
</tr>
<tr>
<td></td>
<td>implementations</td>
<td></td>
</tr>
</tbody>
</table>
Hashing is the operation of applying an algorithm to a variable length string of data and generating a fixed length output:

- abc → # → 01101001
- abcdef → # → 10110010
- zbkk → # → 10110010

8 bit hash = 256 Indexes
Different Inputs Hash to same index
Hashing is a critical path function

- Must be Simple (Small Resource cost)
- Must be Fair” (equally distribute inputs into bins)
- Must be Fast (Throughput, Low Latency)

# of Indexes/Bins impacts CR & Performance

- More Bins = less matches per bin
- More Bins = more resources
SW GZIP Implementation

(Current Input String to Match)
abcdbabdb

H(abc)

H(abc) → head

H(bcd) → prev

H(abc) → prev

H(cdb) → prev

(prev → H(abc) → head)

(prev → H(bcd) → prev)

(prev → H(cdb) → prev)
SW GZIP Implementation

- Hashing and Hash Chaining (1)
  - In SW, LZ77 longest-string matches are implemented with a Hash Table (Head) and a chaining table (Prev).
  - Head is indexed by the hash of all active 3-byte sequences in the window, and is basically the 3-byte “head” of all possible window strings.
  - Prev keeps track of all “suffixes” to the 3 bytes indexed by “head”.
  - Head is $2^{\text{HashLength}}$ in size
  - Prev is $2^{\text{WindowSize}}$ in size
  - Hash of chars $c_i$, $c_{i+1}$, $c_{i+2}$ is:
    - $(((C_i \ll 5) \oplus c_{i+1}) \ll 5) + c_{i+2}) \& \text{WMASK}$
      where $\oplus = \text{XOR}, \ll = \text{SHL}$
SW GZIP Implementation

- Hashing and Hash Chaining (2)
  - The hash tables fill as multiple string matches in the window are found
  - Because of hash collisions, hash table “hits” have to be verified by byte-comparisons in the window, although there are some shortcuts used to simplify this.
  - The “Prev” chain can become quite long if there are multiple matches concurrently in the window (think 32KB of all 0’s).
  - Four parameters are specified to limit the time spent creating / searching / maintaining the hash tables to an acceptable level.
GZIP SW defines the following parameters which are specified by a command line input.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>good_length</td>
<td>Reduce lazy search above this match length</td>
</tr>
<tr>
<td>max_lazy</td>
<td>Do not perform lazy search above this match length</td>
</tr>
<tr>
<td>nice_length</td>
<td>Quit search above this match length</td>
</tr>
<tr>
<td>max_chain</td>
<td>Maximum length of hash chain to follow</td>
</tr>
</tbody>
</table>
## GZIP SW Compression “Levels”
- Default is Level 6

<table>
<thead>
<tr>
<th>Level</th>
<th>Good Length</th>
<th>Max Lazy</th>
<th>Nice Length</th>
<th>Max Chain</th>
<th>Mode</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Raw/Store</td>
<td>Apply Format</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>Static Huffman</td>
<td>No Lazy Matches</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>16</td>
<td>8</td>
<td>Dynamic Huffman</td>
<td>Lazy Matches</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>32</td>
<td>128</td>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>128</td>
<td>258</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>258</td>
<td>258</td>
<td>4096</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GZIP Functional Block Diagram

Input Data → LZ77 Search Engine → Huffman Encoder → Output Data

LLD Tokens

GZIP

Santa Clara, CA
August 2013
Two common approaches:

- **Hash Based**
  - Traditional Implementation – History stored in buffer(s)
  - Logic finds the longest match by stepping through a hash table
  - Limits on length of search within each history required
  - Multiple histories allow parallel searches

- **Systolic Array Based**
  - History stored in hardware registers
  - Hardware priority encoder locates longest match
  - Fast, deterministic operation (No Hashing / Hash Chains)
Hash Based Search Engine – Functional Block Diagram

- Input Data
- Look Ahead Buffer and Byte Alignment
- Hash FCN
- Hash Table
- History 0
- History n
- Near Distance Matcher
- Output Formation
- Output Data (LLD)
Two Huffman Encoder implementations for Deflate:

- **Static Huffman only**
  - Trivially small, simple, low latency
  - Encodes all blocks as “Static Huffman”, so compression ratio is compromised
  - This is what existing industry players usually offer

- **Dynamic Huffman (includes static encoding capability)**
  - Derives optimum Huffman code, creates “Dynamic Huffman” blocks.
  - Options allow selection of smallest size between dynamic Huffman, static Huffman, or stored mode. *No compromises – always produces highest possible compression ratio output for a given search engine.*
  - Numerous run time options to help user logic:
    - Concatenate output blocks
    - Raw buffer management
    - Output byte alignment (ZFLUSH) capability
    - Various tuning capabilities – see user guide

Santa Clara, CA August 2013
Dynamic Huffman Encoding Engine – Functional Block Diagram (simplified)
GUNZIP Functional Block Diagram
Agenda

• Background, Definitions, & Context
• Data Compression Overview
• Data Compression Algorithm Survey
• Deflate/Inflate (GZIP/GUNZIP) in depth
• Software Implementations
• HW Implementations

• Tradeoffs & Advanced Topics
• SSD Benefits and Challenges
• Conclusions
Verification Datasets

Corpora are datasets used to benchmark compression implementations

- Classic corpora
  - Canterbury, Calgary, Large, Artificial, Miscellaneous

- Other corpora
  - Protein, Lukas, Silesia (see [http://www.data-compression.info](http://www.data-compression.info))
  - The “Govdocs1 Million Files Corpus” (see [http://digitalcorpora.org](http://digitalcorpora.org))
    - Several CPU-months of simulation time, many hours of FPGA emulation time

- Locally generated
  - Special test cases which cannot be generated by GNU zip
  - Arbitrary files taken from various *nix, Windows systems, mobile phones

- WWW and other content
  - Video streams (e.g. youtube, news services), image and other near incompressible data
  - DVD files
### Software

- **Standard Data Center Processors** lack Symmetric Multi-Processing (SMP) Capabilities – Must pre-process somehow to take advantage of multi-core CPUs

- **Alternative Processor Architectures** (ARM, PPC, MIPS) can be more capable...

- PigZ is one alternative that attempts to utilize multiple processor cores to process a single stream of data

### Hardware

- Many more alternatives exist for Parallel Processing with HW Implementations of Deflate LZ77 and Huffman Encoders can be “Stacked” to process a stream in parallel
  - Slight decrease in CR – dataset dependent but typically between 1% to 3%
  - Linear scaling in Throughput
  - Linear scaling in consumed resources (area, power, etc).
## GZIP Instance Stacking

<table>
<thead>
<tr>
<th>Item</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objectives:</strong></td>
<td>Provide order of magnitude higher per stream throughput than achievable via traditional techniques applied to individual instances (clock rate enhancement, data bus widening, etc.) Maximize Utilization of Huffman Encoding Engine</td>
</tr>
<tr>
<td><strong>Throughput Enhancement</strong></td>
<td>Effectively n * x – i.e. 4 instances yield a ~ 4x throughput improvement</td>
</tr>
<tr>
<td><strong>Compression Ratio</strong></td>
<td>Negligible – Current benchmarking shows ~ 0.33% to ~ 1.3% C/R degradation (depending upon configuration options) between 1 instance and 2 stacked instances. No further C/R degradation observed for additional stacked GZIP instances (beyond 2nd).</td>
</tr>
<tr>
<td><strong>Resource (Gates, Memory)</strong> Impacts</td>
<td>Scales Linearly – very little overhead for splitter and combiner logic. Potential interconnect impact, depending on configuration options, number of instances used, etc.</td>
</tr>
<tr>
<td><strong>Standards Compatibility</strong></td>
<td>RFC Compliant – can decompress using standard GUNZIP IP or Software</td>
</tr>
</tbody>
</table>
Search Engine Instance Stacking - Typical Application

Search Engine Instance stacking maximizes throughput of Huffman Encoder. Can also stack GZIP Instances containing stacked Search Engines.

Santa Clara, CA
August 2013
GZIP Instance Stacking
- Typical Application

GZIP Instance stacking achieves order of magnitude throughput improvements!

Examples – a GZIP Block with:

- 8 stacked instances @ 4 Gb/s = 4 GB/s (32 Gb/s)
- 10 stacked instances @ 6.4 Gb/s = 8 GB/s (64 Gb/s)

An FPGA proof of concept which compresses at 124 Gb/s has been simulated…
Parallel Processing Challenges

- Decompression: Equivalent of parallel processing / stacking for Inflate/GUNZIP not possible for standards compatible files

- PigZ SW pre & post processing, but core inflate processing CPU bound.
  - Maybe OK for asymmetric systems

- HW Inflate engines optimized by design but ....

- Solution is addition of small amount of meta-data to deflate blocks – but problematic for open systems
State of Industry - Accelerators

• Under the hood in many flash subsystems
• Most current MIPS CPUs offer acceleration of deflate/inflate. ARM & PPC?
• A Recent Intel Server Chipset platform supports GZIP acceleration in HW.
• PCIe Cards (ComTech, Exar)
• Semiconductor IP also available to “roll your own”
• Real time compression “appliances” available
  • HW & SW approaches
Implementation Challenges in SSDs

- 8 Bit Algorithms vs. 32/64 Bit Processors
- Processor Architectures:
  - Lack of Symmetric Multi-Processing capability in some CPUs limits single thread performance (ARM, PPC MIPS excluded)
  - Errors in Data Compression Process (Soft errors)
- Lack of Deterministic Outcome on Compression - One byte change in a block can promulgate large change in compressed block size.
- Mismatch between standard formats and fixed sectors for SSDs / HDDs
- Uncompressible Data can Grow in size
- Mismatch in output between HW & SW implementations
## Solutions to Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft Errors</td>
<td>Error Corrected Memories</td>
</tr>
<tr>
<td>Lack of Determinism in block size</td>
<td>Force Deflate Block Generation on desired boundaries</td>
</tr>
<tr>
<td>Lack of Determinism in throughput / latency</td>
<td>Choose small windows, Static Huffman, or Systolic Array LZ77 implementation</td>
</tr>
<tr>
<td>Data Expansion (Attempting to compress encrypted or pre-compressed data)</td>
<td>Select best of available Deflate Block formats on a block by block basis</td>
</tr>
<tr>
<td>Lack of Equivalency between HW and SW Implementations</td>
<td>SW Model that exactly matches HW implementation</td>
</tr>
</tbody>
</table>
Conclusions

• There is no perfect Data Compression Solution – all are subject to tradeoffs

• To the extent possible – know your data and tune the Data Compression Solution accordingly

• H/W and SW solutions optimize performance for different parameters

• Using both HW and SW solutions in the same system requires careful thought & planning
www.sandgate.com

http://www.ics.uci.edu/~dan/pubs/DataCompression.html

http://mattmahoney.net/dc/dce.html

http://www.zlib.net/feldspar.html


http://www.gzip.org/algorithm.txt
Thank You!
Supplemental Materials
LZ77 Compression – Example

| Step | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | Output |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|      |
| 1    |   | a |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |   wait |
| 2    |   | a | a |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |   wait |
| 3    |   | a | a | c |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    | 0,a    |
| 4    |   | a | a | c | a |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    | 0,a    |
| 5    |   | a | a | c | a | a |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    | 0,c    |
| 6    |   | a | a | c | a | a | c |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    | wait   |
| 7    |   | a | a | c | a | a | c | a |   |   |    |    |    |    |    |    |    |    |    |    |    |    | wait   |
| 8    |   | a | a | c | a | a | c | a | b |   |    |    |    |    |    |    |    |    |    |    |    |    | wait   |
| 9    |   | a | a | c | a | a | c | a | b | c |    |    |    |    |    |    |    |    |    |    |    |    | 0,1,4, |
| 10   |   | a | a | c | a | a | c | a | b | c | a |    |    |    |    |    |    |    |    |    |    |    | wait   |
| 11   |   | a | a | c | a | a | c | a | b | c | a | b |    |    |    |    |    |    |    |    |    |    | 0,b    |
| 12   |   | a | a | c | a | a | c | a | b | c | a | b | a |    |    |    |    |    |    |    |    |    | wait   |
| 13   |   | a | a | c | a | a | c | a | b | c | a | b | a | a |    |    |    |    |    |    |    |    | 0,1,5,3|
| 14   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | a |    |    |    |    |    |    |    | wait   |
| 15   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | c |    |    |    |    |    |    |    | wait   |
| 16   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | a | c |    |    |    |    |    |    |    | wait   |
| 17   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | c | a | a |    |    |    |    |    |    |    | wait   |
| 18   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | c | a | a | b |    |    |    |    |    |    | 0,1,5, |
| 19   |   | a | a | c | a | a | c | a | b | c | a | b | a | a | c | a | a | b |    |    |    |    |    |    | 0,b    |

Input String Index

Santa Clara, CA
August 2013
LZ77 Compression – Example

- Steps 1-2: wait for 3 bytes to search for.
- Steps 3-5: Have 3 bytes, but no 3 byte match, so emit 1 char and try next input char.
- Step 6-7: Have 3B match, then 4
- Step 8: no 5B match on next char, so output 4B match and restart search with 1 new byte (#8)
- Steps 9-10: wait for 3 bytes to search for; no 3B match, so emit 'b' and continue with next byte
- Step 11: Have 3B match, wait to see if it gets bigger
- Step 12: Match ends, emit 3B, start w/1 new byte
- Step 13-14: Get 3 bytes, but no 3 byte match, so emit 1 char and try next input char.
- Step 15-17: Have 3,4,5B match
- Step 18: New byte breaks string match, emit 5B string & restart
- Step 19: End of input file, output remaining bytes as literal
- Results: input=18*8=144 bits, Out=6*9+3*24=126b, 12.5% smaller
Lazy Match

- When scanning new input data to continue a match string, normally at the first character that breaks the string match, the search terminates and the whole match string is output and a new search commences. This is called Greedy Match.

- What if there was a match starting @ N+m in the above case, which continues and includes the newly added last byte, and in fact would be a longer match than the one emitted?

- Lazy Match delays committing to output when a match breaks for one or more additional characters, looking to see if a longer match is possible. If a longer match is found, the longer match is used and the initial character of the smaller match is emitted as a literal.
Steps 1-14: roughly same for both cases, except “lazy wait” in step 8 that doesn't pan out.

Step 15, greedy match: Have 4 byte match, end of match, emit 4B match.

Step 15, lazy match: Don't commit to 4B match, add one more byte and see if we may have another 4B match – and we do.

Step 16, lazy match: Search using next byte, find we now have a 5B match, better than original 4B match, so emit 1\textsuperscript{st} char as literal and continue search.

Steps 17-18, lazy match: extend match to 6 then 7 bytes

Step 19, lazy match: match ends, but hold off committing to output until we see if still no match on next byte (lazy wait).

Step 20, lazy match: no new 7B match, output original 7B match

Step 21, lazy match: output 1B literal

Step 22, lazy match: End of file, output remaining chars as literals

Results: Greedy match=4.38\% smaller vs . Lazy Match=13.75\%
Huffman Encoding (1)

Example:

Huffman Encode the string: ACBCABAADB

Frequency Table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

Build the Binary Tree:

```
A
 /   \
|     |
B---0---C
 |   10
|    |
D---11
```

Huffman Encode the original String:

```
0 110 10 110 0 10 0 0 111 10
```

ASCII Size = 10*8 = 80 Bits
Huffman Size = 19 Bits

Note the receiver needs BOTH the Huffman String AND the encoding binary tree. This is “Dynamic” Huffman. It adapts to and minimally encodes the given dataset.

“Static Huffman” uses a pre-defined binary tree with “typically good” encodings; this eliminates the need to construct, send, and reconstruct the tree with the encoded data, as both sides can have a pre-provisioned static tree, but it is not an optimal encoding.
• Hashing and Hash Chaining – Implications
  • SW implementations require RAM to implement the HASH tables. This has a cost in die size for on-chip applications.
  • Since the hash tables vary in length and search depth based on the input data, the time to search varies greatly based on the input data. This means the search time is non-deterministic within limits set by the compression level.
  • Systolic-Array (HW) based LZ77 implementations can deliver a search result in a fixed-time per new character, regardless of the input data and window, and it can be much less than the worst-case hash-based approach.