Efficient Ways to Devise a High Speed Data Compression Algorithm

Abdullah-Al Mamun

CSE 5095 – Yufeng Wu – Spring 2013
Introduction

- Data compression is the art of reducing the number of bits needed to store or transmit data
- Compression is closely related to decompression
- Data compression is subject to space-time complexity trade-off
- Extensive usage in audio, video, genetics, text, graphics, cryptography
Papers

- **A simple algorithm for computing the Lempel-Ziv factorization**
  
  *Author:* M. Crochemore, L. Ilie, and W.F. Smyth
  
  *Conference:* Data Compression Conference (DCC), 2008

- **Time and Memory Efficient Lempel-Ziv Compression Using Suffix Arrays**
  
  *Author:* A. Ferreira, A. Oliveira, and M. Figueiredo
  
  *Conference:* Data Compression Conference (DCC), 2009

- **Gipfeli – High Speed Compression Algorithm**
  
  *Author:* R. Lenhardt and J. Alakuijala
  
  *Conference:* Data Compression Conference (DCC), 2012
A simple algorithm for computing the Lempel-Ziv factorization

- space-efficient simple algorithm for computing the Lempel–Ziv factorization of a string
- runs in $O(n)$ time independently of alphabet size
- uses $o(n)$ additional space for a string of length $n$
A simple algorithm for computing the Lempel-Ziv factorization

Introduction

- Lempel–Ziv factorization: previous occurrence of longest prefix or a single letter
- `abbaabbbbaaabab` has `a.b.b.a.abb.baa.ab.ab`
- In LZ77-based adaptive compression methods
- In algorithms to compute repetitions in strings
- Inefficient suffix trees or others suffix automata
- More space efficient suffix array

Introduction Papers Paper1 Paper2 Paper3 Conclusion
Suffix Arrays

- $\text{suf}_i = w[i...n-1]$, for $0 \leq i \leq n-1$
- SA: $\text{suf}_{SA[0]} < \text{suf}_{SA[1]} < \cdots < \text{suf}_{SA[n-1]}$
- Computed in $O(n)$ time
- Linear-time algorithms for LCP
- Longest Previous Factor (LPF)

$\text{LPF}[i] = \max(\{l \mid w[i...i + l - 1] \text{ is a factor of } w[j...j + l - 1]\} \cup \{0\})$
Suffix Arrays

A simple algorithm for computing the Lempel-Ziv factorization

### Suffix Arrays

<table>
<thead>
<tr>
<th>$i$</th>
<th>$w[i]$</th>
<th>$SA[i]$</th>
<th>$LCP[i]$</th>
<th>$suf_{SA[i]}$</th>
<th>$LPF[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>8</td>
<td>0</td>
<td>aaabab</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>9</td>
<td>2</td>
<td>aabab</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>3</td>
<td>3</td>
<td>aabbaaabab</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>12</td>
<td>1</td>
<td>ab</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>10</td>
<td>2</td>
<td>abab</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>0</td>
<td>2</td>
<td>abbaabbbaaabab</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>4</td>
<td>3</td>
<td>abbbbaaabab</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>13</td>
<td>0</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>7</td>
<td>1</td>
<td>baaaabab</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>2</td>
<td>3</td>
<td>baabbbbaaabab</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>11</td>
<td>2</td>
<td>bab</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>6</td>
<td>1</td>
<td>bbaaabab</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>1</td>
<td>4</td>
<td>bbaabbbbaaabab</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>b</td>
<td>5</td>
<td>2</td>
<td>bbbbaaabab</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 1.** The arrays $SA$, $LCP$, and $LPF$ for the string $w = abbaabbbbaaabab$. 

**Introduction Papers** Paper 1 Paper 2 Paper 3 Conclusion
Algorithm

- $SA[i - 1] < SA[i] > SA[i + 1]$
  \[ LPF[SA[i]] = \max(LCP[i], LCP[i + 1]) \]
- $SA[i - 1] < SA[i] < SA[i + 1], LCP[i] \geq LCP[i + 1]$
  \[ LPF[SA[i]] = LCP[i] \]
- $SA[i - 1] > SA[i] > SA[i + 1], LCP[i] \leq LCP[i + 1]$
  Never occurs
A simple algorithm for computing the Lempel-Ziv factorization

Algorithm

Figure 3. (i) Solid labeled edges form the graph representing SA and LCP for the text abbaabbbbaaabab. (ii) The graph right before considering the vertex labeled 6.
Algorithm

\textbf{Compute-LPF}(SA, LCP)

1. \text{SA}[n] \leftarrow -1; \text{LCP}[n] \leftarrow 0
2. \text{PUSH}(0, \mathcal{I})
3. \text{for } i \text{ from } 1 \text{ to } n \text{ do}
4. \quad \text{while } (\mathcal{I} \neq \emptyset) \text{ and }
5. \quad \left( (\text{SA}[i] < \text{SA}[\text{TOP}(\mathcal{I})]) \text{ or }
6. \quad \left( (\text{SA}[i] > \text{SA}[\text{TOP}(\mathcal{I})]) \text{ and } (\text{LCP}[i] \leq \text{LCP}[\text{TOP}(\mathcal{I})]) \right) \right) \text{ do}
7. \quad \text{if } (\text{SA}[i] < \text{SA}[\text{TOP}(\mathcal{I})]) \text{ then}
8. \quad \text{LPF}[\text{SA}[\text{TOP}(\mathcal{I})]] \leftarrow \max(\text{LCP}[\text{TOP}(\mathcal{I})], \text{LCP}[i])
9. \quad \text{LCP}[i] \leftarrow \min(\text{LCP}[\text{TOP}(\mathcal{I})], \text{LCP}[i])
10. \quad \text{else}
11. \quad \text{LPF}[\text{SA}[\text{TOP}(\mathcal{I})]] \leftarrow \text{LCP}[\text{TOP}(\mathcal{I})]
12. \quad \text{POP}(\mathcal{I})
13. \quad \text{if } (i < n) \text{ then } \text{PUSH}(i, \mathcal{I})
14. \text{return LPF}

\textbf{Figure 4. Algorithm for computing LPF.}
A simple algorithm for computing the Lempel-Ziv factorization

Algorithm

\textbf{LEMPLE-ZIV-FACTORIZATION(LPF)}

1. \( \text{LZ}[0] \leftarrow 0; \ i \leftarrow 0 \)
2. \textbf{while} (\text{LZ}[i] < n) \textbf{do}
3. \( \text{LZ}[i + 1] \leftarrow \text{LZ}[i] + \max(1, \text{LPF}[\text{LZ}[i]]) \)
4. \( i \leftarrow i + 1 \)
5. \( \text{return LZ} \)

Figure 2. Algorithm for computing Lempel–Ziv factorization using LPF.

\( w = \text{abbaabbbbaaabab} \)

\( \text{LZF} = [0, 1, 2, 3, 4, 7, 10, 12] \)
A simple algorithm for computing the Lempel-Ziv factorization

Complexity

- $w = abab^2ab^3 \ldots ab^l$, stack needs 0, 2, 5, 9, \ldots, $l(l+1) - 1$; requires $\Theta(\sqrt{n})$ space
- thus the maximum size of the stack is $o(n)$
- Lempel–Ziv factorization, using SA and LCP, computed in $O(n)$ time independently of alphabet size and $o(n)$ additional space
Progression

Linear time computation of Lempel–Ziv factorization (paper 1) which is the basis for efficient LZ77 and LZSS encoding
Basis of lossless compression techniques

Hash tables, binary search trees and suffix trees used for this purpose for fast search at the expense of memory

Faster SA-based algorithms for LZ77 encoding and sub-string search, keeping their low memory requirements
Introduction

- Used in universal source coders
- Asymmetric
- Linear time SA construction algorithm SA-IS
- LZF based on SA and auxiliary arrays
- Amount of memory constant
Compression Format

- **Sliding window:** dictionary & LAB
- **LZ77 token** \((\text{pos}, \text{len}, \text{sym})\)
- **LZSS token** \((\text{bit}, \text{code})\)
- **Bit** = \{0, 1\}  **code** = \{(\text{sym}), (\text{pos}, \text{len})\}
Algorithm

- header has 48 bits: 8 np, 8 nl, 32 file size
- |LAB| ASCII symbols
- sequence of LZSS tokens

Figure 2: The LI (LeftIndex) auxiliary array: for each symbol that starts a suffix it holds the index of the SA \( P \) in which that suffix starts. For the symbols that are not the start of any suffix, the corresponding entry is marked with -1, meaning that we have an empty match for sub-strings that start with that symbol.
The LZSS encoding algorithm constructs and updates suffix arrays iteratively.

**Algorithm 1** LZSS Encoding using Suffix Array

| Input: | \( In \), input stream to encode; \( m \), length of dictionary; \( n \), length of LAB. |
| Output: | \( Out \), output stream with LZSS description of \( In \). |

1. Write 48-bit header: \( np, nl \) and \( FileSize \) (as described above).
2. Read the first look-ahead-buffer \( LAB \), with \( |LAB| \) symbols, from \( In \).
3. Write LAB into \( Out \).
4. Initialize every position of LI to -1.
5. Do coded \( \leftarrow 0 \).
Algorithm

6: while coded < FileSize do
7:     Slide in LAB into dictionary D and read next LAB.
8:     if coded < m then
9:         Build SA, using SA-IS algorithm [12], for D and name it P.  /* Dictionary is filling. */
10:    else
11:        Update P (as in algorithm UD and Fig. 3).  /* Runs after each LAB encoding. */
12:    end if
13:    Scan P and update LI (as described in Fig. [2]).
14:    Do i ← 0.
15:    while i < n do
16:        left = LI[LAB[i]].  /* Loop to encode n symbols in the LAB. */
17:        if (left == -1) then
18:            output (0, LAB[i]); i ← i + 1; continue.  /* Empty Match. No suffix starts with LAB[i]. */
19:        end if
20:        Find right, such that D[P[right]] = LAB[i].  /* Get left and right as in Fig. [1] */
21:        From the set of suffixes between P[left] and P[right], choose the suffix at index pos, such that
22:            left ≤ pos ≤ right.  /* Choose between “fast” and “best” compression. */
23:        Do len ← the match-length of sub-strings starting at D[P[pos]] and LAB[i].
24:        Output (1(pos, len)) into Out; i ← i + len.
25:    end while
26:    coded = coded + n.
27: end while
Figure 1: LZ77 and LZSS encoding with SA, with dictionary $D = mississippi$. In part a), with $LAB = issia$, we have four possible matches delimited by left and right. In part b) with $LAB = bsia$ there is no suffix that starts with ‘b’ (which is encoded as a single symbol), but after ‘b’ we find four suffixes whose first symbol is ‘s’; two of these suffixes start with ‘si’.
Algorithm

Algorithm 2 UD - Update Dictionary (line 11 of Algorithm 1)

Input: $P_A, P_B$, $m$-length SA; $P$, pointer to PA or PB; $P_{dst}$, pointer to PB or PA;
$LAB$, look-ahead buffer; $LI$, 256 position length LeftIndex array.

Output: $P_A$ or $P_B$ updated; $P$ pointing to the recently updated SA.

1: if $P$ points to $P_A$ then
2: Set $P_{dst}$ to $P_B$.
3: else
4: Set $P_{dst}$ to $P_A$.
5: end if
6: Compute the SA $P_{LAB}$ for the encoded LAB. {/* Sorts the suffixes in the LAB. */}
7: Using $LI$ and $P_{LAB}$, fill the $|LAB|$-length array $I$ with the insertion indexes (slide in suffixes).
8: for $j = 0$ to $|LAB| - 1$ do
9: $P_{dst}[I[j]] = P_{LAB}[j] + |dictionary| - |LAB|$. {/* The $|LAB|$ Insert Suffixes. */}
10: end for
Algorithm

11: Do updateCounter = |dictionary| − |LAB|.
12: for $j = 0$ to $|dictionary| − 1$ do
13:    if $(P[j] − |LAB|) > 0$ then
15:    updateCounter = updateCounter - 1;
16:    if (updateCounter==0) then
17:        break;  /* Break immediately if |dictionary| − |LAB| updates have been done. */
18:    end if
19: end if
20: end for
21: Set $P$ to $P_{dst}$.
    /* $P$ points to recently updated SA. */
Figure 3: Update step with pointer $P$ set to $P_A$ initially; the first update is done using $P_B$ as destination. Array $I$ holds the indexes where to insert the new suffixes; ‘U’ and ‘R’ are the Update and Remove indexes, respectively. On the right hand side, we have the initial and updated dictionary contents.
Comparisons

Table 2: Amount of memory, total encoding time (in seconds) and average compression ratio (in bpb), for several lengths of (|dictionary|, |LAB|) on the Silesia Corpus, using “best” compression. GZip “fast” obtains Time=19.5 and bpb=3.32 while GZip “best” does Time=74.4 and bpb=2.98. The best encoding time is underlined.

<table>
<thead>
<tr>
<th>#</th>
<th>Dictionary</th>
<th></th>
<th>LAB</th>
<th></th>
<th>Memory</th>
<th>Time</th>
<th>bpb</th>
<th>Memory</th>
<th>Time</th>
<th>bpb</th>
<th>Memory</th>
<th>Time</th>
<th>bpb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2048</td>
<td>1024</td>
<td></td>
<td>24576</td>
<td>118.7</td>
<td>5.66</td>
<td>26636</td>
<td>249.5</td>
<td>5.65</td>
<td>4217856</td>
<td>333.53</td>
<td>3.05</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4096</td>
<td>1024</td>
<td></td>
<td>43008</td>
<td>116.9</td>
<td>5.41</td>
<td>53260</td>
<td>303.4</td>
<td>5.25</td>
<td>4241408</td>
<td>349.05</td>
<td>2.90</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4096</td>
<td>2048</td>
<td></td>
<td>48128</td>
<td>112.9</td>
<td>5.68</td>
<td>53260</td>
<td>694.9</td>
<td>5.63</td>
<td>4241408</td>
<td>349.05</td>
<td>2.90</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8192</td>
<td>2048</td>
<td></td>
<td>84992</td>
<td>143.4</td>
<td>5.44</td>
<td>106508</td>
<td>668.9</td>
<td>5.27</td>
<td>4288512</td>
<td>356.77</td>
<td>2.76</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16384</td>
<td>256</td>
<td></td>
<td>149760</td>
<td>319.1</td>
<td>4.55</td>
<td>213004</td>
<td>254.6</td>
<td>4.44</td>
<td>4382720</td>
<td>366.47</td>
<td>2.62</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32768</td>
<td>256</td>
<td></td>
<td>297216</td>
<td>542.7</td>
<td>4.41</td>
<td>425996</td>
<td>318.1</td>
<td>4.31</td>
<td>4571136</td>
<td>356.34</td>
<td>2.52</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>32768</td>
<td>1024</td>
<td></td>
<td>301056</td>
<td>322.2</td>
<td>4.80</td>
<td>425996</td>
<td>382.6</td>
<td>4.64</td>
<td>4571136</td>
<td>356.34</td>
<td>2.52</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>32768</td>
<td>2048</td>
<td></td>
<td>306176</td>
<td>302.3</td>
<td>5.02</td>
<td>425996</td>
<td>979.8</td>
<td>4.81</td>
<td>4571136</td>
<td>356.34</td>
<td>2.52</td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Based on suffix arrays
- Uses an auxiliary array as an accelerator
- Priori computing exact amount of memory
- 256-position auxiliary array to fast indexing mechanism
Progression

- Linear time computation of Lempel–Ziv factorization *(paper 1)* which is the basis for efficient LZ77 and LZSS encoding
- Time and memory efficient Lempel-Ziv compression using suffix arrays *(paper 2)* which can be further improved

??
Gipfeli –
High Speed Compression Algorithm

- High-speed compression algorithm
- Backward references with a 16-bit sliding window
- Ad-hoc entropy coding for both literals and backward references
- Compression ratio is similar to Zlib
- Bandwidth-bound systems, intermediate data storage and parallel computations
Memory lagged behind CPU
Decrease running-time and memory usage
Snappy, QuickLZ, FastLZ
Very light-weight hashing and sliding window
Two main parts
LZ77
Static entropy code for backward references and ad-hoc entropy code for literals
Compression Format

Fig. 1. Diagram of the algorithm
compressed string has the form ABBB...B
A contains the uncompressed string
block B is result of compressing an input block of length at most $2^{16}$
block B consists of two parts
Commands and Content
emit literals and emit copy commands
## Compression Format

**Encoding Emit Literal Command**

<table>
<thead>
<tr>
<th>length</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.32%</td>
</tr>
<tr>
<td>2</td>
<td>20.67%</td>
</tr>
<tr>
<td>3</td>
<td>11.59%</td>
</tr>
<tr>
<td>4</td>
<td>7.63%</td>
</tr>
<tr>
<td>5</td>
<td>5.01%</td>
</tr>
<tr>
<td>6</td>
<td>3.16%</td>
</tr>
<tr>
<td>7</td>
<td>1.88%</td>
</tr>
<tr>
<td>8</td>
<td>1.38%</td>
</tr>
<tr>
<td>9</td>
<td>1.00%</td>
</tr>
<tr>
<td>≥10</td>
<td>3.40%</td>
</tr>
</tbody>
</table>

**Table 2.** *Length* in Emit literal command for text / html input.
Blocks of size $2^{16}$

Small lengths occur much more often

If length < 53, prefix 00 followed by 6 bits

Otherwise 53 to 63 for bit length (6 to 16) followed by value
Prefix: length and distance

entropy code on the gathered statistics

<table>
<thead>
<tr>
<th>length</th>
<th>distance</th>
<th>prefix</th>
<th>length</th>
<th>bits</th>
<th>distance</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 - 7</td>
<td>1 - 1024</td>
<td>010</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4 - 7</td>
<td>1025 - 8192</td>
<td>011</td>
<td>2</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>4 - 7</td>
<td>8193 - 65536</td>
<td>100</td>
<td>2</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8 - 15</td>
<td>1 - 1024</td>
<td>101</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8 - 15</td>
<td>1025 - 65536</td>
<td>110</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16 - 67</td>
<td>1 - 65536</td>
<td>111</td>
<td>6</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Static entropy code used to encode backward references.
Compression Format
Encoding Emit Copy Command

<table>
<thead>
<tr>
<th>length</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 - 7</td>
<td>81%</td>
</tr>
<tr>
<td>8 - 15</td>
<td>15%</td>
</tr>
<tr>
<td>16 - 67</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 4. Distribution of length of backward references

<table>
<thead>
<tr>
<th>bits representing distance</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2%</td>
</tr>
<tr>
<td>2</td>
<td>0.1%</td>
</tr>
<tr>
<td>3</td>
<td>0.2%</td>
</tr>
<tr>
<td>4</td>
<td>0.7%</td>
</tr>
<tr>
<td>5</td>
<td>1.5%</td>
</tr>
<tr>
<td>6</td>
<td>3.0%</td>
</tr>
<tr>
<td>7</td>
<td>4.3%</td>
</tr>
<tr>
<td>8</td>
<td>5.8%</td>
</tr>
<tr>
<td>9</td>
<td>7.4%</td>
</tr>
<tr>
<td>10</td>
<td>8.9%</td>
</tr>
<tr>
<td>11</td>
<td>10.5%</td>
</tr>
<tr>
<td>12</td>
<td>11.8%</td>
</tr>
<tr>
<td>13</td>
<td>13.3%</td>
</tr>
<tr>
<td>14</td>
<td>14.8%</td>
</tr>
<tr>
<td>15</td>
<td>10.8%</td>
</tr>
<tr>
<td>16</td>
<td>6.6%</td>
</tr>
</tbody>
</table>

Table 5. Distribution of distance of backward references
Compression Format
Entropy Code for Content

- 32 symbols, 6 bits (prefix 0 + index)
- Next 64 symbols, 8 bits (prefix 10 + index)
- Last 160 symbols, 10 bits (prefix 11 + value)
- index is alphabetical order in group
- value is 8-bit char value
- Sampling input to build the entropy code
- Use five samplers
- Estimate the frequency of each symbol

Gipfeli – High Speed Compression Algorithm
Key Implementation Features

- Limited memory usage, optimal parallel performance
  - Fits comfortably in the Level-1 and Level-2 caches
  - Fully parallel

- References to the previous blocks and no need to reset the hash table
  - Backward references to the previous block
Very light-weight hashing in the LZ77 part
- allow $2^{15}$ hash table entries
- read four consecutive symbols
- 16-bit unsigned integer index
Key Implementation Features

- **High performance for incompressible input**
- increasing the size of steps
- **Writing and reading bits**
  - unsigned 64-bit integer as a buffer
  - static tables of constants for bit value of backward reference
Key Implementation Features

- Fast construction of entropy code for content
  - over 20% savings for the content
  - main savings (over 75%) for command
- Unaligned stores are much better than memcpy for short lengths
  - unaligned stores of unsigned 32- or 64-bit integers at the pointer position
## Comparisons

<table>
<thead>
<tr>
<th>File description</th>
<th>Compression ratio</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Snappy</td>
<td>QuickLZ</td>
</tr>
<tr>
<td>text 1</td>
<td>59.8%</td>
<td>54.9%</td>
</tr>
<tr>
<td>text 2</td>
<td>56.2%</td>
<td>49.3%</td>
</tr>
<tr>
<td>text 3</td>
<td>57.1%</td>
<td>51.9%</td>
</tr>
<tr>
<td>html</td>
<td>23.6%</td>
<td>19.4%</td>
</tr>
<tr>
<td>url addresses</td>
<td>50.9%</td>
<td>43.4%</td>
</tr>
<tr>
<td>protocol buffer</td>
<td>23.2%</td>
<td>15.8%</td>
</tr>
<tr>
<td>jpeg</td>
<td>99.9%</td>
<td>100%</td>
</tr>
<tr>
<td>pdf</td>
<td>82.1%</td>
<td>100%</td>
</tr>
<tr>
<td>C source code</td>
<td>42.4%</td>
<td>42.3%</td>
</tr>
<tr>
<td>LSP source code</td>
<td>48.4%</td>
<td>47.7%</td>
</tr>
<tr>
<td>executable</td>
<td>51.1%</td>
<td>45.7%</td>
</tr>
</tbody>
</table>

**Table 7.** Comparison of Snappy, QuickLZ and Gipfeli
## Gipfeli – High Speed Compression Algorithm

### Comparisons

<table>
<thead>
<tr>
<th>Program</th>
<th>Compression ratio</th>
<th>Time</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snappy</td>
<td>53%</td>
<td>2.8 s</td>
<td>354 MB/s</td>
</tr>
<tr>
<td>QuickLZ</td>
<td>46%</td>
<td>3.5 s</td>
<td>284 MB/s</td>
</tr>
<tr>
<td>Zlib fastest</td>
<td>43%</td>
<td>13.5 s</td>
<td>74 MB/s</td>
</tr>
<tr>
<td>Gipfeli</td>
<td>41%</td>
<td>4.3 s</td>
<td>232 MB/s</td>
</tr>
<tr>
<td>Zlib default</td>
<td>32%</td>
<td>41.7 s</td>
<td>24 MB/s</td>
</tr>
</tbody>
</table>

Table 8. First 1 GB of English Wikipedia
Comparisons

![Comparison Graph]

**Fig. 2.** Benchmark for plaintext.txt

**Introduction**

**Papers**
- Paper 1
- Paper 2
- Paper 3

**Conclusion**
Applications

- compression ratios 30% better than Snappy with slow-down around 30%
- higher speed for html content and remote procedure calls
- MapReduce technology
- Bigtable technology
Progression

- Linear time computation of Lempel–Ziv factorization (paper 1) which is the basis for efficient LZ77 and LZSS encoding
- Time and memory efficient Lempel-Ziv compression using suffix arrays (paper 2) which can be further improved
- Gipfeli – high speed and high compression ration data compression algorithm (paper 3)
Conclusion

- reduce resources usage, such as data storage space or transmission capacity
- no such thing as a "universal" compression algorithm
- Data has a universal but uncomputable probability distribution
- Trade-off remains
Thank You

Prepared By:

*Abdullah-Al Mamun*