Online compression of cache-filtered address traces

Pierre Michaud
INRIA
Problem

- RCDMA Tradeoff
  - Compression Ratio
  - Compression speed
  - Decompression speed
  - Memory usage
  - Accuracy (lossy compression)
Why yet another trace compressor?

- Depending on your problem, you may find the offered RCDMA tradeoff useful
  - High lossless compression ratio (on targeted traces) with moderate memory usage
  - Reasonably fast

- Very simple trace format → only addresses

- Leverages existing general-purpose lossless compressors

- Lossy compression mode
Outline

- The *bytesort* reversible transformation
- Lossy compression
- The ATC software
General-purpose lossless compressors

- gzip, bzip2, lzma, …

- Generally work at the byte level

- Able to compress inputs with lots of repeated substrings
Example of trace

No repeated substring, general-purpose compressors do not “see” the structure
Byte unshuffling

The first half of the trace (high-order bytes) exhibits a pattern, but general-purpose compressors do not “see” the structure of the second half (low-order bytes)
Bytesort: a reversible transformation

sort according to high-order byte (stable sort)

binary

The string 00 01 02 … 7F repeats twice
Bytesort: a **reversible** transformation

<table>
<thead>
<tr>
<th>F2</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
<tr>
<td>F2</td>
<td>A1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F2</th>
<th>F2</th>
<th>A1</th>
<th>F2</th>
<th>F2</th>
<th>A1</th>
<th>F2</th>
<th>F2</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7F</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F2</th>
<th>F2</th>
<th>A1</th>
<th>F2</th>
<th>F2</th>
<th>A1</th>
<th>F2</th>
<th>F2</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
</tr>
</tbody>
</table>

| FE | FF |
Bytesort: a reversible transformation
Bytesort: a reversible transformation
Apply bytesort recursively

| 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |
| FF 00 00 00 | 00 00 40 00 | 00 00 40 00 | FF 00 00 00 |
| 00 00 40 00 | 00 00 80 00 | 00 00 80 00 | FF 00 00 01 |
| FF 00 00 01 | 00 00 C0 00 | 00 00 C0 00 | FF 00 00 02 |
| 00 00 80 00 | 00 01 00 00 | FF 00 00 00 | FF 00 00 03 |
| FF 00 00 02 | 00 01 40 00 | FF 00 00 01 | FF 00 00 04 |
| 00 00 C0 00 | 00 01 80 00 | FF 00 00 02 | FF 00 00 05 |
| FF 00 00 03 | 00 01 C0 00 | FF 00 00 03 | FF 00 00 06 |
| 00 01 00 00 | FF 00 00 00 | FF 00 00 04 | FF 00 00 07 |
| FF 00 00 04 | FF 00 00 01 | FF 00 00 05 | 00 01 00 00 |
| 00 01 40 00 | FF 00 00 02 | FF 00 00 06 | 00 00 40 00 |
| FF 00 00 05 | FF 00 00 03 | FF 00 00 07 | 00 01 40 00 |
| 00 01 80 00 | FF 00 00 04 | 00 01 00 00 | 00 00 80 00 |
| FF 00 00 06 | FF 00 00 05 | 00 01 40 00 | 00 01 80 00 |
| 00 01 C0 00 | FF 00 00 06 | 00 01 80 00 | 00 00 C0 00 |
| FF 00 00 07 | FF 00 00 07 | 00 01 C0 00 | 00 01 C0 00 |
Bytesort implementation

• Use memory buffer of B addresses
  – If input trace larger than buffer, cut the input into blocks of size B
  – The larger the buffer, the higher the compression ratio

• Use counting sort $\rightarrow$ stable, time linear with B
  – Second buffer of size B
  – 1st pass: compute byte histogram
  – 2nd pass: put each address at proper position in second buffer using byte histogram information

• Inverse transformation $\rightarrow$ just do the reverse operation
  – Use 2 buffers and compute byte histogram
Evaluation

• 22 address traces
  – SPEC CPU 2006 compiled for x86-64 ➞ 64-bit addresses
  – Obtained with Pin
  – Filtering with 32KB L1 I-cache and 32KB L1 D-cache
  – Each trace is 100M addresses

• Bytesorted traces compressed with bzip2

• Compare with
  – bzip2
  – byte unshuffling + bzip2
  – VPC trace compressor generated with TCgen
    • Use bzip2
    • Memory consumption 230 MB (≈ bytesort with B=10M addresses)
Disk space usage

average bits per address

<table>
<thead>
<tr>
<th></th>
<th>8.6</th>
<th>5.3</th>
<th>3.6</th>
<th>3.3</th>
<th>2.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unshuffle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bytesort 1M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bytesort 10M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- 483
- 482
- 473
- 471
- 470
- 464
- 462
- 458
- 456
- 453
- 450
- 447
- 445
- 444
- 435
- 433
- 432
- 429
- 410
- 403
- 401
- 400
Decompression speed

- Decompress the 22 traces **sequentially** (2.2 billions addresses)
  - Measured on a Dell Precision T3400
  - Core 2 duo (dual core), 3 Ghz, 4MB L2 cache, 4GB memory
  - gcc -O3
  - Traces stored on local disk, output redirected to null device
Lossy compression

- The decompressed trace must “look” like the original trace
  -> **accuracy** problem
    - Accuracy is in the eye of the beholder, it depends on what we want to do with the trace

- Idea: cut the trace into fixed-length intervals and if an interval X “looks” like a previous interval Y, replace X with a pointer to Y
  - akin to Simpoint
What “look like” criterion?

- Somewhat arbitrary

- My choice: something simple that allows compressing random addresses

- \(\Rightarrow\) Sorted byte histograms
Sorted byte histograms (SBH)
Distance between two SBHs

\[ d(h_A, h_B) = \frac{\sum_{i=0}^{255} |h_A[i] - h_B[i]|}{\sum_{i=0}^{255} h_A[i]} \]

\[ = \frac{|256 - 250| + |128 - 131| + |0 - 3|}{256 + 128} \approx 0.03 \]
Distance between intervals X and Y

- Compute the SBHs for each byte column

- Compute the distance $D_n$ between the SBHs of X and Y for the $n^{th}$ byte column

- $D(X,Y) = \max_n D_n$
Algorithm

• Store in a *histogram table* the SBHs of recent intervals

• For each new interval Y, search in the histogram table the interval X that is closest to Y

• If D(X,Y) is less than fixed threshold, replace Y with pointer to X, otherwise store a *chunk* for Y
There is a problem!

- Example: address = random value between 1 and 20M
- All the intervals look like the first one \(\Rightarrow\) excellent compression ratio

- But...
  - First interval (10M addresses) contains only \(\sim 7.9M\) distinct addresses
  - \(\Rightarrow\) wrong working-set size for the whole trace
Solution: byte translation

interval X

interval Y, same column

At decompression, apply permutation on byte values in that column

Store this information along with pointer to X
Tuning

• Byte translation is important for high-order bytes
  – Not so good for low-order bytes
  – E.g., we would like to preserve constant strides

• Keep lowest-order bytes untranslated
  – 2 lowest-order bytes (empirical)
Evaluation

• Each trace is 1 billion addresses = 100 intervals
  – Interval length = 10M addresses

• Distance threshold = 0.1

• Chunks compressed with bytesort + bzip2
  – Buffer B = 1M addresses
Disk space usage

average bits per address

3.4
0.7
Accuracy: cache miss ratio

- Cheetah cache simulator
  - LRU replacement policy
  - Number of sets = 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k
  - Associativity = 1, 2, 3, 4, ..., 32
  - Compute maximum and average absolute error on miss ratio (320 points per trace)
Cache miss ratio: example

462.libquantum, 32k sets

miss ratio

exact
lossy

associativity
Address predictability

- C/DC address predictor
Address predictability (2)
The ATC software

- **Address Trace Compressor**
- Public release
  - [http://www.irisa.fr/caps/people/michaud/atc.html](http://www.irisa.fr/caps/people/michaud/atc.html)
  - Unix systems

**Lossy compression mode**

```
% cat /dev/urandom | bin2atc -k -m 100000000 foobar
% du -h foobar/*
  77M  foobar/1.bz2
  4.0K  foobar/INFO.bz2
% atc2bin foobar | wc -c
  800000000
```

800MB of random data compressed down to 80MB
Conclusion

• Finally…
  – Ran each benchmark to completion (reference inputs)
  – Total 22 traces \( \Rightarrow \) \( \sim \) 500 billions addresses = 4 TB of raw data
    • Recall: L1-filtered traces
  – Use lossy compression \( \Rightarrow \) 9 GB
    • Average 0.14 bits per address

• Caveat: don’t use the lossy compression mode without checking its applicability to your problem
Questions ?