Practical Data Compression for On-Chip Caches

November 26, 2019
Problem: CPUs are cheap and fast.
**Problem:** CPUs are cheap and fast. Memories are cheap, fast, capacious
**Problem:** CPUs are cheap and fast. Memories are cheap, fast, capacious (choose two out of three).
Problem: CPUs are cheap and fast. Memories are cheap, fast, capacious (choose two out of three). Fast CPU with slow memory doesn’t make sense.
Idea: Put fast memory between CPU and main memory.
**Idea:** Put fast memory between CPU and main memory. Fast – so small.

CPU looks for data in the cache;

CPU looks for data in the cache; if it finds it, it gets it from there and continues to work;

CPU looks for data in the cache; if it finds it, it gets it from there and continues to work; if it doesn’t find it, it looks for it in main memory.
Cache Compression

- **L0**: CPU registers hold words retrieved from cache memory.
- **L1**: L1 cache (SRAM) holds cache lines retrieved from L2 cache.
- **L2**: L2 cache (SRAM) holds cache lines retrieved from L3 cache.
- **L3**: L3 cache (SRAM) holds cache lines retrieved from memory.
- **L4**: Main memory (DRAM) holds disk blocks retrieved from local disks.
- **L5**: Local secondary storage (local disks) holds files retrieved from disks on remote network servers.
- **L6**: Remote secondary storage (distributed file systems, Web servers)

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices
<table>
<thead>
<tr>
<th>memory</th>
<th>latency</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>4 cycles</td>
<td>32KiB</td>
</tr>
<tr>
<td>L2</td>
<td>10 cycles</td>
<td>256KiB</td>
</tr>
<tr>
<td>L3</td>
<td>40-75 cycles</td>
<td>8MiB</td>
</tr>
<tr>
<td>DRAM</td>
<td>600 cycles</td>
<td>8GiB</td>
</tr>
<tr>
<td>HDD</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
Cache size: $S \times E \times B$ data bytes
Cache Compression

- 2 lines per set
- Address of short int: 0...01 100
- Find set
- 5 sets

Diagram showing a cache structure with sets and lines for indexing.
Cache Compression

![Diagram showing cache compression logic]

- **Valid? + Match: yes (= hit)**
- **Block Offset**
- **Address of short int:**
  - t bits: 0...01 100
- **Short int (2 Bytes) is here**
What can we do?

Compression!
Unfortunately, directly applying well-known compression algorithms (usually implemented in software) leads to high hardware complexity and unacceptable decompression/compression latencies, which in turn can negatively affect performance.
Compression:
**Compression:** takes place in background upon a cache fill.
Compression: takes place in background upon a cache fill.

Decompression:
Compression: takes place in background upon a cache fill.

Decompression: is on the critical path of a cache hit
**Compression:** takes place in background upon a cache fill.

**Decompression:** is on the critical path of a cache hit; we can only consider compression of the L2 caches.
Compression must be fast, simple, effective
Compression must be fast, simple, effective, the challenge is to find the right balance.
Base+Delta Encoding (B+Δ)
Base+Delta Encoding (B+Δ)
Base-Delta-Immediate (BΔI)
Observation:

for many cache lines, the data values stored within the line have a low dynamic range: i.e., the relative difference between values is small. In such cases, the cache line can be represented in a compact form using a common base value plus an array of relative differences.
**Observation:** for many cache lines, the data values stored within the line have a low dynamic range: i.e., the relative difference between values is small.
Observation: for many cache lines, the data values stored within the line have a low dynamic range: i.e., the relative difference between values is small. In such cases, the cache line can be represented in a compact form using a common base value plus an array of relative differences.
- **Zeros:** Zero is by far the most frequently seen value in application data. For example, zero is most commonly used to initialize data, to represent NULL pointers or false boolean values.
- **Zeros**: Zero is by far the most frequently seen value in application data. For example, zero is most commonly used to initialize data, to represent NULL pointers or false boolean values.

- **Repeated Values**: A large contiguous region of memory may contain a single value repeated multiple times. This pattern is widely present in applications that use a common initial value for a large array, or in multimedia applications where a large number of adjacent pixels have the same color.
Zeros: Zero is by far the most frequently seen value in application data. For example, zero is most commonly used to initialize data, to represent NULL pointers or false boolean values.

Repeated Values: A large contiguous region of memory may contain a single value repeated multiple times. This pattern is widely present in applications that use a common initial value for a large array, or in multimedia applications where a large number of adjacent pixels have the same color.

Narrow Values: A narrow value is a small value stored using a large data type: e.g., a one-byte value stored as a four-byte integer.
Benchmarks:

- libquantum – Physics: Quantum Computing
- ibm – Fluid Dynamics
- mcf – Combinatorial Optimization
- sjeng – Artificial Intelligence: chess
- omnetpp – Discrete Event Simulation
- sphinx3 – Speech recognition
- xalancbmk – XML Processing
- bzip2 – Compression
- leslie3d – Fluid Dynamics
- apache – Web server
- gromacs – Biochemistry/Molecular Dynamics
Benchmarks:

- astar – Path-finding Algorithms
- gobmk – Artificial Intelligence: go
- soplex – Linear Programming, Optimization
- gcc - C Compiler
- hmer – Search Gene Sequence
- wrf – Weather Prediction
- h264ref – Video Compression
- zeusmp – Physics / CFD
- cacutsADM – Physics / General Relativity
- GemsFDTD – Computational Electromagnetics
- Cache Compression
<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Zeros</th>
<th>Rep. Val.</th>
<th>Narrow</th>
<th>LDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decom. Lat.</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Complex.</td>
<td>Low</td>
<td>High</td>
<td>Modest</td>
<td></td>
</tr>
<tr>
<td>C. Ratio</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ZCA [8]</strong></td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>FVC [33]</strong></td>
<td>✔</td>
<td>Partly</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>FPC [2]</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td><strong>B ΔI</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 1: Qualitative comparison of B ΔI with prior work. LDR: Low dynamic range. Bold font indicates desirable characteristics.
Figure 1: Percentage of cache lines with different data patterns in a 2MB L2 cache. “Other Patterns” includes “Narrow Values”.
Figure 2: Effective compression ratio with different value patterns
Figure 3: Cache line from h264ref compressed with $B + \Delta$

Figure 4: Cache line from perlbench compressed with $B + \Delta$
Decompression:
Decompression:

\[ B^* \] – base value
Decompression:

\[ B^* \] – base value
\[ \Delta = \Delta_1, \Delta_2, \ldots, \Delta_n \] – array of differences
Decompression:

\( B^* \) – base value
\( \Delta = \Delta_1, \Delta_2, \ldots, \Delta_n \) – array of differences
\( S = (v_1, v_2, \ldots, v_n) \) – set of real values
Decompression:

\[ B^* \]  – base value
\[ \Delta = \Delta_1, \Delta_2, \ldots, \Delta_n \]  – array of differences
\[ S = (v_1, v_2, \ldots, v_n) \]  – set of real values
\[ v_i = B^* + \Delta_i \]
**Decompression:**

\[ B^* - \text{base value} \]
\[ \Delta = \Delta_1, \Delta_2, \ldots, \Delta_n - \text{array of differences} \]
\[ S = (v_1, v_2, \ldots, v_n) - \text{set of real values} \]
\[ v_i = B^* + \Delta_i - \text{SIMD-style vector adder.} \]
Compression:

Algorithm views a cache line as a set of fixed-size values i.e., 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line. Assume that the size of each value in the set is $k$ bytes and the set of values to be compressed is $S = (v_1, v_2, \ldots, v_n)$. The goal of the algorithm is to determine the value of the base, $B$, and the size of values in the set, $k$, that provide maximum compressibility.
Compression:

Algorithm views a cache line as a set of fixed-size values i.e., 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line.
Compression:

Algorithm views a cache line as a set of fixed-size values i.e., 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line.

Assume that the size of each value in the set is $k$ bytes and the set of values to be compressed is $S = (v_1, v_2, \ldots, v_n)$. 
**Compression:**

Algorithm views a cache line as a set of fixed-size values i.e., 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line.

Assume that the size of each value in the set is $k$ bytes and the set of values to be compressed is $S = (v_1, v_2, \ldots, v_n)$.

The goal of the algorithm is to determine the value of the base, $B^*$ and the size of values in the set, $k$, that provide maximum compressibility.
Compression:

The cache line is compressible only if \( \max(\text{size}(\Delta_i)) < k \).
**Compression:**

The cache line is compressible only if \( \max(\text{size}(\Delta_i)) < k \).

To determine the value of \( B^* \), either the value of \( \min(S) \) or \( \max(S) \) needs to be found.
Compression:

The cache line is compressible only if \( \max(\text{size}(\Delta_i)) < k \).

To determine the value of \( B^* \), either the value of \( \min(S) \) or \( \max(S) \) needs to be found.

The optimum can be reached only for \( \min(S) \), \( \max(S) \), or exactly in between them.
**Compression:**

The cache line is compressible only if \( \max(\text{size}(\Delta_i)) < k \).

To determine the value of \( B^* \), either the value of \( \min(S) \) or \( \max(S) \) needs to be found.

The optimum can be reached only for \( \min(S) \), \( \max(S) \), or exactly in between them.

Check all possible value of \( k \in \{2, 4, 8\} \) and compute \( B^* \).
Compression:

To avoid compression latency increase and reduce hardware complexity use the first value from the set of values as an approximation for the $B^*$. 
Choosing the first value as the base instead of computing the optimum base value reduces the average compression ratio only by 0.4%.
Surprise!

Choosing the first value as the base instead of computing the optimum base value reduces the average compression ratio only by 0.4%.
Some of applications can mix data of different types in the same cache line.
Some of applications can mix data of different types in the same cache line.

**Idea:** multiple bases.
Cache Compression

32-byte Uncompressed Cache Line

19-byte Compressed Cache Line
Why not more bases?
Figure 6: Effective compression ratio with different number of bases. “0” corresponds to zero and repeated value compression.
One simple trick!
One simple trick!
Figure 7: Compression ratio comparison of different algorithms: ZCA [8], FVC [33], FPC [2], B+Δ (two arbitrary bases), and BΔI. Results are obtained on a cache with twice the tags to accommodate more cache lines in the same data space as an uncompressed cache.
Figure 8: Compressor design. CU: Compressor unit.
32-byte Uncompressed Cache Line

8-byte Base Compression

\[ V_0 \quad V_1 \quad V_2 \quad V_3 \]

\[ \Delta_0 \quad \Delta_1 \quad \Delta_2 \quad \Delta_3 \]

1 byte sign extended?

1 byte sign extended?

1 byte sign extended?

1 byte sign extended?

Is every element 1-byte sign extended?

\[ B_0 =_{\text{def}} V_0 \quad \Delta_0 \quad \Delta_1 \quad \Delta_2 \quad \Delta_3 \]

8 bytes 1 byte

12-byte Compressed Cache Line

Figure 9: Compressor unit for 8-byte base, 1-byte $\Delta$
<table>
<thead>
<tr>
<th>Name</th>
<th>Base</th>
<th>Δ</th>
<th>Size</th>
<th>Enc.</th>
<th>Name</th>
<th>Base</th>
<th>Δ</th>
<th>Size</th>
<th>Enc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zeros</td>
<td>1</td>
<td>0</td>
<td>1/1</td>
<td>0000</td>
<td>Rep. Values</td>
<td>8</td>
<td>0</td>
<td>8/8</td>
<td>0001</td>
</tr>
<tr>
<td>Base8-Δ1</td>
<td>8</td>
<td>1</td>
<td>12/16</td>
<td>0010</td>
<td>Base8-Δ2</td>
<td>8</td>
<td>2</td>
<td>16/24</td>
<td>0011</td>
</tr>
<tr>
<td>Base8-Δ4</td>
<td>8</td>
<td>4</td>
<td>24/40</td>
<td>0100</td>
<td>Base4-Δ1</td>
<td>4</td>
<td>1</td>
<td>12/20</td>
<td>0101</td>
</tr>
<tr>
<td>Base4-Δ2</td>
<td>4</td>
<td>2</td>
<td>20/36</td>
<td>0110</td>
<td>Base2-Δ1</td>
<td>2</td>
<td>1</td>
<td>18/34</td>
<td>0111</td>
</tr>
<tr>
<td>NoCompr.</td>
<td>N/A</td>
<td>N/A</td>
<td>32/64</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: BΔI encoding. All sizes are in bytes. Compressed sizes (in bytes) are given for 32-/64-byte cache lines.
Figure 10: Decompressor design
Is it all?
Is it all?

No :(
Is it all?

No :( 

Problems:
Is it all?

No :(  

**Problems:** Unused space in compressed cache lines.
Is it all?

No :( 

**Problems:** Unused space in compressed cache lines.

**Ideas:** Put data in unused space.
Is it all?

No :( 

**Problems:** Unused space in compressed cache lines. We need to address cache lines.

**Ideas:** Put data in unused space.
Is it all?

No :( 

**Problems:** Unused space in compressed cache lines. We need to address cache lines.

**Ideas:** Put data in unused space. Change cache organisation.
Is it all?

No :

**Problems:** Unused space in compressed cache lines. We need to address cache lines. Cache eviction policy.

**Ideas:** Put data in unused space. Change cache organisation.
Is it all?

No :(  

**Problems:** Unused space in compressed cache lines. We need to address cache lines. Cache eviction policy.

**Ideas:** Put data in unused space. Change cache organisation. Evicts multiple LRU cache lines.
Conventional 2-way cache with 32-byte lines

Tag Storage:

Data Storage:

Set_0  Set_1  Set_N
Way_0  Way_0  Way_0
...    Tag_0  ...  ...
Way_1  ...  Tag_1  ...
...    ...    ...  ...
Set_N  ...    ...  ...

Set_0  Set_1  Set_N
Way_0  Way_0  Way_0
...    Data_0  ...  ...
Way_1  ...  Data_1  ...
...    ...    ...  ...
Set_N  ...    ...  ...

32 bytes
BΔI cache: 4-way tag storage, 8-byte segmented data storage

**Tag Storage:**

- Set\textsubscript{0}, Set\textsubscript{1}, ..., Set\textsubscript{N}
  - Way\textsubscript{0}, Way\textsubscript{1}, Way\textsubscript{2}, Way\textsubscript{3}
  - Tag\textsubscript{0}, Tag\textsubscript{1}, Tag\textsubscript{2}, Tag\textsubscript{3}

**Data Storage:**

- Set\textsubscript{0}, Set\textsubscript{1}, ..., Set\textsubscript{N}
  - S\textsubscript{0}, S\textsubscript{1}, S\textsubscript{2}, S\textsubscript{3}, S\textsubscript{4}, S\textsubscript{5}, S\textsubscript{6}, S\textsubscript{7}

- Tag\textsubscript{2} points to S\textsubscript{2}, uses 3 segments: S\textsubscript{2} - S\textsubscript{4}

- Compression encoding bits C

- 8 bytes
Conclusion:

- A new Base-Δ-Immediate compression mechanism
- Many cache lines can be efficiently represented using base+delta encoding
- Key properties
  - Low latency decompression
  - Simple hardware implementation
  - High compression ratio with high coverage
- Improves cache hit ratio and performance
- Outperforms state-of-the-art cache compression techniques: FVC and FPC
(standing ovation)