Dictionary code compression for low-energy instruction fetches for embedded systems

Mats Brorsson
Mikael Collin

KTH Royal Institute of Technology
Stockholm Sweden

Motivation

- Continuous increase of cpu controlled, mobile, and handheld products
  - Ever diminishing distinction between desktop & embedded applications
  - Increased software complexity and performance demands
    - More advanced processors
  - Energy consumption battery life time

- Objective & contribution of this work
  - Introduce a 2-level dictionary code compression method
    - Compression of both instructions and instruction sequences
    - Up to 16 instructions into a 32 bit entity
  - Improve on compression ratio (code density)
    - Reduce energy consumption in the instruction fetch path
Dictionary code compression

- **The basics**
  - Frequently executed instructions are replaced for short code words
  - Code words are then during execution substituted for the real instruction
  - Dictionary contents based on execution profiles
  - Energy benefits due to less I-cache accesses
    - Dynamic compression ratio
  - Smaller binaries, reduced code size
    - Static compression ratio

- **High compression ratio (code density)** depends on number of dictionary entries
  
  ![Diagram of dictionary code compression process]

  - Original code
  - Compressed code
  - Dictionary entries

- **Methods to improve code density**
  - Larger code words
    - More entries, larger dictionaries
    - Code word size affects code density
  - Smaller code words
    - Higher code density
    - Fewer dictionary entries
  - **Sequence compression**, [Lefurgy et al., MICRO-1997]
    - Code words represents a sequence of up to 4 uncompressed instructions
    - Requires a large dictionary
      - 256 code words results in a 4kB large dictionary
      - High possibility for unused dictionary space
Two-level compression

- Combination of instruction- and sequence compression
- Possible to fetch up to 16 instructions in a single 32 bit cache access
- Flexible and storage efficient
  - Compressed sequences are built using individually compressed instructions
  - Separate dictionaries
    - Instruction code words & Sequence code words

1-cache

Sequence dictionary

Code word dictionary

- 1-4 8 bit instruction codes (ICW)
- Re-instate the original instruction

Profiling & Compression procedure

Execution profiling

- Offline profiling of the uncompressed application
- Track & count the access frequency
- Block & split accesses to optimize the compression
- Based on the cost/performance policy

In-use Instruction reSchedule

Fitness function

Code generation

Sequence construction

Instruction allocation

Code word allocation

Sequence dictionary

Code word dictionary
Profiling and code analysis results

Total execution
- ~117000 analyzed Basic Blocks (BB:s)
  - Corresponds to 1.36 Billion instructions

90:10 rule of thumb
- Only 929 BB:s constitutes 90% of all profiled BB:s
  - Average 61.9 blocks per application
  - Only 75 BB:s > 16 instructions, ~ 5 per test application

Compression potential

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td># Basic Blocks (BB)</td>
<td>2381</td>
<td>23453</td>
<td>7833</td>
</tr>
<tr>
<td>Instructions/BB</td>
<td>4.12</td>
<td>5.69</td>
<td>4.96</td>
</tr>
<tr>
<td># Totally compressible BBs</td>
<td>36</td>
<td>178</td>
<td>108.1</td>
</tr>
<tr>
<td>Instructions/BB</td>
<td>4.15</td>
<td>9.56</td>
<td>5.42</td>
</tr>
<tr>
<td># Compressed BB:s using SCW:s</td>
<td>27</td>
<td>98</td>
<td>61.8</td>
</tr>
<tr>
<td>Instructions/BB</td>
<td>5.52</td>
<td>12.41</td>
<td>7.77</td>
</tr>
</tbody>
</table>

< 64
< 16
Merge potential

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td># BB:s with disjoint sequences</td>
<td>91</td>
<td>451</td>
<td>227</td>
</tr>
<tr>
<td>AVG # Instructions/BB</td>
<td>8.14</td>
<td>21.97</td>
<td>14.25</td>
</tr>
<tr>
<td># Merged sequences</td>
<td>202</td>
<td>939</td>
<td>583.3</td>
</tr>
<tr>
<td>AVG # Instructions/Merged sequence</td>
<td>2.76</td>
<td>5.78</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Code generation results

- Static compression, code size reduction, is compromised
  - More focus is put on dynamic properties
    - Profiling strategy and fitness function
  - Differences in method
    - 1-level: individual instructions, 2-level: sequences
- Sequence merge improves on static code size
  - Load-Store re-order has minimal impact on code size
**Architecture**

- Scalar 7 stage in-order pipeline (MIPS based ISA)
  - Dedicated decompression stages
  - On chip level-1 data & instruction caches
  - Off chip main memory
  - Bimodal branch predictor and branch target buffer (BTB)
  - Fetch word size 32 bits
- Sequence dictionary
  - Contains up to 256 35 bit entries (Sequence Words)
- Code word dictionary
  - Contains up to 256 31 bit entries (Code Words)

**Evaluation methodology**

- Architectures studied & compared
  - Base-line: 5-stage scalar pipeline (no compression)
  - 1-level: 6-stage pipeline, 256 entry dictionary + (a theoretical 512 entry dictionary)
  - 2-level: 7-stages pipeline, 256, 256 entry dictionary
    - On-chip level-1 caches, off-chip memory
    - 1024 entry bimodal branch predictor
    - 128 entry branch target buffer (BTB)
- Simulation using SimpleScalar & Wattch
  - Wattch enhanced with energy models for off-chip memory
  - Simulation parameters: 0.18µm, 1.8V, 400MHz
- Test applications
  - 15 MediaBench applications
- Measurements
  - Dynamic compression ratio
  - Energy savings and performance impact
Dynamic compression ratio

- About 8-47% improved fetch ratio, on average ~20%
- Dynamic compression ratio is primary affected by:
  - Number of instruction code words, dictionary size
  - Code composition
    - Number and size of the basic blocks

- More efficient to increase the number of instruction code words

- Dynamic ratio reflects on energy
  - 2-21% improvement on instruction fetch path energy
    - Fewer instruction cache accesses and BTB-lookups
    - Increased energy need for the two decompression stages & dictionaries
  - Yields ~9% energy improvement in total processor energy
    - The fetch path corresponds to ~30% of the entire processor
More performance data

- High miss-predict penalty impacts on performance
  - Prediction accuracy & number of branch/jump instructions

- Virtually increased cache capacity
  - Performance improvement for working sets larger than the physical cache size
Branch predictor performance impact

- High prediction accuracy is directly translated into better dictionary code compression
- Higher branch rates affect results

Conclusions

- 2-level dictionary compression method
  - Compression of up to 16 instructions long sequences, 1 cache access
  - Two separate dictionaries, two distinct code word types
  - Compression of both individual instructions and sequences
  - Storage effective architecture

- On average ~20% improvement on compression ratio
  - Corresponds to ~9% improvement in total processor energy consumption
  - Code word count still is the primary factor for good compression

- Performance
  - Working set and cache size
  - Number of branches & prediction accuracy is a dominant factor
Work in progress & future work

- Super block compression

- Add another level of indirection
  - Allow sequences to reference other sequences

- Analysis of different profiling methods and fitness options

- Feasibility and usage study of 2-level compression in multicore architectures

Thank you for your attention…

Questions?

mikaelco@kth.se
Code coverage

- Compression ratio improves with more code words
- Limits exist
- Room for potential improvements
  - Data organization and method

Properties and differences

**One-level**
- About 8-47% improved fetch ratio, on average ~20%
- Dynamic compression ratio is primarily affected by:
  - Number of instruction code words, dictionary size
  - Code composition
    - Number and size of the basic blocks

**Two-level**
- About 8-47% improved fetch ratio, on average ~20%
- Dynamic compression ratio is primarily affected by:
  - Number of instruction code words, dictionary size
  - Code composition
    - Number and size of the basic blocks
Limitation factors

- Code word design
  - Maximum of number of instructions per fetch
  - Flexible or static
- Compression granularity
  - Instructions, sequences, basic block, or other...
- Aligning policy
  - Is padding required
- Dictionary design
  - Homogenous code words or different sizes
- Handling of branch & jumps

The issue is how to obtain more instructions on each fetch.

Fetch- and sequence word layout

- Fetch word: The 32 bit entity fetched from memory
  - A single uncompressed instruction
    
    | 31 | 31 bit uncompressed (u-class) instruction |
    |----|------------------------------------------|
  
  - or 2-16 compressed instructions
    
    | 31 | compressed code representation |
    |----|---------------------------------|

- Sequence word: The 35 bit wide sequence dictionary entry
  
<table>
<thead>
<tr>
<th>31</th>
<th>Instruction code words &amp; immediate value combinations</th>
</tr>
</thead>
</table>

II

III
## Fetch word configurations

- **Compressed code**
  - Sequence code word(s), 2-16 instructions
  - Instruction code words, 2-3 instructions

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Index</th>
<th>Inst. P.W.</th>
<th>Branch</th>
<th>Data_present</th>
<th>Inst/FW</th>
<th>Instruction code words</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>4 x 7 bit sequence code words</td>
<td>0-127</td>
<td>8-36</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>001</td>
<td>3 x 8 bit sequence code words</td>
<td>0-32</td>
<td>4-8</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>010</td>
<td>2 x 8 bit sequence code words</td>
<td>0-32</td>
<td>4-8</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>011</td>
<td>1 x 8 bit sequence code word</td>
<td>N/A</td>
<td>2</td>
<td>no</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>100</td>
<td>3 x 8 bit instruction code words</td>
<td>N/A</td>
<td>5</td>
<td>no</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
<tr>
<td>101</td>
<td>2 x 8 bit instruction code words</td>
<td>N/A</td>
<td>2</td>
<td>no</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
<tr>
<td>110</td>
<td>2 ICWs + 1 x 8 bit immediate</td>
<td>N/A</td>
<td>2</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
</tbody>
</table>

*jump instructions still possible to use

## Sequence word configurations

- **Compressed code**
  - Sequence code word(s), 2-16 instructions
  - Instruction code words, 2-3 instructions

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Branch</th>
<th>Data_present</th>
<th>Inst/FW</th>
<th>Instruction code words</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>4 plain instruction code words</td>
<td>N/A*</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>001</td>
<td>3 instruction codes + 8 bit value</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>010</td>
<td>2 x 8 bit sequence code words</td>
<td>N/A*</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>011</td>
<td>1 x 8 bit sequence code word</td>
<td>N/A*</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>16 instructions</td>
</tr>
<tr>
<td>100</td>
<td>3 x 8 bit instruction code words</td>
<td>N/A*</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
<tr>
<td>101</td>
<td>2 x 8 bit instruction code words</td>
<td>N/A*</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
<tr>
<td>110</td>
<td>2 ICWs + 1 x 8 bit immediate</td>
<td>yes</td>
<td>COMPR CLAIM</td>
<td>32</td>
<td>3 instructions</td>
</tr>
</tbody>
</table>

*jump instructions still possible to use
Branch instruction issues

- Branch compression
  - Absolute jumps
    - Compress, update dictionary & memory
  - Indirect jumps
    - Compress, update jump tables
  - Relative branches (15-25% of all branches)
    - On average about 55% of the branch displacements are within 8 bits
    - Compress use 8 or 16 bit displacement

- Compress use 8 or 16 bit displacement

<table>
<thead>
<tr>
<th>Code word</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>Off</td>
<td>Off</td>
</tr>
</tbody>
</table>

Limitations

- Limited number of entries in the sequence dictionary
  - Only 32 blocks of size 13-16 are possible to compress into a single fetch
- At least a sequence of two instructions must be compressed
  - Limits both static and dynamic gain
    - Sequence merge helps a bit
- In general the basic blocks are too small to explore the full potential
  - On average 4-7 instructions
  - Oversized blocks are currently not selected for compression
    - The toast program
      - 401 instructions corresponds to 53% of all fetches
- All issues related to the extra pipeline stage
Desirable properties

- Basic block sizes of 8-12 instructions
- A working set consisting of up to 64 basic blocks
- Minimal number of oversized blocks
  - Small impact on the total number of fetches
- Small set of unique instruction patterns
  - Large reuse of same set of instructions throughout the blocks
- Inherently yield good predictions for branches