Fast Software-managed Code Decompression

Charles Lefurgy and Trevor Mudge

Advanced Computer Architecture Laboratory
Electrical Engineering and Computer Science Dept.
The University of Michigan, Ann Arbor

Compiler and Architecture Support for Embedded Systems (CASES)
October 1-3, 1999
Motivation

• **Problem: embedded code size**
  – Constraints: cost, area, and power
  – Fit program in on-chip memory
  – Compilers vs. hand-coded assembly

• **Solution: code compression**
  – Reduce compiled code size
  – Take advantage of instruction repetition

• **Benefits**
  – On-chip memory used more effectively
  – Trade-off performance for code density
  – Systems use cheaper processors with smaller on-chip memories
Hardware or software decompression?

- **Hardware**
  - Faster translation
  - CodePack, MIPS-16, Thumb

- **Software**
  - Smaller physical area
  - Lower cost
  - Quicker re-targeting to new compression algorithms
  - Rivals HW solutions on some (loopy) benchmarks
Kirovski et al., 1997

• **Overview**
  – Procedure Compression
  – Decompress and execute 1 procedure at a time
  – Store decompressed code in procedure cache
  – Cache management

• **Results**
  – 60% compression ratio on SPARC
  – 166% execution penalty with 64KB procedure cache
Dictionary compression algorithm

- Dictionary contains unique instructions
- Replace program instructions with short index
Compression ratio

- \( \text{compression ratio} = \frac{\text{compressed size}}{\text{original size}} \)

- **Compression ratios**
  - Dictionary: 65% - 82%
  - LZRW1: 55% - 63%

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Dict. Compression</th>
<th>LZRW1 Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc1</td>
<td>1,083,168</td>
<td>65.4%</td>
<td>60.4%</td>
</tr>
<tr>
<td>vortex</td>
<td>495,248</td>
<td>65.8%</td>
<td>55.5%</td>
</tr>
<tr>
<td>go</td>
<td>310,576</td>
<td>69.6%</td>
<td>63.9%</td>
</tr>
<tr>
<td>perl</td>
<td>267,568</td>
<td>73.7%</td>
<td>60.2%</td>
</tr>
<tr>
<td>jpeg</td>
<td>198,272</td>
<td>77.2%</td>
<td>61.5%</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>119,600</td>
<td>82.5%</td>
<td>60.5%</td>
</tr>
<tr>
<td>pegwit</td>
<td>88,800</td>
<td>79.5%</td>
<td>56.7%</td>
</tr>
</tbody>
</table>
Decompression code

- **Simple**
  - Small static code size: 25 instructions

- **Fast**
  - Less than 3 instructions per output byte
  - 74 dynamic instructions per decompressed cache line

- **Algorithm**
  - Invoke decompressor on L1 I-cache miss
  - Decompress 1 complete cache line
  - For each instruction in cache line
    - Read index
    - Reference dictionary with index to get instruction
    - Put instruction in I-cache

- **HW Support**
  - L1-cache miss exception
  - Write into I-cache
Optimizations

• **Partial decompression**
  – compress from missed instruction to end of cache line
  – use a valid bit per word in cache line to mark instructions at beginning of line as invalid
  – avoids decompressing instructions that may not be executed
  – up to 12% speedup

• **Second register file**
  – Many embedded processors have an additional register file
  – Avoid save/restore of registers when decompressor runs
  – 2nd register file with partial decompression: up to 16% speedup
Simulation environment

- **SimpleScalar**
  - Modified to support compression
- **5 stage, in-order pipeline**
  - Simple embedded processor
- **D-cache**
  - 8KB, 16B lines, 2-way
- **I-cache**
  - 1 to 64KB, 32B lines, 2-way
- **Memory**
  - 10 cycle latency, 2 cycle rate
Performance: *cc1*

![Graph showing slowdown relative to native code for different cache sizes](chart)

- **Compressed**
- **Partial**
- **Partial+Regfile**
- **Native**

**Axes:**
- **Y-axis:** Slowdown relative to native code
- **X-axis:** I-cache size (KB)

**Legend:**
- Compressed
- Partial
- Partial+Regfile
- Native
Performance: \textit{ijpeg}

- \textit{compressed}
- \textit{partial}
- \textit{partial+regfile}
- \textit{native}

Slowdown relative to native code
Performance summary

- Data from CINT95, MediaBench with several cache sizes
- Control slowdown by optimizing I-cache miss ratio
  - Code layout may help
Performance summary, cont.

- Magnification of previous graph
- Slowdown under 3x when I-miss ratio is under 2%
- Slowdown under 2x when I-miss ratio is under 1%
Conclusions

• **Line-based decompression beats procedure-based**
  – use normal cache as decompression buffer
  – no fragmentation management as in procedure-based decompression
  – order of magnitude performance difference
  – A previous decompressor with procedure granularity had 100x slowdown on *gcc* and *go* [Kirovski97]

• **Compressed code fills gap**
  – has quick execution of native code
  – has small size of interpreted code
http://www.eecs.umich.edu/~tnm/compress