Efficient Execution of Compressed Programs

Charles Lefurgy

http://www.eecs.umich.edu/compress

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

- **Microprocessor die cost**
  - Low cost is critical for high-volume, low-margin embedded systems
  - Control cost by reducing area and increasing yield

- **Increasing amount of on-chip memory**
  - Memory is 40-80% of die area [ARM, MCore]
  - In control-oriented embedded systems, much of this is program memory

- **How can program memory be reduced without sacrificing performance?**
Solution

• **Code compression**
  – Reduce compiled code size
  – Compress at compile-time
  – Decompress at run-time

• **Implementation**
  – Hardware or software?
  – Code size?
  – Execution speed?
Research contributions

- **HW decompression** [MICRO-32, MICRO-30, CASES-98]
  - Dictionary compression method
  - Analysis of IBM CodePack algorithm
  - HW decompression increases performance

- **SW decompression** [HPCA-6, CASES-99]
  - Near native code performance for media applications
  - Software-managed cache
  - Hybrid program optimization (new profiling method)
  - Memoization optimization
Outline

Compression overview and metrics
HW decompression overview
SW decompression
• Compression algorithms
  – Dictionary
  – CodePack
• Hardware support
• Performance study
• Optimizations
  – Hybrid programs
  – Memoization
Why are programs compressible?

- **Ijpeg benchmark** (MIPS gcc 2.7.2.3 -O2)
  - 49,566 static instructions
  - 13,491 unique instructions
  - 1% of unique instructions cover 29% of static instructions
Evaluation metrics

- **Size**

\[ \text{compression ratio} = \frac{\text{compressed size}}{\text{original size}} \]

- **Decode efficiency**
  - Measure program execution time
Previous results

<table>
<thead>
<tr>
<th>Who</th>
<th>Instruction Set</th>
<th>Compression Ratio</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thumb</td>
<td>ARM</td>
<td>70%</td>
<td>16-bit instruction subset of 32-bit ISA</td>
</tr>
<tr>
<td>MIPS-16</td>
<td>MIPS</td>
<td>60%</td>
<td>16-bit instruction subset of 32-bit ISA</td>
</tr>
<tr>
<td>CodePack</td>
<td>PowerPC</td>
<td>60%</td>
<td>Cache line compression</td>
</tr>
<tr>
<td>Wolfe</td>
<td>MIPS</td>
<td>73%</td>
<td>Cache line, Huffman</td>
</tr>
<tr>
<td>Lekatsas</td>
<td>MIPS, x86</td>
<td>50%, 80%</td>
<td>Cache line, stream division, Huffman</td>
</tr>
<tr>
<td>Araujo</td>
<td>MIPS</td>
<td>43%</td>
<td>Cache line, op. factorization, Huffman</td>
</tr>
<tr>
<td>Liao</td>
<td>TMS320C25</td>
<td>82%</td>
<td>Procedure abstraction</td>
</tr>
<tr>
<td>Kirovski</td>
<td>SPARC</td>
<td>60%</td>
<td>Procedure compression</td>
</tr>
<tr>
<td>Ernst</td>
<td>SPARC</td>
<td>20%</td>
<td>Interpreted wire code</td>
</tr>
</tbody>
</table>

- Many compression methods
  - Compression unit: instruction, cache line, or procedure
- Difficult to compare previous results
  - Studies use different instruction sets and benchmarks
- Many studies do not measure execution time
- Wire code
  - Small size: a goal for embedded systems
  - Problem: no random access, must decompress entire program at once
Hardware decompression
CodePack

• **Overview**
  – IBM
  – PowerPC instruction set
  – 60% compression ratio, ±10% performance [IBM]
    • performance gain due to prefetching

• **Implementation**
  – Binary executables are compressed after compilation
  – Decompression during instruction cache miss
    • Instruction cache holds native code
    • Decompress two cache lines at a time (16 insns)
  – PowerPC core is unaware of compression
CodePack encoding

32-bit PowerPC instruction word

Encoding for upper 16 bits

8 0 0 | Tag
32 0 1 | Index
64 1 0 0 | Raw bits
128 1 0 1 |
256 1 1 0 |
1111 |

Encoding for lower 16 bits

1 0 0 | Escape
16 0 1 |
32 1 0 0 |
128 1 0 1 |
256 1 1 0 |
1111 |
CodePack system

- **CodePack is part of the memory system**
  - After L1 instruction cache

- **Dictionaries**
  - Contain 16-bit upper and lower halves of instructions

- **Index table**
  - Maps instruction address to compressed code address

Instruction memory hierarchy
CodePack decompression

Fetch index

L1 I-cache miss address

Index table (in main memory)

Fetch compressed instructions

Byte-aligned block address

Compressed bytes (in main memory)

Compression Block (16 instructions)

Decompress

1 compressed instruction

Hi tag | Low tag | Hi index | Low index

High dictionary

Low dictionary

Native Instruction

High 16-bits

Low 16-bits
Instruction cache miss latency

- **Native code uses critical word first**
- **Compressed code must be fetched sequentially**

### Diagram Explanation

**a) Native code**
- Instruction cache miss
- Instructions from main memory

**b) Compressed code**
- Instruction cache miss
- Index from main memory
- Codes from main memory
- Decompressor

**c) Compressed code + optimizations**
- Instruction cache miss
- Index from index cache
- Codes from main memory
- Two Decompressors

**Timeline**
- $t=0$
- 10
- 20
- 30

**Legend**
- **L1 cache miss**
- **Fetch instructions (first line)**
- **Decompression cycle**
- **Fetch index**
- **Fetch instructions (remaining lines)**
- **Critical instruction word**
Comparison of optimizations

• **Index cache provides largest benefit**

• **Optimizations**
  – index cache: 64 lines, 4 indices/line, fully assoc.
  – 2nd decoder

• **Speedup over native code: 0.97 to 1.05**

• **Speedup over CodePack: 1.17 to 1.25**
Hardware decompression conclusions

- **Performance can be improved at modest cost**
  - Remove decompression overhead
    - index lookup
    - dictionary lookup
  - Better memory bus utilization

- **Compression can speedup execution**
  - Compressed code requires fewer main memory accesses
  - CodePack includes simple prefetching

- **Systems that benefit most from compression**
  - Narrow memory bus
  - Slow memory
Software decompression
Software decompression

• Previous work
  – Whole program compression [Tauton91]
    • Saved disk space
    • No memory savings
  – Procedure compression [Kirovski97]
    • Requires large decompression memory
    • Fragmentation of decompression memory
    • Slow

• My work
  – Decompression unit: 1 or 2 cache-lines
  – High performance focus
How does code compression work?

- **What is compressed?**
  - Individual instructions
- **When is decompression performed?**
  - During I-cache miss
- **How is decompression implemented?**
  - I-cache miss invokes exception handler
- **What is decompressed?**
  - 1 or 2 cache lines
- **Where are decompressed instructions stored?**
  - I-cache is the decompression buffer
Dictionary compression algorithm

- **Goal**: fast decompression
- **Dictionary contains unique instructions**
- **Replace program instructions with short index**

```
Original program

.text segment
lw r2, r3
lw r2, r3
lw r15, r3
lw r15, r3
lw r15, r3

Compressed program

.text segment (contains indices)
lw r2, r3
lw r15, r3

.dictionary segment
```
## Decompression

**Algorithm**
1. I-cache miss invokes decompressor (exception handler)
2. Fetch index
3. Fetch dictionary word
4. Place instruction in I-cache (special instruction)

- **Write directly into I-cache**
- **Decompressed instructions only exist in I-cache**

![Diagram of decompression process](image)
Hardware support

- **Decompression exception**
  - Raise exception on I-cache miss
  - Exception not raised on native code section (allow hybrid programs)
  - Similar to Informing Memory [Horowitz98]

- **Store-instruction instruction**
  - MAJC, MIPS R10K
### Two software decompressors

- **Dictionary**
  - Faster
  - Less compression
- **CodePack**
  - A software version of IBM’s CodePack hardware
  - Slower
  - More compression

<table>
<thead>
<tr>
<th></th>
<th>Dictionary</th>
<th>CodePack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codewords (indices)</td>
<td>Fixed-length</td>
<td>Variable-length</td>
</tr>
<tr>
<td>Decompress granularity</td>
<td>1 cache line</td>
<td>2 cache lines</td>
</tr>
<tr>
<td>Static instructions</td>
<td>43</td>
<td>174</td>
</tr>
<tr>
<td>Dynamic instructions</td>
<td>43</td>
<td>1042-1062</td>
</tr>
<tr>
<td>Decompression overhead</td>
<td>73-105 cycles</td>
<td>1235-1266 cycles</td>
</tr>
</tbody>
</table>
Compression ratio

- \( \text{compression ratio} = \frac{\text{compressed size}}{\text{original size}} \)
  - CodePack: 55% - 63%
  - Dictionary: 65% - 82%
Simulation environment

- **SimpleScalar**
- **Pipeline**: 5 stage, in-order
- **I-cache**: 4KB, 32B lines, 2-way
- **D-cache**: 8KB, 16B lines, 2-way
- **Memory**: embedded DRAM
  - 10 cycle latency
  - bus width = 1 cache line
  - 10x denser than SRAM caches

- **Performance**: slowdown = 1 / speedup (1 = native code)
- **Area results include**:
  - Main memory to hold program (compressed bytes, tables)
  - I-cache
  - Memory for decompressor optimizations (memorization, native code)
Performance

- **CodePack**: very high overhead
- **Reduce overhead by reducing cache misses**

<table>
<thead>
<tr>
<th>I-cache size (KB)</th>
<th>4KB</th>
<th>16KB</th>
<th>64KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodePack</td>
<td>19.19</td>
<td>3.17</td>
<td>1.26</td>
</tr>
<tr>
<td>Dictionary</td>
<td>2.66</td>
<td>1.43</td>
<td>1.04</td>
</tr>
<tr>
<td>Native</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Ghostscript**

- Slowdown
- CodePack
- Dictionary
- Native
Cache miss

- Control slowdown by optimizing I-cache miss ratio
- Small change in miss ratio → large performance impact

![Graph showing the relationship between I-cache miss ratio and slowdown relative to native code. The graph includes data points for different cache sizes (CodePack and Dictionary) at 4KB, 16KB, and 64KB.]
Two optimizations

• **Hybrid programs (static)**
  – Both compressed and native code

• **Memoization (dynamic)**
  – Cache recent decompressions in main memory

• **Both can be applied to any compression algorithm**
Hybrid programs

- **Selective compression**
  - Only compress some procedures
  - Trade size for speed
  - Avoid decompression overhead

- **Profile methods**
  - Count dynamic instructions
    - Example: ARM/Thumb
    - Use when compressed code has more instructions
    - Reduce number of executed instructions

  - Count cache misses
    - Example: CodePack
    - Use when compressed code has longer cache miss latency
    - Reduce cache miss latency
Cache miss profiling

- **Cache miss profile reduces overhead 50%**
- **Loop-oriented benchmarks benefit most**
  - Profiles are different in loop regions

---

**Pegwit (encryption)**

**Slowdown relative to native code**

**Compression ratio**

- **CodePack: dynamic instructions**
- **CodePack: cache miss**
Code placement

Original code

Memory

Whole compression

compressed code     decompress region (in L1 cache only)

dict a b c d

Selective compression

compressed code     native region     decompress region

dict a c B D

Decompress

Same order

dict a c B D

Decompress

Different order!
**CodePack vs. Dictionary**

- **More compression may have better performance**
  - CodePack has smaller size than Dictionary compression
  - Even with some native code, CodePack is smaller
  - CodePack is faster due to using more native code

**Graph: Ghostscript**

- Slowdown relative to native code
- Compression ratio vs. Slowdown
  - Green: CodePack: cache miss
  - Purple: Dictionary: cache miss
Memoization

- **Reserve main memory for caching decompressed insns.**
  - Use high density DRAM to store more than I-cache in less area
  - Manage as a cache: data and tags

- **Algorithm**
  - Decompressor checks memo table before decompressing
  - On hit, copy instructions into I-cache (no decompression)
  - On miss, decompress into I-cache and update memo table

**No memoization**

**With memoization**

![Diagram showing the process of decompression with and without memoization.](image-url)
Memoization results

- 16KB memoization table
- CodePack: large improvement
- Dictionary is already fast: small improvement
Combined

- **Use memoization on hybrid programs**
  - Keep area constant. Partition memory for best solution.
- **Combined solution is often the best**

![Graph showing slowdown for CodePack with 32KB decompression buffer]

- 0KB memo
- 4KB memo
- 8KB memo
- 16KB memo
- 32KB memo (not hybrid)
Combined

- Adding DRAM is better than larger SRAM cache
Conclusions

• **High-performance SW decompression possible**
  – Dictionary faster than CodePack, but 5-25% compression ratio difference
  – Hardware support
    • I-cache miss exception
    • Store-instruction instruction

• **Tune performance**
  – Cache size
  – Hybrid programs, memoization

• **Hybrid programs**
  – Use cache miss profile for loop-oriented benchmarks

• **Memoization**
  – No profile required, but has start-up latency
  – Effective on large working sets
Future work

- Compiler optimization for compression
  - Improve compression ratio without affecting performance
- Unify selective compression and code placement
  - Reduce I-cache miss to improve performance
- Energy consumption
  - Increasing run-time may use too much energy
  - Improving bus utilization saves energy
- Dynamic code generation: Crusoe, Dynamo
  - Memory management, code stitching
- Compression for high-performance
  - Lower L2 miss rate
http://www.eecs.umich.edu/compress