A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems

Luca Benini, Francesco Menichelli, Mauro Olivieri; IEEE Transactions on Computers
Volume: 53, Issue: 4, April 2004

Presenter: Chung-Hung Lai
Abstract

Compression of executable code in embedded microprocessor systems, used in the past mainly to reduce the memory footprint of embedded software, is gaining interest for the potential reduction in memory bus traffic and power consumption. We propose three new schemes for code compression, based on the concepts of static (using the static representation of the executable) and dynamic (using program execution traces) entropy and compare them with a state-of-the-art compression scheme, IBM’s CodePack. The proposed schemes are competitive with CodePack for static footprint compression and achieve superior results for bus traffic and energy reduction. Another interesting outcome of our work is that static compression is not directly related to bus traffic reduction, yet there is a trade off between static compression and dynamic compression, i.e., traffic reduction.
Outline

- What’s the Problem?
- Introduction
- Theoretical Foundation
- Code Compression Schemes
  1. Scheme 1: the basic approach
  2. Scheme 2: optimization for bus traffic reduction
  3. Scheme 3: optimization for memory size reduction
- Evaluation Methodology
- Experimental Results
- Conclusion
What’s the problem?

- RISC instruction sets are generally very regular, with fixed-length instructions, however it hinders code compaction. So achieving sufficient instruction fetch bandwidth is therefore one of the most challenging issues in the design of a memory system for a RISC processor.
Introduction

- In recent years, several techniques have addressed the scarce “information density” of RISC ISAs through various forms of executable code compression:
  1. Dense instruction sets (e.g., ARM Thumb, MIPS16…)
     - This approach has two shortcomings:
       1) Performance penalty
       2) Requiring modification to the processor core itself
  2. External hardware decompression unit
     - External decompression doesn’t require changes to the core architecture. Hence, it is a good choice for embedded designs based on IP cores.
In this paper, we refer to code compression optimized for memory size reduction as static compression, it is based on static frequency distribution.

- During execution, The most statically frequent instructions are not necessarily the most executed.

we refer to code compression optimized for bus traffic reduction as dynamic compression, it is base on execution frequency distribution.

- Dynamic compression doesn’t necessarily realize size reduction.

code size and bus traffic reduction are not necessarily positive correlated
The distribution is highly nonuniform, the first 64 and 256 most frequently executed instructions, the percentages (with respect to total number of fetches) are about 74% and 94%.

- Dynamic optimization can be concentrated on the small group (64 or 256) of most frequently executed instruction.
Relevant Background of Proposed Compression Schemes

- Decompress-on-fetch architectures.
  - The processor generates the same address (noncompressed case)
  - Memory is always accessed on word (4 bytes) boundaries.

- Substituting of fixed-length codewords in place of the original instructions (where \( k \) distinct instructions are replaced by \( \log_2 k \) bit long codewords)

- Dictionary-based external code compression schemes.
  - The method requires two memory access at every instruction fetch, one to read the codeword from the compressed code, and one to read the dictionary.

- Splitting dictionary in two sections
  - keep the most frequency executed instructions in a fast memory (internal to the decompression unit) as fast dictionary.
  - The remaining instructions can be kept in main memory as slow dictionary.
This scheme is our fundamental technique.
When the requested index is contained in the buffer and the instruction is contained in the fast dictionary (which is very frequent), no memory access are needed and decompression can be accomplished in one cycle.
Note that indexes are shorter than scheme 1 as they only have to index fast dictionary. (index size = \( \log_2 256 = 8 \) bits)

- Fast dictionary only contained 255 instructions, as one index is reserved to mark that the requested instruction is not present in the dictionary.

In dictionary instructions will never be read from the program memory, their presence is used to maintain the original memory address of non-in-dictionary instructions.
This scheme represents an optimization of scheme 1 for bus traffic reduction during execution, because only small 8 bits codeword are transferred from main memory, expect for fast dictionary misses.
Patterns are compressed using variable length codewords

Scheme 3 decompression algorithm

- We store instructions in two dictionaries as in scheme 1, however, instructions in the slow dictionary are further compressed.
- The dictionary can’t be compressed considering entire instructions are all distinct. We need to split instructions into smaller patterns (two 16 bit patterns) to find repetitions that allow compression.
We call a group of consecutive equal-length codewords a **block**

- After compressing, Different colors mean different codeword length.
- After reordering, codewords having the same length have become consecutive.
Comparison results allow us to get a pointer to the position of the block’s first instruction.

The block’s starting indexes are compared in parallel with the requested index.
Since all codewords in a block have the same length, the offset of the requested instruction from the block starting address can be computed as in Fig. 9.
Scheme 3 emphasizes static code footprint reduction at a price of increased decoder complexity.
Evaluation Methodology

- We performed our tests using the follow procedure:
  - **Data preparation:**
    - Program compilation.
    - Simulation (using simplescalar simulator) and file logging of memory access traces generated during instruction-fetch.
  - **Compression**, exploiting the statistical data obtained during simulation (instructions execution frequency).
  - **Decompression** (using our simulator), bus-traffic, and execution time computation.
Evaluation Methodology

- In order to have a reference for the performance evaluation of our compression scheme, we compare it with IBM’s CodePack.
  - The proposed compression methods are suited to a decompression on fetch architecture.
  - CodePack targets mainly decompression on instruction cache refills (a 16-instruction compression-block corresponding to a cache line is processed at each decompression request).
  - In our architecture, decompression is activated when the requested instruction belongs to a new compression-block.
Evaluation Metrics

- Static efficiency is evaluated by the **compression ratio**, defined as:
  \[
  \text{compression ratio} = \frac{\text{compressed size}}{\text{original size}}
  \]
  - Compressed size includes all structures requiring storage (i.e., compressed instructions, dictionaries, conversion tables, block mapping).

- The potential traffic reduction is evaluated by **relative memory traffic**, defined as:
  \[
  \text{relative traffic} = \frac{\text{total traffic with compression}}{\text{total traffic without compression}}
  \]
  - Total traffic means the total number of bytes transferred through memory bus because of instruction fetch during execution.
Evaluation Metrics

- Energy efficiency is evaluated by the energy ratio, defined as:

\[
\text{energy ratio} = \frac{\text{total fetch energy with compression}}{\text{total fetch energy without compression}}
\]

where

\[
\text{total fetch energy} = \sum_i \text{ith instruction fetch energy}
\]

\[
= \sum_i (\text{memory read energy}_i + \text{bus energy}_i + \text{decompression energy}_i),
\]

- Decompression speed efficiency is evaluated by the fetch time ratio, defined as:

\[
\text{fetch time ratio} = \frac{\text{total fetch time with compression}}{\text{total fetch time without compression}}
\]

where

\[
\text{total fetch time (cycles)} = \sum_i \text{ith instruction fetch time}
\]

- The ith instruction fetch time depends on the memory access latency and the decompression latency.
Experimental Results - Compression Ratio

- Scheme 1 has an average compression ratio of 75%, Scheme 2 about 125%, Scheme 3 about 62%.

Compared to CodePack, our most optimized for memory size reduction Scheme (Scheme 3) reaches the same compression ratio.
Scheme 2 reduces the average traffic to 35%. Scheme 1 and 3 have similar behaviors and reduce the average traffic to 50% - 52%.

CodePack has the worst traffic performances, its relative traffic ranges from 94% to 125%.

The reason for such high traffic is memory access overhead due to the decompression of a whole block of instructions whenever jumping to a new compression-block.
Experimental Results - Energy

- This measure is related to the compression scheme capability of lowering bus energy consumption by reducing bit toggles per instruction fetch.
  - The potential reduction comes directly from bus traffic reduction.

Fig. 14. Equivalent switching activity per instruction.
Memory access energy is the most significant part of the instruction fetch energy, so every scheme which reduces traffic attains significant fetch energy reduction.

CodePack has the worst performance, the reason is that CodePack has been designed to work in cooperation with cache, activating the decompression only during a cache-refill.
Experimental Results - Energy

This analysis using a cache between the CPU and the decompressor (i.e., the architecture CodePack was designed for, where decompression is activated only on cache refills).

Increasing the cache size, improving CodePack’s performance.

Therefore, in this architecture, the adoption of code compression is less effective for energy reduction.
Experimental Results - Fetch Time

- Fetch time for Scheme 1 and Scheme 2:
  - One cycle if the compressed instruction is in the read buffer and it produces a fast dictionary hit.
  - \(L_1 + 1\) cycles if the compressed instruction is read from memory and it produces a fast dictionary hit.
  - \(L_1 + L_2 + 1\) cycles if the compressed instruction is read from memory and it produces a fast dictionary miss (two memory access).

- Fetch time for Scheme 3:
  - One cycle if the compressed instruction is in the read buffer and it produces a fast dictionary hit.
  - \(L_1 + 1\) cycles if the compressed instruction is read from memory and it produces a fast dictionary hit.
  - \(1 + L_2 + 1\) cycles if the compressed instruction is in read buffer and it produces a fast dictionary miss.
  - \(1 + L_1 + L_2 + 1\) cycles if the compressed instruction is read from memory and it produces a fast dictionary miss (two memory access).

Assuming \(L_1\) and \(L_2\) refer to the latency of two different memory accesses.
Experimental Results - Fetch Time

- When the memory is very slow with respect to the decompression latency, in spite of the additional decompression delay, the system is faster as it performs fewer memory accesses.
- CodePack performances are the worst since it introduces many unnecessary memory access due to block decompression overhead.
Experimental Results – Fetch Time

- Fetch time ratio is highly dependent on fast-dictionary hit-rate.
- When the fast-dictionary hit-rate is about 92%, the compressed and uncompressed architecture have the same total fetch time.
- Scheme 3 is slightly slower than scheme 1 are due to the presence of a clock cycle delay at fast-dictionary miss (dictionary decompression).
Conclusion

- The proposed schemes can be applied in cacheless architectures or in decompress-on-fetch architecture.
  - i.e., architectures where even the first-level cache stores compressed instructions.
  - However, CodePack was primarily developed for precache decompression.

- Our experiment shows that there is a clear trade off between static code size compression and all other metrics (bandwidth, speed, and energy) which are related to dynamic compression.
  - Scheme 2, which actually increase static memory size, outperforms all other schemes for traffic, fetch time, energy.

This means a controlled amount of static footprint increase can be traded off for much reduced energy, access time, and bus traffic.