

Code compression using instruction templates

Jeroen Trum

Products like wireless telephones, set-top boxes and hand-held PCs contain megabytes of embedded software. The memories in which this software is stored, occupy a significant amount of silicon area. With compression techniques, the required amount of memory can be reduced. The code is stored in memory in a compressed form, and decompressed on-the-fly by a decoder. Jumps in the program cause the need for a random-access decompression scheme. Therefore, well-known data compression techniques such as Lempel-Ziv do not apply. Prior work describes solutions in which macro instructions are introduced to replace common sequences of instructions (Lefurgy, Liao, and Philips' ThumbScrews). Operand factorisation was investigated by Lekatsas and Araújo. These techniques yield a good compression ratio, but with a high performance penalty. A cache-line Huffman decoder was presented by Wolfe et al. IBM's CodePack uses a variable-length encoding that reduces encoding lengths of frequent instruction words. Both methods have a low performance impact, but the code size results are less good. Our aim is to achieve the compression ratios achieved with the first techniques at a performance penalty comparable to that of the latter ones.

This thesis presents a code compression technique that can be used on fixed-length encoded instruction sets, like the ones used in popular RISC processors such as ARM and MIPS. Taking the original fixed-length instruction encoding as a basis, we construct a variable-length instruction encoding scheme with which the code size is reduced. The re-encoded program is stored in memory. A decoder translates variable-length encoded instructions to the original fixed-length instructions which can be executed on the original processor. This happens on-the-fly on an instruction-by-instruction basis. The decoder can be implemented either in hardware or as a software interpreter. Like program memory, the decoder occupies silicon area, so the size of the decoder must be taken into account in cost calculations.

An optimal variable-length encoding scheme is Huffman encoding. A typical program contains tens of thousands of different instruction words. A Huffman encoding on this set of instructions requires a mapping of equally many entries. Such a mapping yields an unacceptably large decoder. Instead, we propose a technique that only partly encodes the original instruction words using a Huffman-like encoding. We introduce instruction templates that define the bit value of some of the fixed-length instruction word bits. Instructions that have sufficient bit-level similarity are encoded by the same bit template. An instruction is encoded by two sequences of bits: first the encoding of the template that is used for this instruction, and secondly the bits that were not defined by the template. With a small set of bit templates all possible instructions can be encoded. The size of the decoder is roughly linear with the number of templates. The optimisation problem to tackle is to find a small set of templates yielding a minimal code size.

As we change the length of instruction encodings, we also change the addresses of the instructions. Since we want the decoder to be transparent to the processor, the decoder has to deal with the new encoding's address space. The decoder keeps track of a program counter in the new address space, and assures that control flow changes requested by the processor are dealt with correctly. With a variable-length encoding the addressing grain is finer than with fixed-length encoding. With respect to hardware complexity, it can be attractive to use an encoding grain that is less fine than the finest possible. If we allow templates to have any number of parameter bits, the lengths of the template encodings must assure the total instruction encoding length to be a multiple of the grain. This restriction makes plain Huffman encoding unsuitable. We introduce a Huffman-like algorithm that computes a variable-length encoding for a set of symbols of which for each symbol the encoding length modulo the grain is predefined. We prove that, with these restrictions, the resulting average codeword length is minimal.

The template selection problem is tackled by approximation using simulated annealing. The set of instructions is partitioned into N sets, with N the number of allowed templates. For each set of instructions we compute the corresponding template. In each iteration in the approximation algorithm the partition is slightly changed and a new optimal grain-constrained variable-length encoding is computed.

We built a compression engine for ARM object code. The resulting object code can be linked with ARM's runtime libraries to a compressed executable. We built a decoder simulator and integrated it with the core simulator from ARM. We used the SpecInt'92 and SpecInt'95 benchmark set and part of Philips' GSM code to measure results. A code size reduction of 41% is achieved. This is only 4 percent points less than the upper bound on the possible code size reduction with instruction recoding; the reduction achieved by a full Huffman encoding of all instructions. The measured performance penalty introduced by the decoder is 6% for a typical system with one memory wait state, and 18% worst-case with zero wait states.