x86 instruction reordering for code compression

Zsombor Paróczi

Runtime executable code compression is a special method, which uses standard data compression methods and binary machine code transformations to achieve smaller file size, yet maintaining the ability to execute the compressed file as a regular executable. In a typical case the source code gets compiled with a compiler and linker, the output is a binary executable, which contains the machine code and the data needed by the application. This work focuses only on executable machine code for the Intel x86 instruction set.

Many compression methods for x86 machine code have been developed, most of them use model based compression techniques (such as huffmann coding, arithmetical coding, dictionary-based methods, predication by partial match and context tree weighting), with CPU instruction specific transformations such as jump instruction modification. [1, 2] Compression of executable code is mainly used for lowering bandwidth usage on transfer and decreasing storage needs in embedded devices.

The compiled code can be split into so called basic blocks, sequences of instructions ending with a single control transfer instruction. The internal program representation used to facilitates our program transformations is the Control Flow Graph. This graph contains basic blocks as nodes, and potential control flow paths as edges.

In my work I show, that reordering instructions using data flow constraints can improve code compression, without changing the original behavior of the code. Using the data usage obtained from the analysis of the disassambled code, a basic, local data flow graph can be produced. A basic block instruction data flow is shown on figure 3. , each instruction is in a separated box, the original position is the code is the first line, the actual instruction is in the second line. The arrows represents data flow within the basic block, indirect dependencies are hidden.

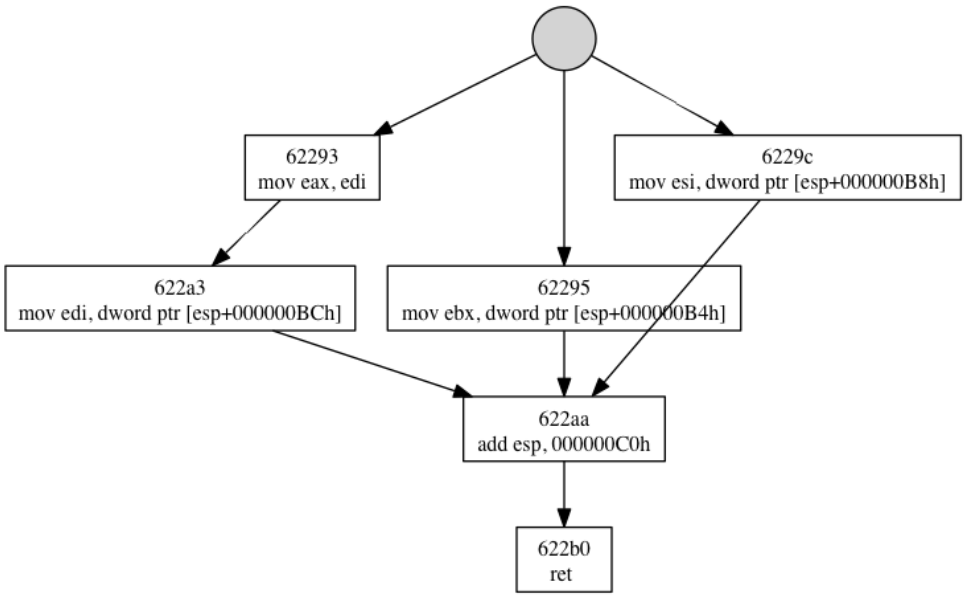

Figure 3: Data dependency graph in a basic block

The order of these instructions can be different from the original, and still the reordered code will be equivalent with the original (in the state machine sense).

In my work I distinguish two kind of data affection (read, write), and use 27 data types include registers: 16 basic x86 registers, 11 eflags, and memory data. The memory data refers to

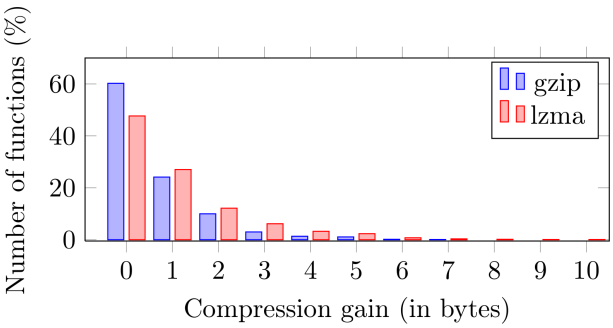

Figure 4: Compressed code size change

any kind of memory read/write including stack instructions. The generated control flow graph has a control flow instruction at the end of each basic block, these instructions are condition and unconditional jump and call instructions are treated as if they write every data type. Among x86 instructions only control flow instructions have relative to current address pointers, that is why reordering instructions can be done by simply changing the instruction's order. The control flow instructions always remain at the end of each basic block.

The complexity of a function is a product of all basic block instruction count within a function. The code generates all possible reordering using depth first search, dependency rules are checked on each search iteration. When all the possible reordering are generated, two different data compression methods are evaluated.

In my work I use the latest stable gzip and lzma software to compress the produced code. Repeating the reordering and compression test on every function of a binary file, a more compressible code can be produced, without modifying the data flow. Using gzip, the compressed code for an avarage function is 0.71% smaller than with the original instruction ordering. Results are even better using lzma, the avarage gain is 1.13%.

A detailed statistics on the compressed code size change can be seen on figure 4. The conclusion is that, more than 40% of functions can have a better compressable reordering than the original one.

References

[1] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223 - 267, September 2003.

[2] Wenrui Dai, Hongkai Xiong, and Li Song. On non-sequential context modeling with application to executable data compression. *In Data Compression Conference*, 2008. DCC 2008, pages 172 - 181, march 2008.