Code Compression *

Fang Yu
Department of Computer Science
University of California, Santa Barbara

June 18, 2006

Abstract

I investigate how *code compression* was achieved by using compressing ideas in information theory in previous researches. The main idea is to take the advantage on data compression to reduce the size of systems/programs. Code compression is a crucial technique in modern application design, especially for those applications highly restricted on device capability.

1 Introduction

We aim to reduce the size of programs via techniques in information theory. In recent years, it tends to be an increasing trend to incorporate computers to wide devices, such as palm-tops, electric appliances, embedded controllers, etc. In many of these devices, the amount of memory is limited due to considerations like space, weight, and power consumption. An application that requires more than that is available on a specific device will not be able to run on that device, no matter how sophisticated the software is. Hence, it seems to be attractive if we can remain the functionality but reduce the program size where possible. This project explores the use of data compression techniques to achieve code compression/compaction, in particular for two traditional techniques: dictionary compression and Lemple-Ziv [10] compression algorithm.

2 Code Compression

2.1 Data v.s. Codes

Before compression methods detailed, one question has to be answered: what's the difference between data and code compressions? A traditional technique used in data compression is replacing frequent words with short codewords; while, the technique used in code

*This is the term paper of the final project of CS225-Information Theory, Spring06

compression is directing similar code segments to identified copies. Both compressions are aim to represent information in the smallest form that still holds the information content. In some sense, we shall serve code compression as a variant of the more general problem of data compression. However, due to the essence of *data* and *codes*, there are some some fundamental differences which further affect algorithms developed.

Lefurgy [6] pointed three assumptions about the data being compressed in traditional compression against code compression. First, it is assumed that the compression must be done in a single sequential pass over the data since in most cases, typical data may be too large to contain in storage at one time. In contrast, programs are small enough to fit in storage, so the compressor can optimize the final compressed representation based on the entire program instead of using only information in a single pass. Second, the required single pass approach in data compression usually takes advantage of history of recent symbols in the data stream. History information allows compressors to utilize repetition in the data. However, this also constrains the decompressor to start at the beginning of the data stream. The decompressor cannot begin decompressing at an arbitrary point in the data stream since in that case it may not have sufficient history information that the decompression algorithm depends upon. For code compression, since a program shall be able to execute any path through its control flow and decompress as the program is executing, it is desirable to begin decompression at arbitrary points in the program. Third, since we target code compression in most microprocessors have alignment restrictions which impose a minimum size on instructions. For example, compressors may restrict their encodings to begin on byte boundaries so that the decompressors can quickly access codewords. This would require the use of pad bits to lengthen the minimum size of codewords. Based on these differences, the following question is how to efficiently identify similar patterns in codes?

2.2 Pattern Identification

Typically, it is hard to find large similar patterns repeated in a well-written program in high-level language. Code compression is usually applied to machine languages where a fixed instruction set are used repeatedly.

Dedray et al. [2] use control-flow graph to extract repeated code fragments. They induce *fingerprint* function to determine that whether basic blocks are identical (or similar). Since instructions usually differ from one another due to using different register names in their register fields, the fingerprint function only checks whether the first 16 instructions are the same despite what registers they use. However, this ignorance may result in incorrect compression/decompression. To address this problem, Debray et al. combine the technique of *register renaming* in their binary-rewriting tool. They find basic blocks with matching data-flow graphs and attempt to rename the registers within the basic

blocks so that the instructions match(the same instruction using the same registers). To achieve this, register move instructions are sometimes inserted before and after the basic blocks. An example of register renaming is given in Figure 1.

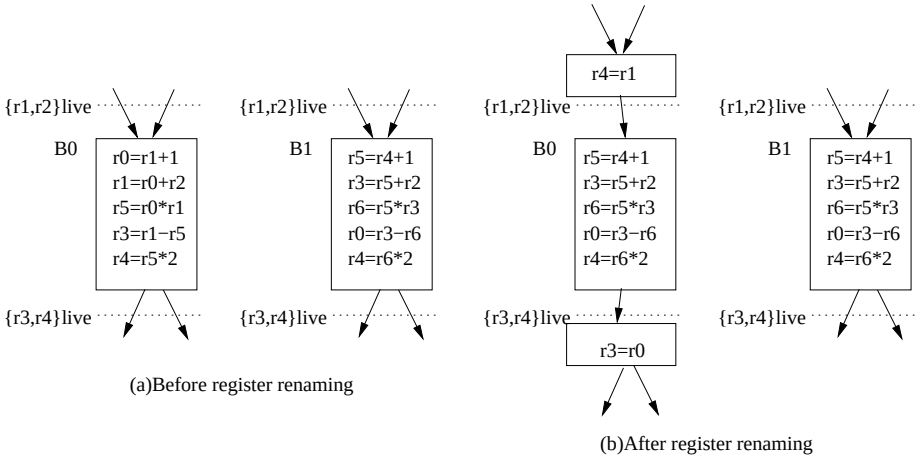

Figure 1: An example of register renaming

Furthermore, they look at the control-flow between basic blocks so that blocks can be combined into larger units of abstraction. If possible, identical blocks are moved into dominating, or post-dominating blocks to remove copies. Otherwise, the identical basic blocks are then used as candidates for procedure abstraction.

Register renaming not only can improve repetition in the program by renaming the registers of an instruction so that it matches another instruction in the program whenever possible but also maintain the correctness of renamed programs.

Cooper and McIntosh [1] also adopt register renaming to increase opportunities to apply procedure abstraction and cross-jumping. The primary difference is that they use live ranges, rather than basic blocks, as the unit of renaming registers. They search the entire executable binary for sequences of instructions (possibly spanning several basic blocks) that have similar data-flow and control-flow. They then attempt to make the sequences identical by renaming registers in the live ranges that flow through the sequences. Once the sequences are identical, then procedure abstraction or cross-jumping is applied.

2.3 Software v.s. Hardware

Another interesting issue is how to support code compression.

Liao et al [7] propose a software method for supporting compressed code. They find mini-subroutines which are common sequences of instructions in the program. Each instance of a mini-subroutine is removed from the program and replaced with a call instruction. The mini-subroutine is placed once in the text of the program and ends with a return instruction. Mini-subroutines are not constrained to basic blocks and may contain

branch instructions under restricted conditions. The prime advantage of this compression method is that it requires no hardware support. However, the subroutine call overhead will slow program execution. This method is similar to procedure abstraction at the level of native instructions, but without the use of procedure arguments.

A hardware modification is proposed to support code compression consisting primarily of a call-dictionary instruction [5, 9]. This instruction takes two arguments: location and length. Common instruction sequences in the program are saved in a dictionary, and the sequence is replaced in the program with the call-dictionary instruction. During execution, the processor jumps to the point in the dictionary indicated by location and executes length instructions before implicitly returning. The advantage of this method over the purely software approach is that it eliminates the return instruction from the mini-subroutine. However, it also limits the dictionary to sequences of instructions within basic blocks.

2.4 With or without decompression

So far for all proposed methods [1,2,4,6,7], the resulting compressed form can be executed without decompression. Another strategy is to apply data compression to executables on disk, then use decompression when the executable is loaded into ram to run [3]. Ernst et al. proposed a compressed *wire* representation that must be decompressed before execution but is, for example, roughly 21% the size of SPARC code when compressing. For this strategy, the techniques of data compression may be benefit directly to code compression; however, it may induce extra memory requirement and power consumption, which considerably suffers the application in particular for embedded systems.

3 Algorithms

In this section, we describe two most common algorithms used in code compression.

3.1 Dictionary

Dictionary compression uses a dictionary of common symbols to remove repetition in the program. A symbol could be a byte, an instruction field, a complete instruction, or a group of instructions. The dictionary contains all of the unique symbols in the program. Each symbol in the program is replaced with an index into the dictionary. If the index is shorter than the symbol it replaces, and the overhead of the dictionary is not large, compression will be realized.

The dictionary compression takes advantage of the observation that the instructions in programs are highly repetitive. The compression method finds sequences of instructions

(some of length one) that are frequently repeated throughout a single program and replaces the entire sequence with a single codeword. All rewritten (or encoded) sequences of instructions are kept in a dictionary which, in turn, is used at program execution time to expand the singleton codewords in the instruction stream back into the original sequence of instructions. Codewords assigned by the compression algorithm are indices into the instruction dictionary.

The following equation and lemma specify the condition to achieve *efficient* dictionary compression. We say a compression is efficient if its compression rate is greater than 1, i.e., the storage size of the original programs is greater than the storage size of the compressed ones.

$$nw \geq n\lceil \lg d \rceil + dw \tag{1}$$

In this equation, n is the number of static instructions in the program, w is the number of bits in a single instruction, and d is the number of symbols in the dictionary.

LEMMA 1 *The dictionary compression rate is greater than 1 if and only if Equation 1 holds.*

The lemma holds by definition. Note that the condition may be more restricted if we account for the size of specific implementations of the decompressor, which we ignore this part here for simplicity.

In [6], each unique 32-bit instruction word in the original program is put in a dictionary. Each instruction in the original program is then replaced with a 16-bit index into the dictionary. Because the instruction words are replaced with a short index and because the dictionary overhead is usually small compared to the program size, the compressed version is smaller than the original. Instructions that only appear once in the program are problematic. The index plus the original instruction in the dictionary are larger than the single original instruction, causing a slight expansion from the native representation. An example from [6] is given in Figure 2 to illustrate the compression method.

3.2 LZ algorithm

In most programs there are common sequences of instructions that appears in the different sections of the code. Recall the Lemple-Ziv(LZ) algorithm in which we use pointers to identify repeated words. Fraser [4] and Lau et al. [9] induced *echo* instruction as a *pointer* to reduce repeated sections of code to a single copy. The idea behind echo instruction is to compress these repeating sequences of instructions by "echoing" existing code sequences. More precisely, all the other sections are replaced with a single echo instruction that tells the processor to execute a subset of the instructions from the single copy. The echo instruction provides a way to represent pointers in the program.

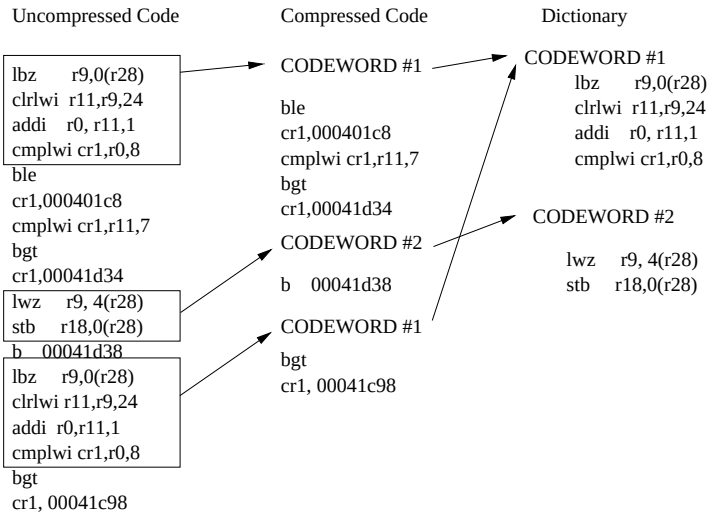

Figure 2: An example of dictionary compression [6].

Fraser proposed the basic echo instruction, called sequential echo, in LZ77 compression [4]. The sequential echo, as you will see, tells the fetch unit where the duplicate instructions can be found and how many duplicate instructions needed to be executed. LZ77 compression accepts a stream of characters and produces a stream that interleaves literals and pointers(echo instructions). Each echo indicates a phrase in the previous N characters and has two parts: a displacement and a length. The displacement gives the distance back to the phrase, and the length identifies the number of characters in the phrase. For example, the byte string `Blah blah.` compresses to `Blah b5,3.` where the underlined material denotes an echo instruction which indicates the phrase `lah`. The displacement is five, and the length is three, because the next three bytes repeat those back five bytes. Fraser also adds LZ77 echo to conventional instruction sets. The assembler instruction `echo .-5,3` commands the (hardware or software) interpreter to fetch and execute three instructions starting five bytes back from the echo instruction.

In this way, sequential echo instructions are like lightweight procedure calls: they cause the processor jump to the target location, execute the desired code sequence, and return to the call site. Note that each echo indicates a fixed number of instructions that will be executed, and hence no return instructions required.

Lau et al. [9] forward the *basic echo* instruction to handle two code sections in a program which are very similar but not exactly identical, differing by a small number of instructions. To compress similar section codes, they extend the echo instruction by allowing it to conditionally include instructions based on a bitmask. Each bitmask echo has two fields: a bitmask and a branch offset. Each bit in the bitmask corresponds to an instruction at the branch target: a one bit indicates that the corresponding instruction is executed, and a zero bit indicates that the corresponding instruction is not executed. By

replacing sequences of code with echo instructions that refer to similar code elsewhere in the program, they achieve a compression ratio of 85%.

4 Conclusion

The main contribution of this project is to summarize the recent issues and researches of code compression. It is interesting to connect dots between data compression and code compression, since both try to find an efficient form to represent information. (I believe this is exactly the key value of Information theory.)

On the other hand, since the later is an essential technique for embedded systems, extending techniques in information theory to this field is an attractive research direction.

References

[1] K. D. Cooper and N. McIntosh, Enhanced code compression for embedded RISC processors. In Proceedings: the ACM SIGPLAN 1999 conference on Programming language design and implementation, pp. 139-149, Atlanta, Georgia, United States, 1999.

[2] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter, Compiler Techniques for Code Compaction, ACM Transactions on Programming Languages and Systems (TOPLAS), vol 22, No. 2, pages 378V415, 2000.

[3] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation, pages 358-365, Las Vegas, NV, June 1997.

[4] C. Fraser, An instruction for direct interpretation of LZ77-compressed programs. Microsoft Technical Report, MSR-TR-2002-90.

[5] M. Game and A. Booker, CodePack code Compression for PowerPc processors, Technical Report, International Business Machines, Research Triangle Park, NC, 1998.

[6] C. Lefurgy, Efficient Execution of Compressed Programs, Doctoral Dissertation, Dept. of CS and Eng., University of Michigan, 2000.

[7] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. TODAES 4(1), pages 12-38, 1999.

[8] C. Lefurgy, E. Piccininni, and T. Mudge, Reducing code size with run-time decompression, In Proceedings of the 6th International Symposium on High-Performance Computer Architecture(HPCA), pages 218-227, Jan 2000.

[9] J. Lau, S. Schoenmackers, T. Sherwood, B. Calder. Reducing code size with echo instructions. In Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, , pages 84-94, San Jose, California, USA, 2003.

[10] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, pages 337-342, 1977.