A Code Compression Advisory Tool for Embedded Processors

Sreejith Menon
Priti Shankar
Department of Computer Science and Automation
Indian Institute of Science
Bangalore

Abstract

We present a tool which is designed to be used as a code compression advisory system for object code to be run on an embedded processor. All the compression schemes support run-time random decompression. Given the machine instruction set architecture, the encoding of instructions, and a set of object programs to be compressed, the tool analyzes the code, gathers statistics about static instruction frequencies and other relevant information, and performs a relative evaluation of a suite of compression strategies. The tool produces as output, the sizes of the compressed code, the Line Address Table(if one is required), and the dictionary(if there is only one) or the sizes of all dictionaries if there are several, for various choices of parameters input by the user. We have used the tool to evaluate alternate schemes for a suite of benchmarks for the TI TMS320C62x instruction set architecture and the Intel ARM processor and report results.

keywords code compression, embedded system tool, run time decompression.

1 Introduction

In an embedded system, perhaps the most restricted resource is the memory. The memory area in a silicon chip can take up from three to seven times the area that is required for the processor. Consequently, a decrease in program size effectively reduces the cost, size and power consumption. A decrease in the program size also reduces the instruction cache misses and hence provides higher instruction bandwidth[11]. There are two major approaches to the reduction in code size. The first uses techniques to effectively reduce code size at compile time[4]. The second approach compresses small blocks of instructions with decompression being performed dynamically during program execution[1, 2, 3, 5]. The decompression overhead thus contributes to execution time. The schemes we use here employ a strategy of the second kind. There are two important factors that affect the choice of a compression scheme of this kind. Firstly, random access to compressed blocks of code must be ensured. Since compressed code is decompressed during program execution, jump, branch and call statements that alter the flow of control must be directed to small compressed blocks within which the target instructions are located without having to decompress the whole segment. Some branch targets cannot be determined at compile time. Therefore fixed to variable length coding schemes need to use a Line Address Table(LAT)[1] which maps targets of branches in the original code to targets in the compressed code. Secondly the decompression scheme must be very fast, as it is on-line and its overhead directly affects program execution time. Compression, however, can be done off-line and hence one can exploit complex strategies to generate good models or dictionaries to be used in the decompression stage.

Several new architectures have flexible instruction formats. A scheme that is efficient for fixed format instruction set architectures may not be suitable for flexible instruction formats as repeated patterns in the instruction that allow for good compression may be more difficult to find in flexible format instructions such as that of the TI TMS320C62x[9]. A fixed length coding scheme such as the one proposed by Lefurgy and Mudge[3] performs very well on fixed format instruction set architectures and does not require a LAT; however on an ISA like that of the TMS320C6x its performance is poor[6]. A feature that affects runtime performance is the degradation of running time due to the overheads of decompression. If a simulator is available for the architecture, one can study the benefits of using profile information to decide which instructions should be compressed and which should be retained in uncompressed form because of their high frequency of execution. Our aim in this work is to provide a frame-

work within which a user can specify inputs regarding the instruction set architecture, the size of blocks for which dictionaries should be built, the upper bound on the size of the dictionary, the suite of benchmarks for which the compressor should be optimized, and profile information if it is available. The tool does a thorough analysis of the code, examines fixed and variable dictionary sizes for the dictionary based schemes, and computes probability tables for the arithmetic coding schemes. The output of the system is a set of options available to the user and the performance of each option on the benchmarks supplied. Typical measures of performance include the compression ratio, the sizes of the dictionary and LAT for various options supplied by the user. In the next section we describe the compression schemes tested by the tool.

2 Compression Schemes Tested by the Tool

The various code compression strategies tested by our tool are outlined below. Most of these schemes are based on dictionary techniques, as a dictionary based scheme results in minimal run time decompression overhead. However, a scheme based on arithmetic coding is also included in the package for the sake of performance comparison.

1. *Fixed size dictionary schemes.* The concept of fixed dictionary was originally proposed by Lefurgy and Mudge[3] and has been implemented on Sharc architecture. In their scheme, each unique instruction word in the program is inserted into an instruction table, replacing each instruction in the program with an index into the table. If the table overhead is small compared with the program size, the compression is effective. An advantage of this scheme is that PC relative branches do not change. Also, absolute branch addresses will change by an amount that can be precomputed because of the fixed length encoding, thereby avoiding the overhead of the LAT. Its main drawback is that the performance on VLIW architectures with flexible formats is poor. We have also included a variant of the scheme which inserts only frequently occurring instructions into the dictionary, and leaving the rest in uncompressed form. This scheme needs a LAT to support branches as it is a fixed to variable length scheme.

2. *A multiple dictionary scheme.* This has been proposed in [7] and [8]. This is a fixed-to-variable scheme and therefore requires the use of a LAT for addressing. Instructions are grouped into classes and each class has a separate dictionary. Compression results are improved by dividing the instruction classes into two segments at some logical point, and constructing separate dictionaries for the different instruction segments. Instructions are encoded as pointers to the respective dictionaries. This performs better than the fixed dictionary scheme on machines with instruction set architectures with a variable format at the cost of slightly more complex decompression hardware. We have implemented the fixed multiple dictionary scheme to reduce the decompression complexity.

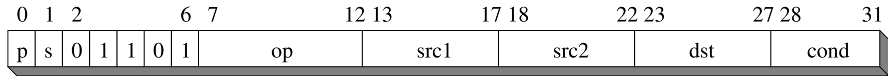
3. *A scheme based on arithmetic coding.* This was proposed by Lekatsas and Wolfe[2] and uses a simplified form of arithmetic coding to increase decompression speed. It resolves jumps using a Line Address Table, but has the disadvantage of complex decompression hardware. We have implemented the scheme using a binary tree model for computing probabilities. The compression phase requires an initial pass to compute probabilities and to build the encoding tables, which are used in the second pass to compress the code.

4. *A scheme based on Hamming distances.* This was proposed by Prakash et al.[5] and with details of the implementation reported in [6]. The basic idea is to find a set of instructions that form the dictionary and which are selected such that most other instructions are at a small Hamming distance from any one of the instructions in the dictionary. (The Hamming distance between two vectors is the number of positions in which they differ). This is also a fixed to variable scheme and requires a little more hardware than simple dictionary schemes. Our implementation uses clustering algorithms to obtain the best dictionary.

3 The Advisory Tool

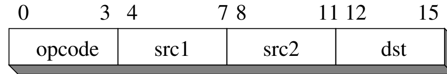
The package may be run with just a set of benchmark programs, if the user does not wish to give detailed information about the processor for which the tool is to be used. In such a case an ISA oblivious scheme will be used.

ISA conscious schemes expect details of the instruction format of the processor as the input. A simple language illustrated in Figure 1 can be used to supply the instruction set format. The tool automatically constructs dictionaries and probability tables and evaluates the first set of compression results.



instr_class --> ps (0:1), const_13(2:6), op(7:12), src1(13:17), src2(18:22), dst(23:27), cond(28:31)

instr_identify --> const_13(2:6)



instr_class --> opcode(0:3), src1(4:7), src2(8:11), dst(12:15)

instr_identify --> const_4(0:3) | const_7(0:3) | const_8(0:3); assuming opcodes 4,7 and 8 are in this class

Figure 1: An example language to denote the instruction formats

Initial results output by the tool essentially consist of a report describing different code compression schemes tested, along with detailed compression results for each scheme. The report also describes the decompression strategies, the hardware complexity of the scheme and specifies the default parameters (which includes the block size for compression, standard dictionary sizes etc.) of the compressor. Depending on the preliminary results supplied by the tool, the user can study the effects of varying different parameters such as block size, dictionary size and various combinations thereof. Average performance for a suite of benchmarks can be obtained.

The tool can support only a fixed set of standard binary formats (such as coff, a.out, elf etc.). If the benchmark programs are not in the supported formats, then the user is expected to supply the raw instructions to the tool along with the byte order information(big/little endian).

Some additional features are provided in the tool which take into account the dynamic behavior of benchmark programs. One such example is execution profile based compression. Instruction blocks that are frequently executed can be left uncompressed to reduce the decompression overhead. Profiling information can be supplied to the tool (in a prescribed format) to study the effects on the compression ratio. An example is shown in the next section to explain the process.

3.1 User Interface

The interface of the tool is simple. Initial iterations require only the set of programs and the instruction word size. The parameters that can be manipulated for various schemes are specified in the report supplied to the user and subsequently, if the user wishes to exercise options, this is made possible with appropriate prompts. A simple language (as shown in Figure 1) is provided to input the instruction set architecture for experimenting with ISA conscious schemes. The tool also supports additional features for the expert user. A separate mode is provided in which a user can view the contents of the internal data structures of the schemes including the dictionary entries, the instruction frequency counts and the probability tables.

3.2 Output Format

In addition to the report which includes a description of the decompressor, the tool reports its results in the form of tables for LAT sizes, dictionaries and probabilities and graphs for compression ratios of each program in the suite for each coding scheme. The user can also view the effects on the compression ratio for a range of parameter values in the form of graphs.

4 Experiments Conducted and Results

We have experimented with our tool on the TMS320C62x platform and Intel StrongARM platform using a set of Mediabench programs[10]. The TMS320C62x processor is representative of a flexible instruction format VLIW architecture and the Intel StrongARM represents a low-cost, low-power RISC architecture.

The initial compression results of the two processors are shown in Figure 2 and Figure 3 respectively. The size of the LAT and the dictionary sizes are not shown separately because of the space limitations and the compression results of the figure include the overheads of the LAT and that of the dictionaries.

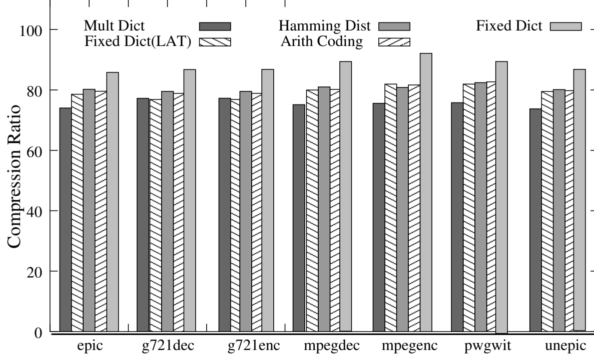


Figure 2: Compression ratios for TMS320C62x

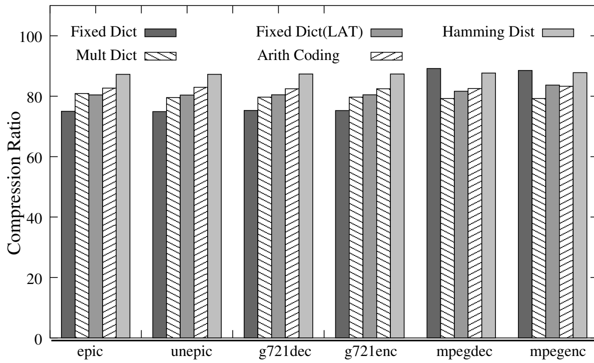


Figure 3: Compression ratios for Intel StrongARM

We have selected the TMS320C62x processor for further iterations. Table 1 shows the average compression ratios of the different code compression schemes on the TI processor. Multiple dictionary scheme produces better results with the default parameters on the TMS320C62x processor. The fixed size dictionary scheme, which gives the next best results can be studied thoroughly for the various combinations of the parameters of the scheme. Figure 4 shows the result of varying the dictionary size of the fixed dictionary scheme in the range of 1 kb - 256 kb. Compression ratio as low as 78% is obtained for the given range

Code compression scheme	Comp ratio
Multiple dict scheme	75.51%
Fixed dict scheme(using LAT)	79.51%
Arithmetic coding scheme	80.21%
Hamming distance scheme	80.5%
Fixed dict scheme(without LAT)	88.17%

Table 1: Average compression ratios for the different compresion schemes

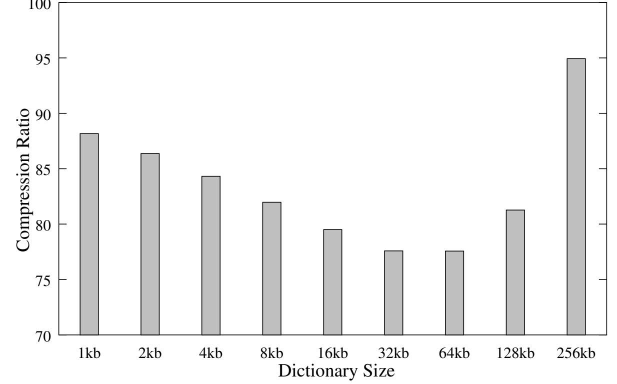


Figure 4: Effect of varying dictionary sizes in the Fixed dictionary scheme

Program	Code size	Dict size	LAT size
epic	89.79%	3.33%	6.86%
g721dec	87.26%	6.13%	6.59%
g721enc	87.26%	6.14%	6.59%
mpgdec	90.17%	3.05%	6.77%
mpgenc	91.32%	1.94%	6.72%
pegwit	90.70%	2.59%	6.70%
unepic	89.94%	3.16%	6.89%

Table 2: Division of compressed code into code, dictionary and LAT for Multiple dictionary method

of dictionary. In a similar manner, other parameters of the different compression schemes can be varied to study the effect of the compression ratios.

The user can fix a compression scheme based on factors like hardware complexity, decompression stages etc., which are provided with the initial report produced by the tool. Table 2 shows a detailed analysis of the Multiple dictionary scheme in which the compression results are separated into code size, dictionary size and the size of the LAT. Further analysis is made in Figure 5, which shows the individual dictionary sizes of the different instruction classes. Each instruction class is divided into two segments by the code compression scheme resulting in two separate dictionaries/class. User can finally decide on a possible combination of the parameters for the selected code compression scheme.

The final scheme can be studied in more detail by exploring the effect of dynamic behavior of the programs using profile information. Figure 6 shows the effect of including profile information in the compression results. The percentage of uncompressed code and the corresponding compression ratio is shown in the figure.

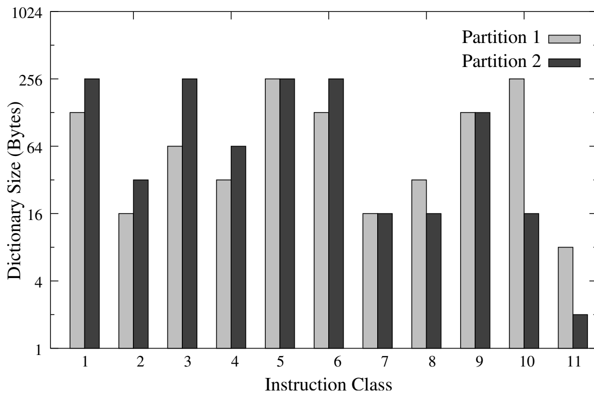


Figure 5: Dictionary sizes of the Multiple dictionary scheme

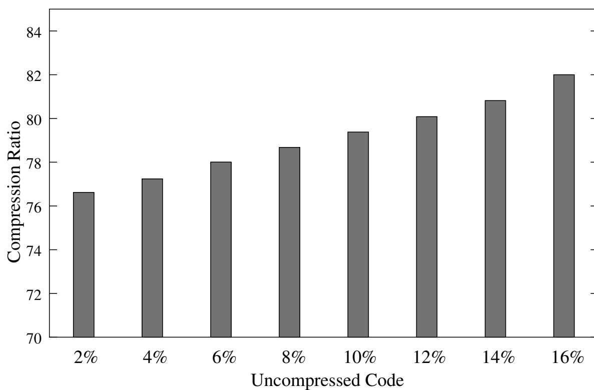


Figure 6: Compression ratio variations with the inclusion of profile information for Multiple dictionary method

5 Conclusion

We have presented a new advisory tool for embedded processors which helps to select a code compression scheme for an arbitrary processor. The working of the tool along with the interface has been explained and was tested for TMS320C62 processor and the Intel StrongARM processor. The advisory tool is expected to be useful for the naive user as well as the expert user because of the different features included in the tool.

References

[1] A.Wolfe and A.Chanin. Executing Compressed Programs on an Embedded RISC Architecture, *Proc.* 25th *Ann. International Symposium on Microarchitecture,p81-91,1992*

[2] Lekatsas and Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE transactions on CAD, pp.1689-1701,December,1999.*

[3] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, Improving Code Density Using Compression Techniques,*Proceedings of the* 30th *Annual International Symposium on Microarchitecture, pp. 194-203, December 1997.*

[4] S. Debray l.William Evans, and Robert Muth. Compiler Techniques for Code Compression, *Workshop on Compiler Support for System Software, 1999.*

[5] Prakash J, Sandeep C, Priti Shankar, Y N Srikant.A Simple and Fast Scheme for Code Compression of VLIW Processors *Data Compression Conference, 2003.*

[6] J. Prakash, C. Sandeep, Priti Shankar, Y N Srikant A Simple and Fast Scheme for Code Compression for Embedded VLIW Processors *Master of Engineering Thesis, CSA-IISc, 2003.*

[7] Montserrat Ros and Peter Sutton. Code Compression Based on Operand-Factorization for VLIW Processors. *Proceedings of the Data Compression Conference (DCC 2004)*

[8] Sreejith Menon and Priti. Shankar Space/time tradeoffs in code compression for the TMS320C6x processor*Technical report IISc-CSA-TR-2004-4 July 2004.*

[9] *TMS320C62xx CPU and Instruction Set: Reference Guide*,Texas Instruments,Jan.1997.

[10] The MediaBench *http://www.cs.ucla.edu/leec/mediabench/.*

[11] I. Chen. Enhancing Instruction Fetching Mechanism using Data Compression,*Ph.D. Dissertation,University of Michigan,1997.*