Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor

Sreejith K Menon
Priti Shankar
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012, India
email: {sreejith, priti}@csa.iisc.ernet.in

IISc-CSA-TR-2004-4
http://archive.csa.iisc.ernet.in/TR/2004/4/

Computer Science and Automation
Indian Institute of Science, India

July 2004

Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor

Sreejith K Menon
Priti Shankar
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012, India
email: {sreejith, priti}@csa.iisc.ernet.in

September 3, 2004

Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor

Abstract

Reducing instruction memory requirements by improving code density using compression techniques has been the aim of much recent work on embedded devices. Previous work has been successful in improving compression ratios with modest decompression overhead for general purpose RISC architectures. However, most traditional compression techniques fail to produce good results for tightly encoded VLIW architectures. Increased popularity of highly flexible VLIW instruction formats have triggered a search for new variants of traditional compression schemes which achieve good compression ratios with low decompression overhead. We propose a simple variant of a dictionary based compression scheme and report the results of simulations on a widely used VLIW architecture, the TI TMS320C62x, exploring various options like field sizes, use of profiling information, and study their effects on compression ratios and decompression overheads. The advantage of our scheme is its simplicity and its easy adaptability to varying instruction formats.

Keywords: Compression, Code density, VLIW architecture

1 Introduction

Embedded devices are supposed to play a key role in the future generation of miniature devices [1, 2]. However, at the present time, memory is found to be a scarce resource. Most of these small devices have constraints including die size, cost and power consumption. Since memory size is directly related to the total die size of the embedded devices, special attention has to be given to the size of application programs developed for these devices. On the other hand, sizes of programs are increasing as automatically generated programs are larger than the handwritten assembly code that they replace. To reduce the total code development cost, highly optimized handwritten assembly code is limited to the core part of the program space. The past decade has witnessed considerable research in the area of object code compression to address the issue of limited instruction memory[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 23]. All such compression techniques have to deliver good compression ratios within the constraints of real time decompression and random instruction access with as little additional hardware as possible. Many proposed techniques produce good compression ratios [1] on RISC architectures. However with the introduction of new flexible VLIW instruction formats, traditional compression techniques are found to be inadequate. One such architecture that has multiple functional units and variable instruction formats is the TI TMS320C62x[1]. Statistical schemes that use a variant of arithmetic coding[9, 10, 11] have been proposed as a solution to improve the compression ratio, but at the cost of complex decompression hardware.

Fast decompression with simple hardware can be obtained with dictionary compression schemes[4, 5, 8, 12]. However these do not work well for variable instruction formats. In this paper, we propose a simple variant of a dictionary based compression scheme that overcomes this difficulty, and look at spacetime tradeoffs on TMS320C62 processor. We divide the instruction set into different classes depending on the instruction set architecture, and create separate dictionaries for each class thereby exploiting similarities among instructions in a single class. Average compression ratios range from 73.5% upwards(inclusive of the space for the line address table) depending on the space/time tradeoff.

Section 2 gives an account of related work in the field. The compression scheme is explained in detail in Section 3. Experiments and results are described in Section 4 and are compared with those obtained from current schemes. Finally Section 5 concludes the paper.

2 Related Work

Wolfe and Chanin[3] proposed the first code compression scheme for embedded processors. Here the main memory contains the compressed code, and the decompression unit is located between the main memory and the instruction cache. This decompresses a cache line whenever there is a cache miss. Byte based Huffman codes are used for compression. Compression ratios of 73% are reported by Kozuch and Wolfe[16] for MIPS code. Benes et al.[17] have designed a fast Huffman decompressor chip which achieves an output rate of 163 Mbytes per second. Wolfe and Chanin also proposed the use of a Line Address Table(LAT) which maps

[1] Compression Ratio = Compressed Size / Original Size

program instruction block addresses into compressed code instruction block addresses. The data in the LAT is generated by the compression tool and stored along with the program.

The idea of replacing frequently appearing groups of instructions by a call to a dictionary which stores each of these sequences once was proposed by Liao et al.[4]. They have reported compression ratios of 82% to 84% for TMS320C25.

Bird and Mudge[5] proposed a simple dictionary-based method which outperforms byte-based Huffman coding. Commonly occurring instruction sequences are replaced by a codeword which is an index into a dictionary which has the original sequence. The final program has both codewords and uncompressed instructions. Branch targets have to be aligned in this scheme and the range of branches is reduced. They report compression ratios of 61%,66% and 74% for the PowerPC, ARM and I386 processors respectively.

Lefurgy and Mudge[12] introduced the use of a fixed length encoding by putting each unique instruction word in the program in an instruction table, and 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.

Ishiura et al.[7] split up VLIW instruction words into fields each of which is compressed using a dictionary based scheme. Their scheme is for fixed instruction formats and compression ratios from 46 to 60 % are reported. Nam et al. [8] also propose dictionary based compression for a fixed format VLIW processor and report compression ratios from 63% to 71%.

Lekatsas and Wolf[9] propose a statistical scheme based on arithmetic coding along with a precomputed Markov model, with multi-bit decompression for use in RISC architectures and have reported compression ratios as low as 50% for a RISC processor. An arithmetic coding scheme for VLIW processors with flexible instruction formats is also described in the work of Xie et al.[14]. Here there is a tradeoff between the speed of decompression and the amount of compression, and the compression ratio for the TMS320C6x that they achieve ranges from 67% to 80% with decompression speeds from 11 bits to 47 bits per clock cycle. However it is not clear if the LAT size is included in these figures.

Prakash et al.[13] use a variation of the fixed size dictionary scheme in which each instruction is separated into two halves of 16 bits. Dictionaries are created for the vectors in each half-space, such that most of the vectors in the half space are at a Hamming distance of one from some entry in the dictionary. Each instruction half is encoded by a pair which is an index into the dictionary and the position where it differs(if at all) from the dictionary entry. A line address table is used to map compressed instructions to the original ones. Average compression ratios of 76% and 78% (including the LAT) were obtained for TI and Mediabench benchmarks by their method, henceforth referred to as the Bit-Flip scheme.

Ros and Sutton[18] have tried out single and multiple instruction dictionary methods for code compression and investigate the performance when applied to various compiler optimizations and parallel instruction orderings.

3 A Dictionary Scheme Based on Variable Instruction Formats

Based on the instruction set architecture one can design a multiple dictionary scheme which is a variation of the fixed size dictionary scheme of Lefurgy and Mudge[12]. Here instructions are separated into different groups based on the classes of the instruction set architecture. Separate tables are created for each of these groups and unique entries are inserted into the table. Instructions are encoded as pointers to the respective entries. Since the tables created are generally of small size, the overhead of storing pointers is small.

3.1 The New Scheme

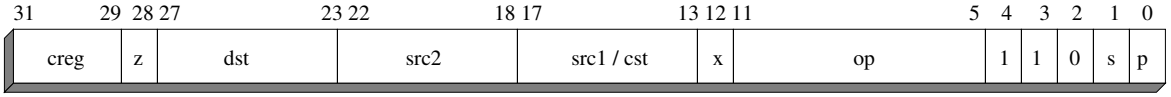

Figure 1: Format of .L instruction class

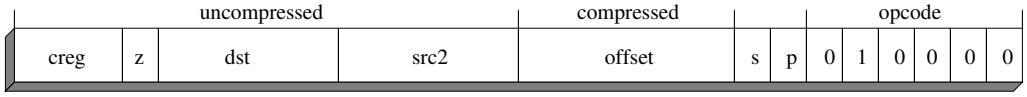

Figure 2: One possible encoding of .L class

Modern VLIW architectures like the TMS320C62x use a flexible instruction format. In our code compression scheme, instructions are grouped at the object code level based on instruction classes. For example, the TMS320C62 architecture specifies twelve instruction classes and therefore object code instructions are partitioned into 12 groups. (In the implementation, we have dealt with only 11 classes as 2 instruction classes, IDLE and NOP have been merged into a single instruction). Within a group, each instruction is partitioned into two segments, division being performed at a logical point, with each instruction class having a different division point in general. In our scheme for the TI processor, we ensure that each segment is contained in two bytes. This size limit simplifies the implementation of the index tables and the decompression hardware.

Consider an instruction class (.L functional unit) of TMS320C62 architecture in Figure 1. The partitioning is made at the 17th bit. So op;x;src1 forms one segment and src2;dst;z;creg forms the second. The three bit opcode of the instruction class, 011, can be ignored because it can be automatically generated at the time of decompression. All other fields of the instruction are relevant for compression. Instruction class division and the instruction partitioning are fixed and hence the decompression hardware can be fixed for a given processor.

3.2 Dictionary Construction

A dictionary for each segment of each class is constructed for a block of object code. In our scheme, there are 11 instruction classes which are divided into two segments needing a total of 22 dictionaries.

An initial pass is made over the whole program to collect all unique entries into their respective dictionaries, associating each entry with a frequency count. Infrequently occurring instruction segments are removed from the dictionary and left uncompressed. If d is the total size of a dictionary corresponding to an instruction segment of size s, then pointers of size $log_2(d)$ will have to be used. If the frequency of the instruction segment is n in the uncompressed program, then an instruction is compressed whenever
$(s + n * log_2(d) < (n * s)$, ignoring the opcode.
A problem with the above dictionary construction method is that the size of the dictionaries varies with the programs to be compressed. It complicates decoding of the instruction segments and therefore its implementation. A possible alternative is to fix the size of the dictionary with the help of a set of common benchmark programs which are frequently run on the processor. A disadvantage of fixing the dictionary size is that some small programs can result in wastage of dictionary space, whereas most large programs would benefit from additional dictionary space. The compression ratios are slightly affected because of these differences, but our experiments have revealed that the fixing the dictionary size results in better performance with a small degradation in compression ratio.

The second difficulty associated with logical division points is the nonuniformity of the size of instruction segments. Dictionary entries can be padded to result in uniform size. In our scheme, all dictionary entries are padded to form two bytes. The overall reduction in compression ratio by forcing this uniformity is less than 0.5% on the average; the advantage lies in easy decoding thereby achieving fast dictionary access. In effect, multiple dictionaries can be logically considered to be a single large dictionary with different base addresses for various instruction segments.

3.3 Encoding

Encoding is simple as each instruction segment is replaced by a pointer to the corresponding entry in the dictionary. If the instruction segments are not present in the dictionary then they are left in uncompressed form. We use an *opcode* for the instructions in the compressed program, which gives information regarding the class of the instruction, the base of the dictionary, and whether each instruction segment is compressed or not.

The TMS320C6200, our experimental processor, is part of the TMS320 DSP family[1]. It uses the VelociTI architecture which is a high performance advanced VLIW architecture. It is a clustered architecture processor and has two clusters of four functional units each. It fetches eight instructions each of length 32 bits at a time. Each such *fetch* packet may be made up of several *execute* packets, all instructions in an execute packet being executed in parallel. Since each fetch packet is made of eight instructions we selected our block size for purposes of compression to be eight instructions i.e. 256 bits or 32 bytes. In other words a decompression operation decompresses a single fetch packet.

Our opcode requires 6 bits (4 bits to indicate the class of the instruction and 2 bits to determine whether each of the segments is compressed or not). If the first segment of an instruction belonging to the class in Figure 1 (.L class) is compressed and the second segment is not, it will be encoded as in Figure 2. The first 4 bits indicate that the instruction is of class zero. The next two bits indicate that the first segment is compressed and the second segment is uncompressed. The compressed segment is replaced with the pointer to the

dictionary entry and the uncompressed segment appears as such.

Since the processor handles eight instructions (a fetch packet) together, a fetch packet has to be addressable. Our variable length encoding scheme does not guarantee that each fetch packet starts at a new byte boundary. Therefore fetch packets have to be made explicitly addressable by padding the last few bits of the last instruction in a fetch packet with zeroes. This reduces the compression by a small amount (around 1.5%).

3.4 Line Address Table

As mentioned earlier, random access of instructions is a constraint to be addressed in object code compression techniques. A Line Address Table (LAT) helps in maintaining a mapping between the compressed and the original instruction addresses, so that control instructions like indirect jumps, whose targets are determined only at run time can be effectively handled. The space overheads in maintaining this line access table are seen to be about 5%.

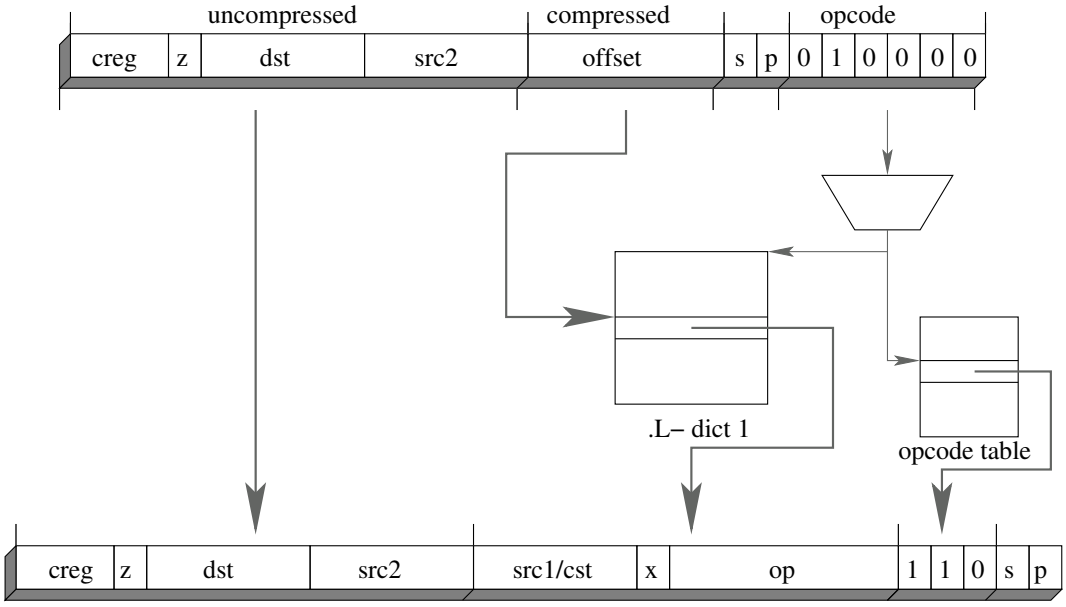

Figure 3: Serial Decompression

3.5 Serial Decompression

The opcode which is the first part of a compressed instruction holds three pieces of information, namely the class of the instruction, whether the instruction is compressed or not, and the base address of the appropriate dictionary.

Figure 3 shows the decoding steps of the encoded instruction of Figure 2. The instruction class and the base address of the dictionaries can be derived from the opcode. The dictionary entry can be obtained with the offset provided in the compressed instruction. An uncompressed segment does not need any processing. The original opcode of the instruction class can be automatically generated from the opcode as shown in the figure.

Since instruction segments of all the dictionaries have uniform size with appropriate padding, the dictionary entries may have to be shifted depending upon the opcode value to get the proper segment.

The serial decompression scheme can be pipelined in three stages. In the first stage, the opcode can be decoded to get the basic information on the dictionaries and the type of the instruction. At the end of the first stage, number of bits taken by the first compressed instruction is known and the next compressed instruction opcode can be identified and decoded. The second stage is used to send the offset to the corresponding dictionary and the original instruction can be retrieved in the third stage. The original opcode of the instruction class can be generated in parallel to form the complete instruction.

3.6 Parallel Decompression

A serial decompression scheme used as such may degrade performance considerably as the instructions in a fetch packet are decoded serially. One way to overcome this is to have a more expensive parallel decompression unit as described below.

The compressed instruction in the Figure 2 can be divided into two sections - the opcode partition (initial 6 bits) and the instruction segments (the remaining bits). Therefore all the compressed instructions of a fetch packet can be organized into these two regions. The first region contains the opcodes of all the eight instructions in the fetch packet and the second region contains the remaining segments of all the eight compressed instructions of the fetch packet. Since the size of the opcode is fixed, the opcode region will take a fixed size partition of the compressed fetch packet. Each opcode can be accommodated in a single byte for the TI processor so that the corresponding opcode can be accessed easily with a simple shift operation. All the instructions of a fetch packet can be decoded simultaneously once the complete mapping of the fetch packet is known. By contrast, in the serial decoding scheme, the starting address of the second instruction is only known after the decoding of the first one. During parallel decompression, all the eight opcodes residing in the first eight bytes of the compressed fetch packet can be decoded together to obtain the complete mapping of the whole fetch packet. Once the starting address and status(compressed or uncompressed) of all the instructions of the fetch packet are known, they can be fetched in parallel. However, the simple hardware of the serial decompression scheme can not be directly used because of the need for parallel access of the dictionaries.

3.7 Implementation of the Dictionary

The dictionary can be implemented as a single large index table with a fixed number of entries. Our compression scheme makes use of non-uniform size instruction segments which have to be shifted appropriately based on the opcode. Therefore the dictionary entries have to be fed to a bit shifter controlled by the opcode of the compressed instruction to form the correct number of bits for the original instruction. Equal sized dictionary entries help in fast and uniform access of segments resulting in simple implementation.

3.8 Using Profile Information

To achieve better performance we examined the use of profiling information in order to guide the compression process. The scheme described so far is a purely static scheme which does not take into account the run-time behavior of the program. In most cases it is observed that the program executes a small number of instructions very frequently. If these are left uncompressed it is likely that the loss in compression ratio is offset by a significant improvement in the overall running time. Our simulations have shown that this is indeed the case and we report the results in the next section.

4 Experiments and Results

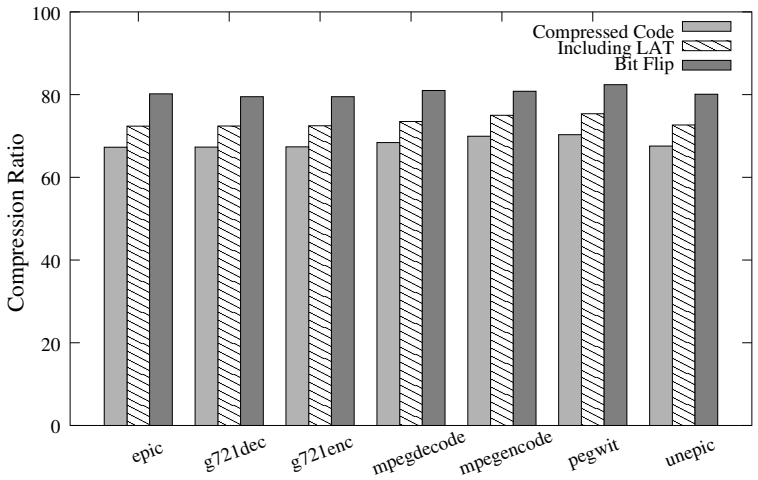

Figure 4: Compression ratios for Mediabench programs (variable dictionary entries)

We have experimented with our compression scheme on the TMS320C62 platform using Media[19] and TI[20] benchmarks. Texas Instruments Code Composer Studio (version 1.2) compiler was used to produce the object code in the coff format[21]. Our compression scheme was applied on the above code to produce compressed object code. For evaluating performance of our new compression scheme, we made use of an open source TI simulator[22], as the source code of the standard TI Code Composer Studio is not freely available. The open source TI simulator had to be modified slightly to accept compressed instructions and decompress them before execution. We have compared our scheme with the Bit Flip scheme proposed by Prakash et al.[13] including the overhead of the LAT in our figures.

Compression results for the Mediabench programs are shown in Figure 4. In the above results, the number of entries of the multiple dictionaries was not fixed and was allowed to vary with the program to be compressed. The average compression ratio was 73.5% (including the dictionary and LAT) for Mediabench programs. The fixed sized dictionary scheme gave an average compression ratio of 75.5% on Mediabench programs and 70.9% for TI benchmarks as shown in Figure 5 and Figure 6 respectively.

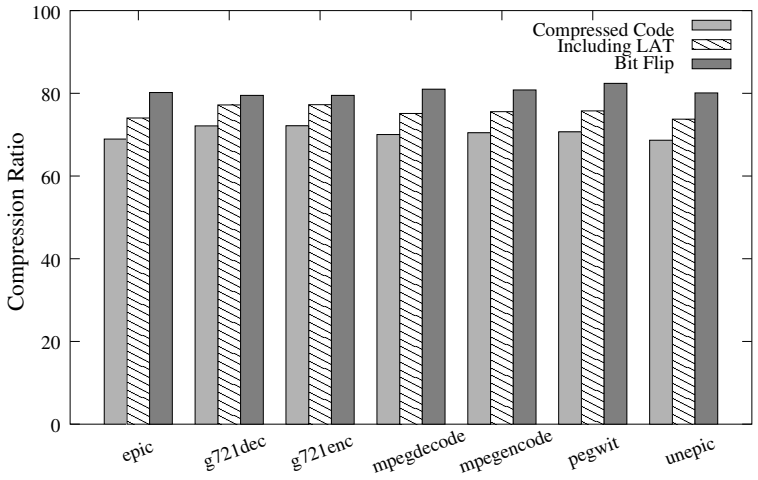

Figure 5: Compression ratios (fixed dictionary entries)

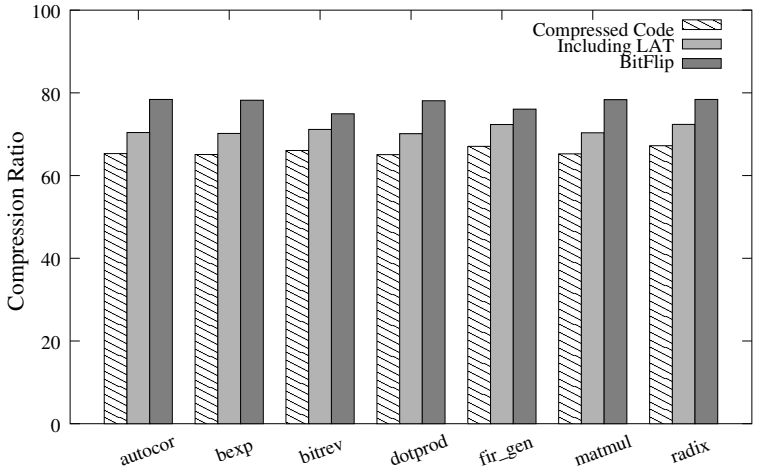

Figure 6: Compression ratios for TI benchmarks (fixed dictionary)

The advantages of a fixed size multiple dictionary scheme are seen in the smaller decompression overhead shown in Figure 8, compared with the overheads of the variable dictionary scheme, as shown in Figure 7, whose decoding hardware will be more complicated as the offset in the main dictionary varies with the program. We have used serial decompression because of its simple decompression hardware. The decompressor was pipelined as explained earlier and placed between the instruction cache and the main memory, so that the instruction cache can hold decompressed instructions. Cache sizes ranging from 256 bytes to 64KB were used for the above experiments. The pipelined serial decompressor gave an average decompression speed of 22 bits/cycle for the fixed size dictionary scheme and 13 bits/cycle for the variable one.

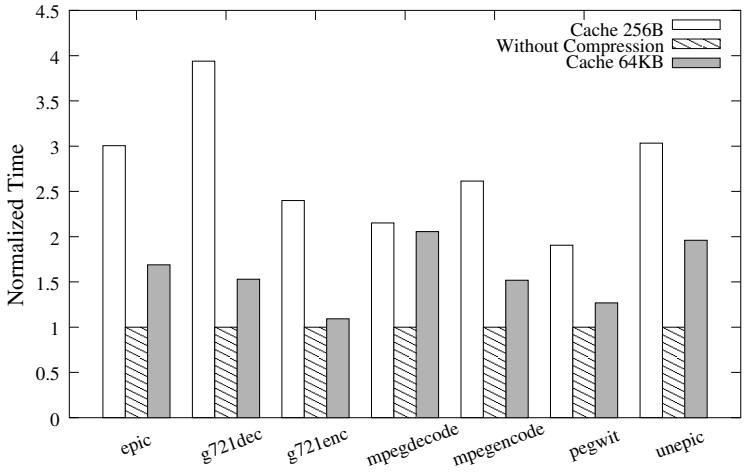

Figure 7: Decompression overhead (variable dictionary)

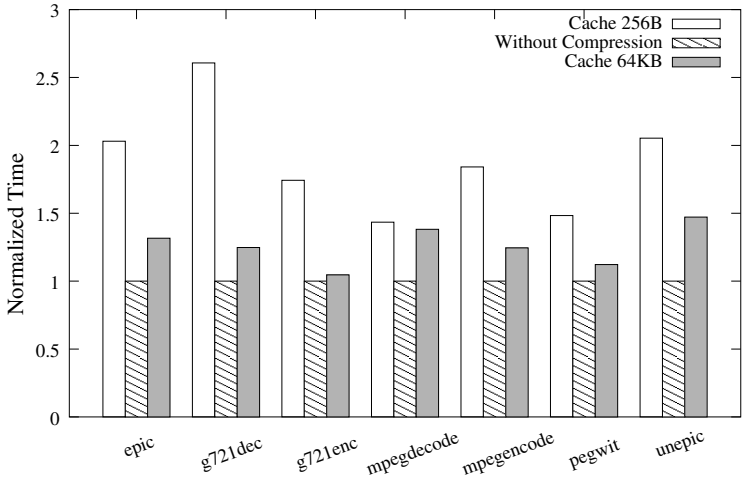

Figure 8: Decompression overhead (fixed dictionary)

4.1 Comparison with other schemes

Lekatsas and Wolf[14] report compression ratios ranging from 67-80% on the same architecture using a variation of standard arithmetic coding with a complex decompression hardware. However it is not clear whether this includes the overhead of the LAT. The Bit Flip scheme developed by Prakash et al.[13] uses a variation of the dictionary scheme to produce an average compression ratio of 76 % and 78% on TI and Mediabench programs(including the overheads of the LAT) with simpler hardware than that of the scheme of Lekatsas and Wolf. Our compression scheme seems to deliver improved compression ratios based on the simple techniques of instruction class division and dictionary construction. The first scheme using variable dictionary sizes produces an average compression of 73.5% but the decompression hardware is not simple as the size of the dictionary has to be extracted before decoding the instruction. The second scheme of fixed size multiple dictionaries overcomes this disad-

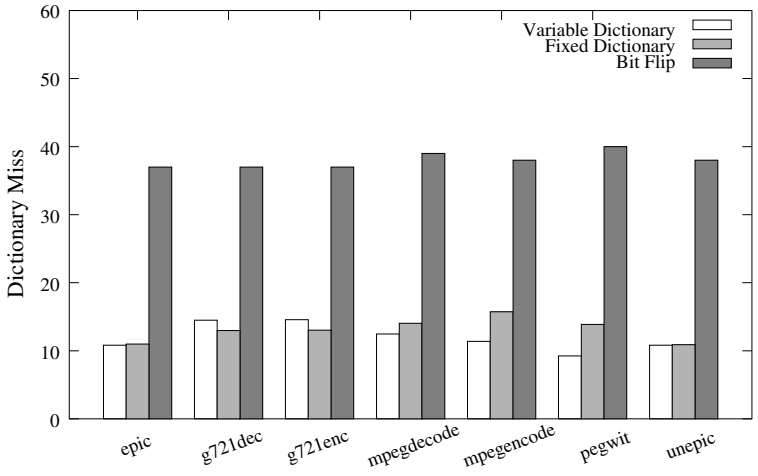

Figure 9: Percentage of uncompressed instruction

vantage. The extra hardware required by our scheme is reduced by the uniform dictionary entries and fixed base address of the dictionary segments. Because of the similar decompressor operation, our compression scheme can be directly compared to the fixed length basic dictionary scheme developed by Lefurgy and Mudge[12], which was able to produce a compression ratio of only 86% on the tightly decoded TI architecture.

The dictionary size of our scheme is also small compared to other dictionary schemes. We have worked with a fixed size dictionary of size slightly less than 2.5 KB for the Mediabench programs. Dictionary sizes for the bit flip scheme ranged from 1.5 KB to 5 KB with an average size of 3 KB. The percentage of uncompressed instructions of our scheme is only in the range of 12% compared to 35% of the Bit Flip scheme. Figure 9 shows a comparison of the percentage of uncompressed instructions of our scheme with the other dictionary method. The results show that multiple dictionaries are effective in capturing similarities between instruction segments.

The result of using profiling information in deciding which instructions to compress proved to be very fruitful. Keeping frequently used instructions uncompressed, results in good performance even with serial decompression. Figure 10 plots the average performance ratio versus the compression achieved for Mediabenchmarks keeping from 3% to 14% of the instructions uncompressed based on profiling information. The performance ratio is with respect to the running time of uncompressed code. The degradation in performance is almost 6% at 78% compression with a 64KB cache.

5 Conclusion and Future Work

A simple variant of the basic dictionary scheme has been proposed and implemented in this paper. We have shown that the new scheme is highly effective for the flexible instruction format architecture, of the TMS320C62x processor. Decompression hardware with fixed size dictionaries provides fast decoding. We have studied the space/time tradeoffs using fixed and variable dictionary sizes, and compression based on static analysis as well as dynamic

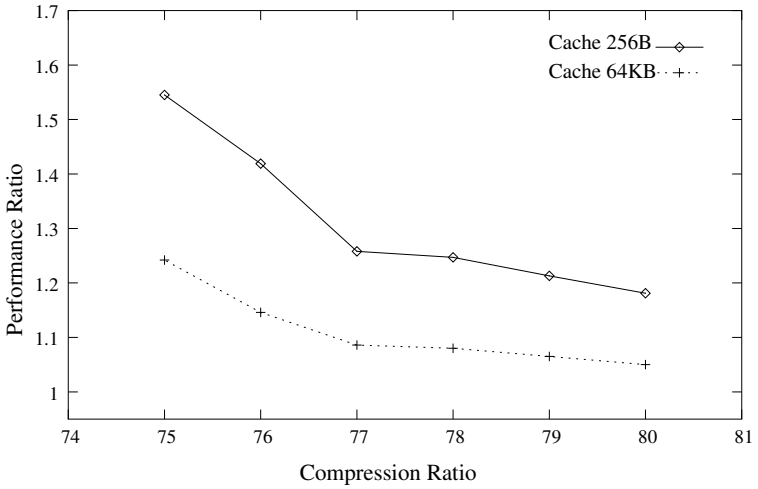

Figure 10: Performance improvement with profiling keeping from 3% to 14% of instructions uncompressed

analysis. The latter has proved to be useful in improving performance at the cost of a small increase in code size.

References

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

[2] Philips Trimedia TM 1300, TriMedia SDE Documentation set *http://www.semiconductors.philips.com.*

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

[4] Stan Liao, S.Devadas and K.Keutzer. Code Density optimization for embedded DSP processors using data compression techniques, *Proc. Conf. on Advanced Research in VLSI,1995*

[5] P.Bird and T.Mudge, An Instruction stream Compression Technique, *Technical report CSE-TR-319-96, EECS dept, University of Michigan, Nov.1996.*

[6] T.M.Kemp, R.M. Montoye,J.D. Harper,J.D. Palmer and D.J. Auerbach, A Decompression Core for PowerPC *IBM Journal of Research and Development,vol.42, no.6, Sept,1998.*

[7] N.Ishiura and M.Yamaguchi, Instruction code compression for application specific VLIW processors based on automatic Field Partitioning, *Proc. The Workshop on Synthesis and System Integration of Mixed Technologies, pp.105-109,Dec.1997.*

[8] Sang-Joon NAM,In-Cheol PARK and Chong-Min KYUNG,Improving Dictionary-Based Code Compression in VLIW Architectures,*IEICE Trans. Fundamentals,Vol.E82-A,No.11,p2318-2324,Nov.1999*

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

[10] Lekatsas and Wolf. Random Access Decompression Using Binary Arithmetic Coding.*Proceedings of the IEEE Data Compression Conference, pp306-315, 1999.*

[11] Lekatsas, Code Compression for Embedded Systems. *Ph.D. dissertation, Princeton University,2000.*

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

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

[14] Yuan Xie, H.Lekatsas and W.Wolf, Compression Ratio and Decompression Overhead Tradeoffs in Code Compression for VLIW Architectures.*Proceedings of International Conference on ASIC (ASION). October, 2001.*

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

[16] M. Kozuch and A. Wolfe.Compression of embedded system programs. *In Proc. Int'l Conf. on Computer Design, 1994.*

[17] M.Benes, S.M.Nowick, A.Wolfe. A fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors, *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, September, 1998.*

[18] Montserrat Ros and Peter Sutton. Compiler Optimization and Ordering Effects on VLIW Code Compression. *Proceedings of CASES 2003.*

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

[20] TMS320C62X Assembly Benchmarks *http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm.*

[21] Appendix A of the *TMS320C6x Assembly Language Tools User's Guide*,Texas Instruments,Jan.1997.

[22] Vinodh Cuppu and Bruce L. Jacob. Simulator for Texas Instruments TMS320C62x *http://www.enee.umd.edu/ ramvinod/c6xsim-1.0.tar.gz.*

[23] Yuan Xie,Wayne Wolf and Haris Lekatsas. Code Compression for VLIW processors using Variable-to-Fixed coding *ISSS-02,October 2-4,2002,Kyoto, Japan.*