Compiler Optimization and Ordering Effects on VLIW Code Compression

Montserrat Ros, Peter Sutton
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane Australia 4072
{ros, p.sutton}@itee.uq.edu.au

ABSTRACT

Code size has always been an important issue for all embedded applications as well as larger systems. Code compression techniques have been devised as a way of battling bloated code; however, the impact of VLIW compiler methods and outputs on these compression schemes has not been thoroughly investigated.

This paper describes the application of single- and multiple-instruction dictionary methods for code compression to decrease overall code size for the TI TMS320C6xxx DSP family. The compression scheme is applied to benchmarks taken from the Mediabench benchmark suite built with differing compiler optimization parameters.

In the single instruction encoding scheme, it was found that compression ratios were not a useful indicator of the best overall code size – the best results (smallest overall code size) were obtained when the compression scheme was applied to size-optimized code. In the multiple instruction encoding scheme, changing parallel instruction order was found to only slightly improve compression in unoptimized code and does not affect the code compression when it is applied to builds already optimized for size.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *code generation, Compilers, Optimization.*

General Terms

Performance.

Keywords

Code Compression, compiler optimizations, VLIW.

1. INTRODUCTION

The main goal of any compression algorithm is to reduce redundancy and increase the information content in a given block of information. However, code compression varies from normal text or data compression in many ways. The majority of text compression techniques view the information to be compressed as a block of data (such as a file) that needs to be compressed in size. When compressing a series of instructions, however, certain information needs to be retrieved at will. For example, branching and function entry points must be able to be decompressed on demand.

Code compression can be used as a method of reducing overall code size in embedded applications to reduce the amount of on- or off-chip memory required, or to increase the amount of code than can be used in those areas of memory.

Code compression efficiency is widely defined [5, 12, 13, 16, 18] by the compression ratio given by the following formula:

$$compression\ ratio = \frac{compressed\ \ program\ size}{original\ \ program\ size}$$

RISC processors have been the main focus for code compression techniques but VLIW (Very Long Instruction Word) processors are now being considered in this area as a result of their increased appeal to not only larger applications, but also the embedded field.

Their attraction stems from their powerful parallel architecture and their simple execution-unit design. Executing multiple instructions in parallel brings with it the obvious speedup of instruction processing, while introducing scheduling issues and resource constraints. Unlike superscalar implementations, VLIW architectures give the compiler responsibility for scheduling instructions and recognizing dependencies instead of the hardware doing so at runtime. As a result, code size can be largely dependant on compiler optimizations and efficiency.

Compilers for VLIW processors are required to package multiple instructions into packet-sized blocks for simultaneous execution. The way in which this is done can greatly increase or decrease the efficiency in compressing this generated code, and can have a large effect on overall code size. As the full responsibility for scheduling and packaging instructions in a VLIW program is given to the compiler, it is necessary to investigate the effects of that compiler's output on the compression ratios achieved as well as the overall code size after compression.

In this paper, we present a dictionary method compression scheme and investigate its performance when applied to various compiler optimizations and parallel instruction orderings. Section 2 presents related work in this field while Section 3 describes the dictionary-method compression scheme used. Section 4 outlines

CASES '03, Oct. 30 – Nov. 2, 2003, San Jose, California, USA.

results from applying the compression scheme to varied compiler output and Section 5 concludes with a discussion and comparison of results.

2. RELATED WORK

The idea of using code compression as a tool for chip size reduction in microprocessors has mostly incited interest in the area of single instruction issue (usually RISC) processors. These compression schemes can be categorized as dictionary methods such as CodePack™ in [8] or SADC in [19], or as statistical such as Arithmetic Coding [9, 23] or Markov models [18]. Some work has been done on the comparison of program optimization and compression for a RISC processor [6], however this is not extensive and there is no published work targeting VLIW compiler optimizations.

2.1 Code Compression on RISC processors

Code compression for RISC processors first emerged in a paper by Wolfe and Channin [22]. This paper proposed a new RISC system architecture based on existing architectures called a CCRP (compressed Code RISC Processor). Due to RISC programs tending to be larger, a CCRP was suggested to compress the code and use a 'code-expanding instruction cache', such that the decompression could be transparent to the processor. Various Huffman-based encoding schemes were used. By using a compression technique that did not give consideration to branch targets and function beginnings, extra hardware was required to fetch addresses.

Further developments in RISC code compression developed code compression methods that looked at compiler techniques [6, 7], expression trees and operand factorization [3, 4], enhanced dictionary schemes and statistical schemes based on Markov models and arithmetic coding.

Dictionary compression schemes have been investigated by Lefurgy et al [12] with fixed and variable length codewords. The dictionary compression is used to determine what portion of a program's object code is made up of its most frequent instructions and encode the more frequent instructions with a 'codeword' whose size is much smaller than the original instruction. This codeword references the dictionary where all original instructions are stored. Their study finds that on average more than 80% of the instructions in CINT95 have instruction words which are used multiple times, and one in-depth case showed that 10% of the most frequent instructions accounted for 66% of the overall code size of that program [12]. Investigation is also undertaken into compression based on multiple instruction dictionary entries.

The CodePack encoding algorithm[8] encompasses a similar idea, as the most common instructions are replaced by the indexes to the smallest dictionary, the next set of instructions (in order of frequency) are replaced by an index into the second-smallest dictionary, etc. This introduces some overhead to determine which dictionary is used to decompress the instruction, but ensures that very few bits are required for the most common instructions. CodePack is said to achieve compression ratios of 35-40%, not including the dictionaries themselves.

2.2 Code Compression on VLIW processors

The code compression techniques applied to date on multiple-issue processors (particularly the more original rigid VLIW processors, but also recently targeting variable execution set architectures) are limited to the works of Nam et al [21], Ishiura and Yamaguchi [10], Prakash et al [20], Xie et al [23-25] and Larin and Conte [11]. This is only a subset of the techniques available for both data compression and single-issue code compression.

Nam et al[21] achieved average compression ratios of 63%-71% on SPEC95 benchmarks for varying VLIW architectures using a dictionary compression method and compared the difference in performance of "identical" and "isomorphic" instruction word encoding schemes. Nam[21] uses the separation into opcodes and operands across the entire fetch-packet, hence for an x-issue processor, there will be x opcodes and x operand streams. Two dictionaries are required, one to hold the opcode entries and the other to hold operand entries. Two methods of investigating common instruction words are compared (identical – whole instructions words; and isomorphic – split into opcode/operand fields) in varying VLIW architectures. Their results show that using the isomorphic instruction words method out-performed the identical instruction words method by a compression ratio difference of at least 17%.

Ishiura and Yamaguchi [10] also investigate code compression for VLIW processors, this time based on a statistical method called Automatic Field Partitioning. Their paper reduces the problem of compressing code to the problem of finding the field partitioning that yields the smallest compression ratio. Each field partition is then encoded with a dictionary scheme. Ishiura and Yamaguchi [10] achieve compression ratios of 46-60%.

Prakash et al [20] present a dictionary based encoding scheme that divides instructions into 2 16-bit halves. For each half, a dictionary is constructed that contains a choice set of vectors such that a majority of the vectors used throughout the program in that half of the instruction differ by one of the dictionary vectors by a small Hamming Distance (the Hamming Distance between two vectors is the number of bits that are different). Each compressed instruction is then replaced by two codewords representing each half-instruction. These codewords are a combination of the indexes into the relevant dictionaries as well as information about which bits are toggled. This method means that two vectors that are different at only one bit will not require both vectors to be stored in the dictionary. Compression ratios of 80% are recorded.

Xie et al.[23, 25] are the first works to really target a VLES (various length execution set) such as the TMS320C6x where bit0 of each instruction tells the architecture whether the next instruction may be executed with the current set of instructions or not. Xie uses a reduced-precision arithmetic coding technique combined with a Markov model (statistical method) and applies it to similar systems with different sized sub-blocks. Increasing the block size decreases the compression ratio, but also increases the time taken to decompress. The 16-byte sub-block scheme yields the best compression rates at 67.3% – 69.7% but processing 11.2 – 11.5 bits per clock cycle; whilst the 4-byte sub-block scheme although processing 47.01 – 47.42 bits per clock cycle has a compression ratio of 76.7% – 80.6%.

Xie et al. [24] also present a Tunstall-based memory-less variable-to-fixed encoding scheme as well as an improved Markov variable-to-fixed algorithm with varying model depths and widths. It is reported that 4-bit encoding produces the best

results. Compression ratio was found to improve with larger codeword sizes until after 4 bits. This was mainly due to the fact that less padding was required in 4-bit codeword compression. The use of variable-to-fixed encoding means that codewords are arbitrarily assigned and this assignment can be used to an advantage to reduce the number of bit toggles on the instruction bus.

Finally, the related work by Larin and Conte [11] conducts a comparison between code compression methods and a tailored encoding of the Instruction Set Architecture. In the tailored ISA method, instructions were compacted into the smallest number of bits required to still represent the same information. This method produced new code at 64% of the original code size, though at a much smaller cost to decoding hardware than standard compression. This was compared to a Huffman encoding with the code treated as bytes (72%), operations separated into streams (75%), and operations as a whole (30%). The Huffman compression applied to instructions as a whole was found to produce These compression ratios did not include the Address Translation Table required to maintain branch target information. This added approximately 15.5% to the compressed code size.

3. ENCODING SCHEMES

In order to analyze the effects of compiler outputs on the compressibility of a program, single and multiple dictionary encoding schemes were used to illustrate the frequencies of instructions associated with VLIW code.

3.1 Single Instruction Encoding Scheme

The single instruction encoding scheme used in this paper is a dictionary compression method that analyses the instructions in a program, builds a dictionary with the most frequent instructions and compresses the original program by replacing common instructions with a reference to the dictionary. This is a technique similar to [13], except that instructions appearing only once are not compressed.

The initial pass of the encoding scheme reads in a compiled object file and gathers statistics of the frequencies of unique instructions. This information is used to decide which instructions will be included in the dictionary. The second pass through the program takes each instruction and either leaves it as it is, or compresses it if it is found in the dictionary. Figure 1 demonstrates this.

This dictionary method has been implemented using dictionaries of 4- and 12- bits which correspond to 8- and 16-bit codewords as a result of the compression overhead required (described in Section 3.5).

3.2 Multiple Instruction Encoding Scheme

The multiple instruction encoding scheme adopted is very similar to the single instruction scheme, except that sequences of 2- to 8-instructions are considered as 'dictionary words' instead of lone instructions. In a way, the scheme in Section 3.1 is a version of this encoding scheme, where sequences of 1-instruction are considered.

Although both encoding schemes are similar, the method is very different. Because sequences of 2 or more instructions are being considered in this scheme, the sequences in a given program can 'overlap'. This means that when a particular instruction sequence

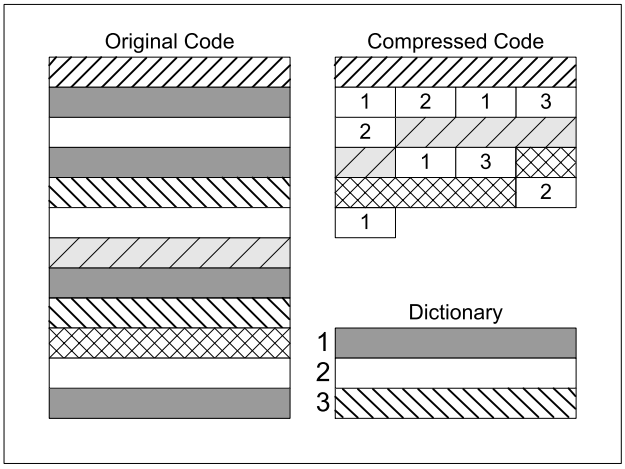

Figure 1 – Dictionary Encoding Scheme Example

is chosen for addition to the dictionary (and replacement throughout the code), this affects the statistics for the remaining sets of sequences. As a result, new statistics must be gathered upon every iteration of the dictionary-filling process.

This brings into question the algorithm to be used for dictionary word selection. In this paper, we present results for a greedy method of dictionary word choice, choosing the most frequent sequence of instructions at all times. It is possible that a better compression ratio could be achieved through an alternative algorithm for the choice of dictionary words; however, as the aim of this paper is to measure compiler optimization effects on code compression, the greedy approach is an appropriate one.

3.3 Parallel Instruction Ordering

Another property of VLIW code investigated in this paper, is the effect of parallel instruction ordering on code compression. As mentioned earlier, for VLIW processors, the compiler assumes responsibility for scheduling and ordering instructions. This includes detecting when instructions can be executed in parallel and adding this information to the code itself. In the TI TMS320C6x Family, this is done by using the last bit of the instruction to signify whether it can be executed in parallel with the following instruction or not. Fetch packets are 8 instructions long, so the longest possible sequence of parallel instructions is 8 in a row. These groups of parallel instructions, in the TMS320C6x series, can be ordered by the compiler in any way, as the instructions themselves contain information as to which execution unit they will be run on. This means that the compiler can arbitrarily choose the order of this sequence, with the end result being the same – they all get executed in parallel and on their respective execution units. To investigate the effect of parallel instruction ordering on compressibility, a canonical sort order[1] was applied to groups of parallel instructions before compression.

Thus, the multiple instruction encoding scheme described in Section 3.2 was applied to benchmark builds before and after the parallel instruction ordering took place. Results were produced

[1] The sort order used was one based on the bitwise comparison of instructions

for benchmarks compiled for the 67xx floating point target without libraries using a byte-aligned best-fit codeword size to encode the dictionary entries.

3.4 Branch Target Patching

One of the major differences between standard data compression and code compression is that function entry points and branch targets must be preserved in some way. This is so that references to memory locations do not return invalid code. The method of branch target patching is a way of manipulating (changing) the code so as to reflect the changes in code size, and was introduced in [12].

As a result, instructions that branch forward x instructions (where y of those are compressed and $x - y$ are maintained) need to be patched. Instead of branching forward to

$$x \times bytes\_per\_instruction ,$$

the branch needs to be changed to

$$y \times bytes\_in\_codewords + (x - y) \times bytes\_per\_instruction$$

bytes.

This ensures that all requests from the CPU for memory locations are already correct and the hardware does not have to be altered to recalculate the correct locations of the instructions wanted.

This method of 'patching' instructions introduces a dilemma for relative branching instructions. What if the relative branch instruction itself is required to be compressed? This would mean that the instruction would be stored in the dictionary, and an index into the dictionary would be stored in place of the original branching instruction. Then the number of bytes to branch would be changed, making the instruction in the dictionary incorrect.

This sort of problem is akin to the problem found in [12] where compressing relative branches is NP-complete. To avoid this problem, relative branches are not compressed.

3.5 Compression Overhead

Overhead is included in all compression schemes albeit in many different ways. In the case of this encoding scheme, overhead is introduced by having to add information that allows an instruction to be decoded as either a codeword or an original instruction. A prefix bit could have been added to determine whether an instruction is compressed or not, however that would result in code not being byte aligned which can cause difficulty in designing a hardware engine to decompress the instructions. The method used in this paper expands the instruction set architecture to make use of the unused opcode-space available. In particular, the TI TMS320C6x series has various classes/types of instructions that are each categorized by the values of bits 2-6 as shown in Figure 2.

The set of 4 bits 1100 does not correspond to any 'normal' instruction, and can be used to flag that the codeword is not an original instruction. The codewords inserted instead of the original instructions will need to include these extra 4 bits which will essentially be the overhead in this encoding scheme. As a result, the codeword size turns out to be 4 bits larger than the

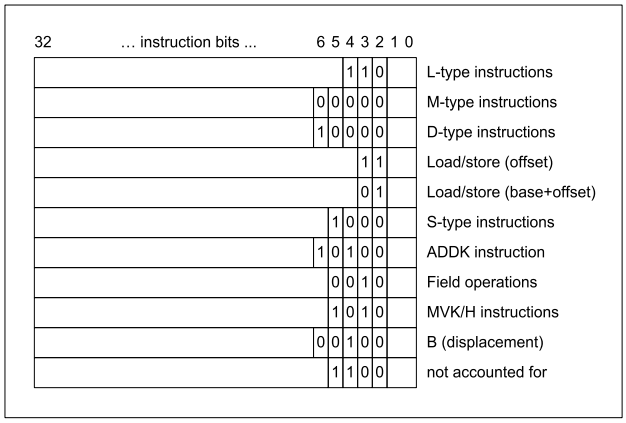

Figure 2 – TI TMS320C6x Opcode Space

index into the dictionary and so 8- and 16-bit codewords result in $2^{8-4} = 16$ and $2^{16-4} = 4096$ entry dictionaries.

The encoding scheme takes care not to compress any instruction that only occurs once, because doing so would increase the number of bytes required to represent the instruction. This may mean that the dictionary is not filled. Dictionary sizes can thus vary from program to program depending on the density of instructions that exist more than once.

3.6 Decompression Hardware and Runtime Overhead

Like most code compression schemes, hardware would be required to analyze instructions as they are fetched from memory and decide whether to allow the instruction to pass on to the CPU unaltered, or whether to decompress the recognized codeword by looking up a dictionary and passing-on the dictionary word instead. This introduces a delay when processing compressed instructions that may affect the performance of the processor. Figure 3 shows a block diagram of the required hardware.

The size of the decompression hardware required to process the compressed instructions also needs to be taken into account. Huge reductions in code size at the cost of a large increase in die size on the processor (as a result of a large dictionary) would not be advisable. As the dictionary is the largest component of the decompression hardware, dictionary sizes need to be taken into account when considering compression techniques. The compression ratios in our study take into account the compression

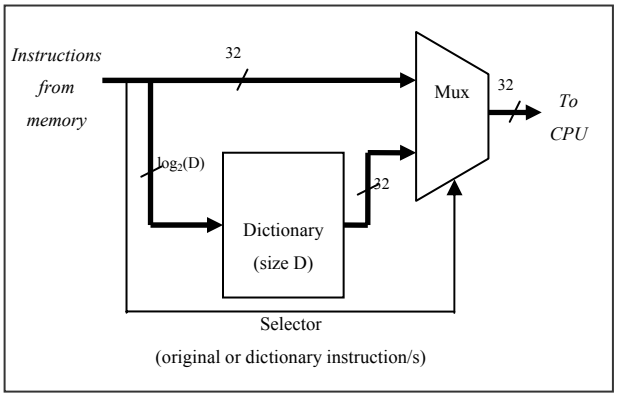

Figure 3 –Block Diagram for Decoder Hardware

overhead associated with instruction patching and the dictionary.

The performance and run-time overhead of this sort of decompression scheme has been investigated in other papers [7, 14, 15, 17, 22, 25] and is beyond the scope of this paper. Although code compression generally results in reduced performance as a result of the hardware required, some studies have shown that applying code compression to post-cache architectures produces a benefit to performance through reduced cache-misses and fewer instruction fetches [19].

4. APPLICATION

The aim of this paper is to investigate the impact of various compiler optimizations on the compressibility of compiled object code. In particular, the TI TMS320C6x DSP processor fam-ily [2] has been chosen as the target VLIW processor family, and the Mediabench benchmark programs [1] have been chosen as appropriate benchmarks for this sort of processor. The TI Code Composer Studio IDE was used to generate various builds for each benchmark, each build using a different set of optimization options. The study presented in this paper is limited to this particular processor and compiler, but as there is no other published work of the effect of compiler optimizations and ordering on VLIW code compression, it serves as an indication of an area that needs to be further examined.

4.1 Mediabench Benchmarks

Mediabench [1] was chosen as an appropriate set of benchmark programs to investigate. These programs were compiled for both fixed point and floating point targets. The benchmarks used included:

- adpcm (rawc- and rawd-audio)
- g721 (encode and decode)
- epic (and unepic)
- mpeg (mpeg2enc and mpeg2dec)
- jpeg (cjpeg and djpeg)

4.2 Compiler optimizations

The TI compiler offered two sets of optimization control through argument flags. The first and most common optimization option is that of the numerical level associated with optimization flags '-o0' to '-o3'. This gives 5 levels of numeric optimization:

- No optimization
- '-o0' (register-level optimization)
  Performs control-flow-graph simplification, loop rotation, allocates variables to registers, eliminates unused code, simplifies expressions and statements, expands inline functions.
- '-o1' (local optimization)
  Performs all –o0 optimizations and: Performs local copy/constant propagation, removes unused assignments, eliminates local common expressions
- '-o2' (global optimization)
  Performs all –o1 optimizations and: software optimizing, loop optimizations and unrolling, eliminates global common subexpressions and unused assignments, converts array references in loops to increment pointer form.
- '-o3' (file-level optimization)
  Performs all –o2 optimizations and: removes uncalled functions, simplifies functions with unused return values, makes functions inline, reorders function declarations, propagates arguments into function bodies when the same value is always passed

Also, the TI compiler offers a separate 5 levels of optimization for code size versus speed (performance).

- (no flag) Speed Most Critical
- ' ms0' Speed More Critical
- '-ms1' Speed Critical
- '-ms2' Size Critical
- '-ms3' Size Most Critical

These levels were found to increase or reduce how many of the instructions in a given program were scheduled for execution on their own, or in parallel. For example, with the '-ms3' option, where size is considered most critical, it was found that more than 99% of the instructions were scheduled to be executed alone.

The two sets of 5-option optimization parameters effectively give 25 levels of optimization, including optimizing for speed or size. The compiler documentation suggests high values of the –o parameter, combined with high values of the –ms parameter to achieve the smallest code size. This was found to be generally true of the benchmarks built, although the smallest code size was not always achieved with the '–ms3 –o3' combination.

5. RESULTS

The heading of a section should be in Times New Roman 12-point bold in all-capitals flush left with an additional 6-points of white space above the section head. Sections and subsequent sub-sections should be numbered and flush left. For a section head and a subsection head together (such as Section 3 and subsection 3.1), use no additional space above the subsection head.

5.1 Single Instruction Encoding Scheme

The built benchmarks were passed through a compression program that applied the encoding scheme defined in Section 3.1. Information was retrieved from this program, including the benchmark build size pre- and post- compression, dictionary size and compression ratios. (All compressed program sizes and compression ratios in this paper make mention of code size with the dictionary to give a truer indication of the compression achieved).

The compression ratios varied from 69.2% to 94.6% with dictionaries. Some of the higher (worse) compression ratios resulted from using codewords that were not of suitable length (i.e. using 1-byte codewords for large benchmarks and 2-byte codewords for smaller benchmarks). When the 'best-fit' codeword size was used for each benchmark, the compression ratio range became 69.2% to 88.5%.

In general, the larger benchmarks compressed best under 16-bit codeword compression, while the smaller benchmarks produced more favorable results with the 8-bit codeword compression. However, this is highly dependant on the portion of repeated instructions in the code. Figure 4 shows the average sizes (pre- and post- compression) and compression ratios for each benchmark (averaged across all builds of the benchmark). The

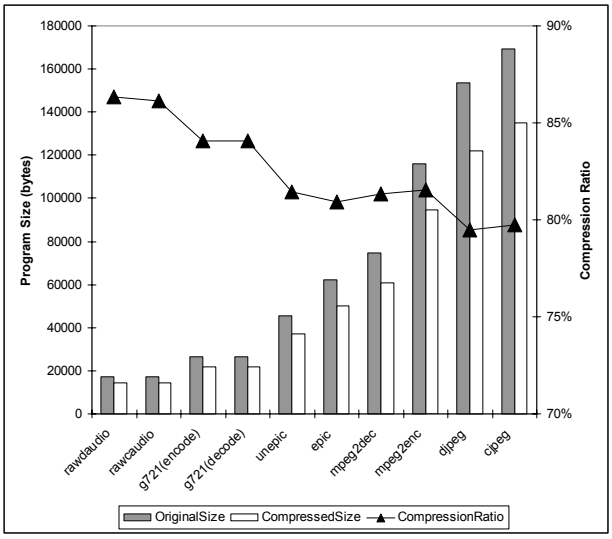

Figure 4 – Relative Sizes and Compression Ratios of Benchmarks

average compression ratios for fixed- and floating- point targets across all benchmarks were very similar. The floating-point builds started smaller and had slightly better (lower) compression ratios.

Analysis of the parameter options in the compiler drew some interesting results. As the benchmarks varied greatly in size, the sizes were 'normalized' before comparing absolute sizes of builds for varying optimization parameters. Normalization was done by comparing each parameter build to the build with no parameters [expressed as '-ms(none)' and '-o(none)'] in the same group. Figures 5 and 6 show the average of the normalized sizes, across all benchmarks used, for original and compressed programs.

As expected, the '–ms3' (Size Most Critical) option produced the smallest original object code out of the '–ms' options. This option corresponds to the (darkest) bars at the forefront of Figure 5. However, when the encoding scheme was applied, the average compression ratio of programs built with the '–ms3' option was worse (higher) than all but one of the other '–ms' options. This reflects the measures already taken to optimize the code for size. Even with this higher (worse) average compression ratio, compressed '–ms3' code was still the smallest of the '–ms' options overall. Figure 6 shows the same combinations of parameters as Figure 5, but after the encoding scheme is applied. Builds with the '–ms3' option were still the smallest overall. Comparison of Figures 5 and 6 shows that the relative sizes of code compiled for each optimization parameter pair are similar before and after code compression is applied and compression does not affect the relative sizes.

The higher levels of optimization ('–o2' and '–o3') seemed to generate larger original code than the '–o0' and '–o1' parameters. This is likely to be as a result of the optimization techniques involved. For example, the act of loop unrolling or propagating arguments into function bodies may optimize the performance of the program, but may also increase the size of the program.

Object code built with no '–o' parameter was by far the largest (rightmost columns in Figure 5). This lack of optimization (and presumed redundancy) resulted in the best (lowest) average compression ratio and this is evident by the lower bars for this category in Figure 7 (left-most bar in each group of 5 bars). Although the compression ratios were better than that of other parameters, this did not reduce the code size enough. The overall code size was still the largest after compression. The compression ratios in Figure 7 are averaged across all benchmarks and highest/lowest values are depicted by error bars, for each parameter combination.

The **jpeg** compression/decompression utilities (**cjpeg**/**djpeg**) seemed to compress well in all situations. Table 1 outlines the performance of **cjpeg** builds with no library, under 16-bit codeword compression for the floating-point target. In this table, compression ratio is defined - as in previous examples – to be the ratio of compressed code to uncompressed code for each build.

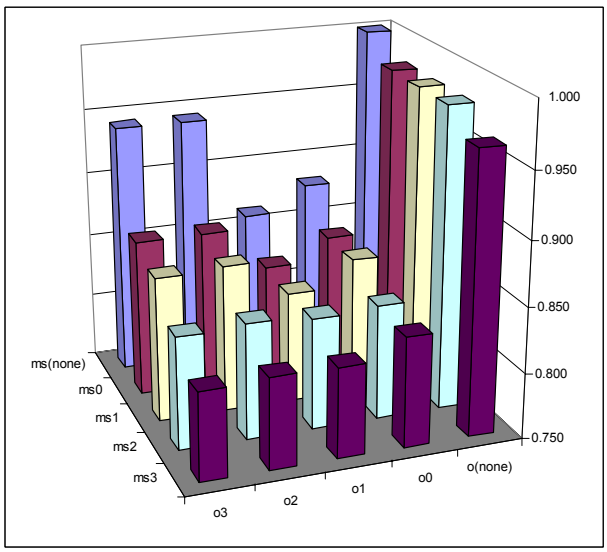

Figure 5 – Normalized Average Original Benchmark Sizes

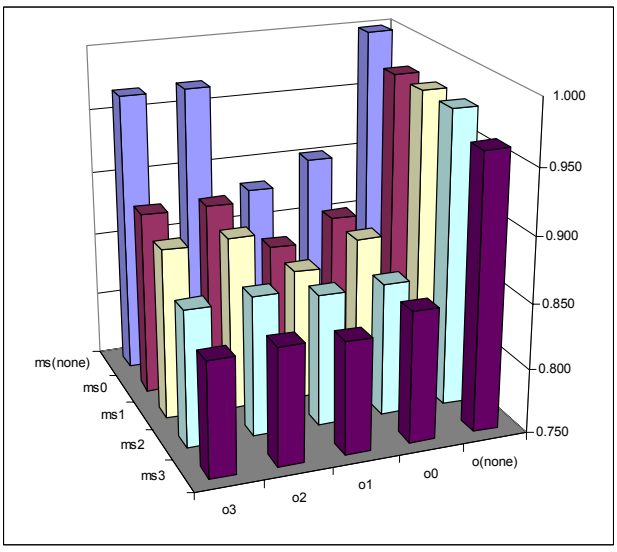

Figure 6 – Normalized Average Compressed Benchmark Sizes

Table 1 – Sizes and Ratios for the cjpeg Benchmark under 16-bit compression

Optimization Parameters	Optimized Code		Compressed Code (including dictionary)		
	Size (bytes)	Fraction of Un-optimized Code† (%)	Size (bytes)	Compression Ratio	Fraction of Un-optimized Code† (%)
-ms(none) -o(none) †	167264	100.0%	117878	70.5%	70.5%
-ms(none) -o0	146720	87.7%	114772	78.2%	68.6%
-ms(none) -o1	140640	84.1%	110662	78.7%	66.2%
-ms(none) -o2	152000	90.9%	120356	79.2%	72.0%
-ms(none) -o3	153088	91.5%	121166	79.1%	72.4%
-ms0 -o(none)	161920	96.8%	113912	70.4%	68.1%
-ms0 -o0	139488	83.4%	108348	77.7%	64.8%
-ms0 -o1	134368	80.3%	105144	78.3%	62.9%
-ms0 -o2	144288	86.3%	113874	78.9%	68.1%
-ms0 -o3	145280	86.9%	114628	78.9%	68.5%
-ms1 -o(none)	161920	96.8%	113912	70.4%	68.1%
-ms1 -o0	139488	83.4%	108348	77.7%	64.8%
-ms1 -o1	133600	79.9%	104808	78.4%	62.7%
-ms1 -o2	142624	85.3%	112674	79.0%	67.4%
-ms1 -o3	142656	85.3%	112712	79.0%	67.4%
-ms2 -o(none)	161920	96.8%	113912	70.4%	68.1%
-ms2 -o0	139488	83.4%	108348	77.7%	64.8%
-ms2 -o1	133600	79.9%	104808	78.4%	62.7%
-ms2 -o2	135552	81.0%	106898	78.9%	63.9%
-ms2 -o3	135712	81.1%	107000	78.8%	64.0%
-ms3 -o(none)	158560	94.8%	110226	**69.5%**	65.9%
-ms3 -o0	135872	81.2%	103698	76.3%	62.0%
-ms3 -o1	**129792**	77.6%	100008	77.1%	**59.8%**
-ms3 -o2	131232	78.5%	101958	77.7%	61.0%
-ms3 -o3	131200	78.4%	101918	77.7%	60.9%

This means that the code it is being compared with is optimized already (with the use of different parameters in each build case). The other two columns compare the optimized and compressed program sizes with the *original* un-optimized, uncompressed program size (shaded in dark grey in Table 1).

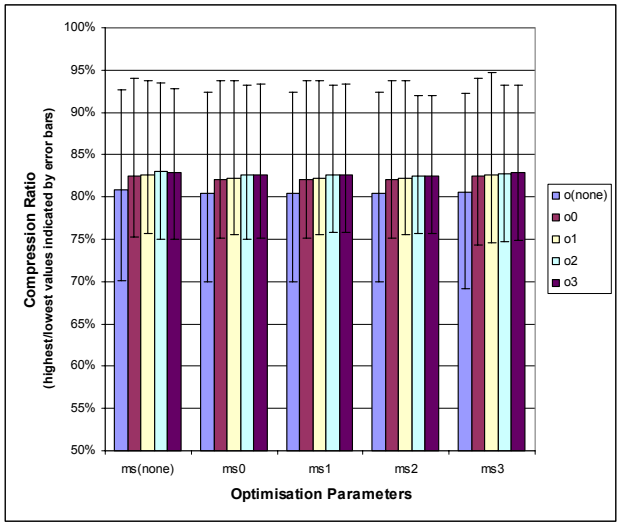

Figure 7 – Average Compression Ratios (including dictionary, all benchmarks)

We see that the smallest original code is generated by the '–ms3 –o1' parameters and that this results in the smallest compressed code size. Note, however, that this is not the build that exhibits the best compression ratio. That "honor" goes to the '-ms3 –o(none)' build (69.5%) but results in code that is 10% larger than for the '-ms3 –o1' build (with a compression ratio of 77.1%).

The results in Table 1 show that although compression ratio adequately measures the relationship of uncompressed code to compressed code, it is not a useful indicator of final code size unless compiler optimization is taken into account.

5.2 Multiple Instruction Encoding Scheme

The same benchmarks were compressed with the multiple instruction encoding scheme described in Section 3.2. Sets of sequences from 2 to 8 instructions long were used and compression schemes using smaller sequences resulted in lower (better) compression ratios. This shows that the reduction in code size attributed to the high frequencies of smaller instruction sequences outweighs the code size reduction attributed to replacing a larger instruction sequence with one codeword. Figure 8 shows the average compression ratios attained across all benchmarks for the sets of 2 to 8 sequences of instructions.

† Un-optimized code size refers to the size of code built with no optimization parameters (167264 bytes)

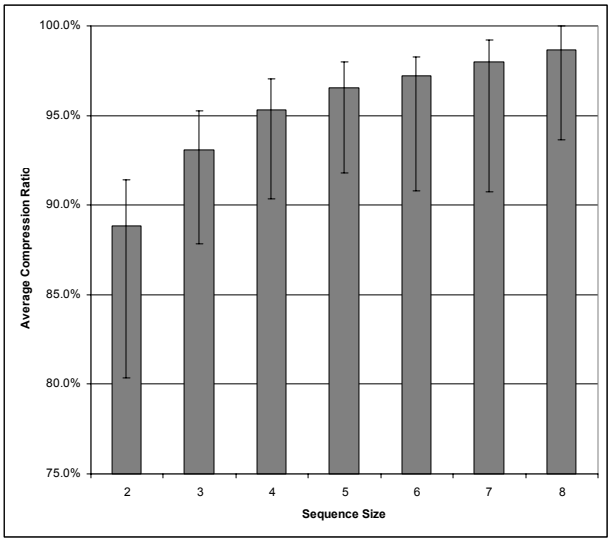

Figure 8 – Average Compression Ratios for Sets of Sequences

To investigate the effect of parallel instruction ordering on the benchmarks, the multiple instruction encoding scheme was applied before and after parallel instructions were sorted. This gave some insignificant results in the unoptimized code, however made no difference whatsoever to the highly optimized code. Tables 2 and 3 show the compression results for the **cjpeg** example benchmark before and after instruction ordering.

In general, the results after ordering were generally better, but only by at most 0.4%. However, the differences only occurred in builds that were less than fully optimized. Fully optimized builds (especially those with the '-ms3' parameter) displayed no evidence of a change in the compression ratio before and after parallel instructions were reordered. Further investigation found that this was because these optimization levels resulted in very few instructions being executed in parallel, e.g., for the cjpeg builds with the '–ms3' option, over 99.9% of the instructions were executed alone, so reordering the remaining less than 0.1% of instructions will certainly have no effect on code compression.

6. CONCLUSIONS

The investigation presented in this paper has looked at the effect of compiler optimizations and parallel instruction ordering on code compression for VLIW code. In particular, code produced by the TI Code Composer Studio IDE for the TI TMS320C6x DSP processor family was examined. It has been found that code compression, and in particular compression ratios, must always be considered in the context of compiler optimization parameters. Compression ratios do differ from one parameter combination to another and unoptimized code seemed to generate higher compression ratios. However, the best compression ratio is not always an indication of best overall size. In general, to obtain the smallest overall size after compression, a compression scheme should be applied to already size-optimized code.

With multiple instruction compression, reordering of parallel instructions was found to have, at best, a small influence on code compressibility. With size-optimized code, there was no effect. This was found to be because the compiler produced code with very few instructions to be executed in parallel.

Further investigation will look at other VLIW processors and compilers in order to be able to formulate a generalization of the impact of VLIW compilers and compilation options on code compressibility.

7. REFERENCES

[1] *Mediabench Benchmarks*, accessed 2003, http://www.cs.ucla.edu/~leec/mediabench/

[2] *TMS320C6000 CPU and Instruction Set Reference Guide*: Texas-Instruments, 2000.

[3] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain, "Expression-tree-based algorithms for code compression on embedded RISC architectures," in *IEEE Transactions on Very Large Scale Integration VLSI Systems. Oct. 2000; 8(5)*: IEEE, 2000, pp. 530-3.

Table 2 – Compression Results Before Instruction Ordering

	-ms(none)			-ms0			-ms1			-ms2			-ms3		
	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio
-o(none)	167264	134480	80.4%	161920	130308	80.5%	161920	130308	80.5%	161920	130308	80.5%	158560	127840	80.6%
-o0	146720	126046	85.9%	139488	119692	85.8%	139488	119692	85.8%	139488	119692	85.8%	135872	116188	85.5%
-o1	140640	120782	85.9%	134368	115282	85.8%	133600	114886	86.0%	133600	114886	86.0%	129792	111322	85.8%
-o2	152000	132600	87.2%	144288	126046	87.4%	142624	124802	87.5%	135552	118188	87.2%	131232	114536	87.3%
-o3	153088	133382	87.1%	145280	126778	87.3%	142656	124822	87.5%	135712	118210	87.1%	131200	114428	87.2%

Table 3 – Compression Results After Instruction Ordering

	-ms(none)			-ms0			-ms1			-ms2			-ms3		
	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio	orig	comp	ratio
-o(none)	167264	134258	80.3%	161920	130154	80.4%	161920	130154	80.4%	161920	130154	80.4%	158560	127840	80.6%
-o0	146720	125692	85.7%	139488	119416	85.6%	139488	119416	85.6%	139488	119416	85.6%	135872	116184	85.5%
-o1	140640	120644	85.8%	134368	115194	85.7%	133600	114674	85.8%	133600	114674	85.8%	129792	111318	85.8%
-o2	152000	131860	86.8%	144288	125322	86.9%	142624	124188	87.1%	135552	117906	87.0%	131232	114532	87.3%
-o3	153088	132676	86.7%	145280	126128	86.8%	142656	124250	87.1%	135712	117984	86.9%	131200	114428	87.2%

[4] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code compression based on operand factorization," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture. 1998*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, pp. 194-201.

[5] P. Centoducatte, G. Araujo, and R. Pannain, "Compressed code execution on DSP architectures," in *Proceedings 12th International Symposium on System Synthesis. 1999*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 56-61.

[6] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in *SIGPLAN Notices. May 1999; 34(5)*: ACM, 1999, pp. 139-49.

[7] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression," in *SIGPLAN Notices. May 1997; 32(5)*: ACM, 1997, pp. 358-65.

[8] M. B. Game, A, "CodePack: Code Compression for PowerPC processors (version 1.0)," PowerPC Embedded Processor Solutions, IBM, North Carolina 2000.

[9] P. G. Howard and J. S. Vitter, "Practical implementations of arithmetic coding," in *Image and text compression. 1992*, J. A. Storer, Ed.: Kluwer Academic Publishers, Dordrecht, Netherlands, 1992, pp. 85-112.

[10] N. Ishiura and M. Yamaguchi, "Instruction Code Compression for Application Specific VLIW Processors BAsed on utomatic Field Partitioning," 1997.

[11] S. Y. Larin and T. M. Conte, "Compiler-driven cached code compression schemes for embedded ILP processors," in *MICRO 32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. 1999*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 82-92.

[12] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proceedings. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture Cat. No.97TB100184. 1997*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1997, pp. 194-203.

[13] C. Lefurgy and T. Mudge, "Code Compression for DSP," presented at Compiler and Architecture Support for Embedded Computing Systems, George Washington University, Washington DC, 1998.

[14] C. Lefurgy, E. Piccininni, and T. Mudge, "Evaluation of a high performance code compression method," in *MICRO 32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture. 1999*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 93-102.

[15] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing code size with run-time decompression," in *Proceedings Sixth International Symposium on High Performance Computer Architecture. HPCA 6 Cat. No.PR00550. 1999*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1999, pp. 218-28.

[16] C. R. Lefurgy, "Efficient execution of compressed programs," University of Michigan, 2000, pp. 201.

[17] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proceedings 2000. Design Automation Conference. IEEE Cat. No.00CH37106. 2000*: ACM, New York, NY, USA, 2000, pp. 294-9.

[18] H. Lekatsas and W. Wolf, "SAMC: a code compression algorithm for embedded processors," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1689-701, 1999.

[19] H. A. Lekatsas, "Code compression for embedded systems," Princeton University, 2000, pp. 171.

[20] J. S. Prakash, C.; Shankar, P.; Srikant, Y.N., "A Simple and Fast Scheme for Code Compression for VLIW processors," presented at Data Compression Conference, 2003.

[21] Sang Joon Nam, In Cheol Park, and Chong Min Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, pp. 2318-24, 1999.

[22] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *SIGMICRO Newsletter. Dec. 1992; 23(1 2)*, 1992, pp. 81-91.

[23] Yuan Xie, H. Lekatsas, and W. Wolf, "Code compression for VLIW processors," in *Proceedings DCC 2001. Data Compression Conference. 2001*, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2001, pp. 525.

[24] Yuan Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *15th International Symposium on System Synthesis IEEE Cat. No.02EX631. 2002*: ACM, New York, NY, USA, 2002, pp. 138-43.

[25] Yuan Xie, W. Wolf, and H. Lekatsas, "A code decompression architecture for VLIW processors," in *Proceedings 34th ACM/IEEE International Symposium on Microarchitecture. 2001*: IEEE Comput. Soc, Los Alamitos, CA, USA, 2001, pp. 66-75.