# A Hamming Distance Based VLIW/EPIC Code Compression Technique

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

This paper presents and reports on a VLIW code compression technique based on vector Hamming distances [19]. It investigates the appropriate selection of dictionary vectors such that all program vectors are at most a specified maximum Hamming distance from a dictionary vector. Bit toggling information is used to restore the original vector.

A dictionary vector selection method which considered both vector frequency as well as maximum coverage achieved better results than just considering vector frequency or vector coverage independently. This method was found to outperform standard dictionary compression on TI TMS320C6x program code by an average of 8%, giving compression ratios of 72.1% to 80.3% when applied to the smallest compiler builds. The most favorable results were achieved with a Hamming distance upper limit of 3.

An investigation into parallel compression showed that dividing the program into 32-bit parallel streams returned an average compression ratio of 79.4% for files larger than 200kb. This approach enables parallel decompression of instruction streams within a VLIW instruction word. Suggestions for further work include compiler/compression integration, more sophisticated dictionary selection methods and better codeword allocation.

## Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]

## General Terms

Algorithms, Performance.

## Keywords

Code Compression, VLIW, Hamming distance.

## 1. INTRODUCTION

Code size management is a significant issue for embedded system design. As consumers require more functionality, applications for embedded devices become more and more complex. Furthermore, abstract programming languages are being chosen for the development of embedded applications such that the development can be steered away from the hardware level and more towards a platform-independent design philosophy. As a result of both of these considerations, embedded application code sizes are increasing and this can pose a problem for designers.

Several methods for compressing or compacting code size have been presented in the literature to date, though most algorithms have focused mainly on RISC processors. Lately, however, VLIW (Very Long Instruction Word) processors have begun to be considered as prime candidates for code compression, given not only their inherent large instruction words but also their appeal to the embedded DSP market.

One example of where code compression has reached the VLIW industry is in Atmel's Diopsis Dual Core DSP implementing a mAgic DSP VLIW core which uses a method of built-in dynamic program decompression [3, 18]. Compressed program code is fed to dynamic program decompression devices (dyprodes) which produce the uncompressed code and this is seamlessly executed. Another advantage of using code compression is that program bus size can be reduced as a result of the smaller instruction word size. This is used to the Diopsis' advantage.

Code compression efficiency is widely defined [4, 12, 15, 19] as the ratio between the compressed program size and the original program size. That is, the smaller the compression ratio, the better the compression. Compression ratio can depend on the size of the original compiler output. Our previous work has found that the smallest overall sizes after compression are obtained when the smallest possible compiler build is used, even though other builds give better compression ratios [20].

In this paper, we present a new compression scheme and investigate its performance. We have taken selected benchmarks from the Spec2000 [2] and the Mediabench [1] benchmark suites, and built them for the Texas Instruments TMS320c6x [21] and the Intel Itanium [9] as representatives of the VLIW/EPIC processor range.

The remainder of this paper is organized as follows. Section 2 presents background and related work in this field. Section 3 describes the compression scheme used and Section 4 outlines results from applying the compression scheme. Section 5 includes a discussion and comparison of results and Section 6 contains conclusions and further work.

## 2. RELATED WORK

The area of text or data compression is a mature one, but code compression dates from 1992, when Wolfe and Channin first published a paper on a Compressed Code RISC Processor (CCRP) [22]. VLIW code compression is an even more recent field with papers published in only the last few years. Code compression is a separate field of study given that many data compression based schemes are inapplicable to program code, where branch targets and function entry points need to be decompressed on demand.

### 2.1 Code Compression on RISC processors

The paper by Wolfe and Channin [22] suggested a CCRP to compress code and used a 'code-expanding instruction cache', such that the decompression could be transparent to the processor. By using a compression technique that did not give consideration to branch targets and function beginnings, extra hardware was required to fetch addresses. Their design used a Line Address Table (LAT) to map original addresses into compressed code addresses.

Lefurgy et al presented dictionary compression in [13] where all unique instructions are recorded in an 'instruction table' and each instruction is replaced by an index into the table. They also present a selective version in [14]. Liao et al offered a dictionary compression scheme based on set-covering in [16] which looks at substrings that occur frequently. Lekatsas presented a semi-adaptive dictionary compression scheme in [15] which generated new opcodes for instructions appearing frequently. Some software/compiler methods have also been presented in [5, 6, 14].

### 2.2 Code Compression on VLIW processors

Code compression techniques have also been applied to VLIW processors. Nam et al [17] achieved average compression ratios of 63%-71% using a dictionary compression method and compared the difference in performance of "identical" (whole instructions words) and "isomorphic" (split into opcode/operand fields) instruction word encoding schemes. Ishiura and Yamaguchi [10] investigated code compression based on Automatic Field Partitioning, achieving compression ratios of 46-60%. They reduced the problem of compressing code to the problem of finding the field partitioning that yields the smallest compression ratio. Larin and Conte [11] compared code compression methods and a tailored encoding of the Instruction Set Architecture. The tailored ISA method produced new code at 64% of the original code size, though at a much smaller cost to decoding hardware than standard compression.

Xie et al. [23, 25] used a reduced-precision arithmetic coding technique combined with a Markov model and applied it to similar systems with different sized sub-blocks. The 16-byte sub-block scheme yields the best compression rates at 67.3% – 69.7%. Xie et al. also present a Tunstall-based memory-less variable-to-fixed encoding scheme and an improved Markov variable-to-fixed algorithm in [24]. 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.

Prakash et al [19] present a dictionary based encoding scheme that divides instructions into two 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 from one of the dictionary vectors by a Hamming distance of at most 1 (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 differ by only one bit will not require both vectors to be stored in the dictionary. One of the two vectors is stored and the other merely references the stored vector and points out which bit needs to be toggled. Average compression ratios of 78.6% including Line Addressing Table are reported. Although some attempt is made to investigate 32-bit vectors, the dictionary selection method they used did not appear to give compression ratios as good as the 16-bit scheme. Their scheme also uses different dictionaries for each sub-block of 2048 bytes as opposed to using one dictionary for the whole program.

### 2.3 Previous Implementations of Code Compression

One successful encoding scheme, commercially used in the PowerPC 405, is the CodePack scheme [7]. The CodePack encoding scheme follows an algorithm analogous to a piece-wise Huffman scheme [8] where the most frequent symbols are assigned smaller codewords. Here, the 16-bit half-words are assigned a two or three bit tag which denotes which 'class' they belong to, differentiated by the tag and then how long the codeword is. CodePack has a reported performance of an overall program size "reduction" of 35-40% [7] (i.e. a compression ratio of 60-65%). CodePack uses variable-length encoding and requires the use of a mapping table to calculate the new address of a given instruction. Lefurgy et al provide further optimisation and enhancement suggestions for a machine with CodePack in [12].

A second example of the implementation of code compression is the Atmel Diopsis example mentioned earlier [3, 18]. This VLIW code compression architecture claims a 2X to 3X compression of code (33 to 50% compression ratio) whereby 128-bit instruction words are compressed to an average of 50 bits per instruction word. This shows the advantage of an integrated code compression and instruction set architecture if designed together from the start.

In most cases, designing a totally new processor complete with integrated code compression and instruction set architecture is beyond the scope (not to mention budget!) of many embedded applications. Instead, research has tended to concentrate on code compression systems that are software-based or where hardware need only be altered slightly in order to achieve a saving of program size (moderate, but a saving nonetheless). An example of where a slight alteration of hardware is possible would be the inclusion of a decompression engine next to a processor core in an ASIC embedded design. In this case, the program to be run on the processor of choice can be compiled and compressed before loading.

## 3. ENCODING SCHEME

The encoding scheme presented in this paper is based on the appropriate selection of dictionary vectors such that all program vectors are at most a specified Hamming distance from a dictionary vector. Bit toggling information is used to accurately restore original code. This scheme is similar to the 16-bit version from [19] where only vectors differing by one bit were considered. Instead, our scheme considers 32-bit vectors and was trialed with Hamming distance upper limits from 1 to 8. Furthermore, we consider multiple dictionary selection methods and offer a stream-based compression method for parallel decompression.

The algorithm is divided into the four steps described in the following subsections. A decoder is required in the hardware to decode the uncompressed instructions and is outlined in Section 3.5.

### 3.1 File Input and Dictionary Construction (First Input Pass)

The first pass in the encoding scheme is equivalent to most dictionary compression schemes. The benchmark to be compressed is read in, one 32-bit vector at a time, and a frequency distribution of all the used vector space is constructed. This histogram-like structure (containing elements from the dictionary) is used in the subsequent compression steps.

### 3.2 Reduced Dictionary Selection (First Dictionary Pass)

The purpose of this pass is to select from the dictionary, a subset of vectors (called the reduced dictionary) such that all original dictionary vectors are at most a set Hamming distance from any one of the reduced dictionary vectors. The purpose of this dictionary-subset selection is to allow for a smaller dictionary, and include information for bit-toggles where the vectors differ in the replacement codewords.

The benchmark programs were profiled for 32-bit vector space usage and three reduced dictionary selection methods were applied – they are described below. They were tested for up to set Hamming distance upper limits ranging from 1 to 7.

#### *3.2.1 Frequency Selection Method*

This method of selecting vectors for inclusion in the reduced dictionary chooses vectors based on their frequencies and continually adds the most frequent vectors until all the vectors in the original dictionary are 'covered' by being at most a set maximum Hamming distance from any of the reduced dictionary vectors. The aim of this method is to include vectors into the reduced dictionary that are very frequent in the original program, thus incorporating a higher number of "zero Hamming distance" entries. This means that fewer bit toggle location fields will be required during compression (see Section 3.4).

#### *3.2.2 Maximum Span Selection Method*

This method finds, for each vector in the dictionary, the total number of other dictionary vectors that are up to a set maximum Hamming distance from it. The vector that spans the most other vectors is chosen and placed in the reduced dictionary. Then, this method discards all vectors in the dictionary that are the set Hamming distance or less from the chosen vector. Of the un-discarded vectors in the dictionary, the vector that spans the most of the remaining vectors is chosen and the process repeats again until all vectors are discarded from the original dictionary. The aim of this method is to reduce the number of vectors needed in the reduced dictionary.

#### *3.2.3 Combination of Frequency and Spanning Method*

This dictionary selection method attempts to combine the best from both of the previous algorithms. It chooses the most frequent vector in the dictionary and places it in the reduced dictionary. Then, it discards all vectors in the dictionary that are the set Hamming distance or less from the chosen vector. Once again, the most frequent vector from the remaining vectors is chosen and the process repeats until all dictionary vectors are covered by the given set Hamming distance.

### 3.3 Reduced Dictionary Fill and Codeword Assignment (Second Dictionary Pass)

The reduced dictionary is analyzed and filled with further vectors such that the bits required for the indexing of the reduced dictionary is unchanged. Essentially, this fills it with vectors from the original dictionary that did not already exist in the reduced dictionary, up to the next power of 2 so that there is no wasted indexing space. In all three dictionary selection methods, the extra filling stage takes the most frequent vectors that are not already in the reduced dictionary, as this method will reduced the number of toggle locations more. The indices into the reduced dictionary serve as codewords for the compression step.

### 3.4 Compression Application (Final Input Pass)

The compression scheme is applied by converting each 32-bit vector into compressed code. The compressed code comprises a codeword (determined in the last step), a set number of bits to denote the number of toggles and up to 7 sets of 5-bit toggle locations. An example of this is shown in Figure 1.

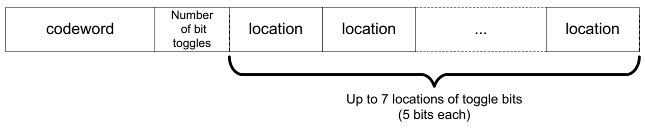

**Figure 1 - Format of Compressed Program Code**

Compressed program code is inserted serially in place of the original code with one exception. To make decoding easier, possible branch targets are aligned at byte boundaries and as a result, some padding is needed at the end of any byte preceding a target location. This padding and the Line Address Table (LAT) – described in Section 3.5 – are part of the overhead associated with this encoding scheme.

### 3.5 Decompression Engine Design

A decompression unit is required to decompress the instructions 'on the fly' and feed them to the CPU. The standard dictionary scheme uses a dictionary as a lookup table, where the compressed instruction acts as an index into the lookup table and the output of the table is the uncompressed instruction.

Our scheme works in a similar fashion, with the codeword from the compressed instruction acting as an index into the reduced dictionary lookup table, and the extra bits in the compressed instruction determining which bits (if any) to toggle from the lookup table output. A block diagram of the dictionary and the bit toggling hardware required for a code compression scheme with a Hamming distance upper limit of 3 is given in Figure 2.

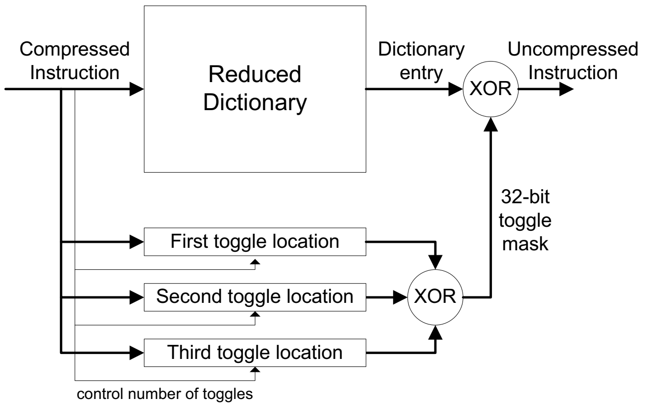

**Figure 2 - Block Diagram of the Decompression Unit**

Because our scheme is a variable length one, we must consider the need for a referencing table of some sort such that instruction locations (such as branch targets) can be retrieved. For this, we have used a LAT similar to [19], however only branch targets are included in the table. The block diagram of this LAT hardware is given in Figure 3. Furthermore, to ensure the branch targets were byte aligned, padding was required at the end of the previous instruction of every target.

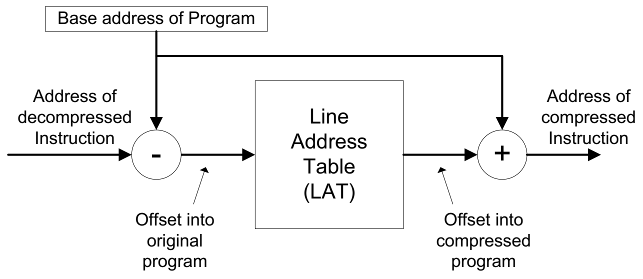
Base address of Program
Address of decompressed Instruction
-
Line Address Table (LAT)
+
Address of compressed Instruction
Offset into original program
Offset into compressed program

**Figure 3 - Block Diagram of Line Address Table**

## 3.6 Stream Encoding

The main problem with the serial decompression of variable-length codes is that performance is affected. In particular, one fetch packet (which consisted of four and eight 32-bit vectors in the processors investigated) can consist of many vectors which are normally fetched simultaneously. If 8 such 32-bit vectors are to be serially decompressed, then the latency associated with 8 sets of dictionary retrievals and bit togglings could be detrimental to performance.

In a bid to parallelize the decompression of the compressed code and avoid the serial decompression latency, the option of compressing the information into streams is trialed. This implementation divides the instruction fetch packet into 32-bit streams and decompression is applied to the program code in a given stream rather than the whole program code. Smaller, individual tables and separate decompressors are required for each stream.

## 4. RESULTS

Benchmarks were taken from both the Spec2000 [2] and the Mediabench [1] benchmark suites. These were built for two targets, the Texas Instruments TMS320c6x [21] using the TI Code Composer compiler and the Intel Itanium [9] using gcc.

Benchmarks taken from the Mediabench suite included adpcm (**rawc**- and **rawd-audio**), g721 (**g721enc** and **g721dec**), **epic** (and **unepic**), mpeg (**mpeg2enc** and **mpeg2dec**) and jpeg (**cjpeg** and **djpeg**). Benchmarks taken from the Spec2000 suite included **mcf**, **art**, **equake**, **parser**, **ammp**, **twolf** and **mesa**.

In both processor cases, the benchmarks were built with every optimization level, and the smallest possible build was used. In most cases, this corresponded with the -ms3 and -o3 flags for the TI compiler, and the -Os flag for all gcc builds.

Compression ratio is an accurate measurement to compare the different versions of this compression scheme, because they are all applied to the same original files (hence starting size will be the same for any benchmark).

The first issue investigated was that of the dictionary selection methods. Compression ratio was found to be very dependent on the selection method thus results are presented for each selection technique in comparison to a standard dictionary compression. The standard scheme places all unique vectors found in the program code in the dictionary and an index is used instead of the original vector. An example of its application is given in [13]. In essence, the 'normal' dictionary compression method is a method that tolerates no bit toggles (and as a result requires no extra information) and can be likened to our method with a Hamming distance upper limit of 0 where the 'reduced' dictionary is identical to the original dictionary.

Compression ratios in the following sections include the compressed code, dictionary and LAT sizes. Dictionary sizes are taken from the number of reduced unique entries required to cover the entire code, and the LAT sizes are derived from the number of branch target locations. Average compression ratios across all benchmarks tested are reported.

## 4.1 Frequency Selection Results

The Frequency Selection method returned compression ratios worse than the standard dictionary compression (left column in Figure 4) for Hamming distance limits of 7 and under, although compression ratios were improving as the Hamming distance limit was raised. This prompted the investigation of larger Hamming distance upper limits and upper limits of up to 16 were investigated. In fact, the results suggested that a Hamming distance upper limit of 10 would give best results.

The results at this Hamming distance returned average compression ratios of 73.1%. This compression scheme uses the fact that although Hamming distances of up to 10 may be allowed, a large portion of the program code is a small Hamming distance

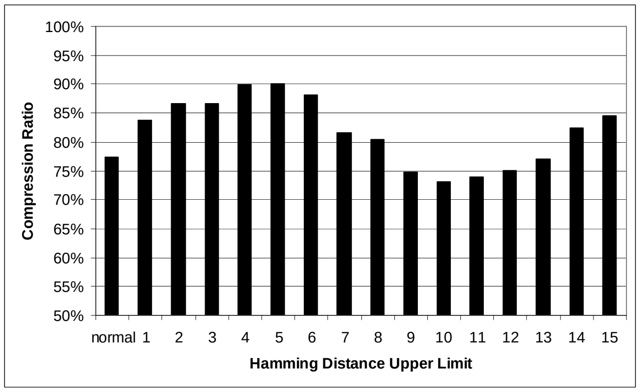

**Figure 4 - Average Compression Ratios for Frequency Selection**

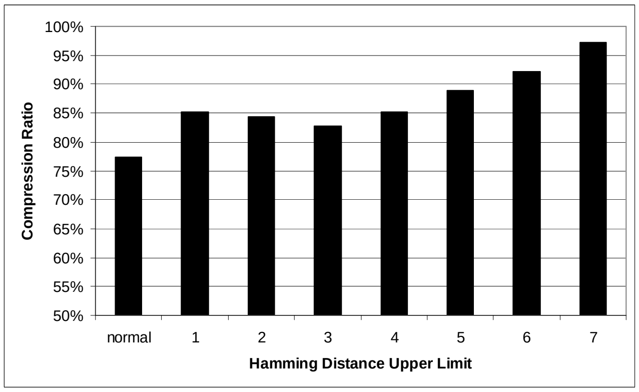

**Figure 5 – Average Compression Ratios for Maximum Span Selection**

from a dictionary vector, because more frequent vectors are added first.

To examine the relative frequencies of different Hamming distances, an example benchmark is profiled. Here, the **djpeg** benchmark, built for the TI TMS320c6700, has been broken down into how many instructions are a given Hamming distance from a dictionary entry, with the upper limit set to 10. The reason compression is achieved is due to just over half of the program's vectors being found in the dictionary even though the number of dictionary entries is low. This is because this algorithm greedily includes the most frequent vectors first.

**Table 1 – Hamming Distance Frequencies for Frequency Method Example**

| Hamming Distance | Number of Program Instructions (%) | |
|---|---|---|
| 0 | 15772 | (54.7%) |
| 1 | 2909 | (10.1%) |
| 2 | 3166 | (11.0%) |
| 3 | 2548 | (8.8%) |
| 4 | 1787 | (6.2%) |
| 5 | 1184 | (4.1%) |
| 6 | 796 | (2.8%) |
| 7 | 470 | (1.6%) |
| 8 | 179 | (0.6%) |
| 9 | 30 | (0.1%) |
| 10 | 15 | (0.1%) |
| *Total Instructions*: | *28856* | |
| *Unique Instructions:* | *11805* | |
| *Dictionary Entries:* | *2048* | |

The main issue arising from this frequency-based scheme is that the length of the compressed instruction could escalate out of hand. In the example case, the codeword length was $\log_2(2048) =$ 11 bits. For a Hamming distance upper limit of 10, 4 'bit-toggle' bits would be required (see Figure 1) and furthermore, up to 10 sets of 5-bit locations toggle locations could be required (as in the case of the 15 instructions shown to be a Hamming distance of 10 from a dictionary entry in Table 1). This means the "compressed" representation would actually be an expansion and would be 65 bits long. The codeword length would only increase with larger programs. Such a large "compressed" instruction (instead of the 32-bit vector without compression) could add significant changes to the instruction fetching, retrieving and decoding hardware.

## 4.2 Maximum Span Selection Results

In order to keep the Hamming distance upper limit to a more manageable level, the maximum spanning method was trialed. The aim in this method was to include in the reduced dictionary, vectors that covered more of the rest of the vectors in the program code, so that with the same number of dictionary vectors, a larger set of program vectors were covered. The best results were obtained at a Hamming distance upper limit of 3 as shown in Figure 5. This was due to fully utilizing the toggle bit number bit-space

Unfortunately, this method did not take into account any information about how frequent the chosen vectors were, and as a result, none of the Hamming distance upper limits investigated achieved compression ratios better than standard dictionary compression. Compression ratios for this method were around 82%.

## 4.3 Combined Frequency and Spanning Results

The combined frequency and spanning selection method was investigated in order to combine the higher frequencies of smaller compressed instructions from the first selection method and the larger set of program vectors covered by vectors in the reduced dictionary from the second selection method.

The results in Figures 6 and 7 showed that, similar to the maximum span method, selecting the Hamming distance upper limit of 3 yielded the best results in this combined dictionary selection method. In the compression for the TI TMS320C6x program code, the compression scheme using the Hamming distance upper limit of 3 outperformed the normal dictionary compression method by an average of 8%, though for some benchmarks, this was as high as 13%. Compression ratios ranged from 72.1% to 80.3%.

The main contributing factor found in experiments concerning the Hamming distance upper limit of 3, was that the reduced dictionary needed was about one eighth the size of the original dictionary. This meant on average, 3 bits were saved from each and every instruction, with only some of the instructions requiring

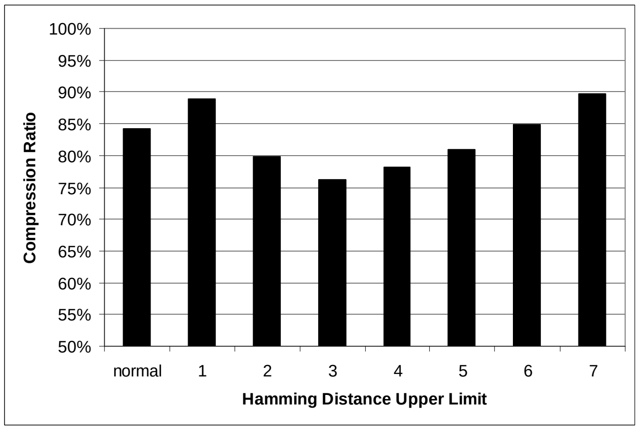

**Figure 6 - Average Compression Ratios for Combined Selection for the TI TMS320C6x**

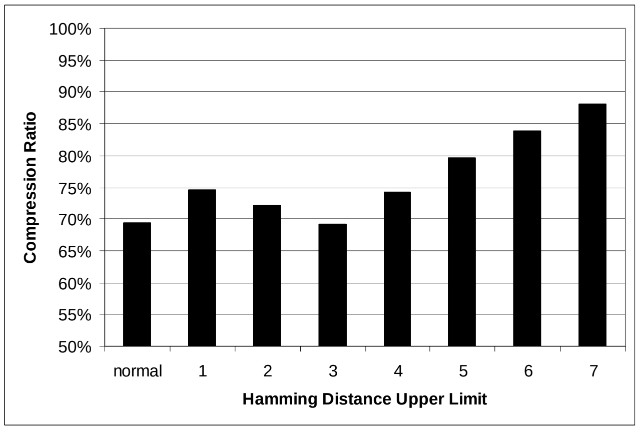

**Figure 7 - Average Compression Ratios for Combined Selection for the Intel Itanium**

extra bit-toggling information. Furthermore, as the dictionary itself was much reduced, this contributed to an overall reduction.

Experimental results for the Intel Itanium program code were not as successful. The Hamming distance limit of 3 was once again the best compression ratio obtained, however this was on average only less than 1% better than standard dictionary decompression. In some cases, the compression ratio was worse. Possible reasons for this are discussed below.

Once again, the number of vectors that were lower Hamming distances from a dictionary entry determined how good the compression would be. The same example benchmark from Section 4.1 (**djpeg**) was profiled under the combined dictionary selection method, with the results in Table 2. Although the number of instructions found in the dictionary was less than in the Frequency method (54.7% - 35.6% = 19.1% less), the Hamming distance upper limit ensured that not as many toggle fields were needed.

**Table 2 – Hamming Distance Frequencies for Combined Method Example**

| Hamming Distance | Number of Program Instructions (%) | |
|---|---|---|
| 0 | 10278 | (35.6 %) |
| 1 | 6992 | (24.2 %) |
| 2 | 9109 | (31.6 %) |
| 3 | 2477 | (8.6 %) |
| *Total Instructions*: | *28856* | |
| *Unique Instructions:* | *11805* | |
| *Dictionary Entries:* | *4096* | |

Figure 8 shows a subset of benchmarks with their original size (white), normal dictionary compressed size (light grey) and reduced dictionary compressed size (dark). For each benchmark, the first group of three bars corresponds to the TI TMS320C6x program code, and the second 3 bars (with diagonal hatching) correspond to the Intel Itanium program code.

## 4.4 Stream Encoding Results

The idea of stream encoding was trialed in order to decompress multiple streams of program code at once, limiting the added delay attributed to the decompression unit. Our study focused on the TI TMS320C6x program code, as results from the previous section showed that Intel Itanium program code did not seem to compress well under 32-bit vectors.

The results obtained in this investigation suggested that compression in streams suited the larger benchmarks. As the program code was divided into 8 smaller streams each one eighth the size of the original code, the sizes of these streams for some of the smaller benchmarks were too small to give good compression results. However, the larger benchmarks responded well, with benchmarks larger than 200kb only adding on average, 4% on the reduced dictionary results to give compression ratios around 79.4%. Figure 9 shows the selected benchmarks with their original code size, reduced dictionary compressed size and the same compression algorithm applied to streams. In the smaller benchmarks, the overhead in the streamed version almost negated the compression, however the larger files still returned good compression results.

## 5. DISCUSSION

For the Hamming-distance based reduced-dictionary compression scheme presented in this paper, the compression ratio has been found to be very dependent on the dictionary selection method. A vector selection method which considers both the frequency of vectors and the codeword-space coverage of vectors outperformed either method considered independently. This combined dictionary selection method achieved its best results with a Hamming distance upper limit of 3 – it outperformed standard dictionary compression on TI TMS320C6x program code by an average of 8% to give an average compression ratios of 76.2% when applied to the smallest compiler builds. Like all code-compressions schemes, this comes at the cost of additional decoding hardware.

When applied to the Intel Itanium program code, our scheme only resulted in a negligible change, and in some cases led to a worse compression ratio than normal dictionary compression. This is likely to be because our approach considered fixed-size code vectors of 32 bits. TI TMS320C6x program code is made up of 32 bit instructions – which corresponded to the code vectors considered; however, the 128-bit Itanium code bundles contain three 41-bit instructions which did not align well with the 32-bit vectors. It is suggested that other vector lengths could be

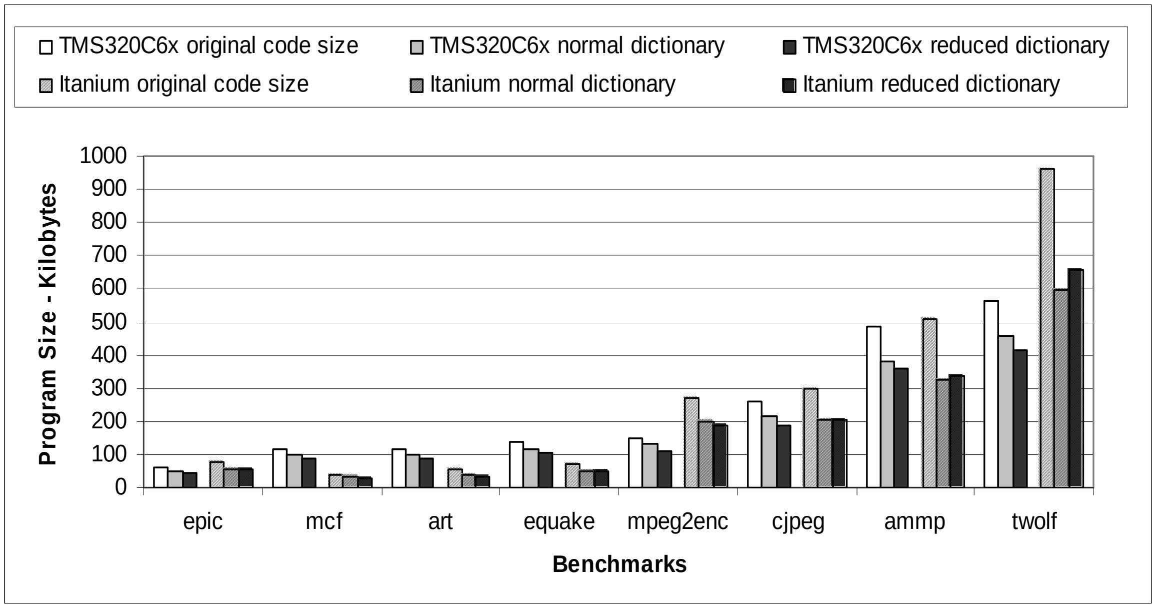

**Figure 8 - Relative Sizes of Program code before and after compression**

examined for the Itanium program code to determine if this type of compression scheme could be applicable under different vector lengths.

An investigation into parallel compression showed that dividing the program into 32-bit parallel streams returned an average compression ratio of 79.4% for programs larger than 200kb. This approach enables parallel decompression of instruction streams within a VLIW instruction word with only a small overhead in compression performance. For small programs, however, there is little advantage to this approach.

## 6. CONCLUSIONS AND FURTHER WORK

This paper has presented a VLIW code compression technique based on vector Hamming distances. Dictionary vectors are selected such that all program vectors are at most a specified maximum Hamming distance from a dictionary vector. Bit toggling information is used to restore the original vector.

A dictionary vector selection method which considered both vector frequency as well as maximum coverage achieved better results than just considering vector frequency or vector coverage independently. This method, with a Hamming distance upper-limit of 3, was found to outperform standard dictionary compression on TI TMS320C6x program code by an average of 8%, giving compression ratios of 72.1% to 80.3% when applied to the smallest compiler builds.

An investigation into parallel compression showed that dividing the program into 32-bit parallel streams returned an average compression ratio of 79.4% for files larger than 200kb.

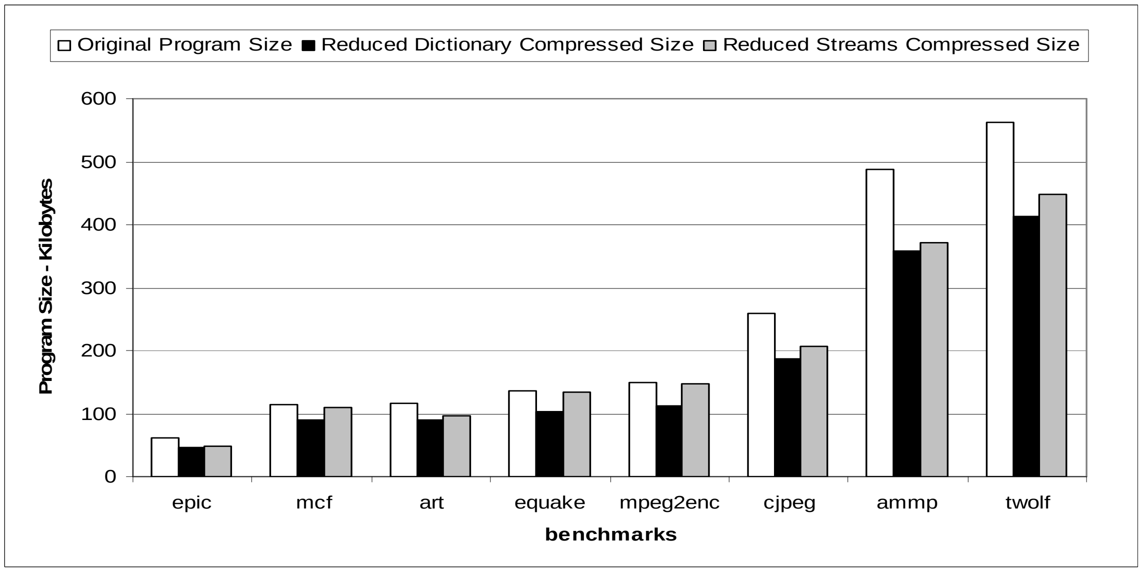

**Figure 9 - Relative Sizes of Program code before and after stream compression**

Further work is suggested in a number of areas. First, compiler techniques such as register renaming could be used to select registers whose binary representations are small Hamming distances from one another. If the compiler was aware of the Hamming distance upper limit of the subsequent code compression applied, it would be possible to output program code such that the 32-bit instructions used as vectors could be grouped more efficiently and separated by Hamming distances within the compression scheme's upper limit.

Second, it is proposed to consider other dictionary selection methods that are not greedy (all methods presented in this paper selected reduced dictionary entries based on the maximum current gain only). Other options could be investigated, such as the use of dictionary vectors that are not limited to the vectors found in the program.

Third, the selection of codewords associated with each reduced dictionary entry could be investigated. In this paper, the codewords used were a fixed length, with a variable length tail appended to denote how many and which bits to toggle. A variable scheme could also be applied to the codeword field such that codewords would be smaller for more frequently accessed dictionary entries and longer for infrequent vectors. This could be achieved by applying either a Huffman [8]-like or CodePack [7]-like scheme.

## 7. REFERENCES

[1] *Mediabench Benchmarks*, 1997, accessed 2003, http://cares.icsl.ucla.edu/MediaBench/

[2] *SPEC CPU2000 Benchmarks*, 2000, accessed 2003, http://www.specbench.org/cpu2000/

[3] Atmel-Corporation, *AT572D740 Summary (Datasheet)*, 2004, accessed 2004, http://www.atmel.com/dyn/resources/prod_documents/7001s.pdf

[4] 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.

[5] 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.

[6] 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.

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

[8] D. A. Huffman, "A method for the constuction of minimum redundancy codes," *Proceedings of the IRE*, vol. 4D, pp. 1098-1101, 1952.

[9] Intel, *Intel Itanium Architecture Software Developer's Manual*, Revision 2.1, 2002, accessed 2004, http://www.intel.com/design/itanium/manuals/iiasdmanual.htm

[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, "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.

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

[16] S. Liao, S. Devadas, and K. Keutzer, "A text-compression-based method for code size minimization in embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, pp. 12-38, 1999.

[17] S. J. 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.

[18] P. S. Paolucci, "Apparatus and Method for Dynamic Program Decompression." United States, Filed: 2002.

[19] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant, "A Simple and Fast Scheme for Code Compression for VLIW processors," in *Proceedings DCC 2003. Data Compression Conference*, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2003, pp. 444.

[20] M. Ros and P. Sutton, "Compiler optimization and ordering effects on VLIW code compression," in *Proceedings of the international conference on Compilers, Architectures and Synthesis for embedded systems*. San Jose: ACM Press, 2003, pp. 95--103.

[21] Texas-Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000, accessed 2004, http://focus.ti.com/lit/ug/spru189f/spru189f.pdf

[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] Y. 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] Y. 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] Y. 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.