Experiments with a new dictionary based code-compression tool on a VLIW processor

J. Prakash **C. Sandeep** **Priti Shankar*** **Y.N. Srikant**

IISc-CSA-TR-2004-5
http://archive.csa.iisc.ernet.in/TR/2004/5/

Computer Science and Automation
Indian Institute of Science, India

July 2004

*Contact Author: priti@csa.iisc.ernet.in

Experiments with a new dictionary based code-compression tool on a VLIW processor

J. Prakash **C. Sandeep** **Priti Shankar*** **Y.N. Srikant**

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012
India
email: prakashj@yahoo-mc.com {priti, srikant }@csa.iisc.ernet.in

Abstract

Reducing program size is a desirable goal in the design of an embedded system, since memory occupies a relatively large silicon area. Instruction compression schemes equipped with random decompression facilities are one answer to this problem. For VLIW architectures having flexible instruction formats, this is a particularly challenging problem beacuse instructions do not have an uniform structure and hence offer limited scope for discovering repeated patterns that can be exploited for compression. We propose a new dictionary based scheme here and evaluate its compression and decompression performance for the TMS62X architecture by carrying out detailed simulations.

keywords code compression, embedded systems, VLIW processors.

1 Introduction

In an embedded system, perhaps the most restricted resource is the memory. The memory area in a silicon chip can take up from three to seven times the area that is required for the processor. Consequently, a decrease in program size effectively reduces the cost, size and power consumption. A decrease in the program size also reduces the instruction cache misses and hence provides higher instruction bandwidth[21]. There are two major approaches to the reduction in code size. The first uses techniques to effectively reduce code size at compile time[13]. The second approach compresses small blocks of instructions with decompression being performed dynamically during program execution[8, 10, 9, 11, 16, 2, 3, 4, 5, 6, 7, 14]. The scheme we propose in this paper employs a strategy of the second kind. There are two important factors that affect the

*Contact Author: priti@csa.iisc.ernet.in

choice of a compression scheme of this kind. Firstly, random access to compressed blocks of code must be ensured. Since compressed code is decompressed during program execution, jump, branch and call statements that alter the flow of control must be directed to small compressed blocks within which the target instructions are located without having to decompress the whole segment. Secondly the decompression scheme must be very fast, as it is on-line and its overhead directly affects program execution time. Compression, however can be done offline and hence one can exploit complex strategies to generate good models or dictionaries to be used in the decompression stage.

Several new architectures are VLIW architectures with flexible instruction formats. The present technology revolution has put DSP processors in a strong position due to their targetted wide application space. Current estimates put a 30% growth in the DSP market[20] the fastest in the entire semiconductor market, for the next five years. These processors are finding application in all kinds of embedded applications like wireless communications, satellite communications, automobiles for cruise control, anti-lock braking, mp3 and other music players, digital cameras, digital video and every kind and form of digital communication. It would be desirable to have a general purpose scheme which is applicable to such processors, which has a fast decompression algorithm. This paper proposes such a scheme and evaluates it on programs for the TMS320C6X processor, which is a fixed point, VLIW DSP processor developed by Texas Instruments[1]. The advantage of this scheme over other schemes is its fast decompression.

Our results show that fairly good compression ratios[1] ranging from 70% to 78% are achieved on benchmarks. We have proposed two schemes for decompression. The first decompresses about 12 bits per cycle; the second decompresses about 50 bits per cycle with about 2% loss in compression ratio. We have also evaluated the overheads of the decompression with respect to the execution time of some programs. We observe that with an instruction cache of size 64K the compressed programs incur an execution time overhead of about 14%.

2 The New Scheme

We view the space of instructions as consisting of two half-spaces, the set of all first halves of instructions and the set of second halves. An element of each half-space is a vector consisting of sixteen bits as each instruction is 32 bits long. The basic idea of the scheme is to find a small set of vectors for each half-space which we term the *dictionary*, such that most of the vectors in the half-space are at a small Hamming distance from some entry in the dictionary. (The Hamming distance between two vectors is the number of positions in which they differ). We will then code each vector as an index into the dictionary and the positions where it differs from the dictionary entry. A vector in the subspace is covered by the dictionary if it is sufficiently close to some entry of the dictionary. The distance that is permitted depends on the choice of the size of the dictionary and the number of bits needed to encode the positions of disagreement. Clearly if the vectors to be encoded are each of length n and the dictionary is of size D then the maximum distance d that can be permitted is such that $d\ log(n) + log(D) + c < n$ where c is a small constant

[1] compression ratio is ratio of compressed program size to uncompressed program size

which encodes the value of d as well as whether the vector is compressed or not. It is possible that there may be a small number of vectors that are at large Hamming distance from every entry in the dictionary and are therefore left uncompressed and transmitted as such. The next choice to be made involves the block size for which a chosen dictionary is valid. One could have a single dictionary for the whole subspace, with the advantage that the dictionary overheads are negligible. On the other hand one could take advantage of the fact the the fraction of the 2^{16} words that occurs in small files is small and hence compute separate dictionaries for sub-blocks of the instruction stream and use a different dictionary for each sub-block. Experiments revealed that a sub-block of 2048 bytes was effective from both the dictionary coverage viewpoint and from the dictionary overhead considerations. (The overhead is about 3%). The coverage is about 60% on the average, that is about 40% of the vectors are at a Hamming distance of two or greater than every dictionary entry. The encoding format is shown in Figure 1. Here "flips" refers to positions of disagreement with the dictionary entry.

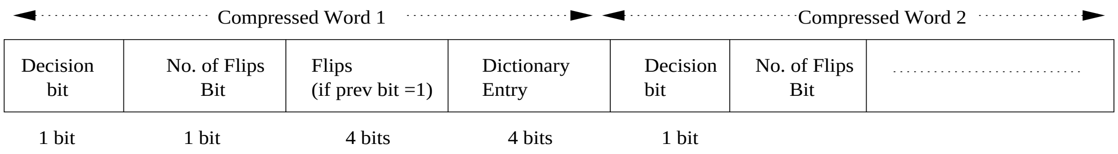

Figure 1: Encoding format for our scheme

2.1 The TI Processor

The TMS320C6200 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 archictecture processor and has two clusters of four functional units each. It fetches eight instructions each of length 32 bits at a time. Each such instruction packet may be made up of several exection packets. All instructions in an execute packet are 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 ie. 256 bits or 32 bytes.

2.2 Construction of the Dictionary

We first describe the scheme we employ to construct the dictionary. Firstly, we note that with a dictionary size of sixteen entries for each subspace consisting of length sixteen vectors, there is virtually no benefit in encoding vectors which are at a Hamming distance more than one from a dictionary entry. This is because for a Hamming distance of two we will need eight bits to encode the positions of disagreement, four bits for the dictionary index, two bits for encoding the number of disagreements and one bit for specifying whether the vector has been compressed or not, ie. a total of 15 bits and hence we save only one bit here. Further there is an additional overhead of one bit per block of sixteen bits for encoding the number of disagreements over the case when we allow only one disagreement. We therefore restrict the set of coded vectors to those that are at a Hamming distance of at most one from some dictionary entry.

Having decided on the maximum number of disagreements that we can tolerate–we call them "flips" here, we next need to search for vectors such that spheres of Hamming radius one around them do not intersect, and cover as much of the corresponding instruction subspace as possible. (A sphere of Hamming radius d around a vector contains all vectors at Hamming distance d or less from the vector.) The final scheme uses a dictionary of size 64 bytes (each eight bits long) for each block of 2048 bytes.

We use the following clustering algorithms to approximate a good choice for the centers of spheres with the constraint that the numbers of such centers does not exceed sixteen.

2.3 A Greedy Algorithm

The first is a greedy algorithm, which is a slight modification of the leader based clustering algorithm. We first count the frequency with which each vector appears in the stream to be compressed, (we are actually looking at frequencies of half instructions) and tabulate the vectors in decreasing order of frequency. We choose the first dictionary entry to be the vector occurring with highest frequency. Subsequent choices are made by selecting at each stage, the highest frequency vector satisfying the condition that it is at a Hamming distance at least two from every vector previously chosen to be in the dictionary. One could argue that a distance three is preferable to ensure that every vector is covered by a distinct dictionary entry and to increase the dictionary coverage. However this tighter condition on the selection of vectors causes the rejection of higher frequency vectors and our experiments showed that the compression achieved was smaller. The process continues until sixteen vectors have been chosen.

2.4 A Random Walk Clustering Algorithm

Harel and Koren[22] describe a clustering algorithm based on random walks. In this algorithm the nodes are clustered in such a way that a node in a cluster reaches any node in the same cluster with a very high probability and every node in another cluster with low probability. Initially a 2-dimensional matrix is created in which entry (i,j) denotes the probability of reaching node j from node i in one step. The entry (i,j) in the k^{th} power of this matrix denotes the probability of reaching node j from node i in k steps. The sum of all powers up to k gives a matrix in which element (i,j) denotes the probability of reaching node j from node i in at most k steps. A similarity measure of the nodes which is just the cosine product of the vectors of each node formed as described earlier, is then computed. The edges are now weighted with this similarity value. Edges with low similarity values are the separating edges. By cutting these separating edges, a set of clusters is determined.

We have adapted this algorithm to solve the problem at hand. The weight of an edge is the product of the inverse of the Hamming distance between its two end vertices and the savings obtained by including either of the vertices in the dictionary. This ensures that we give preference to edges connecting vertices of higher frequencies over edges which connect vertices of low frequencies. Only edges between vertices which are at a Hamming distance less than a threshold are included. If a 16-bit scheme which permits for at most one error is considered, there will be edges only between vertices which are at a Hamming distance of one from each other. Subsequently, for each vertex we divide the weight of the edge leaving it by the sum of the weights of all edges leaving the vertex and assign this number to the probability of that edge leaving the vertex.

The random walk algorithm is then applied on the weighted graph so constructed. The edges were cut based on some threshold. We experimented with two schemes, one in which the threshold was set based on a sequence of runs of the algorithm and another in which the threshold was set to be the average of all edge weights. The latter gave better performance.

The leader of each cluster was then selected by a brute force approach where the node assigned to be leader resulted in maximum savings.

This clustering method gave results which were similar to those using the greedy approach. The clustering algorithm does not include the notion of a leader, and the edge cutting strategy is oblivious to this need. Further the nodes in a single cluster are not always within the chosen threshold of Hamming distance from the leader.

2.5 Decompression

The decompression strategy has the advantage that it is done word by word and not bit by bit. Clearly it involves checking the compression bit to check if any decompression is to be performed, then decoding the number of flips and their positions and finally flipping the erroneous bits. While this can be done quite fast in software two possible hardware schemes are proposed.

Scheme 1: In this scheme each compressed block can be decompressed in approximately 21 cycles. The compressed instruction format and the decompression hardware block diagram is as shown in Figure 2. Here the decompression hardware can be divided into stages and the stages can be pipelined. We need three stages in the pipeline and one more to get the block which is not part of the pipeline. Since it is pipelined the number of cycles required will be $16 + n + 2$ where $n = 3$ is the number of stages of the pipeline. Hence we require 21 cycles.

Scheme 2: In this scheme each compressed block can be decompressed in approximately 5 cycles. The compressed instruction format and the decompression hardware block diagram is as shown in Figure 3. For decompression we need to store a bit to indicate the number of disagreements even if the block is not compressible, thereby reducing the compression. Our tests indicate that this overhead averages to 2%.

3 Related Work

Debray et.al[13] have proposed a scheme for code compaction based on compiler techniques. They use various techniques like code factoring, dead code elimination, unreachable code elimination and abstracting register saves and restores. Their tool Squeeze acheives an average compression ratio of 70% on the Compaq Alpha processor.

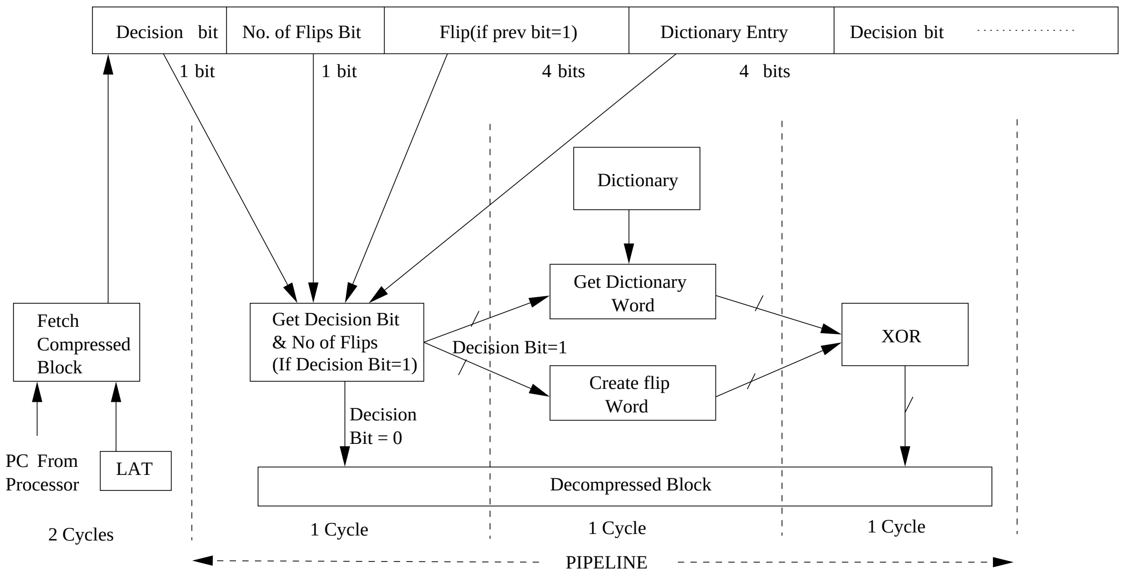

Figure 2: Hardware for serial decompression

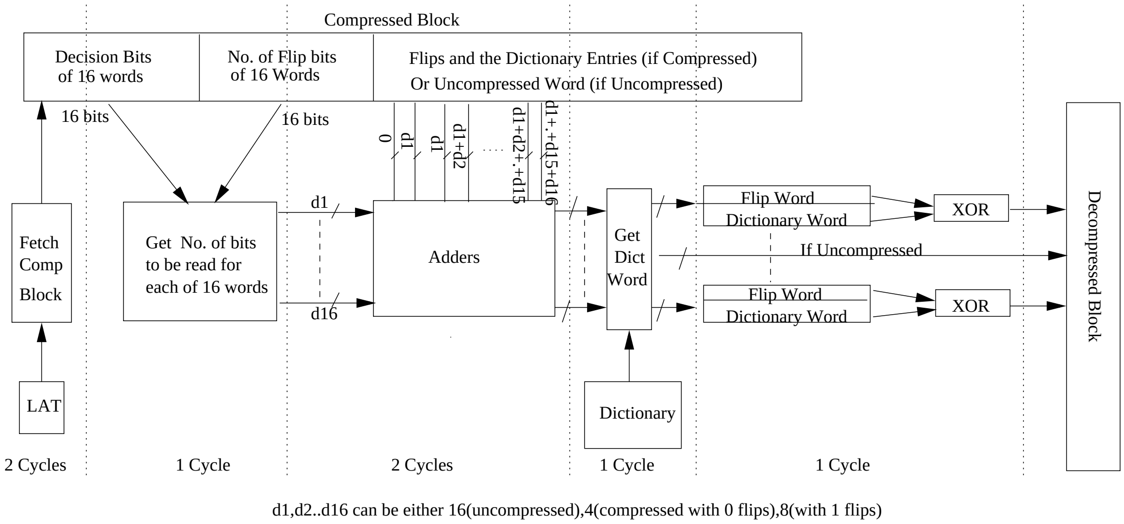

Figure 3: Hardware for parallel decompression

The first code compression scheme for embedded processors was proposed by Wolfe and Chanin[2]. The main memory contains compressed code and between the main memory and the instruction cache is a decompression unit which decompresses a cache line whenever there is a cache miss. Byte based Huffman codes are used for compression and compression ratios of 73% are reported in Kozuch and Wolfe[15] for MIPS code. Benes et.al[16] 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

Access Table(LAT) which maps 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.

Liao et.al[3] used the idea of replacing frequently appearing groups of instructions by a call to a dictionary which stores each of these sequences once. Compression ratios of 84% to 82% have been reported in their study.

Bird and Mudge[4] propose a dictionary-based method which outperforms byte-based Huffman coding.Commonly occuring 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. In a scheme suited for DSP Lefurgy and Mudge[11] use a fixed length encoding by putting each unique instruction word in the program in an instruction table, and replace 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. However absolute branch addresses will change by an amount that is easily computed because of the fixed length encoding. The overhead of a LAT is thus avoided.

Ishiura et al.[6] split up VLIW instruction words into fields each of which are 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. [7] also propose dictionary based compression for a fixed format VLIW processor and report compression ratios from 63% to 71%.

Lekatsas and Wolf[8] 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 high 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[12]. 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.

4 Implementation and Results

The unit for compression was chosen to be a fetch packet, that is a sequence of eight instructions comprising 256 bits. Jumps in the code were encoded using the Line Address Table(LAT) proposed by Wolfe and Chanin[2]. This maps each encoded block address to a compressed code block address. A jump to an address in uncompressed code has to uncompress the compressed block in which it is located. We found that the overhead of the LAT ranged from 3% to 6%. Figures 4 and 5 gives the size of the compressed code including the dictionary size for a whole range of programs from media bench[17] and the TI benchmarks[1]. An interesting observation is that the compression using this scheme(without the overhead of the LAT) is about the same as that obtained by using

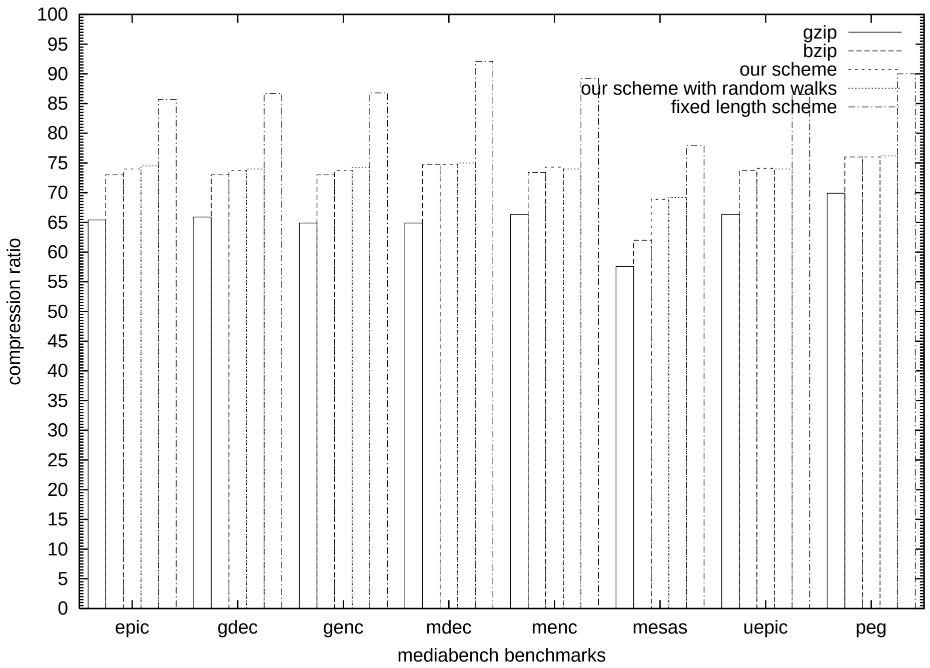

Figure 4: Performance for Media Benchmarks

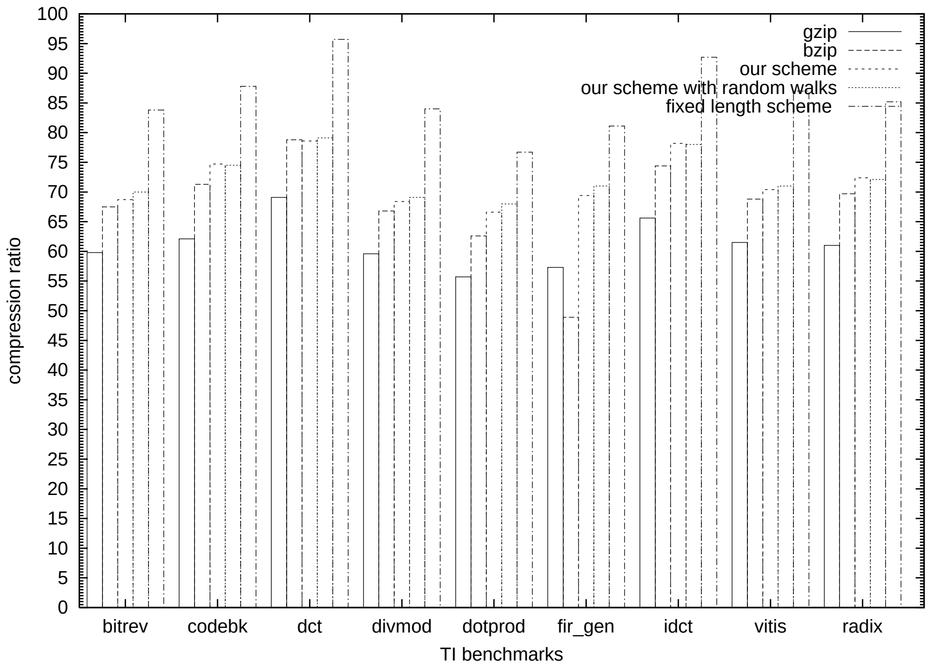

Figure 5: Performance for TI Benchmarks

bzip on a block size of 2048 bits(the programs have been divided into blocks of 2048 bytes and each such block is compressed separately usig gzip and bzip and the aggregate

Benchmarks	Without lat	With lat	Fixed length	gzip	bzip	Xie et al.
TI	72	76	86	61.3	76.6	70-82.5
Media Bench	73.6	78.6	86	65	73	70-82.5

Table 1: Average compression ratios

compression ratio is calculated). It was also observed that an average of 37% of the program words went uncompressed. The programs were compiled using the TI CCS compiler. We compared the benefits of this scheme with a scheme that uses fixed length encoding and a single dictionary that has an entry for each unique instruction. Thus each instruction is coded by an index into this entry. (Such a scheme was proposed by Lefurgy and Mudge[11] and has the advantage that no LAT is required.) The average compression of our scheme with the LAT overhead was seen to be 78.6%(for media benchmarks) and 76%(for TI benchmarks) in comparison with a compression ratio of 86%(for both benchmarks) using the fixed length coding scheme. Also there are advantages to having a small dictionary, as it can be stored in the cache and thus faster access to it is possible at decompression time. Table 1 shows the average compression ratios of our scheme in comparison with the fixed length scheme of Bird and Mudge [4] and the scheme by Xie et al.[23].

We find that on an average 63% of the total number of words are compressed. Of this 63%, 43% (averaging over all benchmarks) of the words are at a Hamming distance one and hence compressed. This explains the difference in performance between our scheme and that of Bird and Mudge.

5 Estimation of Decompression Overhead

An open source freely available C-model TMS320C62X simulator [19] has been used for the simulations. Some changes have been made to the hardware so that it executes compressed instructions. A decompression unit has been added between the EMIF(External Memory Interface) and the program cache. This unit decompressed instructions stored in the external memory and places them in the program cache. The instruction fetch unit fetches the decompressed instruction from the program cache. When the instruction requested for is not available in the cache, the cache line that contains this instruction is fetched from the external memory, decompressed and stored in the cache. In the pipelined scheme of decompression the decompression unit pipeline is augmented with the processor pipeline. This pipeline fetches the next cache line and decompressed it while the processor is executing the current cache line. There will be a pipeline stall occasionally due to a jump or a call. In the parallel scheme of decompression the decompression unit decompresses one fetch packet at a time and sends it to the fetch and decode units. In all the programs that were experimented with, though each fetch packet has eight instructions and can thus have a parallelism of eight, the parallelism in the programs was less than three instructions on an average. A parallelism of only three in a fetch packet, results in three three execution cycles for that packet. The next fetch packet could be fetched and decompressed during this time hence the decompression time could be partially masked by this. Further whenever the fetching of the packets crosses the block boundary(2K bytes in our case) there is overhead of loading the LAT and the dictionary for the block, and this has been taken into account.

A direct mapped cache configuration has been tried out. The cache line size was chosen to be the size of one instruction fetch packet. The cache size was varied from 256 bytes to 64K bytes(2K fetch packets). Simulation results reveal an average decompression overhead of 43% for a cache size of 256 bytes and 14% for a cache size of 64K bytes. Figure 6 shows the execution times(normalized to uncompressed execution time) for benchmark programs.

The executable files are in the COFF executable file format[18]. Two named sections, one for the dictionary and one for the LAT, have been included.

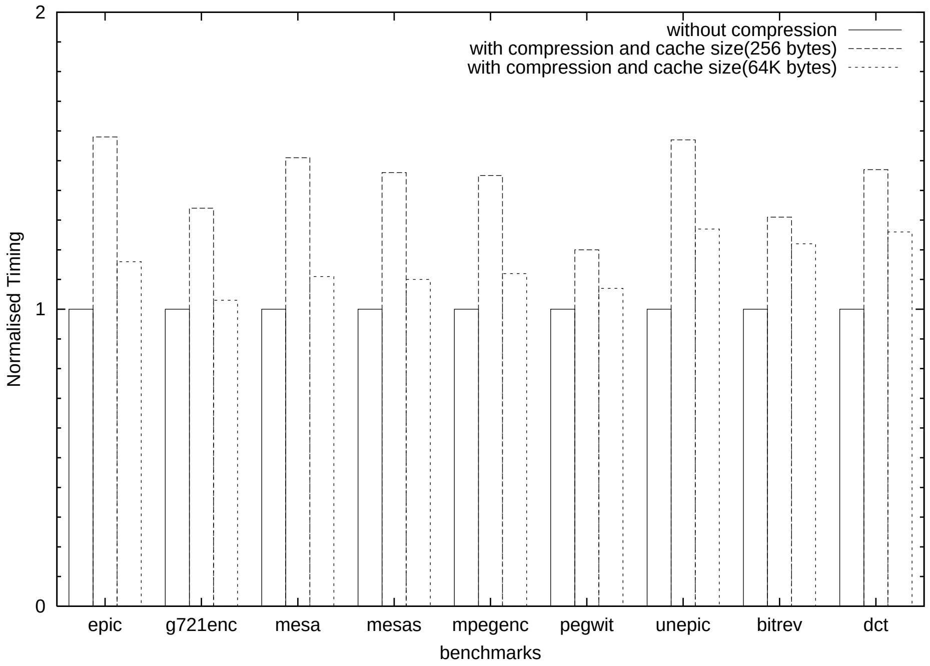

Figure 6: Decompression Overheads

6 Conclusions and Future Work

In this paper we show a new dictionary based scheme to compress executable code with a very simple and fast decompression hardware. We present detailed results of using the scheme for TI TMS320C62xx architecture using MediaBench and TI benchmarks. We find that the compression ratio is around 73.6% without LAT and around 78.6% with the LAT. Also an average decompression overhead of 14% with a cache size of 64K bytes was observed.

Future work can concentrate on trying reduce the LAT size. One possible approach being to store only those entries in the LAT that are targets of some branch. Better ways of designing dictionaries and faster decompression schemes also need to be investigated.

References

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

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

[3] 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*

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

[5] 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.*

[6] 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.*

[7] 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*

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

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

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

[11] 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.*

[12] 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.*

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

[14] IBM, CodePack PowerPC Code Compression Utility User's Manual, Version 3.0, *IBM, 1998.*

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

[16] M.Benes, S.M.Nowick, A.Wolfe. A fast Asynchronous Huffman Decoderfor Compressed-Code Embedded Processors, *Proceedings of International Symposium on AdvancedResearch in Asynchronous Circuits and Systems, September, 1998.*

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

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

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

[20] Why TI focusses on DSP?, *http://www.ti.com/corp/docs/investor/dsp/why.htm.*

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

[22] David Harel,Yehuda Koren. On Clustering Using Random Walks *Foundations in Software Technology and Theoretical Computer Science(FSTTCS) 2001: 18-41.*

[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.*