Clustered Multi-dictionary Code Compression Method for Portable Medical Electronic Systems

Ji Tu, Xiangyi Yu, Yangyang Li, Liu Yuan, Meng Li
Innovation Center, China Academy of Electronics and Information Technology
Beijing, 100041, China
jtu@csdslab.net

Weiwei Fang
School of Computer and Information Technology, Beijing Jiaotong University
Beijing, 100044, China
wwfang@bjtu.edu.cn

Abstract—**Currently, the demand on portable medical electronic systems are increasing, for they provide services and information to both patients and doctors that the traditional medical methods cannot achieve. However, the development of portable medical electronic systems has many limitations, i.e., one has to find a sweet spot among performance, power consumption, size, and cost. For instance, if one only increases the memory of a system, not only does the cost go up, the power consumption also goes up, meanwhile the standby time goes down. In some cases, the hardware size goes up as well. One way of satisfying these constraints while retaining the design and functionality is to compress executable code and data as much as possible. In this paper, a novel clustered multi-dictionary code compression method is proposed to effectively reduce the memory size by replacing the most common codes with shorter codeword. The codes are clustered according to their repeating times. Each cluster is compressed with a different dictionary to make the codeword length different. By pairing clusters and dictionaries with the highest entropy, the compression efficiency becomes the best. Theoretical analysis and experimental results show that this method can achieve significant compression effect. The code of MiBench benchmark compiled under ARM and MIPS instruction set architecture are compressed with this method and the code size decreases by 50%. Aside from high compression ratio, our method also provides relatively fast encoding and very fast decoding.**

Keywords-portable medical electronic systems; code compression; multi-dictionary; cluster

I. INTRODUCTION

Over the past decade, as people are becoming more health conscious and physician-independent, the use of portable medical electronic systems has become increasingly widespread, and the designing trend of such systems is to become smaller, lower power consumption, and more intelligent [1]-[3]. From multipurpose handheld PDAs to dedicated real-time control systems, the variety of the applications also increases [4]. Some applications, such as remote patient and elderly monitoring can be life-saving [5]. Nevertheless, the design criteria of portable medical electronic systems are not easily satisfied. One has to consider functionality, performance, and standby time within the scope of size and cost. Take memory as an example: if it is too small, the system either cannot run or has to abandon some functionality and performance; if selecting a bigger memory, the system might have better functionality and performance, but the power consumption rises, the cost increases, and the whole system fails the design criteria. This is why many portable medical electronic systems operate under tight memory space constraints. To provide quality healthcare at reasonable costs, there is a driving need to extract as much memory space efficiency and performance from the available resources as possible [6]-[10]. One practical way of alleviate those problems is by code compression [11]. Compressing the program binary code and decompressing it at runtime helps us better utilize the limited memory space in portable medical electronic systems.

The reduced binary size of a compressed application has two important features which can improve its performance [12]-[14]. If the compressed code is stored in the main memory, filling up the cache line on a cache miss will require fewer cycles on average, in effect reducing the average latency to fetch an instruction block from the memory. Moreover, placing the compressed code in the cache means that it can hold more instructions, hence increasing the effective cache size and causing a reduction in the miss rate [15]-[16].

There are many methods for code compressing. Nevertheless, we present only those that use dictionary based code compression methods. In 2005, Menon and Shankar divide the instructions into different logical classes and to build multiple dictionaries for them. The size and the number of multiple dictionaries are fixed for a given processor and are determined by the partitioning algorithm which works over the instruction set architecture supplied as input. The frequently occurring unique instruction segments are inserted into the dictionaries and the instructions are encoded as pointers to the respective entries [17]. In 2009, Collin and Brorsson use a two-level dictionary design and are capable of handling compression of both individual instructions and code sequences of 2-16 instructions. The two dictionaries are in separate pipeline stages and work together to decompress sequences and instructions [18]. In 2013, Azevedo Dias proposes a new code compression method using Huffman-based multi-level dictionary which applies two compression techniques. A single dictionary is divided into two levels and it is shared by both techniques [19].

In this paper, we propose a novel clustered multi-dictionary code compression method, abbreviated as CMD (Clustered Multi-Dictionary code compression). CMD separates the different code compiled under ARM and MIPS instruction set architecture (ISA) into different clusters. The codes that only occur one or two times are grouped into a cluster. They will not be compressed. The codes with higher frequencies are in other clusters. Each of those clusters has a dictionary which entries the codes within that cluster and maps them to codewords. The codeword is stored in external memory as compressed binary executable code. We use shorter codeword for the cluster which their codes occur more frequently than the code in other clusters. This design has the best entropy for the entire code set. The best number of clusters corresponds to the highest compression ratio, which is found out by an adaptive cluster algorithm.

II. CMD Code Compression Method

In most of the existing multi-dictionary code compression methods, the dictionaries size is not choosing effectively. A novel clustered multi-dictionary approach (called by us as CMD) is to cluster the code set and compressing each code subset using its own dictionary and the codeword can be decode in parallel.

A. CMD Code Compression Model

Assuming the code set ***H*** contains *m* codes, and each code is *n* bits long. There are *k* different codes among code set ***G*** = {g_1, g_2... g_k}. The repeat times set ***Q*** = { q_1, q_2 ... q_k } ($q_1>q_2>\ldots>q_k$). The code g_i repeats q_i times. The overall number of bits of code set ***H*** is ***A(H)***, and

$$\mathrm{A(H)} = mn = \sum_{i=1}^{k} q_i n \qquad (1)$$

When using CMD method, we first cluster the code set ***H*** and obtain *v* different subset ***C***= { c_1, c_2, ..., c_v }. Assuming j_1, j_2, ..., j_v are the number of elements (code) in c_1, c_2, ..., c_v; which, j_i equals to 2 to the power of i. Assuming p_1, p_2, ..., p_v are the percentage of the number of codes in code set c_i over the number of codes in code set ***H***. For the first *v*-1 subset, we use dictionary-based code compression method. Each dictionary entry is encoded with fixed length index for each subset c_i. For the last subset c_v, we don't compress the code. A flag which is $log_2 v$ bits long is inserted into the dictionary index to indicate which subset it's pointing to. Thus, the codeword is consisting of the index and the flag. Assuming d_1, d_2, ..., d_v are the codeword length of the subsets.

Then, let ***H'*** denote the CMD compressed code set, the overall number of bits of ***H'*** is ***A(H')***, and

$$\mathrm{A(H')} = \sum_{i=1}^{v} \sum_{j=1}^{j_i} q_j d_i + \sum_{i=1}^{v-1} j_i n \qquad (2)$$

which, $(j_1+\ldots j_{v-1})n$ is the sum of bits of the dictionary entries.

From (1)-(2), we can conclude that, the compression ratio (*CR*) of CMD is,

$$CR = A(H')/\,A(H) \qquad (3)$$

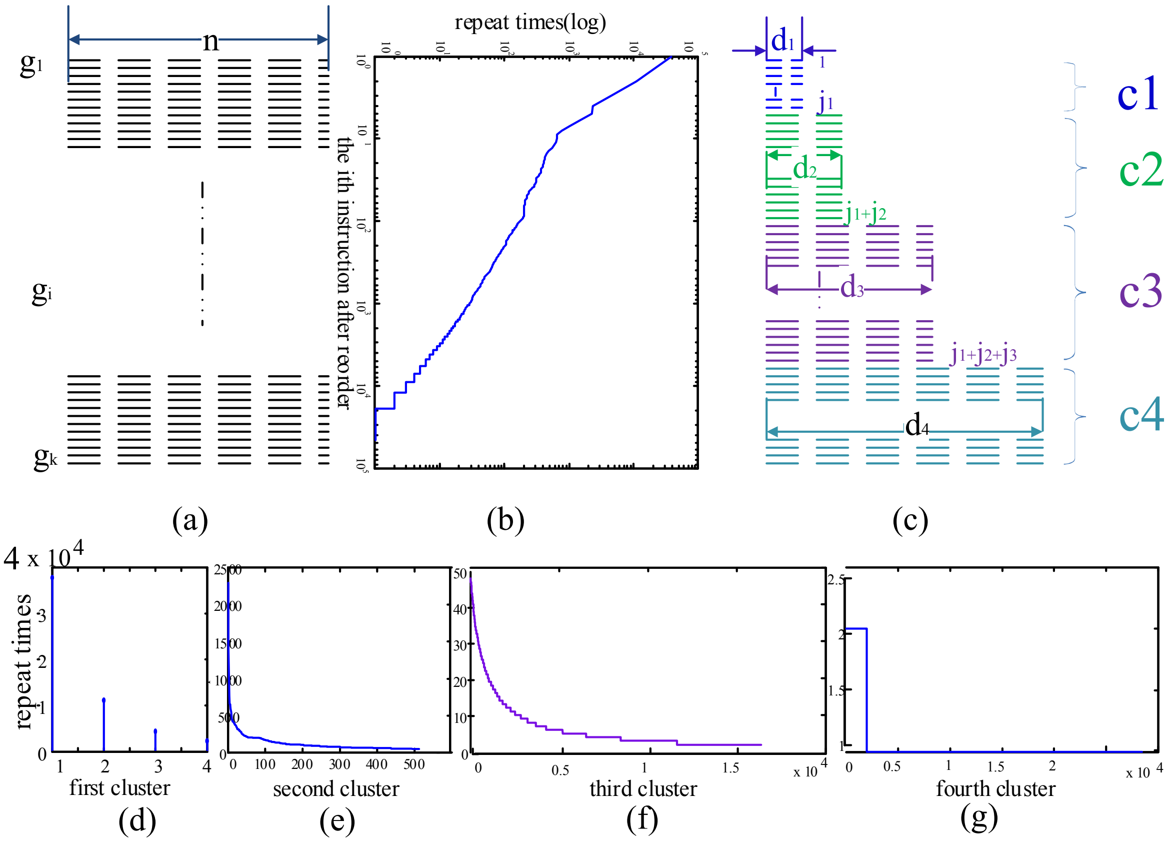

Figure 1. Process of CMD code compression of "basicmath" program.

B. CMD Code Compression Process

To illustrate the compression process, we take the program "basicmath" from MiBench [20] for an example. Suppose Figure 1 (a) is the code set *G*, there are *k* different codes. The *k* different codes are reordered by their occurring frequencies, from high to low, so that the previous code occurs more frequent than or equal to the following code. The logarithm value of the occurrence times of each different code is shown in Figure 1 (b). There are four code repeat more than six hundred times. The logarithm value of the occurrence times of the last hundreds of thousands of codes is zero, which indicates that these codes only occur one time in the whole binary code set.

We clustered the code of "basicmath" benchmark program from MiBench into four different clusters for instance, as shown is Figure 1(c). The most frequently occurred codes use the shorter codeword to point to. These codeword is stored as dictionary entries. When the program is executing, CPU fetch codeword from memory, then look up the dictionary by the codeword to get the dictionary entry. The dictionary entry is the real code to be executed in CPU and it's been store into the cache for late repeat use.

Figure 1 (d-g) are the code repeat times of each cluster of "basicmath" executable binary code. The clusters are chosen by *Algorithm 1* which will be described in the next section.

Figure 1 (d) is the code repeat times in the first cluster. There are four codes in the first cluster and the first code repeat 37787 times by our statistics in experiment. The occurrence times of the first codes are 37787 by our statistics. And the first code is 0x00. The second code is 0Xc2 and it repeats 11141 times. The first cluster has four codes and the code size of the first cluster account for 21.76% of the total uncompressed code size. After compression, the first cluster account for 2.72% of the total uncompressed code size.

Figure 1 (e) is the code repeat times in the second cluster. There are 512 codes in the second cluster and the code repeat times range from 50 to 2500. Most of the code repeat times are less than 500. After compression, the second cluster account for 8.64% of the total uncompressed code size. If use Huffman encode method, the compression ratio will be improved, but the decode system will be more complex.

Figure 1 (f) is the code repeat times in the third cluster. There are 2^{14} different codes in this cluster and the code repeat times range from 3 to 49. Most of the code repeat times are less than10. The code in first three cluster are stored in dictionary and there codeword are stored in external memory as compressed binary executable code.

Figure 1 (g) is the code repeat times in the last cluster. There are 28507 codes and the code repeat times range from 1 to 2. Most of the codes repeat only one times. The codes in the last cluster are not compressed. After add two flag bits, the code is stored in external memory as compressed binary executable code. So the compressed code size in this cluster is bigger than uncompressed code size. The compression efficiency of the first three clusters is significant enough to compensate the negative compression effect of the last cluster.

C. Clustered Algorithm for CMD

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, and bioinformatics [21]. We introduce the cluster algorithm to the code compression area.

The following is the pseudo-code algorithm to find out the best clusters related to our CMD compression method.

```
Algorithm 1: Adaptive Cluster Selection
// $x_i$ is a power of 2, that is, $x_i \in \{2^r \mid r=1,2,3,\cdots,32\}$
for( $j_1 = 2 \rightarrow x_1$){
    for( $j_2 = j_1+1 \rightarrow x_2$){
        ……
            for( $j_{v-1} = j_{v-2}+1 \rightarrow x_{v-1}$){
                Calculate the $CR$ with equation (3);
                Record Min($CR, j_1, j_2, \ldots, j_{v-1}$);
            }
        ……
    }
}
```

The value of $j_1, j_2, \ldots, j_{v-1}$ in the record of Min($CR, j_1, j_2, \ldots, j_{v-1}$) are the parameters automatically selected by computer to choose the best clusters which will obtain the best compression efficiency.

D. Decode Logic

Codes are stored on the external memory in compressed form and transparently decompressed while moved into RAM at run time. The main characteristic of code compression in a system with run-time decompression is the need to have random access in decompression. The decoding technique is based on the system proposed in [2] to attack this address mapping problem. Figure 2 shows an overview of CMD decode system embedded in processors. The compressed code is placed in the main memory and/or in the instruction cache, thus increasing their effective sizes by enabling them to hold more number of instructions.

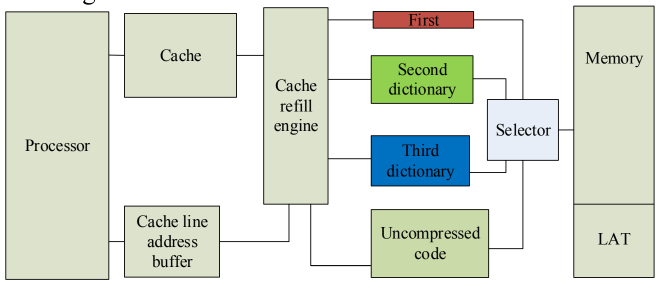

Figure 2. Process of CMD code compression of "basicmath" program.

During runtime, compressed code is fetched, decompressed and sent to the next memory level or to the processor. Decompression introduces certain overhead which may increase the number of cycles for each fetch, which in turn may reduce the program's execution rate.

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

To expedite the decoding process, the decompression logic is customized for efficiency, depending on the choice of dictionaries used. We store the dictionary entries and its codeword in the external memory. When the program is executed by processor, the dictionary entries are loaded into the internal memory of the processor. The processor uses the codeword to find out the dictionary entries in the right cluster. The dictionary entries are the uncompressed instruction and it is sent to the cache for the processor to execute. All instructions are fetched through the cache. Compression schemes used in optimizing code size can be complex and their dynamic decompression can have significant decompression latency. Line address table (LAT) is stored in the external memory and it maps the address of the codes before and after compress. The cache line address buffer can significantly reduce the times of read external memory by store the frequently used line address.

III. Simulations and Analysis

For analysis and validation of our CMD method we have performed simulations with fifteen programs from MiBench, they are: basicmath(large), bitcount, qsort(small), susan, djpeg, dijkstra (large), string search (large), ADPCM (encode, decode)-audio, GSM (encode, decode)-toast, crc32, fft, patricia. We chose these applications so that all six categories of MiBench are considered.

We provide experimental results of Mibench benchmarks compiled for ARM and MIPS ISA, and the code size are reduced significantly. The simulation is implemented in C language and the decode logic is simulated by MODELSIM using verilog language.

A. Code Compression for ARM

The statistics of the experiments are summarized in table 1. They are the results of 4 clusters using CMD. The first column is the file name of MiBench programs. The second column is the compression ratio of CMD method for ARM ISA, ranging from 50% to 52%. The third column is the proportions of multiple dictionaries size over total code size. The larger the dictionary size, the more time will need to look up the proper code. The fourth column is the number of different codes in first cluster. In program "basicmath", *k0* is 2, means that, there are 2^2 different codes. The fifth column is the number of different codes in second cluster. There are 2^9 different codes in the second cluster of "basicmath". From the sixth column we can see that there are 2^{14} different codes. The seventh column is the number of uncompressed codes in the fourth cluster. There are 28507 different codes in the last cluster of "basicmath".

The following column of c_0 to c_3 is the percentage of four clusters code size after compressed over the total code size which is uncompressed. The next four columns of p_0 to p_3 is the percentage of four clusters code size before compressed over the total code size which is uncompressed.

TABLE I. Experiments results of Mibench program compiled for ARM

SrcFile	CR	dict	k0	k1	k2	k3	c0	c1	c2	c3	p0	p1	p2	p3	Time(s)
basicmath	0.5147	0.0661	2	9	14	28507	0.0272	0.0864	0.2038	0.1312	0.2175	0.2514	0.4076	0.1234	27.30
bitcnts	0.5071	0.0563	2	9	14	32275	0.0272	0.0856	0.2032	0.1347	0.2177	0.2491	0.4064	0.1268	36.29
qsort	0.5113	0.0589	2	9	14	31081	0.0269	0.0842	0.2064	0.1348	0.2152	0.2450	0.4128	0.1269	32.59
susan	0.5186	0.0623	2	9	14	31621	0.0267	0.0842	0.2044	0.1409	0.2135	0.2450	0.4089	0.1326	31.70
djpeg	0.5204	0.0556	2	9	14	35828	0.0252	0.0855	0.2044	0.1498	0.2015	0.2488	0.4088	0.1410	40.45
cjpeg	0.5178	0.0695	2	9	14	26591	0.0265	0.0863	0.2092	0.1263	0.2118	0.2509	0.4184	0.1189	27.94
search	0.5124	0.0610	2	9	14	29774	0.0267	0.0847	0.2076	0.1323	0.2138	0.2465	0.4152	0.1245	32.48
dijkstra	0.5106	0.0602	2	9	14	29955	0.0269	0.0852	0.2064	0.1319	0.2153	0.2477	0.4128	0.1242	30.72
patricia	0.5152	0.0706	2	9	14	25829	0.0270	0.0875	0.2065	0.1236	0.2162	0.2545	0.4130	0.1163	26.09
crc	0.5168	0.0698	2	9	14	26838	0.0268	0.0868	0.2068	0.1266	0.2147	0.2524	0.4137	0.1192	28.37
bf	0.5136	0.0675	2	9	14	27262	0.0270	0.0870	0.2060	0.1261	0.2162	0.2531	0.4120	0.1186	29.59
fft	0.5157	0.0709	2	9	14	25673	0.0269	0.0873	0.2076	0.1229	0.2150	0.2541	0.4153	0.1157	26.35
rawcaudio	0.5157	0.0709	2	9	14	25669	0.0269	0.0873	0.2076	0.1229	0.2150	0.2541	0.4152	0.1157	24.68
rawdaudio	0.5156	0.0612	2	9	14	30992	0.0258	0.0886	0.2038	0.1362	0.2066	0.2577	0.4075	0.1282	32.34
toast	0.5156	0.0612	2	9	14	30992	0.0258	0.0886	0.2038	0.1362	0.2066	0.2577	0.4075	0.1282	31.28
untoast	0.5147	0.0661	2	9	14	28507	0.0272	0.0864	0.2038	0.1312	0.2175	0.2514	0.4076	0.1234	27.30

By comparison between c_i and p_i, we found that, the first cluster is compressed more efficiently than the other clusters. The size of the fourth cluster is smaller than the other clusters. The shorter the codeword of a cluster, the better compression efficiency the cluster obtained. The fourth cluster is increased after compression. Due to the high compression efficiency of the other three clusters, the final compression ratio is still significant.

The last column is the CMD execution time of MiBench program using a 2.4 GHz, Intel i5 dual core PC. Larger program needs more time to implement CMD method. The execution time of the program which named toast is almost half a minute, it is too long for programmer to compile the program code using a compiler. The CMD execution time has no relation with the decode system processor execution time. The decode system processor execution time is depending on the processor clock frequency, the code size, and the cache miss rate (the memory architecture).

The code sizes of these programs are less than 10 megabytes and they are compiled using the ARM cross compiler under the highest code optimization condition in gcc. The code size will larger when using the normal optimization command in gcc, meanwhile, the compression ratio will better when not use the highest code optimization command in gcc. The compression ratio will be better if we use the Huffman algorithm to encode the codeword for the first three clusters. But the decoding system will be much complex and the latency will be increased.

B. *Compression under Different Clusters*

Figure 3 is the experiment results which cluster the code set into 2, 4 and 8 clusters separately. The compression ratio is the best when clustering the code set H into 4 clusters. It's obvious to known that the compression effect of 3 clusters is worse than 4 clusters, because 3 clusters scheme also need two bit for each code to indicate which cluster the code is in. In the same way, the compression effect of 5 to 7 clusters is worse than 8 clusters. The compression ratio of 2 clusters is the worst among the three kind of different experiments. So we can conclude from Figure 3 that, clustering the code set H into 4 clusters is the best choice for the CMD code compression method of MiBench benchmark.

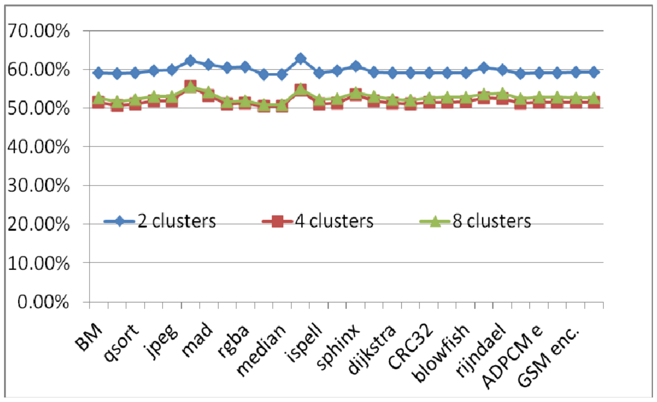

Figure 3. Compression ratio of different clusters from MiBench.

C. *Comparison between MIPS and ARM ISA*

Experiments for ARM and MIPS show that the compression ratio is better for ARM than for MIPS. The compression ratios of ARM are ranging from **50% to 52%**, while MIPS are ranging from **65% to 71%**, which can be seen from the stacked value generated by MATLAB

In Figure 4, CR means compression ratio, it consists of the compress ratio of each clusters. "Basicmath.arm" means the code is compiled by arm-linux-gcc, and the suffix of "mips" means the code is compiled by mipsel-linux-gcc. The parameters of c_0 to c_3 and p_0 to p_3 are the same as it means in table 1. The label 'd' indicates dictionary. The sum of p_0 to p_3 is 1 and the sum of c_0 to c_3 plus d is equal to the compression ratio. From Figure 4 we can see that, the compression efficiency of ARM code of the first and the second cluster are better than that of MIPS.

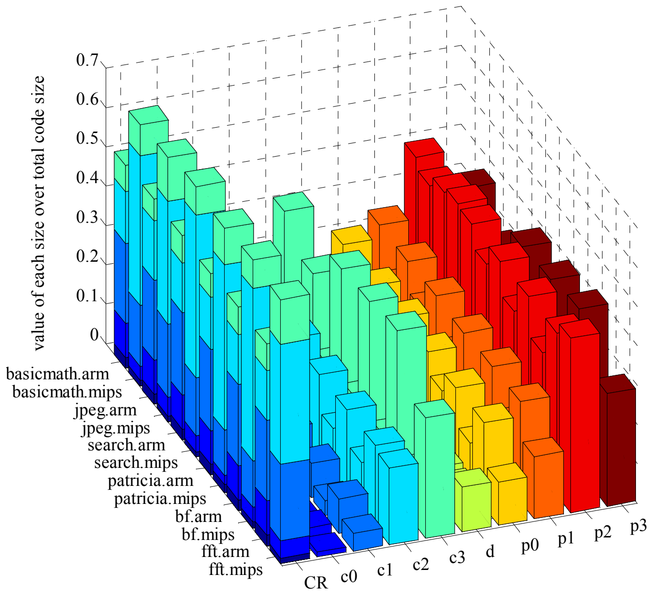

Figure 4. Comparisons between ARM and MIPS.

D. *Comparison between Different Methods*

In table 2, we present the CMD method compared with other methods found in previous literature. The entire MiBench program are compiled and then compressed. The average compression ratio for ARM and MIPS are 50 to 55% and 65 to 71%. Our decompression system for CMD supports one instruction per cycle delivery as well as parallel decompression. Our implementation in a real system is easy. CMD method is bandwidth and ISA independent. The compression ratio of CMD is only depended on the number of different code in the binary code set and the code set size. The compression ratio of our approach for ARM and MIPS is the best among all the methods shown in table 2. Thus, we conclude that CMD method is more suitable for embedded system applications.

The decode system hardware overhead and the latency delay for CMD code compression method are also better than the methods listed above. Our decode system can decode multiple dictionaries in parallel and doesn't need Huffman decoder in real system.

TABLE II. COMPRESSION WITH VARIOUS METHOD

Compression Method	**Target Architecture**	**Compression Ratio**	**Decompression Bandwidth**
Wolfe&Chanin[2]	MIPS	73%	8 bits
Dias & Moreno[9]	ARM	64-68%,	32 bits
Dias & Moreno[19]	ARM, MIPS	67%, 68%	32 bits
Menon & Shankar [17]	ARM, TMS320C62x	69-78%, 68-75%	32-64 bits
Garofalo[15]	ARM	64%	32 bits
Haider [16]	ARM	80%	32bits
Our Approach-**CMD**	ARM, MIPS	**50%-55%, 65%-71%**	32bits

IV. CONCLUSION

The clustered multi-dictionary code compression gives a practical solution to the code density problem of the portable medical electronic systems. Experiments show that for a given range of parameters one can achieve good program compression and maintain performance. In the case where traditional memory is used, performance actually increases. However, there are other aspects to be optimized, such as memory space requirements, power, real-time predictability, and reliability.

We are interested in the further research into LAT compaction methods and decoder implementations. The next step is to embed the CMD decode system into MIPS processor and download the Netlist into FPGA to observe the real performance. The matured technology will bring forward the functionality and performance of the portable medical electronic systems, better serving the community.

.

REFERENCES

[1] Misra, Subhas Chandra, and Sandip Bisui. "Adoption of Personalized Medicine: Towards Identifying Critical Changes." *Healthcare Engineering. Springer Singapore, 2017*. 55-57.

[2] Gomez, Rafael, and Anna Harrison. "Beyond wearables: experiences and trends in design of portable medical devices." *International Conference of Design, User Experience, and Usability*., pp 261-272, 2014.

[3] Kay, Misha, Jonathan Santos, and Marina Takane. "mHealth: New horizons for health through mobile technologies." *World Health Organization* 64.7 (2011): 66-71.

[4] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," *SIGMICRO Newsletter*, vol. 23, pp. 81-91, Dec. 1992.

[5] Boriani, Giuseppe, et al. "Effects of remote monitoring on clinical outcomes and use of healthcare resources in heart failure patients with biventricular defibrillators: results of the MORE - CARE multicentre randomized controlled trial." *European Journal of Heart Failure* 19.3 (2017): 416-425.

[6] Rüedi, P-F., et al. "Ultra low power microelectronics for wearable and medical devices." *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017

[7] Dhawan, Atam P., et al. "Current and future challenges in point-of-care technologies: A paradigm-shift in affordable global healthcare with personalized and preventive medicine." *IEEE journal of translational engineering in health and medicine 3* (2015): 1-10.

[8] Z.Yu, W.Han, Z.Jiying, "Depth map compression based on platelet coding and quadratic curve fitting," *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, vol., no., pp.1-4, 4-7 May 2014.

[9] W.R.A. Dias and E.D. Moreno, "Code Compression in ARM Portable medical electronic systems Using Multiple Dictionaries," in *2012 IEEE 15th International Conference on Computational Science and Engineering (CSE),* pp.209-214, 2012.

[10] O.Ozturk, H.Saputra, M.Kandemir, I.Kolcu, "Access pattern-based code compression for memory-constrained portable medical electronic systems," *Design, Automation and Test in Europe*, 2005. Proceedings, pp.882-887 Vol. 2, 7-11 March 2005.

[11] H.Kikuchi, T.Deguchi, M. Okuda, "Lossless compression of LogLuv32 hdr images by simple bitplane coding," *Picture Coding Symposium (PCS)*, 2013, pp.265-268, 8-11 Dec. 2013.

[12] I.Daribo, D.Florencio,C. Gene, "Arbitrarily Shaped Motion Prediction for Depth Video Compression Using Arithmetic Edge Coding," *Image Processing, IEEE Transactions on* , vol.23, no.11, pp.4696-4708, Nov. 2014.

[13] K. Shrivastava and P.Mishra, "Dual Code Compression for Portable medical electronic systems," VLSI Design (VLSI Design), *2011 24th International Conference on* , pp.177-182, 2-7 Jan. 2011.

[14] K. Basu and P. Mishra, "Test Data Compression using Efficient Bitmask and Dictionary Selection Methods," *IEEE Transactions on VLSI*, 18(9), pp. 1277-1286, 2010.

[15] V.Garofalo, E. Napoli, N.Petra, A.G.M.Strollo, "Code compression for ARM7 portable medical electronic systems," *Circuit Theory and Design, 2007. ECCTD 2007. 18th European Conference on*, pp.687-690, 27-30 Aug. 2007.

[16] S. I. Haider and L. Nazhandali, "A Hybrid Code Compression Technique using Bitmask and Prefix Encoding with Enhanced Dictionary Selection," *In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Portable medical electronic systems (CASES'07)*, Austria, pp. 58-62, 2007.

[17] S. K. Menon, P. Shankar, "An instruction set architecture based code compression scheme for embedded processors," *Data Compression Conference, 2005. Proceedings. DCC 2005,* pp.470, 29-31 March 2005.

[18] M. Collin and M. Brorsson, "Two-level dictionary code compression: a new scheme to improve instruction code density of embedded applications," *2009 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2009)*, pp. 231-242, 2009.

[19] W.R.A. Dias and E.D. Moreno, "Code compression using Multi-Level Dictionary," in *2013 IEEE Fourth Latin American Symposium on Circuits and Systems (LASCAS),* pp.1-4, 2013.

[20] R. Matthew, S. Guthaus, Ringenberg, et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE International Workshop on Workload Characterization*, pp. 3-14,2001.

[21] R.Krishnamoorthy, S.S.Kumar , "A new inter cluster validation method for unsupervised clustering techniques," *Communication and Computer Vision (ICCCV), 2013 International Conference on* , pp.1-5, 20-21 Dec. 2013.