Code Compression Techniques for Low Power Embedded Systems

Abstract- With the advancement in Embedded system technology, memory and power consumption of devices are becoming a major issue. To make devices which consumes less power researchers are looking for new techniques and algorithms, amongst which code compression technique is gaining more and more popularity. In this method, many processor instructions are combined into one single instruction. Also, the power consumption increases as the number of times the access to the memory increases. So, to reduce power, the number of times memory is accessed should be reduced as much as possible. This could again be achieved by code reduction technique because as the size of the code decreases the memory size decreases correspondingly and the number of fetches from the memory decreases. To implement code compression, there are several different techniques which will be discussed at length in this paper.

Keywords: Dictionary Techniques, Hamming Distance, RISC, VLIW, Embedded Systems, SRAM.

I. Introduction

In today's high performance computing environment, there has been the explosion of data and in order to store this data, the size of the memory has increased [2]. For the processors to fetch the data from the memory and process, it requires power and power consumptions is one of the major constraints in microprocessors and embedded system design. Not only this, the researchers are on the verge of creating embedded system which would execute hundreds of giga operations per second [1]. To achieve such high performance, system consume lot of power. Also, embedded system applications now a day are becoming more and more intensive and hardware independent. This is increasing the embedded system's code size and power consumption. To reduce this power various methods are being suggested among which some of them are implemented on hardware such as clock gating or implementing 3D stack architecture, simple processor design [1] etc. while the techniques implemented on software are compiler optimization, memory optimization, hardware/software co-design [1], code compression techniques etc. All these techniques perform some type of approximation which yields to some form of imperfection. But these imperfections alter the output in a trivial manner. And at the same time, it provides advantages such as cost reduction and energy efficiency [2]. This paper focuses at large on code compression techniques. There are several methods which can be perform code compression as discussed below.

II. Fundamentals of Code Compression

As described earlier, code compression technique converges multiple processor instructions into a single instruction, which helps reducing the power consumption by reducing the number of times the memory is accessed. Power consumption reduction techniques can either be done on hardware such as clock gating, dynamic frequency and voltage scaling etc. But majority of embedded applications now a day are hardware independent. So, to reduce consumption of power, power reduction techniques are performed on the software itself such code compression techniques. The algorithm for code compression is described below.

The steps involved in code compression are [3]:

1. Start compression flow
2. Read new instruction code
3. Check if the instruction already occurs
- Yes: increase the counter of this instruction
4. Check if the end of the file reaches?
- Yes: go to step 5
- No: go back to step 2
This loop is performed until the program reaches the end of file.
5.
- Sort all counter in decreasing order
- Replace original code with compressed code
- Write contents of ROM and Dictionary initial files

The next step after the compression process is to decompress the data for execution. Decompression is generally performed on the fly by the hardware module as shown in the fig below. The decompression block is placed between program memory and fetch unit and is called as post cache decompression [4] [16] as shown in fig. 1.

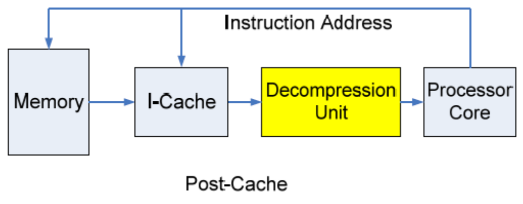

Fig. 1: Post Cache Decompression

Once the code is loaded in the program memory, the steps performed by decompression block is exactly opposite as in compression block. The steps performed are as follows [3]:

1. Get instruction request from MPU core
2. Check if this address is buffered
- Yes: access Dictionary and return original instruction
- No: send request to ROM
3. Read program memory
4. Check if data valid to read
- No: wait until data is valid
- Yes: go to step 5
5.
- Latch data into instruction buffer. It can be read again for the next request
- Access dictionary
- Use bit 2 of previous address to select an instruction
- Set ready instruction
6. Check if a new request occurs
- Yes: go to step 2
- No: go to IDLE state

The results as shown in [3] shows that this algorithm can give reduction in power consumption by 40% but the reduction is code size is not significant. In fact, in some cases the size of the program increases. Hence new techniques are being found out to overcome this problem of large program size.

III. Code Compression Techniques

1. Dictionary Construction Technique for Code Compression Systems

This is the earliest and most used dictionary compression technique for code compression. It replaces the sequence of codes which are repeated by a code word, which is decoded and decompressed by the compiler during run-time. It just keeps a single instance of that sequence in the dictionary and replaces that sequence with the code word which is an index that points to place where that sequence is stored in the dictionary [8]. While executing the program, whenever the compiler come across the code word, the control is transferred to the index pointed by the code word in the dictionary and the sequence is executed and then the control is transferred to the instruction following the code word [8].

This dictionary compression technique takes care of only static compression. To ensure dynamic compression, two level of dictionaries which runs in separate pipeline are used. It increases the code density. There are two dictionaries as discussed which can handle compression of individual instructions and sequence of 2-16 instructions. Both the dictionaries run simultaneously in different pipelines and decompress individual instructions and sequence of instructions in parallel. The general concept of two level dictionary is that once the code has been compressed, it contains combination of compressed sequence of instruction and individual compressed instructions and uncompressed instructions [9]. If the FW has fetched sequence code word (SCW), then it decodes all the instructions one by one. The SCW points to the sequence word (SW) in the sequence dictionary. Each SW has sequence of instruction code word (ICW). Hence it decompresses all the ICW and gives us the original instruction [9]. Now, if the FW fetches ICW, then the compiler can bypass the sequence dictionary and fetches and decompress the instructions from ICW to retrieve the original instruction [9]. Also, if the FW fetches uncompressed instruction, then the compiler can bypass both the dictionary and execute the instruction [9]. The entire process is shown in the fig. 2. The results for dictionary techniques for code compression show that one-level of code compression gives better result in terms of static

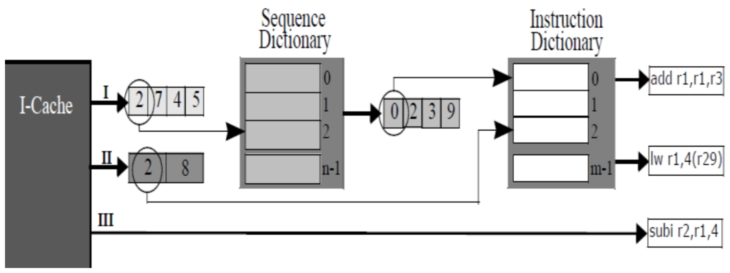

Fig. 2: Dictionary Based compression

code compression giving a compression ratio in the range of 21.14% to 29.76% [8]. But when it comes to dynamic code compression, two-level dictionary techniques outperform the one level dictionary technique with compression ratio of about 23% with energy reduction in the range of 2% - 21%.

2. Nop Compression Scheme Based on VLIW Architecture

Before discussing Nop Compression scheme it is necessary to know what is VLIW architecture? VLIW stands for Very Long Instruction word. Just as RISC is simpler than CISC, similarly VLIW is even more simpler and cheaper than RISC due to further hardware simplification [11]. VLIW has become a leading processor in embedded applications because of its features of executing multiple operations, bundled instructions and large register files, and low hardware cost simultaneously as all these results into reduced energy consumption. But these VLIW processor does not have run time support, so compiler explicitly uses Nop operations to support VLIW architecture. But this increases the instruction size and with this increases the power consumption. To take care of this condition, we need to compress these Nop instructions. To implement this, reconfigurable processor is used which contains core known as VLIW core which executes control intensive operations and Coarse-Grained Architecture (CGA) which accelerates the slow operations. For compression, a stop bit compression scheme is used. In this, every instruction has one bit reserved. If this bit is “1”, then next instruction can continue in the same cycle and if the it is “0”, then the next instruction would be executed in the different cycle [10]. These instructions in a single cycle are known as Nop bundles. In VLIW Nop compression scheme, the processor uses a queue which fetches the next instructions in advance. But this increases the possibility of caches miss because of the jump statements. In order to avoid this, VLIW architecture uses early compression technique which proposes schemes viz. 1) no compression first bundle in which the first bundle of every building block is not compressed. 2) Merged-Compression in which if the compression bits of first two bundles is the same then we can merge them to the previous building block [10]. 3) Superblock compression in which we compress only the not taken path (path fetched when branch prediction is not taken) and do not compress basic blocks in the taken path (path fetched when branch prediction is taken) and finally 4) Branch-hinted Compression in which we can compress both taken and not-taken building blocks. This compression technique gives approximately 25% improvement in clock frequency and about 23% code compression by reducing the instruction size [10].

3. Variable Length Instruction Compression

To reduce power consumption because of SRAM, variable length instruction compression scheme has been developed which uses instruction template based compression technique and two variable length instruction encoding technique. It removes part of the instruction to make it variable length instruction. It uses templates like Nop removal, and to achieve this it uses static multiple issue architecture like Target Triggered Architecture (TTA) [12]. It has two stages viz. instruction template selection and compression and second is variable length decoding. In the first stage, the instructions are encoded to include the template field. These templates indicate the number of moves slots available in instruction format of the processor architecture. The move slots which are not encoded are implicitly used for Nop. It is necessary to choose a selection technique which removes majority of these Nop's from the program code with minimum number of templates. One way to achieve that is to include new instruction formats which would have few templates assigned for instructions and rest of them would be for Nop. All the templates with Nop would be just removed. Hence, by merging the templates used for instructions with those used for Nop, the number of templates used for Nop has been reduced by a power of two. The actual compression takes place while scheduling, where all the instructions are tried to match to the list of templates. The templates with majority of Nop instructions gives the best result of compression [12]. To decode, it is necessary to match the template, fill in the Nop slots and expand it back to its original form [13]. Using this technique 44% reduction in program size can be achieved [12].

4. A Hamming Distance Based Code Compression Technique

Another way of code compression is based on the hamming distances. It makes use of dictionary vector which is used as reference by program vectors. Also, it uses bit toggling for decoding [7]. Compression ratio is a parameter to indicate the amount of compression achieved and is defined as the compressed code size divided by original code size. It should be as small as possible [7]. The Idea behind this Compression technique is to build a dictionary based on the classification of the instructions into different logical classes. The partitioning algorithm decides the size and number of dictionaries which is fixed for a given processor [6]. There are four steps for encoding in this method

1) First Input File Construction (first input pass): In this stage, the instructions to be compressed are read in vector form which is 32-bit one at a time and then all vectors are distributed in frequency domain which are then used for next steps of compression [7].
2) Reduced Dictionary Selection (first dictionary pass): The purpose of this stage is to reduce the size of vector dictionary and include bit toggling information for efficient decoding. The selected vector in the reduced dictionary is at the most a specified hamming distance from the original dictionary. There are three methods of vector selection for reducing dictionary size explained below [7].

i) Frequency Selection Method: In this, the vectors which most frequently occur in the original dictionary are included in the reduced dictionary. The advantage of this method is that in the reduced vector dictionary there are higher number of the vectors that have zero hamming distance [7].

ii) Maximum Span selection method: In this, for each vector in the dictionary, the hamming distance of all the vectors in dictionary is found out. After this, the vector which spans most distance is included in the reduced dictionary and discards all the vectors that are at the set distance or less from the chosen vector. And this process continues until all the vectors from the original dictionary are discarded. The advantage of this method is that it reduces the number of vectors in the reduced dictionary size [7].

iii) Combination of frequency selection and maximum spanning model: It reaps the benefits of both the above methods and creates dictionary by choosing the most frequent vector in the original dictionary and discards all the other vectors that are at a set hamming distance or less than the chosen vector which is the most frequent vector [7].

3) Reduced Dictionary Fill and Code Word Assignment (second dictionary pass): In this stage, the reduced dictionary created are filled with most frequent vectors from original dictionary which are not included in the reduced dictionary. This is done to avoid wasting indexing space and to reduce more number of toggling locations. These indices then serve as the code word for dictionary compression [7].

4) Compression scheme (final input pass): In this, the vectors are converted into their code. It consists of a code word as described in the previous stage, number of toggles bits and up to 7 toggle locations of 5-bits each as shown in the fig. 3.

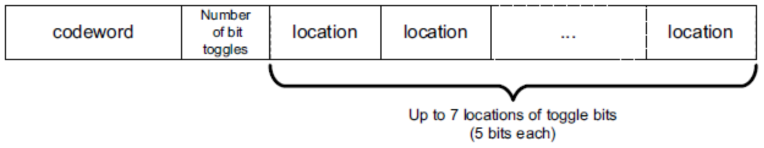

Fig. 3: Format of Compressed Program Code

Decompression Engine: In this, the inputs are the instructions from the previous compression stage. These act as the index to the reduced dictionary and the extra bits denotes which bits are to be toggled. But the problem is that, it's a serial decompression method and with variable length instructions, its performance is affected. To avoid this, decompression is done in parallel with instruction fetching. It fetches and compresses 32-bit stream of instructions and data and decompresses them first rather than waiting for the whole program to be fetched. This is known as stream decompression.

The experiments show that with combination of frequency and maximum span selection techniques and hamming distance upper bound to be 3, a compression ratio of 76.2% can be obtained [7].

5. Arithmetic Coding for Low Power Embedded System Design

Arithmetic coding is designed with the aim of reducing the power consumption and bit toggling. The algorithm is based on quasi arithmetic coding which has random access and fast-table based decoding [5]. This algorithm is a boon for embedded systems because of the design constraints in the memory availability which imposes limitations on the size of the program. Hence, reduction in the binary coding is used to overcome that restriction [5]. There are various effects of code compression on the energy and performance because since the code size is smaller there is less memory access. As the memory access is less, usage of the bus for fetching the data from memory reduces and so does the power consumption and since there are fewer fetches, the performance increases. Also, to increase performance we need to perform parallel decompression. This is performed in blocks of small size. To perform decompression, there are couple of terms which are necessary to know.

1) Simple and Basic Block: Simple block is the sequence of instructions, while a basic block is a simple block which is not contained in any other simple block.
2) Byte Alignment: It means that a compressed basic block can only start at a byte boundary.
3) Branch offset patching: In this the address of the jump is to be given in the compressed form rather than in uncompressed form. This could be achieved by making a table which would take uncompressed address and map it to the compressed address.
Also, to reduce power it necessary to reduce toggle count as

$$\textbf{Energy} = 0.5 * n * C_{eff} * V^2$$

Where, n = toggle count,
C_{eff} = Capacitance,
V = Voltage difference

To perform arithmetic coding and reduce bit toggling Markov encoding method is used as described in [5].

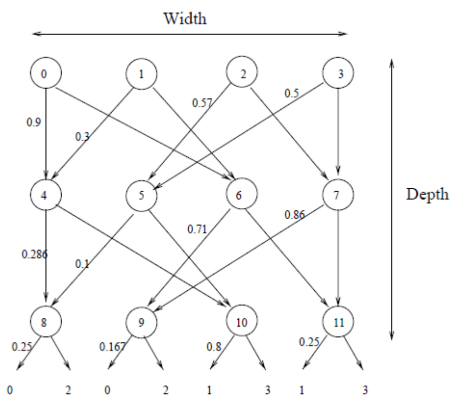

Fig. 4: Markov model

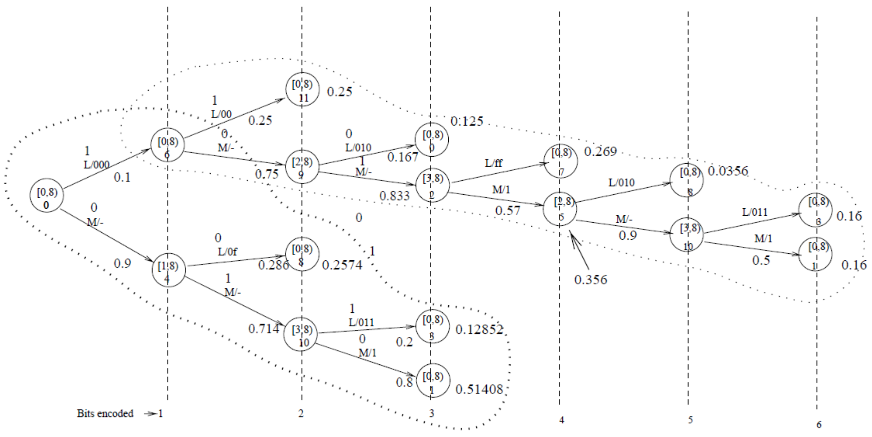

Fig. 5: Expanded state graph based on Markov model

It makes use of regular assignment and inverse assignment techniques to reduce bit toggling [5] and based on the ESG a decompression table is made which is in compressed from. This method gives power consumption optimization by 35% without impacting the performance of the system and without increasing the cost and hence it proves out to be one of the most efficient methods for code compression.

Another variation to arithmetic coding is context base adaptive binary arithmetic coding (CABAC). [17] describes the algorithm of CABAC. It includes three steps viz. Binarisation in which it converts all the syntax elements into stream of binary elements. The next step is known as context modelling in which each binary symbol from the sequence generated in Binarisation step is given to a probability model, and depending on that the bins are modelled into four types of contexts as described in [17]. The third and final step is binary arithmetic coding (BAC). It has three types of execution modes viz. regular, bypass and termination. For BAC to be executed in regular mode, the output of context modelling should be such that the neighboring elements i.e. the current and previously encoded bin should be closely correlated. Bypass mode is used when the correlation between neighboring bits is low. And termination mode is used to terminate the CABAC.

In CABAC, due to context modelling that is updated repeatedly the compression ratio increases significantly due to high correlation. But at the same time the memory access increases and so does the power consumption. So in order to improve the performance even further, a cache memory is embedded in the encoder architecture. Due to this inclusion of cache memory, the power consumption decreases and is given by,

$P_{avg} = R_{Hit} \times P_{cache} + R_{Miss} \times (P_{cache} + P_{SRAM})$

Where, P_{avg} = average power consumed,

R_{Hit} = cache hit rate ratio

R_{Miss} = cache miss rate ratio

P_{cache} = power consumed by cache

P_{SRAM} = power consumed by SRAM [19].

Including the cache memory, in an embedded processor would improve the energy efficiency as memory access decreases as the cache hit ratio increases. When a cache hit occurs the processor checks whether at least one instruction in the block is compressed or not and it accordingly selects the instruction and sends it to the cache output line. In case of miss, the miss handler is invoked and it fetches the requested block from the program memory to the cache. Once the block is transferred to the cache, the task that takes place in case of cache hit would be implemented [17]. Using this scheme, a reduction in power consumption by approximately 50% is achieved [17].

6. Multi-Profile Based Code Compression

As discussed in the previous, there is a trade-off between dictionary compression methods when it comes to static and dynamic compression techniques. Hence, a new technique known as Multi-Profile base code compression has been developed which takes the best of both the techniques. It uses a fixed 32-bit index to point the address in dictionary. It is called as ComPacket. It avoids misalignment problems and thus avoids double cache access and which in turn increases power efficiency. In this the dictionary construction is different from what we have discussed so far. Here the dictionary constructed is a blend of static compression profile and dynamic compression profile. The dictionary constructed is called as Unified Dictionary (UniD). Initially the code is ordered in their natural selection criteria i.e. static count for the instructions having static profile and dynamic count for instructions having dynamic profile [14]. After that the first instruction from SD i.e. the most frequent instruction is checked whether it is present in the UniD. If it is not present, then it is added to the UniD. Then the first instruction from the DD is checked whether it is there in UniD and if not it is added to it and this cycle continues till all the instructions from both the dictionaries are covered. Also, the compression method allows the branch instructions to reside in the dictionary provided the branch offset is small. It stores 10 most significant bits for branch offset are stored in the dictionary and the slots inside the ComPacket are used to patch the offset [13]. The different formats of ComPacket are shown fig. 6 below.

Once all these steps are done, the first instruction in the dictionary is considered and the compiler tries to build ComPacket. It tries making format 4 first if that is not possible then it tries making format 3B or 3 and if nothing works out then it tries making format 2B of ComPacket. If there are branches the last bit of the ComPacket would contain the offset patch or else, it would be padded. Similarly, the ComPacket formats are built for all the instructions in the dictionary and applied to the code [13].

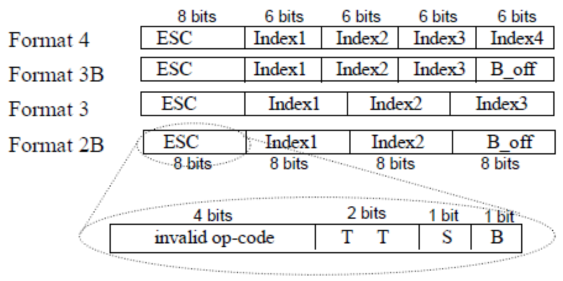

Fig. 6: ComPacket Formats

This technique of code compression which uses multiple profiles to make dictionary have proven to be very efficient as it provides compression ratio of about 70% and reduction in cache access is 50% which results in reduction in energy consumption by 46% [13].

7. An Improved BitMask Based Code Compression Algorithm

In traditional dictionary method, the instructions or sequence of instructions were stored as it is and encoded to form a code word that would act as the index to the dictionary. But there are some instructions that are not present in the dictionary and must be fetched from the program memory instead of cache which reduces the performance of the system. To include those instructions in the memory, bitmask based encoding is used. The instructions which are not included in the dictionary are compared with the instructions that are present in the dictionary and based on their hamming distance they are encoded.

In this algorithm, there are two different types of instructions, one is dictionary entry (DE) which is same as normal instructions present in the dictionary and its code word points to the index in the dictionary. The other type of instructions is called as dictionary children (DC). The type of instructions is decided by dictionary selection method in which the instructions with frequency above the pre-decided threshold are DE and below that threshold are DC instructions. Their encoding is done with bitmask technique which indicates which bits are to be toggled [14] [17]. It contains a C bit which tells whether the instruction is in its compressed form or not. In uncompressed instruction, the C bit is followed by the original instruction whereas in compressed instruction the C bit is followed by the dictionary index which corresponds to the original instruction in the dictionary in case of DE instruction while in case of DC instruction it corresponds to a DE instruction. After dictionary index, there is a field known as mask code which tell the number and type of mask that should be used to decompress and retrieve the original instruction. In case of DC instruction there are additional mask info field which gives us information about the location of the mask, bits to be toggled etc. [17]. This technique improves the compression ratio by 20% when the dictionary size is small and by 9% if large dictionary size is considered [17].

8. CRAMES: Compressed RAM for Embedded Systems

As discussed earlier, memory is a scarce resource in embedded systems. To increase the memory without any increase in the hardware cost, CRAMES could be implemented. It uses operating systems virtual memory to store swapped out pages in the compressed form rather than using RAM. Basically, CRAMES divides the RAM into two parts, one part is called the compressed RAM and other is the working area [19]. The size of these area is random. When the memory is low, the pages that are not used frequently are swapped out from the compressed area of the RAM to uncompressed area of the RAM to accommodate new pages. Hence, the size of the compressed RAM changes dynamically. To access the memory and swap pages it must be registered with the kernel. The function of CRAMES is shown in the fig. 7 below.

This is how CRAMES provides RAM compression for embedded device with no hard disk. Since there is no external memory, the power consumption can be reduced using this technique.

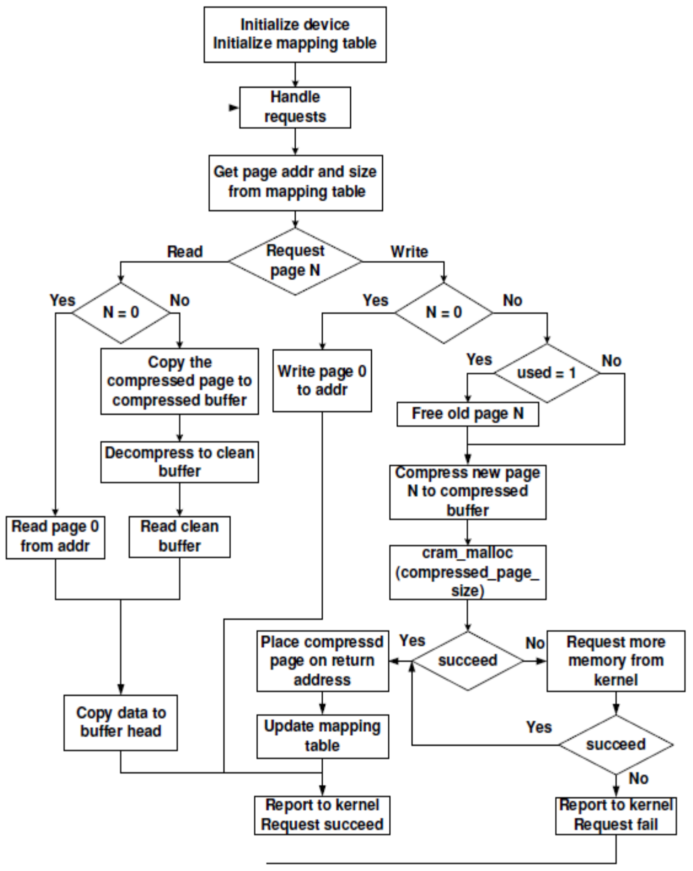
Initialize device
Initialize mapping table
Handle requests
Get page addr and size from mapping table
Request page N
Read
Write
Yes
N = 0
No
Copy the compressed page to compressed buffer
Decompress to clean buffer
Read page 0 from addr
Read clean buffer
Copy data to buffer head
Write page 0 to addr
used = 1
Free old page N
Compress new page N to compressed buffer
cram_malloc (compressed_page_size)
succeed
Place compressd page on return address
Update mapping table
Request more memory from kernel
Report to kernel Request succeed
Report to kernel Request fail

Fig. 7: Handling request in CRAMES device

IV. Conclusion

Unlike the concept of Data Compression which has established its roots, code compression is fairly a new area of research which is speeding up since late 90's. Its development has been a boon for embedded system technologies. Due to this, the power consumption and memory size has decreased. But due to the rapid development of technology and platform independent nature of new applications, the need for further development of technology and reduction of power is leading the researchers to the develop new technologies like compiler techniques which would help compressing the code dynamically during compile time as explained in [21] [22].

V. References

[1] Arun Rodrigues, David Jensen, "Embedded Systems and Exascale Computing".

[2] "Models for Energy-Efficient Approximate Computing" Ravi Nair, IBM Thomas J. Watson Research Center Yorktown Heights, NY 10598 nair@us.ibm.com.

[3] "Low Power Embedded System Design Using Code Compression", Quynh Ngoc Do†, Thong Chi Le‡, quynh.dongoc@icdrec.edu.vn, chithong@hcmut.edu.vn.

[4] "Code Compression Architecture for Memory Bandwidth Optimization in Embedded Systems", Yi-Ying Tsai, Ke-Jia Lee and Chung-Ho Chen *Department of Electrical Engineering National Cheng Kung University, Taiwan, R.O.C. {magi,over}@casmail.ee.ncku.edu.tw; chchen@mail.ncku.edu.tw.*

[5] "Arithmetic Coding for Low Power Embedded System Design", Haris Lekatsas (Princeton University), J¨org Henkel (NEC USA, Princeton), Wayne Wolf (Princeton University).

[6] "An Instruction Set Architecture Based Code Compression Scheme for Embedded Processors", Sreejith K Menon, Priti Shankar, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

[7] "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.

[8] "A Dictionary Construction Technique for Code Compression Systems with Echo Instructions", Philip Brisk, Jamie Macbeth, Ani Nahapetian, Majid Sarrafzadeh, Department of Computer Science,

University of California, Los Angeles Los Angeles, CA 90095.

[9] “Two-Level Dictionary Code Compression: A New Scheme to Improve Instruction Code Density of Embedded Applications”, Mikael Collin and Mats Brorsson, KTH School of Information and Communication Technology, Royal Institute of Technology, Stockholm Sweden. Member of HiPEAC EU Network of Excellence email: {mikaelco, matsbror}@kth.se.

[10] “Nop Compression Scheme for High Speed DSPs Based on VLIW Architecture”, Taisong Jin, Minwook Ahn, Donghoon Yoo, Dongkwan Suh, Yoonseo Choi, Do-Hyung Kim, Shihwa Lee, Samsung Advanced Institute of Technology.

[11] “Very-Long Instruction Word (VLIW) Computer Architecture”, Philips Semiconductors

[12] “Variable Length Instruction Compression on Transport Triggered Architectures”, Janne Helkala, Timo Viitanen, Heikki Kultala, Pekka J¨a¨askel¨ainen, Jarmo Takala, Department of Pervasive Computing Tampere University of Technology, Finland Email: fjanne.helkala, timo.2.viitanen, heikki.kultala pekka.jaaskelainen, jarmo.takalag@tut.fi, Tommi Zetterman, Heikki Berg, Radio Systems Laboratory Nokia Research Center Espoo, Finland. Email: ftommi.zetterman, heikki.bergg@nokia.com.

[13] “Multi-Profile Based Code Compression”, E. Wanderley Netto **(**CEFET/RN IC/UNICAMP), R. Azevedo **(**IC/UNICAMP), P. Centoducatte **(**IC/UNICAMP), G. Araujo **(**IC/UNICAMP), Caixa Postal 6176, 13084-971 Campinas/SP Brazil, +55 19 3788 5838, {braulio, rodolfo, ducatte, guido}@ic.unicamp.br.

[14] Chang Hong Lin, Wei Jhih Wang, "An Improved BitMask Based Code Compression Algorithm for Embedded Systems", vol. 00, no., pp. 152-157, 2011, doi:10.1109/ISED.2011.15

[15] “A Hybrid Code Compression Technique using Bitmask and Prefix Encoding with Enhanced Dictionary Selection”, Syed Imtiaz Haider and Leyla Nazhandali, Virginia Polytechnic Institute and State University, 302 Whittemore, Blacksburg, VA 24061. {syedh, leyla}@vt.edu.

[16] “CoCo: A Hardware/Software Platform for Rapid Prototyping of Code Compression Technologies”, Haris Lekatsas, J¨org Henkel, Srimat Chakradhar, Venkata Jakkula, Murugan Sankaradass, NEC Labs America, 4 Independence Way, Princeton, New Jersey 08540, USA. Vorras Corporation, 1 West Drive, Suite 1202, Princeton, New Jersey 08540, USA. flekatsas,henkel,chakg@nec-labs.com.

[17] “Low-power context-based adaptive binary arithmetic encoder using an embedded cache”, S.-F. Lei, C.-C. Lo, C.-C. Kuo, M.-D. Shieh, Department of Electrical Engineering, National Cheng Kung University, Taiwan, E-mail: leisf@mail.ncku.edu.tw.

[18] “Code compression architecture for cache energy minimization in embedded systems”, L. Benini, A. Macii and A. Nannarelli.

[19] “CRAMES: Compressed RAM for Embedded Systems”, Lei Yang† (†l-yang, dickrp@northwestern.edu, Northwestern University, Evanston, IL 60208), Robert P. Dick†, Haris Lekatsas‡ (‡lekatsas, chak@neclabs.com, NEC Laboratories America, Princeton, NJ 08540), Srimat Chakradhar‡.

[20] “Survey of Low-Energy Techniques for Instruction Memory Organizations in Embedded Systems”, Antonio Artes, Jose L. Ayala, Jos Huisken, Francky Catthoor.

[21] “Compiler Techniques for Code Compaction”, SAUMYA K. DEBRAY and WILLIAM EVANS, University of Arizona, ROBERT MUTH, Compaq Computer Corp. and BJORN DE SUTTER University of Ghent.

[22] “Design and Run Time Code Compression for Embedded Systems”, Sri Parameswaran, J¨org Henkel, Andhi Janapsatya, Talal Bonny Aleksandar Ignjatovic, *School of Computer Science and Engineering*, *The University of New South Wales*, *NSW 2033. Australia, Department of Computer Science Karlsruhe University*, *Zirkel 2, D-76131 Karlsruhe. Germany*