Compressed Code for RISC Processor (CCRP) in Real-Time Embedded Systems

D.Lakshmi Narayana [1*]
ResearchScholar,
Sathyabama University,
Chennai,Tamilnadu,India
Lakshman5504@gmail.com

Dr.V.Ganesan [2*]
Associate professor,
Sathyabama university,
Chennai,Tamilanadu,India
Vganesh1711@gail.com

Article Info
Volume 83
Page Number: 1963 - 1967
Publication Issue:
May - June 2020

Article History
***Article Received**: 11August 2019*
***Revised**: 18November 2019*
***Accepted**: 23January 2020*
***Publication**: 10 May2020*

Abstract:

Embedded systems have become an essential part of everyday life, and are widely used worldwide. Embedded systems must be cost effective, and memory occupies a substantial portion of the entire system. The complexity and performance requirements for embedded programs grow rapidly. Thus, reducing the code size and providing a simple decompression engine are both challenges when applying code compression to embedded systems. Memory plays a crucial role in designing embedded systems. Embedded systems are constrained by the available memory. A larger memory can accommodate more and large applications but increases cost, area, as well as energy requirements. Code-compression techniques address this issue by reducing the code size of application programs. It is a major challenge to develop an efficient code-compression technique that can generate substantial reduction in code size without affecting the overall system performance. We present an efficient code-compression technique, which significantly improves the compression ratio.

Keywords: *Code Compression, Embedded Systems, Cache Line Address, Line Address Table*

1. Introduction

With the exponential growth of the number of interconnected computing platforms, computer security has become a critical issue. The utmost importance of system security is further underscored by the Expected proliferation of diverse Internet enabled embedded systems ranging from home appliances, cars, and sensor networks to personal health monitoring devices. Moreover, as capabilities of embedded processors increase, the applications running on these systems also grow in size and complexity, and so does the number of security vulnerabilities.

2. Code Size and RISC Processors

The high performance RISC processors are widely used as embedded processors. The RISC processors tend to decrease and simplify the hardware electronics of a processor on a single chip. Because of its effect on code size, it enabled RISC processor for embedded systems. A comparison [1] of distribution of Arithmetic/logic instructions and data transfer instructions for two benchmark programs on VAX and MIPS is shown in Table 1. The VAX is a popular CISC processor and MIPS are a popular RISC processor. The 50% to 133% increase in data transfer instructions for the MIPS, compared to the VAX, is due to use of several load and store instructions in MIPS. This 'code size bloating' problem of RISC processors is depicted in [2] which compares the object code size of an MPEG2 encoder compiled on multiple processors of different architectures. The RISC processors ARM Thumb and SHARC need 68.2 kB and 106.2 kB respectively.

Attacks that impair code integrity by injecting and executing malicious code are one of the major security issues. A common method for code compression is the Compressed Code RISC Processor (CCRP), which is show in figure-1.

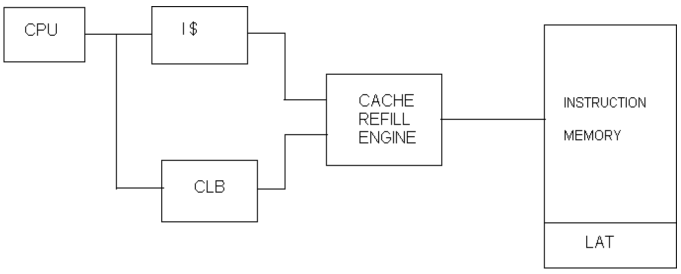


Figure.1. Compressed Code RISC Processor

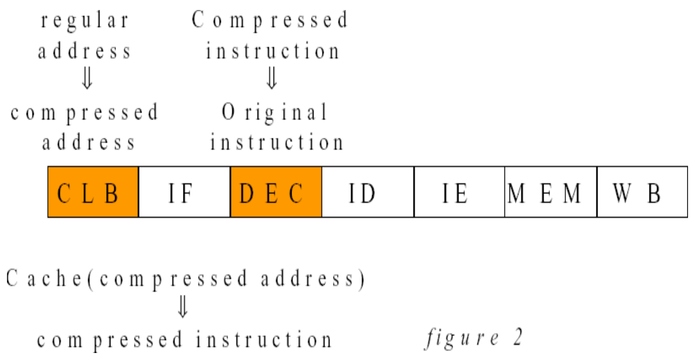
It breaks the program into blocks of n, where n corresponds to the cache line size.Each block is compressed individually by a suitable algorithm. Line Address Table (LAT) is used to compress cache line within the compressed program. Cache Look aside Buffer (CLB) is added to alleviate LAT lookup cost. During program execution, upon each cache miss, LAT entry cached used to calculate the compressed Cache Line Address (CLA).[1] The compressed cache line is then fetched from memory, decompressed, and placed into the instruction cache.

3. Modification of the Processor

We consider a 64-bit simple scalar instruction set, which is similar to MIPS .We assume a simple 5-stage pipeline processor for ease of discussion. In order to compress the instruction cache, decompression must be done after the cache [2]. Therefore the CPU must now be modified to include 2 additional functions. Address translation from the uncompressed version to the compressed (CLB).

3.1. Instruction Decompression (DEC)

These 2 additional sections are added to the usual 5-stage pipeline as shown in figure 2.



The CLB section implements address translation from the uncompressed version to the compressed version for each instruction and DEC implements Instruction Decompression. [3]The Instruction Fetch (IF) stage will now fetch the compressed instruction instead of the uncompressed instruction. The stage length for CLB and DEC are both dependent on the compression algorithm. The additional CLB and DEC sections are both before the Instruction Decode (ID) stage.[4] Thus stages within these sections will add to branch penalty. In addition, these penalties cannot be reduced by branch prediction since they are before the ID stage, where instruction opcode is first being identified.

3.2. Compression Algorithm

Problems: Compression algorithm used must have simple CLB and DEC sections, or it will lead to large branch penalty. [5]The problems we face with the usual CCRP compression algorithm that compresses serially within the block, byte wise are for a 64-bit instruction, we require 8 DEC stages to decompress.

3.3Granularity of Random Access

With CCRP, random access is only at block level. If the program counter jumps from one compressed block to another and it is the very last instruction in the block that it is jumping to, then all instructions within the block must be decompressed before the actual instruction will be fetched. This will lead to additional stages of delay.

Modified Algorithm: We devise such an algorithm. It can be thought of as a binary Huffman

with only two compressed symbol lengths, but due to its similarity to table lookup we refer to it as Table. It uses 4 tables, each with 256, 16-bit entries. [6] Each 64-bit instruction is compressed individually as follow:
Divide the 64-bit instruction into 4 sections of 16-bit.Search content of table 1 for section 1 of the instruction. If found, record the entry number. Repeat Step (ii) for section 2 with table 2 and so on with remaining sections. If entry number in the corresponding table replaces all the sections, then the instruction is compressed. Otherwise, the instruction remains uncompressed. A 96-bit LAT entry tracks every 64 instructions. 32-bits equal the compressed address. The rest of 64 bits are on or off depending on whether the corresponding instruction is compressed or not. Since this method tracts each instruction individually, the inter-block jump problem is gone.[7] For the DEC section, decompression is either nothing or 4 table lookups in parallel.So one stage is sufficient. For the CLB section, it is simply a CLB lookup follow by some adding in a tree of adders. This could be done in one cycle, with CLB lookup taking half and adding taking another half.

Details: With the Table compression algorithm, the compressed instructions can be either 4 or 8 byte long. This could lead to misalignment problem for instruction cache access in case of 8-byte instruction. For example, an 8-byte instruction can have its first 4 byte in the end of one cache line and its last 4 byte in the start of another cache line. This result in 2 cache accesses for a single instruction fetches. To solve this, instruction cache is divided into two banks, with each bank 4-byte wide. The first bank is used to store the upper half word of the 64-bit word. [8] The second bank is used to store the lower half. Since every 8-byte instruction must have its two 4 byte halves in different banks, simple logic can be used to ensure each bank is accessed correctly to fetch the instruction in one cycle. The determination of the content for the 4 tables is critical in achieving a good compression for this method. Between the 4 tables, they can keep 256 power 4, or 4 gig entries of distinct instructions. However, for every entry within a table, 256 power 3, or 16 Meg entries of instructions will take on that same section. So it is quantity without quality.

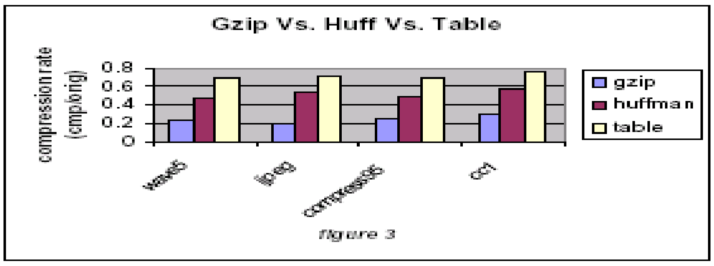


Figure.3. Compression Result for Table Compression

3.4.Result for the Modified Algorithm

On average Table can compress to 70% of the original size. In comparison to CCRP with Huffman byte serial compression at block size of 8, we loose about 20% on average. This 20% lost is partly due to the smaller granularity of random access we have achieved.[9] In general the smaller granularity of random access, the less compression will be achieved. This can be observed again with result, which does not provide any random access. On average it compresses to 20% of the original size, which is 30% better than CCRP with Huffman.

4. Branch Compensation Cache (BCC)

Problem: As mentioned earlier, in order to perform decompression after instruction cache (between the I-cache and the CPU), we add two more stages into the pipeline before ID: CLB (Cache Look Aside Buffer) and DEC (decompression). As a result, branch penalty is increased to three cycles. This is illustrated in figure 4.

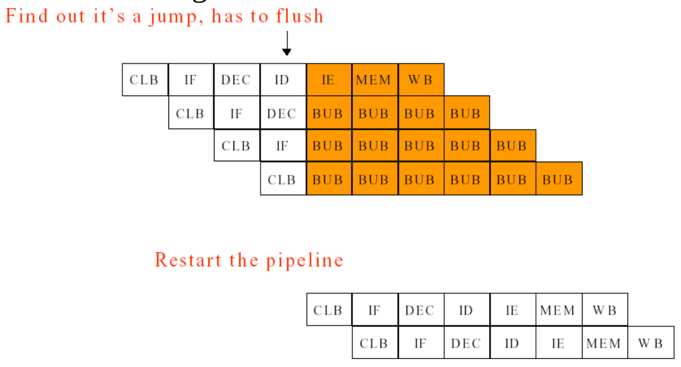

Find out it's a jump, has to flush
CLB IF DEC ID IE MEM WB
CLB IF DEC BUB BUB BUB BUB
CLB IF BUB BUB BUB BUB BUB
CLB BUB BUB BUB BUB BUB BUB
Restart the pipeline
CLB IF DEC ID IE MEM WB
CLB IF DEC ID IE MEM WB

Figure-4: Increased Branch Penalty

Originally, in regular five-stage pipeline, the branch penalty is only one stage (IF), but in our seven-stage pipeline, the branch penalty increases to three stages (CLB, IF, DEC).[10] Obviously it is not good, so we need a solution.

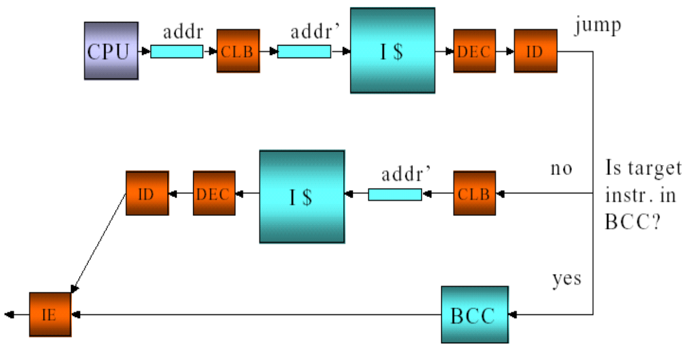
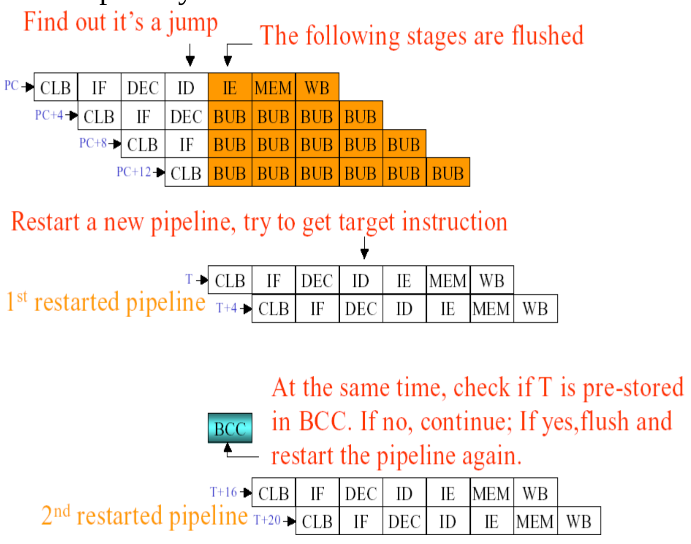


Figure-5 Use BCC to Reduce Branch Penalty

Solution: The solution is to add a branch compensation cache (BCC) and try to pre-store the target instructions there. Whenever we encounter a branch (or more precisely, a PC jump), go to the BCC and check if the target instruction is there. The basic idea is shown in *figure 5*.

In figure 5, a PC jump is found at the ID stage, so we go to check BCC to see if the target instruction is pre-stored there. If no, then we have to go through CLB, DEC and ID pipeline stages but if yes, then we simply fetch it and keep going, no branch penalty at all in such case.


Find out it's a jump
The following stages are flushed
PC CLB IF DEC ID IE MEM WB
PC+4 CLB IF DEC BUB BUB BUB BUB
PC+8 CLB IF BUB BUB BUB BUB BUB
PC+12 CLB BUB BUB BUB BUB BUB BUB
Restart a new pipeline, try to get target instruction
1st restarted pipeline
T CLB IF DEC ID IE MEM WB
T+4 CLB IF DEC ID IE MEM WB
At the same time, check if T is pre-stored in BCC. If no, continue; If yes,flush and restart the pipeline again.
BCC
2nd restarted pipeline
T+16 CLB IF DEC ID IE MEM WB
T+20 CLB IF DEC ID IE MEM WB
Figure-6: BCC Implementation

At ID stage we find out a jump instruction being taken, so we must flush the current pipeline and restart. Figure 6 shows a more detailed pipeline implementation, where the key points.[11].The restarted pipeline is used to get the target instruction through normal stages, i.e. using CLB to get the compressed address from the regular address, using IF to fetch the decompressed instruction, using DEC to decompress it to get target instruction. Simultaneously, we also go to the BCC to check if the required target instruction is pre-stored there. If no, then keep running the restarted pipeline; if yes, then we directly fetch the target instruction from BCC and flush/restart the pipeline for a 2nd time[12].

The 2nd restarted pipeline is used to provide the sequential instructions after the jump. In order to completely eliminate the penalty incurred by a jump, we require T, T+8, T+16, T+24 instructions are all pre-stored in BCC upon a hit.

So the 2nd restarted pipeline starts CLB stage with T+32 instructions, where T is the target instruction, T+8 is the next instruction, and so on.

Result: To quantify the BCC performance, we observed several applications such as ijpeg, wave5, etc.

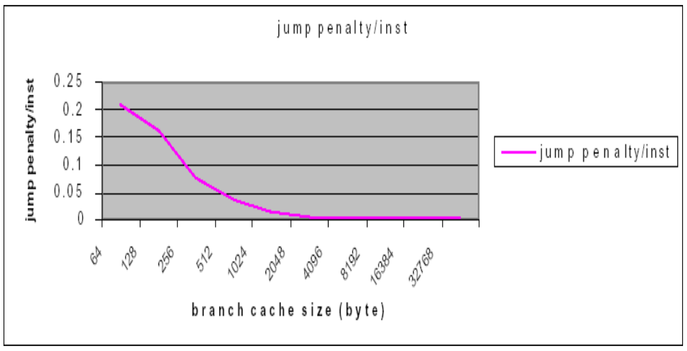


Figure-7: BCC Performance

Here the cache block size is kept as 8-byte long, while the block number varies from 1, 2, 4, 8, ¼, to 4096 to give different cache size. Directly mapped cache is used and the replacement policy is least recently used (LRU). From the observed results we can see that for a fairly small branch compensation cache (1KB ~ 2KB), the branch penalty is significantly reduced, which indicates our approach is very effective.As far as the compressed cache performance is considered, compressed cache with Table algorithm do outperform uncompressed cache, especially within a region of instruction cache size where uncompressed cache is getting from 10 to 90 % hit rate[13].The improvement could go as high as near 40%. This is as expected since compressed cache contains more instructions and this translates to better hit rate. The reason that there is a window of sizes where cache compression is more effective is because this is the thrashing region and a small increase in cache size could give big improvement.

Compression in cache is like a virtual cache increase. For example 70% compression in cache could give a cache that is 1.4 times the original size.[14] Therefore cache compression is especially effective within this thrashing region. So the original idea that compressed instruction cache can lead to smaller instruction cache is valid.

5. Conclusions

In addition to real-time requirements, the program code size is a critical design factor for real-time embedded systems. To take advantage of the code size vs. execution time tradeoff provided by reduced bit-width instructions, we propose a design framework that transforms the system constraints into task parameters guaranteeing a set of requirements. The goal of our design framework is to derive the temporal parameters and the code size parameter of each task in such a way that they collectively guarantee the system end-to-end timing requirements while the system code size is minimized.

6. References

[1] J.L.Hennessy and D.A.Patterson, “Computer Architecture: A quantitative Approach”, Fourth edition, Morgan Kaufmann publishers, 2007.

[2] J. Heikkinen, J.Takala, and H.Corporaal, “Dictionary based program compression on customizable processor architectures”, Microprocessors and Microsystems, vol. 33, pp. 139 – 153, 2009.

[3] Y. Xie, W. Wolf, H. Lekatsas, “Code Compression for VLIW Processors using Variable – to- Fixed Coding,” IEEE Trans. VLSI Systems, vol. 14, no.5, pp. 525 – 536, May 2006.

[4] L.Benini, F.Menichelli, and M.Olivieri, “A class of code compression schemes for reducing power consumption in embedded microprocessor systems”, IEEE Trans.Computers, vol.53, no.4, pp.467 – 482, April 2004.

[5] A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture”, in Proc. Int. Symp. Microarch, 1992, pp.81 – 91.

[6] T.M.Kemp, R.K. Montoye, J.D. Harper, J.D.Palmer, and D.J. Auerbach, “A decompression core for power PC,” IBM J.Res.Develop., vol.42, no.6, pp. 807 – 812, Nov. 1998.

[7] J.A. Fisher, P.Faraboschi, and C.Young, “Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”, Morgan Kaufmann publishers, 2005.

[8] A.N.Sloss, D.Symes and C.Wright, “ARM System Developer’s Guide: Designing and optimizing System Software”, Morgan Kaufmann Publishers, 2004.

[9] C.H.Lin, Y.Xie, and W.Wolf,”Code Compression for VLIW Embedded Systems using a self-generating table”, IEEE Trans. VLSI Systems”, Vol.15, no.10.pp.1160- 1171, Oct.2007.

[10] “microMIPS Instruction Set Architecture”, MIPS Technologies, Inc., October, 2009.

[11] B.Govindarajalu, “Computer Architecture: and Organization: Design Principles and Applications”, Second Edition, Mc Graw-Hill publishers, 2010.

[12] http://www.mips.com/products/supporttraining/documentation/ [13] D. Sima, T. Fountain, and P. Kacsuk, “Advanced Computer Architectures: A design space approach”, Pearson Education, 1997.

[14] D.A.Patterson, and J.L.Hennessy,"Computer Organization & Design: The Hardware / Software Interface”, Second Edition, Morgan Kaufmann, 1998.