International Journal of Future Generation Communication and Networking
Vol. 13, No. 4, (2020), pp. 4986–4997

Design of Hybrid RISC Processor for Code Compression

[1]Lakshminarayana Devarakonda, [2]Ganesan.V

Sathyabama Institute of Science & Technology, Chennai, Tamilnadu, India.
E-Mail: lakshman5504@gmail.

Abstract

This investigation utilizes compiler techniques for diminishing memory needed for running and loading program executables. Based on economic incentives, embedded system needs to reduce ROM and RAM is extremely stronger, compiler cost size is also increasing significantly. Alike of network and mobile based computing, the necessity of transmitting an executable code is before placing it to a premium code size. This work concentrates on compressing programmable code size with Hybrid Huffman Code compression and Look up Table with cost minimization. It helps to combine and recognize repeated instruction sequences. In contrary to other approaches, this hybrid method maintains competency to execute programs directly devoid of any intervention in decompression stage. This method is merging with industrial strength for optimizing compiler which facilitates users to provide interaction among conventional code optimization approaches and code compression techniques. It is contended with various complexity associated with optimizing code. The foremost contribution of this investigation is code compression in RISC architecture which is a most resourceful factor for enhancing code compression with repeated code fragments and newer form of hybridized code compression that diminishes cost wise penalty while performing compression.

Keywords- *RISC, code compression, Huffman code, Look Up table, instruction*

I. Introduction

With the development of technologies, embedded systems are considered to be more complex, as embedded program size is rising constantly [1]. This outcome of this constant improvement is that program memories are accountable for huge storage of die area, which is higher that microprocessor core area and other on-chip modules. As an outcome, compressing or reducing program size has been measured as an essential factor for performing designing effort (cost) in association with embedded system. This may indirectly influences the reduction of instructions size. This model is executed in the design of RISC architecture and advanced processors. Reduced instructions are attained significantly with the constraints of number of bits that encodes intermediates and registers [2]. Some registers are drastically based on freedom for compiler to carry out essential factors such as global register allocation and huge instruction to carry out similar amount of computation [3]. The overall outcome is

roughly about 35-45% smaller programs that runs 20-25% lesser than programs with conventional RISC architectures. Subsequent way for reducing program size is the modelling of processors that may perform compressed code [4]. To do it, decompression engine should carry out code decompression in real time environment. However, this is due to branching instructions, decompression that may restart from target of branch instruction. There are two essential features that differentiate data compression and code compression crisis, turning unreasonable for code compression procedures such as Ziv and Lempel approach and variations [5]. This work handles problem of determining code compression approaches that facilitate effectual execution of real time compression engines.

Numerous factors describe compiled code size. Instruction set based framework of target machine has stringer effect. For instance, stack machine may generate compact code, whilst three address RISC machine provides larger code (every operand is named explicitly) [6]. Certain code sequences are chosen by compiler of that effect, this may perform certain transformations during optimization.

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

This work utilizes one approach for compressing code size during final compilation stage. This model is technically simpler and it is constructed based on prior background studies. This code matching approach recognizes identical code for executing the sequence. Compiler utilizes either cross jumping or abstraction to channel execution of repetition via single code copy. The basic algorithm is improved with hybridization of Huffman compression technique and Look up table for cost minimization.

This work provides numerous essential contributions towards literature. Initially, this work provides an extensive evaluation for code compression with RISC architecture. This work helps in saving around 15% of benchmark standards with an average of about 5% approximation [7]. Next, optimization based code space computation and extraction among these code compression and optimization is explained. Here, two approaches are hybridized for performing code compression and cost reduction. Here, a series of approaches are computed and improves baseline compression structure to deal differences among register assignment [8]. This work specifies that this hybridization make crucial significance to carry out compression. At last, these techniques provide a mechanism for handling trade off among over all execution time and code size.

This work is structured as: Section II provides backdrop studies related to compression technique. Section III depicts proposed framework for code compression in RISC architecture for cost minimization. Section IV depicts numerical results and discussion based on proposed framework. Section V illustrates conclusion with direction for future extension.

II. Related works

The essential characteristics of code compression procedure are encoding techniques that deal with decompression approaches and branch instructions. There exist extensively two kinds of compression approaches- dictionary and statistical coding. Former, tries to substitute series or symbols with dictionary index, where addresses change owing to compression may deal branch instruction either with translation table that performs translation of prior address to novel address with offset to branch instructions. Latter is individual symbols are substitution with various sized code words based on frequency occurrence.

Author in [9] depicted that greedy approach may substitute’s instruction groups with dictionary entries that are decompressed during execution time. This provides restriction of target instructions will not be compressed till it initiates such groups. This is due to the fact that branch target instruction happens in intermediate group as it cannot be evaluated directly devoid of accessing every previous instruction in group. They are utilized illegal codes to differentiate compressed and ordinary instructions [10]. There are enormous works that is accessible in prevailing methods which are not capable for space restrictions.

One amongst the extensively utilized industrial standards is THUMB approach. In this approach, 32 bit ARM processor are utilized where concurrent instructions which utilize smaller fields/operands that are specified with 16 bit that get rid of redundant information bits and decompresses to 32 bit instructions during execution [11]. In certain cases, variable length processor like frequent instructions need least fields/operands which for part of significant instructions with least encoding. Therefore, this type of approach is applied to variable length processors [12]. As THUMB code is not of compilation procedure, the crisis of varying address with branch instruction is owing to compression that does not arise.

Processor runs on ARM and implements in THUMB 16 bit instruction for decoding with 32 bit instructions before decoding it. Instead of using ARM instructions, THUMB instructions are used for providing superior compression, however it causes performance degradation as more number of instructions is executed. Some code comprises of mixed THUMB and ARM instructions for mode change are performed with switch instruction. These instructions initiate com precision overhead. Author in [13], have depicted as guided procedures for producing THUMB and ARM code to acquire compression devoid of performance degradation Algorithm functions by substituting these THUMB

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

series with ARM instruction for all compression and recital enhancement. MIPS 16 is alike of MIPS processer extension based architecture where frequent occurrence of 32/64 instructions are specified as 16 bits. They are converted to instruction original before execution.

With Power PC processor based code pack approaches where every instruction is partitioned to two parts and every halves are entropy dependent coded. Owing to resultant based variable encoded instruction size are index table essential to plot older instruction to resolve newer [14]. To reduce index table based overhead, these are compressed and placed, therefore one address translation is essential for grouping. Degradation owing to variable lengths is eliminated by maintaining output buffer which preserves decompressed instructions group accessible for quicker access [15]. Techniques are not applied directly to variable length processors as instruction splitting will causes diverse sized values and will not acquire repetition values, therefore it results in extremely lesser compression.

III. Proposed method

This section comprises RISC architecture that includes some outcomes. This RISC architecture are simpler to pipeline and usually assists superior performance and higher clock rate, however usually needs more instructions to carry put similar work. This may translate higher instruction bandwidth needs which may generally satisfy instruction cache. However, all instructions posses same size where some amount out both instruction bandwidth and program memory that are wasted with simple instruction that are explained with lesser bits. However, RISC architecture may have relatively poor code density. Code compression is measured as a solution for bandwidth and code density issues in RISC design. This work explains about Huffman coding that is utilized as compressor to compress instructional code. It is compression technique which is dependent on probabilistic distribution. Here, frequently applied instructions are encoded with code words. Code works are used for specifying decoding table which comprises of original instructions.

With this Huffman coding model, an effectual compression is used as it offers average codeword length. Benefit of using this is based on prefix tree, that is, no codeword is prefix to another as in Fig 1. This leads to simpler decoding process and more easily for implementation. This is the ultimate cause of using Huffman code as compression approach to encode instruction based object code. Moreover, decoding table size is generated for decompression is larger and directly influences final compression ratio. Therefore, it reduces benefits that have been attained by instruction compression. Compression ratio is not achieved only by reducing encoded instructions, however decoding table is considered.

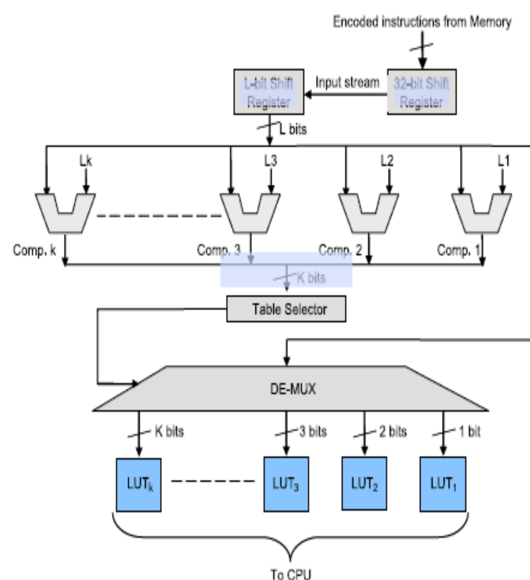

Fig 1. Huffman code based decoder

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

a) Pre-fix tree based Look up Table compression

Here, superior hardware based compression approach is initiated. Indeed of encoded instructions in Look up table is compressed which is significant in size, and if it is higher compression is appropriate. This may deploy conjunction with prefix tree to produce Look up table.

1) Instruction based object code is encoded with fixed code length with sequential encoding approach. The instructions are preserved in Look up table,
2) This table is compressed with Prefix tree based compression technique.

To resolve this crisis, locating branch based target address in memory may patch address to compress code. As these encoded instructions posses fixed variable length, this will not align branch target address instruction at boundaries. Owing to hardware decoder may evaluate encoded instruction address and if it not handled with memory border. There is no cost based penalty for compression.

b) Compressed instructions generation

To produce encoded instruction, here initially consider uncompressed instruction words. Here, instruction words are extracted and placed in Look up Table. With originally available code, all unique instruction words uses numerical sequence encoding approach. With encoding technique, each instruction word may be substituted with index to ascending order based LUT. Index posses fixed length which is equivalent to unique $\log_2$ instruction words.

Assume, number of initial instructions are $N = 8$, number of instructions is $n = 5$, and $index_{length} = encoded_{instruction\ length} = \log_2(5) = 3\ bits$. With this, compression ratio is evaluated with Eq. (1) given below:

$$size_{\ original\ instruction} = W * N$$
$$Size_{encoded\ instructions} = N * \log_2(n)$$

$$size_{decoding\ table} = size\ (table\ column) = \sum_{i=1}^{W} C_i$$

Compression ratio with Look up table using pre fix tree is computed by,

$$Compression\ ratio = \frac{N * \log_2(n) + \sum_{i=1}^{W} C_i}{W * N} \quad (1)$$

Where, W is table column number (instruction word length), N is original instructions, $'n'$ is total table entries and C_i is table column size (bits). Subsequently, to enhance compression ratio by table size reduction, either with table column size or with number of columns that has to be reduced. $'W'$ is a fixed value with reducing table column size to compress decoding table.

c) Decoding table compression

Generally, uncompressed instruction words are placed in look up table. Here, total table column is equal to instructional word length in bits. Consider, for example, if instruction length is 8 bits, total table column is 8. Reducing table cost is attained by diminishing table column size (bits). The ultimate objective to compress decoding table is to reduce total bit transitions and preserves bit indeed of saving column. Table size is 56 bits. Total table instructions are 8 with length is 3 bits. Therefore, column are

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

compressed with maximal transitions. In some cases, column is compressed and column is left devoid of compression. Table size after compression is reduced to 37 bits. $(56\ to\ 37\ bits)$.

To compute cost and to compress Look up Table, prefix tree structure is provided. Column cost is table indices occur in column. If sum is lesser than cost (n), then column is compressed. Else, column remains to be left devoid of compression. This process will be repeated in table columns. At last, function seems to provide compression table cost. This algorithm shows that bits start from either 1 or 0 based on column entry. If number of zero in column entry is more than number of ones, algorithm considered to be bits commences from 0. Else, it starts from 1.

Algorithm 1:

Input: Total table entries, Number of ones and zero entries in table entry, table width, table index length.

Output: Cost computation and compression ratio

1. Functions of prefix tree $(entry, cost)$
2. {
3. Parameter Initialization
4. Toggles = 0; cost = 0;
5. If $n_0 > n_1$ then
6. Toggle commences from 0
7. Else
8. Toggle commences from 1
9. End if

//Initialization//

10. For column 'i' to 'W' do
11. For 'n'entries do
12. If $(i)[0 \rightarrow 1\ or\ 1 \rightarrow 0]$ is valid then
13. $T(i) = T(i) + 1\ \{counting\}$
14. End of
15. End for
16. Column cost $C(i) = T(i) * L$

//validate column based compression//

17. If $n > (i)$ then {column compress}
18. Save index of each toggle
19. $cost = cost + C(i)$
20. Else
21. Maintain column // without compression
22. End if
23. Return
24. }

Attaining superior compression ratio is based on sorting. Determining optimal sorting solution is NP complete. Testing each sorting probability based on (n) entries may need $n!$ comparison. Henceforth,

ISSN: 2233-7857 IJFGCN
Copyright ©2020 SERSC

entries are sorted in two different phases. Here, initial phase is generation of Gray code for given bits (table width), then locating every table entry (conversion to decimal) to position to produce gray code. Number of transactions among entries is reduced and columns are compressed. If table comprise 2^W diverse entries (probable bits combinations) then code provides solution for sorting entries, total transactions among two successive instructions 1). When entries are lesser than 2^W, sorting will not offer solution. Distance among two entries is computed with Hamming distance, i.e. sum of bit among available entries. It is performed with XOR gate. Algorithm may return sorted entries. Sorting table complexity is provided with $O\ (n \log n)$, where $'n'$number of original instructions is provided in Look up table (entries).

Algorithm 2: Prefix tree based sorting

Input: Number of table entries, table entry length, table entries.

Output: Tree based sorted entries.

1. Tree distance = 0
2. Minimal distance = 1000 // defining minimal distance
3. New_distance = 100000 // initial value
4. Sort entries of tree based on prefix entries.
5. $A = X_i\ XOR\ X_{i+1}$
6. $Distance\ = Distance + D_i$
7. $D_i =\ \sum_{j=1}^{W} X_j$ (bit position in X)
8. $end\ for$
9. If $distance < New_{distane}$then
10. $New_{distance} \leq Min_{distance}$ then
11. Goto 18
12. Else
13. Goto 4
14. End if
15. Else {better solution is not found}
16. Goto 4
17. End if
18. Return $\{S_1', S_2', \dots, S_n'\}$ // sort entries

To demonstrate significance of table entries based sorting and to enhance compression table cost, demonstrate compressing LUT with total entries $n = 7,$ where instruction length $W = 8.$ It size goes to 47 bits. After compression, the table size reduces to 35 bits.

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

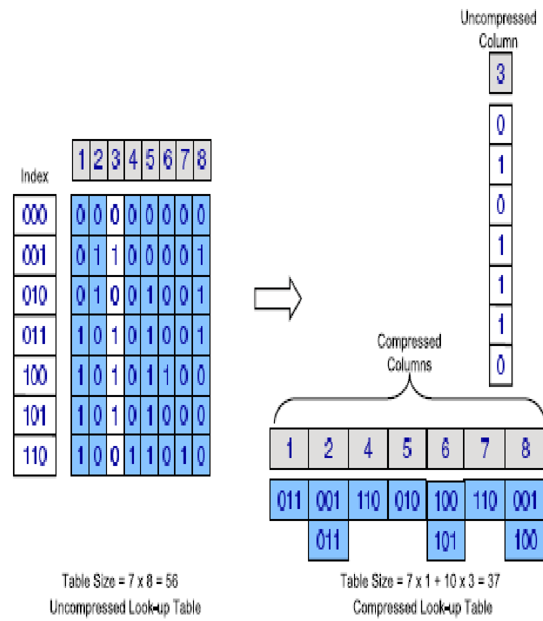

Fig 2: Code compression steps

IV. Simulation

Here, decompression hardware comprises of two essential factors: Huffman decoder and Look Table decoder. As Huffman code based algorithm is not used in Prefix tree, decompression hardware comprises LUT based decoder as in Fig 2.

With this decoder, compression column are placed on FPGA based RAMs, every column in every RAM, as uncompressed column posses own column decoder and works with other column in parallel. If total compressed column is determined as $'m'$, then total uncompressed table in ROM is $W - m$. Every decoder posses data regarding position with original LUT and total toggles. Decoder comprises comparator for scanning RAM entries till compressed instruction is determined. Every column decoders functions based on asynchronous and attain similar encoded instruction, where $'L'$ length concurrently. When it maintains encoded instruction, it determines position of every block RAM. Decoder may produces $'0'$ for position, else it may generate $'1'.B$its with uncompressed table column are directly attained from ROM. Accumulator merges bits with appropriate position and produces $'W'$ bits decompressed instruction.

If compressed instructions are placed in table with entry 101, ROM outputs is ‘1’ to accumulator at position 3. In case of bits position at 1, 2, 4, 5, 6, 7 and 8, controller may determine 101 at initial location, second place and in zeroth location. Therefore, generated bits may be provided as 1, 0, 0, 1, 0, 0 and 0 correspondingly. At last, decompressed instructions may be acquired for 101010000.

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

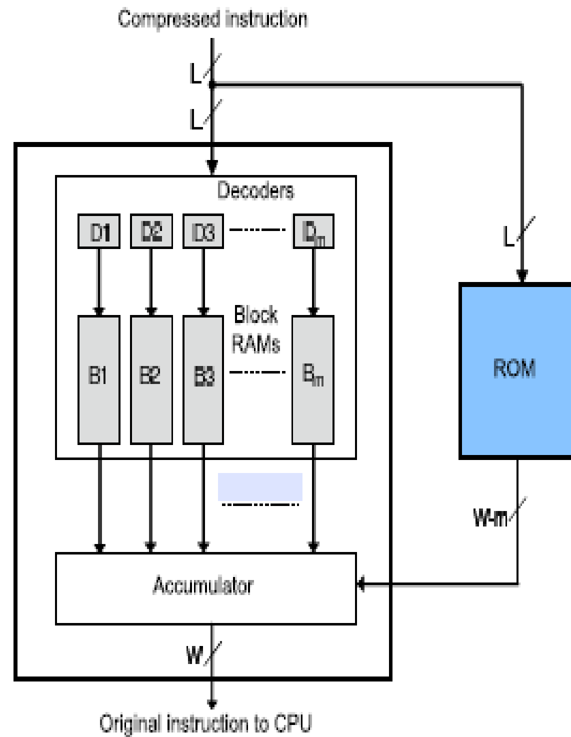

Fig 3: LUT decoder

In pre-fix tree, LUT decoder has to perform some additional tasks as in fig 3. It acquires address from CPU and evaluated compressed 1 of memory based instruction. This is performed as encoded instructions in memory are with fixed length. Therefore, handling branch target address instruction with addressed boundary is not needed, this may eradicates branch penalty produced during code compression. Compressed address will be evaluated as in Eq. (2):

$$Memory\ based\ compressed\ address = \frac{uncompressed\ address\ from\ CPU}{encoded\ instruction\ length\ in\ memory} \quad (2)$$

This decoder may be executed in VHDL and synthesis may be done in Xilinx for FPGA based prototype with Virtex family. Maximal frequency attained with LUT is 330 MHz (access time with 3 ns). Total slices required for decoding is 430.

V. Cost Minimization with LUT

Here, for reducing cost (bit size) of Look up table can be attained with two approaches:

i) Cost minimization separately

Instructions that come under this LUT will be sorted in table and compressed based on above compression techniques. This may reduce Look up table cost and will not show any influence towards instruction size as total instructions with code length may be $'i'$, $(i.e. N_i)$ is not be varied after sorting. Look up table cost is evaluated with functional Pre-fix tree.

ii) Cost minimization with complete LUT

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

Instructions that come under any available LUT will be transferred to newer LUT; if it is enhanced with optimal compression ratio. Instructions are transferred to new LUT, if index size of new Look up Table is higher than original LUT index (to preserve Huffman based Pre-fix tree property). This process may reduce total LTU by eradicating instructions from certain Tables and insert it to another. This may provide superior compression chance more columns in every table and subsequently reduce total compressed table cost. This may counter produce encoded instructions size as it may produce non-resorted LUT. For instance, moving instructions from second LUT to free entry in 4th LUT may increases size of encoded instruction by 2 times the instruction frequency, $(3 * 10 = 30\ bits)$.

If certain instructions are moved from LUT to another, efficiency is formulated as in Eq. (3):

$$E = compressed\ table\ gain - encoded\ instruction\ loss \quad (3)$$

Here, compressed gain is variance among compressed table size transferring instructions (before and after). Loss of encoded instructions is difference among encoded instruction size after and before transferring instructions. Transferring instructions that are determined form LUT to another as size difference may have indices higher than '1' bit, is reduce instructions loss in compressed table gain. It may show negative consequences with compression ratio. Henceforth, it may transfer instructions among two successive LUTs (i.e. size difference of indices is $'1'$ bit).

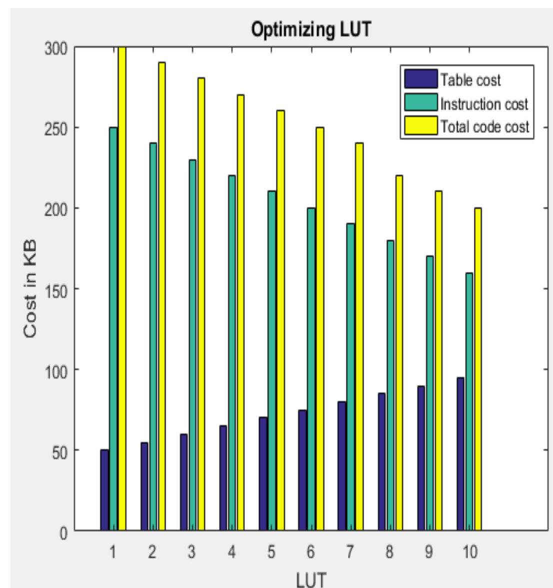

Fig 4: Optimizing LUT

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

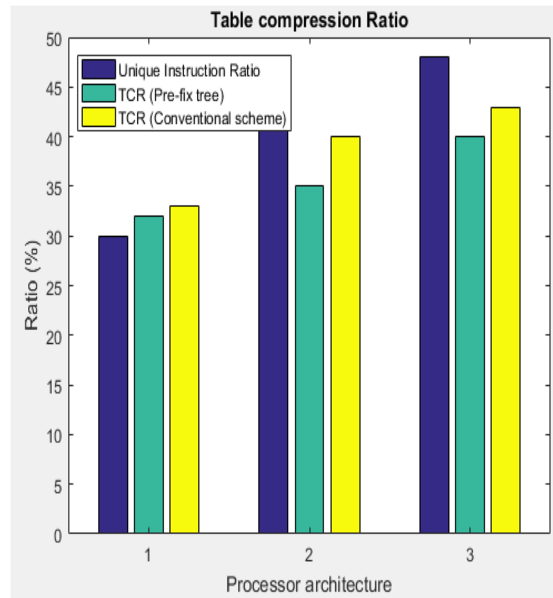

Fig 5: Table compression ratio

From Fig 4, the consequences of reduction in number of LUT may be encoded with instruction size, total compressed code size and compressed LUT size. Reducing number of LUT will be $'1'$ may attain finest compression table as it will provide superior chance to compress every column of table with re-occurring patterns as in Fig 5. This may raise cost of encoded instructions to maximal value as all instructions (less/more frequent sequence) may possess highest codeword. Subsequently, code cost may be reduced. Optimal solution shows 8 LUT here. This may reduce cost of tables, however may reduce instruction cost and total cost may be diminished.

Algorithm 3:

Input: Instruction frequency, Total LUT, Number of instructions, Minimal and Maximal transferred instructions, Table index length, tables during transformation.

Output: New LUT after instruction transfer.

1. Default. value = 0
2. $cost \min = k = 1$
3. $cost.\max = N_1$
4. $repeat = 10$

/*Evaluate table cost before transfer */

5. $cost1 = Pre - fix\ tree\ (N_1, index1, cost1, old\ transfer)$
6. $cost2 = pre - fix\ tree\ (N_2, index2, cost2, old\ trnasfer\ 2)$
7. $cost = cost1 + cost2$
8. $temp1 = old\ transfer1$
9. $temp2 = old\ transfer\ 2$
10. While $k < cost.max$ do
11. Repeat

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

12. For all instructions do
13. Transfer k random instructions
14. Evaluate loss
15. End for
16. $cost1 = prefix\ tree\ (N1, temp_t1,\ cost1$
17. $cost2 = prefix\ tree\ (N2, temp_t2,\ cost2$
18. $After\ transfer\ cost = cost1 +\ \ cost\ 2$
19. Compute efficiency with $Gain - loss$
20. $\Delta = efficiency - default\ efficiency$

/* validate transfer*/

21. If $\Delta > 0,$ then
22. $default\ efficiency = efficiency$
23. $new_{t1} = temp_t1,\ temp_t2$
24. Else
25. Return transfer instruction
26. End if
27. End for
28. K++
29. End while
30. Return $(new_t1,\ new\ _t2)$

Algorithm 3 depicts LUT cost minimization by instruction transfer among one another: After parameter initialization, cost computation is initiated from consecutive tables before providing instructions to Pre-tree function (Line 5-7). Total instructions are provided by $[k = \#\ and\ k = 1]$ for instructions. Repetition step is provided by $'repeat = 10'$. Algorithm may determines (k) random instructions from table and transforms to next table (line 12), evaluates size loss of encoded instructions (line 13). It may compute table cost after instruction transfer and gain table size. However, algorithm may evaluate efficiency. If outcomes are considered to be enhanced, algorithm maintains new table, else it may transfer two other ne tables to acquire finest efficiency (line 30). Repeat algorithm to compute cost minimization for all consecutive tables. Pre-tree based complexity is provided by $O\left[\frac{n\ (n+1)}{2}\right]\ repeat.$ Complexity may depends on total amount of original instructions in $'n'$ LUT (table entries) and total repetition step. Raising 'repeat' parameter may enhance outcomes, however it may rise algorithm time. Algorithm complexity may acquire optimal outcomes as $O(2^n - 1)$.

V. Conclusion

'Lookup Table compression' approaches are provided with Pre-fix tree model. It concentrated on overhead when compression approaches are termed as decoding table size. This work significantly concentrates on code density and provides attention towards overhead. Compression ratio requires all overhead incurred. This approach is orthogonal with certain characteristics and provided with compression approach that produces large table size. It is cast off in conventional compression approach; average compression ratio is 54%, 55% and 60% is reported for PowerPC, MIPS and ARM processor respectively.

REFERENCES

[1] T. Bonny and J. Henkel. Using Lin-Kernighan Algorithm for Look-Up Table Compression to Improve Code Density. Proc. of the 16h Great Lakes Symposium on VLSI-(GLSVLSI'06), pp. 259-265, Philadelphia, USA, April 2006.

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC

[2] T. Bonny and J. Henkel. Instruction Re-encoding Facilitating Dense Embedded Code. IEEE/ACM Proc. of Design Automation and Test in Europe Conference (DATE08), pp. 770-775, Munich, Germany, March 2008.

[3] T. Bonny and J. Henkel. Efficient Code Density Through Look-up Table Compression. IEEE/ACM Proc. of Design Automation and Test in Europe Conference (DATE07), pp. 809-814, Nice, France, April 2007.

[4] M. Collin and M. Brorsson, Low Power Instruction Fetch using Profiled Variable Length Instructions. in Proceedings of the IEEE International SoC Conference, pp. 183-188, 2003.

[5] L. Benini, D. Bruni, A. Macii and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. Proceedings of the conference on Design, Automation & Test in Europe DATE02, pp. 449-453, 2002.

[6] L. Benini, A.Macii and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. International Symposium on Low Power Electronics and Design, pp. 322-327, Aug. 2001.

[7] H. Lekatsas, J. Henkel, V. Jakkula and S. Chakradhar. A unified architecture for adaptive compression of data and code on embedded systems. Proc. of 18th. International Conference on VLSI Design, pp. 117-123, 2005.

[8] H. Lekatsas, J. Henkel, and W.Wolf. H/S Embedded Systems: Design and simulation of pipelined decompression architecture for embedded systems. Proceedings of the international symposium on systems synthesis, 2001.

[9] B. Chazelle, A Minimum Spanning Tree Algorithm with Inverse Ackermann Type Complexity. Journal of the ACM (JACM), Vol. 47, No. 6, pp. 10281047, November 2000.

[10] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an One-cycle Decompression Hardware for Performance Increase in Embedded Systems. in Design Automation Conference (DAC'02), pp. 34-39, 2002.

[11] Y. Xie, W. Wolf and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. In IEEE Transactions on Very Large Scale Integration (VLSI) System, Vol. 14, No. 5, pp. 525-536, 2006.

[12] Y. Xie, W. Wolf and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. ACM Proceedings of 15th International Symposium on System Synthesis, pp. 138-143, 2002.

[13] Y. Xie, W. Wolf and H. Lekatsas. Compression ratio and decompression overhead tradeoffs in code compression for VLIW architectures. In International Conference on ASIC, pp. 337-340, 2001.

[14] C. Lin, Y. Xie and W. Wolf. LZW-based code compression for VLIW embedded systems. Proceedings of the Design, Automation and Test in Europe conf. (DATE04), pp. 76-81, 2004.

[15] Seong and P. Mishra. A Bitmask-based Code Compression Technique for Embedded Systems. 24th IEEE/ACM International Conference on Computer-Aided Design (ICCAD06), pp. 251254, 2006.

ISSN: 2233-7857 IJFGCN
Copyright ⓒ2020 SERSC