Use Lossless Data Compression in Embedded Systems

Himali B. Kotak, Dr. J. S. Shah

1 A. V. Parekh Tech Insti. Computer Deptt ,Rajkot, Gujarat,India
Email: Hemalikotak@yahoo.com
2 L.D. Engg College ,Computer Deptt, Ahmedabad, Gujarat, India
Email: jssld@yahoo.com

Abstract—Data Compression refers to reducing the amount of data required to represent a source of information. Embedded systems are often sensitive to space, weight, and cost considerations. Reducing the size of stored programs can significantly improve these factors. The main goal of the system is to make use of LZW data compression technique to compress & decompress the text on an Embedded Processor. We present a way by which we can implement data compression algorithm on an embedded system. Implementation of LZW on embedded processor gives better result as compare to Huffman Compression.

Index Terms—Compression, Embedded, LZW, Processor, Huffman

I. INTRODUCTION

Many of today’s embedded systems are providing more sophisticated solutions to a wide variety of applications and industries. With this increase in sophistication, there is a corresponding increase in the amount of data being collected and stored. This rise in data volume is placing even more emphasis on design solutions that need to meet stringent performance requirements while staying within cost targets. Collecting and storing large amounts of data can create a number of technical challenges. In addition, the embedded system itself can become overtaxed by the increased burden imposed by processing large amounts of data. Faced with the issues of collecting and storing data, embedded system designers are finding data compression as a viable means of dealing with large amounts of data.

Memory is one of the most restricted resources in many modern embedded systems. Code compression can provide substantial savings in terms of size. In distribution and embedded systems, data compression is often used to reduce the size of flash RAM and transmission data, while a rapid decompression speed enables faster rebooting of the compressed program code. Faced with the issues of collecting and storing data, embedded system designers are finding data compression as a viable means of dealing with large amounts of data.

II. EMBEDDED DATA COMPRESSION

There are a significant number of data compression techniques available to meet the specific needs of most applications. But whatever the solution may be, all compression algorithms fall into two main categories, lossy and lossless data.

Compression ratio, widely accepted as a primary metric for measuring the efficiency of code compression, is defined as:

\[
\text{Compression Ratio} = \frac{\text{Compressed Program Size}}{\text{Original Program Size}} \quad \text{Eqn. 1.1}
\]

Therefore, the smaller the compression ratio is, the better the compression technique. Dictionary-based code compression techniques are popular because it provides both good compression ratio and fast decompression mechanism. The basic idea is to exploit repeating instruction sequences by using a dictionary.

III. THE CONVENTIONAL LZW DATA COMPRESSION

LZW is one of the most popular compression programs available and is the easiest dictionary-based algorithm to implement into an embedded design. It is the compression technique used to generate TIFF and GIF files. It is also the compression technique used for UNIX’s Compress program. LZW was also the compression technique used by computer programs that claimed to "double your hard drive's capacity!".

At the heart of the LZW compression program is the dictionary which translates string patterns from the original data into a single token. The goal of the dictionary is to replace redundant data strings with single value tokens.

A. LZW Encoding Algorithm

When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes. As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table.

```plaintext
s = empty string;
while (there is still data to be read) {
  ch = read a character;
  s = s + ch;
  if (s appears in the dictionary) {
    // output code
  } else {
    // output code and add s to dictionary
    s = ch;
  }
}
```
if (dictionary contains $s+ch$)
    
}{
    s = s+ch;
}
else
{
    encode s to output file;
    add $s+ch$ to dictionary;
    s = ch;
}
}

B. LZW Decoding Algorithm

The LZW decompressor creates the same string table during decompression. It starts with the first 256 table entries initialized to single characters. The string table is updated for each character in the input stream, except the first one. Decoding achieved by reading codes and translating them through the code table being built

prevcode = read in a code;
decode/output prevcode;
while (there is still data to read)
{
    currcode = read in a code;
    entry = translation of currcode from dictionary;
    output entry;
    ch = first char of entry;
    add (translation of prevcode)+ch to dictionary;
    prevcode = currcode;
}

C. Advantages of LZW over Huffman

LZW requires no prior information about the input data stream. LZW can compress the input stream in one single pass. Another advantage of LZW its simplicity, allowing fast execution.

IV. DICTIONARY BASED APPROACH IN EMBEDDED DATA COMPRESSION

Dictionary-based code compression techniques provide compression efficiency as well as fast decompression mechanism. The basic idea is to take the advantage of commonly occurring instruction sequences by using a dictionary. The repeating occurrences are replaced by a codeword that points to the index of the dictionary that contains the pattern. The compressed program consists of both codeword and uncompressed instructions. Figure 2 shows an example of dictionary based code compression using a simple program in binary.

The binary consists of ten 8-bit patterns i.e., total 80 bits. The dictionary has two 8-bit entries. The compressed program requires 62 bits and the dictionary requires 16 bits. In this case, the compression ratio is 97.5% (using Equation (1.1)). This example shows a variable length encoding.

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Compressed Program</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 ....</td>
<td>0 1 1000010 0</td>
<td>00000000 01000010</td>
</tr>
<tr>
<td>01001110 ....</td>
<td>0 1 1000010 0</td>
<td>00100110 01001100</td>
</tr>
<tr>
<td>11000000 ....</td>
<td>0 1 1100000 0</td>
<td>11000000 11000000</td>
</tr>
</tbody>
</table>

V. IMPROVED DICTIONARY BASED APPROACH IN EMBEDDED DATA COMPRESSION

Recently proposed techniques improve the dictionary based compression technique by considering mismatches. The basic idea is to determine the instruction sequences that are different in few bit positions (hamming distance) and store that information in the compressed program and update the dictionary (if necessary). The compression ratio will depend on how many bit changes are considered during compression. Figure 3 shows the encoding format used by these techniques for 32-bit program code

It is obvious that if more bit changes are allowed, more matching sequences will be generated. However, the size of the compressed program will increase depending on the number of bit positions.

VI. CONCLUSIONS

Embedded systems are constrained by the memory size. Code compression techniques address this problem by reducing the code size of the application programs. Dictionary based code compression techniques are very popular since they generate good compression by exploiting repeating patterns. As is true of any compression technique, it is important to make sure that the structure of the data is conducive to the LZW compression algorithm. Very impressive compression ratios can be achieved with the LZW algorithm, as long as the data is structured properly. Another thing to consider is the execution time of the LZW algorithm. Most of the algorithm’s time is
involved in building and searching strings in the dictionary. Some applications may not work well with the time required to compress the data. If the data is the right fit and your system can tolerate the time required to run the algorithm, the LZW compression algorithm might be the right fit for your application.

ACKNOWLEDGMENT

The authors wish to thank A, B, C. This work was supported in part by a grant from XYZ.

REFERENCES

[1] New LZW Data Compression Algorithm and its FPGA implementation Wei Cui
[2] Embedded.com The official cite for embedded development community