IT 08 012
Examensarbete 30 hp
April 2008

Transparent Data Compression and Decompression for Embedded Systems

Gustav Tano

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Transparent Data Compression and Decompression for Embedded Systems

Gustav Tano

Embedded systems are often developed for large volumes; the cost for storage becomes a critical issue in embedded systems design. In this report we study data compression techniques to reduce the size of ROM in embedded applications. We propose to compress RW data in ROM to be copied into RAM at the system initialization. The RW data will be collected and compressed during compiling and linking but before loading into the target platform. At the system initialization on the target platform, an application program will decompress the compressed data. We have implemented three compression algorithms and measured their performance with different configurations and data. Our experiments show that the Lempel-Ziv-Welch and Burrows-Wheeler Transform algorithms performs very well and are both suitable for data compression on embedded systems.

Handledare: Gunnar Blomberg
Ämnesgranskare: Wang Yi
Examinator: Anders Jansson
IT 08 012
Sponsor: IAR Systems

Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Introduction

Embedded systems resources are usually as small and cheap as possible. If a product is produced in large volumes each hundred bytes of memory that can be reduced may save substantial amounts of money. Memory usage mostly depend on how the software is written and what kind of optimizations that were used when compiling. This paper will discuss an approach to shrink the memory footprint - compression of Read-Write (RW) data.

Data compression is a method of representing information in a more compact form. In computer science we are interested in representing information using fewer bits than the original representation. The downside with compression is the time or computational resources the algorithm consumes as it is codes and decodes data.

Data compression is used everywhere in the computer and electronic industry. It is technology that can be found in mp3 and DVD players, GSM phones, digital cameras, digital television and nearly all other media devices you can think of. Data compression is one of the important technologies that enables us to send, receive, view multimedia content. Computer users may be familiar with the ZIP-, RAR-, MP3- or JPEG-formats that all include data compression.

In this implementation the decompression will be executed in some kind of embedded system, which usually has limited resources in terms of computational power and the amount of available RAM and ROM memory. Because the decompression algorithm is running on an embedded system environment

we are more concerned with the decompression efficiency than the compressor‘s.

Data that can be compressed is initialized data that will be changed during runtime of the application. An example of initialized data is global arrays. This kind of data is stored at a ROM and will be copied to a RAM during startup. Data that is compressed will need less ROM but the same amount of RAM. Read-only data usually do not need to be copied to the RAM and can stay permanent on the ROM, hence will not be compressed.

The main objective of this thesis is to study different compression algorithms, to implement them and to evaluate which are suitable for use in an embedded system application. We also want to get an understanding on how much data that is required to justify the compression application.

Organization of this paper

The report is divided into 6 chapters. *Chapter 2* consists of background information about compression and the compression algorithms that we have implemented. *Chapter 3* presents how we have chosen and tested the compression algorithms. *Chapter 4* shows the results from the various tests we made. *Chapter 5* contains related work and *Chapter 6* is a summary with discussion about the work and results. *Appendix A* holds a glossary of terms used in this thesis.

Chapter 2

Background

2.1 Data compression

Data compression is the process of encoding information to a representation using fewer bits than the original representation. Data compression is done by computer programs that identify and use patterns that exist in the data. Data compression is important in the area of data transmission and data storage. Data that is compressed can be characters in a text-file or numbers that represent audio, video or images. By compressing data it is possible to store more information on the same amount of memory. If data can be compressed to half of its size then the capacity of the memory is doubled. The compression however has a cost. The compression and decompression consumes computation time and requires, in some cases, a lot of runtime memory. You also need to have the decoding program at the receiving end.

2.2 Run Length Encoding

Run Length Encoding (RLE) [1] is a very simple form of compression. The only pattern it compresses is a series of equal bytes. Such series are found in some encodings of simple graphics images such as icons and line drawings. RLE can be very powerful when it is preceded by another algorithm such as Burrows-Wheeler Transform.

Each series of bytes is preceded by a prefix with the length of the series. The prefix code is of the type `signed char`. If the prefix code `px` is smaller

Original string:
0xAA 0xBB 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xDD 0xDD 0xDD 0xDD 0xDD 0xEE 0xFF 0xFA 0xFB

Result, post RLE compression:
0x01 0xAA 0xBB 0xF9 0xCC 0xFB 0xDD 0x03 0xEE 0xFF 0xFA 0xFB

Figure 2.1: Description of the Run-Length-Encoder algorithm.

than zero it indicates that the following byte shall be repeated `-px` times. If the prefix code `px` is larger or equal to zero that means that there are `px+1` single bytes to be printed before the next prefix code.

Figure 2.1 gives an example of RLE compression on a byte array. In the result table there is a line break for each block of data. The code can be translated to two single bytes `0xAA 0xBB`, followed by seven bytes of `0xCC`, five bytes of `0xDD` and four single bytes `0xEE 0xFF 0xFA 0xFB`. The original representation is 18 bytes long and the compressed code is 12 bytes.

2.3 Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT) was invented by Michael Burrows and David Wheeler in 1994 at Digital Systems Research Center in Palo Alto, California [2]. BWT is a lossless block-sorting compression algorithm. The algorithm does not reduce the size of the data, it permutes a block of data to a sequence that is easier to compress by simple compression algorithms. The significant feature of this transformation is that is tends form groups of equal bytes in the block. Compression algorithms that successfully can be applied to the transformed string are RLE or move-to-front transform [1] combined with Huffman or arithmetic coding.

Input (IS)	Step 1 Rotations (RS)	Step 2 Sorted rows (SR)	Step 3 Encoded Output (EO)
xABCBCBy	xABCBCBy ABCBCByx BCBCByxA CBCByxAB BCByxABC CByxABCB ByxABCBC yxABCBCB	ABCBCByx BCBCByxA BCByxABC ByxABCBC CBCByxAB CByxABCB xABCBCBy yxABCBCB	xACCBByB Index: 6

Figure 2.2: Burrows-Wheeler Transform encode.

2.3.1 Encoding

The BWT algorithm does not process the input byte for byte, instead if handles blocks of bytes. If we consider the block as a rotating buffer, we can list all possible shifts of the buffer under each other (see column 2 in figure 2.2). The second step of the algorithm is to sort these rows in lexicographical order. The third step is to output the last byte of the sorted rows. Figure 2.2 shows each step of the algorithm. Each block has a prefix that holds the block length and an index number. The index number is used to decode the block and it tells which row that holds the original string.

2.3.2 Decoding the naive way

The decoding algorithm is a bit more complicated than the encoder. By using the encoded block (`EO`) and an the row index (`I`), we can recreate the matrix created in the encoding algorithm. First we create an empty table with the height and the width of the encoded block. We will recreate the matrix column by column. The first step is to insert `EO` as a column in the table (see "Add 1" in figure 2.3). The second step is to sort the rows in the table ("Sort 1"), and then add another instance of the block `EO` (step "Add 2"). These steps are then repeated for each column in the table. After the last sorting (step "Sort 7") we have the matrix `SR` from the encoding phase.

Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4	Add 5
x	A	xA	AB	xAB	ABC	xABC	ABCB	xABCB
A	B	AB	BC	ABC	BCB	ABCB	BCBC	ABCBC
C	B	CB	BC	CBC	BCB	CBCB	BCBy	CBCBy
C	B	CB	By	CBy	Byx	CByx	ByxA	CByxA
B	C	BC	CB	BCB	CBC	BCBC	CBCB	BCBCB
B	C	BC	BC	BCB	CBy	BCBy	CByx	BCByx
y	x	yx	xA	yxA	xAB	yxAB	xABC	yxABC
B	y	By	yx	Byx	yxA	ByxA	yxAB	ByxAB

Sort 5	Add 6	Sort 6	Add 7	Sort 7	Add 8	Sort 7
ABCBC	xABCBC	ABCBCB	xABCBCB	ABCBCBy	xABCBCBy	ABCBCByx
BCBCB	ABCBCB	BCBCBy	ABCBCBy	BCBCByx	ABCBCByx	BCBCByxA
BCByx	CBCByx	BCByxA	CBCByxA	BCByxAB	CBCByxAB	BCByxABC
ByxAB	CByxAB	ByxABC	CByxABC	ByxABCB	CByxABCB	ByxABCBC
CBCBy	BCBCBy	CBCByx	BCBCByx	CBCByxA	BCBCByxA	CBCByxAB
CByxA	BCByxA	CByxAB	BCByxAB	CByxABC	BCByxABC	CByxABCB
xABCB	yxABCB	xABCBC	yxABCBC	xABCBCB	yxABCBCB	**xABCBCBy**
yxABC	ByxABC	yxABCB	ByxABCB	yxABCBC	ByxABCBC	yxABCBCB

Figure 2.3: Burrows-Wheeler Transform decode.

The index I tells us which line in the table that represents the original input string `IS`.

The decompression algorithm described above is very slow because of the large amount of sorting that is performed, this is not desirable. However, this algorithm can be optimized if the recurrent sorting of each row is avioded. Below we will describe how to do this and get an algorithm that has linear complexity, i.e. `O(n)`.

2.3.3 Decoding the efficient way

To decode a block in an efficient manner the algorithm first constructs two arrays with meta information about the elements of the input, `block[]`. The first array is `predicate[]`, and it is of the same length as the block. The value in `predicate[i]` is the number of times the value at `block[i]` has

```
i = index;  // i is the index of the last byte in block[].

for(j = blockSize - 1; j >= 0; j--)
{
  unrotated[j] = block[i];
  i = predicate[i] + count[block[i]];
}
```

Figure 2.4: The important for-loop in BWT algorithm.

appeared in the substring `block[0...i-1]`, (i.e. the string between position `0` and `i-1` in `block[]`). The second array is `count[]` and it is 256 bytes long. In this array each element `count[i]` contain the number of bytes in `block[]` that lexigraphically proceeds the byte at `block[i]`. The verbatim code in figure 2.4 show how these two arrays combined with the encoded array will generate an decoded block of bytes. During each iteration of the for-loop the element `unrotated[j]` are assigned the value in `block[i]`. At the first iteration the value of index `i` is given (see output in figure 2.2). At each of the following iterations a new `i` is calculated. When a block is decoded another one is read and the algorithm must calculate new `predicate[]` and `count[]` arrays.

2.4 Lempel-Ziw - LZ77, LZ78

LZ77 [3] and LZ78 [4] published by Abraham Lempel and Jacob Ziv are both dictionary compression algorthims. They are also known as LZ1 and LZ2. The input stream is encoded by replacing parts of the stream with a reference to a part that has already passed. The reference is typically a pointer to the beginning of the referenced string and the length of it. Both the encoder and decoder keep track of a fixed length sliding window.

2.5 Lempel-Ziw-Welch

Lempel-Ziv-Welch (LZW) [5] is a lossless compression algorithm. Its creators are Abraham Lempel, Jacob Ziv amd Terry Welch. The LZW algorithm is an improved variant of the LZ78 and LZ77 algorithms published by Lempel and

Ziw in 1977 and 1978. Welch published the LZW algorithm in 1984. The algorithm is not optimal since it only does a limited analysis of the input data.

2.5.1 Patent issues

There have been various patents issued in the USA and other countries for the LZW algorithms [6]. There have been two US patents issued for the LZW: U.S. Patent 4,814,746 on June 1, 1983 and U.S. Patent 4,558,302 on June 20, 1983. In June 2003 the U.S. patent on the LZW algorithm expired and by July 2004, the patents in United Kingdom, France, Germany, Italy and Canada had expired.

2.5.2 The dictionary

The central part of the LZW algorithm is the dictionary. Both the encoding and decoding algorithms create the same dictionary, there is no need to send the dictionary along with the encoded message. The dictionary will keep track of stored strings and a reference code (see figure 2.5). The reference code has a fixed length, depending on the size of the dictionary used. References with the length of 12 bits will enable a dictionary of a maximum size of 4096 entries. The smallest possible size of the dictionary is 512 entries i.e. 9 bit references. If the library uses 12 bit references that means that all characters that are stored are 12 bits long, hence both the encoder and decoder must use dictionaries of the same size.

The first 256 entries in the dictionary is the full ASCII (American Standard Code for Information Interchange) alphabet. These characters do not need to be stored since they already have a code assigned to them. The algorithm will build the dictionary from the stream that is being compressed or decompressed. The algorithm stores each set of two characters that are not already in the dictionary. When a set of characters are matched in the dictionary the algorithm also checks if the set with the next character in sequence appended are matched in the dictionary. If there is a match, a longer string is tried. If there is no match, the unmatched string is added to the dictionary. This procedure continues for each character in the string until the dictionary is full. Depending on which implementation of the algorithm we can either reset the dictionary and start over or use the current dictionary

Original string: A B B A A B B A A B A B B A A A A B A A B B A .

Code:

A	B	B	A	AB	BA	AB	ABB	AA	AA	BAA	BBA
65	67	67	65	256	258	256	260	259	259	261	257

Dictionary			
Index	Characters	Index	Characters
256	AB	263	ABBA
257	BB	264	AAA
258	BA	265	AAB
259	AA	266	BAAB
260	ABB	267	BBA
261	BAA	262	ABA

Figure 2.5: Lempel-Ziv-Welch algorithm.

without adding additional strings. Of course both the encoder and decoder must use the same principle when the dictionary is full.

2.5.3 The encoder

Section 2.5.2 describes how sets of characters are added to the dictionary. When the input string is read the algorithm tries to match as long string as possible in the dictionary reading one character at a time. The longest set of characters that match an entry in the dictionary is replaced by a number referencing to that entry.

2.5.4 The decoder

The decoder operates in a similar fashion as the encoder (section 2.5.3) when using the dictionary, described in section 2.5.2. Instead of replacing found sets of characters with codes it replaces n bit codes with a set of characters.

2.6 Other compression algorithms

Here are a couple of compression algorithms that we choose not to implement.

2.6.1 Prediction by Partial Matching

Prediction by Partial Matching (PPM) is a context-based algorithm. It was first published by Cleary and Witten [7] in 1984. The disadvantage of PPM has to e.g. the LZW algorithm is that it has slower execution time. PPM algorithms try to predict incoming symbols by analyzing previous read symbols. The predictions are usually made by ranking the symbols. Research on PPM implementations were made on the 1980s but the implementations were not popular until the 1990s because of the large amount of RAM that was required.

2.6.2 Arithmetic Coding

Arithmetic coding is like Huffman coding a form of variable-length entropy encoding. It converts a string into a representation where frequently used characters are represented using fewer bits and infrequently used characters as symbols with more bits. The purpose of this approach is to create a representation of the string using fewer bits total. The differences between arithmetic encoding and other similar techniques is that is encodes the entire message into a single number n where, (n ¿= 0; n ¡ 1.0).

2.6.3 Huffman Coding

The Huffman coding algorithm is a very popular algorithm used for lossless data compression [1]. The algorithm is named after its inventor, David Huffman. As part of an assignment in a course in information theory at Massachusetts Institute of Technology, Huffman developed the technique that produces the "Huffman Code". The "Huffman code" are synonym with "prefix codes". The definition of prefix codes are "the bit string representing some particular symbol is never a prefix of the string representing any other symbol". Huffman coding is optimal for coding a known input string symbol by symbol, but it does not consider recurring patterns of sets of symbols. The LZW are an example of a dictionary algorithm that make use of this and is often better at compressing data.

2.6.4 Dynamic Markov Compression

Dynamic Markov Compression is a data compression algorithm developed by Gordon Cormack and Nigel Horspool. It is similar to the PPM algorithm in its implementation of predictive arithmetic coding. The difference is that the input is predicted one bit at a time. One unfortunate feature that it shares with the PPM algorithm is that it consumes a lot of memory.

Chapter 3

Methodology

This Master's Thesis is executed at IAR Systems AB in Uppsala, Sweden. The purpose of this thesis is to study different kinds of data compression algorithms that can be implemented into the IAR Systems linker software, Ilink. Ilink is a part of the IAR Systems Embedded Workbench (EW), which is an Integrated Development Environment (IDE) including an editor, compiler and debugger.

Before the work on this thesis started, there was support for a very basic compression algorithm in Ilink. If the linker estimated that it was advantageous to do so, initialized data in the data segment were compressed. The original compression algorithm is a variant of the RLE algorithm which only compresses consecutive bytes initialized to the value `0x00`.

This thesis work consist of three major parts. The first part is to evaluate different compression algorithms and to select two or more that seem interesting in this kind of environment. The second part is to implement the chosen algorithms and the last part is to measure and compare the implementations. The special characteristic with this particular implementation of compression is that there are high requirements on the decompression phase, which will execute on hardware with limited resources.

Compression algorithms are nothing new, most of the existing ones have been around since the 1980's and have been tested and modified to a large extent. However every implementation has its own characteristic and we are interested in finding out how the properties of the embedded system will co-

operate with the compression algorithm. The interesting questions we want to answer is:

- How will the compression work in the linker software?
- How will the decompression perform in an embedded environment?
- How large will the decompression algorithm become?
- How much application data is required to justify the compression?

The first step of this project was to implement a simple extension to the zero compressor to get familiar with the system. We implemented the Run Length Encoding algorithm. This implementation gave good experience before implementing the more complicated ones later. In the following subsections we will describe the evaluation, implementation and finally the testing of the compression algorithms.

3.1 Evaluation of compression algorithms

The requirements on the implementation of the compression algorithms are that they shall utilize small amounts of memory and have an execution time that is acceptable, even for large amounts of data. There is also another parameter to consider because the decompresser sent along with the compressed data - the code size of the decompresser. That means that an algorithm which is possible to implement using a small amounts of code is desired.

3.1.1 Lempel-Ziv-Welch

There was one algorithm that we decided to test at the beginning of this thesis work, the LZW-algorithm (see section 2.5). It is a common algorithm for this kind of applications. The memory usage and execution time of the decompresser of an straightforward LZW implementation have a favorable worst case execution time [10]. Execution and memory usage combined with the compression performance gives us an algorithm that fulfills all of our requirements. The LZW algorithm will be implemented with different dictionary sizes and with and without a flushing dictionary.

3.1.2 Burrows-Wheeler Transform

The BWT algorithm requires computational complexity comparable to that of the LZW algorithms [9] but has some performance improvements over LZ codes [11]. The original BWT algorithm combined with MTF and Arithmetic coding [2] gives a 33% [9] improvement in average compression compared to Unix compress and 9.7% improvement over gzip (gzip implements a version of the Lempel-Ziv algorithm) in the cited tests. BWT prepares the byte stream for compression by e.g. a RLE algorithm. The RLE algorithm is very simple and will not consume much memory or have a long execution time. These properties make BWT serious alternative to the LZW algorithm. Since the BWT algorithm can be configured with different block sizes we have implemented three block sizes to see the differences in performance. Our implementation does not include the move-to-front encoder or the Huffman coder because of its code size. As mentioned above this implementation combines BWT with the RLE algorithms.

3.1.3 Run-Length-Encoding

The RLE compression will be tested by itself and be compared with the BWT algorithm. The RLE implementation is much smaller compared to the LZW and the BWT algorithms in code size. Run-time memory usage is also a lot smaller. The RLE algorithm is very simple and does not require a buffer. The only data that need to be in memory is a counter byte and the last byte from the input stream.

3.2 Implementation

This project includes two kind of implementations. The first one is the encoder which is implemented in the linker, Ilink. The algorithm in the linker is implemented in C++. The second one is the decoder which is implemented i C and will run on the target environment.

In the linker the task for the compression algorithm is to collect all the bytes of the initialized data and run the compression on these. There is no predetermined structure of the data that can be used for the compression. The decompression algorithm will be linked together with the rest of the program that is run in the linker. The decompression will run at the startup phase

of the target environment (i.e. an embedded system). The decompression algorithm is kept as simple as possible. It does not use any of the libraries because the decompression is executed very early in the initialization of the system. The input to the decompression is a byte stream and it will return a compressed byte stream.

3.2.1 LZW configurations

The LZW compression can be configured with different sizes of the dictionary. The size of the dictionary affects the number of bits with which each symbol will be represented. The smallest size of the dictionary is 512 entries, which requires a minimum of 9 bits per symbol. The next step is 10 bit symbols which gives a dictionary of 1024 entries. The size of the dictionary affects the amount of memory that is required. The second configuration available in this configuration is what the algorithm is supposed to do when the dictionary is full. It can either continue as before without any new insertion or it can flush the dictionary and restart the insertion. The measuring of the LZW compression has been done with the flush implemented and dictionaries of sizes 512 and 1024 entries. Tests with the flush function disabled were run with dictionaries of sizes 512, 1024 and 4096 entries. The reason we wanted to test the dictionary sizes 512 and 1024 is that we wanted to see if those small implementations still proved to be effective.

3.2.2 BWT configurations

The BWT algorithm operates on blocks of a specified size. The block size of the coder and the decoder must be equivalent. A big block of size `n` need a lot of memory to operate and will take longer time to execute than two blocks of size `n/2`. However bigger blocks will most likely give better compression. The block sizes used in these experiments are blocks containing from 500 bytes up to 4000 bytes. To change the block size both the linker and the decoding functions need to be recompiled.

3.3 Testing

To evaluate the compression we have run two test suites and an internal suite named Makebench and also an externally created benchmark test suite

named EEMBC (see section A. The EEMBC tests provide realistic test cases for the linker. The test suite are supposed to behave like actual automotive, networking or office applications used in the embedded industry. There were 37 different EEMBC applications available to measure. Of the 37 applications about 10 ten of them had enough read/write data to be interesting for the compression.

To run the compression tests we have created a Ruby-script [12] that runs all applications in a selected test suite with a selected compression algorithm and extracts the amounts of data before and after compression. The information about the amounts of data is extracted from the .map-files created by the linker.

Chapter 4

Results

There are a couple of properties to measure in an implementation of this kind. It is important to test that the algorithm implementation is correct i.e. that the data is not distorted in any way by the compression or decompression. Measurements of the compression performance between the different compression algorithms as well as execution time and ROM and RAM memory usage are interesting. These properties are important because we are running on an embedded system with limited resources. The special characteristic of this implementation is that the decompresser is sent along with the compressed data to the target environment, therefore the interesting measure we want look at is:

```
(reduced data) - (decompresser size) = bytes of ROM reduced.
```

The amount of compression achieved and memory consumption can be read in the list-files that is generated by the linker. Execution time can be measured by different methods. One way to do this is to run in on embedded hardware and try different data sizes. Another way is to run the code in a simulator e.g. the IAR Systems C-SPY debugger, and let the software count how many processor cycles the algorithm needs to complete.

4.1 Compression results

The results from measurements of the implemented compression are presented in table 4.1. The table show the size of the data with and without

compression. Every application that has been run is represented by an id. The column named "Orig." contain the size of data before compression. The column with the name "LZW 9" indicates that it is the LZW algorithm with symbols of 9 bits and a maximum dictionary size of 512 entries. LZW 10 has a maximum dictionary of 1024 entries and LZW 12 of 4096. The last two columns with a F after the number implies that it has flushing enabled (see section 2.5). The best result of each row is marked by italic and bold font. The amount of data varies a lot in the test suite and we have only included the 15 most interesting test cases in this paper. Due to secrecy issues around the test applications we can not present the exact data each case contain. The tables show that the BWT performs best in most of the tested application. The differences in compression efficiency between the LZW and the BWT algorithms are rather small. Table 4.2 show the compression factor in bits per byte for three of the best performing implementations. In table 4.3 we show an estimation of how many bytes that are reduced by the compressor if we add the size of the decompresser i.e. `decompresser size - reduced data`. If the result is negative that means that the compression is effective and actually reduces ROM usage for that specific application.

4.2 Size of the decompresser

The tables 4.4 and 4.5 show the size of each decompression application in the code and data memory. The tables show each implementation compiled with optimizations for speed by the IAR Systems iccarm compiler. The actual size of the application when it is running are a bit larger. Some of the applications require a big amount memory during runtime. Below is a summary of the data structures that require memory in each of the applications.

The LZW decompresser program has a struct for the dictionary. The size of the dictionary is determined by the number of bits with which each symbol is represented. The size is set by the `#define CODE_LEN` macro. Like the verbatim code in figure 4.2 shows that the size of the dictionary is `DICT_SIZE * sizeof((uint8_t) + sizeof(uint_16t))`. Hence, with a dictionary of size 1024 the array will require approximately `1024 * (1 + 2) = 3072 bytes` of runtime memory.

The BWT decompresser has three big arrays with the same size as the block

Size of initialized data (bytes)									
Test	Orig.	RLE	BWT 500	BWT 1K	BWT 2K	LZW 9	LZW 10	LZW 12	LZW 9F	LZW 10F
1	7633	8577	7255	7270	7390	***6464***	6640	7094	6529	6614
2	1759	1863	1616	1570	1564	1466	1448	1634	***1374***	1438
3	1567	1576	1541	***1538***	1544	1794	1963	2079	1619	1750
4	1631	1559	536	438	***395***	646	573	687	521	569
5	1699	1713	***531***	541	569	1068	775	929	742	775
6	4335	4393	***3342***	3454	3480	4202	4413	4283	3821	3925
7	1058	1089	470	452	***450***	482	470	564	***443***	470
8	11980	13018	***5443***	5576	5810	7010	7159	6518	6320	6490
9	19346	19570	11389	11178	9883	11592	9310	***7370***	14908	12190
10	62554	8556	6457	3446	***2491***	9664	5809	6971	13718	7819
11	14171	14333	7766	7442	***7398***	12181	11097	8249	7645	7510
12	4616		3863	3468	***2956***	3371	3065	3116	3465	3179
13	37448	37982	20014	20014	***19448***	36348	32137	32907	21433	21270
14	1300		1152	1118	1107	992	***922***	1106	1032	922
15	665		***596***	611	611	611	734	881	666	734

Table 4.1: Results from the tests of RLE, LZW and BWT compression.

Bits per bytes			
Test	LZW 10 F	BWT 1000	BWT 2000
1	6.93	7.62	7.75
2	6.54	7.14	7.11
3	8.93	7.85	7.88
4	2.79	2.14	1.93
5	3.64	2.55	2.68
6	7.24	6.37	6.42
7	3.55	3.41	3.40
8	4.33	3.72	3.88
9	5.04	4.62	4.09
10	0.99	0.44	0.32
11	4.23	4.20	4.18
12	5.51	6.01	5.12
13	4.54	4.28	4.15
14	5.67	6.88	6.81
15	8.83	7.35	7.35

Table 4.2: Bits per bytes for three selected compression implementations.

length. There is the encoded block (`uint8_t block[BLOCK_SIZE]`) that is from the input stream. There is a block that will contain the decoded block `utin8_t unrotated[BLOCK_SIZE]` and a third array that contain the predicates `uint16_t predicates[BLOCK_SIZE]`. There are also an array of 256 elements `uint16_t count[UCHAR_MAX+1]`, which keep track of which bytes in the encoded block that has been decoded. The approximate size of the BWT decompression programs runtime data can be calculated with the following equation:

```
2 * BLOCK_SIZE * sizeof(uint8_t) + BLOCK_SIZE * sizeof(uint16_t)
+ 256 * sizeof(uint8_t)
```

Figure 4.2 shows a part of code from the BWT decompresser with the arrays that are allocated on the stack.

Bytes reduced			
Test	LZW 10 F	BWT 1000	BWT 2000
1	-219	+437	+557
2	+479	+611	+605
3	+983	+771	+777
4	-262	-393	-436
5	-124	-358	-330
6	+390	-81	-55
7	+212	+194	+192
8	-4690	-5604	-5370
9	-6356	-7368	-8663
10	-53935	-58308	-59263
11	-6861	-5929	-5973
12	-637	-348	-860
13	-15378	-16634	-17200
14	+422	+618	607
15	+869	+746	746

Table 4.3: Bytes reduced (including decompresser). See section 4.1

4.3 Speed of decompression

The tables 4.6 and 4.7 show cycle count for 3 different applications from the EEMBC-suite. The cycle count results come from the IAR Systems CSPY debugger. The debugger does not give a very accurate cycle count since it is only a simulation. The results can however give us a picture of the differences between the different algorithms. The table shows that the differences in cycle count between LZW and BWT are small.

```
#define CHAR_BIT        8
#define CODE_LEN        10
#define FIRST_CODE      (1 << CHAR_BIT)
#define MAX_CODES       (1 << CODE_LEN)
#define DICT_SIZE       (MAX_CODES - FIRST_CODE)

typedef struct
{
  uint8_t suffixChar;
  uint16_t short prefixCode;
} dictionary_t;

void __iar_lzw_init(struct __iar_init_data data)
{
  dictionary_t dictionary[DICT_SIZE];
...Hidden code...
}
```

Figure 4.1: Memory consuming structures in LZW implementation.

```
#define BLOCK_SIZE  2000

void __iar_bwt_init(struct __iar_init_data data)
{
  uint16_t count[UCHAR_MAX + 1], predicates[BLOCK_SIZE];
  uint8_t unrotated[BLOCK_SIZE];
  uint8_t block[BLOCK_SIZE];
  ...Hidden code...
}
```

Figure 4.2: Memory consuming structures in BWT implementation.

Size of decompresser (bytes) *With optimizations for speed*						
Mem.	**None**	**LZW 9**	**LZW 10**	**LZW 12**	**LZW 9F**	**LZW 10F**
Code	0	684	684	684	792	812
Data	0	4	4	4	4	4

Size of arrays on the stack (bytes)						
Mem.	**None**	**LZW 9**	**LZW 10**	**LZW 12**	**LZW 9F**	**LZW 10F**
Size	0	1536	3072	12288	1536	3072

Table 4.4: Size of the LZW decompressors.

Size of decompresser (bytes) *With optimizations for speed*					
Mem.	**None**	**RLE**	**BWT 500**	**BWT 1000**	**BWT 2000**
Code	0	96	756	784	784
Data	0	0	20	20	20

Size of arrays on the stack (bytes)					
Mem.	**None**	**RLE**	**BWT 500**	**BWT 1000**	**BWT 2000**
Size	0	0	2512	4512	8512

Table 4.5: Size of the RLE and BWT decompressors.

Cycle count (million cycles)					
Test	**LZW 9**	**LZW 10**	**LZW 12**	**LZW 9F**	**LZW 10F**
8	1.0	1.0	1.0	1.3	1.3
9	1.2	1.2	1.2	1.3	1.4
6	0.2	0.2	0.3	0.3	0.3

Table 4.6: Cyclecount on three applications.

Cycle count (million cycles)				
Test	**RLE**	**BWT 500**	**BWT 1000**	**BWT 2000**
8	0.2	1.2	1.3	1.3
9	0.2	1.4	1.5	1.5
6	0.2	0.3	0.4	0.4

Table 4.7: Cycle count on three applications.

Chapter 5

Related Work

There have been a great quantity of research in the area of compression since the 1970's [1] and there are a lot of literature available. However we have only found a limited amount of previous work on compression in linkers for embedded systems. There are a few of other techniques that are used in compilers and linkers to reduce an applications memory usage. Examples of such techniques are dead-code elimination and in lining of functions. ARM's RealView Software development tools [13] is one of the applications that has data compression built into its linker.

5.1 Linker data compression

The RealView linker performs compression on Read-Write data in a way that is similar to our implementation. According to the documentation [14] the ARM linker software has two types of compression algorithms implemented; Run-Length Encoding and LZ77 [3] compression. Their RLE implementation seem similar to our algorithm. Though there are some differences between the LZ77 and LZW algorithms. While LZW creates a separate dictionary the LZ77 are using the previous read string as a dictionary. This type of dictionary is called a "sliding window". The LZW algorithm will flush its dictionary when it is full. The LZ77 will always have the last `n` characters available in the "sliding window" dictionary. The LZ77 dictionary will store less symbols on the same amount of memory compared to LZW.

5.2 Code compression

Code compression is a type of compression that is used in some embedded systems [15]. As apposed to the arbitrary type of data that the linker handle, code has some useful patterns. These patterns can be made use of by Split-stream compression [16]. Split stream compression is a method of compressing instructions by splitting the fields of the instructions into different streams. Each stream can then be compressed by for example LZW and have its own dictionary.

Chapter 6

Discussion and Conclusion

The goal of this thesis is to find out which compression algorithms that are suitable for implementation in the Ilink linker. We wanted to know how large the implementation of the decompression algorithms would be and how much application data that is needed to justify compression.

6.1 Summary

We have implemented compression based on three different compression algorithms:

- **Run-Lenght-Encoding** is a simple compression algorithm that uses very little resources. This algorithm is not very good at compression because it only compresses repeated characters.
- **Lempel-Ziv-Welch** is a dictionary based compression algorithm. It proved to be very effective on some of the test cases. It is probably the most famous of the three algorithms and has a lot i commercial implementations.
- **Burrows-Wheeler Transform** is a compression algorithm that we combined with the RLE algorithm. The compression performance was very good and the decompresser is very fast.

Before implementation the LZW algorithm was the predicted winner among the three algorithms. It has been tested in similar environments before and has proved to be effective. The BWT algorithm is usually combined with

Move-To-Front encoding and Huffman or Arithmetic coding. In our implementation we combined it with RLE because of its small code size. Before implementation we did not know how effective it would at compressing data. It proved to be very good, and the decompression algorithm has a linear time complexity.

6.2 Conclusions

In the cases where there are large amounts of data to compress it is a competition between the LZW and BWT algorithms. None of the compared algorithms requires relatively large proportions of memory or is too slow. It is not easy to choose one of them. The compression algorithms performance will depend very much on the type of data that is compressed. Some algorithms perform very well on a type of data where other algorithms do little compression.

The BWT configuration using blocks of 2000 bytes (BWT 2K in figure 4.1) was the implementation that was best at compressing data in the majority of the test cases. This was the biggest block size we tested and it requires about 9000 bytes of RAM. The paper written by Burrows and Wheeler [2] states that the BWT algorithms performance increases as the block size gets bigger. This was true for most of the tested applications. There are a couple cases where the LZW compression performs better than BWT and the difference are big. This means that the LZW is not completely dominated by BWT. The decompresser in the LZW implementation is a bit slower than BWT as it searches the dictionary for each symbol that is read from the input.

The future implementation of compression in the ilink linker will most likely contain more than one algorithm and more than one configuration of the size of dictionaries or blocks. That means that we can choose two or three of the implemented algorithm configurations. This are the most interesting configurations we would consider:

- **LZW 10 bits with Flushing enabled.** The decompresser algorithm requires only about 800 bytes on ROM and 3000 bytes of RAM memory. It is also effective in compressing data if considering its RAM usage. This algorithm may be interesting using other sizes of the dictionary to. With big chunks of data it can be interesting to try dictionaries of

larger size. Compression performance may increase as the dictionary grows until about 4096 entries (i.e. 12 bit symbols).

- **BWT with a block size of 2000 bytes.** BWT with a buffer of 2000 bytes proved to be the best compressor i many cases. The decompresser is also very fast. The implementation of the decompresser requires about 800 bytes if ROM and 8500 bytes of RAM. This algorithm should really use as big blocks as possible, which means that an implementation where blocks of double or triple the size of this one may be interesting.

- **RLE** Even the stand alone RLE compressor might be favorable in some specific cases where there are very little data available. The decompresser requires only about 80 bytes and almost no RAM apart from the decompressed bytes.

Bibliography

[1] *Introduction to Data Compression*
Khalid Sayood
2006. 3rd Edition, ISBN-13: 978-0-12-620862-7, ISBN-10: 0-12-620862-X

[2] *A Block-sorting Lossless Data Compression Algorithm*
Michael Burrows, David J. Wheeler
May 10, 1994. Systems Research Center, Palo Alto, California, USA

[3] *A Universal Algorithm for Sequential Data Compression*
Jacob Ziv, Abraham Lempel
May, 1977. IEEE Transactions on Information Theory, VOL. IT-23, NO. 3

[4] *Compression of Individual Sequences via Variable-Rate Coding*
Jacob Ziv, Abraham Lempel
September, 1978. IEEE Transactions on Information Theory, VOL. IT-24, NO. 5

[5] *A Technique for High-Performance Data Compression*
Terry A. Welch
1984. Sperry Research Center, Sudbury, Massachusetts, USA

[6] *Patent Information for LZW-based Technologies*
December, 2007. http://www.unisys.com/about__unisys/lzw

[7] *Data compression using adaptive coding and partial string matching*
J.G. Clearly and I.H. Witten.
1984. IEEE Transactions on Communications, 32(4):396-402

[8] *EEMBC Embedded Microprocessor Benchmark Consortium*
December 2007. http://www.eembc.org

[9] *PPM Performance with BWT Complexity: A Fast and Effective Data Compression Algorithm*
Michelle Effros
August 8, 2000. Publisher Item Identifier S 0018-9219(00)09984-9

[10] *LZW Data Compression. Dr. Dobb's Journal*
M. Nelson.
Oct. 1989.
(Copy of article: http://dogma.net/markn/articles/lzw/lzw.htm)

[11] *Lexical permutation sorting algorithm*
Z. Arnavut, S. S. Magliveras
1997.
Comput. J., vol. 40, no. 5, pp. 292-295

[12] *Programming Ruby - The Pragmatic Programmers' Guide* Dave Thomas, Chad Fowler, Andy Hunt
May, 2006. Version 2006-5-4, ISBN: 0-9745140-5-5

[13] *ARM Documentation*
http://infocenter.arm.com/help/index.jsp
March, 2008.

[14] *RealView Compilation Tools*
Version 3.1, Linker and Utilities Guide
http://infocenter.arm.com/help/index.jsp
March, 2008.

[15] *Survey of Code-Size Reduction Methods*
Arpad Beszedes, Rudolf Ferenc, Tibor Gymothy, Andre Dolenc, Konsta Karsisto
ACM Computing Surveys, Vol. 35, No. 3
September 2003

[16] *Split-Stream Dictionary Program Compression*
Steven Lucco
Transmeta, 3940 Freedom Circle, Santa Clara, CA 95054
2000

Appendix A

Glossary

ARM Architecture

Advanced RISC Machine. ARM is a 32-bit RISC (Reduced Instruction Set Computer) developed by ARM Limited. The architecture has power saving features which makes it interesting for the mobile electronics market.

Byte Values

The value of a byte are expressed in hexadecimal form in the given examples. Consider this example of four bytes with the values 1, 2, 169 and 170:

`0x01 0x02 0xA9 0xAA`.

Data Segment

The data segment is a section in memory or in an object file which contains data initialized by the programmer. The data holds global variables and has a fixed size.

EEMBC Benchmarks

The EEMBC tests are real world benchmarks from the Embedded Microprocessor Benchmark Consortium [8]. The EEMBC benchmarks collection are a kind of industry standard to test performance of hardware, compilers

and Java implementations. The collection consists of a couple of applications that reflect real world applications of different types.

IAR Systems

Founded in 1983 is a leading provider of embedded development tools. IAR Systems products include: IAR Embedded Workbench, visualSTATE, IAR KickStart Kit, IAR Advanced Development Kit, IAR PowerPac.

IAR Systems Embedded Workbench

IAR Embedded Workbench is a set of development tools for building and debugging embedded applications using assembler, C and C++.

Compiler

A program that translates text written in a programming language into a representation more understandable by a computer. The output often has a form suitable for processing by other programs (e.g. a linker).

Linker

The linker is a software that takes one or more objects generated by compilers and assembles them into a single executable program. The objects consist of machine code and information for the linker. The information in the objects show symbol definitions, such as functions and variables. A part of he linkers job is to sort out references to symbols between the objects. The references are replaced by an address to the symbol. Another job for the linker is to arrange symbols in the programs address space. When references are solved and address space is arranged the linker will output a executable output.

Ruby

Ruby [12] is a object-oriented programming language. It has a syntax inspired by Perl and has object-oriented features like the Smalltalk. Ruby is a single-pass interpreted language.