Master's Thesis

Real-Time Lossless Compression of SoC Trace Data

Jing Zhang

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, December 2015.

Master's Thesis

Real-Time Lossless Compression of SoC Trace Data

Jing Zhang

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

December 2015

Abstract

Nowadays, with the increasing complexity of System-on-Chip (SoC), traditional debugging approaches are not enough in multi-core architecture systems. Hardware tracing becomes necessary for performance analysis in these systems.

The problem is that the size of collected trace data through hardware-based tracing techniques is usually extremely large due to the increasing complexity of System-on-Chips. Hence on-chip trace compression performed in hardware is needed to reduce the amount of transferred or stored data.

In this dissertation, the feasibility of different types of lossless data compression algorithms in hardware implementation are investigated and examined. A lossless data compression algorithm LZ77 is selected, analyzed, and optimized to Nexus traces data. In order to meet the hardware cost and compression performances requirements for the real-time compression, an optimized LZ77 compression algorithm is proposed based on the characteristics of Nexus trace data.

This thesis presents a hardware implementation of LZ77 encoder described in Very High Speed Integrated Circuit Hardware Description Language (VHDL). Test results demonstrate that the compression speed can achieve16 bits/clock cycle and the average compression ratio is 1.35 for the minimal hardware cost case, which is a suitable trade-off between the hardware cost and the compression performances effectively.

Acknowledgments

If we meet someone who owes us thanks,
we right away remember that.
But how often do we meet someone to whom
we owe thanks without remembering that?
— JOHANN WOLFGANG VON GOETHE
Ottilie's Diary (1809)

The completion of this thesis gets a lot of help from my supervisor and colleagues in Ericsson, my supervisors in Lund University, and my lovely friends. Here, I want to thank all of them.

Firstly, I would like to express my deepest gratitude to my supervisor Lars Johan Fritz, who encouraged and motivated me all the time. I am grateful for his support, guidance, and constructive advices on my work. In addition, I also thank Pierre Rohdin G and Mats Fredriksson for their help and encouragement in this period.

Secondly, I am grateful to the supervisor Erik Larsson and Liang Liu for their trust and help on my thesis. I should to thank for their suggesting many ways in how to improve my thesis work. The same acknowledgement should also go to Professor Peter Nilsson who acts as my examiner.

Thirdly, I would like to thank my friends, Zhou Ruoxing and Liu Dan. Their friendly advices and supportive attitude are of great help for me. In particular, I am thankful to Yuan Mengze for his patiently instructions.

Lastly but not least, a special thank goes to my family, this thesis might not exist without their support and understanding.

Many more peoples participated in various ways to ensure my research succeeded and I am thankful to them all.

Jing Zhang

Contents

CHAPTER 1

1. Introduction

Collecting execution traces of programs is an important task for embedded system verification and debugging. With the increasing complexity of System-on-Chip, the size of collected trace data through hardware-based tracing techniques is usually extremely large. For example, monitoring a 32 bits bus in an RISC microprocessor clocked at 1 GHz generates a trace of 4G bytes per second. Storing such huge data or transferring the data out of the chip in real time brings a high hardware cost [1]. Therefore, the key problem with hardware-based tracing techniques is to reduce the size of collected trace data. Trace data size reduction techniques include three types – filtering, sampling, and compression. Filtering discards all references. Sampling stores relatively short references at regular intervals, discarding the intervening references. Only trace compression technique can retain all of the trace data information [2]. Compression techniques are divided into lossy compression technique and lossless compression technique. This master thesis focuses on implementing the lossless compression optimized based on the trace data from Ericsson's ASIC platform.

1.1. Nexus Trace Data

In this thesis, the trace data adopts the Nexus IEEE-ISTO5001-2012 standard. Nexus 5001 is a standard for global embedded processor debug interface, which is published by IEEE´s Industry Standard and Technology Organization.

The Nexus standard defines trace and debug interface, including associated protocols and infrastructure that can serve tracing and controlling of multiple cores on a chip from the software debugger [3].

The original standard is the 1999 version, which was aimed to define a general-purpose specification that addressed the rigorous challenges for debug interfaces, and the need for efficient use of embedded processors that requires software and hardware development tools to access critical processor functionality [4]. Trace data can be read out through a JTAG port.

As a modern System-on-Chip has entered into multicore era, a large number of logics have been integrated on the embedded processor.

Increased hardware complexity enables growing software complexity. The traditional debugging and testing approach through a JTAG port is obtrusive and time-consuming, which is not suitable for the advanced real-time embedded system.

The newest standard is the 2012 version, which support a minimum pin interface (IEEE 1149.7) and a high speed serial protocol (Aurora), also continuing to support existing transport mechanisms: parallel (AUX) and IEEE 1149.1. Using an AUX port can achieve faster response when the systems are configured or used.

The Nexus data employs a packet-based messaging tracing scheme with packet headers providing information about data source, destination, and type of payload [3]. Nexus standard defines 4 classes' development features for vendors to use according to their needs. The higher the levels are adopted, the more debug information can be obtained.

- Class1: Basic Run Control. Class1 standard implement the fewest Nexus development features for run-control debugging, including single stepping, breakpoints, watchpoints, and access to registers and memory while the processor is halted.
- Class2: Instruction Trace. Besides class1 features, class2 standard add debug support for capturing program execution traces, watchpoint traces, and ownership traces in real time. Ownership traces are useful to correlate simultaneously executed threads in time [3].
- Class3: Data Trace. Besides class2 features, class3 standard add debug support for data traces, memory, and I/O read/write traces. This mainly related to implementation of full tracing capabilities.
- Class4: Remote processor control and advanced trace. In addition to class3, class4 standard add memory substitution traces. Memory substitution allows the processor to execute instructions from the trace port rather than from the memory [3]. Memory substitution is implemented using address remapping for the I/O space (where the trace port is located) and for the memory (where the original executing program resides) [3].

According to the Nexus Trace Data Standard, the trace messages can be classified into three categories depending on the type of information they contain: program (or instruction) traces, data traces (from the memory bus), and system traces (various signals of interest in debugging the implemented hardware or observing the inter-cores dependencies and so on) [3]. The trace data messages mainly include 5 types as below.

- Status message: This type of message indicates the status information from the target whenever there is a state change. Such as, Debug Status Message.
- General register read/write message: This type of message is used for run control and configuration of watchpoint/breakpoint operations [3]. Such as, breakpoint/watchpoint messages.
- Program trace message: This type of message is commonly used in multi-core systems. It has capability to detect and signal program trace errors.
- Data trace message: This type of message is used for tracing data addresses and values. The Data Trace feature defines a standard protocol for data trace visibility of accesses to vendor-defined internal peripheral and memory locations.
- Memory Access message: Non-intrusive access to internal memory blocks [3].

The Nexus Trace Messages can also be divided into two types, both Public Messages and Vendor-defined Messages. The format and meaning of Public Messages are defined by the specification of Nexus Standard. The Vendor-defined Messages are defined by the target processor vendor. In this thesis, Ericsson uses the class 4 of Nexus standard. It includes both Nexus public messages (like, data trace message, program trace message, and ownership trace message) and Ericsson's own defined trace messages.

1.2. Thesis Motivations

Collecting execution traces of system programs become an important task for debugging and testing embedded systems. The traditional method of collecting trace data is intrusive and time-consuming, which cannot meet the usage requirements of real-time embedded systems. Especially, in multi-core systems, the traditional method can cause timing requirements violations.

Recognizing these issues, many vendors have developed modules with tracing capabilities and integrated them into their embedded platforms in order to improve debugging and testing efficiency, e.g., ARM's Embedded Trace Macrocell, MIPS's PDTrace [3].

In order to improve the debugging capability of the debug software, a dynamic real-time trace debug block has been implemented in Ericsson's ASIC platform also. The debug block provides a way to enable both traditional static target debugging but also enable non-intrusive real-time

target debugging while the system is up and running. The debug block can collect all different kinds of trace messages, which are sent out from the DSPs in Ericsson's platform. With these messages the software tools can recreate the complete program-flow, a better observability and controllability are achieved in the chip through using the debug block [5].

However, the main problem with hardware tracing is the extremely large trace data that need to be collected. For example, a processor running at 1GHz produces gigabytes trace information for just one second of execution time [3]. Storing and analyzing such huge amounts of data results in expensive hardware and requires a very large on-chip buffer. Furthermore, transferring the trace data in real-time requires wide trace ports. To meet such requirements would significantly increase the system complexity and cost. Therefore, reducing the trace data becomes a necessary and high valuable technique.

The target of this thesis is to design a hardware compressor, to reduce the requirements of large on-chip trace buffers and wider trace ports for the embedded system with high complexity such as Ericsson's ASIC platform.

1.3. Thesis Objectives

The main objective of this dissertation is to design a real-time lossless compressor optimized to the trace messages. In the Ericsson platform, the trace data messages are sent from several DSPs and received by the debug block.

The specific requirements of this compressor proposed by Ericsson are as following.

- Compression ratio: 2 to 5. ($CR=\frac{original\ data\ size}{compressed\ data\ size}$)
- Compression speed: 16bits/clock to 64bits/clock.
- Lossless Compression
- Real-time Compression

To achieve this objective, a real-time trace data compressor that can get a balance with acceptable compression ratio and low hardware cost is designed.

1.4. Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents the background for the compression algorithms. Chapter 3 presents the work

principle and selection reason of the selected compression algorithm, and the results of parameter simulation. Chapter 4 presents the architecture of the system design and explaining internal modules in detail. In addition, the test results are shown in this Chapter. Chapter 5 summarizes this thesis work and presents the potential directions of the future work of this thesis.

CHAPTER 2

2. Background of Compression Algorithms

For proposing an efficient compression algorithm optimized to the Nexus trace data, this chapter outlines the background study based on the compression algorithms.

Data compression is a process of encoding data with fewer bits through removing the redundancies found in the input data stream. The more redundancies are found, the better compression ratio can be achieved. Regarding the Nexus trace data messages, there are two types of compression methods to extract the redundancies. One is general-purpose compression methods, and the other one is special-purpose compression methods.

In Section 2.1, the general-purpose compression algorithms´ principles and operations are described. The specialized techniques and approaches of special-purpose compression are discussed in Section 2.2.

2.1. General-Purpose Compression Algorithms

General-purpose compression algorithms are generally classified into two categories, lossless compression algorithms and lossy compression algorithms. Lossy compression algorithms can achieve much better compression ratio with the sacrifice of losing part of original information. However, lossless compression algorithms can recover all the information compared to lossy compression algorithms.

2.1.1. Lossy Compression

In lossy compression, only approximation information can be recovered when the compressed data is decompressed, which means the retrieved data is not identical to the original data. Hence the results of lossy compression can be used only in some special applications.

Lossy compression is usually applied to the transmission and storage of images, audio, and video where some finer details of that information can be thrown away during the compression process and the introduced distortion in original information can be acceptable when uncompressed such data. In these applications, the difference between the recovered data and the original data can be omitted to the human ears or eyes considering the

idiosyncrasies of human anatomy. For example, the human eyes and ears can see and hear only certain frequencies of lights and sounds.

In contrast to lossless compression algorithms, the advantage of lossy compression algorithms is a significant improvement to the compression ratio with accepted distortions which can meet the requirements of the applications. Some lossy methods and their applications are summarized in the table 2.1 [6].

TABLE 2.1. LOSSY COMPRESSION METHODS.

Application	**Image**	**Audio**	**Video**
Compression methods	Fractal compression	AAC	H.261
	JPEG	ADPCM	H.263
	Dolby	ATRAC	MNG
	Wavelet compression	MP2	MPEG-1
		MP3	MPEG-2
		HILN	MPEG-4
		WMA	Motion JPEG

2.1.2. Lossless Compression

Lossless compression allows the original data to be exactly recovered from their compressed form. Lossless compressions are usually used in the applications where losing a single bit cannot be accepted, such as text files, archival storage database, which mostly contain vital information, and some special classes of images like medical imaging, fingerprint data, and astronomical images. Considering the importance of trace data integrity, this thesis will only focus on the lossless compression algorithms

The lossless compression can be broadly classified into three types – dictionary based compression, statistical compression, and combinational compression. Combinational compression algorithms are commonly the combination of dictionary based compression algorithms and statistical compression algorithms. Usually a combinational compressor can achieve better compression ratio than both dictionary-based compressor and statistical compressor. However, the calculation complexity of combinational compression algorithms is higher.

2.1.2.1 History of Lossless Compression

With the widely use of computer science and Internet technology, data compression has started to play a significant role since 1970s. The earliest compression technique is Morse code, which was invented in 1838. Modern work on data compression began in the late 1940s with the development of information theory [7].

In 1949, Claude Shannon and Robert Fano invented Shannon-Fano coding. The algorithm of Shannon-Fano coding assigns shorter codes to symbols, which have more occurring probability. In 1951, David Huffman invented Huffman coding, which is a more efficient coding method than Shannon-Fano coding. In statistical algorithms, Huffman coding is widely used considering the computation complexity of arithmetic coding algorithms.

In 1977, the first dictionary based compression algorithm LZ77 was published by Abraham Lempel and Jacob Ziv [8]. Typically, LZ77 uses a sliding window to compress data. In 1978, another dictionary based algorithm LZ78 was published by Abraham Lempel and Jacob Ziv [9]. Unlike LZ77, LZ78 uses whole input symbols as the dictionary to parse the input data.

Both the LZ77 and LZ78 algorithms grew rapidly in popularity and have lots of variation algorithms. Most of the commonly used algorithms, such as, LZSS, LZO, and LZ4, shown in the Fig 2.1, are derived from the LZ77 algorithm and LZ78 algorithm [10]. Among them, Lempel-Ziv-Welch algorithm, which was created in 1984 by Terry Welch, is the most used compression algorithm in LZ78 family. LZSS algorithm is the representative algorithm in LZ77 family.

In combinational algorithms family, Bzip2, Deflate, and LZMA are the most popular used algorithms in data compression applications. In 1993, Phil Katz invented Deflate algorithm, which belongs to a combinational compression algorithm, is the combination of LZ77 algorithm and Huffman algorithm [11]. Bzip2 is developed in 1996 and maintained by Julian Seward [11].

The whole classification of lossless algorithms is shown in Fig. 2.1.

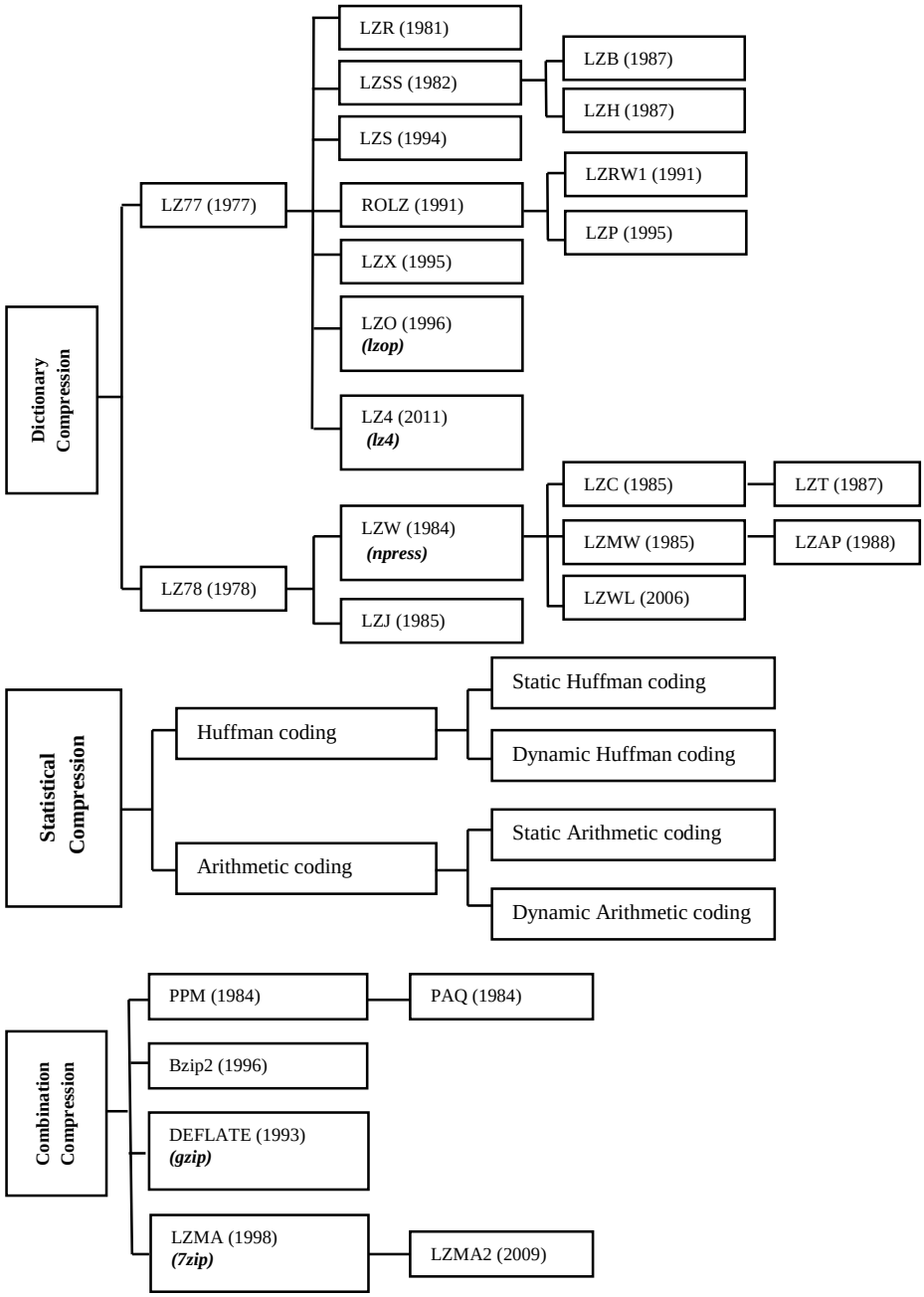


Fig. 2.1. Data Compression Classification Chart.

2.1.2.2 Dictionary Based Compression

The main idea of dictionary based algorithms is to replace repeated symbols with shorter codes, thus it is also called as substitution codes. In these methods, the dictionary, which is built during the compression process, contains a set of input strings appeared in the past. The principle of dictionary based algorithm is to find the match in the strings contained in the dictionary and to replace the match by shorter code words.

In these techniques, no prior knowledge or statistical characteristics of the data being compressed are required. The compression speed is faster compared to statistical methods since several symbols can be encoded at a time.

The dictionary based techniques can be static, dynamic or adaptive. The approach using the static dictionary is most often used when input strings are determined before coding begins and do not change in the whole compression process. As the most popular scheme of dictionary based algorithm, the dictionaries of Lempel and Ziv algorithms are adaptive or dynamic. LZ77 and LZ78 are the two most basic Lempel and Ziv algorithms. All others are the variants of LZ77 or LZ78, such as LZSS, LZW, LZR, LZT and LZJ etc. The difference between LZ77 and LZ78 is the method of building the dictionary. LZ77 uses part of the input symbols to set up the dictionary, however, LZ78 uses whole input symbols to set up the dictionary. In this part, LZ77 and LZ78 will be discussed.

2.1.2.2.1 LZ77 Algorithm

LZ77 algorithm is based on a sliding window. The sliding window is divided into two parts. The left part is called as *search buffer* or *dictionary*, which includes the symbols that have been input and encoded recently. The right part is called as *look-ahead buffer*, which includes the symbols needed to be encoded. The length of the search buffer is equal to the dictionary size. The length of the look-ahead buffer is equal to the value of the maximum length of the identical symbols. During the compression, the dictionary will be changed dynamically with the movement of sliding window.

The compressed result is represented as *(distance, length, next symbol)*. *Distance* indicates the offset from the start of the match symbols found in the sliding window to the current symbol. *Length* indicates the length of the match symbols. *Next symbol* indicates that a new phrase was found [10].

Take an example, in Fig. 2.2, we assume that the new input text is “sir sid eastman”. The encoder scans the search buffer from right to left to find a match for the first symbol “s” in the look-ahead buffer. The search buffer is

empty at the beginning and there is no symbol stored in the search buffer, thus no match can be found in the search buffer. The output is (*0, 0, s*). The sliding window is shifted one position to the right. When the second symbols "si" are the input, the same symbols can be found in the dictionary, and the output is (*4, 2, d*). The window is then shifted to the right two positions. The encoder continues encoding the symbol until all the symbols are shifted from the look-ahead buffer to the search buffer.

Search Buffer	Look-ahead Buffer	Output
	sir_ sid_eastman_	(0,0,s)
s	irk_ sid_eastman_	(0,0,i)
si	r_ sid_eastman_	(0,0,r)
sir	_ sid_eastman_	(0,0,_)
sir_	sid_eastman_	(4,2,d)

Fig. 2.2. Example of LZ77 Compression Algorithm.

The pseudo code of LZ77 compression algorithms is shown in Fig. 2.3 [12].

LZ77 Compression Algorithm

```
1: while look-ahead buffer is not empty do
2:    go backwards in search buffer to find longest match of the look-ahead buffer
3:    if match found  then
4:      print: (offset from window boundary ;)
5:      print: (length of match;)
6:      print: (next symbol in look ahead buffer ;)
7:      shift window by length + 1;
8:    else
9:      print: (0, 0, first symbol in look-ahead buffer);
10:     shift window by 1;
11:   end if
12: end while
```

Fig. 2.3. Pseudo of LZ77 Compression Algorithm.

2.1.2.2.2 LZ78 Algorithm

Instead of using a sliding window as used by LZ77, the dictionary of LZ78 contains all the previous symbols, which have been compressed. The outputs are two-field tokens (*index, next symbol*). Every input symbol is added into the dictionary after it has been compressed. Nothing is ever deleted from the dictionary during the whole encoded process. The dictionary starts with the null string at position zero. As symbols are input

and encoded, strings are added to the dictionary at positions 1, 2, and so on [11].

Take an example, in Fig. 2.4, we assume that the new input text is “sir sid eastman”. The encoder searches the dictionary for an entry with the first input symbol “s”. There is no matched entry, the output is (*0, s*). The symbol “s” is added to the dictionary with the entry number 1. When the second symbols “si” is input, the dictionary is searched for an entry containing the two-symbol string “si”. If no match can be found, the string “si” is added to the next available position in the dictionary with entry number 5. The token (*1, i*) is output. The process continues until the end of the input stream is reached [11].

Dictionary Index	Dictionary Content	Input	Output
0	Null		
1	S	sir_ sid_ eastman	(0,s)
2	I	ir_sid_ eastman	(0,i)
3	R	r_ sid_ eastman	(0,r)
4	_	_ sid_ eastman	(0,_)
5	Si	sid_ eastman	(1,i)
6	D	d_ eastman	(0,d)
7	_e	_ eastman	(4,e)

Fig. 2.4. Example of LZ78 Compression Algorithm.

The pseudo code of LZ78 compression algorithms is shown in Fig. 2.5.

LZ78 Compression Algorithm

```
1: while input stream is not end do
2:    go backwards in dictionary to find the index with match of the input symbol
3:    if match found  then
4:      print: (index)
5:      print: (next symbol in input stream ;)
7:      read the next symbol;
8:    else
9:      print: (0, symbol);
10:     read the next symbol;
11:   end if
12: end while
```

Fig. 2.5. Pseudo of LZ78 Compression Algorithm.

2.1.2.3 Statistical Compression

The main idea of statistical algorithms is to assign values to events depending on their occurring probability. Specifically, the data with higher occurring probability is represented by shorter code words.

Generally in these techniques, either Huffman coding or Arithmetic coding needs prior knowledge or statistical characteristics of the data being compressed. A pre-scanning of the real data before coding is needed in order to get a code word table, which contains the mapping information between the real data and the code words for coding.

There are two ways to eliminate the pre-scan procedure to adapt the requirements of real-time applications. The first method, Static Huffman coding or Static Arithmetic coding, is to use a known or default code word table for encoding [13]. The second method, Adaptive Huffman coding or Adaptive Arithmetic coding, is to use an encoding tree, which is adaptively constructed and maintained at sender as well as receiver side [13].

The most famous statistical algorithms are Huffman coding and Arithmetic coding. In this part, both Huffman coding and Arithmetic coding will be discussed.

2.1.2.3.1 Huffman Coding

In order to compress a string of symbols, Huffman coding is based on representing the symbols that have high occurrence probabilities with shorter code words and assigning longer code words to symbols that have low occurrence probabilities.

Huffman coding technique can be static or dynamic (adaptive). Static Huffman coding uses a look-up table that stores the pre-defined frequency for each symbol. Dynamic Huffman coding calculates the frequency of every symbol according to the real occurring frequency.

Huffman Coding algorithm basically builds a binary tree. The leaves represent the symbols of the input file. The code length for these symbols equals their depth in the tree (that is, their distance to the root node) [12]. Once the symbol frequency has been determined, the two elements with the lowest frequency are selected and inserted as leaves of a node with two branches. The sum of these two elements' frequency becomes the frequency for a new node. The algorithm selects another two new elements with the lowest frequency in the left elements and inserts them in the tree. A Huffman tree is completed until the root node having a 100% frequency.

The procedure of the Huffman tree generation is shown in Fig. 2.6 [10].

Huffman Coding Algorithm

1: Parse the input, counting the occurrence of each symbol.
2: Determine the probability of each symbol using the symbol count.
3: Sort the symbols by probability, with the most probable first.
4: Generate leaf nodes for each symbol, including P, and add them to a queue.
5: While (Nodes in Queue > 1)
    Remove the two lowest probability nodes from the queue.
    Prepend 0 and 1 to the left and right nodes' codes, respectively.
    Create a new node with value equal to the sum of the nodes' probability.
    Assign the first node to the left branch and the second node to the right branch.
    Add the node to the queue
6: The last node remaining in the queue is the root of the Huffman tree.

Fig. 2.6. Huffman Coding Algorithm.

2.1.2.3.2 Arithmetic Coding

Arithmetic coding was developed by IBM Company in 1979. Arithmetic coding is one of the most optimal entropy coding techniques if the objective is the best compression ratio [10]. However, its complexity is the most complicated.

Arithmetic coding can achieve better compression ratio compared with Huffman coding. It is because Arithmetic coding uses fractional bits for its code words, while Huffman coding uses an integral number of bits. Thus the efficiency of Arithmetic coding can be made arbitrarily close to the entropy or information content by controlling its precision [14].

Arithmetic coding transforms the input data into a single rational number between 0 and 1[10]. For each symbol, the current interval is divided into subintervals. The length of subinterval is proportional to the occurring frequencies of the symbol. Then the subinterval of the current symbol is chosen again. This procedure is repeated for all symbols from the input file. At the end, the compressed result, which is a fixed-point binary number, is the output from the final interval.The procedure of Arithmetic coding is shown in Fig. 2.7 [10].

Arithmetic Coding Algorithm

1: Calculate the number of unique symbols in the input. This number represents the base b (e.g. base 2 is binary) of the arithmetic code.

2: Assign values from 0 to b to each unique symbol in the order they appear.

3: Using the values from step 2, replace the symbols in the input with their codes.

4: Convert the result from step 3 from base b to a sufficiently long fixed-point binary number to preserve precision.

5: Record the length of the input string somewhere in the result as it is needed for decoding.

Fig. 2.7. Arithmetic Coding Algorithm

2.1.2.4 Combinational Compression Algorithm

Combinational compression algorithms are normally based on the combination of dictionary based algorithm and statistical algorithm. In combinational compression algorithm, Gzip, Bzip2, and LZMA are three of the most widely used algorithms. Among them, Gzip and LZMA are the combination of statistical algorithm and dictionary based algorithm. Bzip2 is a combination of Burrows-Wheeler Transformation (BWT) and Huffman coding. In Table 2.2, the constitutions of these three algorithms are shown.

TABLE 2.2. CONSTITUTIONS OF COMBINATIONAL ALGORITHM.

Algorithm	Algorithm Constitution
Gzip	• LZ77 • Huffman coding
LZMA	• LZ77 • Arithmetic coding
Bzip2	• Run-length encoding • Burrows-Wheeler transform • Move to front transform • Huffman coding

Normally, a combinational compressor could achieve a better compression ratio than a dictionary based compressor or a statistical compressor. However, the calculation complexity is higher and this conclusion could also apply to Nexus trace data based on the experimental result.

2.2. Special-Purpose Compression Algorithms

Besides general-purpose compression algorithms, there are some researches on specialized compression algorithms focusing on Nexus trace data. These compression mechanisms are typically used to compress program trace message or address trace message. In this part, some specialized algorithms are introduced briefly.

- Instruction compression

For instruction compression, a technique named Packed Differential Instruction (PDI) [2] is proposed. Instruction compression is used to compress the special trace messages, which contain instruction execution information. The PDI technique divides the instructions into frequently used instructions and unfrequently used instructions and uses a type of dictionary based algorithm to compress the frequently used instructions.

- Address compression

For address compression, several techniques exist to exploit the redundancies for address compression. Address compression is used to compress the address trace messages, which include data address and instruction address. Mache [15] replaces a data address with an offset, which is the difference between the last same type address and the current address. Packed Differential Address and Time Stamp (PDATS) have the same function as Mache. PDATS [2] converts the absolute addresses and time stamps into address offsets and time offsets. However PDATS is more complex than Mache, as it introduces variable length encoding of the offset. That means using the minimum number of bytes to encode address and time stamp offsets. Stream-based Compression (SBC) [16] relies on extracting instruction streams. A stream table is needed to keep relevant information about streams. Using indexes in the stream table replaces the whole stream. Value Prediction-based Compression (VPC) uses an address predictor, which is a cache like structure to store recently executed data addresses. The compression is achieved by replacing a data address with an identifier, which points to the cache entry. The cache entry stores the correctly predicted address.

In most applications, in order to yield excellent compression ratio, special-purpose algorithms normally work together with general-purpose algorithms.

CHAPTER 3

3. Compression Algorithm Selection

3.1. Algorithm Selection

Selecting a suitable compression algorithm among so many and different algorithms is not an easy thing. While one algorithm can achieve good compression ratio, the other can be with high throughput and another may require less hardware cost. Hence selecting an algorithm is a trade-off process.

According to the thesis requirements, the three key performances, compression ratio (CR), throughput rate, and hardware cost, need to be considered in high priority when the compression algorithms are selected.

In this thesis, the basic selection rules of the lossless data compression algorithms are as following: Firstly, the compressor must meet the real-time requirement. The operation time of data compression cannot be larger than the input rate of trace data. Secondly, the hardware cost of the data compressor should be low enough and the high compression ratio can be achieved [1]. Thirdly, the compressor should be optimized for the characteristics of the compressed data.

3.1.1. General Algorithm vs. Special Algorithm

General-purpose compression algorithms are chosen in this thesis project. Selecting general-purpose compression algorithms is mainly based on the three below reasons.

First of all, the characteristics of the trace data used in this thesis are considered. In the application of this thesis, Ericsson adopts high level of the Nexus trace class. There are more than 15 types of Nexus trace messages. That means the employed compression algorithm need to perform stably for all kinds of trace messages, not just for one or several types of them, such as program trace message, data trace message, etc. However, the modern special-purpose compression algorithms mainly focus on one specific type of trace messages, such as program trace message or address trace message. Thus choosing special-purpose algorithms and designing different compressor for each different trace message will lead to a high hardware cost and a high compression ratio in return. Considering the high

priority of hardware cost in this thesis, using a general-purpose algorithm is a more reasonable approach.

Secondly, many specialized methods are not suitable for real-time trace compression. Special-purpose techniques could be categorized into two types: software-based and hardware-based. Most software-based algorithms, such as PDAT, Mache, PDI, are not single-pass mechanisms, they need to collect and save the source data firstly and then compress the data. Hence the compression algorithms might not be fast enough to keep up with the trace generation rate in hardware [1]. The hardware-based algorithms are usually complex and with low throughput.

Lastly, in order to achieve a better compression ratio, most of the special-purpose algorithms still need to work with another or several general-purpose algorithms, such as, PDI work together with LZ77 in [2]; value predictor combines with Bzip2 in [18]; SBC need add Gzip in [16]. Such approaches would be cost-prohibitive or infeasible for real-time compression in hardware [19]. Moreover, specialized compression algorithms normally do not support an open license, which is not preferred by this application.

Thus a general-purpose algorithm is a better choice for this application.

3.1.2. Dictionary-Based vs. Statistical vs. Combinational

In this thesis, dictionary-based compression algorithms are selected based on the following four requirements.

The first requirement is the real-time compression. According to the introduction of compression algorithms in section 2.1.2.3, static coding of statistical algorithms use a known or default code word table for encoding. The code word table stores a pre-defined frequency for each symbol of the input file. When the input symbols' distribution fits the expected distribution stored in the static table, the static coding scheme can be effective. However, if the input symbols do not match well with the stored statistics, a poor compression result would be generated. It is possible that output files are larger than input files. In this application, however, the frequency distribution of Nexus trace data changes often and it is impossible to know and store the distribution of every trace message in advance. Therefore, static coding of statistical algorithms cannot be used in real-time compression.

The other three key requirements are compression speed, compression ratio and area overhead. These three requirements are related and affect each other. A traded-off is thus needed. The algorithm with higher

compression ratio normally has higher complexity than the algorithm with lower compression ratio. A complex algorithm means extra hardware cost and slower compression speed. Therefore, the process of selecting a suitable algorithm is a process to trade-off these three requirements.

Considering the requirements of this thesis, the design target is to implement a compressor, which can provide fast compression speed and use less area overhead with an acceptable compression ratio. In order to choose a suitable algorithm from the three types of compression algorithms, some test results of different types of data compressors processing the Nexus trace data are shown in the below.

3.1.2.1 Evaluation Method

- Original file with trace data

All original files include Nexus trace data generated by DSPs in Ericsson hardware platform.

- Timing measurement

All timing measurements refer to the sum of the user and the system time reported by UNIX shell command time. In other words, only the CPU time are reported and ignore any idle time such as waiting for disk operations [18].

- Compression ratio measurement

Compression ratio is a metric to measure the efficiency of compression algorithms. The definition of the compression ratio is shown in equation1. The algorithm with a higher compression ratio means the algorithm has a better compression.

$$\text{Compression Ratio} = \frac{Original\ Trace\ File\ Size}{Compressed\ Trace\ File\ Size} \qquad (1)$$

- Compressor

All these compressors proposed in the following are implemented in software.

Compressor using dictionary based algorithms:

(1) LZ77 algorithm compressor: LZ4.

(2) LZ78 algorithm compressor: LZW.

Compressor using statistical algorithms: Dynamic Huffman coding.

Compressor using combinational algorithms: Bzip2, LZMA.

- Compressor Performance Setting

All compressors use the "-fast" option. In order to obtain the best compression performance, the dictionary size is increased to the maximum size allowed in each compressor.

3.1.2.2 Evaluation Results

Fig. 3.1 shows the average compression ratio of the compressors, when the Nexus trace data is compressed. Fig. 3.2 depicts the average compression speed of compressors processing the Nexus trace data. In Fig. 3.1, the combinational algorithms (Bzip2 and LZMA) deliver the best compression ratio. The difference of compression ratio between dictionary based algorithms (LZ4 and LZW) and statistical algorithms (Huffman coding) is not significant. However, the advantage of using complex algorithms to get an excellent compression ratio results in low compression speeds. In Fig. 3.2, the results show that combinational algorithms (Bzip2 and LZMA) need most compression time. The processing time of statistical algorithms (Huffman coding) is more than those of dictionary based algorithms (LZ4 and LZW).

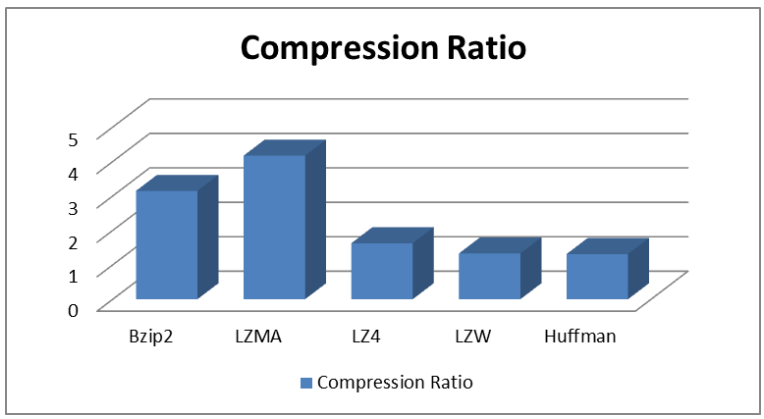


Fig. 3.1. Average compression ratio for Nexus trace data.

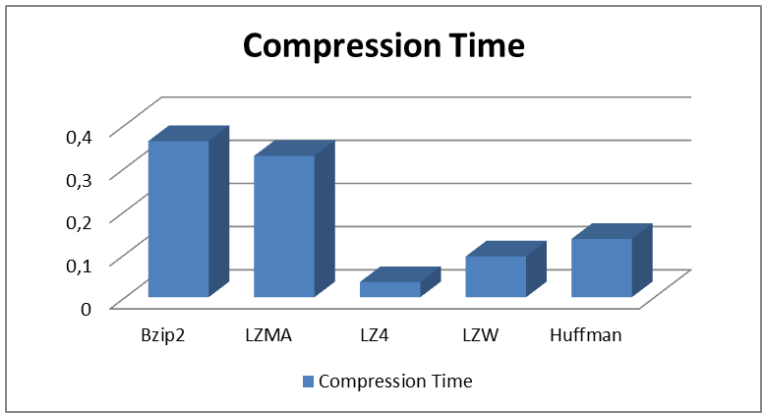


Fig. 3.2. Average compression ratio for Nexus trace data.

Comparing the compression speed of the compressors in term of execution time is not definitive. A comparison about average complexity of the compressors is preferred. In this part, the complexity of LZ4 and the complexity of Dynamic Huffman coding are compared. Since both LZ4 algorithm and Dynamic Huffman coding algorithm have lower complexities compared with other dictionary based and dynamic statistical algorithms.

In order to compress data, dynamic Huffman coding uses a code word table, which is adaptively constructed and maintained. Maintaining and reconstructing the table is the most time consuming process for Dynamic Huffman coding. Dynamic Huffman requires O(n) time step for maintaining the code table, compared with $O(n^2 \log^n)$ steps for reconstructing a new one [20]. Hence, Dynamic Huffman coding has not been available for high-speed applications because the maintaining and reconstructing is time-consuming. In contrast with Dynamic Huffman coding, searching the longest match string is the most time-consuming step of a LZ4 compressor and forms the bottleneck of the performance. The average time complexity of the searching step is O(1) [21]. The complexity of combinational algorithms must be higher than other types of algorithms since the algorithms are a combination of statistical algorithms and dictionary based algorithms. Therefore, it is possible for dictionary-based algorithm to achieve high throughput.

Hardware implementations of the three types of lossless algorithms have been reported in some literatures [14], [20], [22], [23], [24], [25], and [26]. In table 3.1, throughput and hardware cost reported in these applications are listed. For both combinational compressors (Gzip, Bzip2, and LZMA) and statistical compressors (dynamic Huffman coding), it is difficult to achieve a high throughput due to their complex algorithms. The

throughput of Huffman coding is around 0.5 bits/clock cycle, which are quite much lower than the thesis requirement. Moreover, the high complexity of the combinational compressors (Gzip, Bzip2, and LZMA) and statistical compressors (dynamic Huffman coding) is more hardware area consuming. For example, the hardware implementation of LZMA needs a 1.7M bit dual-port RAM and 120k gates. However, the dictionary based compressors can achieve high throughputs with low hardware cost. For example, the throughput of LZW can achieve 14 bits/clock cycle, at the same time, the hardware area is lower compared with the area overhead of combinational and statistical algorithms.

TABLE 3.1. HARDWARE IMPLEMENTATION OF COMPRESSOR.

Algorithm	Processor	Complexity		Throughput (bit/clock)	Reference
		RAM bits	Equivalent Gates		
LZ77	AHA3521 (40MHz)	Not Stated	Not Stated	4	[14]
	Spartan II XC200 Xilinx FPGA	36k	27K	Not Stated	[22]
LZW	Virtex II XC2V250-6fg456 Xilinx FPGA (50MHz)	140k dual-port	24K	14	[23]
Dynamic Huffman	ASIC	4.5K	17.7K	0.5	[20]
Gzip	ASIC (IP core)	1.1M	250K	8	[24]
	ASIC (IP core)	1.1M	610K	8	[24]
Bzip2	Stratix II EP2S180F1020C3	3.3M	162K	1.5	[25]
	Stratix II EP2S180F1020C3	58.4M	1575K	1.5	[25]
LZMA	Virtex 5 XC5VFX70T Xilinx FPGA (100MHz)	1.7M dual-port	120K	max4	[26]

Based on the above analysis, it is clear that an implementation of a statistical compressor in hardware is not an attractive design selection. It would be cost-prohibitive in terms of on-chip area and would be low in respect to throughput. Moreover, the compression ratio of a statistical compressor is lower than others. In Fig. 3.1, the compression ratio of Dynamic Huffman coding is the lowest. Actually, the implementation of compression is mainly depended on finding and reducing redundancy. Nevertheless, statistical algorithms have not this function. They are commonly used to optimize the compression ratio. Hence, statistical

algorithms usually work together with dictionary based algorithms or the other types of algorithms. Such as, in the application of Gzip, Huffman coding is used to improve the compression result of the compression of LZ77. For combinational algorithms, the complex algorithms produce the best compression ratio but introduce huge area overhead. In addition, the hardware realization of combinational models are quite complex for achieving high throughputs. On the other hand, dictionary based algorithms introduce acceptable area overhead and high throughput but sacrifice the compression efficiency [27].

In general, according to the requirements of this thesis, a dictionary based algorithm will be the best choice based on the following facts. 1) A dictionary based compressor does not require prior knowledge or statistical information of the symbols. Hence it can be in real-time. 2) A dictionary based compression has low complexity and can achieve high throughput with small hardware area and an acceptable compression ratio.

3.1.3. LZ77 vs. LZ78

Dictionary based algorithms have two basic categories, LZ77 and LZ78. In this section, the selection between LZ77 and LZ78 is discussed. The comparison is performed in two aspects, complexity and compression ratio.

3.1.3.1 Compression Ratio

Fig. 3.3 depicts the compression ratio of LZ77 and LZ78 algorithms compressing Nexus trace data files. The result shows that LZ77 algorithm outperforms LZ78 algorithm for smaller volumes of input data, which typically characterize Nexus trace data. When the dictionary size is 512 bytes, the compression ratio of LZ77 is 1.64, which is better than the compression ratio 1.29 of LZ78. LZ78 algorithm performs effectively on large volumes of trace data. When the dictionary size is larger than 16M bytes, the compression ratio of LZ78 exceeds the compression ratio of LZ77. In addition, according to the work principle of LZ77 and LZ78, the smallest dictionary size needs to be 512 bytes for LZ78 compressor. Conversely, LZ77 can use 256 bytes dictionary. Therefore, LZ77 is more suitable to compress Nexus trace data for small hardware area consuming.

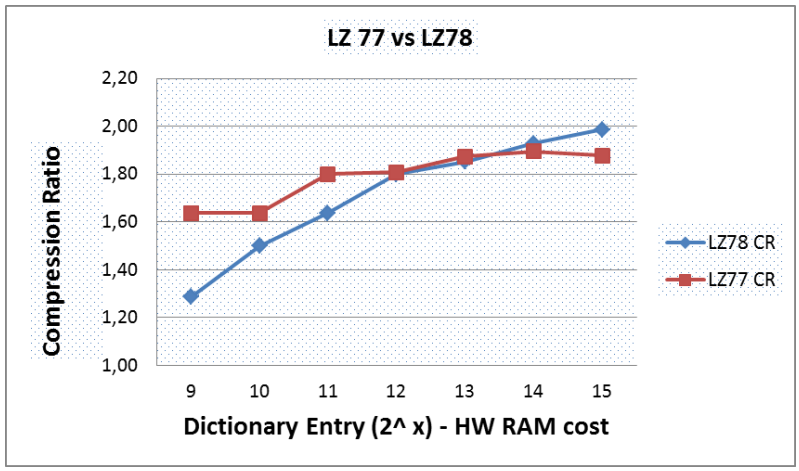


Fig. 3.3. Compression ratio of LZ77 and LZ78.

3.1.3.2 Complexity

Table 3.2 shows the average time complexity of LZ77 and LZ78 algorithms. The compression speed for both LZ78 and LZ77 algorithms depend on finding the maximum matching strings from the dictionary. For LZ77 algorithm, a hash table is used to find the match strings. LZ78 uses trie data structure to achieve matching strings [11]. The complexity of LZ77 is O(1), in contrast, the complexity of LZ78 is O(n) [28]. Hence LZ78 is more complex than LZ77, which is supported by the result of compression time in Fig. 3.4. In Fig. 3.4, LZ78 take more time to compress the same amount of data. Therefore, it is more likely to achieve high throughput with small area overhead using the LZ77 algorithm.

TABLE 3.2. COMPLEXITIES OF LZ77 AND LZ78.

Algorithm	Data Structure	Time Complexity(average)
LZ77	Hash Table	O(1)
LZ78	Trie	O(n)

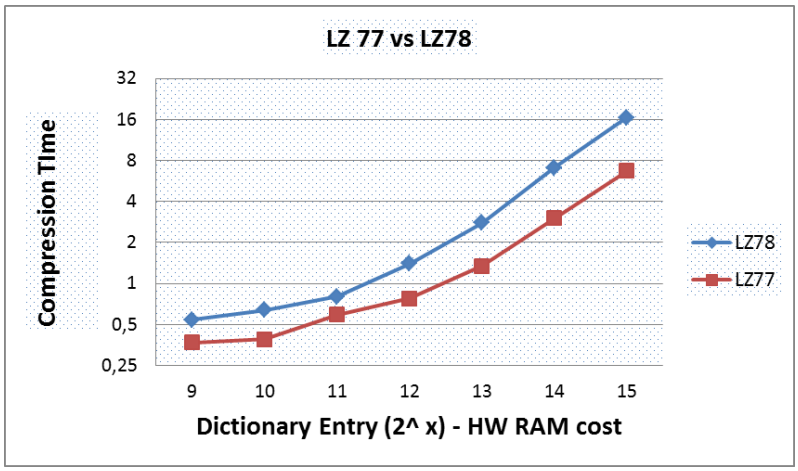


Fig. 3.4. Compression time of LZ77 and LZ78.

3.1.3.3 Scalability

Using LZ77 algorithm is more convenient to achieve a complex compressor. Most of combinational algorithms are a combination of LZ77 and a statistical algorithm. For example, Gzip compressor is the combination of LZ77 and Huffman coding; LZMA compressor is the combination of LZ77 and arithmetic coding.

In a conclusion, LZ77 could be a good choice for Nexus trace data. In the following section, LZ77 algorithm and its implementation are discussed.

3.2. LZ77 Algorithm

The LZ77 algorithm was proposed by Ziv and Lempel in 1977[8]. It is a dictionary based compression algorithm and does not require prior knowledge or statistical characteristics of the symbols. This character makes the LZ77 algorithm suitable for real-time compression.

The compression idea behind the LZ77 algorithm is to find a match string in a dictionary and replace the same string by a triplet (*distance*, *length*, *next symbol*). The third component "*next symbol*" is needed in cases where no string has been matched. However, the compression performance of LZ77 will be reduced when the third component is a part of every triplet.

In this thesis, a variation of the LZ77 algorithm used in Deflate algorithm is implemented. The algorithm eliminates the third component "*next symbol*" and writes a pair (*distance*, *length*) on the compressed stream [11]. *Distance* indicates the offset from the start of the match symbols found

in the sliding window to the current symbol. *Length* indicates the length of the match symbols. When no match is found, the compressed output is (*0, literal*). When a match is found, the compressed output is (*1, distance, length*).

Considering that the available memory is limited, LZ77 algorithm uses a sliding window. The window can be divided into two buffers, one is the searching buffer, called dictionary, containing *N* symbols. The dictionary maintains the strings used recently. The other is the coding buffer, called look-ahead buffer, containing *M* symbols. The look-ahead buffer contains the symbols to be processed.

The LZ77 processes data from left to right, inserting every string into the dictionary and outputting the compressed results. Data compression is achieved by performing four steps.

The first step is to initialize the dictionary and the look-ahead buffer. The dictionary is filled with zeros and the look-ahead buffer is filled with the input strings.

The second step is to search the dictionary and find the longest match L_{max} for a string in the look-ahead buffer. Finding the longest match string is a key and the most time-consuming operation in the algorithm. The approach to find the maximum matching string is based on a hash table structure.

The third step is to output the compressed results. If a match string cannot be found in the dictionary, the output is a literal, (*0, literal*). If a match is found, the output is a distance-length pair, (*1, distance, length*). The value of distance is the distance from the current string to the start of the matching string, the value of length is the match length.

The last step is to insert the processed symbols into the dictionary and input new symbols into the look-ahead buffer. If a match is found, L_{max} new symbols are placed in the look-ahead buffer by left-shifting the symbols in both the look-ahead buffer and the dictionary.

An example is described to illustrate the compression algorithm. Supposing the input sequence is “aacaacabcabac”. (Dictionary size N = 6 bytes, Look-ahead Buffer size M=4 bytes, min match length=2 bytes, max match length L_{max}=4 bytes). The compression procedure is shown in Fig. 3.5.

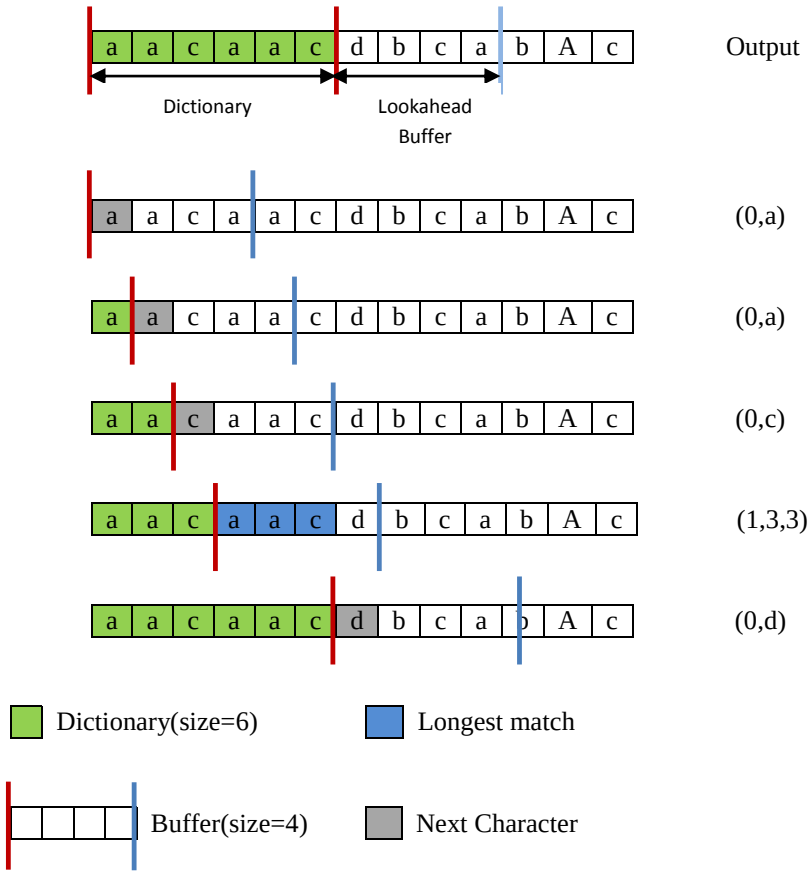


Fig. 3.5. Example of LZ77 compression.

In the beginning, the dictionary is empty. The look-ahead buffer is filled with “aaca”. There is no match symbol found in the dictionary, the output is a literal (*0, a*). The processed symbol “a” is moved in the dictionary with index 0. The new input symbol “a” is moved into the look-ahead buffer.

The process restarts by finding the match symbol in the dictionary. There is a match “a” found in the dictionary. However, the match length is less than the minimum match length 2 bytes. Therefore, the output is a literal (*0, a*). The processed symbol “a” is moved in the dictionary with index 0. The new input symbol “c” is moved into the look-ahead buffer.

The process restarts. When there is no matching symbol found in the dictionary, the output is a literal (*0, c*). The processed symbol “c” is moved

in the dictionary with index 0. The new input symbol “d” is moved into the look-ahead buffer.

The process starts again. There is now a new matching string found in the dictionary. The longest match length is 3 bytes, which is longer than the minimum matching length 2 bytes. The distance is 3 from the current symbol to the beginning matched symbol. Therefore, the output is a literal, (*1, 3, 3*).

The compressed steps continue until the end of the input file.

3.3. LZ77 Algorithm Simulation

In this section, the simulation results with different values of parameters on the LZ77 algorithm performance are presented. The simulation code is implemented in C. The performance is evaluated according to the simulation results when the Nexus trace data generated from Ericsson’s ASIC platform are compressed.

3.3.1. Parameters Analysis

In order to optimize the solution, there are three aspects, compression ratio, compression speed, and hardware cost, which need to be explored when implementing the LZ77 algorithm. There are six parameters affecting these three performances. The parameters are dictionary size, look-ahead buffer size, hash table size, minimum match length, minimum move length, and the length of code word. In this part, an analysis on the influence of these parameters on the performances is discussed.

- Dictionary Size

The larger the dictionary size is, the more the match symbols can be found. Thus the compression ratio can be improved with the increased dictionary size. However, the increased dictionary size will slow down the compression speed. Because that the increased amount of matching iterations make the compressor take more time to search the dictionary and find the longest match string. At the same time, the larger dictionary size means increasing hardware area.

- Hash Table Size

Increasing the hash table size can improve the compression speed, since a larger hash table size can reduce the hash collision probability and the amount of matching iterations. However, a larger hash table size needs more memory space; hence higher hardware cost.

- Look-ahead Buffer Size

The size of look-ahead buffer is the maximum match length, which is an input parameter depending on the input file and compression requirements. A larger look-ahead buffer brings higher hardware cost. The effect of the look-ahead buffer size on the compression ratio and the compression speed is related to the characteristic of the input file, which is needed to be analyzed through simulation.

- Minimum Move Length

Minimum move length means the number of symbols output from the look-ahead buffer. Increasing the minimum move length means a decreased number of match symbols, namely, a worse compression ratio. Basically, there is no effect on the compression speed and hardware cost by increasing the minimum move length.

- Code Word Length

The compression of LZ77 algorithm is achieved through transforming variable length strings into fixed length code word (*1, distance, length*). That means, the more match strings and the longer match strings are found and replaced by the code word, the better compression ratio is obtained. At the same time, the shorter code word is used, the better compression ratio is achieved. Hence the compression efficiency depends on the size of compressed result, namely the length of code word (*1, distance, length*).

A code word consists of 3 parts, (*1, distance, length*). The flag part "1" needs 1 bit. The value of distance is the offset from the start of the match symbols found in the dictionary to the current symbol. Therefore, the distance part is determined by the dictionary depth. The range of it is from 1 to the depth of dictionary minus 1, which needs $\log_2^{(\text{dictionary depth})}$ bits. For example, if the depth of dictionary is 256, the distance part needs 8 bits to represent the range of distance.

The value of length indicates the length of the match symbol. It is determined by the look-ahead buffer size. The maximum length is the length of look-ahead buffer. It needs $\log_2^{(\text{look-ahead buffer length})}$ bits. For example, if the look-ahead buffer size is 8 bytes, the length needs 3 bits to represent the range of length.

- Minimum Match Length

Minimum match length is the match length, which is allowed to be replaced by the code word. The minimum match length depends on the code word length. In order to achieve a compression, the minimum match length

must be longer than the code word length. Otherwise, the compressed Nexus trace file will expand other than shrink.

For example, in this paper, if the code word length is 10 bits or 11 bits, thus the minimum match length must be 16 bits. If the match string is less than 2 symbols, we do not replace the match string. Such as, if the match string is 8 bits, the file will be expanded if the match string (8 bits) is replaced by a code word (10 bits or 11 bits).

Increasing minimum match length does not affect the compression ratio. However, the compression speed will be improved by increasing the minimum match length. Since the probability of finding the matching strings is reduced through increasing the minimum match length.

Based on the above analysis, the effects of the parameters on the performances are not independent. Such as, increasing the dictionary size can improve the compression ratio. However, the compression speed would be slow and the hardware cost would be increased. In addition, these parameters also interact with each other. Like the dictionary size determines the range of code word, the length of code word determines the minimum match length. Hence, optimizing the algorithm parameters is necessary to trade off the compression performances (compression speed, compression ratio) and hardware cost. To find optimized parameters, some simulations with different parameters´ values need to be performed.

3.3.2. Simulation Results

Simulation with different parameters setting for the performances of the LZ77 algorithm is performed using the Nexus trace data. Nexus trace data are generated from Ericsson hardware platform.

1) Dictionary Size & Hash Table Size

The compression ratio and the compression speed with the different dictionary size and hash table size are shown in Fig. 3.6 and Fig. 3.7. The range of dictionary size M is from 256 bytes to 32K bytes. The range of hash key N is from 8 bits to 15 bits. The hash table size is equal to $\log_2{}^M * 2^N$ bits.

Fig. 3.6 shows that when the dictionary size is the same, increasing the hash table size does not effect on the compression ratio. The compression ratio increases as the dictionary size increases. Fig. 3.7 shows that the compression time increases with the dictionary size. The increased hash table size reduces the compression time.

Increasing the dictionary size brings a higher compression ratio, conversely, the hardware cost and the compression time increases exponentially. Considering the high throughput requirement and CR=1.5(about 33%) is acceptable, a dictionary size of 256 bytes is a reasonable choice for hardware implementation with a good compression efficiency. Hash key N of 15 bits could be a better choice to achieve a high compression speed.

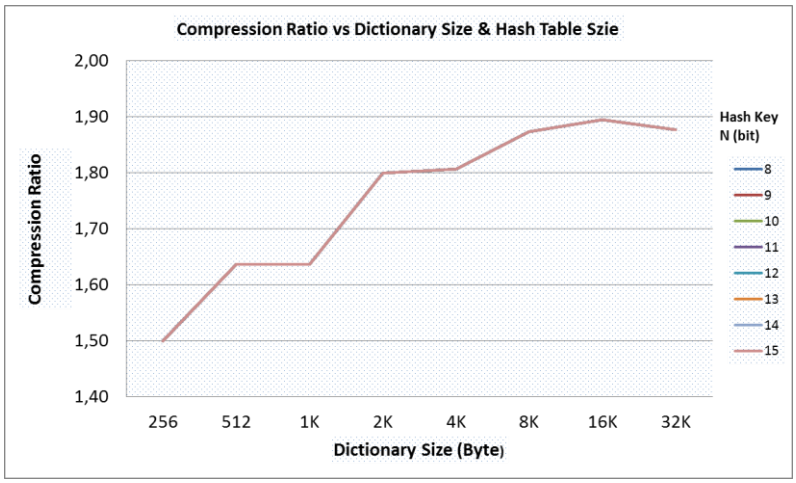


Fig. 3.6. Compression ratio with different dictionary size and hash table size.

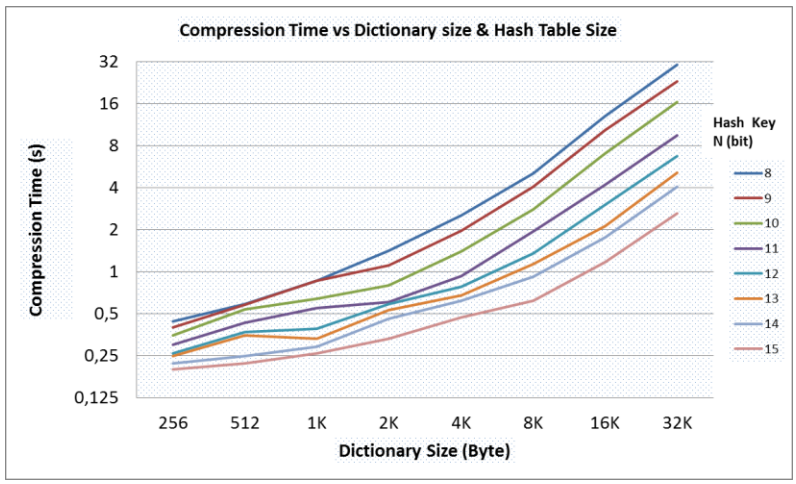


Fig. 3.7. Compression time with different dictionary size and hash table size.

2) Look-ahead Buffer Size

Fig. 3.8 and Fig. 3.9 shows the compression ratio and the compression speed how to change with the different dictionary size and look-ahead buffer size.

The size of the look-ahead buffer is equal to the longest match length, which is an input parameter of the algorithm. The effect of the look-ahead buffer size on the compression ratio and the compression speed is related to the characteristic of input file.

As shown in Fig. 3.8, for different dictionary size, the highest compression ratio could be achieved when the look-ahead buffer size is equal to 4 bytes. When the dictionary size is 256 bytes, the best compression ratio is found when the look-ahead buffer size is from 2 bytes to 8 bytes. Fig. 3.9 shows that for a given dictionary size the look-ahead buffer size do not impact very much on the compression time as all curves are almost flat.

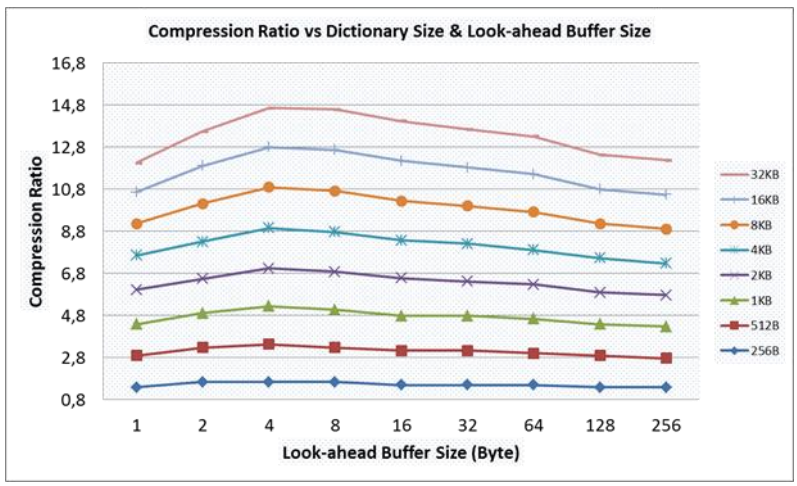


Fig. 3.8. Compression ratio with different dictionary size and look-ahead buffer size.

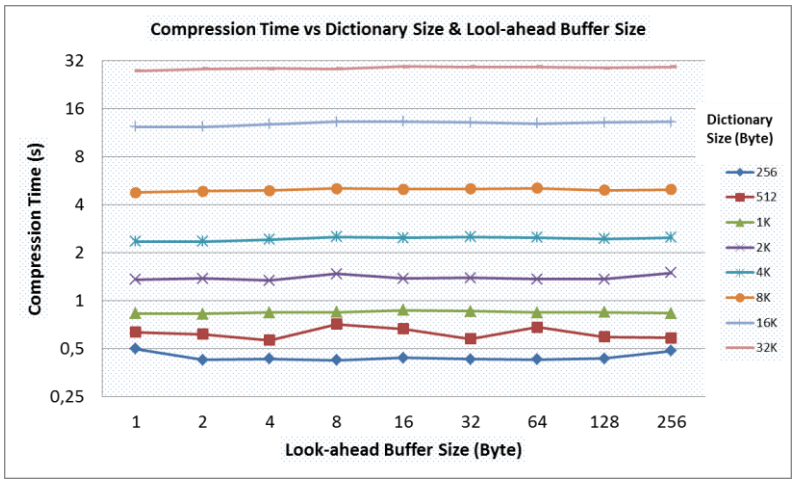


Fig. 3.9. Compression time with different dictionary size and look-ahead buffer size.

3) Minimum match length

In Fig.3.10, the minimum match length has no impact on the compression ratio. In Fig.3.11, the minimum match length slightly affects the compression time. The longer the minimum match length is, the shorter the compression time is. Hence, the minimum match length of 24 bits is a good choice considering the compression speed requirement.

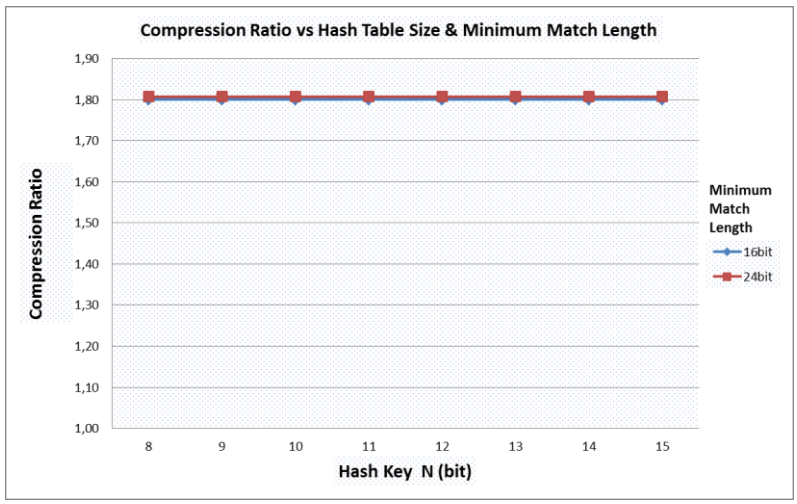


Fig. 3.10. Compression ratio with different hash table size and minimum match length.

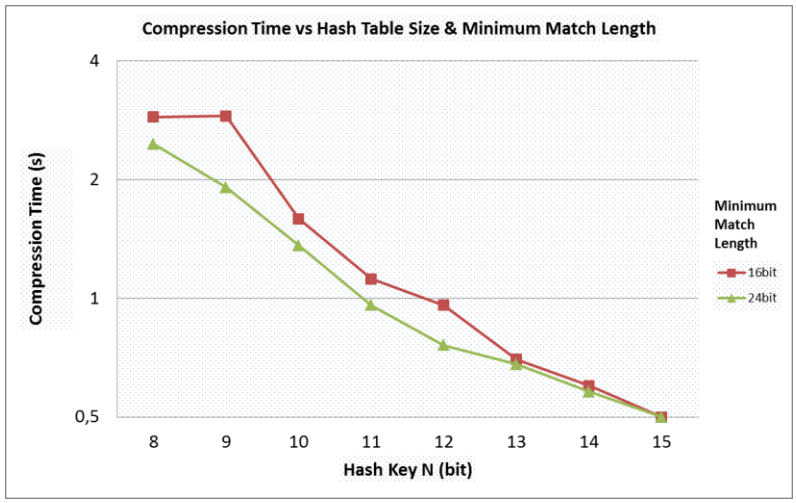


Fig. 3.11. Compression time with different hash table size and minimum match length.

4) Minimum move length

In Fig. 3.12, the bigger minimum move length will lead to a lower compression ratio. Fig. 3.13 shows that the minimum move length has no impact on the compression time. Hence, in order to achieve a better compression ratio, minimum move length could be choose as 1 byte.

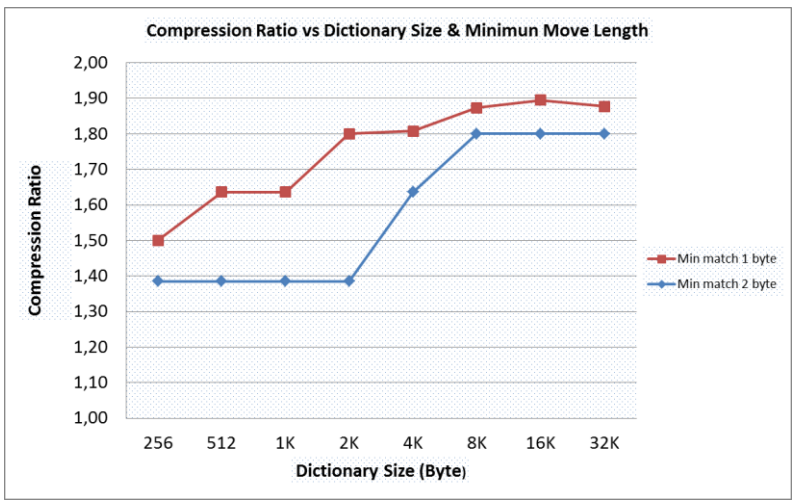


Fig. 3.12. Compression ratio with different dictionary size and minimum move length.

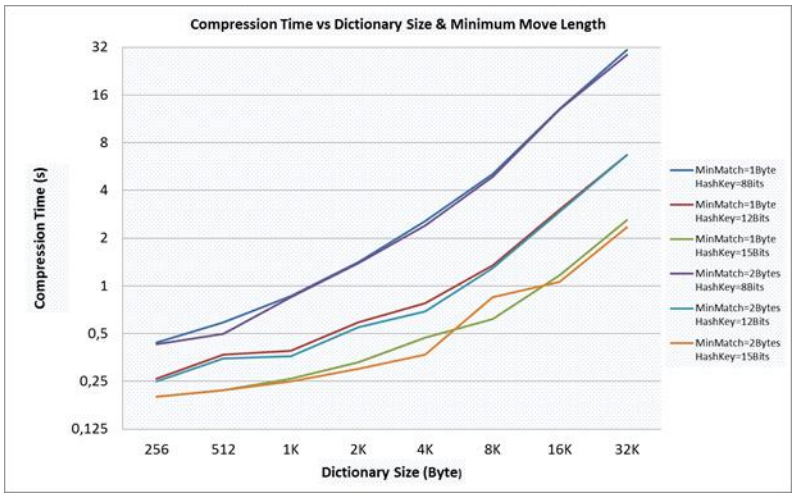


Fig. 3.13. Compression time with different dictionary size and minimum move length.

According to the above analysis, table 3.3 shows the selected compression parameters, which can achieve a good trade-off between hardware complexity and application performance requirements.

TABLE 3.3. PARAMETERS SELECTION.

Compression Parameters Size(bits)		Hardware Cost(Bytes)		Compression Performance	
Dictionary Size	256*8	Dictionary RAM	256	Compression Ratio	30% (estimated)
Look-ahead Buffer Size	4*8	Hash Table1 RAM	32K	Compression Throughput	16bits/cc (estimated)
Minimum Match Length	24	Hash Table2 RAM	256		
Minimum Move Length	8				
Hash Key Width	15				

CHAPTER 4

4. LZ77 Hardware Implementation

This Chapter describes the LZ77 hardware implementation. Section 4.1 proposes a modified LZ77 algorithm. Section 4.2 provides a structure overview of the LZ77 encoder and Section 4.3 gives an in-depth explanation of the LZ77 hardware implementation.

4.1. Optimized LZ77Algorithm

In this section, based on the characteristic of Nexus trace data, an optimized LZ77 algorithm will be discussed. The main aim of optimizing the LZ77 algorithm is to improve the compression speed.

According to the principle of LZ77, the common structure of LZ77 compressor is shown in Fig. 4.1. There are 4 pieces of RAM, dictionary RAM, hash RAM1, hash RAM2, and look-ahead buffer RAM. Dictionary RAM is used to store the input Nexus trace data. Look-ahead buffer contains Nexus trace data, which are needed to be processed. Hash RAM1 and hash RAM2 maintain the address information of Nexus trace data in dictionary. Controller is used to find the match string and output the compressed results.

For LZ77 encoding, the main task of the compressor is to find a match string. Finding the longest match string in LZ77 algorithm is based on the hash table chain structure [11]. The hash RAM1 and hash RAM2 contain the address information of a 3-symbol string in the dictionary. LZ77 compressor computes a hash value using a 3-symbol string output from the look-ahead buffer. The hash value is used as an index of the hash RAM1 that has 2^{15} entries. A 3-symbol string is hashed according to the hash value. The following cases can occur when we search for the maximal match string. (1) If there is no the address information in the hash RAM1, which means no match string can be found in the dictionary. Then the offset of a 3-symbol string in the dictionary is stored in the hash RAM1. (2) If the hash RAM1 has the offset information and the hash RAM2 has no offset information, which means just one match might be found, thus the offset in the dictionary of a 3-symbol string is stored in the hash RAM1 and the old offset information stored in the hash RAM1 are moved into the hash RAM2. At the same time, the encoder examines the match string. (3) If both the hash RAM1 and hash RAM2 have the address information, which means a

hash collision happens. The performance of finding the maximum match string will be performed according to the address information in both the hash RAM1 and hash RAM2. A hash collision occurs basically as the hash index has 2^{24} values and the hash RAM1 has 2^{15} entries. It means that many 3-symbol strings are mapped to the same entry in hash RAM1. When a hash collision occurs, the old address information stored in the hash RAM1 are moved into the hash RAM2, and the new address information is stored in the hash RAM1. This is time consuming.

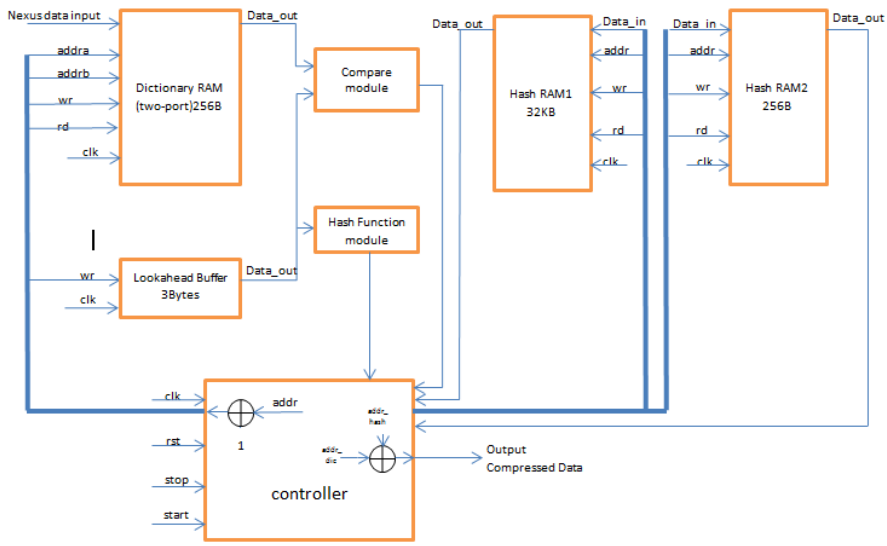


Fig. 4.1. Structure of the LZ77 compressor.

Finding the maximum match string is the key and the most time consuming operation in the compression process. The compression speed mainly depends on the time of finding the longest match string in the dictionary. In this thesis, the input speed of Nexus trace data is 16 bits/clock cycle. In order to satisfy the real-time requirement, the operation time for data compression cannot be larger than the input rate of trace data. However, when the hash collision occurs, many clock cycles are used to read the dictionary and match the string. This makes it hard to meet the requirement of compression speed (16 bits/clock cycle) at a small hardware cost. To address this problem, an optimized LZ77algorithm is proposed. There are two aspects modified. One is the format of code word. The other is the hash table.

- Code Word

We propose to change the format of a code word from (*1, distance, length*) to (*1, distance*). It means that we have fixed match length. The code word design is crucial in achieving a good compression ratio. It should be noted that the compression efficiency greatly depends on the ratio of code word length and the length of replaced match string. Higher compression ratio can be obtained when the code word length is shorter.

In general a code word contains 3 parts, which is flag, distance, and length. The distance part is determined by the dictionary depth. In this thesis, the depth of dictionary is 128 (2^7) bytes, the distance need 7 bits. The flag part "1" is 1 bit. The length part, which is determined by the look-ahead buffer size, represents the maximum match length. In Fig. 3.8, the compression ratio is the highest when the maximum match length is 2 bytes to 8 bytes when the dictionary size is 256 bytes. If the maximum match length is selected to 2 bytes, the length should be 2 bits. Thus the code word length is 10 bits. If the maximum match length is 8 bytes, the length should be 3 bits. Then the code word length is 11 bits. A consequence is that there is no point to compress an input symbol of length 8 bits, as it will result in a compressed result of 10 (or 11) bits. Thus we set the minimum match length to be 16 bits when the code word length is either 10 bits or 11 bits.

We can make the code word length shorter by setting the maximum match length to be equal to the minimum match length. The format of code word is then simplified from (*1, distance, length*) into (*1, distance*). The advantage is that the information about length is not needed as every match length is 2 bytes. The benefit is that instead of using 10 bits, we only need 8 bits. The compression ratio can be increased when the same match string can be replaces by shorter code word.

- Hash Table

The hash table is optimized by avoiding hash collisions, which can improve compression speed. When the minimum match length is 3 bytes and the hash table entries are set at 2^{24}, no hash collision can occur. However, the hardware cost will be 16M bytes, which is cost-prohibitive. Instead, if the minimum match length is 2 bytes, the hash table will be 64K bytes and no hash collision will occur. Compared to the hardware cost 16M bytes, the hardware cost 64K bytes can be accepted. The advantage with the 2 bytes alternative is that the cost of memory is significantly reduced. At the same time, the compression speed is improved. Finding match string can be implemented in one clock cycle.

In general, the hardware cost and compression performances can be traded off through optimizing the LZ77 algorithm. The parameters of optimized LZ77 algorithm are listed in table 4.1.

TABLE 4.1. PARAMETERS SELECTION.

Compression Parameters Size(bits)		Hardware Cost (bytes)	
Dictionary Size	128*16	Dictionary RAM	256
Look-ahead Buffer Size	2*8	Hash Table1 RAM	64K
Minimum Match Length	16		
Minimum Move Length	16		
Hash Key Width	16		

4.2. Hardware System Structure

4.2.1. Hardware Approaches

To fulfill real-time compression and speed up the string match time, two major hardware implementation methods are presented in the literature [17], [22], and [29]. One is Content Addressable Memories (CAM) approach, the other is systolic array approach.

- CAM approach

The CAM approach performs string match by full parallel searching. A CAM-based LZ77 compressor can process one input symbol per clock cycle. CAM has been considered the fastest architecture among all proposed hardware solutions. However, CAM uses much hardware and high power consumption [22]. The area of the CAM is approximated to be twice as much as the area of a RAM of the same capacity [17].

- Systolic Array Approach

The systolic array is a regular pattern of processing elements interconnected in a simple way. Each processing element is connected to its adjacent element. The basic idea is to lay out an identical pattern of processing elements with simple interconnections [29]. Systolic array approach performs string match by pipelining. Compared with CAM approach, systolic array compressors are slower, however better in hardware cost [22].

Both the CAM approach and the systolic array approach result in poor portability. Hence, a state machine approach, which can be easily transferred to different architecture, is adopted in this thesis.

4.2.2. Architecture Overview

The hardware structure of the optimized LZ77 encoder is illustrated in Fig. 4.2. The hardware of LZ77 compressor consists of a controller (finite state machine) and two two-port RAM. In this application, a format block is added based on the platform requirement. According to the parameters selection in section 4.1, the size of dictionary memory is 128*16 bits and the hash table memory size is 2^{16} *8 bits.

The controller block, which is a finite state machine (FSM), is responsible for processing synchronization and administrating the collaborations between different sub-modulations. Nexus trace messages are stored in the dictionary RAM and sequentially clocked into the system. The addresses of processed trace messages in the dictionary are stored in the hash table RAM. The compressed results are sent to the format block and output in 64 bits/line format from the format block.

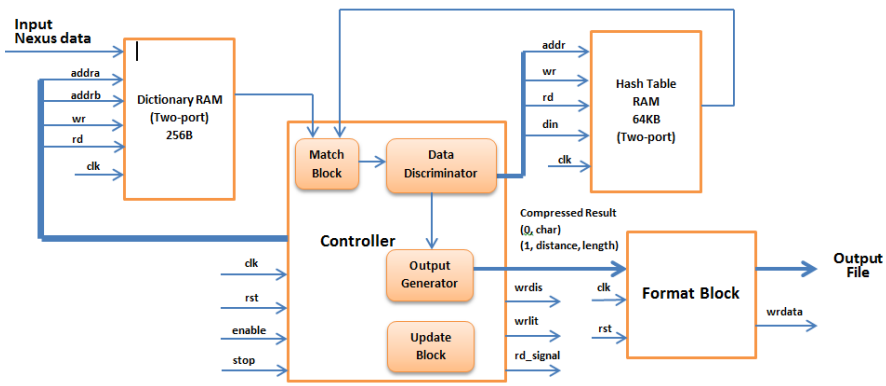


Fig. 4.2. Modified LZ77 compressor structure.

4.2.3. Operating Principle

The whole system works in one clock domain and the reset is an active low input signal used to reset the entire system. When the system receives a "enable" signal, the system starts to work and sends "rd_signal" signal to the testbench. Then the testbench starts to send the Nexus trace data to the compressor.

The Nexus trace messages are firstly stored in the dictionary RAM. When the first trace message is stored, the system starts to read the trace messages from the dictionary RAM and finds the match string in the dictionary RAM through checking the index of the hash table. If a match string is found, the controller sends a pair (*1, distance*) to the format block with a high signal "wrdis". If no match string is found, the signal "wrlit" is set high and the original literal is sent to the format block in (*0, literal*) format. In the format block, the trace messages are sent out in 64 bits/line format when the signal "wrdata" is high.

4.3. Block Descriptions

In this section, the functions of all the sub-modules are described. Both the controller and the format blocks are implemented with a state machine approach.

4.3.1. Storage Blocks

In Fig. 4.2, there are two two-port RAMs used in the compressor: a dictionary RAM and a hash table RAM.

- Dictionary RAM

The dictionary RAM is used to store the input trace data. Considering the input speed of Nexus trace data is 16 bits/clock cycle and the size of dictionary is 256 bytes, therefore the RAM size is 128*16 bits.

Every row of the dictionary is used as the look-ahead buffer since the minimum move length and the minimum match length are 2 bytes.

The initial values of the dictionary RAM are zero. When the compressor starts to work, the trace messages are stored in the dictionary RAM and sequentially clocked into the look-ahead buffer for processing.

- Hash Table RAM

The hash table RAM contains the address information of the trace data in the dictionary. The entries of the hash table are 2^{16}. The RAM size is 2^{16}*8 bits.

The initial values of the dictionary RAM are 128. The highest bit of every row is a flag for refreshing the hash table. If the row is never written, the value of the bit is 1. Otherwise, if the row is written, the value of the bit is 0. The other 7 bits are used to store the address of the trace data in the dictionary since the maximum distance of a match symbol is 127.

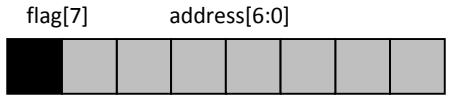
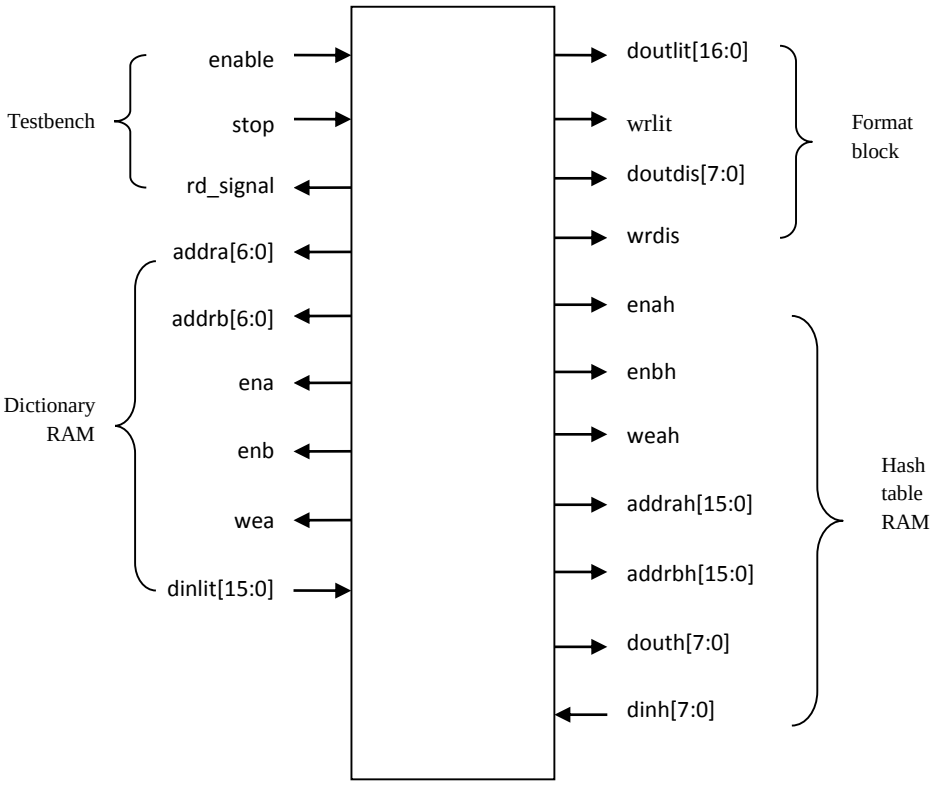


Fig. 4.3. Description of hash table RAM.

4.3.2. Controller Block

The controller block is the main functional block. The actual implementation in the controller block can be broken down into four key parts: match block, update block, data discriminator, and output generator. These blocks are detailed below.

4.3.2.1. Interface Overview



4.3.2.2. Functionality

When the compressor begins to work, it assigns the correct start-up values to all variables. The hash table RAM needs to be set the initial value 128. The initial values of literal register group are 65536. The initial values stored in dictionary RAM are 0.

- Match block

When this block is trigger by the "enable" signal, the Nexus trace data are sequentially clocked into and stored in the dictionary RAM. The input speed of trace data is 16 bits/clock cycle, the width of dictionary RAM is 16 bits. Therefore the trace symbol, which is 16 bits wide, can be written in a clock. After data is written in the dictionary RAM, Nexus trace data is read out in the next clock cycle. The reading operation is always delayed one clock cycle compared to the writing operation. As soon as Nexus trace data are available, the matching operation begins.

The trace data read from the dictionary RAM is used as the hash table address to lookup the hash table. If the value in the corresponding address of hash table is 128, no match string is found in the dictionary RAM. The compressor output the pair (*0, literal*). The address of the trace symbol in the dictionary RAM is then written into the corresponding address of the hash table. For example, assuming the trace symbol is 13 and its address in the dictionary RAM is 27. The compressor will check the hash table address 13 and write 27 in the address when there is no match symbol found.

If the value in the corresponding address of hash table is not 128, a match string is found in the dictionary RAM. The controller reads the value from the Hash table RAM and calculates the difference between the value output from the Hash table RAM and the address of trace data in dictionary RAM. The compressor output the pair (*1, distance*). For example, assuming the trace symbol is 13 and its address in the dictionary RAM is 27, the value in hash table address 13 is 5. The compressor will check the hash tables address 13 and output (*1, 27-5*) when there is a match symbol found.

- Update Block

The hash table needs to be refreshed when the dictionary is full since the contents of hash table are the address information of trace symbols in the dictionary RAM. This means when a trace symbol is written in the address 127 of the dictionary RAM. The hash table will be refreshed.

In this thesis, the compressor uses a register group and a two-port RAM to refresh the hash table instead of two pieces of RAM considering the hardware cost and compression speed. Using this approach, the system does

not need to stop the compression process and spend extra clock cycles waiting for the hash table to be refreshed. The advantage is that the compression speed can always be kept at 16 bits/clock cycle.

The register group has 128 registers and the width of the register is 17 bits. The compressor initializes each register with 65536 as the dictionary RAM is full. The highest bit of the register is a flag for judging whether the register is modified. If the value of the flag is 1, the register is never written. Otherwise if the register is modified with writes, the value of the flag is 0. The other 16 bits is for storing the trace symbols.

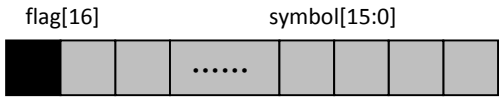


Fig. 4.4. Description of register group.

Every time when the addresses of trace symbols are stored in the hash table RAM, the trace symbols are also written in the register group. When the output from the hash table is not 128, the system will check the register group. If the same trace data can be found, it means there is a match string in the dictionary RAM. The compressor output the pair (*1, distance*). Otherwise if there is no the same trace symbol found, it means no match string in the dictionary RAM. The address of the trace symbol in the dictionary RAM is written into the corresponding index of the hash table. The trace symbol is stored in the register group. The compressor output the pair (*0, literal*).

- Output data

The replacement principle is as following. When a match string is found, the output data is a pair (*1, distance*). When no match string is found, the output data is a pair (*0, literal*).

Data compression is achieved through finding a match string in the dictionary and replacing the match string in the look-ahead buffer with the distance from current string to the match string. When a match string is found, the string is compressed and the output data is a pair (*1, distance*). In this thesis, the length of a string is 16 bits. The size of dictionary is 128*16 bits. Hence the maximum distance is 127, which can be represented with 7 bits. The length of a pair (*1, distance*) is 8 bits. Therefore, a trace file can be compressed through replacing a string which is 16 bits with a pair (*1, distance*).

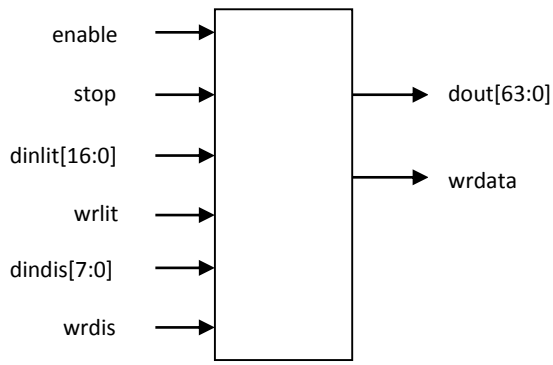
- Data Discriminator

This block is used to discriminate if there is a match string in three successive trace data. A pipeline is used to improve the throughput. There is one clock cycle delay when read of the trace data from RAM. So the special situation when there are the same trace symbols are input successively within three clock is needed to be discriminated in advance to compensate the delay of reading operations from the dictionary RAM and the hash table RAM.

There are three special situations to be discriminated when the trace messages are sequentially clocked into the system. The first situation is that three same trace symbols are input continuously, such as “222”. The second situation is that two same trace symbols are input continuously, such as “221”. The last situation is that two same trace symbols are input and a different symbol is put between these two symbols, such as “212”.

4.3.3. Format Block

4.3.3.1. Interface Overview



4.3.3.2. Functionality

This block is used to arrange the format of the compressed results from the controller. The compressed results are either a pair (*0, literal*) with 17 bits or a pair (*1, distance,*) with 8 bits. The width of data is 64 bits/line outputted from the format block. Since the width of an output data is 64 bits, which cannot be divided exactly by 17 bits, there are several specific situations to be considered. Table 4.2 lists the situations in detail.

When the 64 bits output signal “dout” is full of compressed messages, the signal “wrdata” becomes high and the data is sent out to the test bench.

TABLE 4.2. INPUT AND OUTPUT LOGIC TABLE.

Width Y of input data	Empty bits X of output	Condition
17 bits (*0*, *literal*)	X=17 bits	X=Y, the literal can be put exactly.
17 bits (*0*, *literal*)	X<17 bits	X<Y, only part of the literal can be put, the next input data is a literal.
		X<Y, only part of the literal can be put, the next input data is a pair (distance, length).
8 bits (*1*, *distance*)	X=8 bits	X=Y, the literal can be put exactly.
8 bits (*1*, *distance*)	X<8 bits	X<Y, only part of the literal can be put, the next input data is a literal.
		X<Y, only part of the literal can be put, the next input data is a pair (distance, length).

4.4. Tests and Results

To evaluate the LZ77 compressor, a series of tests are completed on a set of Nexus trace data. Fig. 4.5 shows the test approach, the compressed files are decompressed and the correctness of compressed results is analyzed through comparing the decompressed file with the original input file. The decompression program is implemented in C code.

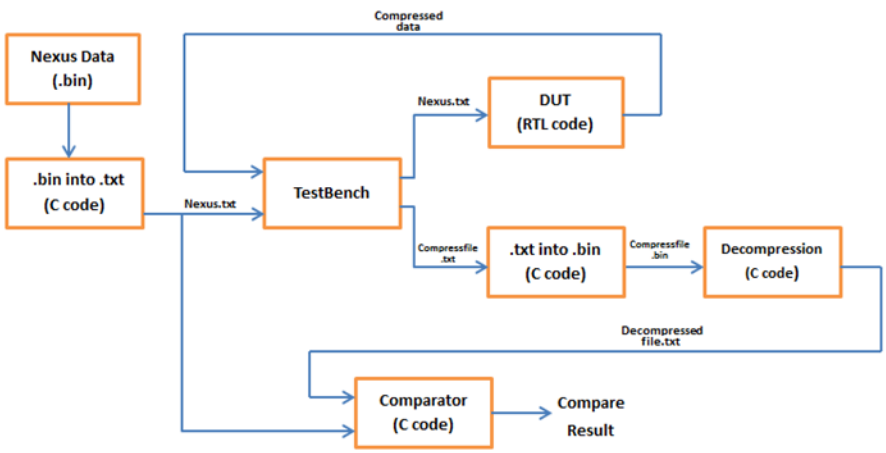


Fig. 4.5. Verification environment.

The original input file is binary files, which need to be converted into .txt files. The LZ77 compressor processes the input files and sends the compressed results to the test bench. The compressed data are generated into compressed files through the test bench. The compressed files are text files and need to be converted into binary files. After decompressed by the decompression program, the decompressed files are compared with the original Nexus files. The comparator shows the analysis result.

4.4.1. Evaluation Method

- Test file

All test files are the Nexus trace data generated by DSPs in Ericsson hardware platform. The files are named file1, file2, and file3, which are listed in Table 4.3.

- Compressor

All these compressors described in the following are implemented in software.

Compressor using dictionary based compression algorithms:

(1) LZ77 algorithm compressor: LZ4.

(2) LZ78 algorithm compressor: LZW.

Compressor using statistical compression algorithms:

(3) Dynamic Huffman coding

Compressor using combinational compression algorithms:

(1) Bzip2, LZMA, Gzip

- Compressor Performance Setting

All compressors use the "-fast" option. In order to obtain the best compression performance, the dictionary size is increased to the maximum size allowed in each compressor.

4.4.2. Test Results

- Compression Ratio

In Table 4.3, the compression ratio for each of the three files is reported for each compression algorithm. The average compression ratio of this thesis is 1.35, which is lower than the compression ratio requirement 2 proposed in chapter 1. However, the achieved compression ratio is based on the smallest dictionary size 256 bytes. In contrast, the compression ratio of LZW is 1.32 with dictionary size 512 bytes. The compression ratio of Gzip

is 2.16 with the dictionary size 32K bytes. The compression ratio of LZ4 is 1.62 with the dictionary size 16K bytes. In Section 3.3.2, we know increasing the dictionary size can improve the compression ratio. Hence the compression ratio can be improved by increasing the dictionary size. The simulation results show that the compression ratio can achieve 1.9 when the dictionary size is 32K bytes. We notice that the compression ratio, 2 to 5, is hard to achieve if only one type of compression algorithms is used, unless a huge dictionary size is adopted. A combinational compression algorithm is recommended to get a 2 to 5 compression ratio. Such as, all the compression ratios of combinational algorithms are higher than 2 in Table 4.3. At the same time, we compare the compression ratio with optimized code style against the compression ratio with the regular code style. The results are collected in Table 4.4. We observe that the optimized code word style improves the compression ratio.

- FPGA Resources Utilized

Table 4.5 summarizes the FPGA utilizations for the hardware implementation of the compressor. The design is compiled using Xilinx ISE13.4 software. The type of FPGA is Virtex7XC7VX485T-1. The delay and the hardware cost are mainly generated by the format block. Table 4.6 lists the device utilization of the compression part.

TABLE 4.3. COMPARASION OF COMPRESSION RATIO.

Data Sample (Nexus Trace Data)	Compression Ratio (Size of uncompressed data / Size of compressed data)						
	Combinational			Dictionary			Statistical
	Gzip	Bzip2	LZMA	Thesis	LZ4	LZW	Huffman
File1 (1.8MB)	2.0	2.95	4.21	1.4	1.5	1.28	1.29
File2 (2.6MB)	2.0	2.97	4.09	1.9	1.52	1.3	1.3
File3 (3.5MB)	2.5	3.5	4.19	2.5	1.84	1.4	1.34
Average Compression Ratio	2.16	3.14	4.16	1.35	1.62	1.32	1.31

TABLE 4.4. COMPARASION OF CODE STYLE.

Data Sample (Nexus Trace Data)	Proposed code style (*1*,*distance*)	Traditional code style (*1*,*distance*,*length*)
	Compression Ratio	Compression Ratio
File1 (1.8MB)	1.28	1.2
File2 (2.6MB)	1.36	1.24
File3 (3.5MB)	1.4	1.3
Average Compression Ratio	1.35	1.25

TABLE 4.5. FPGA RESOURCE UTILIZATION OF THE COMPRESSOR.

Synthesis configuration		Complexity			FPGA	Maximum Frequency	Throughput
opt_mode	opt_level	RAM (Bytes)	Number of Slice LUTS	Number of Slice Registers			
speed	normal	64K +256 (two-port)	26825	2564	Virtex 7 XC7VX 485T-1	123MHz	16 bits/cc
speed	high	64K +256 (two-port)	24121	2516		120MHz	
area	normal	64K +256 (two-port)	26571	2500		55MHz	
area	high	64K +256 (two-port)	22960	2499		95MHz	

TABLE 4.6. FPGA RESOURCE UTILIZATION OF COMPRESSION PART.

Synthesis configuration		Complexity			FPGA	Maximum Frequency
opt_mode	opt_level	RAM (Bytes)	Number of Slice LUTS	Number of Slice Registers		
Speed	normal	64K +256 (two-port)	5865	2386	Virtex 7 XC7VX 485T-1	158MHz
Speed	high	64K +256 (two-port)	4911	2420		158MHz
Area	normal	64K +256 (two-port)	5899	2308		116MHz
Area	high	64K +256 (two-port)	4859	2308		138MHz

It is hard to directly compare the hardware cost with other hardware implementations reported in literature. Since the used FPGAs are different. In addition, the selected parameters are different also. Here a rough

comparison with other implementations is listed in the Table 4.7. The calculation of Equivalent Gates is according to the Xilinx's 7 series FPGA specifications. According Xilinx's design guide, in this thesis, we assume one LUT equals 15 gates and one register equals 7 gates [30], [31]. This compressor can achieve a higher throughput with hardware complexities compared to previous compression approaches.

TABLE 4.7. COMPARISION HARDWARE COST AND THROUGHPUT.

Algorithm	Processor	Hardware Cost		Throughput (bits/clock)
		RAM size (bytes)	Equivalent Gates	
Dynamic Huffman [20]	ASIC	4.5K (CAM)	17.7K	0.5
LZRW [32]	ASIC	38K	60K	7
Thesis	Virtex 7 (XC7VX485T)	64K+256 (two-port)	89K	16
Gzip[24]	ASIC	143K	610K	8
Bzip2	Stratix II (EP2S180F1020C3)	7.3M	1575K	1.5
LZMA[26]	Virtex 5 XC5VFX70T	212K (dual-port)	120K	4

CHAPTER 5

5. Conclusion and Future Work

5.1. Conclusion

The subject of the dissertation is to implement a real-time lossless Nexus trace data compressor. This thesis investigates the state-of-the-art of lossless algorithms. The background of compression algorithms is presented in Chapter 2. In Chapter 3, the feasibility of different types of lossless algorithms in hardware implementation is examined. The LZ77 algorithm is selected through analyzing the three main performances, compression ratio, compression speed, and hardware overhead. We explore the characteristics of Nexus trace data through simulation and determine the parameters based on the thesis requirements.

An optimized LZ77 compression algorithm is introduced in Chapter 4. There are two modified aspects. The first aspect, the compressed data are output in a pair (*1, distance*) instead of a triple (*1, distance, length*). The match length information is omitted. Because the maximum match length is equal to the minimum match length based on the characteristics of Nexus trace data – the compression ratio perform the best on a short match. The second aspect, the entries of the hash table are set equal to 2^{15} in order to improve the compression speed. Compared with the original LZ77 algorithm, this approach reduces the hash collision probability and the amount of matching iterations. Therefore, the speed of finding the maximum match string becomes faster.

The goal of this thesis is to achieve a data compressor that will meet the performance requirements, while minimizing the implementation cost. Test results have shown that the compression speed is 16 bits/clock cycle, which meet the speed requirement. The average compression ratio is 1.35 with the minimum hardware cost. The compression ratio is improved from 1.25 to 1.35 when the optimized code word style (*1, distance*) is used.

In general, the optimized scheme achieves a trade-off between the hardware cost and the compression performances. The optimized LZ77 algorithm proposed in this dissertation is a cost-effective compressor focusing on Nexus trace data.

5.2. Future Work

There is still a space to improve the compression ratio without increasing the hardware cost hugely.

The first approach is to reduce the minimum move length from 2 bytes into 1 byte. According to the simulation results in section 3.3.2, the compression can be improved about 4% when the minimum move length is decreased from 2 bytes into 1 byte. But the compression time might be affected by this modification.

The second approach is to do pre-processing on the Nexus trace data to make it more suitable for the dictionary based compression algorithms. Such as, for the time part, a differential processing can be used. Thus only the differential time is needed to be kept in the trace messages. Therefore the time part could be represented in a shorter code word. Based on calculation, the compression ratio can be improved approximately 7%.

However, it is hard to increase the compression ratio higher than 2 if only the above two approaches are used. If the application requires the compression ratio greater than 2, a combinational compression algorithm is recommended. However, the hardware cost will increase.

References

[1] C.Kao, S.Huang, and I.Huang, "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Transactions on Circuits and Systems*, vol. 54, no. 3, pp. 530-542, March 2007.

[2] E.Johnson, J.Ha, and M.Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 158-173, February 2001.

[3] V.Uzelac, "Algorithms and Hardware Structures for Real-Time Compression of Program Traces," 2010.

[4] The Nexus 5001Forum Standard for a Global Embedded Processor Debug Interface, IEEE-ISTO, http://www.nexus5001.org/standard

[5] Design Specification for Debug in Ericsson.

[6] M.Morales-Sandoval, "Hardware Architecture for Elliptic Curve Cryptography and Lossless Data Compression," 2004.

[7] PM.Nishad, "A Novel Approach to Reduce Computational Complexity of Multiple Dictionary Lempel Ziv Welch Mdlzw Using Indexed K Nearest Twin Neighbor Ikntn Clustering and Binary Insertion Sort Algorithms," 2015.

[8] J.Ziv and A.Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 65, no. 3, pp. 337-343, May 1977.

[9] J.Ziv and A.Lempel, "A compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. IT-24, no. 5, pp. 530-536, Sep. 1978.

[10] http://ethw.org/History_of_Lossless_Data_Compression_Algorithms

[11] D.Salomon, "Data Compresssion Fourth Edition," British, Springer, ISBN978-1-84628-602-5, 2007.

[12] T.Bonny, "Huffman-based Code Compression Techniques for Embedded Systems," 2009.

[13] T.Kumaki, Y. Kuroda, T.Koide, and H.Mattausch, "CAM-Based Huffman Coding Architecture for Real-Time Applications," *Japan*.

[14] J.Nunez and S.Jones, “Gbit/s Lossless Data Compression Hardware,” *IEEE Transaction on Very Large Integration Systems*, vol. 11, no. 3, pp. 499-510, June 2003.

[15] A.D.Samples, “Mache: No-less trace compression,” *ACM Performance Eval. Rev.*, vol. 17, no. 1, pp. 89-97, May 1989.

[16] A.Milenkovic and M. Milenkovic, “Streamed-Based Trace Compression,” *IEEE Computer Architecture Letters*, vol. 2, 2003.

[17] E.Anis and N.Nicolici “On Using Lossless Compression of Debug Data in Embedded Logic Analysis,” *IEEE International Test Conference*, pp. 1-10, 2007.

[18] M.Burtscher, I.Ganusov, S.Jackson, and N.Sam, “The VPC Trace-Compression Algorithms,” *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1329-1344, November 2005.

[19] V.Uzelac, A.Milenkovic, M.Milenkvic, and M.Burtscher, “Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing,” IEEE Transactions on Computers, vol. 63, no. 4, pp. 1008-1020, April 2014.

[20] L.Liu, J.Wang, J.Lee, and R.Wang, “CAM-Based VLSI Architecture for Dynamic Huffman Coding,” IEEE Transactions on Consumer Electronics, vol. 40, no. 3, pp. 282-289, August 1994.

[21] http://bigocheatsheet.com/

[22] M.EL, A.Salama, and A.Khalil, “Design and Implementation of FPGA-based Systolic Array for LZ Data Compression”.

[23] S.Naqvi, R.Naqvi, R.Riza, and F.Siddiqui, “Optimized RTL Design and Implementation of LZW Algorithm for High Bandwidth Applications,” *PRZEGLĄD ELEKTROTECHNICZNY (Electrical Review)*, Vols. ISSN 0033-2097, p. R. 87 NR, April 2011.

[24] http://inomize.com/index.php/content/index/gzip-hw-accelerator.

[25] S. Arming, R.Fenkhuber, and T.Handl, "Data Compression in Hardware - The Burrows-Wheeler Approach," *IEEE*, 2010.

[26] I.Shcherbakov and N.Wehn, “A Parallel Adaptive Range Coding Compressor: Algorithm, FPGA Prototype, Evaluation,” in the *Data Compression Conference*, pp. 119-128, 2012.

[27] K.Basu and P.Mishra, “Efficient Trace Data Compression using Statically Selected Dictionary,” *IEEE VLSI Test Symposium*, pp. 14-19, 2011.

[28] http://bigocheatsheet.com/.

[29] J.Storer and J.Reif, "Parallel Architecture for high Speed Data Compression," *3rd Symposium on the Frontiers of Massively Parallel computation*, 1990.

[30] http://www.xilinx.com/training/fpga/what-is-the-difference-between-an-fpga-and-an-asic.htm.

[31] http://www.xilinx.com/publications/prod_mktg/low-end-portfolio-product-selection-guide.pdf.

[32] http://www.heliontech.com.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2015-478

http://www.eit.lth.se