ELSEVIER

Signal Processing 69 (1998) 149–155

SIGNAL PROCESSING

# Hierarchical dictionary model and dictionary management policies for data compression

Chia-Lun Yu, Ja-Ling Wu*

*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, 106, Taiwan, ROC*

Received 29 September 1994; received in revised form 20 January 1997

**Abstract**

In this paper, an adaptive multi-dictionary model for data compression is proposed. Dictionary techniques applied in lossless compression coding can be modeled from the dictionary management point of view which is similar to that of cache memory. The behavior of a compression technique can be described by nine parameters defined in the proposed model, which provides a unified framework to describe the behavior of lossless compression techniques including existing probability-based Huffman coding and arithmetic coding, and dictionary-based LZ-family coding and its variants. Those methods can be interpreted as special cases under the proposed model. New compression techniques can be developed by choosing proper management policies in order to meet special encoding/decoding software or hardware requirements, or to achieve better compression performance. 

**Zusammenfassung**

In diesem Artikel wird ein adaptives Wörterbuch-Modell zur Datenkompression vorgestellt. Wörterbuchtechniken, die in verlustloser Kompressionscodierung angewandt werden, können aus der Sicht der Verwaltung des Wörterbuches modelliert werden, die ähnlich zu der von Cache-Speichern ist. Das Verhalten einer Kompressionsmethode kann durch neun Parameter beschrieben werden, die im Rahmen des vorgestellten Modells definiert werden. Diese stellen einen einheitlichen Rahmen zur Beschreibung des Verhaltens verlustloser Kompressionstechniken zur Verfügung, darunter die existierenden wahrscheinlichkeitsbasierende Huffman-Codierung, arithmetische Codierung und wörterbuchbasierende Codierung mit LZ-Familien. Diese Methoden können als Spezialfälle des angegebenen Modells angesehen werden. Neue Kompressionsmethoden können durch angemessene Verwaltungsstrategien entworfen werden, um spezielle Codierungs/Decodierungs-Software benutzen zu können oder Hardwareanforderungen zu erfüllen, oder um einen bessere Kompression zu erreichen. 

**Résumé**

Un modèle multi-dictionnaires adaptatif pour la compression des données est proposé dans cet article. Les techniques par dictionnaire appliquées en codage par compression sans perte peuvent être modélisées du point de vue de la gestion du dictionnaire, qui est similaire à celle d'une mémoire cache. Le comportement de la technique de compression peut être décrit par neuf paramètres définis dans le modèle proposé. Ceci fournit un cadre unifié pour décrire le comportement des

* Corresponding author.

0165-1684/98/$ – see front matter 
PII: S0165-1684(98)00096-6

techniques de compression sans perte incluant le codage de Huffman et le codage arithmétique basés sur les probabilités et le codage de type LZ basé sur un dictionnaire et ses variantes. Ces méthodes peuvent être interprétées comme des cas particuliers du modèle proposé. Des techniques de compression nouvelles peuvent être développées en choisissant des stratégies de gestion appropriées afin de satisfaire des exigences spéciales de logiciel ou de matériel pour l'encodage/décodage, ou pour obtenir de meilleures performances de compression. 

*Keywords:* Huffman coding; Arithmetic coding; LZ coding; Dictionary management

## 1. Introduction

In general, there are two kinds of compression coding schemes: *lossless* and *lossy* compression. Lossless compression requires that original data be reconstructed without any distortion after inverse operation. Usually, texts, financial statistics, or computer execution files are typical examples to which lossless compression is applicable. On the other hand, lossy compression does not guarantee that the original and recovered data are identical, but it often provides better performance (in terms of compression ratio) than lossless methods. Lossy compression can be applied to voice, image, and video media applications because they do not necessarily require perfect recovery if the reconstructed quality is good enough for human perception. However, in this paper, we will focus on the topic of lossless compression.

The major work that compression tries to do is to eliminate the redundancies in source data in order to achieve a more concise representation. Common redundancies, sometimes overlapping to some extent, include character distribution, character repetition, high-usage patterns, and positional redundancy [10]. Many techniques had been developed during the past decades to achieve better compression performance. Well-known algorithms, such as run-length [9], Huffman [5], arithmetic [11], and Lempel–Ziv coding [12,13] are applicable to general lossless coding applications.

One obvious redundancy of many data is the repeated occurrence of substrings or patterns. Techniques that factorize common substrings are known as dictionary techniques. A collection of common substrings could be constructed using dictionary approaches either on the fly or in a separate pass. It may use the same dictionary for all input data sets (static) or construct a different dictionary for each data file (adaptive or semi-adaptive) [1]. Lempel–Ziv coding can be classified as one of the adaptive dictionary techniques. One variant of Lempel–Ziv coding devised by Welch, the LZW coding [10] uses a large string table (dictionary) to store the coding history for the usage of backward reference in order to achieve higher coding efficiency. Each codeword in LZW contains only a fixed-length table entry number. The LZW algorithm performs well on several data types and can be easily implemented using either hardware or software with reasonable cost.

There is interesting analogy between cache memory [4,6] and LZW coding in their internal data structure. Both cache and LZW use table structure in their data manipulation. We proposed a new coding model for dictionary techniques from the viewpoint of the management policy of cache memory's hierarchical organization. It was borrowed from the fast access feature of cache memory and then applied to data compression in order to achieve better compression result. In the following sections we will describe a new coding model for dictionary techniques with adaptive multiple dictionaries. An example is given in Section 5 to show the capability of the proposed model in performance improvement of LZW and to discuss how parameters defined in that model affect the overall compression performance.

## 2. Survey of the Lempel–Ziv coding

Since LZW is chosen to be the competitor of the proposed approach, a brief review of the Lempel–Ziv coding is given in this section.

Lempel–Ziv data compression algorithms are a family of methods that compress source data into a compact form using an incremental parsing procedure. Various modifications of that fundamental scheme had been developed during recent years. Conceptually, the Lempel–Ziv-type scheme treats the process of compression as generating a special 'program' or 'instruction' that interprets input data sequence into serial string copying commands [8]. The source data can be exactly recovered on receiving end by executing such copying commands in order. It also provides a new aspect of complexity of a given sequence which differs from traditional Shannon entropy $H$. Lempel–Ziv algorithms provide the capability of grouping several source symbols to one 'meta-symbol' so that the total number of redefined 'symbols' can be reduced.

LZ77 algorithm [13] partitions the input symbols into substrings (or 'words'), then encodes each substring with a fixed-length codeword. There are one sliding window recording the history of processed symbols and one lookahead buffer containing current input data. The LZ77 algorithm tries to find a longest substring in the window that matches the string in the lookahead buffer. A fixed-length (*position*, *length*, *new symbol*) tuple is coded and transmitted, indicating the starting position and the length of the substring found in the window. After the tuple is determined, LZ77 updates the contents in the window and lookahead buffer by shifting length symbols out before next string matching process.

LZ78 algorithm [12] is different from LZ77 in the presentation of coding history. Instead of using a sliding window, LZ78 uses a string table with infinite size to store the substrings determined during coding process. Each substring is identified by a unique table entry number. Whenever LZ78 finds the longest matched substring, tuple (*entry number*, *new symbol*) is coded as codeword. A new substring generated by concatenating the first unmatched symbol in lookahead buffer with the matched substring is then inserted into the string table, which is expected to appear in the data sooner or later. As the number of registered substrings augmented, bits required to represent all possible entry numbers increased logarithmically. Therefore, the codeword length of LZ78 scheme is variable.

LZW is an implementation of LZ78 with finite table size and fixed codeword length (usually 12 bits, 4096 table entries). The major difference between LZW and LZ78 is that the first unmatched symbol (new character) is not included in the current transmitted codeword of LZW, which is just used to build new candidate substring that we guess may appear in the near future.

## 3. Cache memory

Cache memories are high-speed memory buffers which are inserted between the processors and main memory to capture those portions of the contents of main memory which are currently in use. The memory hierarchy usually consisted of many levels, but it is managed between two adjacent levels at a time.

The *upper* level (cache) – the one closer to the processor – is smaller and faster than the *lower* level (main memory). Success or failure of an access to the upper level is designated as a hit or a miss: a *hit* is a memory access found in the upper level, while a *miss* means it is not found in the level. *Hit time* is the time to access the upper level in the memory hierarchy, which includes the time to determine whether the access is a hit or a miss. *Miss penalty* is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the requesting device (normally the CPU). Usually, the most important measure of cache performance is the average access time of memory:

$$\text{Average memory-access time} = \text{Hit-time} + \text{Miss rate} \times \text{Miss penalty}.$$

### 3.1. Design aspects

In the following, some common questions encountered in the design of cache memory and some well-known management policies are listed:

- Where can a block be placed in a cache? (*Block placement*)
  Policy: Direct mapped, Fully associative, Set associate.

- How is a block found if it is in the cache? (*Block identification*)
- Which block should be replaced on a cache miss? (*Block replacement*)
  Policy: Random, Least-recently used (LRC), First-in-first-out (FIFO).
- What happens on a write? (*Write strategy*)
  Policy: Write through, Write back, Write allocate, No write allocate.

Detail definitions and descriptions of the items used in this subsection can be found in [14] and [15].

### 3.2. Compression versus cache

The idea of fast access in cache can be applied to data compression. In order to get better compression performance, the substrings that occur more frequently should be assigned with shorter codeword length. If we collect frequently occurred substrings (patterns) in a small cache-like dictionary and encode these patterns with fewer bits, and other rarely occurring substrings in another large dictionary with longer codewords, the overall compression performance should be improved.

Policies that maintain dictionary contents can be adopted from those used in cache management. For example, when dictionary is full, several replacement strategies such as FIFO or LRU used in cache can also be applied to data compression. Another example is the placement policy of cache; it can be modified to be used as a hash function to decide where should a substring be placed in dictionary. Possible relationships between dictionary-based coding and cache memory are illustrated in Table 1.

Table 1
Relationships between cache and compression coding

| Cache | | Compression coding |
|---|---|---|
| Cache | ↔ | Small dictionary, short codeword |
| Main memory | ↔ | Large dictionary, long codeword |
| Block placement | ↔ | Hash function, Code tree |
| Block replacement | ↔ | Adaptive substring movement between dictionaries |
| Write strategy | ↔ | Add new substring to lower level dictionary |

## 4. Adaptive multi-dictionary model (AMDM)

In this section, we describe the proposed model for dictionary-based coding. The initial idea was derived from [16]. An adaptive multi-dictionary model (AMDM) can be modeled by a 9-tuple:

$$\mathrm{AMDM} = \{N,S,\mathrm{GP},\mathrm{CR},\mathrm{FR},\mathrm{PP},\mathrm{RP},\mathrm{UP},\mathrm{AP}\}.$$

The following parameters characterize the behavior of an adaptive dictionary-based coding system:

$N$: The number of dictionaries (the level of dictionary hierarchies).

$S_d$: (Size)
The sizes of the dictionaries,
We use the notation s($d$), $d = 1,2,\ldots,N$, to denote the size of the $d$th dictionary. For example, $s(1)$ is the size of the first dictionary and $s(3)$ is that of the third dictionary. We can choose different sizes for different dictionaries.

$\mathrm{GP}_d$: (Generate policy)
The policy for generating new words to extend contents of the $d$th ($d = 1,2,\ldots,N$) dictionary. Most probability-based techniques do not create new words during processing. In general, the predefined words are just the source symbols. The contents of the dictionary is fixed. On the other hand, Lempel–Ziv algorithms and BSTW method [2] create new words from the characteristics of input data. It is interesting to demonstrate the policy chosen for GP that LZ-78 and LZW concatenate first unmatched symbol with the found word to form a new word hence the vocabulary of the dictionary is augmented in each step.

$\mathrm{CR}_d$: (Codeword representation)
The codeword patterns of the $d$th ($d = 1,2,\ldots,N$) dictionary.
This set of parameters is determined by a function which maps the words defined in the dictionary to the corresponding codeword patterns. The mapping is not restricted to one-to-one mapping. Sometimes it may be a many-to-one mapping. We use a function

*codeword* $(d,e,t)$ to denote the codeword patterns. Since the mapping is with a time-dependent property, the 'time' becomes an important factor in the definition of the above function. The meaning of *codeword* $(d,e,t)$ is the codeword pattern for the $e$th entry of the $d$th dictionary at step $t$. The parameter $AP_d$ (described later) also affects the $CR_d$ significantly because usually the value (pattern) of *codeword* $(d,e,t)$ is obtained from the policy specified by $AP_d$.

$FR_d$: (Flagbits representation)
The leading flagbit patterns of the $d$th $(d = 1,2,\ldots,N)$ dictionary.
Function *identifier* $(d,t)$ is used to represent the flagbit pattern designed to indicate which dictionary the found word belongs to. We can send the pattern at each step to point out explicitly which dictionary is in use or we can transmit it only when a transition of currently used dictionary occurs. The failure symbol approach used in multi-group Huffman coding is a good example to illustrate the function of parameter $FR_d$.

$PP_d$: (Placement policy)
Where to place the new generated words in the $d$th $(d = 1,2,\ldots,N)$ dictionary?
Usually we number and place the new word in ascending order. But it is possible for us to apply additional techniques such as hash functions to the decision of placement in order to accelerate the speed of searching or string matching. Particular placement policies may be helpful to enhance the desired performance. For example, we can either choose the root or some particular leaf node to be the location where the new word be placed if tree structure is adopted. In BSTW coding, this parameter can be described as 'put the new word in the front of list'.

$RP_d$: (Replacement policy)
When the $d$th $(d = 1,2,\ldots,N)$ dictionary fills and a miss occurs, which one of the existing words in this dictionary should be removed to leave room for a new word?
The problem we face is the overflow of dictionary contents. In practical applications, input data are usually large enough so that the number of generated new words exceeds the capacity of dictionaries. There are many heuristics available to solve the problem. Bunton [3] surveyed several methods including FIFO, FREEZE, LRU, SWAP and TAG. Good policy for this parameter can help encoder to capture the locality of input source. At the same time, it may have a great influence on the hardware implementations.

$UP_d$: (Update policy)
How and when to update the contents of other dictionaries if the contents of the $d$th $(d = 1,2,\ldots,N)$ dictionary have been modified?
Adaptive contents exchange capability between dictionaries is provided in the model. Parameter $UP_d$ describes the policy chosen to move vocabulary between different dictionaries to achieve better performance.

$AP_d$: (Adjustment policy)
How to adjust the codeword representation in the $d$th $(d = 1,2,\ldots,N)$ dictionary, if need, after each coding step?
After each encoding step, the codeword mapping may be altered. The principle of how to perform the modification is defined by parameter AP.
For instance, dynamic Huffman coding reshapes the Huffman tree to decide the new mapping for next encoding. The definition of 'reshape' is described by AP. Usually, the decision is based on past coding history. Therefore, codeword$(d,e,t)$ may be dependent on codeword$(d,e,t-1)$, codeword $(k,e,t-1),\ldots$, etc.
Now, we define a new function hit$(d,e,t)$. If at step $t$, the matched word is the $e$th entry of $d$th dictionary then the value of hit$(d,e,t)$ is set to 1, otherwise is 0.

$$\text{hit}(k,i,t) = \begin{cases} 1 & \text{dictionary } d,\ \text{entry } e, \\ & \text{step } t \text{ is active}, \\ 0 & \text{otherwise.} \end{cases}$$

Assume the number of total encoding steps for a particular input source is denoted as variable $k$. The length of pattern is represented by |codeword$(d,e,t)$| and |identifier$(d,t)$|, respectively. The

total length of output code can be expressed by the following equation:

$$\sum_{t=1}^{k}\left(\sum_{d=1}^{N}(|\text{identifier}(d,t)| + \sum_{e=1}^{s(d)}(|\text{codeword}(d,e,t)|^{*}\text{hit}(d,e,t)))\right).$$

That means, in each step $t$, the total length of output is contributed by two bit patterns, identifier($d$,$t$) and codeword($d$,$e$,$t$). For all possible ($d$,$e$) pairs, hit($d$,$e$,$t$) chooses the correct one and disables wrong choices by multiplying them by zero.

## 5. Design of new techniques

By choosing proper management policies defined in the adaptive dictionary model, it is convenient to design new coding algorithms or to improve current techniques. In this section, we propose a new algorithm to improve LZW by using two dictionaries. There are two dictionaries kept during the coding process. The first dictionary is a small one that contains only 128 entries; substrings stored in this dictionary are organized as a splay tree [7]. Codeword length in the first dictionary is variable. The second dictionary, larger than the first one, contains 2048 entries and uses fixed-length codewords (11 bits).

Initially the 256 single-character substrings are filled in both the first and the second dictionaries to guarantee the success of string matching. The main encoding process begins with finding the longest matched substring in the first dictionary. When a successful matching happens in the first dictionary, transmit one '0' as a flag bit to indicate the decoder that the following bits should be recognized according to the path of splay tree. Then adjust the tree by executing semi-splay operation. The reason for such adjustment is that the matched substring may occur again soon; shortening the code length of that substring will benefit the compression efficiency.

If a miss occurs, transmit one '1' bit followed by 11 bits of information to inform the decoder to interpret the following 11 bits as the entry number of the second dictionary. The substring found in the second dictionary is inserted into the first one. Newly built predictive substring is inserted into both the first and the second dictionaries in each encoding step. Replacement policy used in this algorithm is to choose the substring with longest path in codetree to be removed when the first dictionary fills. Substrings will be removed from the first dictionary after a few encoding steps if they are not used too frequently. Most substrings with high occurring frequency should be collected in the first dictionary and be encoded with codewords as short as possible to get salient compression efficiency.

```
Init :   Fill 256 single-character substrings into both dictionaries.
Encode:  While (input not exhausted) do {
             Find longest matched substring in the first dictionary.
             If (successful matching) {
                 send a '0' flag bit
                 send variable-length codeword
                 semi-splaying
                 generate new predictive substring to both dictionaries
             }
             Else {
                 find longest matched substring in second dictionary
                 send a '1' flag bit
                 send fixed-length codeword (11 bits)
                 insert the found substring to the first dictionary
                 generate new predictive substring to both dictionaries
                 }
             }
```

Fig. 1. Algorithm for two-level dictionary LZW.

Table 2
Parameter description of two-level LZW coding

| Parameter | Description |
|---|---|
| $N$ | 2 |
| $GP_1$, $GP_2$ | Concatenate unmatched symbol to matched word |
| $S_1$ | 128 |
| $S_2$ | 2048 |
| $CR_1$ | Splay tree |
| $CR_2$ | Fixed-length entry number (11 bits) |
| $PP_1$ | Bottom level of splay tree |
| $PP_2$ | Ascending numbering |
| $RP_1$ | The word with longest codeword length |
| $RP_2$ | FIFO or LRU |
| $UP_1$, $UP_2$ | (Null) |
| $AP_1$ | Semi-splaying |
| $AP_2$ | (Null) |

Table 3
Compression performances

| File type | File size (bytes) | LZW | ratio (%) | Two-level LZW | ratio (%) |
|---|---|---|---|---|---|
| EXE 1 | 1018928 | 1021984 | 100.3 | 957792 | 94.0 |
| EXE 2 | 113391 | 95475 | 84.2 | 91394 | 80.6 |
| TEXT 1 | 110510 | 51056 | 46.2 | 47298 | 42.8 |
| TEXT 2 | 126569 | 45058 | 35.6 | 39742 | 31.4 |
| IMAGE 1 | 153718 | 19984 | 13.0 | 17216 | 11.2 |
| IMAGE 2 | 672000 | 139776 | 20.8 | 05504 | 15.7 |

Note: $\text{ratio} = \frac{\text{compressed size}}{\text{original size}} * 100\%$
EXE 1: Binary execution file 1.
EXE 2: Binary execution file 2.
TEXT 1: English text file of software installation and documentation.
TEXT 2: LATEX typesetting file of part of a master's thesis which contains Chinese and English text.
IMAGE 1: The CHESS.BMP image file in Microsoft Windows 3.0 package.
IMAGE 2: A color F-16 fighter picture with 24-bit per pixel format.

Both the encoder and the decoder operate synchronously so that there is no need to transmit whole dictionary contents explicitly. Fig. 1 shows the pseudocodes of the algorithm.

The two-level LZW algorithm is characterized in Table 2 by the parameters of the dictionary model. Experimental results of the performance of the new variant compared with the original LZW algorithm are listed in Table 3 and show that the proposed method achieves better compression efficiency.

## References

[1] T.C. Bell, J.G. Cleary, I.H. Witten, Text Compression, Prentice-Hall, Englewood Cliffs, NJ, 1990.
[2] J.L. Bentley, D.D. Sleator, R.E. Tarjan, V.K. Wei, A locally adaptive data compression scheme, Commun. ACM 29 (4) (April 1986) 320–330.
[3] S. Bunton, G. Borriello, Practical dictionary management for hardware data compression, Commun. ACM 35 (1) (January 1992) 95–104.
[4] J.L. Hennessey, D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufman, San Mateo. CA, 1990.
[5] D.A. Huffman, A method for the construction of minimum redundancy codes, Proc. IRE 40 (9) (September 1952) 1098–1101.
[6] K. Hwang, F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, 1984.
[7] D.W. Jones, Application of splay trees to data compression, Commun. ACM 31 (8) (August 1988) 996–1007.
[8] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inform. Theory IT-22 (1) (January 1976) 75–81.
[9] H. Tanaka, A. Leon-Garcia, Efficient run-length encoding, IEEE Trans. Inform. Theory IT-28 (6) (September 1952) 880–890.
[10] T.A. Welch, A technique for high performance data compression, IEEE Comput. 17 (6) (June 1984) 8–19.
[11] I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic coding for data compression, Commun. ACM 30 (6) (June 1987) 520–540.
[12] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory IT-24 (5) (September 1978) 530–536.
[13] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory IT-23 (3) (May 1977) 337–343.
[14] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufman, San Mateo, CA, 1990.
[15] K. Hwang, F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, 1984.
[16] C.L. Yu and Ja-Ling Wu, Modeling adaptive multilevel-dictionary coding scheme by cache memory management policy, in: Proc. 1992 Data Compression Conf. (DCC'92), Snowbird, UT, March 1992.