International Symposium on Information Theory and its Applications, ISITA2006
Seoul, Korea, October 29–November 1, 2006

# Improving LZ77 Data Compression using Bit Recycling

Danny DUBÉ[†] and Vincent BEAUDOIN[‡]

Département d'informatique et de génie logiciel
Université Laval
(Québec) G1K 7P4, Canada
E-mail: `Danny.Dube@ift.ulaval.ca`[†] `Vincent.Beaudoin.1@ulaval.ca`[‡]

**Abstract**

Many data compression techniques allow for more than one way to encode the compressed form of data. In particular, this is the case for the LZ77 technique and its derivatives, where matches can often be described in more than one way. The very existence of multiple encodings for the same data acts as a side-channel through which additional information can be conveyed from the compressor to the decompressor, and so, for free. We show that this side-channel can be used to carry parts of the compressed file itself, thereby shortening the latter, and improving compression ratios. We call our technique *bit recycling* and show how it applies to Huffman encoding. We present it as a way to improve LZ77 compression and demonstrate it through many experiments. One of these experiments has already been presented but the new ones take better advantage of recycling. The experiments show that we can improve compression efficiency significantly and tend to point to a method with which bit recycling is most profitable.

## 1. Introduction

Different compression techniques typically perform differently on particular data. Often, the differences in the performances are explained by saying that some technique *models* data more accurately than another one. However, the fitness of the model to the data is not necessarily the only factor that influences compression efficiency. A technique tends to lose some efficiency by allowing particular data to have numerous different encodings. Indeed, the more there exist ways to encode the same data, the lengthier these encodings tend to be, thus having an adverse effect on compression efficiency. Fortunately, we show that the multiplicity of encodings exhibited by a compression technique need not necessarily ruin its competitiveness. Part of the loss in compression efficiency can be recovered through what we call *bit recycling*.

Our research on recycling is applied to the LZ77 compression technique [13]. LZ77 compression is a well known lossless technique and it is widely used in popular tools. In particular, the Deflate method that is included in `gzip`, among others, is an efficient derivative of LZ77 compression [3, 5]. Of particular interest to us, LZ77 compression has a very high tendency to allow for multiple encodings of the same data.

The rest of the paper is organized as follows. Section 2 presents a few basic definitions. Section 3 highlights the essential concepts of conventional LZ77 compression. Section 4 gives an intuitive idea of how the compressor and the decompressor may exchange more information without transmitting more bits. Section 5 presents bit recycling. Section 6 presents the raw experimental results and Section 7 gives details about the six experiments and comments on the results they provide.

## 2. Background

A *coding function* $f : S \to \{0,1\}^*$ translates any symbol taken from set $S$ into a sequence of bits. We sometimes refer to coding functions as *codes* and to $f(s)$ as the *codeword* for symbol $s$.

On some occasions, we will use function Huffman : $(S \to \mathbb{R}^{>0}) \to (S \to \{0,1\}^*)$ as the conventional code construction technique proposed by Huffman [6] that takes symbols along with their respective occurrence counts and that returns an optimal encoding function for the symbols. Note that the "occurrence counts" need not really be counts. They could be scaled by any strictly positive factor and the same code would still be built by 'Huffman'.

We will also use function 'flat' as a function that builds codes that assign encodings of (almost) equal lengths to all symbols. Although it could be defined directly and more efficiently, we prefer to define it simply as:

$$\mathrm{flat}(\{s_1,\ldots,s_k\}) = \mathrm{Huffman}(\{(s_1,1),\ldots,(s_k,1)\}).$$

Compressor

```
1.  while not end of input do
2.      ⟨l, d⟩ := find best match;
3.      if $l < l_{\min}$ then
4.          emit $C_1$([next input byte]);
5.      else
6.          emit $C_1(\langle l, \cdot \rangle)$;
7.          emit $C_2(d)$;
where
8.  procedure emit w:
9.      $\sigma := \sigma \cdot w$;
10.     return;
11.
```

Decompressor

```
1.  while end not signaled do
2.      msg := receive $C_1$;
3.      if msg = [c] then
4.          store c;
5.      else let $\langle l, \cdot \rangle$ = msg
6.          d := receive $C_2$;
7.          store copy of ⟨l, d⟩;
where
8.  procedure receive C:
9.      let msg, σ′ s.t. $C(msg) \cdot \sigma' = \sigma$;
10.     $\sigma := \sigma'$;
11.     return msg;
```

Figure 1: Essentials of the LZ77 compressor and decompressor.

## 3. Conventional LZ77 Compression

Figure 1 presents an abstract version of the compressor and decompressor algorithms of the LZ77 technique. In its simplest form, the LZ77 technique consists in a serial description of some original file using a *sequence* of *messages*, where each message is either a *literal byte* $[c]$ or a *match* $\langle l, d \rangle$. Message $[c]$ explicitly tells the decompressor that the next byte of the original file is $c$. Message $\langle l, d \rangle$ tells the decompressor that the next $l$ bytes of the original file are exact copies of the already described bytes located $d$ bytes before the current position.

Let us briefly comment on the algorithms. First of all, the transmission of a message proceeds in one or two steps: the first step is used to inform the decompressor either that the message is one particular literal byte ($[c]$) or that the message is a match of a particular length ($\langle l, \cdot \rangle$); in the case of a match, a second step is used to indicate the distance $d$ to that match. Second, we allow the use of Huffman encoding (or some other encoding) through the use of *coding functions* $C_1$ and $C_2$, one for each step. The coding functions allow a particular LZ77 compression technique to optimize compression. Third, we explicitly refer to the bit stream that forms the communication channel between the compressor and the decompressor as variable $\sigma$.

On the other hand, things that both algorithms of Figure 1 do not mention include: the input operations performed by the compressor; the output operations performed by the decompressor; the match-searching method; buffering issues; and additional control-related messages communicated through the bit stream, such as signals for the end of data or the transmission of new coding functions $C_1$ and $C_2$. While these things are important in the design of an efficient compression technique, they can be considered orthogonal to the matter at hand. We simply ignore them throughout the paper.

Finally, there remains the question of what is considered to be a *best match*. Since compression is obtained mostly through the use of matches, it is natural to favor matches that are as long as possible. In this work, we also choose to seek longest matches. However, note that this choice *does not* lead to a well-defined notion of best match, in general. What many techniques choose to select are the *closest* longest matches (a choice that *does* lead to a well-defined notion of best match). The rationale behind this choice is that favoring shorter distances tends to skew the statistics on distances and the bias can then be exploited by the coding function $C_2$. In this work, we choose *not* to do the same.

## 4. Multiplicity of Encodings and Side-Channels

Instead of systematically selecting the closest longest match, we exploit the multiplicity of longest matches to do a *directly meaningful choice* among the various distances. By having the compressor to choose carefully among multiple distances and the decompressor to notice these choices, implicit transmission of information becomes possible through what can be seen as a side-channel. But let us present these ideas more clearly.

Generally speaking, multiplicity of encodings and the implicit communication that derives from it can be explained as follows. Let us consider the compressed file abstractly as a sequence of messages $m_1, \ldots, m_n$ Now, suppose that for many of these messages, there are one or more alternatives, i.e. *different* messages that refer to different copies of the same data. Let us denote by $\{m_{i,1}, \ldots, m_{i,k_i}\}$ the set of alternatives for message $m_i$, including $m_i$ itself. With the alternatives so expressed, it becomes clear that a compressed file is any sequence $m_{1,j_1}, \ldots, m_{n,j_n}$ such that $\forall\ 1 \leq i \leq n.\ 1 \leq j_i \leq k_i$. Since there are $\prod_{i=1}^{n} k_i$ different

```
1.  while not end of input do
2.     $\langle l, \{d_1, \ldots, d_k\} \rangle$ := find best matches;
3.     if $l < l_{\min}$ then
4.        emit $C_1([next\ input\ byte])$;
5.     else
6.        emit $C_1(\langle l, \cdot \rangle)$;
7.        ND-let $i \in \{1, \ldots, k\}$;
8.        emit $C_2(d_i)$;
9.        recycle $R(\{d_1, \ldots, d_k\}, C_2)(i)$;
where
10. procedure emit $w$:
11.    if $w = \epsilon$ or $\rho = \epsilon$ then
12.       $\sigma := \sigma \cdot w$;
13.    else if $w = b \cdot w'$ and $\rho = b \cdot \rho'$
14.          /* where $b \in \{0, 1\}$ */ then
15.       $\rho := \rho'$;
16.       emit $w'$;
17.    else
18.       error;
19.    return;
20. procedure recycle $w$:
21.    $\rho := \rho \cdot w$;
22.    return;
```

Compressor

```
1.  while end not signaled do
2.     $msg$ := receive $C_1$;
3.     if $msg = [c]$ then
4.        store $c$;
5.     else let $\langle l, \cdot \rangle = msg$
6.        $d$ := receive $C_2$;
7.        store copy of $\langle l, d \rangle$;
8.        $\langle l, \{d_1, \ldots, d_k\} \rangle$ :=
9.           find copies of $\langle l, d \rangle$;
10.       let $i \in \{1, \ldots, k\}$ s.t. $d_i = d$;
11.       recycle $R(\{d_1, \ldots, d_k\}, C_2)(i)$;
where
12. procedure receive $C$:
13.    let $msg, \sigma'$ s.t. $C(msg) \cdot \sigma' = \sigma$;
14.    $\sigma := \sigma'$;
15.    return $msg$;
16.
17.
18.
19.
20. procedure recycle $w$:
21.    $\sigma := w \cdot \sigma$;
22.    return;
```

Decompressor

Figure 2: Essentials of the recycling compressor and decompressor.

but equivalent compressed files, choosing one of them carries $\sum_{i=1}^{n} \log_2(k_i)$ bits of information. This is the amount of information that the side-channel is able to transport. Although we just measured the capacity of the side-channel by considering the compressed file as a whole, transportation capacity is generated on a per-message basis.

The idea of a side-channel has been presented and used on many occasions. It has been used in a tool that directly allows one to hide a secret file inside of a (sufficiently large) compressed file [2]. More specific applications of the hidden information include *authentication* of compressed documents [1] and embedding of *error-correction codes* inside of the compressed documents to make them resilient in case of transmission or storage errors [8, 9, 7, 12]. All these techniques exploit the multiplicity of the longest matches except for the first tool which instead chooses to occasionally shorten a match by one byte. Note that, in the latter case, the exploitation of the multiplicity does not exactly fit our model of "sequences of messages with equivalent alternatives". All the techniques aim at performing information hiding (or an application that uses it) and at keeping the format of compressed files unchanged so that a regular decompressor can still rebuild the original files without ever noticing that additional information was present. In general, *steganography* (the more exact term for information hiding) can also be accomplished using other means than the selection of encodings for compressed files and can be used in many other applications. We refer the reader to a survey on steganography by Petitcolas *et al.* [10].

As a different application, we propose to use the side-channel to reduce the size of compressed documents. The idea is to remove some bits from the compressed document and to send them through the side-channel instead. Note that, by doing so, we resolutely change the format of compressed files and only adapted decompressors may now be able to rebuild the original files. This application extends work presented in 2005 in a paper by the authors [4]. That paper presented the principle of recycling applied to LZ77 compression along with the results of an experiment performed using a modification of the `gzip` tool. That experiment is the one we call Experiment 1, here. Five additional experiments have been conducted since.

## 5. LZ77 Compression with Recycling

In Figure 2, we present modified compression and decompression algorithms that perform bit recycling. The two main differences are that potentially *many* best matches are considered and that extra machinery takes care of recycling. Recycling is more easily explained by inspecting the behavior of the decompressor. Once the decompressor has learned exactly what match $\langle l, d \rangle$ has been transmitted (line 6), it finds out about the bytes that it describes (line 7), so it can determine the whole set of distances $\{d_1, \ldots, d_k\}$

| **Name** | **Original** | **Gzip** | **Exp. 1** | **Exp. 2** | **Exp. 3** | **Exp. 4** | **Exp. 5** | **Exp. 6** |
|---|---|---|---|---|---|---|---|---|
| `bib` | 111 261 | 34 900 | 34 305 | 34 445 | 34 049 | 33 859 | 33 943 | 33 863 |
| `book1` | 768 771 | 312 281 | 301 955 | 308 261 | 303 925 | 301 695 | 303 092 | 301 888 |
| `book2` | 610 856 | 206 158 | 202 430 | 203 369 | 201 592 | 200 547 | 200 456 | 200 165 |
| `geo` | 102 400 | 68 414 | 66 542 | 68 020 | 67 286 | 66 347 | 66 343 | 65 889 |
| `news` | 377 109 | 144 400 | 142 808 | 142 395 | 141 181 | 140 709 | 140 565 | 140 418 |
| `obj1` | 21 504 | 10 320 | 11 147 | 10 366 | 10 428 | 10 405 | 10 315 | 10 314 |
| `obj2` | 246 814 | 81 087 | 84 289 | 79 782 | 80 043 | 79 825 | 79 208 | 79 179 |
| `paper1` | 53 161 | 18 543 | 18 783 | 18 382 | 18 360 | 18 232 | 18 180 | 18 147 |
| `paper2` | 82 199 | 29 667 | 29 401 | 29 354 | 29 157 | 29 027 | 28 977 | 28 933 |
| `paper3` | 46 526 | 18 074 | 18 198 | 17 916 | 17 860 | 17 764 | 17 717 | 17 689 |
| `paper4` | 13 286 | 5 534 | 5 986 | 5 500 | 5 482 | 5 460 | 5 446 | 5 440 |
| `paper5` | 11 954 | 4 995 | 5 467 | 4 959 | 4 951 | 4 935 | 4 917 | 4 922 |
| `paper6` | 38 105 | 13 213 | 13 783 | 13 191 | 13 150 | 13 124 | 13 048 | 13 055 |
| `pic` | 513 216 | 52 381 | 54 257 | 52 023 | 52 303 | 52 040 | 51 742 | 51 621 |
| `progc` | 39 611 | 13 261 | 13 852 | 13 104 | 13 082 | 13 176 | 12 974 | 13 097 |
| `progl` | 71 646 | 16 164 | 16 653 | 15 942 | 15 923 | 15 848 | 15 782 | 15 771 |
| `progp` | 49 379 | 11 186 | 11 787 | 11 047 | 11 104 | 11 041 | 10 957 | 10 943 |
| `trans` | 93 695 | 18 862 | 19 340 | 18 658 | 18 641 | 18 518 | 18 490 | 18 454 |

Figure 3: Sizes of the benchmark files when left uncompressed, when compressed using `gzip` set to maximal compression, and when compressed using each of the six prototypes.

among which the compressor could have chosen (line 8), and so it is able to identify the option number $i$ such that $d = d_i$ (line 10). Finally, the recycled bit string $R(\{d_1, \ldots, d_k\}, C_2)(i)$ is added in front of the input bit stream $\sigma$ (line 11). $R$ is the recycling function. It first takes a set of distances and the coding function for distances. Then, $R(\{d_1, \ldots, d_k\}, C_2)$ acts as a code for numbers in $1, \ldots, k$. After augmenting the bit stream, the decompressor proceeds with the reception of the next message. Note that the latter may well be decoded from a mixture of recycled and explicitly transmitted bits.

On the compressor side, unfortunately, things do not go as smoothly. The problem is that the compressor has to choose what distance it should use (line 7), which has an effect on the bits that get recycled by the decompressor, which will contribute on the encoding of the next messages. However, when it chooses, the compressor does not yet know what messages it will transmit next. Consequently, the algorithm uses a non-deterministic choice. The correctness of this choice is later checked by the '**emit**' procedure. An auxiliary bit stream $\rho$ is used to hold the "predictions" of the compressor about the future. Conceptually, the bits put in $\rho$ are those that are implicitly sent through the side-channel instead of explicitly through $\sigma$.

Let us examine the effect of the length of the recycled codewords on the choices made by the compressor. If one inspects the workings of the recycling compression algorithm, one soon realizes that the probability that some option $i$ gets chosen depends on the length of the corresponding recycled bit sequence. The longer the recycled bit sequence, the smaller the chances for option $i$. Equivalently, the shorter the recycled bit sequence, the higher the chances for option $i$. If one makes the reasonable assumption that the compressed bit stream is a random-like sequence, then option $i$, whose corresponding recycled bit sequence $w_i$ is $|w_i|$-bit long, has probability $2^{-|w_i|}$ of being chosen. It means that, by assigning recycled codewords of same length to all options, any distance to a best match has an equal chance of being chosen, no matter the cost of encoding it.[1] In order to introduce some kind of penalty for the distances that are more costly, these should be assigned longer recycled bit sequences. Although the assignment of a longer recycled bit sequence may at first glance look more like a reward than a penalty, the "rewarded" option inevitably suffers the consequence of being chosen more rarely.

## 6. Experimental Results

Before we describe each experiment in detail, we invite the reader to take a look at the experimental results and we make some general comments. Figure 3 presents the measurements obtained by compressing each benchmark file using `gzip` set at maximal compression and using each prototype. The files are those found in the Calgary Corpus [11], which is commonly used as a benchmark. Each prototype has been implemented as a compressor/decompressor pair to make sure that the original files could always be recovered

[1]For the sake of simplicity, we ignore the issue of the lengths of the encodings varying by 1 when the number of symbols is not a power of two.

correctly and that our measurements are not the deceptively happy consequence of a buggy compressor.

Due to the need to manipulate sets of distances, the prototypes are slower than `gzip`. The slowdown is roughly the same for all prototypes. Compressing the whole set of benchmark files takes about twice as long using one of the prototypes than using `gzip`.

## 7. Description of the Prototypes

Figure 4 presents a (pretty terse) summary of the six prototypes used in our experiments. In order to interpret its contents, some tools and notions have to be introduced first.

Three elements vary among the prototypes: the function $C_2$ that encodes distances, the notion of best matches, and the recycling function $R$. The other elements of the compression and decompression algorithms do not change; in particular, function $C_1$. The possible values of each of three changing elements are presented below.

We need to briefly present the method normally used by `gzip` to build its function $C_2$. The Deflate algorithm used in `gzip` performs compression block by block. It first *parses* the contents of a block into a sequence of not-yet-encoded messages, where some are literal bytes and the others are closest longest matches. `Gzip` determines $C_1$ based on the set of collected literal bytes and the set of collected lengths and, more important for us, it determines $C_2$ based on the set of collected distances. When some distances do not appear into the latter, `gzip` simply *does not* provide an encoding for them, i.e. $C_2(d)$ is undefined for such a distance $d$. In the context of recycling, where potentially any longest match may be referred to, such incompleteness in the definition of $C_2$ reduces the number of options.

There are three different codes for distances that are used by the prototypes. The first one, $C_2^{\text{flat}}$, assigns flat encodings to the distances and it is defined for all distances, i.e.:

$$C_2^{\text{flat}} = \text{flat}(\{1, \ldots, d_{\max}\}).$$

The second one, $C_2^{\text{gzip}}$, is the usual encoding function for distances chosen by `gzip`. The third one, $C_2^{\text{full}}$, is similar to $C_2^{\text{gzip}}$ except that it is defined everywhere. It is obtained by tweaking the statistics collected by `gzip`'s parse by levying a "tax" on the occurrence counts of the popular distances and redistributing it in order to artificially increase the counts for the never-occurring distances.

There are three definitions of "best matches" that are used by the prototypes. All three definitions agree on the fact that only longest matches are acceptable but they disagree on what is considered to be an *eligible* distance to a longest match. The first definition considers any distance to be eligible. The second one only considers as eligible a distance that $C_2^{\text{gzip}}$ can encode. The third one is even more restrictive: a distance is considered as eligible if $C_2^{\text{gzip}}$ can encode it and its encoding is no longer than that of the distance to the closest longest match.

| **Exp. #** | **$C_2$** | **Eligible distances** | **R** |
|---|---|---|---|
| 1 | $C_2^{\text{flat}}$ | all distances | $R^{\text{flat}}$ |
| 2 | $C_2^{\text{gzip}}$ | "short" distances | $R^{\text{flat}}$ |
| 3 | $C_2^{\text{gzip}}$ | encodable distances | $R^{\text{flat}}$ |
| 4 | $C_2^{\text{full}}$ | all distances | $R^{\text{flat}}$ |
| 5 | $C_2^{\text{gzip}}$ | encodable distances | $R^{\propto}$ |
| 6 | $C_2^{\text{full}}$ | all distances | $R^{\propto}$ |

Figure 4: Description of the six prototypes.

There are two recycling functions that are used by the prototypes. The first one, $R^{\text{flat}}$, builds flat codes, i.e.:

$$R^{\text{flat}}(\{d_1, \ldots, d_k\}, C_2) = \text{flat}(\{1, \ldots, k\}).$$

In other words, the number of bits that one expects to recycle does not depend on the particular distance that gets chosen. The other recycling function, $R^{\propto}$, builds what we call *proportional* codes. It behaves roughly as if it were defined this way:

$$R^{\propto}(\{d_1, \ldots, d_k\}, C_2) = \text{Huffman}(\{(1, 2^{-|C_2(d_1)|}), \ldots, (k, 2^{-|C_2(d_k)|})\}).$$

Intuitively, we can say that $R^{\text{flat}}$ tends to level the number of recycled bits, for any choice of distance, while $R^{\propto}$ tends to level the *net cost* of the transmission, for any choice of distance.

We may now proceed with the description of each prototype. Note that we refer to the prototype that is used in experiment $i$ as Prototype $i$.

Prototype 1 completely ignores `gzip`'s encoding of the distances and uses $C_2^{\text{flat}}$. This makes all distances eligible. Flat recycling is used. Note that, in this particular case, proportional recycling would produce a similar encoding for recycled bit sequences. We chose to encode distances flatly based on the assumption that recycling would tend to scatter the chosen distances evenly between 1 and $d_{\max}$. This assumption turned out to be naive as the documents to compress typically are locally similar or have some other spatial regularity. Consequently, there is some profit to make by Huffman-encoding the distances and, by choosing not to do so, we damage the compression efficiency up to the point that, most of the time, it more than ruins any gain that recycling could have brought.

Prototype 2 is a reaction to our observations on the performance of Prototype 1. We choose to use $C_2^{\text{gzip}}$ and, to make sure alternate distances chosen because of recycling do not cause the compression to deteriorate, we deem only "short" distances to be eligible. Flat recycling is used. Compression efficiency of Prototype 2 was hardly comparable with that of Prototype 1: it improves for some files while it deteriorates for others. At this point, it was not clear if the highly selective criterion for eligible distances did not rule out too many options for recycling.

Consequently, Prototype 3 differs from the previous one only by the fact that all distances to which `gzip` normally gives an encoding are now considered eligible for recycling. That modification did improve compression on most files while deteriorating it only slightly on the others. The lesson that we learned was: *by offering it more options, recycling is able to bring a greater benefit even though the additional options are* more costly *to encode.* Given this lesson, could recycling bring further improvement in compression efficiency if it had the freedom to consider even distances that `gzip` does not know how to encode?

Prototype 4 does exactly that and encodes distances using $C_2^{\text{full}}$. That makes all distances to longest matches eligible. Recycling is still made using flat codes. Results show that compression improves on almost all files. Once again, giving more options to the recycling pays off even if the introduction of the formerly ignored distances could only make the encodings for the formerly considered ones to become lengthier.

Since flat recycling chooses distances to longest matches regardless of their cost, it may cause some compression inefficiency. To make prototypes more "responsible", Prototypes 5 and 6 are made identical to Prototypes 3 and 4, respectively, except that $R^{\text{flat}}$ is replaced by $R^{\propto}$. In each case, proportional recycling improves the compression efficiency over flat recycling. This was the second lesson: *recycling should be done proportionally.* Results obtained in experiment 6 show that all files are better compressed using Prototype 6 than using `gzip` set at maximal compression.

## Acknowledgements

We wish to thank the anonymous referees whose comments helped to improve this paper. This work has been funded by NSERC of Canada.

## References

[1] M. J. Atallah and S. Lonardi. Authentication of LZ-77 compressed data. In *Proceedings of the 18th ACM Symposium on Applied Computing*, pages 282–287, Melbourne, Florida, USA, mar 2003.

[2] A. Brown. `gzip-steg`, 1994.

[3] P. Deutsch. Request for comments: 1051, 1996. `http://www.ietf.org/rfc/rfc1051.txt`.

[4] D. Dubé and V. Beaudoin. Recycling bits in LZ77-based compression. In *Proceedings of the Conférence des Sciences Électroniques, Technologies de l'Information et des Télécommunications (SETIT 2005)*, Sousse, Tunisia, mar 2005.

[5] J. L. Gailly and M. Adler. The GZIP compressor. `http://www.gzip.org`.

[6] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, sep 1952.

[7] S. Lonardi and W. Szpankowski. Joint source-channel LZ'77 coding. In *Proceedings of the IEEE Data Compression Conference*, pages 273–282, Snowbird, Utah, USA, mar 2003.

[8] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 scheme and its analysis. In *Proceedings of IEEE International Symposium on Information Theory (ISIT'04)*, page 56, Chicago, Illinois, USA, 2004.

[9] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 data compression: Algorithms, analysis, and experiments. (Under submission), 2006.

[10] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding—a survey. *Proceedings of the IEEE*, 87(7):1062–1078, jul 1999.

[11] I. Witten, T. Bell, and J. Cleary. The Calgary corpus, 1987. `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.

[12] Y. Wu, S. Lonardi, and W. Szpankowski. Error-resilient LZW data compression. In *Proceedings of the IEEE Data Compression Conference*, pages 193–202, Snowbird, Utah, USA, mar 2006.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–342, 1977.