Improving LZ77 Bit Recycling using All Matches

Danny Dubé
Université Laval, Canada
Email: Danny.Dube@ift.ulaval.ca

Vincent Beaudoin
Université Laval, Canada
Email: Vincent.Beaudoin.1@ulaval.ca

***Abstract*— There exist lossless compression techniques, such as LZ77, that have the particularity that some original file may be compressed in more than one way, e.g. by choosing other matches than the closest longest ones only. The existence of multiple encodings per original file causes redundancy, i.e. it tends to make compressed files longer than necessary, on average. Recently, a technique called bit recycling was introduced to help reduce the redundancy caused by the multiplicity of encodings. It has been used to improve LZ77 compression. It exploits the fact that there often exists more than one longest match and it is called longest-match bit recycling. This work presents a more general, and more powerful, bit recycling technique that exploits shorter matches also. We call the technique all-match bit recycling. Our experiments demonstrate that at least 1 bit out of 11 results from the multiplicity of encodings, in LZ77 compression.**

I. Introduction

When one thinks about general, lossless compression, one often thinks about a compressor, $\mathcal{C}$, and a decompressor, $\mathcal{D}$, that are functions from files to files (or sequences of characters to sequences of characters) such that, for any file F, $\mathcal{D}(\mathcal{C}(\mathsf{F}))$ is F. However, the specification of a lossless compression technique often comes in two pieces: 1. what a *valid* compressed file is; 2. what original file $\mathcal{D}$ should *recover* from a valid compressed file. Note that such a specification does not describe $\mathcal{C}$ precisely. There may exist some original file F for which there are many corresponding compressed files, i.e. $\mathsf{G}_1, \ldots, \mathsf{G}_n$ such that $\mathcal{D}(\mathsf{G}_1) = \ldots = \mathcal{D}(\mathsf{G}_n) = \mathsf{F}$. $\mathcal{C}$ is free to map F to any G_i. Normally, we expect $\mathcal{C}$ to select a G_s that is among the shortest. Still, no matter $\mathcal{C}$'s choice, it remains that every G_i is valid and contributes to fill up the space of compressed files too quickly. So a compression technique that allows multiple encodings for most of the original files suffers from what we call *redundancy from the multiplicity of encodings*.

In this paper, we take LZ77 as an instance of a compression technique that is particularly prone to associate multiple compressed files to original files. We start by describing LZ77 schematically. Then, we come back to a technique, called *bit recycling*, that aims at reducing the negative effects of the multiplicity of encodings. The bit recycling techniques presented earlier all exploit the existence of multiple longest matches. We present a new bit recycling technique that is more general and powerful and that exploits the existence of all matches, longest or not. Finally, experiments that we performed are described and commented.

II. Basics of LZ77

Let us denote the original file by F. LZ77 [15] is a lossless data compression technique that works by communicating a sequential description of F. By "sequential", we mean two things: 1. the description is a *sequence* of *messages*; and 2. F is described by $\mathcal{C}$ *from the start to the end*. Each message describes the next single, or few, characters of F.

A message is of one of two kinds. It may be a *literal*, denoted by $[c]$, which explicitly indicates that the next character is c. Otherwise, it is a *match*, denoted by $\langle l, d\rangle$, which indirectly indicates that the next l characters are a copy of the l characters that appear d characters before in F. A message of either kind is denoted by M. Typically, most of the compression achieved by the technique comes from the use of matches, as many characters may be described at once by a single message.

We define the function L that gives the *length* of a message. L is defined this way: $L([c]) = 1$ and $L(\langle l, d\rangle) = l$. Our presentation of LZ77 compression makes use of an *encoding function*, or *code*, C for messages.[1] $\mathcal{C}$ uses C to translate messages into sequences of bits and $\mathcal{D}$ uses C to translate sequences of bits back into messages. C has to be a prefix-free code. Typically, C is chosen so that the expected cost of the encoding of a message is minimized. For instance, C may be built using Huffman's algorithm [8]. The *cost* of a message M, when encoded by C, is the number of resulting bits, i.e. $|C(M)|$.[2]

We introduce a sketch of the implementation of the conventional LZ77 technique. Figure 2 presents pseudo-code for both $\mathcal{C}$ and $\mathcal{D}$. The code is extremely simplified and most of the details are omitted. At each step, $\mathcal{C}$ searches for a longest match (according to L). When no match is found, or when no sufficiently long one is found, a literal is used instead. When there exist many longest matches, the *closest* one is selected, i.e. the one with the smallest d. Note that a longest match is searched for in the hope of describing F with as few messages as possible. Also, the systematic selection of the *closest* longest match has a tendency to make short distances more probable and helps to construct a C that makes the expected cost of messages lower. For the sake of simplicity, we suppose that C remains constant during the compression of F. Finally, the pseudo-code for $\mathcal{C}$ explicitly manipulates the output bit stream (or sequence) σ. The contents of σ after the last iteration can be seen as the compressed file.

In the pseudo-code for $\mathcal{D}$, the reverse operations are performed. Bit stream σ is assumed to be initialized to the compressed file. The "interpretation" consists in identifying

[1] The code C must not be confused with the compressor $\mathcal{C}$. C is used to encode individual messages only, while $\mathcal{C}$ is used to process whole files.

[2] Note that the length of a message has no direct relation to its cost.

the single or few characters that are described by M_s and in storing them in a buffer that accumulates the recovered parts of F. The interpretation may also include a variety of other operations.

Among the details that are omitted, there are the input of F by $\mathcal{C}$ and the output of F by $\mathcal{D}$, the method used by $\mathcal{C}$ to look up for matches, the way $\mathcal{C}$ and $\mathcal{D}$ convene on the choice of C, the handling of the end of F and that of the compressed file, and the conventions about the minimal and maximal lengths and distances. In our examples, we will consider that matches have to be at least 3 characters long.

Figure 1 presents two short examples. In each one, a different original file F_i^{ex} is considered. Along with each F_i^{ex}, we give all possible sequences of messages that describe F_i^{ex}, which we call *parses*. Note that the implementation of LZ77 presented in Figure 2 has no choice but to produce (and consume) Parses 1a and 2a.

III. Parses

In our study of all-match recycling, we need to be precise about the positions in F that $\mathcal{C}$ and $\mathcal{D}$ visit and where matches and literals may be used. Let us define a few functions and relations. We say that a message M *leaps* from position p to position q, denoted by $p \xrightarrow{M} q$, if M is a valid description of the characters appearing at positions p to $q-1$, inclusively. More specifically, $p \xrightarrow{[c]} p+1$ if the character at position p is c and $p \xrightarrow{\langle l,d \rangle} p+l$ if the characters appearing at positions p to $p+l-1$, inclusively, are identical to those appearing at positions $p-d$ to $p-d+l-1$, inclusively. Clearly, if $p \xrightarrow{M} q$, we have that $p + L(M) = q$. A *parse* of F consists in a sequence of messages, $M_0, \ldots, M_{n-1}$, and in a sequence of positions, $p_0, \ldots, p_n$, such that $p_0 = 0$, $p_n = |\mathsf{F}|$, and $p_i \xrightarrow{M_i} p_{i+1}$, for $0 \leq i \leq n-1$. Most of the time, we omit the positions when we present parses. Note that it also makes sense to talk about a parse of a sequence of characters other than F, for instance, a prefix of F. Finally, we will need to refer to the set of messages leaping *from* a position p and the set of those leaping *to* a position q. We denote these sets by $\vec{\mu}(p)$ and $\overleftarrow{\mu}(q)$, respectively, and they can be defined formally as follows: $\vec{\mu}(p) = \{M \mid \exists q.\ p \xrightarrow{M} q\}$ and $\overleftarrow{\mu}(q) = \{M \mid \exists p.\ p \xrightarrow{M} q\}$. Note that $M \in \vec{\mu}(p)$ if and only if $M \in \overleftarrow{\mu}(p + L(M))$.

IV. Bit recycling and principles

Bit recycling aims at reducing the negative impact of the multiplicity of encodings present in a compression technique. Instead of trying to eliminate or reduce the multiplicity itself, bit recycling exploits it and extracts a *compensation* from it.

Example 1:	Parse 1a:	[a] [b] [c] [1] $\langle 3,4 \rangle$ [2] $\langle 3,4 \rangle$
F_1^{ex}: `abc1abc2abc`	Parse 1b:	[a] [b] [c] [1] $\langle 3,4 \rangle$ [2] $\langle 3,8 \rangle$
	Parse 1c:	[a] [b] [c] [1] $\langle 3,4 \rangle$ [2] [a] [b] [c]
	Parse 1d:	[a] [b] [c] [1] [a] [b] [c] [2] $\langle 3,4 \rangle$
	Parse 1e:	[a] [b] [c] [1] [a] [b] [c] [2] $\langle 3,8 \rangle$
	Parse 1f:	[a] [b] [c] [1] [a] [b] [c] [2] [a] [b] [c]
Example 2:	Parse 2a:	[a] [b] [c] [1] [b] [c] [d] [2] $\langle 3,8 \rangle$ [d]
F_2^{ex}: `abc1bcd2abcd`	Parse 2b:	[a] [b] [c] [1] [b] [c] [d] [2] [a] $\langle 3,5 \rangle$
	Parse 2c:	[a] [b] [c] [1] [b] [c] [d] [2] [a] [b] [c] [d]

Fig. 1. Examples of original files and the corresponding parses.

$\mathcal{C}$:

```
1. while description incomplete do
2.     let M_s = clos. long. match;
3.     emit(C(M_s));
where
4. procedure emit(w):
5.     σ := σ · w;
6.
7.
```

$\mathcal{D}$:

```
1. while description incomplete do
2.     let M_s = receive();
3.     interpret M_s;
where
4. procedure receive():
5.     let M, σ' s.t. C(M) · σ' = σ;
6.     σ := σ';
7.     return M;
```

Fig. 2. Conventional implementation of LZ77.

First, bit recycling exploits the capacity to establish implicit communication between $\mathcal{C}$ and $\mathcal{D}$. Implicit communication is possible due to the multiplicity of encodings and may happen when, at some point in the compression process, we have that: 1. $\mathcal{C}$ has more than one option; and 2. $\mathcal{D}$ is able to recognize that situation. When both conditions are met, $\mathcal{C}$ may choose a particular option with the intent to mean something to $\mathcal{D}$, as if it were making an eye wink to $\mathcal{D}$. The possibility of $\mathcal{C}$ to transmit information to $\mathcal{D}$ through its choices can be seen as a *side-channel* of communication. The side-channel does not have an arbitrary capacity like the main channel, which is the compressed file. Roughly speaking, the capacity of the side-channel depends on how often both conditions are met and on how many options are available each time. Note that the realization of the existence of a side-channel due to the multiplicity of longest matches predates the introduction of bit recycling. The side-channel has been used in various applications such as steganography (information hiding), authentication, and error correction [1], [2], [9], [10], [11], [13].

Second, the particularity of bit recycling is that it tries to transmit as many bits as possible through the side-channel instead of the main channel. By doing so, it reduces the size of the compressed files. "As many bits as possible" means "as many as the side-channel can carry".

Bit recycling is performed using a *recycling code*, say r. When $\mathcal{C}$ has to select one of many options and $\mathcal{D}$ is aware of these options, r is used to assign a *recycled bit sequence* to the options. Let us say that the options presented to $\mathcal{C}$ are the messages $M_1, \ldots, M_n$. Option M_i, if selectable, is associated to the recycled bit sequence $r(M_i)$. If $\mathcal{C}$ selects M_i, then $\mathcal{D}$ is able to detect that $M_i \in \{M_1, \ldots, M_n\}$ was selected by $\mathcal{C}$, and recovers the recycled bit sequence $r(M_i)$. We say that $r(M_i)$ has been transmitted through the side-channel. Performing bit recycling means that $\mathcal{D}$ adds $r(M_i)$ to its input bit stream σ. Because of that, $\mathcal{C}$ cannot select M_i on a whim, but rather in such a way that the addition of $r(M_i)$ to σ allows $\mathcal{D}$ to properly decode the rest of F. Note that the entropy of the bits in σ is virtually 1 since σ contains compressed data. Consequently, the selection of M_i by $\mathcal{C}$ is essentially a random process.

Using r is only part of the story. The rest of the story is *how* to build an r and how to build it so that it improves compression as much as possible. The construction of an effective r has to consider the *cost* of the options. The cost of an option is measured in bits. We collect a set of options $M_1, \ldots, M_n$ presented to $\mathcal{C}$ along with their associated costs in a *cost function*, that we denote by K. $K(M_i)$ is the cost, in bits, of selecting M_i. K has to be defined on all options

and only on the options, i.e. $\mathrm{Dom}(K) = \{M_1, \ldots, M_n\}$.

A *valid* recycling code r for K must obey two conditions. First, r has to be defined on at least one option, i.e. $\emptyset \neq \mathrm{Dom}(r) \subseteq \mathrm{Dom}(K)$. Second, r has to be a prefix-free code. We say that an option M_i is deemed *worthy* if $M_i \in \mathrm{Dom}(r)$. Not all worthy options need be assigned the same number of recycled bits, i.e. $|r(M_i)| \neq |r(M_j)|$ is allowed. Note that there is much freedom in the choice of r.

Then we use the optimal recycling code constructor [6], called R, to build r from K. The desired r is $R(K)$. $R(K)$ is *a* valid recycling code for K because it is defined on *some* options and because it is a prefix-free code. $R(K)$ is also optimal for K in the sense presented below.

Let r be a valid (but not necessarily optimal) recycling code for K. Let us suppose that M_i is an option in $\mathrm{Dom}(r)$. We define the *raw cost* of M_i to be $K(M_i)$. We define the *compensation* for M_i to be the number of recycled bits, $|r(M_i)|$, assuming M_i is selected. The *net cost* of M_i is $K(M_i) - |r(M_i)|$. Due to the random nature of the bits that need to be recycled, M_i has *probability* $2^{-|r(M_i)|}$ of being selected. So, the *expected cost* of recycling using r is the expected net cost of the options in $\mathrm{Dom}(r)$, i.e. $\sum_{M_i \in \mathrm{Dom}(r)} (K(M_i) - |r(M_i)|)/2^{|r(M_i)|}$. We say that r is *optimal* if the expected cost of recycling using r is minimal among all valid recycling codes for K.

Due to lack of space, our explanations on the principles of bit recycling are rather terse. We refer the reader to our previous papers [5], [6] for more detailed explanations along with examples. There are indications on how to build an optimal recycling code but in an inefficient way [6]. Still, it is possible to build an optimal recycling code for a set of n options in $O(n)$ time. We plan to write a paper describing the technique soon (i.e. in 2008).

V. Bit recycling based on longest matches

The bit recycling presented in previous work exploits only a restricted form of multiplicity of encodings: that caused by *equivalent* messages. In the case of LZ77 compression, these equivalent messages are the multiple longest matches available to $\mathcal{C}$ at some steps. In our example, $\mathsf{F}_1^{\mathrm{ex}}$ may be described by either Parse 1a or Parse 1b, if $\mathcal{C}$ is forced to use longest matches only but has the liberty to choose arbitrarily among them. The beginnings of both parses are identical but, at the last step, either one of $\langle 3, 4\rangle$ and $\langle 3, 8\rangle$ could be selected by $\mathcal{C}$, provided both would be considered worthy. A bit would be recycled by $\mathcal{D}$ in the process. Since both messages describe "abc", we consider them to be equivalent. Note that a bit recycling LZ77 technique based on the longest matches selects a parse that goes through easily predictable positions, no matter what particular matches are selected. Note also that, in our example, $\mathsf{F}_2^{\mathrm{ex}}$ can only be parsed according to Parse 2a, because longest matches are selected eagerly.

In Figure 3, we present the pseudo-code for a bit recycling implementation of LZ77 that exploits the multiplicity of longest matches. Since this work has already been presented [4], [5], [6], [14], we only mention the key features.

$\mathcal{C}$:

```
1. while description incomplete do
2.    let M̄ = longest matches;
3.    let M_s = ND-select in M̄;
4.    emit(C(M_s));
5.    for all M ∈ M̄ do
6.       let K(M) = |C(M)|;
7.    if R(K)(M_s) defined then
8.       recycle(R(K)(M_s));
9.    else abort;
where
10. procedure emit(w):
11.    if w = ε or ρ = ε then
12.       σ := σ · w;
13.    else if w = b · w' and ρ = b · ρ'
14.    /* where b ∈ {0,1} */ then
15.       ρ := ρ';
16.       emit(w');
17.    else abort;
18. procedure recycle(w):
19.    ρ := w · ρ;
```

$\mathcal{D}$:

```
1. while description incomplete do
2.    let M_s = receive();
3.    interpret M_s;
4.    let M̄ = equiv. class of M_s;
5.    for all M ∈ M̄ do
6.       let K(M) = |C(M)|;
7.
8.    recycle(R(K)(M_s));
9.
where
10. procedure receive():
11.    let M, σ' s.t. C(M) · σ' = σ;
12.    σ := σ';
13.    return M;
14.
15.
16.
17.
18. procedure recycle(w):
19.    σ := w · σ;
```

Fig. 3. Implementation of a longest-match bit recycling technique.

In line *2*, we see that $\mathcal{C}$ considers all longest matches to be options, not just the closest one. In lines *3* and *4*, it selects one of them, M_s, and sends it. In lines *2* to *4*, $\mathcal{D}$ is able to receive M_s and to recover all the options. In lines *5* and *6*, both $\mathcal{C}$ and $\mathcal{D}$ define the cost function for the options. Note the use of R, in line *8*, which constructs an optimal recycling code from K. The rest of the pseudo-code allows $\mathcal{D}$ to recycle the bit sequences transmitted implicitly through the side-channel and it allows $\mathcal{C}$ to make the appropriate selections in order to have $\mathcal{D}$ recycle the "right" bit sequences. The use of non-determinism in $\mathcal{C}$ (**ND-select** on line *3*), along with an auxiliary bit stream ρ, is pretty involved but, due to lack of space, we refer the reader to previous papers [5], [6]. Non-determinism is used only in order to make the presentation of bit recycling easier but it is not strictly needed. A concrete implementation may use another, more "realistic" tool.

VI. Bit recycling based on all matches

A. Longest-match parses versus all-match parses

Let us illustrate how all-match parsing can account for encodings that longest-match parsing cannot. Consider the parses of $\mathsf{F}_2^{\mathrm{ex}}$ in Example 2. Recall that longest-match bit recycling can only consider Parse 2a. Considered globally, Parse 2b seems to be "as longest-match oriented as" Parse 2a. However, at position 8 (where the last "a" is located), longest-match parsing is forced to choose $\langle 3, 8\rangle$ instead of [a].

Now, let us illustrate the challenges that an all-match bit recycling technique faces. Suppose that, during the compression of $\mathsf{F}_2^{\mathrm{ex}}$, the "right" decisions force $\mathcal{C}$ to select Parse 2b. Clearly, for the first 8 messages, $\mathcal{C}$ does not have any choice and no bit recycling happens.[3] For the next message, $\mathcal{C}$ has a choice between $\langle 3, 8\rangle$ and [a]. This choice looks like an opportunity for recycling. Note however that these messages are *not* equivalent. According to our hypothesis, $\mathcal{C}$ selects [a] and sends it to $\mathcal{D}$. When $\mathcal{D}$ receives [a], it learns about "a" but it *cannot* detect that $\mathcal{C}$ had two options. Fortunately,

[3] Technically, bit recycling does happen but only with empty bit sequences.

it does not mean that the opportunity for recycling is lost. The opportunity is only postponed. When $\mathcal{D}$ receives the next message, $\langle 3, 5\rangle$, it learns about "bcd" and it is able to detect that these 3 characters could have been described in either Parse 2b's or Parse 2c's way. Consequently, a bit would be recycled in the process, provided both descriptions would be deemed worthy. But that is not the whole story, as $\mathcal{D}$ also has the capacity to detect that "abc" could have been described in either Parse 2a's or Parse 2c's way, and that "abcd", globally, could have been described in any Parse $2x$'s way.

The apparent "delay" observed in $\mathcal{D}$'s capacity to detect recycling opportunities is a consequence of the principles of bit recycling: it is not sufficient for $\mathcal{C}$ to have options, $\mathcal{D}$ must also have the capacity to *detect* these options.

B. Grouping the prefixes of the parses together

After our illustration of all-match bit recycling, it might seem unclear how and when bits could effectively by recycled. Moreover, a technique that would try to manage the whole set of possible parses is condemned to be too slow, as the number of parses is simply gigantic for most files. To make the problem tractable, we take an approach where parses are "factored" together. The approach is based on *dynamic programming*.

First, we consider the set of possible parses for *prefixes* of F. Let P_p denote the p-character prefix of F. Note that $\mathsf{P}_0 = \epsilon$ and $\mathsf{P}_{|\mathsf{F}|} = \mathsf{F}$. Note also that, if two parses of P_p, namely parse $M_0 M_1 \dots M_{k-1} M_k$ and parse $M'_0 M'_1 \dots M'_{k-1} M'_k$, have the same last message (i.e. $M_k = M'_k$), then the shortened parses $M_0 M_1 \dots M_{k-1}$ and $M'_0 M'_1 \dots M'_{k-1}$ are both parses of P_q, where $q \xrightarrow{M_k} p$.

Second, note that the sets of possible parses for P_0, $\mathsf{P}_1, \dots,$ $\mathsf{P}_{|\mathsf{F}|}$ can be obtained by induction. Let S_p be the set of possible parses for P_p. Then we have that $S_0 = \{P_0\}$, where P_0 is the empty sequence of messages, and, for $p > 0$, we have that:

$$S_p = \left\{ PM \mid q \xrightarrow{M} p \text{ and } P \in S_q \right\}.$$

Third, because the S_p's are too huge, instead of working with them, we define the notion of *summary* of S_p, called $\mathcal{S}_p$. A summary $\mathcal{S}_p$ is a mathematical entity that acts as S_p's *representative*. It has the capacity to produce a parse for P_p, usually by selecting one among many possibilities, and to associate a recycled bit sequence to the particular choice that it makes.

Fourth, $\mathcal{S}_p$'s construction starts by partitioning S_p into *groups*. A group contains the parses that have a particular message $M \in \overleftarrow{\mu}(p)$ as their *last* message. We call M the *label* of the group. Clearly, there is a one-to-one correspondence between $\overleftarrow{\mu}(p)$ (i.e. the labels) and the set of groups taken from S_p. Note also that any parse in the group labeled by M has the form PM, where $P \in S_q$ for position q such that $q \xrightarrow{M} p$, which means that P is represented by $\mathcal{S}_q$.

Fifth, summary $\mathcal{S}_p$ happens to be a recycling code that maps options to recycled bit sequences. Here, the options are the groups. When a given parse gets selected, the bit sequence associated to its group gets recycled. More precisely, a parse may get selected provided that the group to which it belongs (the option which it is part of) is deemed worthy. The whole construction of $\mathcal{S}_p$ consists in the following steps: each group is taken as an option, each option is attributed a cost (using a cost function K), and then $\mathcal{S}_p$ is the recycling code $R(K)$. There remains to determine the cost of an option. Let us take a group of $\mathcal{S}_p$ labeled by $M \in \overleftarrow{\mu}(p)$, where $q \xrightarrow{M} p$. The cost of M's option is the cost of a parse of P_q plus $|C(M)|$. However, the parse of P_q is produced by $\mathcal{S}_q$, which is itself a recycling code, not a plain sequence of bits. It means that we do not have a definitive cost for the parse produced by $\mathcal{S}_q$. Moreover, since we do not know ahead of time which of $\mathcal{S}_p$'s options will be selected, we can only hope to obtain an *expected cost* for $\mathcal{S}_p$. The expected cost for each summary is computed inductively and stored in an array $\mathcal{E}$. For each position p, $\mathcal{E}[p]$ is the expected cost of $\mathcal{S}_p$. In fact, we compute only *estimates* of the expected costs with a precision up to the quarter of a bit. We have that $\mathcal{E}[0] = 0$ because $\mathcal{S}_0$ represents a single, empty parse. For $p > 0$, we have that:

$$\mathcal{E}[p] = \sum_{M \in \mathrm{Dom}(R(K))} \left(K(M) - |R(K)(M)|\right) \;/\; 2^{|R(K)(M)|},$$

where the cost function K is defined in lines *6* and *7* of Figure 4.

$\mathcal{C}$:

```
1. p := 0;
2. while description incomplete do
3.    let M_s = ND-select in μ⃗(p);
4.    emit(C(M_s));
5.    p := p + L(M_s);
6.    for all M ∈ μ⃗(p) do
7.       let K(M) = cost(M);
8.    if R(K)(M_s) defined then
9.       recycle(R(K)(M_s));
10.   else abort;
where
11. procedure cost(M):
12.    return E[p − L(M)] + |C(M)|;
13. procedure emit(w): …
21. procedure recycle(w): …
```

$\mathcal{D}$:

```
1. p := 0;
2. while description incomplete do
3.    let M_s = receive();
4.    interpret M_s;
5.    p := p + L(M_s);
6.    for all M ∈ μ⃗(p) do
7.       let K(M) = cost(M);
8.
9.    recycle(R(K)(M_s));
10.
where
11. procedure cost(M):
12.    return E[p − L(M)] + |C(M)|;
13. procedure receive(): …
21. procedure recycle(w): …
```

Fig. 4. Implementation of an all-match bit recycling technique.

C. The algorithms

Figure 4 presents the pseudo-code for $\mathcal{C}$ and $\mathcal{D}$ in an implementation of an all-match recycling technique. Note that the current position, p, is now updated explicitly. There is a new function **cost** that helps in computing the costs of the options. Note how $\mathcal{C}$ selects M_s from the current position (where M_s leaps from) but recycles from the new position (where M_s leaps to). This is because $\mathcal{D}$ does not have access to any other information than that contained in M_s and its predecessors. The interpretation operation performed by $\mathcal{D}$ includes the computation of $\mathcal{E}$'s entries on the fly, as soon as additional characters become known to $\mathcal{D}$.

VII. Experiments

We evaluated the performance of the all-match recycling technique by modifying the implementation of the Deflate compression method [3] that is part of the popular compression tool gzip [7]. We kept Deflate's particular choices for the size of the sliding window and for the bounds on the lengths

and distances. We slightly modified Deflate's algorithm for the construction of codes to ensure that every length and distance can be encoded. Deflate's normal behavior is to assign a bit sequence only to the lengths and distances that it has to encode. Because of bit recycling, other lengths and distances might get chosen and so they must be made encodable as well.

We compared three compression techniques: `gzip` set to maximal compression; a longest-match (**L-M**) bit recycling technique; and the all-match (**A-M**) bit recycling technique presented in this paper. Note that the results for **L-M** are not exactly the same as those we presented in the 2006 paper [5]. The implementation used to obtain the results presented in that paper does not use optimal recycling, while the one used here does. The benchmarks used in our experiments are the files contained in the Calgary corpus [12].

Figure 5 presents the results. The measurements are in bytes. We can see that recycling over all the matches improves the compression efficiency significantly more than recycling over the longest matches only. It is particularly remarkable that, in the case of the files for which longest-match recycling does not bring much improvement, all-match recycling is able to obtain a significant advantage. Also, note that the improvement that all-match recycling achieves comes from the fact that it allows itself to consider parses that are, in principle, more costly than those considered by longest-match recycling. It constitutes one additional evidence supporting our belief that recycling works better when more options are offered to it, even if many or most of the new options are more costly.

Taken all together, the benchmarks are 2.9% smaller when compressed using longest-match recycling than when compressed using `gzip`. Using all-match recycling, they are 9.2% smaller than using `gzip` (excluding `pic`, this time).

Our prototype was not able to process the file `pic`. This file is a binary image. A large part at the end of this file is filled with the NUL character. When $\mathcal{C}$ parses this part, it has to consider more and more options as the sliding window fills up with NUL characters. Eventually, *every* match that can be expressed by the Deflate method is a valid match, i.e. $\langle l, d\rangle$ is part of $\breve{\mu}(p)$ for all l and d allowed by Deflate.

Name	Orig.	Gzip	L-M	A-M
bib	111 261	34 900	33 829	31 757
book1	768 771	312 281	301 538	279 435
book2	610 856	206 158	199 906	185 321
geo	102 400	68 414	66 133	63 341
news	377 109	144 400	140 142	132 679
obj1	21 504	10 320	10 304	10 043
obj2	246 814	81 087	79 068	75 360
paper1	53 161	18 543	18 129	16 938
paper2	82 199	29 667	28 892	26 720
paper3	46 526	18 074	17 675	16 422
paper4	13 286	5 534	5 440	5 156
paper5	11 954	4 995	4 916	4 688
paper6	38 105	13 213	13 031	12 225
pic	513 216	52 381	51 440	N/A
progc	39 611	13 261	13 069	12 212
progl	71 646	16 164	15 704	14 509
progp	49 379	11 186	10 911	10 186
trans	93 695	18 862	18 420	17 477

Fig. 5. Experimental results.

In terms of speed, our prototype of all-match bit recycling is very slow. The time it takes to process the benchmarks is about two orders of magnitude longer than that taken by `gzip`. On the other hand, the prototype of longest-match recycling is a bit above an order of magnitude slower than `gzip`. We did not put much effort to obtain fast prototypes. Note that, going from `gzip` to the longest-match recycling prototype and then to the all-match one, more and more of the code that is run is code that has been rewritten by us. This factor contributes to make the prototypes slower.

VIII. Conclusion

We have presented, implemented, and tested a bit recycling technique for LZ77 compression that exploits all matches. The measured 9.2% reduction in the size of the compressed files means that, when compressing some original file using `gzip`, about 1 bit out of 11 is produced only to indicate which particular compressed file has been selected among all possible compressed files. Each of these eleventh bits encodes useless information. Since our bit recycling technique cannot pretend to recover all the bits wasted by the multiplicity of encodings, it means that, in fact, even more bits are useless. This result clearly indicates that, with LZ77 compression, the redundancy solely caused by the multiplicity of encodings can be quite high. It is so, even when using a high performance compressor like `gzip`.

References

[1] M. J. Atallah and S. Lonardi. Authentication of LZ-77 compressed data. In *Proc. of the ACM SAC*, pages 282–287, Melbourne, Florida, USA, March 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 *Proc. of SETIT*, Sousse, Tunisia, March 2005.

[5] D. Dubé and V. Beaudoin. Improving LZ77 data compression using bit recycling. In *Proc. of ISITA*, Seoul, South Korea, October 2006.

[6] D. Dubé and V. Beaudoin. Bit recycling with prefix codes. In *Proc. of DCC*, page 379, Snowbird, Utah, USA, March 2007. Poster.

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

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

[9] S. Lonardi and W. Szpankowski. Joint source-channel LZ'77 coding. In *Proc. of DCC*, pages 273–282, Snowbird, Utah, USA, March 2003.

[10] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 scheme and its analysis. In *Proc. of ISIT*, page 56, Chicago, Illinois, USA, 2004.

[11] S. Lonardi, W. Szpankowski, and M. Ward. Error resilient LZ'77 data compression: Algorithms, analysis, and experiments. *IEEE Trans. on Information Theory*, 53(5):1799–1813, 2007.

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

[13] Y. Wu, S. Lonardi, and W. Szpankowski. Error-resilient LZW data compression. In *Proc. of DCC*, pages 193–202, Snowbird, Utah, USA, March 2006.

[14] H. Yokoo. Lossless data compression and lossless data embedding. In *Proc. of the Asia-Europe Workshop on Concepts in Information Theory*, Jeju, South Korea, October 2006.

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