An Extremely Fast ZIV-Lempel Data Compression Algorithm

Ross N. Williams

Renaissance Software
GPO Box 171
Adelaide 5001, Australia

ross@spam.ua.oz.au

Abstract

A new, simple, extremely fast, locally adaptive data compression algorithm of the LZ77 class is presented. The algorithm, called LZRW1, almost halves the size of text files, uses 16K of memory, and requires about 13 machine instructions to compress and about 4 machine instructions to decompress each byte. This results in speeds of about 77K and 250K bytes per second on a one MIPS machine. The algorithm runs in linear time and has a good worst case running time. It adapts quickly and has a negligible initialization overhead, making it fast and efficient for small blocks of data as well as large ones.

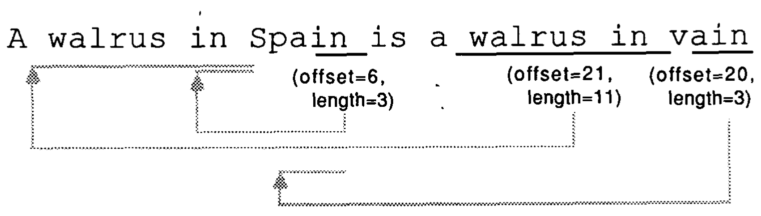
1 Introduction

Text data compression techniques can be partitioned into ad hoc techniques, dictionary techniques, and statistical techniques (see reviews in **[Williams90][Bell90a]** **[Storer88]**). Of these, the class of adaptive dictionary algorithms pioneered by Ziv and Lempel**[Ziv77][Ziv78]** provides the most practical trade off between flexibility, compression, and speed. Most modern text data compression programs, including the popular Unix *compress* utility (derived from **[Welch84]**), use Ziv and Lempel (LZ) algorithms.

Although current LZ algorithms are generally fast, they are not always fast enough for embedded real-time systems with tight throughput requirements. In such systems time is the important constraint and designers of compression algorithms must concentrate primarily on speed, with compression taking second place. This paper presents a new LZ variant called LZRW1 that was designed with speed as the primary objective. The algorithm takes only a few instructions to process each byte, but still yields useful compression. LZRW1 is based on the A1 algorithm by Fiala and Greene[**Fiala88**] which itself is a member of the class of LZ77 algorithms[**Ziv77**].

2 The LZRW1 Algorithm

LZRW1 uses the single pass literal/copy mechanism of [**LZ77**]. At each step, the next few bytes of input are transmitted either directly as a string or as a pointer to the text already transmitted (the **history**). **Figure 1** shows how a particular message would be divided into **literal items** and **copy items** if the minimum length of an uncompressed copy item were three bytes. Compression is achieved by representing as much of the message using copy items, resorting to literal items only when a match of three or more bytes cannot be found.

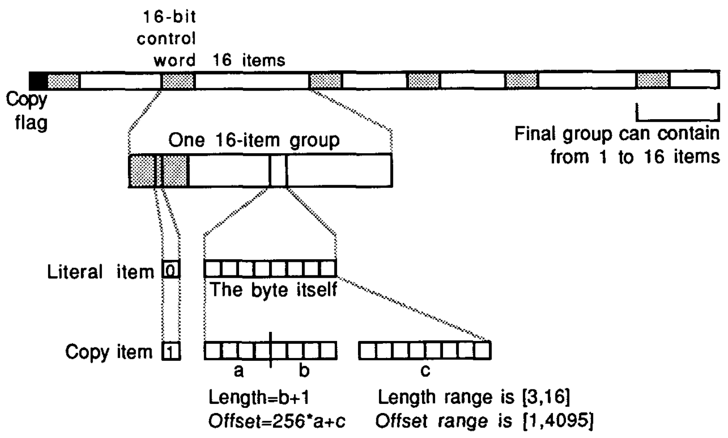


LZ77 class algorithms (of which LZRW1 is a member) achieve compression by converting the text into a sequence of items, each of which can be either a literal item or a copy item. Literal items consist of the text they represent. Copy items consist of an offset and a pointer that together point to a substring of the text already transmitted.

Figure 1: The literal/copy compression technique.

Figure 2 shows how LZRW1 arranges the items into compressed data. A literal item contains a single byte that is coded directly. A copy item contains two bytes that specify the length[3,16] and offset[1,4095] of a string appearing in the most recent 4095 bytes of the history (the **Lempel**). Control bits indicate whether items are literal or copy items, and are clustered into groups of 16 to preserve byte alignment.

Figure 3 gives a snapshot of the LZRW1 compression algorithm in execution. The algorithm's data structures, which form its source model, consist of a few



This figure gives LZRW1's compressed data format. LZRW1 uses a single bit to determine whether each item is a literal item or a copy item. To preserve byte alignment, items are clustered into ·16-item groups preceded by a 16-bit control word.

Figure 2: LZRW1's compressed data format.

scalar variables, the input block, and a hash table of 4096 pointers. The hash table maps any three byte key to a single pointer that can point anywhere in memory, but which is likely to point to a matching key somewhere in the Lempel. At each step the hash table is used to try to find a match in the Lempel for the first three or more bytes of the **Ziv** (the next 16 bytes to be coded) with a match resulting in the generation of a copy item. Because the algorithm checks all pointers that it fetches from the hash table, the hash table does not need to be initialized. LZRW1 updates its hash table after every item rather than every byte, making the hash table update rate inversely proportional to compression. This policy also exploits the phrase structure[**Langdon84**] present in most data.

Decompression is extremely simple and fast. The decompressor processes one item at a time, translating it into one or more bytes which are appended to the end of the output. If the item is a literal item, its single literal byte is appended. If the item is a copy item, the length and offset fields are used to locate and copy a string already in the output block. Control bits must be buffered.

LZRW1's hash function is constructed in accordance with the advice in [**Knuth81**] and seems to be near optimal. A recently published hash function that used a lookup table[**Pearson90**] was tried, but yielded identical performance. LZRW1's hash table provides at most the most recent Lempel match of a key. Bell[**Bell90b**] discusses other LZ77 search techniques.

A precise specification of the LZRW1 text data compression algorithm is given in **Figures 4–6** which provide a turnkey implementation in the C programming language[**Kernighan88**].

3 Experiments

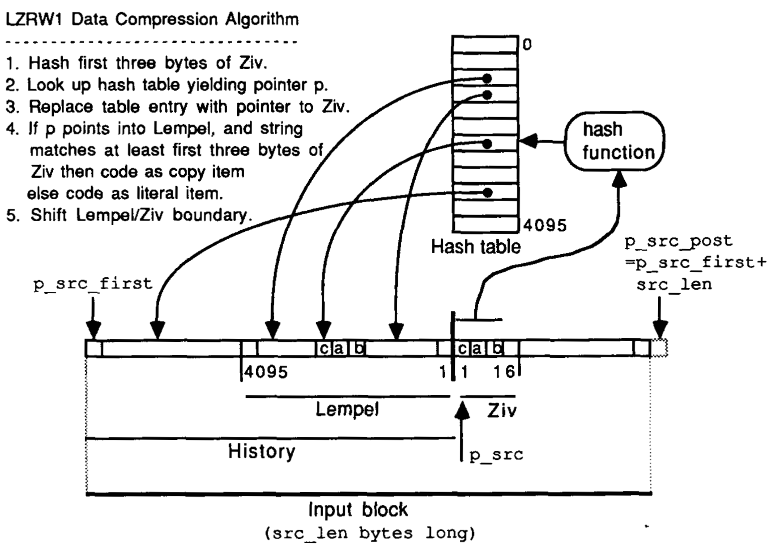
To test its performance, LZRW1 was implemented in high and low level languages and applied to the standard corpus of test files described in Appendix B of [**Bell90a**].

Figure 7 gives the results of running the C implementation of LZRW1 of **Figures 4–6** against a similar implementation of the A1 algorithm and against the Unix *compress* utility (LZC) on a Pyramid 9820 computer running Unix. LZRW1 compresses about 10% worse than LZC, but runs four times faster. LZRW1 compresses about 4.3% worse than the A1 algorithm, but runs ten times faster. This indicates that A1's exhaustive search of the Lempel and its use of a two-byte minimum copy length does not greatly improve its compression.

Figure 8 gives the performance of an hand-optimized 68000 assembly language implementation of LZRW1 running it on an 8MHz Macintosh-SE computer. LZRW1 achieves reasonable compression while requiring an average of just thirteen machine instructions to compress and four machine instructions to decompress each byte. The performance of LZRW1 on the files *zeros* and *noise* demonstrates the stability of the algorithm. For the 68000 runs, each file was divided into 16K blocks each of which was compressed independently. This impaired compression by at most 2% absolute. This fact coupled with LZRW1's negligible initialization cost makes LZRW1 ideal for small blocks of data.

4 Conclusion

Use of a simple hash table mechanism along with ruthless elimination of housekeeping has led to the extremely fast LZRW1 text compression algorithm that requires about 13 machine instructions to compress each byte and about 4 machine instructions to decompress each byte. This results in speeds of about 77K and 250K per second on a one MIP machine. LZRW1 compresses ten percent worse than the popular Unix *compress* utility, but runs four times faster, making it possibly the fastest adaptive text compression algorithm yet.



LZRW1 Data Compression Algorithm
1. Hash first three bytes of Ziv.
2. Look up hash table yielding pointer p.
3. Replace table entry with pointer to Ziv.
4. If p points into Lempel, and string matches at least first three bytes of Ziv then code as copy item else code as literal item.
5. Shift Lempel/Ziv boundary.
0
4095
Hash table
hash function
p_src_first
p_src_post =p_src_first+ src_len
c a b
4095
1
1
16
Lempel
Ziv
History
p_src
Input block
(src_len bytes long)

This figure gives a snapshot of the LZRW1 compression algorithm in execution. The horizontal bar represents the input block (in memory) which is used directly as a read-only data structure. The hash table maps three-byte keys to pointers that can point anywhere in memory, but which are likely to point to a recent occurrence of the key in the input already scanned (the history). At each step the hash table is used to map the first three bytes of the Ziv (defined to be the first sixteen bytes of the remaining part of message) to such a pointer. To keep the hash table up to date, the hash table entry from which the pointer was just fetched is replaced by a pointer to the Ziv. If the pointer fetched points to one of the most recent 4095 bytes of the history (the Lempel) and points to a match with the Ziv of at least three bytes, a copy item is constructed representing the bytes matched, otherwise a literal item is constructed representing the first byte in the Ziv. Because the algorithm checks the pointers that it obtains from the hash table, the hash table need not be initialized.

Figure 3: The LZRW1 algorithm in execution.

```
void lzrw1_compress(p_src_first,src_len,p_dst_first,p_dst_len)
/* Input  : Specify input block using p_src_first and src_len.          */
/* Input  : Point p_dst_first to the start of the output zone (OZ).     */
/* Input  : Point p_dst_len to a ULONG to receive the output length.    */
/* Input  : Input block and output zone must not overlap.               */
/* Output : Length of output block written to *p_dst_len.               */
/* Output : Output block in Mem[p_dst_first..p_dst_first+*p_dst_len-1]. */
/* Output : May write in OZ=Mem[p_dst_first..p_dst_first+src_len+256-1].*/
/* Output : Upon completion guaranteed *p_dst_len<=src_len+FLAG_BYTES.  */
UBYTE *p_src_first,*p_dst_first; ULONG src_len,*p_dst_len;
#define PS *p++!=*s++  /* Body of inner unrolled matching loop.         */
#define ITEMMAX 16     /* Maximum number of bytes in an expanded item.  */
{UBYTE *p_src=p_src_first,*p_dst=p_dst_first;
 UBYTE *p_src_post=p_src_first+src_len,*p_dst_post=p_dst_first+src_len;
 UBYTE *p_src_max1=p_src_post-ITEMMAX,*p_src_max16=p_src_post-16*ITEMMAX;
 UBYTE *hash[4096],*p_control; UWORD control=0,control_bits=0;
 *p_dst=FLAG_COMPRESS; p_dst+=FLAG_BYTES; p_control=p_dst; p_dst+=2;
 while (TRUE)
   {UBYTE *p,*s; UWORD unroll=16,len,index; ULONG offset;
    if (p_dst>p_dst_post) goto overrun;
    if (p_src>p_src_max16)
      {unroll=1;
       if (p_src>p_src_max1)
         {if (p_src==p_src_post) break; goto literal;}}
    begin_unrolled_loop:
       index=((40543*((((p_src[0]<<4)^p_src[1])<<4)^p_src[2]))>>4) & 0xFFF;
       p=hash[index]; hash[index]=s=p_src; offset=s-p;
       if (offset>4095 || p<p_src_first || offset==0 || PS || PS || PS)
         {literal: *p_dst++=*p_src++; control>>=1; control_bits++;}
       else
         {PS || PS || PS || PS || PS || PS || PS ||
          PS || PS || PS || PS || PS || PS || s++; len=s-p_src-1;
          *p_dst++=((offset&0xF00)>>4)+(len-1); *p_dst++=offset&0xFF;
          p_src+=len; control=(control>>1)|0x8000; control_bits++;}
    end_unrolled_loop: if (--unroll) goto begin_unrolled_loop;
    if (control_bits==16)
      {*p_control=control&0xFF; *(p_control+1)=control>>8;
       p_control=p_dst; p_dst+=2; control=control_bits=0;}
   }
 control>>=16-control_bits;
 *p_control++=control&0xFF; *p_control++=control>>8;
 if (p_control==p_dst) p_dst-=2;
 *p_dst_len=p_dst-p_dst_first;
 return;
 overrun: fast_copy(p_src_first,p_dst_first+FLAG_BYTES,src_len);
          *p_dst_first=FLAG_COPY; *p_dst_len=src_len+FLAG_BYTES;
}
```

Figure 4: The LZRW1 compression algorithm.

```
#define UBYTE unsigned char /* Unsigned     byte (1 byte )        */
#define UWORD unsigned int  /* Unsigned     word (2 bytes)        */
#define ULONG unsigned long /* Unsigned longword (4 bytes)        */
#define FLAG_BYTES    4     /* Number of bytes used by copy flag. */
#define FLAG_COMPRESS 0     /* Signals that compression occurred. */
#define FLAG_COPY     1     /* Signals that a copyover  occurred. */
void fast_copy(p_src,p_dst,len) /* Fast copy routine.             */
UBYTE *p_src,*p_dst; {while (len--) *p_dst++=*p_src++;}
```

Figure 5: Definitions used by LZRW1 code.

```
void lzrw1_decompress(p_src_first,src_len,p_dst_first,p_dst_len)
/* Input  : Specify input block using p_src_first and src_len.          */
/* Input  : Point p_dst_first to the start of the output zone.          */
/* Input  : Point p_dst_len to a ULONG to receive the output length.    */
/* Input  : Input block and output zone must not overlap. User knows    */
/* Input  : upperbound on output block length from earlier compression. */
/* Input  : In any case, maximum expansion possible is eight times.     */
/* Output : Length of output block written to *p_dst_len.               */
/* Output : Output block in Mem[p_dst_first..p_dst_first+*p_dst_len-1]. */
/* Output : Writes only  in Mem[p_dst_first..p_dst_first+*p_dst_len-1]. */
UBYTE *p_src_first, *p_dst_first; ULONG src_len, *p_dst_len;
{UWORD controlbits=0, control;
 UBYTE *p_src=p_src_first+FLAG_BYTES, *p_dst=p_dst_first,
       *p_src_post=p_src_first+src_len;
 if (*p_src_first==FLAG_COPY)
   {fast_copy(p_src_first+FLAG_BYTES,p_dst_first,src_len-FLAG_BYTES);
    *p_dst_len=src_len-FLAG_BYTES; return;}
 while (p_src!=p_src_post)
   {if (controlbits==0)
      {control=*p_src++; control|=(*p_src++)<<8; controlbits=16;}
    if (control&1)
      {UWORD offset,len; UBYTE *p;
       offset=(*p_src&0xF0)<<4; len=1+(*p_src++&0xF);
       offset+=*p_src++&0xFF; p=p_dst-offset;
       while (len--) *p_dst++=*p++;}
    else
       *p_dst++=*p_src++;
    control>>=1; controlbits--;
   }
 *p_dst_len=p_dst-p_dst_first;
}
```

Figure 6: The LZRW1 decompression algorithm.

		%Rem			K/Sec					
File	K	A1	LZC	RW1	A1		LZC		RW1	
bib	109	53.8	41.8	59.4	27	435	58	94	213	388
book1	751	61.6	43.2	67.9	21	419	46	90	186	368
book2	597	52.5	41.1	59.0	24	435	49	92	215	375
geo	100	94.0	76.0	84.4	13	270	48	74	167	313
news	368	56.8	48.3	61.3	27	433	44	87	210	392
obj1	21	60.8	65.3	61.7	4	350	53	75	191	420
obj2	241	47.0	52.1	51.3	23	482	41	86	225	423
paper1	52	51.5	47.2	57.8	26	433	58	91	226	371
paper2	80	54.0	44.0	61.0	23	401	64	94	206	382
pic	501	*23.3	12.1	25.6	37	604	108	148	346	506
progc	39	49.1	48.3	54.6	29	387	55	90	242	387
progl	70	35.5	37.9	43.7	20	466	65	103	241	412
progp	48	35.4	38.9	42.8	19	438	63	98	241	344
trans	91	40.8	40.8	46.1	25	457	61	96	241	436
Average	219	51.2	45.5	55.5	22	429	58	94	224	394

This table compares the performance of the A1 (a simple LZ77 class algorithm by Fiala and Greene), LZC (Unix *compress* which is based on the LZW algorithm), and LZRW1 (the topic of this paper) algorithms coded in C and running on a Pyramid 9820 computer. The implementation of A1 used a hash table indexing into a bounded buffer array holding linked lists of hash matches. A standard corpus of files was used for the test. The %Rem columns give compression as a percentage remaining. The K/Sec columns give the compression and decompression speeds in kilobytes (1024) per second. Decompression speeds are given relative to *output* (uncompressed) bytes. Speeds were calculated from the *user* time field given by an application of the unix *time* command to a block of ten consecutive compression runs. (*Note: The A1 algorithm took so long to run on the file *pic* that it was terminated and re-run, for the file *pic* only, with an upperbound of ten on its search). A comparison of LZRW1 to its ancestor algorithm A1 shows that, while LZRW1 compresses about 4.3% worse than the A1 algorithm, LZRW1 runs ten times faster. LZRW1 compresses about 10% absolute worse than LZW, but runs four times faster. LZRW1 compresses relatively poorly on English text files (e.g. book1), but compresses relatively well on non-English text files such as program texts and object files for which it betters LZW on both compression and speed.

Figure 7: Performance of algorithms on a Pyramid 9820.

File	K	/16K	%Rem	K/Sec		Ins/Byte	
bib	109	7	61.0	43	147	13.4	4.1
book1	751	47	69.5	40	132	14.7	4.6
book2	597	38	60.5	44	142	13.1	4.3
geo	100	7	85.6	31	131	18.5	4.7
news	368	24	63.0	41	150	14.0	4.0
obj1	21	2	61.9	41	161	14.2	3.7
obj2	241	16	52.5	49	156	11.8	3.8
paper1	52	4	59.5	45	143	12.9	4.2
paper2	80	6	62.4	44	138	13.3	4.4
pic	501	32	25.8	87	187	6.5	3.1
progc	39	3	55.7	47	149	12.3	4.0
progl	70	5	44.7	58	157	10.0	3.8
progp	48	4	44.2	58	158	9.9	3.7
trans	91	6	47.9	53	162	10.9	3.7
Average	219	14	56.7	49	151	12.5	4.0

zeros	78	5	13.4	136	205	4.1	2.7
noise	78	5	100.0	23	1178	24.4	0.25

This table gives the performance of an hand-optimized 68000 assembly language implementation of LZRW1 running on an 8MHz, 24-bit bus, Macintosh-SE computer. The input files were divided into 16K blocks which were compressed independently. The %Rem column gives the compression as a percentage remaining. The K/Sec column gives the compression and decompression speeds in kilobytes (1024) per second. Decompression speeds are given relative to *output* (uncompressed) bytes. The speeds are for execution only and do not include IO. The Ins/Byte column gives the average number of 68000 instructions required to process each byte during compression and decompression and was obtained by inserting counting instructions into the code and performing a separate run. All results were rounded to the given accuracy. A comparison of these results to those of **Figure 7** shows that the division of the input file into 16K blocks worsened compression by an average of only 1.2% absolute with the maximum impact being 1.8% absolute. The results for the files *zeros* (zero bytes) and *noise* (uniformly and independently random bytes) suggest that LZRW1's best running time is about one third of its average running time, and that its worst running time is about twice its average running time. For ordinary text data, LZRW1 requires about 13 instructions to compress and about 4 instructions to decompress each byte.

Figure 8: Performance of optimized 68000 LZRW1.

5 References

[Bell90a] Bell T.C., Cleary J.G., Witten I.H., "Text Compression", Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Bell90b] Bell T.C., "Longest Match String Searching For Ziv-Lempel Compression", Department of Computer Science, University of Canterbury, Christchurch, New Zealand.

[Fiala88] Fiala E.R., Greene D.H., "Data Compression with Finite Windows", *Communications of the ACM*, Vol. 32, No. 4, pp. 490–505.

[Kernighan88] Kernighan B.W., Ritchie D.M., "The C Programming Language", Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Knuth81] Knuth D.E., "Sorting and Searching", The Art of Computer Programming, Vol. 23, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[Langdon84] Langdon G.G, "On Parsing Versus Mixed-Order Model Structures for Data Compression", IBM Research Report RJ-4163 (46091) 1/18/84, IBM Research Laboratory, San Jose, CA 95193, 1984.

[Pearson90] Pearson P.K., "Fast Hashing of Variable-Length Text Strings", *Communications of the ACM*, Vol. 33, No. 6, June 1990, pp. 677–680.

[Storer88] Storer J.A., "Data Compression: Methods and Theory", Computer Science Press, 1803 Research Boulvard, Rockville, Maryland 20850, 1988.

[Welch84] Welch T.A., "A Technique for High-Performance Data Compression", *IEEE Computer*, Vol. 17, No. 6, pp. 8–19.

[Williams90] Williams R.N., "Adaptive Data Compression", Kluwer Academic Publishers, 1990.

[Ziv77] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343.

[Ziv78] Ziv J., Lempel A., "Compression of Individual Sequences via Variable-Rate Coding", *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536.