The ZPAQ Open Standard Format for Highly Compressed Data - Level 2

Version 2.05: Mar. 15, 2016 by Matt Mahoney.

Abstract

This document specifies the ZPAQ open standard format for highly compressed byte string data. The format supports memory to memory compression, single file compressors, and archivers, either solid or with independently compressed files. The compression algorithm uses an optional bitwise context mixing model (like PAQ8 [1]), followed by arithmetic decoding, packing into bytes, and an optional post-processing transform. The format supports future improvements in the compression of arbitrarily complex data types without loss of compatibility because the compressed stream contains instructions for specifying the model architecture, and byte-code programs to compute arbitrarily complex contexts and transforms. The format may be used as a container for streaming and journaling archives, described herein. A streaming archive may be extracted in a single pass. A journaling archive supports deduplication and is append-only to support rollback. The standard is open, in that no license is required to develop or use software that reads or writes ZPAQ compliant data.

Scope

This document describes the ZPAQ level 1 and 2 specifications. Level *n* implementations, where $n \geq 1$, shall be able to read data produced by any level *m* implementation, where $1 \leq m \leq n$.

The reference decoder **unzpaq** is an integral part of this specification. Only the decompression algorithm is defined.

Both this document and the program may be updated after finalization of the standard to fix software errors or resolve discrepancies between the specification and the code with the goal of preserving compatibility with earlier versions of the code. As of this revision, the current version is v2.05 (file name **unzpaq205.cpp**).

1. Introduction

The ZPAQ open standard specifies a compressed representation for one or more byte (8 bit value) sequences. A ZPAQ stream consists of a sequence of *blocks* that can be decompressed independently. A block consists of a sequence of *segments* that must be decompressed sequentially from the beginning of the block. Each segment might represent an array of bytes in memory, a file, or a contiguous portion of a file.

ZPAQ optionally uses a context mixing data compression algorithm based on the PAQ series (PAQ8, PAQ9, LPAQ) [1]. The decompressed stream is decoded one bit at a time, packed into bytes, and then transformed through an optional post-processor to undo transforms that were intended to make the data more compressible. A bit is decoded by a *model*, which predicts (assigns a probability to) the next bit based on previously decoded bits, an arithmetic decoder which takes the prediction and the compressed data and outputs the bit. The bit is fed back to the model so that it can refine future predictions.

The decoded data for each block starts with a flag to indicate whether it is should be output directly or post-processed. In the latter case, it consists of a program followed by its input data. The output of this program is the output of the decompresser. In either case, the output may then be divided into separate

arrays or files as described in the segment headers.

A model is a set of *components* that make independent predictions given a *context* and/or by combining the predictions of other components. Each component has a context that is computed from the previously decoded bits by a program described in the block header. For example, the context could be a hash of the last 20 bits. Up to 255 components of the following types may be connected in an arbitrary manner for each block:

- CONST - The prediction is a fixed value.
- CM - Context Model - A table maps the context to a prediction (initially $p_0 = p_1 = 1/2$) and a counter. After a bit is decoded, the prediction is adjusted in proportion to the prediction error and inversely proportional to the count, and the count is incremented up to a specified limit.
- ICM - Indirect Context Model - A hash table maps the context to a bit history, a state representing bounded counts of previously seen 0 and 1 bits (initially both 0) and the most recent bit. A second table maps the history to a prediction. After a bit is decoded, the history is updated and the prediction is adjusted to reduce the prediction error.
- MATCH - An index maps the context to the most recent occurrence of the same context in the output buffer. The bits following the match are predicted with a confidence that depends on the length of the match. The index is updated every 8 bits.
- AVG - Two predictions are combined by weighted averaging. The model specifies the weights.
- MIX2 - Two predictions are combined by weighted averaging. The weight (initially 1/2) is selected from a table by context. After decoding, the weight is adjusted in proportion to the prediction error times the input difference times a specified learning rate. This has the effect of favoring the most accurate component in each possible context.
- MIX - A mixer like MIX2 but with input from a contiguous block of *m* components. There is a weight for each input. The weights are adjusted to favor the most accurate components, but are not constrained to add to 1. The weights are initially 1/*m*.
- ISSE - Indirect secondary symbol estimation. A table maps the context to a bit history as with an ICM. The history is used as the context to a 2 input MIX with independent weights and one input fixed. The PAQ9A [1] model is a cascade of these ISSE with increasingly higher context orders. After decoding, the bit history and weights are updated as with an ICM and MIX.
- SSE - Secondary Symbol Estimation. SSE takes a quantized input prediction and a context and outputs a new (interpolated) prediction from a table. After decoding, the nearest table entry is adjusted to reduce the prediction error as with a CM.

One possible architecture is shown below. This example is similar to PAQ8.

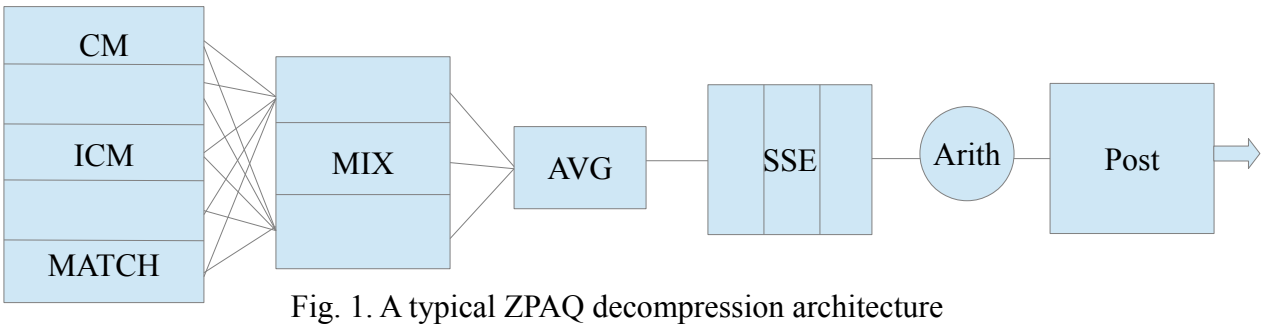

Fig. 1. A typical ZPAQ decompression architecture

Component input and output predictions are expressed as log odds. If a component predicts that a 0 or 1 will occur with probability p_0 and p_1 respectively, then the output is $p = \text{stretch}(p_1) = ln\ p_1/p_0$. Predictions are computed along a fixed sequence of components with input from earlier components. The input to the arithmetic decoder is p_1 from the last component, where $p_1 = \text{squash}(p) = 1/(1 + e^{-p})$ is the inverse of $\text{stretch}(p_1)$. This has the effect of weighting models with high confidence predictions (large magnitudes) more heavily.

Contexts hashes are computed by a program described in the block header and run once every 8 bits during decompression, and combined with the partially decoded current byte to form a complete context. The program runs on a virtual machine which takes the last decoded byte as input and writes the context hashes to the components. The language, called ZPAQL, resembles an assembly language so that it can be implemented efficiently while allowing for arbitrarily complex contexts. Another program in the same language is used for post-processing. Both programs are called once for each byte of input data.

The compressed data represents a high precision binary number in the range (0, 1) with the most significant bits of the fraction first. The arithmetic decoder maintains a range (low, high), initially (0, 1), which bounds the data and shrinks as decoding proceeds. The decoder receives a prediction and splits the range into two parts in proportion to p_0 and p_1. Whichever part now contains the compressed data determines the decoded bit and the new range. Arithmetic coding effectively codes each bit Y at a cost very close to the theoretical limit of $\log_2 1/p_Y$ bits. To mark the end of the data, each decoded byte is preceded by a EOS (end of segment) bit, which is 1 after the last byte, coded with p_1 very near 0. The decoder is designed so that the end of the coded segment can also be found by quickly scanning without decompressing the data.

The model and post-processor are initialized at the start of each block and maintain state information across segments. The arithmetic coder is initialized at the start of each segment. Segment boundaries are invisible to the model; the block appears as a continuous stream of bytes. The purpose of segments is to allow decompression to different destinations (e. g. to different files) and to signal the post-processor at the end of each file.

Each segment is optionally followed by the SHA1 hash [2] of the original compressed data. A decompresser may compute the hash of the output and compare it with this checksum to detect errors.

Level 2 allows the context model to be omitted. In this case, the data may be stored uncompressed or decompressed solely by the postprocessor using an arbitrary algorithm. The purpose is to allow fast storage and retrieval of lightly compressed or uncompressed data. For example, the postprocessor may implement a fast algorithm like LZW, LZ77, or dictionary decoding.

2. Syntax

A ZPAQ level 1 stream shall have the syntax described in this section. In this description, the notation "X ::= Y Z" means that symbol X is composed of Y followed by Z. Terminal symbols (those not expanded further) are single bytes with range (0...255) inclusive. Symbols in parenthesis are nonterminal. The notation X[0...n-1] means an array of X of n elements, individually X[0]...X[n-1]. When X is used as a number, it means an n-byte number in base 256, least significant byte (LSB) first, in the range $(0...256^n-1)$. The notation X[0...] means X repeated 0 or more times. A string enclosed in double quotes is interpreted as a sequence of ASCII bytes, e.g. "zPQ" means 122 80 81. The notation X=n means that X is a symbolic constant with value n (in 0...255). The notation (X | Y) means either X or Y.

ZPAQ ::= (block)[0...]

block ::= "zPQ" level(1...2) HPROG=1 hsize[0..1] (header) (segment)[0...] EOB=255

(Also, header must be hsize bytes long.)

header ::= hh hm ph pm n(0...255) (comp)[0...n-1] END=0 (hcomp) END=0

comp[i] ::= (

CONST=1 c

| CM=2 sizebits(0...32) limit

| ICM=3 sizebits(0...26)

| MATCH=4 sizebits(0...32) bufbits(0...32)

| AVG=5 j(0...i-1) k(0...i-1) wt

| MIX2=6 sizebits(0...32) j(0...i-1) k(0...i-1) rate mask

| MIX=7 sizebits(0...32) j(0...i-1) m(1...i-j) rate mask

| ISSE=8 sizebits(0...26) j(0...i-1)

| SSE=9 sizebits(0...32) j(0...i-1) start limit)

segment ::= 1 (filename) 0 (comment) 0 RESERVED=0 (ecd) eos

filename ::= c(1...255)[0...]

comment ::= c(1...255)[0...]

(Also, filename and comment must each be 0..65535 bytes in length)

ecd ::= c[0...] 0 0 0 0

(Also, if n > 0 then there are never more than 3 consecutive bytes of c = 0)

eos ::= (254 | 253 sha1[0...19])

(*where sha1 is the SHA1 hash [2] of the decompressed segment*)

If *n* = 0 then *ecd* (entropy coded data) is stored as a sequence of uncompressed blocks, each with a 4 byte header giving the length of the block as a 4 byte number, LSB first, and at least 1. It has the format:

ecd ::= (elen[0..3](1...2^{32}-1) edata[0...elen-1])[0...]

If *n* > 0, then *ecd* is decoded using the context mixing model described by (*comp*), with contexts computed by the program (*hcomp*), which is executed once for each decoded byte with that byte as input. In either case, the decoded data (*dd*) for each block has the format:

dd ::= (PASS=0 output[0...] | PROG=1 plen[0...1] (pcomp) pdata[0...])

(Also, pcomp must be plen bytes long.)

If *dd*[0] = PASS then output[0...] is output to the destination specified by (*filename*) in each segment header. Otherwise, the program (*pcomp*) is run once for each byte of *pdata* with that byte as input. The output of the program is output to the destination specified by (*filename*). Both (*hcomp*) and (*pcomp*) have the same syntax:

pcomp ::= (opcode)[0...]

hcomp ::= (opcode)[0...]

Opcodes are one to three bytes. Valid opcodes are shown in Table 1 in section 6. A program may have invalid opcodes as long as they are not executed.

A ZPAQ level 2 compliant stream shall conform to this syntax. In addition, a stream where either program does not halt, or executes ERROR or any reserved instruction, or where the program counter goes outside the range of the program is not compliant.

A ZPAQ level 1 compliant stream is identical except that it requires that *level* = 1 and that $n > 0$. Thus the *comp* section must not be empty and *ecd* must be arithmetic coded (not uncompressed). All level 1 compliant streams are also level 2 compliant.

3. Decoding

A ZPAQ decoder has the following state information:

- A virtual machine HCOMP as described by (*hcomp*), containing the externally accessible context array H[0...2^{hh}-1] with each element in (0...2^{32}-1), initially 0.
- An array of n (1...255) components, COMP[0...n-1] as described by (*comp*), such that COMP[i] takes context hash H[i] (and also the output of COMP[0...i-1]) as input.
- An array of n predictions, P[0...n-1], where P[i] in (-2^{11}...2^{11}-1) is the output of COMP[i]. P[i] represents a belief by COMP[i] that the next bit will be a 1 with probability $1/(1 + e^{-P[i] / 64})$.
- A partially or fully decoded byte C8 in (1...511).
- An arithmetic decoder (section 4).
- POST, a post-processor (section 5).

Throughout this specification, we will use the convention that array indexes are modulo the array size. Thus if *hh* = 8, then H[256] = H[0].

Let the function predict(COMP[i]) assign a prediction to P[i]. The function update(COMP[i], Y) updates the state of the component with decoded bit Y (0...1) in a way intended to reduce future prediction errors. The function decode(p) returns Y from the arithmetic coder given a probability p in [0,1), as described in section 4. The function write(POST, C) writes byte C (0...255) to the post-processor (section 5). The overall structure of the decompression algorithm is:

decompress() =

    For each block do
        For i in (0...n-1) initialize COMP[i](*comp[i]*)
        Initialize HCOMP(*hh, hm)*
        Initialize P[0...255] := **0**, C8 := 256
        Initialize POST(*ph, pm*)
        For each segment
            Select output, depending on *filename*
            Initialize ZRLE/arithmetic decoder (sec. 4)
            If (n > 0) then

```
                While decodeBit(0) = 0 do
                    C8 := 1
                    While C8 < 256 do
                        For i in (0...n-1) do
                            P[i] := predict(COMP[i], P[0...i-1], H[i], C8)
                        Y := decodeBit(squash(P[n-1]) + 0.5) / 2^15) (Y in 0...1)
                        For i in (0...n-1) do
                            update(COMP[i], P[i], Y)
                        C8 := C8 * 2 + Y
                    write(POST, C8 - 256) (sec. 5)
                    H := run(HCOMP, hcomp, C8 - 256) (sec. 6)
            else (n = 0, level 2 or higher only)
                While (C8 := decodeByte()) != EOS do
                    write(POST, C8)
            write(POST, 2^32-1) (end of segment)
```

Define squash() as follows. squash() is the approximate inverse of stretch(). It is not exact due to integer roundoff. The exact definitions are:

squash(x) = floor(32768 / (1 + $e^{-x/64}$)), x in (-2048...2047)

stretch(x) = round(64 * ln((x + 0.5)/(32767.5 - x))), x in (0...32767)

where round(x) = floor(x + 1/2), and floor(x) is the largest integer not greater than x.

If correctly implemented these functions should satisfy the following computations:

$\sum_{i=0...32767} 3^i$ stretch(i) = 3887533746 (mod 2^{32})

$\sum_{i=0...4095} 3^i$ squash(i-2048) = 2278286169 (mod 2^{32})

squash($\geq$ 666) = 32767 stretch(32767) = 710

squash(0) = 16384 stretch(16384) = 0

squash($\leq$ -666) = 0 stretch(0) = -710

P[] thus represents stretched probabilities. Squash(P[i]) is in (0...32767) and represents the belief by COMP[i] that the next bit will be a 1 with probability (squash(P[i]) + 0.5) / 32768.

The following functions are defined for each component:

- initialize(COMP[i]) sets the initial state at the start of a block.
- predict(COMP[i], H[i], P[0...i-1], C8) writes a prediction to P[i].
- update(COMP[i], P[0...i-1], Y) modifies the state of COMP[i] to reduce the prediction error for the decoded bit Y.

The (comp) instructions are as follows (with unused parameters omitted):

3.1. CONST c

There is no state to initialize or update. The prediction is P[i] := (c - 128) * 4

3.2. CM sizebits limit

A context model uses a table CM to map a context into a prediction. When updated, it adjusts the table entry to reduce the prediction error. To control the learning rate, it counts predictions in each context in a table CMCOUNT. Initialize:

SIZE := $2^{sizebits}$

CM[0...SIZE-1] := $\mathbf{2^{21}}$ (a 22 bit probability in (0...2^{22}-1)

CMCOUNT[0...SIZE-1] := **0** (count, range 0...1023)

predict(COMP[i], H[i], C8) =

CXT := H[i] XOR hmap4(C8)

P[i] := stretch(floor(CM[CXT] / 2^7))

update(COMP[i], Y) =

train(i, CM[CXT], CMCOUNT[CXT], limit * 4)

The train() function updates the prediction in CM[CXT] to reduce the prediction error in inverse proportion to CMCOUNT[CXT], then updates the count up to LIMIT. The function is also used by other models:

train(i, T, TCOUNT, LIMIT) =

ERROR := Y * 32767 - floor(T / 2^7)

T := T + floor(ERROR * floor(2^{16} / (TCOUNT + 1.5)) / 2^9)

TCOUNT := min(LIMIT, TCOUNT + 1)

hmap4() is a function intended to improve cache locality on 64 byte aligned arrays.

hmap4(C8) =

If (C8 < 16) then return C8

Else if (C8 < 2^{5+0}) then return floor(C8 / 2^0) * 16 + (C8 mod 2^0) + 2^0

Else if (C8 < 2^{5+1}) then return floor(C8 / 2^1) * 16 + (C8 mod 2^1) + 2^1

Else if (C8 < 2^{5+2}) then return floor(C8 / 2^2) * 16 + (C8 mod 2^2) + 2^2

Else if (C8 < 2^{5+3}) then return floor(C8 / 2^3) * 16 + (C8 mod 2^3) + 2^3

hmap4() has the effect of splitting the partially decoded byte into two 4-bit nibbles. After the first nibble is fully decoded, it occupies bits 7...4 of the output with bit 8 set to 1.

0000xxxx -> 00000xxxx

0001xxxx -> 1xxxx0001

001xxxxx -> 1xxxx001x

01xxxxxx -> 1xxxx01xx

1xxxxxxx -> 1xxxx1xxx

3.3. ICM sizebits

An indirect context model uses a hash table HT to map a context to a bit history, and then a direct lookup table CM to map the history to a probability. When a bit is decoded, the history is updated to reflect the new bit, and the history map is adjusted to reduce the prediction error. Initialize:

SIZE := 4 * $2^{sizebits}$ (size of context map)

HT[0...SIZE-1][0...15] := **0** (checksum and 15 histories, all in 0...255)

CM[BH in (0...254)] := cminit(BH) (history map, initial probability of 1 in 0...2^{23}-1).

The next bit is predicted by computing a hash index HI from H[i] and C8 and looking up in the history BH in the hash table HT. The low bits of HI are used as the index, and the next higher 8 bits are used as a checksum to detect (most) hash collisions. If a hash confirmation is not found among 3 adjacent elements, then the lowest priority element is replaced. Then HT is mapped to P[i] through CM.

predict(COMP[i], H[i], C8) =

If C8 = 1 or C8 in (16...31) then HI := find(HT, H[i] + 16 * C8) (first index into HT)

BI := hmap4(C8) (mod 16) (second index in HT, in 1...15)

BH := HT[HI][BI] (bit history)

P[i] := stretch(floor(CM[BH] / 2^8)).

find(HT, CXT) finds the hash index for CXT, replacing an element in HT if needed. It is defined as:

find(HT, CXT) =

CHK := floor(CXT / SIZE) (mod 256) (checksum for hash confirmation)

H0 := CXT mod SIZE (hash index)

H1 := H0 XOR 1 (candidate locations)

H2 := H0 XOR 2

If HT[H0][0] = CHK then return H0

Else if HT[H1][0] = CHK then return H1

Else if HT[H2][0] = CHK then return H2

Else if HT[H0][1] ≤ HT[H1][1] and HT[H0][1] ≤ HT[H2][1] then

HT[H0] := (CHK, **0**[1...15]**)**, return H0

Else if HT[H1][1] < HT[H2][1] then

HT[H1] := (CHK, **0**[1...15]), return H1

Else CM[H2] := (CHK, **0**[1...15]), return H2.

HT uses element 0 of each row as a confirmation checksum and element 1 as a priority. Element 1 represents the bit history for a context ending on a 4 bit boundary. This is the history updated most frequently in the row, because the other histories are for contexts that are 1 to 3 bits longer. The histories are represented by states sorted by increasing n0+ n1, the sum of the 0 and 1 bit counts. Thus, the cache replacement policy is roughly LFU (least frequently used).

When a bit Y is decoded, the bit history is updated in CM, HM[BH] is adjusted to reduce the prediction error.

update(COMP[i], Y) =

HT[HI][BI] := next(BH, Y)

ERROR := Y * 32767 - floor(CM[BH] / 256)

CM[BH] := CM[BH] + floor(ERROR / 4).

A bit-history has the form of an ordered triple, (N0, N1, LB). N0 and N1 represent counts of the 0 and 1 bits Y that have been used to update BH. LB represents the last update bit, either 0 or 1, or 1/2 indicating that the last bit is not stored. The values N0, N1, and LB are restricted to allow 255 possible values of BH according to the following rules:

- If N0 > N1 then BH is allowed only if inverse(BH) is allowed, where inverse(N0, N1, LB) = (N1, N0, 1-LB).
- If N0 + N1 is in (1...17) then LB is in (0...1) else LB = 1/2.
- If N0 = 0 then N1 is in (0...20).
- If N0 = 1 then N1 is in (0...48).
- If N0 = 2 then N1 is in (0...15).
- If N0 = 3 then N1 is in (0...8).
- If N0 = 4 then N1 is in (0...6).
- If N0 = 5 then N1 is in (0...5).

The function next(BH, Y) updates the bit history by appending bit Y but keeping BH within the allowed set of values by discarding counts as needed. In most cases this is done by discarding part of the opposite count, e. g. reducing N1 if Y = 0. It is defined:

next(BH = (N0, N1, LB), Y) =

If N0 < N1 then return inverse(next(inverse(BH), 1-Y))

Else if BH = (20, 0, 1/2) and Y = 0 then return (20, 0, 1/2)

Else if BH = (48, 1, 1/2) and Y = 0 then return (48, 1, 1/2)

Else if BH = (15, 2, *) and Y = 0 then return (8, 1, 0) (* means 0 or 1)

Else if BH = (8, 3, *) and Y = 0 then reutrn (6, 2, 0)

Else if BH = (8, 3, *) and Y = 1 then return (5, 3, 1)

Else if BH = (6, 4, *) and Y = 0 then return (5, 3, 0)

Else if BH = (5, 5, *) and Y = 0 then return (5, 4, 0)

Else if BH = (5, 5, *) and Y = 1 then return (4, 5, 1)

Else if Y = 1 then return bound_LH(discount(N0), N1+1, 1)

Else (Y = 0) return bound_LH(N0+1, discount(N1), 0).

discount(N) =

If N > 7 then return 7

Else if N > 5 then return N - 1

Else return N.

bound_LH(BH = (N0, N1, LH)) =

If N0 + N1 > 17 then return (N0, N1, 1/2)

Else return BH.

In the function find(HT, CXT), the meaning of ≤ when comparing bit histories is to compare their priorities defined by the function:

priority(BH = (N0, N1, LB)) = N0 * 128 + N1 * 130 + LB.

The priority function defines a strict ordering over bit histories by increasing total count, breaking ties by N1, then by LB. Bit histories may be sorted by priority and mapped to numbers in the range (0...254) when used as an index into the array CM. In HT, the meaning of 0 is the initial state (0, 0, 1/2), which has the lowest priority.

The function cminit(BH) returns 2^{23} times the estimated probability that the next update will be a 1 in state BH:

cminit(BH = (N0, N1, LB)) = floor(2^{22} * (N1 * 2 + 1) / (N0 + N1 + 1)).

3.4. MATCH sizebits bufbits

A match model finds the most recent context match in an output buffer and predicts the next bit as a function of the length of the match. The match is maintained until a bit mismatch is found. On each byte boundary, if there is no current match then it looks up the context in the index to find a new one. On each byte boundary it updates the output buffer and the index. Initialize:

SIZE := $2^{sizebits}$

INDEX[0...SIZE-1] := **0**

OFFSET := 0 (distance back to match)

LEN := 0 (length of match in bytes, up to 255)

BUF[0...$2^{bufbits}$-1] := **0** (decoded data buffer in BUF[0...POS-1](0...255))

POS := 0 (number of decoded bytes)

BP := 0 (number of decoded bits after last full byte, 0...7)

predict(COMP[i]) =

If LEN = 0 then P[i] := 0

Else

BIT := floor(BUF[POS - OFFSET] / 2^{7-BP}) (mod 2) (predicted bit)

If BIT = 1 then P[i] := stretch(32768 - floor(2048 / LEN))

Else P[i] := stretch(floor(2048 / LEN))

update(COMP[i], Y) =

If BIT ≠ Y then LEN := 0

BUF[POS] := BUF[POS] * 2 + Y (mod 256)

BP := BP + 1

If BP = 8 then (a byte was fully decoded)

POS := POS + 1

BP := 0

If LEN = 0 then (look for a match)

OFFSET := POS - INDEX[H[h]]

If OFFSET ≠ 0 (mod BUFSIZE) then

While LEN < 255

and BUF[POS - LEN - 1] = BUF[POS - LEN - OFFSET - 1]

LEN := LEN + 1

Else if (LEN < 255) then LEN := LEN + 1

INDEX[H[i]] = POS

3.5. AVG j k wt

There is no state to initialize or update.

predict(P[0...i-1]) =

P[i] := floor((P[j] * wt + P[k] * (256-wt)) / 256).

3.6. MIX sizebits j m rate mask

A MIX adaptively combines m predictions by weighted averaging, where the weights are selected by a context. After a bit is decoded, the weights are adjusted to favor the most accurate models. Initialize:

SIZE := $2^{sizebits}$

WT[0...SIZE-1][0...m-1] := **floor(2^{16}/m)**

The output prediction is a weighted sum of inputs P[j...j+m-1]

predict(COMP[i], H[i], P[0...i-1], C8) =

CXT := H[i] + (C8 AND mask)

P[i] := clamp2k(floor(($\sum_{k \text{ in } (j...j+m-1)}$ floor(WT[CXT][k] * P[k] / 256) / 256))

where clamp2k(x) bounds x to a 12 bit signed integer:

clamp32k(x) = min(2047, max(-2048, x)).

After decoding, the weights are adjusted to favor the most accurate input models for the given context.

update(COMP[i], P[0...i-1], Y) =

ERROR := floor(Y * 32767 - squash(P[i])) * rate / 16)

For k in (j...j+m-1) do

WT[CXT][k] := clamp512k(WT[CXT][k] + round(ERROR * P[k] / 2^{13}))

where clamp512k(x) = min(2^{19}-1, max(-2^{19}, x)) clamps x to a 20 bit signed integer.

3.7. MIX2 sizebits j k rate mask

A MIX2 is a MIX with m = 2 inputs, P[j] and P[k] instead of P[j...j+m-1]. Additionally, the weights are constrained to add to 1. We may represent a MIX2 using a single weight per context. Initialize:

SIZE := 2^{sizebits}

WT[0...SIZE-1] := $\mathbf{2^{15}}$

predict(COMP[i], H[i], P[0...i-1], C8) =

CXT := H[i] + (C8 AND mask)

P[i] = floor((P[j] * WT[CXT] + P[k] * (65536 - WT[CXT])) / 65536)

update(COMP[i], P[0...i-1], Y) =

ERROR := floor((Y * 32767 - squash(P[i])) * rate / 32)

WT[CXT] := min(65535, max(0, WT[CXT] + round(ERROR * (P[j] - P[k]) / 2^{13})))

3.8. ISSE sizebits j

An indirect secondary symbol estimator maps a context to a bit history (like ICM), which is then used as the context for a 2 input MIX with independent weights and one input fixed. The MIX takes P[j] and constant 64 as inputs. The weights are initialized to 2^{15} for P[j] and an initial guess based on CMINIT (sec. 3.3) for the constant. The learning rate for P[j] is fixed. Initialize:

SIZE := 4 * 2^{sizebits} (size of hash table)

HT[0...SIZE-1][0...15] := **0** (checksum and 15 histories, all in 0...255)

WT[0...254][0] := $\mathbf{2^{15}}$ (input for P[j], range -2^{19}...2^{19}-1)

WT[0...254][1] := clamp512k(stretch(floor(cminit(0...254) / 2^{8})) * 2^{10})

predict(COMP[i], H[i], P[0...i-1], C8) =

If C8 = 0 or C8 in (16...31) then HI := find(HT, H[i] + 16 * C8) (first index into HT)

BI := hmap4(C8) (mod 16) (second index in HT, in 1...15)

BH := HT[HI][BI] (bit history in 0...254)

P[i] := clamp2k(floor((WT[BH][0] * P[q] + WT[BH][1] * 64) / 2^{16}))

update(COMP[i], P[0...i-1], Y) =

HT[HI][BI] := next(BH, Y)

ERROR := Y * 32767 - squash(P[i])

WT[BH][0] := clamp512k(WT[BH][0] + round(ERROR * P[i] / 2^{13}))

WT[BH][1] := clamp512k(WT[BH][1] + round(ERROR / 2^5))

The functions cminit() and next() are defined in section 3.3.

3.9. SSE sizebits j start limit

A secondary symbol estimator (SSE) takes an input prediction P[j] quantized to 32 levels and a context H[i] and outputs a new prediction. The prediction is interpolated between the two nearest quantized value. The closer of those two points is then updated. The table SM is initialized to output the same prediction as the input for all contexts. Each element is associated with a count SMCOUNT in (start...limit*4) that determines the update rate. Initialize:

SIZE := $2^{sizebits}$

For k in (0...31) do

SM[0...SIZE-1][k] := squash(k*64 - 992) * 2^7

SMCOUNT[0...SIZE-1][k] := start.

predict(COMP[i]) =

CXT := H[i] + BUF[POS]

PQ := min(1983, max(0, (P[j] + 992)))

W := PQ (mod 64) (interpolation weight)

PQ := floor(PQ / 64) (quantized to 0...30)

P[i] := stretch(floor((SM[CXT][PQ]*(64-W) + SM[CXT][PQ+1]*W) / 2^{13}))

If W ≥ 32 then PQ := PQ + 1.

When bit Y is decoded, the prediction is adjusted to reduce the prediction error in inverse proportion to its count, and the count is incremented to a maximum of limit*4.

update(COMP, Y) =

train(i, SM[CXT][PQ], SMCOUNT[CXT][PQ], limit*4) (section 3.2).

4. Arithmetic Decoder

When there are *n* = 0 components (permitted only if level ≥ 2), the data is stored in uncompressed blocks, each preceded by the block length as a 4 byte number and terminated by a block of size 0. When *n* > 0, the data is arithmetic coded as described in section 3. The arithmetic decoder receives bit predictions (PR = squash(P[n-1]) + 0.5)/32768, and the compressed input stream and outputs uncompressed bits, Y. The end of segment is decoded with PR = 0. All other bits are decoded with PR equal to odd multiple of 1/65536 between 0 and 1.

The decoder state is initialized:

LOW := 1 (in 1...2^{32}-1)

HIGH := 2^{32}-1 (in 0...2^{32}-1, HIGH > LOW)

CURR := 0

Do 4 times: CURR := CURR * 256 + next_byte(*ecd*).

decodeBit(PR) returns a bit as follows:

PR := PR * 2^{16} (an integer in 0...65535)

MID := LOW + floor((HIGH - LOW) * PR / 2^{16})

If CURR ≤ MID then Y := 1, HIGH := MID

else Y := 0, LOW := MID + 1.

While floor(LOW / 2^{24}) = floor(HIGH / 2^{24}) do

LOW := LOW * 256 (mod 2^{32})

If LOW = 0 then LOW := 1

HIGH := HIGH * 256 + 255 (mod 2^{32})

CURR := CURR * 256 + next_byte(*ecd*) (mod 2^{32})

Return Y.

next_byte() reads one byte of the compressed data, *ecd*. When decoding ends, next_byte() will have read the 4 trailing 0 bytes so that LOW > 0, CURR = 0, HIGH = 2^{31}-1. At all other times, LOW ≤ CURR ≤ HIGH and LOW < HIGH. The next byte to read would be EOS.

When *n* = 0, decodeByte() returns a byte or EOS as follows:

If CURR = 0 then return EOS

Else

C := next_byte(*ecd*)

CURR := CURR -1

If CURR = 0 then

Do 4 times: CURR := CURR * 256 + next_byte(*ecd*) (mod 2^{32})

Return C

5. Post Processing

Recall that a block decoded as described in sections 3 and 4 has the following syntax:

dd ::= (PASS=0 output[0...] | PROG=1 plen[0...1] (pcomp) pdata[0...])

This data is written to a post-processor in the decoding algorithm in section 3 by calling write(POST, C) for each byte C in *dd*. POST has the following state:

PCOMP, a virtual machine, initialized PCOMP(*ph, pm*)

PBUF, an input buffer string, initialized to ""

write(POST, C) =

If PBUF = "" then append C to PBUF

Else if PBUF[0] = PASS then output C

Else if |PBUF| < 3 or |PBUF| < *plen* + 3 then append C to PBUF (*where plen = PBUF[1..2])*

Else run(PCOMP, *pcomp*, C). *(run program pcomp (in PBUF[3...plen+2]) with input C)*

When the first byte of *dd* is PROG, the output of run() is the output of the decompresser.

6. ZPAQL

There are up to 2 ZPAQL virtual machines: HCOMP in the bit prediction model, and PCOMP in the post-processor. A machine COMP is initialized (at the beginning of a block):

COMP(*hbits, mbits*) =

PC := 0 (program counter)

A, B, C, D := 0 (general purpose registers in $0...2^{32}-1$)

F := 0 (condition flag in 0...1)

H[$0...2^{hbits}-1$] (memory, each element in $(0...2^{32}-1)$, initialized to **0**. In HCOMP,

H[i] is the input to COMP[i])

M[$0...2^{mbits}-1$] (memory, each element in (0...255) initialized to **0**)

R[0...255] (memory, each element in $(0...2^{32}-1)$, initialized to **0**).

A program is executed by calling run(COMP, *prog*, *input*), where *prog* is a string of opcodes as in table 1, and *input* is an input in $(0...2^{32}-1)$.

run(COMP, prog, input) =

PC := 0

A := input

prog := (prog 0 0) (append two 0 bytes (ERROR opcodes))

Do forever

If PC not in (0...|prog|-3) then exit with an error

If prog[PC] = 255 then OPCODE = prog[PC...PC+2] (LJ long jump opcode)

Else if prog[PC] = 7 (mod 8) then OPCODE := prog[PC...PC+1], PC := PC + 2

Else OPCODE := prog[PC], PC := PC + 1

If OPCODE = ERROR or is undefined then exit with an error

Else if OPCODE = HALT then return

Else execute(OPCODE).

Opcode	0	1	2	3	4	5	6	7
0	ERROR	A++	A--	A!	A=0			A=R N
8	B<>A	B++	B--	B!	B=0			B=R N
16	C<>A	C++	C--	C!	C=0			C=R N
24	D<>A	D++	D--	D!	D=0			D=R N
32	*B<>A	*B++	*B--	*B!	*B=0			JT N
40	*C<>A	*C++	*C--	*C!	*C=0			JF N
48	*D<>A	*D++	*D--	*D!	*D=0			R=A N
56	HALT	OUT		HASH	HASHD			JMP N
64	A=A	A=B	A=C	A=D	A=*B	A=*C	A=*D	A= N
72	B=A	B=B	B=C	B=D	B=*B	B=*C	B=*D	B= N
80	C=A	C=B	C=C	C=D	C=*B	C=*C	C=*D	C= N
88	D=A	D=B	D=C	D=D	D=*B	D=*C	D=*D	D= N
96	*B=A	*B=B	*B=C	*B=D	*B=*B	*B=*C	*B=*D	*B= N
104	*C=A	*C=B	*C=C	*C=D	*C=*B	*C=*C	*C=*D	*C= N
112	*D=A	*D=B	*D=C	*D=D	*D=*B	*D=*C	*D=*D	*D= N
120								
128	A+=A	A+=B	A+=C	A+=D	A+=*B	A+=*C	A+=*D	A+= N
136	A-=A	A-=B	A-=C	A-=D	A-=*B	A-=*C	A-=*D	A-= N
144	A*=A	A*=B	A*=C	A*=D	A*=*B	A*=*C	A*=*D	A*= N
152	A/=A	A/=B	A/=C	A/=D	A/=*B	A/=*C	A/=*D	A/= N
160	A%=A	A%=B	A%=C	A%=D	A%=*B	A%=*C	A%=*D	A%= N
168	A&=A	A&=B	A&=C	A&=D	A&=*B	A&=*C	A&=*D	A&= N
176	A&~A	A&~B	A&~C	A&~D	A&~*B	A&~*C	A&~*D	A&~ N
184	A|=A	A|=B	A|=C	A|=D	A|=*B	A|=*C	A|=*D	A|= N
192	A^=A	A^=B	A^=C	A^=D	A^=*B	A^=*C	A^=*D	A^= N
200	A<<=A	A<<=B	A<<=C	A<<=D	A<<=*B	A<<=*C	A<<=*D	A<<= N
208	A>>=A	A>>=B	A>>=C	A>>=D	A>>=*B	A>>=*C	A>>=*D	A>>= N
216	A==A	A==B	A==C	A==D	A==*B	A==*C	A==*D	A== N
224	A<A	A<B	A<C	A<D	A<*B	A<*C	A<*D	A< N
232	A>A	A>B	A>C	A>D	A>*B	A>*C	A>*D	A> N
240								
248								LJ N M

Table 1. ZPAQL opcodes

Note that the state of COMP is retained between runs except for A and PC. Opcodes are given in Table 1. The numeric value is the row number plus the column number. Opcodes in column 7 are two bytes where the second byte is N in (0...255). Opcode 255 (LJ) is 3 bytes.

The meaning of execute(OPCODE) is as follows. Most opcodes have the form "X op Y" where X and Y are one of A, B, C, D, *B, *C, *D, or N. A, B, C, and D are 32 bit registers with values in $(0...2^{32}-1)$. N is a number in (0...255), the second byte of a 2 byte opcode. *B means M[B]. *C means M[C]. *D means H[D]. Operations on *B and *C are modulo 256. Operations on A, B, C, D, and *D are modulo 2^{32}. As usual, indexes into M and H are modulo 2^{mbits} and 2^{hbits} respectively. Operations are as follows:

- ERROR causes the decompresser to fail (for debugging). It is equivalent to any undefined instruction, except that it is not reserved for future use.
- X++ means add 1 to X. (Note that *B++ increments *B, not B).
- X-- means subtract 1 from X.
- X! means X := -1-X (complement all bits).
- X=0 means set X := 0. (This is a 1 byte opcode. It is equivalent to the 2 byte opcode X=N when N is 0).
- X<>A means swap X with A. If X is *B or *C then only the low 8 bits of A are changed.
- X=R N means X := R[N].
- R=A N means R[N] := A.
- JT N (jump if true) means if F = 1 then add ((N+128) mod 256) - 128 to PC. This is a conditional jump in the range (-128...127) relative to the next instruction, e.g. JT 0 has no effect, 1...127 jumps forward and 128...255 jumps backwards from the next instruction.
- JF N (jump if false) means if F = 0 then add ((N+128) mod 256) - 128 to PC.
- JMP N means add ((N+128) mod 256) - 128 to PC (regardless of F).
- LJ N M (long jump) means PC := N + 256 * M, where N and M are in (0...255). This is the only 3 byte instruction.
- HALT terminates execution and returns to the calling algorithm.
- OUT means to output A. In PCOMP, A (mod 256) is written to output. In HCOMP it has no effect.
- HASH means A := (A + *B + 512) * 773 (a useful byte hashing function for HCOMP).
- HASHD means *D := (*D + A + 512) * 773.
- X=Y assigns X := Y.
- A+=Y adds A := A + Y.
- A-=Y subtracts A := A - Y.
- A*=Y multiplies A := A * Y.
- A/=Y divides: if Y > 0 then A := A / Y else A := 0.
- A%=Y: if Y > 0 then A := A (mod Y), else A := 0.

- A&=Y computes A := A AND Y, which clears any bit in the binary representation of A if the corresponding bit of Y is 0.
- A&~Y computes A := A AND NOT Y, which clears any bit in A that is set in Y.
- A|=Y computes A := A OR Y, which sets any bit in A that is set in Y.
- A^=Y computes A := A XOR Y, which complements any bit in A that is set in Y.
- A<<=Y (left shift): A := A * $2^{Y \bmod 32}$
- A>>=Y (right shift): A := floor(A / $2^{Y \bmod 32}$)
- A==Y (equals): If A = Y then F := 1 else F := 0.
- A<Y (less than): If A < Y then F := 1 else F := 0.
- A>Y (greater than): If A > Y then F := 1 else F := 0.

7. Compliance

A program that accepts any data that conforms to the requirements in sections 2 through 6 in this document is ZPAQ level 2 compliant. There is no requirement for a compliant program to behave in any particular way for any non conforming data. There is no requirement that a compressor that produces ZPAQ level 2 data shall support all of the features described. However, it is the responsibility of the compressor to produce compliant data.

A decompresser might not have enough memory to decompress a compliant stream. A decompresser is said to be compliant up to its memory limit if it will accept all streams that require less memory. The memory requirement may be computed from information in the block header. It is $4*2^{hh} + 2^{hm} + 4*2^{ph} + 2^{pm}$ bytes plus the following by component in (*comp*), in bytes, where SIZE = $2^{sizebits}$.

- For each CM, 4*SIZE
- For each ICM, 64*SIZE + 1024
- For each MATCH, 4*SIZE + $2^{bufbits}$
- For each MIX2, 2*SIZE
- For each MIX, 4*SIZE*m
- For each ISSE, 64*SIZE + 2048
- For each SSE, 128*SIZE.

A block header begins with "zPQ" followed by LEVEL=1 or LEVEL=2 to indicate the compression level supported. Future versions will use (3...127) in increasing order. Each level *L* shall support reading all levels in the range (1...L). Levels (128...255) are reserved for private use and are not part of this or any future standard. Likewise, HPROG=1, PROG=1, and RESERVED=0 shall not use values in the range (128...255) in future versions for the same reason. HPROG and PROG are intended to indicate the language used in the (*hcomp*) and (*pcomp*) sections, respectively.

LEVEL=0 is experimental. Different versions of level 0 programs are not required to be compatible with each other or with level 1.

When a block or sequence of blocks is embedded in a stream of non-ZPAQ data, it is recommended that the first block be preceded by the following 13 byte string to help locate it:

37 6B 53 74 A0 31 83 D3 8C B2 28 B0 D3 (in hexadecimal).

The value was chosen randomly. In addition, non-ZPAQ data following a block sequence, if any, should not begin with "zPQ" (7A 50 51 hex). A stream may contain more than one block sequence. A locator tag is useful for marking the start of the data in a self extracting archiver.

8. Archive Format

This section describes streaming and journaling archives and indexes. A streaming archive is a simple collection of files. A journaling archive is designed to be append-only and to store multiple dated versions of files for incremental backup with deduplication and rollback capability. An index is a journaling archive with only file metadata.

In streaming format, file names are stored in the filename field. An empty field indicates a continuation of the previous file. If the filename field of the first segment of the first block is empty, then no file name is specified. The comment fields of all segments must not end with the 4 bytes “jDC\x01”.

In journaling format, all blocks contain exactly one segment. The comment field always has the form “size jDC\x01” where “size” is the uncompressed size as a decimal string in range (“0”..”4294967295”) (or $0..2^{32}$-1) followed by a single space, and “\x01” is a single character with ASCII value 1. An archive may not contain both streaming and journaling blocks. In journaling format, only blocks with a first segment filename field in the following 28 character format and lexicographically greater than the previous filename are allowed:

“jDC” year[4](1900..2999) month[2](01..12) day[2](01..31)
hour[2](00..59) minute[2](00..59) second[2](00..59)
type(c,d,h,i) n[10](0000000001..4294967295)

for example, “jDC19991231235959c0123456789”. Characters 3..16 give the transaction date in universal (UTC) time, which must be in the past. The uncompressed contents of each block has the following byte syntax according to the type field in character 17:

c: csize[8]

d: fragment[f][fsize[n..n+f-1]] (0[4]|n[4]) 0[4] | (fsize[n..n+f-1][4] (0[4]|n[4]) f[4])

h: bsize[4] (sha1[20] fsize[4])[n..n+f-1]

i: (0[8] name[] 0 | date[8] name[] 0 na[4](0..65535) attr[na] ni[4] ptr[ni][4])[]

where *n* matches decimal characters 18..27 of the filename. Numbers are LSB first. *fsize* must be in the range $(0..2^{31}-1)$. *name* must be (0..65535) bytes long.

In type *c* blocks, *csize* is the total compressed size of the type *d* block sequence that follows in the current transaction. A value in $(2^{63}..2^{64}-1)$ indicates the end of the archive. All blocks that follow should be ignored. Otherwise, advancing the archive file pointer by *csize* should advance to the next block that is not type *d* or to end of file, whichever comes first.

Type *d* blocks contain a sequence of *f* variable sized fragments with IDs consecutively numbered *n*..*n*+*f*-1. No two fragments anywhere in the archive may have the same ID. No ID may be less than *n* given in the preceding type *c* block with the same date. No ID may be 0. The fragment size list *fsize* is optional (to allow recovery of damaged *h* blocks). If omitted, *f*[4] is 0. *n*[4] may either be 0 or match *n* from the filename.

There is one type *h* block for each type *d* block. Both blocks in the pair have the same *n*. In a type *h*

block, *bsize* gives the compressed size of the corresponding *d* block. This is followed by a list of the SHA-1 hashes and sizes of the fragments of that *d* block. The *fsize* list of sizes, if present in the *d* block, must agree in both blocks.

Type *i* blocks describe edits to the central filename index as processed in order from the beginning of the archive. An edit may either be an update (new or existing file) or a deletion. A deletion is indicated by a *date* field of 0. Deleted files are not extracted. Otherwise, *date*, when converted to decimal has the 14 digit format YYYYMMDDHHMMSS giving the last modified date (UTC) in the same format as in the filename field. The *name* field is the filename in UTF-8 format, using "/" as the path separator character. A list of *na* attribute bytes follow, and then a list of *ni* fragment IDs. A file is extracted by concatenating the corresponding fragments, which must appear earlier in the archive.

The *attr* string may be of any length. The meaning is defined if it has one of the following prefixes:

"w" windows_attr[4]

"u" unix_attr[2]

windows_attr is interpreted as returned by the Windows function **GetFileAttributes()** defined in **windows.h**. *unix_attr* is interpreted as type **mode_t** as defined in **sys/stat.h** and returned in the **mode** field of the Unix/Linux function **stat()**.

An index is a journaling archive with no *d* blocks. Thus, *csize* must be 0 except to mark the end of the archive.

9. Encrypted Format

An archive of length n bytes may be encrypted with AES-256 [4] in CTR mode using a 32 byte salt prefix as follows:

salt[0..31] (archive[i=32..n+31] xor AES256(key, salt[0..7] (i/16)[7..0]))

where the 32 byte key is derived from a SHA-256 [3] hash of a password and the salt using Scrypt [5] as follows:

key = Scrypt(SHA256(password), salt[0..31], N=16384, r=8, p=1)

The password string must contain at least one byte and not contain any NUL (0) bytes.

To decrypt, the 32 byte salt prefix of the n + 32 byte ciphertext is removed. Then the remaining n bytes is XORed with a keystream E(IV+2), E(IV+3), E(IV+4)..., where E() means encryption with AES-256, and IV is a 16 byte number in big-endian (MSB first) format consisting of the first 8 bytes of the salt followed by 8 zero bytes.

The following 61 byte encrypted string:

d3 77 41 c5 6a a0 f9 cf 64 11 97 2e 27 0e 7a ae

09 b1 9e 87 ef b8 82 79 3e 18 a1 13 e6 1d f5 3f

a1 41 75 c0 b5 1b 9b ef 16 83 a6 bd c3 1d 34 9e

08 84 27 ee 77 7f ed 95 25 1c cf 26 a5

should decrypt to the following 29 byte minimal archive with 3 byte password "abc":

7a 50 51 02 01 07 00 00 00 00 00 00 00 00 01 00

00 00 00 00 00 01 00 00 00 00 00 fe ff

Notes:

The format is designed to provide fast random access without decrypting the whole archive.

No authentication is provided. An attacker with write access who knows or can guess any bits of the plaintext can arbitrarily set those bits without knowing the password or key.

Scrypt is designed to slow down password guessing attacks. It requires 128Nr bytes = 16 MiB memory and 1536Nrp = 3 x 2^{26} 32-bit operations (add, xor, rotate) to compute the key from the password. It uses repeated rounds of salsa20/8 [7] and a final step of HMAC-SHA256 [8].

The salt is not secret but should be chosen randomly to make it unlikely that two archives will use the same salt. If two archives are encrypted with the same password and salt then the XOR of the encrypted archives will reveal the XOR of the two plaintexts without knowing the password or key. Also, a random salt makes it impractical to generate password cracking tables in advance.

The minimal archive in the test vector above should decompress to an empty string. It encodes one level-2 block with no tag, no compression model, no post-processor, and one segment with no filename, comment, or checksum.

10. Implementation Notes

This document does not specify a compression algorithm. However, it will generally be the case that the models (on the input side of the arithmetic decoder) will be identical for compression and decompression. For the arithmetic coder, range splitting and normalization would be identical, except that when the high bits of the range are shifted out, they are written to the output. Immediately after coding EOS, the compressor should write an end of segment marker (0 0 0 0). It is not necessary to flush the encoder.

It is the responsibility of the compressor to ensure that preprocessing is exactly reversed by post-processing. The recommended way to do this is to test during compression by running both transforms and comparing with the original.

Memory requirements for a typical decompresser implementation can be calculated almost entirely from information in the block headers. Compression typically requires at least as much memory as decompression.

The design is optimized for arrays aligned on 64 byte cache line boundaries (in particular, ICM and ISSE).

The following is a recommendation for choosing fragment boundaries to support deduplication compatibility in journaling archives. Fragments should not span file boundaries. Fragment sizes should otherwise be in the range 4096..520192. A fragment not of maximum size or ending at end of file should have, and no prefix of length 4096 or longer should have, a 32 bit hash of 65535 or less, where the hash of *n* byte string x[0..n-1] (each byte in 0..255) is defined as follows:

hash("") = 0 (empty string).

hash(x[0..n-1]) = (hash(x[0..n-2]) + x[n-1] + 1) * 314159265 (mod 2^{32}) if x[n-1] is predicted.

hash(x[0..n-1]) = (hash(x[0..n-2]) + x[n-1] + 1) * 271828182 (mod 2^{32}) if x[n-1] otherwise.

where x[n-1] is predicted (by an order 1 model) if the last occurrence of the value of x[n-2] in x[0..n-3] is followed by the same value as x[n-1], or if either n < 3 or x[n-2] does not occur in x[0..n-3] and x[n-1] is 0. For example, in the string "THIS IS A TEST", the second "S" and second space are

predicted.

An archive may be either a single file or a concatenation of multiple files matching a common filename pattern and numbered consecutively starting with 1, for example *part01.zpaq, part02.zpaq*... . A multi-part archive is encrypted with a single keystream with the salt in the first 32 bytes of part 1.

Optionally, a multi-part archive may have an index with part number 0 (e. g. *part00.zpaq*). If it is present, then the archive must be in journaling format with no streaming blocks, and each part must consist of exactly one update (i. e. with exactly one *c* block, which must be the first block). The index must be a copy of the concatenation of the other parts in numerical order except that the *d* blocks are omitted and the *c* blocks have a *csize* value of 0. If the archive is encrypted, then the index is encrypted separately with the same password and the same salt except that salt[0] is replaced with salt[0] XOR 0x4D (ASCII 'M').

11. Intellectual Property

I (Matt Mahoney) am not aware of any patents protecting any of the techniques needed to fully implement a compression or decompression algorithm or product according to this specification. I have not filed for patents on any of the techniques described here and will not do so.

12. Revision History

Mar. 12, 2009. Original level 1 final specification released.

Sept. 29, 2009. Revision 1 adds a recommendation in section 7 for embedding in non-ZPAQ data.

Feb. 1, 2012. Level 2 standard released. Allows the context model to be omitted (n = 0). Also in the definition of find() in section 3.3, the following line changes "≤" to "<" to conform to the level 1 and 2 reference implementations.

Else if HT[H1][1] < HT[H2][1] then

Sept. 28, 2012. Level 2 revision 1 inserts section 8 "Archive Format", incorporating some material from section 7.

June 3, 2013. Level 2 revision 2 makes journaling "d" block redundant fragment lists and fragment ID optional in section 8. Adds a recommendation for fragmentation to section 9.

Jan. 16, 2014. Level 2 revision 3 inserts section 9, "Encrypted Format". Sections 9-11 are renumbered 10-12. Adds references 3..8.

Nov. 18, 2014. Level 2 revision 4 updates section 10 with a recommendation for multi-part archives and indexes. It corrects the hash constant "271827182" to "271828182" to conform to existing implementations.

Mar. 15, 2016. Level 2 revision 5 updates the archive format description. Mixed streaming and journaling formats are not allowed. Limits sizes of streaming filename and comment fields, journaling file names, attribute strings, fragments, and uncompressed blocks.

References

1. M. Mahoney, PAQ data compression programs. 2000-2008. http://cs.fit.edu/~mmahoney/compression/

2. D. Eastlake, P. Jones, RFC 3174, US Secure Hash Algorithm 1, 2001, http://www.faqs.org/rfcs/rfc3174.html

3. FIPS 180-4, Secure Hash Standard (SHS), 2012, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

4. FIPS 197, Advanced Encryption Standard (AES), 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

5. C. Percival, Stronger Key Derivation via Sequential Memory-Hard Functions, Proc. BSDCan'09, 2009. http://www.tarsnap.com/scrypt/scrypt.pdf

6. B. Kalinski, RFC-2898, Password-Based Cryptography Specification, Version 2.0, 2000, http://www.ietf.org/rfc/rfc2898.txt

7. D. J. Bernstein, The Salsa20 Core, 2005, http://cr.yp.to/salsa20.html

8. H. Krawczyk, M. Bellare, R. Canetti, HMAC: Keyed Hashing for Message Authentication, RFC-2104, 1997. http://tools.ietf.org/html/rfc2104