The ZPAQ Compression Algorithm

Matt Mahoney

Dec. 29, 2015

Abstract

ZPAQ is a tool for creating compressed archives and encrypted user-level incremental backups with rollback capability. It deduplicates any new or modified files by splitting them into fragments along content-dependent boundaries and comparing their cryptographic hashes to previously stored fragments. Unmatched fragments are grouped by file type and packed into blocks and either stored or compressed independently in multiple threads using LZ77, BWT, or context mixing in a self-describing format depending on the user selected compression level and an analysis of the input data. Speed and compression ratio compare favorably with other archivers.

Introduction

ZPAQ [1] is a tool for producing compressed archives and user-level incremental backups. Figure 1 compares ZPAQ with some popular archivers and backup utilities at default and maximum compression settings on on the 10GB corpus [2], a set of 79K files in 4K directories.

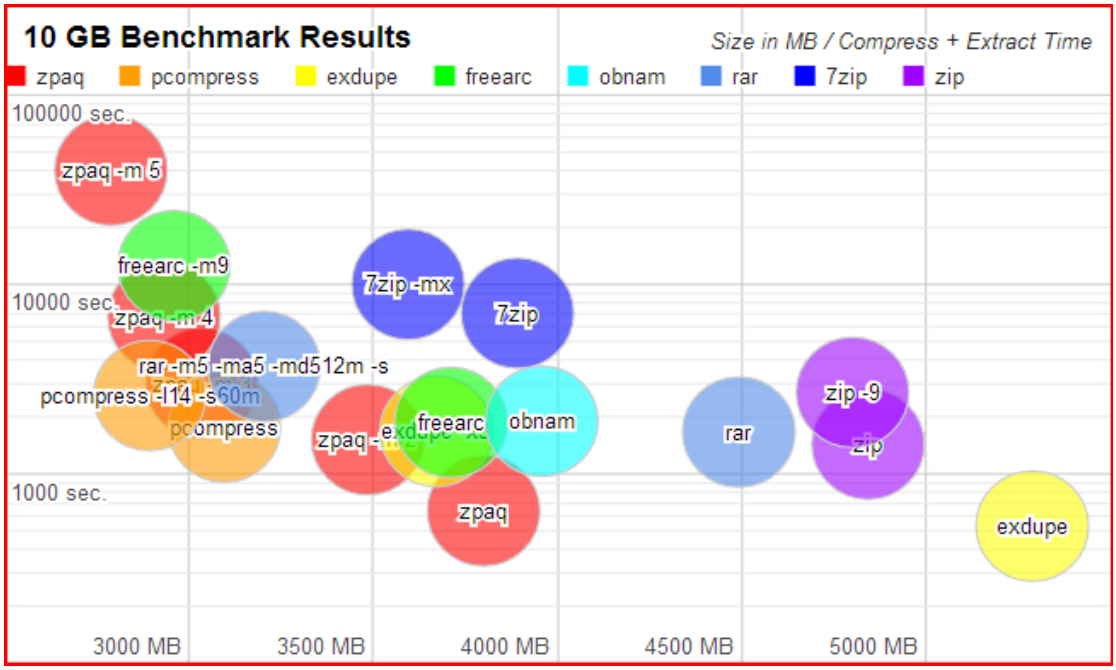

Figure 1. 10 GB compression speed and size at default and maximum settings.

Compression plus decompression times are in real seconds for 10 GB on a 2.67 GHz Core i7 M620, 2+2 hyperthreads, 4 GB memory under Ubuntu Linux with the archive on an external USB drive. The 10 GB corpus consists of 79K files in 4K directories with a wide range of file types. It includes about 2 GB of already compressed files (ZIP, JPEG, etc), and 1.5 GB of duplicate data to simulate realistic backup scenarios.

ZIP [3], 7ZIP [4], RAR [5], and FREEARC [6] are primarily archivers. The compression algorithms are

optimized for fast decompression, typically some variant of LZ77 [7]. They do not support deduplication or rollback. ZIP compresses each file separately. The others have a “solid” mode which compresses better but requires decompressing all files in order to just extract one or to update. ZIP and RAR support incremental updates based on changes to file last-modified dates.

ZPAQ, PCOMPRESS [8], EXDUPE [9], and OBNAM [10] are primarily backup utilities. The compression algorithms are optimized for fast compression rather than decompression. All of them support deduplication by storing cryptographic hashes of file fragments split along content dependent boundaries. If two fragments have the same hash then they are assumed to be identical and stored only once. All but PCOMPRESS support incremental updates. PCOMPRESS archives cannot be updated at all. PCOMPRESS and OBNAM run only in Unix or Linux. OBNAM produces a large directory tree rather than a single archive file. ZPAQ and OBNAM can be reverted to an earlier state to extract old versions of files that have since been modified. ZPAQ has one of its 5 compression levels (-method 2) with fast decompression to support its use as an archiver.

PCOMPRESS has the best overall compression performance in the middle range, and ZPAQ in the low and high range. Table 1 shows the data from Figure 1 with separate compression and decompression times. Times with a * are on the Pareto frontier (no program is faster and compresses better).

Table 1. 10GB corpus compression and decompression times in real seconds.

Size	Compress	Extract	Program	Options
2788126729	20291*	19898*	zpaq 6.51	-m 5
2893742274	1756*	838*	pcompress 3.1	-l14 -s60
2932527615	3428	3354	zpaq 6.51	-m 4
2959337875	11676	905	freearc 0.666-win32	-m9
3035559935	1689*	1313	zpaq 6.51	-m 3
3096159495	1154*	610*	pcompress 3.1	-l6
3204716369	3454	238*	rar 5.04b	-m5 -ma5 -md512m -s -r
3485028038	1282	234*	zpaq 6.51	-m 2
3595106502	9580	446	7zip 9.20	-mx
3671183744	1138	538	exdupe 0.5.0b	-x3
3711048040	1448	422	freearc 0.666-win32	
3800332723	382*	256	zpaq 6.51	-m 1
3892353718	6953	472	7zip 9.20	
3958233075	1337	560	obnam 1.1	--compress-with=deflate
4166697886	552	226*	exdupe 0.5.0b	-x2
4493860462	1423	239	rar 5.00b7	
4802080309	2450	245	zip 3.00	-9
4844449350	1206	229	zip 3.00	
5290752762	513	250	exdupe 0.5.0b	-x1
10065018880	408	368	tar 1.26	

ZPAQ Archive Format

The ZPAQ archive format is specified precisely from the viewpoint of decompression by a specification [11] and a reference decoder [12]. ZPAQ uses a self describing format: the decompression algorithm is described in the archive using a virtual machine code that is executed when files are extracted. This allows older versions of ZPAQ to decompress archives created by newer versions using improved compression algorithms.

ZPAQ supports two archive formats: streaming and journaling. A streaming archive compresses or

decompresses byte sequences in a single pass. The archive may contain file names and attributes to support decompression to a set of files. A journaling archive contains blocks of compressed fragments and additional metadata to support deduplication. It is append-only to support rollback. An update searches for files whose last-modified date has changed and then compares fragment SHA-1 [13] cryptographic hashes with those stored in the archive. Unmatched fragments are grouped by file type, packed into blocks, compressed, and appended to the archive, followed by a list of fragment hashes and sizes and a list of added and deleted files. Each update is preceded by a small block indicating the current date and time and a pointer to skip over the compressed block for fast access to the metadata. An archive can be reverted to an earlier state by truncating it at the first future date.

All blocks headers (streaming or journaling) describe the decompression format as a byte code in the ZPAQL language. The description has three parts: a tree of context models, a virtual machine for computing contexts, and an optional virtual machine to post-process the decompressed output. The context model tree may be empty, in which case the data in the archive is passed directly to the post-processor. Otherwise the tree computes a sequence of single bit probabilities which are then arithmetic decoded. On x86 hardware, ZPAQ first translates the ZPAQL byte code into x86 32-bit or 64-bit instructions and executes it directly. This approximately doubles compression and decompression speed. On other hardware, the byte code is interpreted.

ZPAQ uses arithmetic coding. The code for an n-bit string $x = x_{1..n}$ is a base-256 big-endian (most significant byte first) fractional number in the probability range LOW..HIGH = P(< x)..P(≤ x) in 0..1, where P(< x) means the probability of all strings lexicographically less than x. By the chain rule, $P(x) = \prod_{i=1..n} P(x_i \mid x_{1..i-1})$, the product of the conditional bit predictions.

LOW and HIGH are infinitely long byte strings divided into three parts. The leading digits are identical between LOW and HIGH and are written to the archive. Four middle digits are kept in memory as 32-bit integers. The trailing digits are implicitly assumed to be 0 in LOW and 255 in HIGH. After each bit prediction, the range LOW..HIGH is split into two parts in proportion to $P(x_i = 0)$ and $P(x_i = 1)$. The bit is coded by replacing the range with the corresponding half. If this results in the leading digits matching, then those digits are written to the archive. Decoding consists of making the same bit predictions and range splits, then reading from the archive to determine which half was selected.

A byte is compressed by coding a 0 bit with probability $1 - 2^{-15}$ and then coding the 8 bits MSB first using a context model. EOF is marked by coding a 1 bit. To find the end of the compressed data without decompressing, the end is also marked with four 0 bytes. To ensure that the arithmetic coder never outputs four consecutive 0 bytes, the part of LOW in memory is always set to at least 1.

Uncompressed data is stored as a series of blocks preceded by their size, up to $2^{32} - 1$. The end is marked with a 0 sized block. ZPAQ normally uses 64K blocks.

Context Models

ZPAQ context models are based on the PAQ architecture [14][15]. The data is predicted one bit at a time, and the probabilities are passed to an arithmetic coder or decoder. A model consists of a linear tree of components, where each component inputs a context and possibly the probabilities of earlier components and outputs a new probability. The final component at the root of the tree outputs the bit probability to the arithmetic coder. The component types are as follows:

CONST - outputs a fixed probability between 0 and 1.
CM - context model, a table that maps a context to a probability.

AVG - fixed weighted average of two probabilities in the logistic domain, log(p/(1-p)).
MIX2 - context sensitive, adaptive average of two probabilities.
MIX - generalizes a MIX2 to a range of components.
SSE - secondary symbol estimator: adaptive, context sensitive adjustment of a probability.
ICM - indirect context model, maps a context to a bit history and then to a probability.
ISSE - indirect SSE, adjusts by mixing a weighted constant selected by a bit history.
MATCH - predicts whatever bit followed the last context match.

Contexts are computed by a ZPAQL program which is called after each byte is coded. The computed context is combined with the individual bits already coded in the current byte (starting with the most significant bit). A ZPAQL virtual machine has the following state:

Four 32-bit registers: A, B, C, D (accumulator and three index registers).
A 1-bit condition flag: F.
256 extra 32-bit registers: R0..R255.
A byte array M, indexed by the B and C registers.
A 32-bit integer array H, indexed by D.

The sizes of M and H (up to 2^{32}) are specified as powers of 2 in the ZPAQL byte code. The last modeled byte is passed in A. The output contexts are placed in H. Most of the arithmetic and logic instructions are a one-byte opcode possibly with a one byte immediate operand. Most instructions operate on the A register and one other operand and leave the result in A. A ZPAQL program is limited to 64K bytes of code. There is no stack or CALL instruction.

If there is a post-processor, then a separate virtual machine is called once for each decompressed byte and EOF (-1) passed in A. Output is by the OUT instruction. Otherwise the decompressed bytes are output directly. The post-processor code is prefixed to the block before compression. The first decompressed byte indicates whether a post-processor is present or not. If so, then the compressor is responsible for computing the inverse transform prior to modeling. It is possible to have uncompressed code with post-processing only by specifying an empty context model tree.

The following is an example of the ZPAQL source code generated by compressing BOOK1 from the Calgary corpus [16] with the advanced option `-method x0c0.0.255.255i4`. The compressed size is 231 KB (compared to 312 KB for zip).

```
comp 9 16 0 0 2
  0 icm 14
  1 isse 14 0
hcomp
  c-- *c=a a+= 255 d=a *d=c
  d= 0 *d=0 b=c a=*b hashd b++ a=*b hashd
  d= 0 b=c a=*d d++ hash b++ hash b++ hash b++ hash *d=a
  halt
end
```

The meaning of the `-method` argument is as follows:
`x` - use journaling mode (the default). s would select streaming mode.
`0` - Block size 2^0 = 1 MB.

`c0` - ICM. c1...c256 would select a CM with different learning rates.
`.0` - no special contexts.
`.255.255` - bit masks selecting an order 2 context.
`i4` - chained ISSE with an order 4 context.

The `comp` line of the generated ZPAQL code selects the size of H and M as 2^9 and 2^{16} elements respectively. The remaining values would select the post-processor array sizes if it were present, and the number of components. The components are numbered starting at 0 for readability. The arguments to the ICM and ISSE select the table sizes (2^{14} 64-byte elements, equal to the block size). The ISSE gets its input from component 0 (the ICM).

The first line in the HCOMP section is standard in ZPAQ generated code. It uses M as a rotating 64K history buffer, written backward. C points to the last byte written. H[255..510] record the position (in C) of the last occurrence of bytes 0..255. These values would be used if special contexts were specified. The instructions *C and *D refer to elements of M and H respectively (modulo their size). The state is initialized to all zeros and retains its value between calls.

The second line computes the ICM context and puts it in H[0]. The `hashd` instruction computes *D = (*D + A + 1) * 773. The hash multiplication can be implemented efficiently using shift and add instructions, or x86 LEA instructions.

The third line computes the ISSE context hash. It retrieves the hash from H[0], combines it with the last 4 input bytes in M[C..C+3], and stores it in H[1]. The `hash` instruction computes A = (A + *B + 1) * 773. *B points to M.

The `halt` instruction returns. If there were a post-processor, it would be specified in a `pcomp` section, followed by more ZPAQL code before the `end`. ZPAQL source code is free format, case insensitive, and may contain comments. However, two-byte instructions like `a+= 255` require an explicit space to separate the opcode and operand.

During compression, ZPAQ would compile the byte code, preprocess the data, prefix the `pcomp` section to it, compile the remaining byte code to x86, compress the data, and write the byte code to the block header.

CM - Context Model
A CM maps a context to a 22 bit probability *p* initially representing 0.5, and a 10 bit counter *n*, initially 0, which determines the rate at which *p* is updated. After a bit (0 or 1) is coded, the selected state is updated:

error = bit - p.
p := p + error / (n + 1.5).
n := min(n + 1, limit).

The limit can range up to 1020. A higher limit is better for stationary data. A lower limit adapts quickly to changing statistics.

A CM can take either a direct context or a hash. Only the low bits are used, depending on the table size. The byte-level context computed by the ZPAQL code is combined with the partially coded byte by

expanding the already coded bits to 9 bits and XOR-ing it with the computed context. The 9 bit expansion is an optimization designed to produce only two cache misses on 64 byte cache lines per byte as follows, where x represents the already coded bits.

000000001 (cache miss)
00000001x
0000001xx
000001xxx
1xxxx0001 (cache miss)
1xxxx001x
1xxxx01xx
1xxxx1xxx

The ZPAQ advanced options always compute a context hash and select a memory size large enough to avoid collisions until the table memory usage is the same as the block size (at 4 bytes per context). Table 2 shows the compressed size of BOOK1 from the Calgary corpus at various model orders, counter limits, and component sizes. The optimal order is found to be 3. Text is generally stationary, so large counter limits do better. Adding memory helps until collisions are eliminated. In the `-method` argument, `cn` means a limit of 4n - 4.

Table 2. BOOK1 compression with a single CM.

order	limit	memory	BOOK1	-method
			768771	(uncompressed)
			312507	(zip -9)
0	64	2K	437083	s0c17
0	256	2K	434768	s0c65
0	1020	2K	434706	s0c256
1	256	128K	345898	s0c65.0.255
2	256	1M	291005	s0c65.0.255.255
3	256	1M	296848	s0c65.0.255.255.255
3	256	16M	259899	s4c65.0.255.255.255
3	256	256M	**253669**	s8c65.0.255.255.255
4	256	256M	278446	s8c65.0.255.255.255.255

Mixers
AVG, MIX2, and MIX components compute a weighted average of the predictions of other components in the logistic domain as follows:

$\text{squash}(x) = 1/(1 + e^x)$.
$\text{stretch}(x) = \ln(x/(1 - x)) = \text{squash}^{-1}(x)$.
$\text{AVG}(x_1, x_2, w) = \text{MIX2}(x_1, x_2, w) = \text{squash}(w\ \text{stretch}(x_1) + (1 - w)\ \text{stretch}(x_2))$.
$\text{MIX}(x_{1..n}) = \text{squash}(\sum_{i=1..n} w_i\ \text{stretch}(x_i))$.

In an AVG, the weight w is fixed. In a MIX or MIX2, the weights are adjusted to reduce the prediction error. In a MIX, the weights are not constrained to add to 1 and can range from -8 to 8 in steps of 2^{-16} (as a 20 bit signed integer). Initially all weights are 1/n. After a MIX or MIX2 predicts that the next bit will be 1 with probability p, it updates the weights as follows:

error = bit - p.
w := w + L * error * stretch(x).

where L is the learning rate, typically around 0.01. ZPAQ supports values between 2^{-12} and 2^{-4}. ZPAQ represents probabilities as odd multiples of 2^{-16} and stretched probabilities as 12 bit signed integers between -32 and 32 in steps of 1/64.

A MIX2 or MIX can take a context to select a set of weights, which sometimes improves compression. The computed context is combined with the bit level context by adding the extra bits with a leading 1. (There is no cache optimization). Table 3 shows that mixing an order 2 and order 4 context improves compression over order 2, 3, or 4 alone, even when the total memory is constrained to be the same (16 MB or 2 x 8 MB). The method t*n.L* or m*n.L* means a MIX2 or MIX respectively with *n* context bits and learning rate $L/2^{12}$. `t0.0` simply averages the last two components with fixed weights of 1/2. `t0.24` allows the weights to adapt at rate $L = 0.0059$. `m0.24` mixes all previous components instead of the last two. The difference here is that the weights are no longer constrained to add to 1. Finally `m8.24` and `m16.24` select the pair of weights by an order 0 or 1 context, respectively. Each of these changes improves compression.

Table 3. Effect of mixing contexts.

order	BOOK1	-method
2	288637	s4c65.0.255.255
3	259900	s4c65.0.255.255.255
4	289931	s4c65.0.255.255.255.255
2,4	241457	s3c65.0.255.255c65.0.255.255.255.255t0.0
2,4	240295	s3c65.0.255.255c65.0.255.255.255.255t0.24
2,4	236230	s3c65.0.255.255c65.0.255.255.255.255m0.24
2,4	231797	s3c65.0.255.255c65.0.255.255.255.255m8.24
2,4	**230060**	s3c65.0.255.255c65.0.255.255.255.255m16.24

SSE - Secondary Symbol Estimator
A SSE adjusts the probability of another component by looking up the prediction and a context in a table and outputting a new prediction. Then it adjusts the table entry to reduce the prediction error. To keep the table size reasonable, the input prediction is stretched, quantized to 32 values between -8 and 8 in steps of 0.5, and interpolated. Each table entry contains a 22 bit prediction *p* and a 10 bit count *n* like a CM. After the prediction, the closer of the two interpolated entries is updated as follows:

error = bit - p.
p := p + error/(n + 1.5).
n := min(n + 1, limit).

The initial value of p is equal to the input prediction. The initial n is selectable between 0..255 and is typically larger than 0 (like 32) to prevent large initial swings in p. The limit can range from 4 to 1020. Larger values are better for stationary sources like text.

Table 4 shows the effect of appending an SSE to some of the models from table 3. The option *sC.M.N* means to adjust the previous component using C bits of context (computed as with a MIX or MIX2), an

initial count of *M* and a limit of 4*N*. The defaults are `s8.32.255`.

Table 4. Effects of SSE.

order	BOOK1	-method
2	288637	s4c65.0.255.255
2,1	285294	s4c65.0.255.255s8.32.255
2,2	283729	s4c65.0.255.255s16
3	259900	s4c65.0.255.255.255
3,2	250487	s4c65.0.255.255.255s16
2,4,2	230060	s3c65.0.255.255c65.0.255.255.255.255m16.24
2,4,2,1	227042	s3c65.0.255.255c65.0.255.255.255.255m16.24s8
2,4,2,2	227617	s3c65.0.255.255c65.0.255.255.255.255m16.24s16
2,4,2,1,2	**225737**	s3c65.0.255.255c65.0.255.255.255.255m16.24s8s16

Indirect Context Modeling

An indirect context model (ICM) maps a context to a bit history and then to a prediction. It solves the problem of whether to use a stationary or fast adapting model by learning which strategy is more successful. For example, given a 10 bit sequence in some context like 0000000001, a stationary model would predict p(1) = 0.1, but an adaptive model would give greater weight to the most recent bit. An ICM would just look up the sequence and learn how it behaves.

An ICM uses two tables. The first is a large hash table mapping context hashes to bit histories represented as 8 bit states. The second table is a CM with a fixed adaption rate of 2^{-10} instead of a counter.

A bit history represents (N0, N1, LB), where N0 and N1 are counts of previously seen zeros and ones, and LB is the last bit. The initial state is (0, 0, -) (no last bit). Each state initially maps to a prediction of (N1 + 0.5)/(N0 + N1 + 1). Since a bit history is represented by one byte, the maximum allowed counts for N0 and N1 are (0, 20), (1, 48), (2, 15), (3, 8), (4, 6), (5, 5), (6, 4), (8, 3), (15, 2), (48, 1), (20, 0). In addition, the last bit is not stored if N0 + N1 > 17 or if either counter is 0. If either counter reaches its limit, then both counts are reduced to keep the ratio about the same, except that the opposite count is limited to 7 and counts over 5 are decremented. For example, in state (1, 46), the sequence 11100000 would result in states (1, 47), (1, 48), (1, 48), (2, 7, 0), (3, 6, 0), (4, 5, 0), (5, 5, 0), (5, 4, 0), (6, 4, 0). Experimentally, reducing the opposite count this way improves compression.

The hash table is designed to limit cache misses to one per lookup, with 2 lookups per byte. The table maps a context ending on a nibble boundary to a 16 byte array containing a 8-bit checksum to detect most (but not all) collisions, and 15 bit histories for each of the possible 0 to 3 bit combinations after the nibble boundary. A lookup uses the low bits of the context hash to index the table, and the next higher 8 bits as a checksum. It tests 3 adjacent slots in the same 64 byte cache line by testing the index XOR-ed with 0, 1, and 2. If it cannot find a matching checksum or an unused slot (indicated by state (0, 0) in the first bit history), then it discards the slot with the lowest N0 + N1 in the first bit history.

On BOOK1, a single CM compresses better than an ICM for low order contexts, but an ICM is better at higher orders. The method `c0` selects an ICM, with the other arguments the same as a CM.

Table 5. Comparison of CM and ICM.

order	BOOK1	-method
2 CM	288637	s4c65.0.255.255
2 ICM	292539	s4c0.0.255.255
3 CM	259900	s4c65.0.255.255.255
3 ICM	260009	s4c0.0.255.255.255
4 CM	289931	s4c65.0.255.255.255.255
4 ICM	278861	s4c0.0.255.255.255.255
2,4,1 CM	231797	s3c65.0.255.255c65.0.255.255.255.255m8.24
2,4,1 ICM	226848	s3c0.0.255.255c0.0.255.255.255.255m8.24
2,4,1,1,2	**221300**	s3c0.0.255.255c0.0.255.255.255.255m8.24s8s16

Indirect SSE

An ISSE modifies a prediction like an SSE, except that it uses a bit history (like an ICM) and a linear adjustment in the logistic domain rather than an interpolated table. It maps a context hash to a bit history, which is used to select a pair of weights for mixing the input prediction with a constant. Given an input prediction p, the output is p' = squash(w_1 stretch(p) + w_2), where w_1 and w_2 are selected by a bit history. The initial weights are 1 and 0, which results in no change to p. After the bit is coded, the weights are adjusted:

error = bit - p'.
L = 1/256.
w_1 := w_1 + L * error * stretch(p).
w_2 := w_2 + L * error * 4.

An ISSE is better suited for higher order contexts than an SSE because it uses 1 byte of memory per context instead of 128 bytes. A typical use is to chain them onto a CM or ICM with each link increasing the context order. The method `i`n_1`.`n_2`.`n_3... defines a chain in which the first ISSE context is an order n_1 context hash combined with the context of the previous component from which it takes its prediction, and the remaining n_2, n_3... are additional ISSEs that increase the context order by that amount.

Table 6. Comparison of ICM-MIX with ICM-ISSE.

order	BOOK1	-method
2,4,1 ICM	226848	s3c0.0.255.255c0.0.255.255.255.255m8.24
2,4 ISSE	226521	s3c0.0.255.255i4
1,2,3,4	**220312**	s2c0.0.255i2.1.1

Match

A match model searches for a previous match to the current context and predicts whatever bit came next with probability 1 - 1/2n, where n is the length of the match in whole bytes. If there is no match found, then the prediction is 1/2 and the model searches for another match by looking up the context in a hash table after the next byte boundary. For a block size of N, method `a`*`M.B.H`* (default `a24.0.0`) computes a context hash of h("") = 0, h($x_{1..n}$) = Mh($x_{1..n-1}$) + x_n + 1 (mod 2^{-H-2} N) on byte string x, and

uses a buffer size of 2^{-B} N. The hash table uses 4 bytes per index and is updated after every byte with no collision detection. For example, if the block size is 8 MB, then a MATCH with default parameters would use a buffer size of 8 MB and a 21 bit hash, also using 8 MB memory. A hash multiplier of M = 24 = 3 x 2^3 would shift 3 bits left, effectively computing an order 7 context. Table 7 shows the effects of mixing an order 2 ICM and various match orders on BOOK1. The last two entries show the effect of adding an order 7 MATCH to an order 0-1-2-3-4 ICM-ISSE chain.

Table 7.

order	BOOK1	-method
2	292235	s3c0.0.255.255
2,23,1	291134	s3c0.0.255.255a6m
2,12,1	283821	s3c0.0.255.255a12m
2,7,1	272452	s3c0.0.255.255a24m
2,6,1	267375	s3c0.0.255.255a48m
2,4,1	270098	s3c0.0.255.255a192m
0,1,2,3,4	219952	s2ci1.1.1.1
0,1,2,3,4,1	218243	s2ci1.1.1.1m
0,1,2,3,4,7,1	**215780**	s2ci1.1.1.1am

Word Models

A word model is an ICM-ISSE chain in which the ZPAQL code computes whole word contexts to improve compression of text files. A method `wL.A.Z.U.M.S` (default `w1.65.26.223.20.0`) creates a chain of length L and considers a word to be a sequences of bytes in the range A..A+Z-1 after AND-ing each byte with U (to convert to upper case). Each component is allocated memory equal to the block size N times 2^{-S}, which gives it a context of $\log_2(N) + 2 - S$ bits. The hash is computed as in a MATCH with multiplier M. Thus, for the default M = 20, which shifts 2 bits left, the hash depends on the last $\log_2(N) + 2 - S) / 2$ characters in the word. Table 8 shows the effect on an order 0..4 ICM-ISSE chain replacing a MATCH with various word models.

Table 8. Comparison of word models.

order	BOOK1	-method
MATCH order 7	215780	s2ci1.1.1.1am
Non-whitespace words	212461	s2ci1.1.1.1w1.33.94.255.20.0m
Case sensitive letters	212281	s2ci1.1.1.1w1.65.62.255.20.0m
Case insensitive	212181	s2ci1.1.1.1w1.65.26.223.20.0m
Order 0-1 ICM-ISSE chain	**208734**	s2ci1.1.1.1w2m

The ZPAQ advanced options can also generate code to compute sparse (non-contiguous) and periodic contexts that are useful for modeling structured data such as tables, databases, images, and numeric samples such as audio. Table 9 shows the effect of various sparse models on PIC from the Calgary corpus. PIC is a FAX image of a page from a textbook with one bit per pixel (MSB first) and 216 bytes per scan line (left to right).

Table 9. Sparse methods for PIC

order	PIC	-method
	513216	(uncompressed)
	52605	(zip -9)
1 (left)	53251	sc0.0.255
1 (above)	32301	sc0.0.1215.255
2 (left)	56525	sc0.0.255.255
2 (left, above)	33674	sc0.0.255.1214.255
1.5 (left, above)	**30784**	sc0.0.15.1214.255

A value N over 1000 is shorthand for N - 1000 zeros. The option `sc0.0.15.1214.255` means streaming (s), ICM (c0), no periodic models (0), just the last 4 bits of the byte to the left (15), skip the next 214 bytes (1214), and the 8 bits of the byte above (255).

The file GEO in the Calgary corpus is a sequence of 32 bit (4 byte) IBM formatted floating point numbers representing seismic samples with at most 16 bits of precision. The periodic structure means that the offset mod 4 is a useful context because it distinguishes the exponent and the high, middle, and low bytes of the mantissa, which have different statistics. In particular, the low byte is always 0. The option `c0.4` means an ICM that includes the offset mod 4 as part of the context hash.

order	GEO	-method
	102400	(uncompressed)
	68636	(zip -9)
0	72540	sc0.0
0 periodic	52088	sc0.4
1	57181	sc0.0.255
1 periodic	**51830**	sc0.4.255

Transforms
ZPAQ can generate ZPAQL post-processor code to decode E8E9, BWT, and LZ77 in either a bit-packed format for direct storage, or byte-aligned for further modeling. When used, ZPAQ performs the corresponding transform during compression.

E8E9 Transform
The E8E9 transform improves the compression of x86 code (.EXE and .DLL files, or Linux executables) by replacing the 32-byte little-endian addresses of the CALL and JMP opcodes (E8 and E9 hex) with absolute addresses to improve compression. Specifically, ZPAQ scans a block backward searching for 5 byte patterns of the form ([E8,E9] xx xx xx [00,FF]) hex and adds the block offset to the middle 3 bytes, modulo 2^{24}. The post-processor does the reverse transform scanning forward. The reason for testing the last byte is to reduce false detections, because offsets are typically less than 16 MB except in huge executables.

Burrows-Wheeler Transform (BWT)
ZPAQ uses libdivsufsort [17] to compute the forward BWT [18], and generates ZPAQL code to invert it. The BWT sorts the block by its right context, treating the end of the block as a character -1. Because

this symbol has no representation as a byte, ZPAQ substitutes a different value (255) and appends its location to the end in big-endian format.

The transformed data can be compressed efficiently using a low order, fast adapting model. ZPAQ normally uses an order 0-1 ICM-ISSE chain. For example, `-method s0.3ci1` compresses BOOK1 to 215333 bytes. The method selects streaming mode (s), 1 MB block size (0), BWT (3), an order 0 ICM (c), and a chained order 1 ISSE (i1).

In the inverse implementation, ZPAQ stores the transformed block in M. At the end of the data, it builds a linked list in H and then traverses it to output the original data. It uses the last 256 elements of H for an array of counters to build the list. Thus, the BWT block size must be at least 256 bytes smaller than a power of 2. As an optimization, for blocks of 16 MB or smaller, it first copies M into the upper 8 bits of H so that during the list traversal the number of cache misses is reduced from 2 to 1 per byte. The forward and reverse transforms each require 5 times the block size in memory.

LZ77 Transform

LZ77 (Limpel-Ziv 1977) [7] replaces duplicate strings with pointers to an earlier occurrence. Decompression is simple and fast, although the compression ratio is typically not as good as BWT or context modeling. Compression ratio depends on how much time is spent finding matching substrings.

ZPAQ uses two LZ77 formats. In one, codes for literals, literal lengths, match offsets, and match offsets have variable bit lengths, intended to be packed into bytes and stored with no further compression. In the other, the codes are whole bytes, which compresses worse by itself but can be modeled for overall better (but slower) compression.

In either case, ZPAQ can use either one or two hash tables or a suffix array to find matches. When two hash tables are used, the tables represent different orders and the higher order is searched first, and if a match is found, the second search is skipped. In either case, the longest match is selected, breaking ties by picking the closer match. There is a look-ahead mode in which case ZPAQ considers extending a literal in order to find a better match that starts later.

The hash tables do not use collision detection. A table is divided into buckets whose size depend on the search depth. ZPAQ searches every index in the bucket and picks the best match. After each byte is coded, a bucket is selected by a rolling hash of the current location in the block, and one element of the bucket is chosen pseudo-randomly. The chosen location is overwritten with a pointer to the current location.

ZPAQ can also use a suffix array (computed by libdivsufsort) to find matches. A suffix array (SA) is a lexicographical ordering of the starting locations of all suffixes of the block. Thus, the longest matches are at adjacent locations in the SA. Since LZ77 doesn't allow future matches, ZPAQ searches in both directions from the current location in the SA for the first match in the past. Because the matches are sorted longest to shortest, only two matches need to be tested.

A suffix array for a block of size N requires 4N memory. The algorithm also needs an inverse suffix array (ISA) to map the current location to the corresponding location in the SA. The ISA also requires 4N memory. To save memory, ZPAQ divides the block into 8 parts and computes a partial ISA for each part only as needed.

Byte aligned LZ77 is coded as follows, where m is the minimum match length (typically 4):

00xxxxxx = literal of length xxxxxx (1..64) follows.
01xxxxxx oooooooo oooooooo = match, length xxxxxx (m..m+63), 16 bit offset.
10xxxxxx oooooooo oooooooo oooooooo = match, 24 bit offset.
11xxxxxx oooooooo oooooooo oooooooo oooooooo = match, 32 bit offset.

Variable length LZ77 for block sizes up to 16 MB are coded as follows:
00, 1x1x..0 = literal of length 1xx.. follows.
mm, mmm, 1x1x...0, xx, ooo... = match length 1xx...xx, offset 1ooo... (m - 8 bits of o).

For example, a match of length 9 and offset 17 would be coded as 01,100,100,01,0001. The first two fields form the binary number 01100 = 12, indicating that 12 - 8 = 4 bits are used in the last field to encode the offset 0001. When a 1 is prefixed, it becomes 10001 = 17.

The length is the unary number 100 in the third field, followed by the two fixed bits 01 in the fourth field. A unary number is written using interleaved Elias Gamma coding, by dropping the leading 1, then preceding the other bits with a 1 and terminating with a 0. Thus, 100 decodes to 10, and the whole length decodes to 1001 = 9. The smallest possible length is 0,00 = 4.

For block sizes of size $R = 2^{RB} > 2^{24}$, matches are coded as follows:

mm, mmm, 1x1x...0, xx, rrr.., ooo... = offset 1ooo...rrr...

The extra field (rrr...) is appended to the offset to allow values up to R - 1.

Variable length codes shown separated by commas are packed into bytes LSB first. The last byte is padded with up to seven 0 bits, which is too short to be confused with any valid code.

LZ77 codes are optimized for the case of 25% literals and 75% matches, where literal length probabilities are proportional to $1/n^2$, match length probabilities are proportional to $1/(n + 4)^2$, and match offset probabilities are proportional to 1/n. These approximate distributions were found experimentally.

The ZPAQ compression algorithm
ZPAQ compresses a set of files or directories as follows. First it compares the last-modified file dates with those stored in the archive and skips any that are unchanged. Next it collects a list of the remaining files and sorts them first alphabetically by file name extension, then by decreasing size rounded to 16 KiB, and finally alphabetically by full path. The purpose of this step is to group similar files together. Rounding the size improves disk access speed by increasing grouping within directories. Sorting largest to smallest improves compression because it increases the chances that a new file type will be placed in a new block because the current one doesn't have room.

Next the files are split along content-dependent boundaries to an average fragment size of 64 KiB. The fragment SHA-1 hash is computed and compared to the hashes already added. If the hash matches, then the fragment is assumed to match too. There is a probability of 2^{-160} that two different fragments will produce the same hash. Unmatched fragments are packed into blocks to be compressed in parallel by separate threads.

Files are split into fragments by computing a rolling hash along a variable sized window that depends

on the last 32 bytes not predicted by an order-1 context, as well as any predicted bytes in between. When the 32-bit hash is less than 2^{16}, then a new fragment is started. This happens with probability 2^{-16}, producing an average fragment size of 2^{16} bytes. Fragment sizes are also limited to between 4096 and 520,192 bytes. The reason for splitting on content-dependent boundaries is so that if one copy of a file is edited slightly by inserting or deleting data, then the remaining fragments of the file will still match.

Each fragment requires 32 extra bytes of storage: 20 bytes for the hash, 8 bytes for two redundant copies of the fragment size (appended to the data block and again in the metadata), and one pointer in the file fragment list. This introduces an overhead of 0.05% for a fragment size of 2^{16}. ZPAQ keeps this information in memory, as well as an index to look up hashes, requiring about 1 MB of memory for every 1 GB of archive size. Smaller fragments would improve deduplication but increase overhead.

Fragmentation

Rather than a Rabin filter [19], ZPAQ uses a variable sized hash window that grows on highly compressible data. It maintains a 256 byte order-1 context table which predicts the next byte. It updates the rolling hash $h(x_{1..n}) = m(h(x_{1..n-1}) + x_n + 1) \pmod{2^{32}}$, where m is an odd number if x_n is predicted correctly, and m is an even number not divisible by 4 otherwise. The table is then updated by replacing the predicted byte with the actual byte. ZPAQ uses the multipliers 314159265 for predicted bytes and 271828182 for mispredicted bytes.

Fragment analysis

During fragmentation, the contents of the order-1 prediction table are used to determine the file type and degree of compressibility used to select the block packing strategy and compression algorithm. Highly random data is packed into smaller blocks and stored without compression. Using smaller blocks makes no difference in this case, but can improve compression by avoiding splitting files later because the block was full.

ZPAQ rates compressibility on a scale of 0 to 255 where 0 means random. At the end of the fragment it applies 4 tests and picks the highest score:

1. Fraction of correctly predicted bytes out of 256 while computing the rolling hash.
2. Deviation of the prediction table from a random distribution.
3. Number of zeros (unseen bytes) in the prediction table.
4. Number of matches between corresponding elements of the final versions of the prediction tables for the last two fragments.

The deviation from randomness in step 2 is measured by computing a histogram of the prediction table. As the 256 predictions are counted, a score of 1/n is accumulated, where n is the new count. If the data is random, the score should be about T = 208. The compressibility score is 256 - min(1, 256T/208).

The analysis step also classifies the block as text or x86 so that appropriate compression models can be used, such as word models or an E8E9 transform. To detect text, a fragment gets one point for each space character predicted by a letter, digit, period, or comma. It loses one point for each prediction of an invalid UTF-8 byte or two byte sequence. If the fragment has at least 3 points, it is classified as text. A block is classified as text if at least half of its fragments are.

A fragment is detected as x86 if the byte value 139 (8B hex, a MOV opcode) is detected in at least 5 contexts. A block is detected as x86 if at least 1/4 of its fragments are.

Compression Levels

ZPAQ supports 5 compression levels selected by the option `-method 1` through `-method 5` with increasing levels of compression at a cost in speed.

1. Fast compression using variable length LZ77.
2. Fast decompression using longer searches for matches.
3. Context modeled LZ77 or BWT.
4. A small context model.
5. A large context model with additional analysis for periodic data.

All levels apply an E8E9 step if the data is classified as x86. All levels except 5 will use a faster compression algorithm or simply store the data if it is classified as highly random.

Level 1 uses 16 MiB blocks. Levels 2 and higher use 64 MiB blocks. The block size can be overridden to improve compression at a cost of memory, for example, `-method 17` to specify method 1 with a block size of 2^7 = 128 MiB.

-method 1
Method 1 stores or compresses using variable length LZ77 depending on the compressibility score. There is no context modeling.

0..9: -method x0.0
10..19: -method x4.1.4.0.1.15
20..31: -method x4.1.4.0.2.16
32..63: -method x4.1.4.0.2.23
64..240: -method x4.1.5.0.3.23
241..255: -method x4.1.6.0.3.23

The advanced method x0.0 means journaling mode (x), no compression (0), and a block size of 2^0 = 1 MiB. x4.1.4.0.1.15 means journaling mode (x), block size 2^4 = 16 MiB, variable length LZ77 (1), minimum match length 4, no secondary hash table (0), search depth (bucket size) 2^1 = 2, and a hash table size of 2^{15} elements (requiring 128 KiB memory).

Highly compressible data uses a higher minimum match length and a larger hash table and greater search depth to improve compression. For compressibility scores of 32 or higher, if a different block size is selected, then the hash table size (2^{23} elements using 32 MiB) is adjusted accordingly to use twice as much memory as the block size.

-method 2
Method 2 stores in the same format as 1. It has slower but better compression with no effect on decompression speed.

0..7: -method x0.0 (no compression).
8..15: -method x6.1.4.0.3.25 (128 MiB hash table).
16..255: -method x6.1.4.0.7.27.1 (256 MiB suffix array with lookahead).

Again the hash table or suffix array size is scaled to the block size. The meaning of the last method is journaling (x), 64 MiB block size (6), LZ77 (1), minimum match length 4, no secondary hash table, search up to 2^7 in the suffix array (indicated by 27 = 21 higher than the block size), and 1 byte lookahead (1).

-method 3
Method 3 uses BWT for text or highly compressible data. Otherwise it uses byte aligned LZ77 with order 2 modeling of literals and the parse state.

0..4: -method x0.0 (no compression).
5..11: -method x6.1.4.0.3.25 (fast LZ77).
12..159: and not text: -method x6.2.12.0.7.27.1c0.0.511i2 (order 2 LZ77 with SA).
160..255: or text and 12..255: -method x6.3ci1 (BWT).

The meaning of the slow LZ77 method is journaling (x), block size 2^6 = 64 MiB, byte aligned LZ77 (2), minimum match length 12, no secondary hash table, suffix array (27 = 6 + 21), lookahead (1), ICM (c0), no periodic or distance context (0), order 1 LZ77 state (511 = mask + 256), order 2 ISSE chain (i2). The parse state causes the context hash to depend on whether the currently modeled byte is a match length, match offset byte, literal length, or literal byte.

-method 4
0..2: -method x0.0 (no compression).
3..5: -method x6.1.4.0.3.25 (fast LZ77)
6..11: -method x6.2.5.0.7.27.1c0.0.511 (order 1 LZ77 with SA, minimum match length 5).
12..224 and not text: -method x6.0ci1.1.1.1.2am (order 0-1-2-3-4-6 ICM-ISSE chain and match).
12..224 and text: -method x6.0ci1.1.1.1.2awm (adds order 0 word model).
225..255: x6.3ci1 (BWT).

-method 5
Method 5 uses a large context model regardless of whether the data is compressible. It uses different models for text and non-text. It does an initial analysis to test for periodic data and adds appropriate models.

Text: -method
x6.0.w2c0.1010.255i1c256ci1,1,1,1,1,1,2ac0,2,0,255i1c0,3,0,0,255i1c0,4,0,0,0,255i1mm16ts19t0
Non text: -method x6.0w1c256ci1,1,1,1,1,1,2ac0,2,0,255i1c0,3,0,0,255i1c0,4,0,0,0,255i1mm16ts19t0

Both models include a stationary order 0 model (c256), an order 0-1-2-3-4-5-6-8 ICM-ISSE chain, a MATCH, and three periodic order 1 sparse models with period 2, 3, and 4 modeling the byte above, each chained to an ISSE modeling the byte to the left. All of this is mixed using order 0 and 1 mixers. Those two are mixed with an order 0 MIX2, followed by an SSE and a final MIX2 that mixes the SSE input and output. Non text includes an order 0 word model (just in case). Text includes an order 0-1 ICM-ISSE chain word model and a gap model whose context is the distance to the last linefeed character (c0.1010) to model ASCII tables with an order 1-2 ICM-ISSE chain.

ZPAQ tests for periodic data of length at least 5 by searching for 3 equal gaps in 4 successive occurrences of the same byte. If it finds a period, say 50, it adds a model like c0.0.1049.255i1c0.50i1. c0.0.1049.255 means 1049-1000 = 49 zeros followed by 255, which is the byte above. i1 chains the byte to the left. c0.50 models the offset mod 50 as a context.

Example
BOOK1 is detected as text with a compressibility score of 196. It is compressed as follows:

Method	BOOK1
-method 1	345436
-method 2	316874
-method 3	216233
-method 4	210962
-method 5	**201100**

References

1. M. Mahoney. ZPAQ (2015). http://mattmahoney.net/dc/zpaq.html

2. M. Mahoney. 10 GB Compression Benchmark (2015). http://mattmahoney.net/dc/10gb.html

3. G. Roelofs et. al. Info-ZIP (2008). http://www.info-zip.org/

4. I. Pavlov. 7-zip (2015). http://www.7-zip.org/

5. A. Roshal. RARLAB (2015). http://www.rarlab.com/

6. B. Ziganshin. FreeArc (2010). http://freearc.org/

7. J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Information Theory*. **23** (3): 337-343 (1977).

8. M. Ghosh. Pcompress (2013). https://github.com/moinakg/pcompress

9. L. Reinhold. Exdupe (2013). https://web.archive.org/web/20150908042905/http://www.exdupe.com/

10. L. Wirzenius. Obnam (2015). http://obnam.org/

11. M. Mahoney. The ZPAQ open standard format for highly compressed data - level 2, version 2.04 (2014). http://mattmahoney.net/dc/zpaq204.pdf

12. M. Mahoney. unzpaq200.cpp reference decoder (2012). http://mattmahoney.net/dc/unzpaq200.cpp

13. D. Eastlake. U.S. Secure Hash Algorithm 1 (SHA1). RFC-3174 (2001).

14. M. Mahoney. Adaptive Weighting of Context Models for Lossless Data Compression. *Technical Report CS-2005-16*. Florida Tech. (2005).

15. M. Mahoney. PAQ8 (2007). http://mattmahoney.net/dc/#paq

16. Calgary corpus. http://corpus.canterbury.ac.nz/descriptions/#calgary

17. Y. Mori. libdivsufsort. (2015). https://github.com/y-256/libdivsufsort

18. M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124*. Digital Equipment Corp. (1994).

19. M. O. Rabin. Fingerprinting by Random Polynomials. *Tech. Report TR-CSE-03-01*, Harvard Univ. (1981).