Freie Universität Berlin
Department of Mathematics and Computer Science
Institute of Computer Science

Bachelor Thesis

Design of a Python-subset Compiler in Rust targeting ZPAQL

Kai Lüke

kailueke@riseup.net

Supervisors: Prof. Dr. Günter Rote
Dipl.-Inform. Till Zoppke

Berlin, August 23, 2016

Abstract

The compressed data container format ZPAQ embeds decompression algorithms as ZPAQL bytecode in the archive. This work contributes a Python-subset compiler written in Rust for the assembly language ZPAQL, discusses design decisions and improvements. On the way it explains ZPAQ and some theoretical and practical properties of context mixing compression by the example of compressing digits of π. As use cases for the compiler it shows a lossless compression algorithm for PNM image data, a LZ77 variant ported to Python from ZPAQL to measure compiler overhead and as most complex case a implementation of the Brotli algorithm. It aims to make the development of algorithms for ZPAQ more accessible and leverage the discussion whether the current specification limits the suitability of ZPAQ as an universal standard for compressed archives.

Contents

1 Introduction

It takes time until new and incompatible data compression algorithms become distributed in software. Also different input data should often be handled with different compression techniques, utilizing best knowledge about the data.

The ZPAQ standard format for compressed data is a container format which also holds the needed decompression algorithms. They can be specified through a context mixing model of several predictors with a bytecode which computes the context data for them (used for arithmetic coding), a bytecode for postprocessing (used for transformations or stand-alone algorithms) or any combination of both.

Arithmetic coding spreads symbols over a number range partitioned according to the probability distribution. That way a likely to be encoded symbol can get a bigger part. The whole message is encoded at once from the beginning. And every time a symbol was chosen the – possibly modified – probability distribution is applied again to segment its part of the number range. Then for the next symbol this is the number range partition to choose from. When the last symbol has been processed, the number range is very narrow compared to the beginning and every number of it now represents the whole message. So the shortest in terms of binary representation can be selected and a decoder can do the same steps again by choosing the symbol which has this number in its range and applying then the partitioning to this range again according to the probability distribution. In practice, one has either to use a special end-of-message symbol or specify the message length before to define an end to this process.

The history which led to the development of ZPAQ shall shortly be explained in this chapter, followed by the motivation of writing a compiler for ZPAQL and the research question of this work. In the following chapter the whole picture of ZPAQ compression is visualized and the building blocks, i.e. context mixing and the bytecode, are explained in detail. General theory of data compression and its limits are shortly noted on afterwards. The main part is preceded by a short introduction to compiler construction, the implementation language Rust and the source language Python. One chapter outlines the conditions and difficulties of ZPAQL as a compiler target. The following chapters cover the chosen Python-subset (6), the developed API and compiler internals (7), example programs (8) and finally the evaluation (9).

References to compressors and project websites or more detailed documentation and short remarks are located in footnotes. All websites have been accessed until August 19th 2016. Academic publications are listed in the bibliography.

1.1 Preceding History

A big milestone being reached in data compression has been the Lempel-Ziv algorithm which is a form of dictionary compression due to its consistence of backward references to data slices which have already been seen. The area of statistical data compression was first stationary and Huffman optimal prefix codes can be mentioned for entropy coding before arithmetic coding was known. Basically, few bits should be used for frequent symbols but more for infrequent symbols. Entropy coding developed further with adaptive coding from the mid-80s. One major innovation called PPM (Prediction by Partial Matching) usually works on order-N byte contexts and predictions [1] whereas DMC (Dynamic Markov Compression) predicts bits [2]. Adaptive Huffman coding was developed as well. A decade later CTW (Context Tree Weighting) also utilizes bit prediction but mixes the learned distributions of the order-N byte contexts [3].

The predecessors to the series of PAQ compressors, called P5, P6, P12, are based on a 2-layer neural network with online training that predicts one bit for context inputs up to order-5. At that time they were on par with variants of PPM concerning speed and compression ratio [4].

In PAQ1 the neural network is not used anymore as the probability and confidence of various models are combined through weighted averaging. There is a non-stationary n-gram model up to order-8, a match model for occurrences longer than 8 bytes in the last 4 MB, a cyclic model for patterns in rows and a whole word model up to order-2. As statistical data the models mostly hold the bit counts of the contexts as one byte in a hash table. At its release it could produce best results for a concatenation of the Calgary corpus (a data compression benchmark) [5].

The models and their semi-stationary update were improved and PAQ2 introduced SSE (Secondary Symbol Estimation) which improves a given probability by mapping it to a bit history. PAQ4 changed to adaptive linear weighting and PAQ5 added another mixer. Many other versions have model improvements and added special models or transforms. PAQAR, a fork of PAQ6, reworked the mixing architecture and could turn models off. PAsQDa uses a transformation to map words to dictionary codes. PAQ6 variants were one after the other leading in the Calgary challenge [6].

PAQ7 is a rewrite using neural network logistic mixing. Specifically it includes a JPEG model that predicts the Huffman codes based on the DCT coefficients. PAQ8 variants come with more file models, dictionary or x86 call address preprocessors or a DMC model. The PAQ8HP series won the *Hutter Prize for Lossless Compression of Human Knowledge*[1] about compressing a part of an English Wikipedia dump. Many achievements were mainly made through special models for various file formats. The simpler LPAQ provides faster but less compression and later versions aim on text compression. PAQ9A has LZ (Lempel-Ziv) precompression and cascaded ISSE (Indirect Secondary Symbol Estimation). The detailed history including source code is tracked on the PAQ history website[2] and in M. Mahoney's book on data compression [7].

[1] `http://prize.hutter1.net/`

[2] The PAQ Data Compression Programs `http://mattmahoney.net/dc/paq.html`

Parts of PAQ variants made it into the recent context mixing compressor cmix that leads the *Large Text Compression Benchmark*[3] and the *Silesia Open Source Compression Benchmark*[4] but with around 30 GB RAM usage[5].

»ZPAQ is intended to replace PAQ and its variants (PAQ8, PAQ9A, LPAQ, LPQ1, etc) with similar or better compression in a portable, standard format. Current versions of PAQ break archive compatibility with each compression improvement. ZPAQ is intended to fix that.«[6]

The development of the ZPAQ archiver started from early 2009 and defined the first version of *The ZPAQ Open Standard Format for Highly Compressed Data* [8] after some months. The main idea is to move the layout of the context mixing tree as well as the algorithm for context computation and the one for postprocessing into the archive before each block of compressed data. There are nine components[7] to choose from for the tree, mostly context models coming from PAQ. In addition the algorithms are provided as bytecode for a minimal virtual machine. Hence the algorithm implementation is mostly independent of the decompressor implementation and compatibility is preserved when improvements are made. Also depending on the input data an appropriate compression method can be chosen. The main program using libzpaq is an incremental journaling backup utility which supports deduplication and encryption and provides various compression levels using LZ77, BWT (Burrows-Wheeler Transform) and context models [9]. But there are also reference decoders unzpaq and tiny_unzpaq, a simple pipeline application zpipe and a development tool zpaqd[8].

The fastqz compressor [10] for Sanger FASTQ format DNA strings and quality scores also uses the ZPAQ format and was submitted to the *Pistoia Alliance Sequence Squeeze Competition*[9].

1.2 Motivation

The development of ZPAQ continued specially for the use case of the incremental archiver program. But the appearance of new algorithms for ZPAQ reached a rather low level, as well as the number of authors[10] despite the fact that it offers a good environment for research about context mixing methods and crafting of special solutions for use cases with a known type of data because one can build upon existing parts. The leading reason could be that the assembly language ZPAQL with its few registers is not very accessible and programs are hard to grasp or get easily unmanageable when complex tasks like parsing a header should be accomplished. Therefore, the first question is whether another language can support ZPAQ's popularity.

[3] `http://mattmahoney.net/dc/text.html`

[4] `http://mattmahoney.net/dc/silesia.html`

[5] `http://www.byronknoll.com/cmix.html`

[6] `http://mattmahoney.net/dc/zpaq.html`

[7] See chapter 2.2 or `http://mattmahoney.net/dc/dce.html#Section_437`

[8] `http://mattmahoney.net/dc/zpaqutil.html`

[9] `http://www.pistoiaalliance.org/projects/sequence-squeeze/`

[10] The majority of available configurations is listed on `http://mattmahoney.net/dc/zpaqutil.html`

If that is the case, then a compiler for a well-known programming language could help to overcome the obstacle of learning ZPAQL for implementing new algorithms.

The second question is whether the design decisions of the ZPAQ specification allow for arbitrary compression algorithms to be used with ZPAQ, even if they bring their own encoder or a dictionary or whether ZPAQ is rather meant to be a platform for algorithms that use the predefined prediction components like (I)CM and (I)SSE[11] together with the built-in arithmetic coder.

1.3 Research Question

As an approach towards these questions mentioned above this work wants to contribute a compiler that emits ZPAQL for a subset of the popular Python programming language. The subset should be oriented to the data types of the ZPAQL virtual machine. The architecture and operation of the Python code should still maintain similarity to the ZPAQL execution workflow to avoid abstractions. That means only integers, no objects, floating-point numbers or strings, but the ability to address the two memory sections of the VM as arrays. Running the Python code should be possible as usual which helps to keep debugging out of the ZPAQL VM.

All this would ease the development of new algorithms and discover limitations of the current ZPAQ standard. An example for a new algorithm should be provided by developing a model for uncompressed PNM image data which should be compared to other lossless image compressors.

As an example for a complex all-purpose algorithm the recent Brotli compression algorithm[11] will be brought to ZPAQ. It was proposed for HTTP2 compression, includes a dictionary and combines LZ77, Huffman coding and order-2 context modeling. This is a challenge because it has to be solely implemented in the postprocessing step and also needs memory management.

The quality of the compiler will be evaluated by comparing to an existing hand-written implementation of a LZ compressor. Design decisions of the compiler and difficulties faced will be discussed.

[11] (Indirect) context model and (indirect) secondary symbol estimation, four of the nine ZPAQ components.

2 The ZPAQ Standard Format for Compressed Data

This chapter gives an introduction to the ZPAQ archive specification in the current version 2.06. A small use case is explained as well as the needed commands. Then the context mixing components and the language ZPAQL are presented. Finally, two real-world examples are explained.

2.1 ZPAQ Specification and Working Principle

The Level 1 specification of this container format required the data always to be encoded with the arithmetic coder and at least one predictor. A backwards incompatible change leading to Level 2 was to allow raw data to be stored and thus no prediction component to be used, as needed for incompressible data or standalone postprocessor algorithms e.g. a fast LZ variant [8]. It also defines a standard encryption for the whole archive file and introduces an append-only journaling mode for incremental backups and deduplication on top of the streaming format which is still the basis of the archive. These features are out of scope for this work and therefore the focus is on the compression architecture and the virtual machine.

It is only specified what a valid archive is and how decompression takes place (nevertheless, probability and thus context computation have to be symmetric for compression and decompression with an arithmetic coder). In case of ambiguity in the specification there is the reference implementation unzpaq[1], a tiny version[2] of it and the more mature library libzpaq that is used by the zpaq archiver[3], the zpaqd development tool[4] or as plug-in for file archivers and incorporates a x86/64 JIT compiler for ZPAQL.

»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.« [8]

A block header has no information on the length of the block because like for segments an end marker is used. Memory requirements for the two bytecodes *hcomp* and *pcomp* are defined in the header. It is noted

[1] `http://mattmahoney.net/dc/unzpaq206.cpp`

[2] `http://mattmahoney.net/dc/tiny_unzpaq.cpp`

[3] `http://mattmahoney.net/dc/zpaq715.zip`

[4] `http://mattmahoney.net/dc/zpaqd715.zip`

whether arithmetic coding is used and what predicting components make up the context mixing tree. Each component has arguments also determining memory use. If needed the bytecode *hcomp* is embedded to compute context data for the components of the context mixing tree for each byte. All components give bit predictions for the partially decoded byte (these are passed upwards the tree) and are trained afterwards with the correct bit which was decoded based on the root (i.e. the last) node probability for each bit.

The optionally arithmetic coded data which comes from all segment content (not the segment filename or comment) in the block can start with an embedded *pcomp* bytecode or declare that no *pcomp* bytecode is present. Therefore, the *hcomp* section can already be used for context computation to compress the *pcomp* bytecode (0 for empty or 1 followed by the length and the bytecode). The *pcomp* code is used for postprocessing, may it be a simple transform or the decompression of LZ codes. It gets each decoded byte as input and outputs a number of bytes not necessarily equal to the input.

That means the four combinations for a block are in total no compression, only context mixing with arithmetic coding, only postprocessing the stored data or context mixing with subsequent postprocessing (from the decompressor perspective). The chosen selection applies for all (file) segments in the block.

The following charts illustrate the named parts and their relation to each other for a sample compression use case. The transform for x86 machine code enhances compressibility by converting relative to static addresses after each CALL and JMP instruction (0xE8 and 0xE9). It is applied on the two input files, a x86 executable binary and shared library. Therefore a ZPAQL *pcomp* program needs to be supplied in the archive block to revert that transform. Encoding takes place based on the probability distribution of 1 and 0 for each bit of the current byte as they are provided as prediction by the root node of the simple context mixing tree. The *hcomp* program is loaded into the ZPAQL VM and computes contexts for the two components. The ISSE maps the context to a bit history which is used as context for a learning mixer that should improve the probability provided by the first component, a CM (Context Model) which should learn good predictions for the given context. The whole model and the *hcomp* bytecode are also embedded into the archive block. The two files are stored as two segments in the block (like a ”solid” archive). Because the preprocessor might be any external program or also included in the compressing archiver and is of no use for decompression it is therefore not mentioned in the archive anymore.

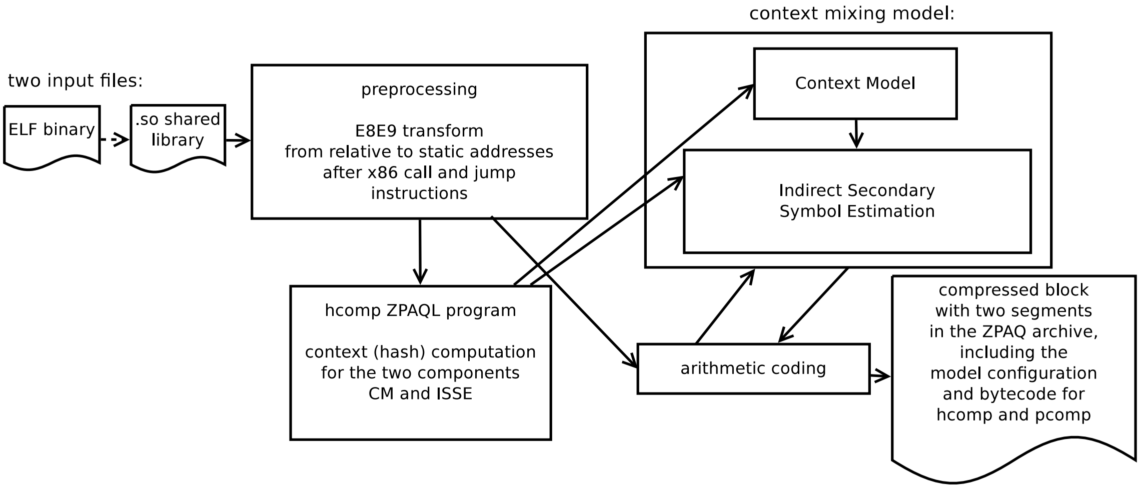

Figure 2.1: Possible compression scheme

Decompression takes place in reverse manner and *hcomp* is loaded into the ZPAQL VM to compute the context data for the components of the model. They supply the predictions to the arithmetic coder and are corrected afterwards. For the reverse transform of each segment *pcomp* is read from the decoded stream and loaded in another VM. Then the two segments follow in the decoding step and go through the postprocessing transform before they are written out as files.

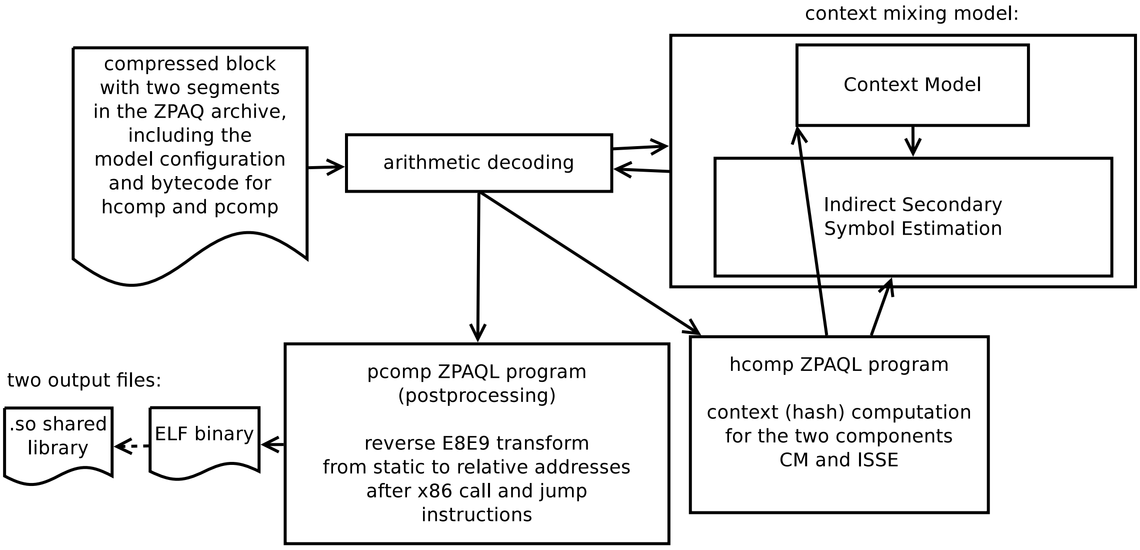

Figure 2.2: Accompanying decompression

The tool zpaqd only supports the streaming format and can be used to construct this example setup by writing a configuration file for it and then adding two files as segments into a block. But beside the algorithms that are already defined for compression in libzpaq for the levels 1 to 5 (LZ77, BWT and context mixing) it also offers the ability to specify a customized model (E8E9, LZ77 transformations or also word models are supported) given as argument, so that the above configuration can also be brought to life with something like `zpaq a [archive] [files] -method s8.4c22.0.255.255i3` (denotation documented in libzpaq.h and the zpaq man page[5]). Here the first 8 accounts for $2^8 = 256$ MB blocks, so that both segments should fit into one block (yet the zpaq application uses the API in a way that creates an additional block), then an order-2 CM and an order-3 ISSE are chained. The resulting configuration including the two ZPAQL programs stored in the archive can be listed with `zpaqd l [archive]`.

For a more general view c.f. the compression workflow of the zpaq end user archiver as described in the article mentioned [9]. It selects one of its predefined algorithms based on their performance for the data and uses deduplication through the journaling format.

2.2 Context Mixing: Components and Context Data

The way the mixing takes place has evolved from the neural network approach in P5 over weighted averaging from PAQ1 and adaptive linear weighting to logistic mixing in PAQ7 (like in a neural network and instead of back propagation the weights are updated). In ZPAQ the probabilities are also handled in the logistic domain $stretch(p) = ln(\frac{p}{1-p})$ as output of the predicting components and can be averaged or refined by other components before the output of the last component is transformed $squash(x) = stretch^{-1}(x) = \frac{1}{1+e^{-x}}$ to be used in the arithmetic coder to code one bit [7].

ZPAQ defines nine components. Up to 255 of them can be used in a model but only linear trees can be constructed as mixing network i.e. when they are listed, they can only refer to predictions of preceding components. The last one listed is the top node [8]. They get their context data input through computation in *hcomp*. To supply the component with a higher order context, a rotating history buffer needs to be maintained by *hcomp*. Now the components are listed without their arguments and internals which can be found in the specification [8].

CONST is the most simple and gives a fixed prediction. It does not receive context data.

CM is a direct context model that holds counts of zeros/ones and a prediction for each of the eight bits expected in the specified context. The counts are used to update the prediction to reduce the error. The given context data is often a hash (and collisions can hurt compression). Also the partially decoded byte is expanded to nine bits and XORed with the context after coding each bit, so that the next bit's prediction is accessed in the table.

ICM is an indirect context model consisting of a hash table to map the given context and the partially decoded byte to a bit history which is then mapped to a prediction like in a direct context model. Histories are counts of 1 and 0 and represented by states (explained in [7], chapter *4.1.3. Indirect Models*) and adjusted after the prediction to a better fitting state. The prediction is also updated.

[5] `http://mattmahoney.net/dc/zpaqdoc.html`

MATCH is a model that takes the following byte of a found string match as prediction until a mismatch happens. Therefore it keeps its own history buffer and the prediction is also varied depending on the length of the match. The match is found based on the (higher order) context hash input. This works because the search does not take place in the history buffer, but a table maps each context hash input to the last element in the history buffer.

AVG outputs a non-adaptive weighted average of the predictions of two other components. It does not receive context data.

MIX maps a context and the masked partially decoded byte to weights for producing an averaged prediction of some other components. Afterwards the weights are updated to reduce the prediction error.

MIX2 is a simpler MIX and can only take two predictions as input.

SSE stands for secondary symbol estimation (Dmitry Shkarin, also known as *Adaptive Probability Map* in PAQ7 or LPAQ) because it receives a context as input and takes the prediction of another component which is then quantized. For the given context this quantized prediction and the next closest quantization value are mapped to predictions which result in an interpolation of both. Initially this map is an identity, but the update step corrects the prediction of the closer-to-original quantized value the same way as in the CM update phase.
»A typical place for SSE is to adjust the output of a mixer using a low order (0 or 1) context. SSE components may be chained in series with contexts typically in increasing order. Or they may be in parallel with independent contexts, and the results mixed or averaged together.« [7]

ISSE is an indirect secondary symbol estimator that, as an SSE, refines the prediction of another component based on a context which is mapped to a bit history like in an ICM. That bit history is set as context for an adaptive MIX to select the weights to combine the original prediction with a fixed prediction.
»Generally, the best compression is obtained when each ISSE context contains the lower order context of its input.« [7]

2.3 ZPAQL: Virtual Machine and Instruction Set

At the beginning of each block the two bytecodes *hcomp* and *pcomp*, if needed, are loaded to a virtual machine for each. It consists of the program counter PC, 32-bit registers A, B, C and D, an 1-bit condition flag F, 256 32-bit registers $R_0 ... R_{255}$ and the arrays M (8-bit elements) and H (32-bit elements). Initially, all registers and arrays hold 0. The size of H and M is defined in the block header along with the information which context-mixing components are used.

For each encoded or decoded byte *hcomp* gets run to set the contexts for the components. The context data for a components $i < 256$ has to be stored in $H[i]$. As first input *hcomp* sees whether a postprocessor is present and if yes, then its length and bytecode. Afterwards the data of the segments is coming, without any separation. Except for the *PC* which is set to 0 and the *A* register which is used for the input all state is preserved between the calls.

The postprocessor *pcomp* is run for each decoded byte of the block to revert the preprocessing and puts out the data via an instruction. After each segment it is invoked with $2^{32} - 1$ as input to mark the end.

There is an assembly language for the bytecode which is also used in the table describing the corresponding opcodes in the specification [8]. In this assembly language whitespace can only occur between opcode bytes in order to visualize that `a=b` is a 1-byte opcode while `a= 123` is a 2-byte opcode. Comments are written in brackets.

Operations on the 32-bit registers and elements of H are *mod* 2^{32} and interpreted as positive numbers in comparisons. Indexes access into M and H is *mod msize* or *mod hsize* and denoted as `*B` for $M[B]$, `*C` for $M[C]$ and `*D` for $H[D]$. Because M holds bytes operations on `*B` and `*C` are *mod* 256 and swapping via `B*<>A` or `C*<>A` alters only the lower byte of A [8].

Instructions	Semantics and Constraints
`error`	cause execution to fail
`X++`	increment X by 1 (X is one of A, B, C, D, *B, *C, *D)
`X--`	decrement X by 1
`X!`	flip all bits
`X=0`	set X to 0
`X<>A`	swap (X is not A)
`X=X`	set X to X
`X= N`	set X to 0 <= N <= 255
`A+=X`	add X on A
`A-=X`	subtract X from A
`A*=X`	multiply A by X
`A/=X`	divide A by X (set $A = 0$ if $X = 0$)
`A%=X`	$A = A \mod X$ (set $A = 0$ if $X = 0$)
`A&=X`	binary AND with X
`A&~=X`	binary AND with flipped bits of X
`A|=X`	binary OR with X
`A^=X`	binary XOR with X
`A<<=X`	bitwise left shift of A by X *mod* 32 bits
`A>>=X`	bitwise right shift of A by X *mod* 32 bits

`A+= N, A-= N, A*= N, A/= N, A%= N, A&= N, A&~= N, A|= N, A^= N, A<<= N, A>>= N`	same as previous instructions but with $0 <= N <= 255$
`A==X`	$F = 1$ if $A = X$ otherwise $F = 0$
`A<X`	$F = 1$ if $A < X$ otherwise $F = 0$
`A>X`	$F = 1$ if $A > X$ otherwise $F = 0$
`A== N, A< N, A> N`	same as above but with $0 <= N <= 255$
`A=R N, B=R N, C=R N, D=R N`	set A, B, C or D to R_N
`R=A N`	set R_N to A
`HALT`	ends current execution of the program
`OUT`	output A *mod* 256 in *pcomp*, ignored in *hcomp*
`HASH`	$A = (A + M[B] + 512) \cdot 773$
`HASHD`	$H[D] = (H[D] + A + 512) \cdot 773$
`JMP I`	add $-128 <= I <= 127$ to PC relative to the following instruction, so $I = 0$ has no effect and $I = -1$ is an endless loop (in the specification a positive N is used, so $PC_{following}$ += $((N + 128)$ *mod* $256) - 128)$
`JT N`	jump if $F = 1$
`JF N`	jump if $F = 0$
`LJ L`	$PC_{following} = L$ with $0 <= L < 2^{16}$ (in the specification written as `LJ N M` because it is a 3-byte instruction with $PC_{following} = N + 256 \cdot M$)

Beside the opcodes libzpaq also supports using helper macros like `if(not)? … (else)? … endif`, the long jump versions `if(not)?l … (elsel)? … endifl` and `do … (while|until|forever)` which will be converted to conditional jump opcodes. An if-block is executed if $F = 1$, the while-jump is executed if $F = 1$, an until-jump is executed if $F = 0$. So that means the test for the condition has to be written before the macro: `a> 255 if … endif`. The statements can also be interweaved, e.g. write a do-while-loop or a continue-jump as `do … if … forever endif`.

In a ZPAQ config file the two sections for *hcomp* and *pcomp* are written behind the context mixing model configuration. The *pcomp* section is optional. Comments can appear everywhere within brackets.

Syntax: (where $I < N$ and 2^{HH}, 2^{HM}, 2^{PH} and 2^{PM} define the size of H and M for each section)

```
comp HH HM PH PM N
  (I COMPONENT (ARG)+ )*
hcomp
  (ZPAQL_INSTR)*
  halt
(pcomp (PREPROCESSOR_COMMAND)? ;
  (ZPAQL_INSTR)*
  halt
)?
end
```

2.4 Examples: A simple Context Model and LZ1 with a Context Model

The following example configuration is based on fast.cfg from the utility site[6] and can be used for text compression and adaptively combines (independently of contexts, just based on the success of the last prediction) the prediction of a direct order-1 context model with the prediction of a order-4 ISSE which refines the prediction of a order-2 ICM. The arguments for the components are documented in the specification [8].

```
comp 2 2 0 0 4 (hh hm ph pm n)
                    (where H gets the size of 2^hh in hcomp or 2^ph in comp,
                     M 2^hm or 2^pm and n is the number of
                     context-mixing components)
  0 cm 19 4   (will get an order 1 context)
  1 icm 16    (order 2, chained to isse)
  2 isse 19 1 (order 4, has reference to ICM component 1)
  3 mix2 0 0 2 24 0 (moderate adapting mixer between CM and ISSE
                     based on which predicts better, no contexts even for bits)
  (ICM and ISSE part adapted from fast.cfg)
hcomp
  r=a 2 (R2 = A, input byte in R2)
  d=0
  a<<= 9 *d=a (H[D] = A) (set context to actual byte)
  (leaving first 9 bits free for the partially decoded byte)
  a=r 2 (A = R2)
  *b=a (M[B] = A) (save input byte in rotating buffer)
                (full M is used with pointer b)
  a=0 hash (shortcut for A = (A + M[B] + 512) * 773)
  b-- hash
  d= 1 *d=a (order 2 hash for H[1])
  b-- hash b-- hash
  d= 2 *d=a (order 4 hash for H[2])
```

[6] http://mattmahoney.net/dc/zpaqutil.html, config files with a ”post” instead of ”pcomp” are in the old format of the Level 1 specification

```
  (H[3] stays 0 as fixed context for MIX2)
  halt (execution stops here for this input byte)
end
```

Listing 2.1: Example mfast.cfg without a pcomp section

To demonstrate the compression phases and parts involved in detail the LZ1 configuration from the utility site is chosen, but also the BWT.1 examples are worth to look at.

The LZ1 configuration relies on a preprocessor `lzpre.cpp` which turns the input data into a compressed LZ77-variant representation of codes for match copies and literal strings. This is further compressed through arithmetic coding with probabilities provided by an ICM (indirect context model).

The contexts are always hashed as $hash(hash(a) + b)$ from two values a and b as follows. For the first byte of an offset number of a match the length of the match and the current state (2...4, i.e. 1...3 bytes to follow as offset number) are used as context. For the remaining bytes of an offset number (or a new code if no bytes are remaining) the previous context and the current state (previous state - 1, i.e. 0...2 bytes to follow) are used as context. For the first literal of a literal string the number of literals and the state 5 are used as context. For the following literals the current literal and the state 5 are used as context. For a new code after a literal string instead of a hash of the first value just 0 and the current state (1) are used as context. The bytecode of *pcomp* is not specially handled.

To revert the LZ1 compression *pcomp* parses the literal and match codes and maintains a $16 \text{ MB} = 2^{24}$ byte buffer in M.

```
(lz1.cfg

(C) 2011 Dell Inc. Written by Matt Mahoney
Licensed under GPL v3, http://www.gnu.org/copyleft/gpl.html)

comp 0 0 0 24 1
  0 icm 12 (sometimes ”0 cm 20 48” will compress better)
hcomp
  (c=state: 0=init, 1=expect LZ77 literal or match code,
   2..4=expect n-1 offset bytes,
   5..68=expect n-4 literals)
  b=a (save input)
  a=c a== 1 if (expect code ccxxxxxx as input)
    (cc is number of offset bytes following)
    (00xxxxxx means x+1 literal bytes follow)
    a=b a>>= 6 a&= 3 a> 0 if
      a++ c=a (high 2 bits is code length)
      *d=0 a=b a>>= 3 hashd
    else
      a=b a&= 63 a+= 5 c=a (literal length)
      *d=0 a=b hashd
    endif
  else
    a== 5 if (end of literal)
      c= 1 *d=0
    else
      a== 0 if (init)
        c= 124 *d=0 (5+length of postprocessor)
      else (literal or offset)
        c--
        (model literals in order 1 context, offset order 0)
        a> 5 if *d=0 a=b hashd endif
      endif
    endif
  endif

  (model parse state as context)
  a=c a> 5 if a= 5 endif hashd
  halt
pcomp ./lzpre c ; (code below is equivalent to ”lzpre d”)
  a> 255 if (end of segment)
   b=0 d=0  (reset, is last command before halt)
  else
  (LZ77 decoder: b=i, c=c d=state r1=len r2=off
    state = d = 0 = expect literal or match code
                1 = decoding a literal with len bytes left
                2 = expecting last offset byte of a match
                3,4 = expecting 2,3 match offset bytes
    i = b = position in 16M output buffer
    c = c = input byte
    len = r1 = length of match or literal
    off = r2 = offset of match back from i
  Input format:
    00llllll: literal of length llllll=1..64 to follow
    01lllooo oooooooo: length lll=5..12, offset o=1..2048
    10llllll oooooooo oooooooo: l=1..64 offset=1..65536
    11llllll oooooooo oooooooo oooooooo: 1..64, 1..2^24)
  c=a a=d a== 0 if
    a=c a>>= 6 a++ d=a
    a== 1 if (state?)
      a+=c r=a 1 a=0 r=a 2 (literal len=c+1 off=0)
    else
      a== 2 if a=c a&= 7 r=a 2 (short match: off=c&7)
        a=c a>>= 3 a-= 3 r=a 1 (len=(c>>3)-3)
      else (3 or 4 byte match)
        a=c a&= 63 a++ r=a 1 a=0 r=a 2 (off=0, len=(c&63)-1)
      endif
    endif
  else
    a== 1 if (writing literal)
      a=c *b=a b++ out
      a=r 1 a-- a== 0 if d=0 endif r=a 1 (if (--len==0) state=0)
    else
      a> 2 if (reading offset)
        a=r 2 a<<= 8 a|=c r=a 2 d-- (off=off<<8|c, --state)
      else (state==2, write match)
        a=r 2 a<<= 8 a|=c c=a a=b a-=c a-- c=a (c=i-off-1)
        d=r 1 (d=len)
        do (copy and output d=len bytes)
          a=*c *b=a out c++ b++
        d-- a=d a> 0 while
        (d=state=0. off, len don’t matter)
        endif
      endif
    endif
    endif
    halt
end
```

Listing 2.2: lz1.cfg

For comparison a Python port for usage with the zpaqlpy compiler can be found in `test/lz1.py`. It differs in processing the *pcomp* bytecode with the current opcode byte as context for the next.

3 Notes on Data Compression

Not all data can be compressed. Because if we build a – necessarily bijective[1] – compressing scheme

$$f : \bigcup_{0<i\leq n\in\mathbb{N}} \Sigma^i \rightarrow \bigcup_{0<i\leq n\in\mathbb{N}} \Sigma^i$$

for strings $m \in \Sigma*, \ |m| \leq n$ to strings $z \in \Sigma, \ |z| \leq n$, starting from an identity, every time we remap an input m_1, whereas $f(m_1) = z_1$, to a shorter compressed representation z_2, whereas $f^{-1}(z_2) = m_2$ and $|z_2| < |z_1|$, so that now $f'(m_1) = z_2$ and $|f'(m_1)| < |f(m_1)|$, we have to map m_2 to z_1 so that $f'(m_2) = z_1$ and indeed make this swap in order to maintain the bijection, ending up with $|f'(m_2)| > |f(m_2)|$ because $|z_1| > |z_2|$. So while f' compresses m_1 it expands m_2 and this holds for each iteration when we change our compression scheme.

Luckily, most data which is relevant to us and interesting for compression has patterns and other data where we do not understand the patterns appears to be random and is out of scope for compression algorithms. If we do not have any knowledge about the data except that its symbols are equally distributed with probability $p_i = \frac{1}{\Sigma}$ for each symbol i, best we can do is to use an optimal code reaching the Shannon entropy as coding length per symbol:

$$H = -\sum_i p_i \, log_2 \, p_i$$

For $|\Sigma| = 256$ H would be 8 bit as usual and we can simply store the data instead of encoding it again. In general, given that the distribution is known, we can choose e.g. a non-adaptive arithmetic encoder to almost reach the limit of H bits per symbol. But adaptive arithmetic coding with PPM and others could even give a better average because the distribution is adjusted by exploiting patterns. Therefore, to craft a well performing algorithm for the expected input data knowledge about the patterns is needed in order to give good predictions and go beyond the Shannon entropy as average size.

To define the lower limit that can be reached the concept of algorithmic information or Kolmogorov complexity of a string is developed. Basically it is the length of the shortest program in a fixed language that produces this string. The language choice only influences a constant variation because an interpreter could be written. When comparing different compressors in a benchmark it is common to include the decompressor size in the measurement because it could also hold data or generate it via computation.

[1] We want to allow for every string to be compressed, resulting in the same number – because they have to differ – of compressed strings which also should be decompressible. Another common approach to prove that there is no universal lossless compression is a counting argument which uses the pigeonhole principle.

Using this principle with ZPAQ and its *pcomp* section the first million digits of π if form of the ~1 MB text file `pi.txt` from the *Canterbury Miscellaneous Corpus*[2] can be compressed to a 114 bytes ZPAQ archive which consists of no data stored and a postprocessing step which computes π to the given precision and outputs it as text[3]. This extreme case of the earlier mentioned knowledge about the data can serve as a bridge between Kolmogorov complexity and adaptive arithmetic coding. For faster execution of the ZPAQ model we only take the first ten thousand digits of π. Normally the limit most compressors would stay above (because the digits are equally distributed) is $\frac{H}{8} \cdot 10000 = \frac{-10\cdot(0.1\cdot log_2(0.1))}{8} \cdot 10000 = 4153$ byte instead of 10 KB.

With a ZPAQ context model we can, instead of generating the digits in *pcomp* phase, also use the next expected digit as context so that the predictor will quickly learn that e.g. character `3` comes in context 3. But prediction can not be 100 % for one symbol as other symbols could occur and there has to be a probability greater zero assigned to them. Also whether the end of the message is reached is encoded as a special symbol. So the range of the arithmetic encoder gets still narrowed when a perfectly predicted digit is encoded, but on such a small level that still only 121 bytes are needed for the ZPAQ archive consisting of the CM model configuration, *hcomp* bytecode and the arithmetically coded ten thousand digits of π. That shows that we can go much beyond the entropy limit down to Kolmogorov complexity by using context modeling and adaptive arithmetic coding. And still the context model is usable for all input data in opposition to computing π in *pcomp*. The overhead of the fact that 100 % can not be used when predicting seems to be linear for the message size and can be observed when compressing 50 or 200 MB zeros with a CM, resulting in around 0.0000022 byte per input byte.

```
(pi10k.cfg 2016 Kai Lüke and Matt Mahoney
instead of generating the digits in pcomp phase, use the next expected digit as context
To compress: zpaqd cinst pi10k.cfg pi10k.zpaq pi10000.txt)

comp 0 14 0 0 1  (2^14 > 10000)
  0 cm 13 0
hcomp
  ifnot (only first run)
    (Compute pi to 10000 digits in M using the formula:
       pi=4; for (d=r1*20/3;d>0;--d) pi=pi*d/(2*d+1)+2;
     where r1 is the number of base 100 digits.
     The precision is 1 bit per iteration so 20/3
     is slightly more than the log2(100) we need.)
    a= 100 a*=a r=a 1 (r1 = digits base 100)
    a*= 7 d=a (d = n iterations)
    *b= 4 (M=4)
    do

      (multiply M *= d, carry in c)
      b=r 1 c=0
      do
        b--
```

[2] http://corpus.canterbury.ac.nz/descriptions/#misc

[3] http://mattmahoney.net/dc/pi.cfg

```
        a=*b a*=d a+=c c=a a%= 10 *b=a
        a=c a/= 10 c=a
      a=b a> 0 while

      (divide M /= (2d+1), remainder in c)
      a=d a+=d a++ d=a
      do
        a=c a*= 10 a+=*b c=a a/=d *b=a
        a=c a%=d c=a
      a=r 1 b++ a>b while
      a=d a>>= 1 d=a

      (add 2)
      b=0 a= 2 a+=*b *b=a
    d-- a=d a== 0 until
    halt
  endif
  a=*b a<<= 9 *d=a b++ (set context for expected digit taken from M)
  halt
end
```

Listing 3.1: pi10k.cfg

Even if this configuration is usable for other input than π it does not give good compression. It can be merged with the general text model from Listing 2.1 and change between using the CM for order-1 contexts and for the expected digits contexts every time the start of π is detected until a mismatch is found. This way all occurrences of π are coded with only a few bits.

```
(mixed_pi2.cfg
use the next expected digit as context for CM or a general text model of fast.cfg
a MIX2 will select between them
To compress: zpaqd c mixed_pi2.cfg text_pi.zpaq text_with_appearance_of_pi.txt
)
comp 2 14 0 0 4  (2^14 > 10000)
  0 cm 18 0   (order 1 or digits of pi)
  1 icm 16    (order 2, chained to isse)
  2 isse 19 1 (order 4)
  3 mix2 0 0 2 24 0 (moderate adapting mixer between CM and ISSE based on which predicts
    better)
hcomp
  r=a 2
  a=r 0
  a== 0 if (only first run)
    (Compute pi to 10000 digits using the formula:
       pi=4; for (d=r1*20/3;d>0;--d) pi=pi*d/(2*d+1)+2;
     where r1 is the number of base 100 digits.
     The precision is 1 bit per iteration so 20/3
     is slightly more than the log2(100) we need.)
    a= 100 a*=a r=a 1 (r1 = digits base 100)
    a*= 7 d=a (d = n iterations)
    *b= 4 (M=4)
    do
      (multiply M *= d, carry in c)
      b=r 1 c=0
      do
        b--
        a=*b a*=d a+=c c=a a%= 10 *b=a
        a=c a/= 10 c=a
      a=b a> 0 while

      (divide M /= (2d+1), remainder in c)
      a=d a+=d a++ d=a
      do
        a=c a*= 10 a+=*b c=a a/=d *b=a
        a=c a%=d c=a
      a=r 1 b++ a>b while
      a=d a>>= 1 d=a

      (add 2)
      b=0 a= 2 a+=*b *b=a
    d-- a=d a== 0 until
    c= 2 (point to 4 of 3.14)
    a= 1
    r=a 0
    a<<= 14 a-- (last element of ring buffer)
    b=a
    a-= 4 (first element of ring bufer, pointer in r3)
    r=a 3
    halt (input 0 came from pcomp, also to restart c=2 is enough)
  endif
  (CM part)
  d=0
  a=r 2  a-= 48
  c--
  a==*c
  c++
  if (pi: set context for expected digit)
    a=*c c++ a<<= 1 a++ a<<= 9 *d=a (distinguish between pi number context and character
    context by 1 bit for sure)
  else (other:)
    a=r 2 a<<= 10 *d=a c= 2 (set context to actual byte)
  endif
  (a in r2, lower border of ring buffer in r3)
  (ICM and ISSE part adapted from fast.cfg)
  a=r 2
  *b=a a=0 (save in rotating buffer M)
  hash b--
    d=a (save hash) a=r 3 a>b if b++ b++ b++ b++ endif a=d
  hash d= 1 *d=a
  b--
    d=a (save hash) a=r 3 a>b if b++ b++ b++ b++ endif a=d
  hash b--
    d=a (save hash) a=r 3 a>b if b++ b++ b++ b++ endif a=d
  hash d= 2 *d=a

  halt
end
```

Listing 3.2: mixedpi2.cfg

For real-life use cases it is often not possible to give perfect predictions. Good contexts can help to bring order into the statistics about previous data. Beside the manual approach heuristic context models can be generated for data by calculating the data's autocorrelation function [12] or as done in PAQ by recognizing two-dimensional strides for table or graphical data.

Even compressed JPEG photos can be further compressed by 10% - 30% by predicting the Huffman-coded DCT coefficients when using the decoded values as contexts (done in PAQ7-8, Stuffit, PackJPG, WinZIP [7]). The ZPAQ configuration `jpg_test2.cfg` uses a preprocessor to expand Huffman codes to DCT coefficients and later uses them as contexts[4]. The PackJPG approach continues to be developed by Dropbox under the name lepton[5] and supports progressive beside baseline JPEGs.

Overall modeling and prediction are an AI problem because e.g. for a given sentence start a likely following word has to be provided or how a picture with a missing area is going to continue. Remarkable results have been accomplished by using PAQ8 as machine learning tool for e.g. building a game AI with it serving as a classifier, for interactive input text prediction, text classification, shape recognition and lossy image compression [13].

[4] `http://mattmahoney.net/dc/zpaqutil.html`

[5] `https://github.com/dropbox/lepton`

4 Compilers, Rust and Python

Compilers and their parts covering the subproblems are mentioned here. Then the programming language Rust which was used for the compiler development is shortly presented as well as Python as the source language.

4.1 Classical Compiler Architecture

The task to translate a program from a source language into an equivalent program in the target language is usually split into independent phases where each phase passes its result to the next phase (see [14], Figure 1.6). In most cases the target language is very low level, like machine instructions of a CPU. One variation of a compiler structure including an intermediate representation language will be explained.

The tokenizer or lexer cuts the input stream into tokens which represent words/units of the source language which are normally the allowed terminal symbols of the grammar. The parser reads the tokens in sequence and fits them into nonterminals of the source language grammar by following the production rules until all tokens are consumed and the start production is completed. On the way it generates an AST (Abstract Syntax Tree) as semantic representation of the program in the source language. This completes the analysis done in the frontend.

Synthesis takes place in the backend. The AST is traversed to generate IR (intermediate representation) instructions. This step usually keeps track of variable names and their assignments in a symbol table. An optimization pass can be applied to the IR. The code generator produces instructions in the target language and they can be passed through optimization steps as well.

Checking for errors has to be done in every analysis step to find lexical errors in the tokenizer, syntactical errors while parsing, type errors and others during semantic analysis.

If the target language is not the end product but only an assembly language than an assembler produces object files which will be combined by a linker to a final executable program.

There are also preprocessors for the source language, multi-pass compilers and JIT (just-in-time) compilers which are used in interpreters and translate only an often-used section of the source program during runtime for a performance gain.

4.2 The Rust Programming Language

»Rust is a systems programming language focused on three goals: safety, speed, and concurrency. It maintains these goals without having a garbage collector, making it a useful language for a number of use cases other languages aren't good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races. Rust also aims to achieve 'zero-cost abstractions' even though some of these abstractions feel like those of a high-level language. Even then, Rust still allows precise control like a low-level language would.«[1]

The first stable 1.0 release of the Mozilla-sponsored language was in May 2015, the current version is 1.11 from August 2016. Pattern matching, definition of data structures and traits together with strong types bring Rust close to functional languages like Haskell. A program can in most cases only be compiled if it is free from crashes and undefined behavior unless explicitly a panic is allowed when an alternative result case of a function is not handled. So sometimes there are many iterations until a program compiles, but then as a gain there are assurances for the execution like memory safety instead of buffer overflows. But on the other side one has to be aware of the flow and reference of data through variables and has to comply to the rules imposed by the move semantics of the language, because the ownership model is the core for the safety guarantees.

Part of the language is the definition of test functions for software testing and inline documentation which can include test cases. As tooling the build system and package manager Cargo comes along with the compiler rustc which is invoked by Cargo. The inline documentation can be extracted to HTML.

Because of the noted advantages it seems that Rust fits well for compiler construction and offers a pleasing workflow once accustomed with its type system because of the more rare runtime errors leading to less debugging.

4.3 The Python Programming Language

Python has been around for two decades now and is still representative for an easy to learn imperative language with a simple syntax. It features dynamic types and a pragmatic approach to object orientation with functional elements and closures. Freeing memory is done by reference counting and the language is tightly coupled with the way it is interpreted. In order to fulfill a protocol/trait it relies on functions an object needs to implement.

[1] Book Introduction ”The Rust Programming Language”: `https://doc.rust-lang.org/book/index.html`

5 Considerations and Challenges with ZPAQL as Target Language

The ZPAQL bytecode that runs on the virtual machine is executed for every input byte, from the beginning i.e. the first instruction. Then A is set to the input byte while all other state is maintained between the executions. Also if running as *hcomp* section then the first elements of H are not general purpose memory but serve as input for the context mixing components. Execution in a ZPAQL VM is sequential with no parallelism. Beside the defined size of H and M there is no heap memory allocation or stack available. If dynamic memory allocation is wanted it has to be implemented on top of H or M.

It is Turing-complete but the bytecode size is restricted to around 64 KB (in *hcomp* few bytes less as the context model needs to be defined before). Also there is no keyword for a data section defined in the ZPAQL assembly language and it would be of no use as the bytecode content itself is not addressable and also unmodifiable. But program data could be prepended before the stored data and retrieved by reading in *hcomp* or *pcomp* and discarded while *pcomp* generates the output. If data should just be accessed by *hcomp* it could also be stored as a skipped part of the *pcomp* bytecode as this is seen by *hcomp* before the decoded data. This could also be used if an interpreter is written in ZPAQL.

The jump instructions only support a fixed and not a variable destination. That means instead of jumping to a return address which the caller passes to the function the return can only be achieved through an identifier for a jump table with predefined jump points.

The instruction set is rather simple and aimed at 32-bit integers. To keep expensive abstractions away, the source language should e.g. not include floating point numbers or 64-bit integers. Most calculations have to be done using register A as accumulator because the others lack advanced instructions.

There is no instruction for reading the next byte or saying that context computation has finished - the program needs to halt and is then executed again from the beginning. Therefore, additional runtime code is needed to continue execution where it stopped if e.g. convenient input reading should be allowed without disrupting the program flow or the finished computation of context data should be signaled.

It seems to be a viable plan to keep the organizational structure of a source program close to the resulting ZPAQL program but with added convenience through a runtime API which is anyway needed for input and output. Instead of introducing special functions to set the context data as first elements in H, H and also

M can just be fully exposed as data structure in the source language and also be used as arrays for other means with low abstraction costs. This keeps the structure of handwritten ZPAQL programs close to those in the source language. But in order to keep variables in a stack, *R* with its 256 elements is not enough, so expanding *H* seems to be a good solution. To model the repetitive execution of *hcomp* and *pcomp* they could be defined as functions in the source program (think main function) and it would also be possible to pass the input byte as argument which also keeps the similarity to a handwritten ZPAQL source.

As the runtime code abstractions for providing the mentioned read-API are not too high, the similarity to original ZPAQL files is more a cosmetic design decision. And if a context-set-API which halts and continues execution through runtime code is present then *hcomp* and *pcomp* functions could be replaced by main functions which are entered only once and thus hide the fact that execution starts from the beginning for every input byte. Still dynamic memory management on top of *H* and *M* seems to be costly and thus departing to far from ZPAQL and adding more complicated data structures could hurt performance too much.

It would be helping if the source program is standalone executable without being compiled to ZPAQL to ease debugging by staying outside the ZPAQL VM as long as possible.

Before ZPAQL instructions are generated by the compiler it would be helpful if most complicated operations are solved on the IR level already, like the saving, restoring and other memory management.

6 Selection of the Python-subset as Source Language

Considerations are mentioned again before the chosen subset is presented together with the API functions and organizational source code constraints. Then the grammar is listed with a short note on the behavior of elements.

6.1 Decision on the Feature Set

Based on the preceding thoughts the input should be a single Python file containing both the code for *hcomp* and *pcomp* as functions which could also call other functions but no imports are allowed. The compiler should not deal with strings, lists, arbitrary big numbers, classes, closures and (function) objects but only support a reduced set with variables having 32-bit integer values. The context model should be defined like in a ZPAQL configuration and the predefined arrays H and M should be exposed as well as the size exponents hh, hm, ph, pm and n as number of components used.

API functions `out(a)` for the *pcomp* `out` instruction and `read_b()` to read the next input byte by halting and returning to the current state of execution should be provided. As well as helpers if a dynamic memory management for additional arrays is implemented in Python on top of H and M. This API needs to be included as corresponding ZPAQL runtime in the output of the compiler. The Python runtime should also allow execution as standalone program with a similar interface to the zpaqd development tool which can run *pcomp* and *hcomp* on a given input.

All the code segments need to be separated, namely the context model definition and other common definitions/functions, *pcomp* and *hcomp* code with their variables and functions and then the runtime API and finally the code for standalone execution. Thus a template approach was chosen which uses comment lines as separation marks. Then the compiler extracts only the sections it needs to compile.

source.py Template Sections	Editable?
Definition of the ZPAQ configuration header data (memory size, context mixing components) and optionally functions and variables used by both *hcomp* and *pcomp*	yes
API functions for input and output, initialization of memory	no
function *hcomp* and associated global variables and functions	yes
function *pcomp* and associated global variables and functions	yes
code for standalone execution of the Python file analog to running a ZPAQL configuration with `zpaqd r [cfg] p|h`	no

Of course it could be organized in a different way that is more appealing based on the alternatives mentioned in the previous chapter e.g. with two more idiomatic Python files for *hcomp* and *pcomp* without special entry functions and the runtime in a different file which either imports the two files or vice versa.

6.2 Grammar and Semantics

The Python grammar as specified in the language reference[1] has been simplified where possible. It still allows dictionaries and strings, however not for program code but just for the definition of the context mixing components. Tuples, unpacking, lists and list comprehensions, for-loops, with-blocks, support for async-coroutines, import, try/raise/except, decorators, lambda expressions or named arguments have been removed.

Not all what is parsed is allowed as source code, e.g. nonlocal, dicts, strings or the @-operator for matrix multiplication but therefore a better error message can be provided than just a parser error. The names of the productions have been kept even if they are simplified. It would be best if the grammar is expanded to parse full Python again and let the code generator decide what to support.

Grammar with NUMBER, NAME, ”symbols”, NEWLINE, INDENT, DEDENT or STRING as terminals

Prog	(NEWLINE* stmt)* ENDMARKER?
funcdef	”def” NAME Parameters ”:” suite
Parameters	”(” Typedargslist? ”)”
Typedargslist	Tfpdef (”=” test)? (”,” Tfpdef (”=” test)?)* (”,” (”**” Tfpdef)?)?
Tfpdef	NAME (”:” test)?
stmt	simple_stmt | compound_stmt
simple_stmt	small_stmt (”;” small_stmt)* ”;”? NEWLINE
small_stmt	expr_stmt, pass_stmt, flow_stmt, global_stmt, nonlocal_stmt
expr_stmt	(store_assign augassign test) | ((store_assign ”=”)? test)

[1] `https://docs.python.org/3/reference/grammar.html`

store_assign	NAME (”[” test ”]”)?
augassign	”+=” | ”-=” | ”*=” | ”@=” | ”//=” | ”/=” | ”%=” ”&=” | ”|=” | ”^=” | ”<<=” | ”>>=” | ”**=”
pass_stmt	”pass”
flow_stmt	break_stmt | continue_stmt | return_stmt
break_stmt	”break”
continue_stmt	”continue”
return_stmt	”return” test
global_stmt	”global” NAME (”,” NAME)*
nonlocal_stmt	”nonlocal” NAME (”,” NAME)*
compound_stmt	if_stmt | while_stmt | funcdef
if_stmt	”if” test ”:” suite (”elif” test ”:” suite)* (”else” ”:” suite)?
while_stmt	”while” test ”:” suite (”else” ”:” suite)?
suite	simple_stmt, NEWLINE INDENT stmt+ DEDENT
test	or_test
test_nocond	or_test
or_test	and_test (”or” and_test)*
and_test	not_test (”and” not_test)*
not_test	comparison | (”not” not_test)
comparison	expr (comp_op expr)*
comp_op	”<” | ”>” | ”==” | ”>=” | ”<=” | ”!=” | ”in” | ”not” ”in” | ”is” | ”is” ”not”
expr	xor_expr (”|” xor_expr)*
xor_expr	and_expr (”^” and_expr)*
and_expr	shift_expr (”&” shift_expr)*
shift_expr	arith_expr | (arith_expr (shift_op arith_expr)+)
shift_op	”<<” | ”>>”
arith_expr	term | (term (t_op term)+)
t_op	”+” | ”-”
term	factor (f_op factor)*
f_op	”*” | ”@” | ”/” | ”%” | ”//”
factor	(”+” factor) | (”-” factor) | (”~” factor) | power
power	atom_expr (”**” factor)?
atom_expr	(NAME ”(” arglist? ”)”) | (NAME ”[” test ”]”) | atom
atom	(”(” test ”)”) | (”” dictorsetmaker? ””) | NUMBER | STRING+ | ”...” ”None” | ”True” | ”False” | NAME
dictorsetmaker	dictorsetmaker_t (”,” dictorsetmaker_t)* ”,”?
dictorsetmaker_t	test ”:” test
arglist	test (”,” test)* ”,”?

The semantics of the language elements as described in the reference[2] stay mostly the same, even if the usable feature set is still reduced as stated before. In particular, one has to be aware of integer overflows which are absent in Python but are present in ZPAQL and thus all computations are in $\mathbb{Z}_{4294967296}$ i.e. $mod\ 2^{32}$. Except for the bit shift operations with a shift of more than 32 bits. In this case the Python-subset will do a shift by $X\ mod\ 32$ bits. To achieve the semantics of `(v << X) % 2**32` or `(v >> X) % 2**32` with $X > 31$ the resulting value should directly be set to 0 instead. Also `/ 0` and `% 0` does not fail in ZPAQL but results in a 0 value.

For a wrong Python input the current compiler specification might behave in a different way and even accept it without a failure. Therefore it is required that the input is a valid Python program which runs without exceptions.

This requirement is also important because the current compiler does not check array boundaries, so `index%len(hH)` or `index&((1<<hh)-1)` should be used e.g. for a ring buffer because after the original size of H there comes the stack. If run as plain Python file, an exception is thrown anyway then because it checks array boundaries.

[2] `https://docs.python.org/3/reference/index.html`

7 Design of the Compiler in Rust

The source code for the **zpaqlpy** compiler is located at `https://github.com/pothos/zpaqlpy` and includes the test data and other files mentioned. This chapter is about the developed API the source file uses and about the process of developing the compiler and its internals.

7.1 Exposed API

The 32- or 8-bit memory areas H and M are available as arrays `hH, pH, hM, pM` depending on being a *hcomp* or *pcomp* section with size $2^{hh}, 2^{hm}, 2^{ph}, 2^{pm}$ defined in the header as available constants `hh, hm, ph, pm`. There is support for `len(hH), len(pH), len(hM), len(pM)` instead of calculating 2^{hh}. But in general instead of using `len()` for dynamically allocated arrays as well, special functions like `len_hH()` are used to visibly expose their types and do runtime checks already in Python. `NONE` is a shortcut for $0 - 1 = 4294967295$.

Other functions	Description
`c = read_b()`	Read one input byte, might leave VM execution to get the next input byte before return
`push_b(c)`	Put read byte c back, overwrites if already present (no buffer)
`c = peek_b()`	Read but do not consume next byte, might leave VM execution to get the next input byte before return
`out(c)`	In *pcomp*: write c to output stream
`error()`	Execution fails with ”Bad ZPAQL opcode”
`aref = alloc_pH(asize), …`	Allocate an array of size *asize* on `pH/pM/hH/hM`
`aref = array_pH(intaddr), …`	Cast an integer address back to a reference
`len_pH(aref), …`	Get the length of an array in `pH/pM/hH/hM`
`free_pH(aref), …`	Free the memory in `pH/pM/hH/hM` again by destructing the array

If backend implementations `addr_alloc_pH(size), addr_free_pH(addr), …` are defined then dynamic memory management is available though the API functions `alloc_pM` and `free_pM`. The cast array_pH(numbervar) can be used to save a type check in ZPAQL at runtime. Also in plain Python the cast from an address is needed after an array reference was itself stored into H and thus became an address number and is then retrieved as a number again instead of a reference. In general, there are no boxed types but by context a variable is used as address.

The last addressable starting point for any list is $2147483647 == (1<<31) - 1$ because the compiler uses the 32nd bit to distinguish between pointers to M and H.

The provided implementations of addr_alloc_pM, addr_free_pM, … can be found in the template (run ./zpaqlpy --emit-template or see src/template.rs). The returned pointer is expected to point at the first element of the array. One entry before the first element is used to store whether this memory section is free or not. Before that the length of the array is stored, i.e. $H[arraypointer - 2]$ for arrays in H and the four bytes $M[arraypointer - 5]...M[arraypointer - 2]$ of the 32-bit length for arrays in M.

Beside these constraints the implementations are free how to find a free region. The example uses getter and setter functions for the 32-bit length value as four bytes in M. For allocation it skips over the blocks from the beginning until a sufficiently sized block is found. If this block is bigger then the rest of it is kept free and might be merged with the next block if it is also free. That also happens when a block is freed again, then it is even merged with the previous block if that is free.

7.2 Parser and Tokenizer

Tokens are known Python keywords and symbols like operators or brackets, names or strings and numbers as converted values. Initially the Python module *tokenize* was used through piping source code in python3 -m tokenize -e and processing its output to build tokens as a Rust data structure (see src/tok.rs). Later on the code commonly residing in /usr/lib/python3.5/tokenize.py and /usr/lib/python3.5/token.py was ported to Rust (see src/rtok.rs) in order to be independent from external calls. It is assumed that the input is UTF-8 without a BOM (Byte Order Mark).

The parser is built with a parser generator out of the grammar specification. It constructs the AST with each production. The parser library lalrpop[1] was chosen in LALR(1) recursive ascent mode for that purpose.

The elements of the produced AST are based on the abstract grammar of the Python module *ast*[2] but simplified (see src/ast.rs) and give a structured representation of the source program. In fact it is not a tree but a list of statements. They can be one of FunctionDef which holds the function body as list of statements, Return, Assign and AugAssign which hold the expressions that are concerned, While and If which hold their body and else-part as list of statements and the test as expression, Global, Pass, Break, Continue and Expr which encapsulates an expression. These expressions can be one of BoolOpE to evaluate AND and OR

[1] https://github.com/nikomatsakis/lalrpop

[2] https://docs.python.org/3.5/library/ast.html#abstract-grammar

over expressions, `BinOp` for evaluation of an arithmetic expression over two expressions, `UnaryOpE` for an unary operation, `Compare` for comparisons over expressions, `Call` for a function call including arguments as expressions, `Num` for an integer, `NameConstant` for True and False, `Name` for a variable or `Subscript` for index access to an array variable. The precedence of `and` and `or` is resolved during parsing into a binary tree of `BoolOpE` elements. Contrary to that one has to be aware that with `Compare` the semantics of `a == b == c` and `(a == b) == c` differ, the middle operand is split up and passed to the next comparison. Their results are evaluated and merged with `and` until the result can not be `True` anymore, so this was better left to the IR generator.

7.3 Grammar of the Intermediate Representation

The IR was chosen to be close to ZPAQL but easier to write. While ZPAQL has the registers $A, B, C, D, F, R0 \ldots 255$ and arrays H and M the IR gives only control about R, H and M and no other registers. There are no new data structures added. So the other registers can be used for address computation and temporary calculations when the IR is converted to ZPAQL. That means the temporary variables `t0…t255` of the IR are a direct mapping to registers R_i. Because the input byte is in A at the beginning of *hcomp/pcomp* execution the IR relies on the guarantee that $R_{255} = A$ before the first instruction.

stmt	var ”=” var (op var)? var ”=” uop var ”if” var ”goto” label ”ifN” var ”goto” label ”ifEq” var var ”goto” label (to be used for optimizations) ”ifNeq” var var ”goto” label ”goto” label ”:” label ”:” ”halt” ”error” ”out” var
var	t | ”H[” t ”]” | ”H[t0+” x ”]” | ”H[t252+” x ”]” | ”H[” x ”]” | ”M[” t ”]” | ”M[” x ”]” | x
op	”+” | ”-” | ”*” | ”/” | ”//” | ”%” | ”**” | ”<<” | ”>>” | ”|” | ”^” ”&” | ”or” | ”and” | ”==” | ”!=” | ”<” | ”<=” | ”>” | ”>=”
uop	”!” | ”~” | ”-”
t	”t0” | … | ”t255”
x	”0” | … | ”4294967295”
label	[a-z_0-9~A-Z]+
comment	”#…\n”

The `var` on the left side of an assignment can not be a number. The operators `or` and `and` differ from the binary versions `|` and `&` as they represent the semantics of Python `or` and `and` i.e. they evaluate to the original value and not simply to the boolean choices of 1 and 0 for `True` and `False` while the binary operators use bitwise AND and OR. Operator `!v` tests against `v==0` while `~v` inverts the bits.

The choices made might not be the best and a totally different IR is possible which could introduce variables or stack operations. For efficiency it might be interesting to go into the direction of static single assignment with linear scan register allocation or an algorithm with graph-coloring register allocation. Currently local Python variables are on the stack and temporary variables are in R because together they could be more than 256 and anyway local variables need to be stored on stack before a call. But they could be joined in a common pool and the current limit of maximum 256 temporary variables in R could be widened by using elements of H as needed. Maybe even LLVM as a very popular IR could be used with its many optimization passes available.

7.4 IR Generation and the Stack

Due to the reduced features in the subset the compilation of the Python source can be done in a similar way a C source would be compiled. The local variables of a function are held on the stack which is produced by expanding H beyond its defined size for the configuration. Global variables are held in the beginning of the stack. The temporary variables `t0…t251` are used for intermediate or address computations whereas `t0` is the base pointer for the stack and `t252` is a copy of the global base pointer. `t255` holds the last input byte, `t254` the reading state and `t253` the read byte for the API function `read_b()` which stops the execution and returns to the caller with the newly acquired byte when the bytecode is run again.

The original size $x = 2^{hh}$ of H is extended to hold the stack of size y. To calculate a valid resulting size of $2^{\lceil z \rceil} \geq x + y$ the formula $z = log_2(x + y) = log_2(x) + log_2(1 + \frac{y}{x})$ is used.

There are no IR instructions for calling a function or stack operations on H, but there are helping meta instructions. These will be converted to simple IR instructions and have been defined for handling blocks, saving and loading variables on the stack in H, calls and returns to comply to a calling convention, predefined IR code sections for the runtime API and the jumping table to continue execution after a return. The initial IR code is responsible to either call the Python functions `hcomp(c)/pcomp(c)` with the new input byte or continue execution if it was interrupted through a `read_b()` API call. Also it sets the base pointer for the first run and defines the API function `read_b`.

The function `traverse()` in `src/gen_ir.rs` generates IR instructions for the given (part of an) AST. It is recursively used and consults the symbol table for free temporary variables, position of local variables and the mapping of global variables. Also it uses the recursive function `evaluate()` which takes only an expression part of an AST and returns IR instructions and the IR variable that holds the value after these instructions have been executed. Returning the instructions makes testing easier. If `pcomp(c)/hcomp(c)` only contain a `pass` then the whole *pcomp/hcomp* section is omitted.

The temporary variables have to be saved on the stack before a call. Also the current base pointer and then the return ID for the jump table need to be saved there as part of the calling convention. The new base pointer in `t0` points at the return ID. Arguments passed come afterwards and the called function will address them via $H[t0+x]$. On return the previous base pointer will be restored to `t0` and the return ID is copied in `t2` for the jumper table while the return value is in `t1` before the jump to the code for the jump table is done in order to return after the call instruction.

It should be possible to treat local variables like temporary variables to reduce code size and stack usage by a compiler flag. Also the calling convention could be changed to avoid the stack and globals could be stored in tx.

For now the compiler passes IR code around as vectors (lists), but for a more idiomatic Rust style iterators could be used.

7.5 Optimizations in the Intermediate Representation

Unused functions occupy space in the bytecode and hence are removed by searching for calls and if none is found it can be removed. This is repeated until no function was removed anymore. Assignments which are unused or could be merged into one assignment are not yet recognized.

During IR generation it is not known whether a temporary variable really needs to be saved on the stack before a call or whether it is not in use afterwards anyway. Therefore the helper macro instructions `MarkTempVarStart` and `MarkTempVarEnd` are inserted around each function scope. Saving and restoring temporary variables is done through the macros `StoreTempVars{identifiers}` and `LoadTempVars{identifiers}`. The optimization pass goes through the IR instructions in reverse order to reason about the lifetimes and removes the identifiers from the load and store macro instructions which are not live i.e. referred to after the load.

Compiler books have much more to offer and there are many classical optimizations which could be looked on for further improvements. Caching could be used and constant expressions evaluated during IR generation. This is just done for a row of assignments on allocated arrays in M.

7.6 ZPAQL Generation and Register Assignment

The IR was chosen to be easily convertible to ZPAQL by using the registers A, B, C and D to calculate and move values between H and R. Thus for the beginning a very simple but inefficient solution was taken. The function `emit_zpaql()` in `src/gen_zpaql.rs` takes IR code and yields ZPAQL code by calling `assign_var_to_a(var)` for the operands (if needed saving one value from A in C), and applying the operator on them. The result is moved from A to the target by calling `assign_a_to_var(var)`. In order to load a number in A `calc_number(value)` is used and for values greater 255 multiple bit shifts are needed. All of this helper functions emit ZPAQL code which is combined as result of each function in a similar way to the IR generation.

Then a more advanced solution was chosen with pattern matching to avoid the ubiquitous transfer over A and detect an augmented assignment like `+=1` and simply increase the value at its location. Therefore the helper function `gen_loc_for_var()` was introduced to get the location handle and the other helpers were widened to operate with the notion of a location meaning registers or memory positions through pointers instead of only A. Tracking recent values of variables in the registers or memory locations for reuse was also helpful in order to produce less code output. Yet it is a more careful issue now to make changes because the cache entries concerning the location and the variable need - at least - to be invalidated for a correct output.

After the code generation a special optimization takes place to simplify byte assignments on arrays in M. They have been produced in the IR generation if a row of successive assignments like in an initialization was detected. Thus the pointer variable is always just increased and even not saved on R until all assignments are finished.

Resolving labels to positions for the jump destinations is done as last step before the ZPAQL assembly code is written out. The opcode size for each instruction is known and thus the label's positions can be hold in a hash table. The second pass through the code can then replace the virtual `GoTo` placeholder instruction with the real long jump.

7.7 Debugging

It is important to run the plain Python version because it contains assertions and test it before trying to compile it to ZPAQL. For the Python runtime a compare option was introduced to test the correctness of the *pcomp* code which should restore the preprocessor input. Each output byte is compared to the expected output and a debugger shell spawned if a mismatch occurs.

Because compiler optimizations are likely to introduce bugs there is a small test suite in the Makefile. It compares the output of the Python code for *pcomp* and *hcomp* to the output of running the compiled ZPAQL code. Mostly `zpaqd r CFG p` is used but also a ZPAQ VM implementation in Rust included in the compiler (option `--run-hcomp`) because `zpaqd r CFG h` does not print $H[0], ..., H[n-1]$.

8 Exemplary Configurations and Programs

As a showcase of the benefits of having a compiler for a high-level language like Python this chapter presents and evaluates two new ZPAQ configurations. The first ia a new image compression model which can parse a header and combine various context predictors. And the second is an implementation of the Brotli decompression scheme in the *pcomp* phase.

8.1 Compression of PNM Image Data using a Context Model

PNM is a lossless image format which in variant P6 with a maximum color value of 256 saves 8-bit RGB color channels as bytes after a short header with height and width[1]. To predict the value of a color channel the value of the neighboring pixels can be used as context as far as they have been seen in the stream. L is referring to the left pixel, LU to the upper left, U to the pixel above and UR to the upper right.

For grayscale images in a custom format such a context model has already been developed[2] for ZPAQ and performed better than PNG compression. Even if it does not match the ZPAQ configuration `bmp_j4c.cfg`, it is therefore much faster because the context mixing network is simpler.

The PNM model `test/pnm.py` parses the header in Python to define the history buffer for one row and takes a similar approach as the grayscale model but with separated context models for each channel. They are mixed together also with an averaged value context model and indirect models. This mixed prediction is again combined by a mixer with the prediction of a chain of ISSE (with a hash of various contexts and the last byte as context) refining the prediction of an ICM (with an averaged value context).

Color transforms like the subtract green transform in WebP can improve compression by decorrelating colors because one color component is often also a good predictor for the others[3] and that link should be reduced to improve prediction for the same color. The PNM model can use it as pre- and postprocessor, mapping (r,g,b) to $(r-g,g,b-g)$ and back. The preprocessor is in the separate Python file `test/subtract_green`.

[1] `http://netpbm.sourceforge.net/doc/ppm.html`

[2] `http://www.modejong.com/blog/post15_zpaql_grayscale/index.html`

[3] `https://developers.google.com/speed/webp/docs/webp_lossless_bitstream_specification#subtract_green_transform`

To measure the compression ratio four image files were converted to PNM in `test/` and compressed (512 pixel high `peppers.pnm`, `monarch.pnm`, `kodim23.pnm`[4] and 1020 pixel high `rafale.pnm`[5]). FLIF is a new lossless format[6].

(a) kodim23 **(b)** peppers **(c)** monarch **(d)** rafale

Figure 8.1: Images of the benchmark (here in grayscale)

Method	Total file size in byte
ZPAQ archive with bmp_j4c.cfg (in BMP format, uses special color transform)	1785405
ZPAQ archive with pnm.cfg	1908258
FLIF	1914297
WebP	2256458
PNG	2938153

For now it does not use e.g. delta coding or other predictors beside the average from neighbor values. Also it could provide another predicted color value which is estimated by the recent development. Beside that the color transform should depend on the image or a way needs to be found to move its advantage out from the preprocessor into the predictors. Like the mentioned grayscale model it uses an additional context for the mixer depending on image noise.

By using Python the model can easily be improved and writing it in ZPAQL would have been a bigger an bug-prone effort. The bytecode overhead as of writing is around 3 KB.

For the large 53 MB photo `canon24.pnm`[7] it would reach the 6th place with 21,326,964 bytes (89 seconds for (de)compression) in the benchmark for lossless picture compression published in a paper about BWT image compression [15]. For `kodim23.pnm` it would reach the 4th place with an archive size of 369,519 bytes. An overall benchmark for a huge number of files like in [8] or [9] has not been done up to now.

[4] Commonly used for image compression benchmarks, available under `http://r0k.us/graphics/kodak/`

[5] `http://www.maximumcompression.com/data/bmp.php`

[6] `http://flif.info/`

[7] `https://web.archive.org/web/20140702040431/http://www.squeezechart.com/canon24.pnm`

[8] `http://www.squeezechart.com/bitmap.html`

[9] `http://imagecompression.info/gralic/LPCB.html`

8.2 Bringing the Brotli Algorithm to ZPAQ

The algorithm has to be implemented as postprocessor in *pcomp* and uses the Brotli compressor[10] as preprocessor. Because the dictionary itself is almost twice the maximum bytecode size allowed it has to be prepended before the first segment of the compressed Brotli data for being read into memory. Following Brotli streams as additional segments in the same block can use a byte different from the first dictionary byte as prefix instead of prepending the whole dictionary again.

Instead of porting the C library or writing a new implementation based on the Brotli specification [11] it was easier to port a decompressor written in Rust[11] to the Python-subset (see `test/brotli.py`). It works as standalone Python program `test/brotli.py` and can be compiled to ZPAQL with a bytecode size of 64870 bytes (ZPAQ config file `test/brotli.cfg`).

This means the size of an empty archive is already 184 KB if no context mixing is used for the dictionary and bytecode. A simple order-1 ICM model brings it down to 90 KB. The slightly more advanced model of `mfast.cfg` and its *hcomp* code reach 64 KB in total.

The overhead is less significant for bigger archives and trying to reduce it by applying context mixing to the whole block might not always improve the result because the Brotli data could be expanded in coding size. For the four sample PNM files it saves 133 KB which is almost the size of the dictionary and bytecode (from 3615 KB with no arithmetic coding down to 3482 KB in total). For the enwik8 benchmark[12] (100 MB Wikipedia dump) however this would not even make sense because `mfast.cfg` alone performs better by reaching 24720 KB instead of 30303 KB when used together with Brotli data and the dictionary/bytecode overhead. The same Brotli data without arithmetic coding through context model can be stored in around 30416 KB, depending on the Brotli compressor options it might go down to around 26000 KB.

Decompression is much slower than with the C or Rust Brotli implementations. The ZPAQL implementation needs 118 seconds for enwik8 instead of 0.5 or 2.1 seconds for the reference C implementation and the one of Mark Adler[13]. The memory management is not very efficient, specially dealing with byte arrays on M is expensive and the compiler could sometimes produce better code by avoiding the intermediate variables in R. Beside that ZPAQL is JIT-compiled by libzpaq in a rather simple way and so an advanced JIT backend on top of LLVM could help.

But it shows that ZPAQ can be used with a very different algorithm and it still maintains backwards compatibility. A ZPAQ decompressor could also detect if the embedded *pcomp* bytecode is in fact a Brotli implementation and skip its execution by using one of the faster C or Rust implementations. That way new algorithms become instantly available while new versions of decompressors can have a speed advantage by replacing the bytecode execution with the equivalent library call.

[10] `https://github.com/google/brotli`

[11] `https://github.com/ende76/brotli-rs`

[12] Used for the Hutter Prize compression challenge, comparison here: `http://mattmahoney.net/dc/text.html`

[13] `https://github.com/madler/brotli`

9 Evaluation

What follows in this chapter is a review on the outcome so far of working on a compiler as well as an attempt to provide one possible answer to the question whether — or how well — ZPAQ suits to be a general standard for data compression.

9.1 Compiler Construction

Rust provided a good environment to prevent multiple bugs by detecting them at compile time. Yet it takes up to four minutes for an iterative build which is also due to the big parser generated by lalrpop. Recently there is development in a table-driven mode which could provide faster compile times by producing less parser code. Also the upcoming MIR intermediate language will allow the Rust compiler to cache compiled code segments if they have not been altered. This way test-driven development would be more easy because of fast iterations.

Even if Rust provides inbuilt test support is has not been used as the compiled code output was mainly tested with `zpaqd`. This could be improved, specially with the small ZPAQL VM implementation that is already included and used for testing the correctness of computations through the Makefile. The code lacks inline documentation and the structure is sometimes not idiomatic to Rust. Error handling currently uses panics instead of propagation of success or failure via result types.

9.2 Performance of the Compiler

The choice of the Python-subset and IR seemed to be supporting a fast development of the compiler even if a lot of optimization passes are still missing. There might be compiler bugs which have not been discovered up to now as the test coverage is not too extensive.

While the original ZPAQL file for `test/lz1.orig.cfg` needs 251 bytes for an empty archive the port `test/lz1.py` compiled to bytecode needs 1255 bytes for an empty archive which is an overhead of around 1000 bytes compared to handwritten LZ1.

Runtime Benchmarks of the LZ1 Configuration in ZPAQL and Python for the four PNMs

ZPAQ config	Compression time (sec)	Archive size (bytes)	Decompression time (sec)
zpaqlpy lz1.cfg	1.6	3350016	1.85
handwritten lz1.orig.cfg	1.3	3349062	1.53

9.3 Analysis of the generated Code and Comparison with handwritten Code of LZ1

The original `test/lz1.orig.cfg` is very dense and tries to hold values only in the registers if possible. Only in the *pcomp* code two R variables are used along with M as history buffer. Sometimes variables are moved between the registers because certain operations are only available on A. Without comments it would be difficult to guess what it does and make changes.

On the other hand the compiled `test/lz1.cfg` from the Python version is rather large, specially when all comments are embedded which is the default. It does not harm the bytecode size but is useful for debugging and partly allows extraction of the original source as well as the IR code by invoking:

```
egrep '\( *[0-9]+: ' test/lz1.cfg                    # Python code
egrep '\( +' test/lz1.cfg | egrep -v '\( *[0-9]+: '    # IR code
```

The compiled file starts with the runtime code which decides whether it is the first run. For the first run it sets the base pointer and starts the global definitions before calling the `pcomp/hcomp` function with the input. When initialization did take place already then the code decides whether a read was interrupted to get a new input byte or if just the `pcomp/hcomp` function should be called again with the new input. As stated in the previous chapters all Python variables are in the stack and thus it takes more opcodes to do calculations on them. The code does not use custom arrays but instead used M as history buffer as the original code. Everytime when the function returns the jump table is used to jump behind the call and then the execution halts.

It is an advantage that changes can be made more quickly in Python and the code is more comprehensible. But the compiled code uses a stack while it is not really necessary and instead of using the jump table for the return it could just use the `halt` instruction. So for simple programs like that it would be desirable if the compiler offers a mode without a stack by holding variables in R.

9.4 Suitability of ZPAQ as universal Standard

ZPAQ serves well for research in context mixing algorithms and as development platform for custom algorithms. But whether it will replace general compression implementations in various libraries of deflate and newer variants is open to question. Because a huge memory allocation or an endless loop can be unsafe for decompressing untrusted archives, at least configurable limits need to be implemented into libzpaq. It might also not be perfect for universal use as the mixing type (logarithmic) is quite fixed. In addition to that the definition of an own encoder can not use the inbuilt context mixing. A component which takes something like $H[i] - 2048$ as stretched prediction value would be interesting (in a valid range to leave range space for the other bit). Also it is not possible to define new components which maybe could be moved in an additional section with its identifier mentioned. While *pcomp* gets notified about segments *hcomp* gets not which could also be of use.

As a compiler target the ZPAQL instruction set lacks a variable jump instruction like `jmpa` which would take the value of A as destination or `getpc` which sets A to the current program counter. It would be useful to set 32-bit values directly in one instruction like `a=L 3210987654`. Memory management on top of H/M is possible but a new data structure which allows using the OS heap would be more efficient. These proposals could be considered for the next level of the ZPAQ specification.

10 Conclusion

It was shown that it is possible to have a compiler for ZPAQL which is also helpful for development of new algorithms. To write the compiler large parts of the Python grammar have been ported to the format of the LALR(1) parser library lalrpop. The core functionality of the Python module *tokenize* has been ported to Rust. As a goal for a source input which should be supported a Rust implementation for Brotli decompression has been ported to Python. A compilation scheme from a Python-subset to ZPAQL including an IR has been planned and implemented. The compiler is platform independent as long as a Rust compiler is available. It features an implementation of the ZPAQ VM to print the calculated context data for each input byte which can then be compared with the calculated values of the Python source to expose ZPAQL specific semantics or even compiler bugs.

As a result it can be said that more different approaches should be tried to reach an acceptable performance for a large and complex code base. The ZPAQ specification does at its current point not offer various features which might be considered for future improvements, from additional instructions up to ways of being even more flexible on how prediction takes place. But most important would be a variable jump instruction.

Compilation can be configured with command line arguments to some extend. The *pcomp* or *hcomp* part can be disabled so no changes to the input file are needed to vary between using only context mixing, only a preprocessor or both. Documentation can be printed via arguments as well. Four example source files are provided: A small context mixing compression for run length encoded data (see appendix tutorial), the LZ1 port and the PNM model which already showed good results with less efforts. An extreme case is the Brotli algorithm which needed many compiler optimizations to fit under the 64 KB bytecode limit and utilizes dynamic memory allocation.

While the compiler was developed two bugs in ZPAQ tools were found and are already resolved through two new releases. One was a simple crash in zpaqd and the more serious one a wrong instruction in the x86 JIT code of libzpaq which caused a miscomputation in the Brotli decompressor.

For most results there have been measurements which can be repeated because all needed tools are published as free/libre software.

While PAQ and its internals were covered and modified in publications, ZPAQ was often only used as just another compressor instead of a platform for compression algorithms. This work exposed the crucial part of ZPAQ which is the embedding of the two bytecodes for *hcomp* and *pcomp* in the archive.

Bibliography

[1] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, Apr 1984. ISSN 0090-6778. doi: 10.1109/TCOM.1984.1096090. `http://dx.doi.org/10.1109/TCOM.1984.1096090`.

[2] G. V. Cormack and R. N. S. Horspool. Data compression using dynamic markov modelling. *Comput. J.*, 30(6):541–550, December 1987. ISSN 0010-4620. doi: 10.1093/comjnl/30.6.541. `http://dx.doi.org/10.1093/comjnl/30.6.541`.

[3] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664, May 1995. ISSN 0018-9448. doi: 10.1109/18.382012. `http://dx.doi.org/10.1109/18.382012`.

[4] Matthew V. Mahoney. Fast text compression with neural networks. In James N. Etheredge and Bill Z. Manaris, editors, *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference, May 22-24, 2000, Orlando, Florida, USA*, pages 230–234. AAAI Press, 2000. ISBN 1-57735-113-4. `http://www.aaai.org/Library/FLAIRS/2000/flairs00-044.php`.

[5] Matthew V. Mahoney. The PAQ1 data compression program. 2002. `https://cs.fit.edu/~mmahoney/compression/paq1.pdf`.

[6] Matthew V. Mahoney. Adaptive weighing of context models for lossless data compression, Florida Tech. Technical Report CS-2005-16. 2005. `https://cs.fit.edu/~mmahoney/compression/cs200516.pdf`.

[7] Matthew V. Mahoney. Data compression explained, 2010. `http://mattmahoney.net/dc/dce.html`.

[8] Matthew V. Mahoney. The ZPAQ open standard format for highly compressed data - Level 2. 2016. `http://mattmahoney.net/dc/zpaq206.pdf`.

[9] Matthew V. Mahoney. The ZPAQ compression algorithm. 2015. `http://mattmahoney.net/dc/zpaq_compression.pdf`.

[10] James K. Bonfield and Matthew V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PLoS ONE*, 8(3):1–10, 03 2013. doi: 10.1371/journal.pone.0059190. `http://dx.doi.org/10.1371%2Fjournal.pone.0059190`.

[11] Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, July 2016. `https://rfc-editor.org/rfc/rfc7932.txt`.

[12] John Scoville. Fast autocorrelated context models for data compression. *CoRR*, abs/1305.5486, 2013. `http://arxiv.org/abs/1305.5486`.

[13] Byron Knoll and Nando de Freitas. A machine learning perspective on predictive coding with PAQ. *CoRR*, abs/1108.3298, 2011. `http://arxiv.org/abs/1108.3298`.

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN 978-0321486813.

[15] Aftab Khan and Ashfaq Khan. Lossless colour image compression using RCT for bi-level BWCA. *Signal, Image and Video Processing*, 10(3):601–607, 2016. ISSN 1863-1711. doi: 10.1007/s11760-015-0783-3. `http://dx.doi.org/10.1007/s11760-015-0783-3`.

A Tutorial

A context mixing model with a preprocessor for run length encoding is written. Three components are used to form the network. Create a new template which will then be modified at the beginning and at the *pcomp/hcomp* sections:

```
$ ./zpaqlpy --emit-template > rle_model.py
$ chmod +x rle_model.py
```

First the size of the arrays H and M for each section, hcomp and pcomp needs to be specified:

```
hh = 2  # i.e. size is 2**2=4, because hH[0], …, hH[2] are the inputs for the components
```

Listing A.1: rle_model.py

One component should give predictions based on the byte value and the second component based on the run length, both give predictions for the next count and the next value. Then the context-mixing components are combined to a network:

```
n = len({
  0: ”cm 19 22”,  # context table size 2*19 with
                  # partly decoded byte as 9 bit hash xored with the context,
                  # count limit 22
  1: ”cm 19 22”,
  2: ”mix2 1 0 1 30 0”,
  # will mix 0 and 1 together, context table size 2**1 with and-0 masking of the
  # partly decoded byte which is added to the context, learning rate 30
})
```

Listing A.2: rle_model.py

Each component i gets its context input from the entry in $H[i]$ after each run of the `hcomp` function, which is called for each input byte of the preprocessed data, which either is to be stored through arithmetic coding in compression phase or is retrieved through decoding in decompression phase with following postprocessing done by calls of the `pcomp` function.

The context-mixing network is written to the archive in byte representation as well as the bytecode for hcomp and pcomp (if they are used). The preprocessor command is needed when the compiled file is used with zpaqd if a pcomp section is present. As the preprocessor might be any external programme or also included in the compressing archiver and is of no use for decompression it is therefore not mentioned in the archive anymore. This way we specify a preprocessor:

```
  pcomp_invocation = ”./simple_rle”
```

Listing A.3: rle_model.py

```
$ chmod +x simple_rle  # create the preprocessor as executable file and fill it as follows
```

```
#!/usr/bin/env python3
import sys
input = sys.argv[1]
output = sys.argv[2]
with open(input, mode='rb') as fi:
  with open(output, mode='wb') as fo:
      last = None
      count = 0
      data = []
      for a in fi.read():
        if a != last or count == 255:  # count only up to 255 to use one byte
          if last != None:  # write out the pair
            data.append(last)
            data.append(count)
          last = a  # start counting
          count = 1
        else:
          count += 1  # continue counting
      if last != None:
        data.append(last)
        data.append(count)
      fo.write(bytes(data))
```

Listing A.4: simple_rle

Then we need code in the pcomp section to undo this transform:

```
case_loading = False
last = NONE

def pcomp(c):
  global case_loading, last
  if c == NONE:  # start of new segment, so restart our code
    case_loading = False
    last = NONE
    return
  if not case_loading:  # c is byte to load
    case_loading = True
    last = c
  else:  # write out content of last c times
    case_loading = False
    while c > 0:
      c-= 1
      out(last)
```

Listing A.5: rle_model.py

So now it should produce the same file as the input file:

```
$ ./simple_rle INPUTFILE input.rle
$ ./rle_model.py pcomp input.rle input.norle
$ cmp INPUTFILE input.norle
```

We can already try it, even if hcomp does not compute the context data yet (so compression is not good):

```
$ ./zpaqlpy rle_model.py
$ ./zpaqd c rle_model.cfg archive.zpaq FILE FILE FILE
```

Now we can add hcomp code to improve compression by adaptive prediction:

```
at_counter = False  # if false, then c is byte, otherwise c is a counter
last_value = 0
last_counter = 0

def hcomp(c):  # pcomp bytecode is passed first (or 0 if there is none)
  global at_counter, last_value, last_counter
  if at_counter:
    last_counter = c
  else:
    last_value = c
  # first part of the context for the first CM is the byte replicated and
  # the second is whether we are at a counter (then we predict for a byte) or vice versa
  hH[0] = (last_value << 1) + at_counter  # at_counter will occupy 1 bit, therefore shift
  hH[0] <<= 9  # again shift to side because of the xor with the partially decoded byte
  # second CM same but uses the counter for prediction
  hH[1] = (last_counter << 1) + at_counter
  hH[1] <<= 9
  hH[2] = at_counter + 0  # context for mixer: is at counter (1) or not (0)
  at_counter = not at_counter
```

Listing A.6: rle_model.py

We need to compile again before we run the final ZPAQ configuration file:

```
$ ./zpaqlpy rle_model.py
$ ./zpaqd c rle_model.cfg archive.zpaq FILE FILE FILE
```

zpaqd needs to have `simple_rle` in the same folder because we specified `pcomp_invocation = ”./simple_rle”`.