AwkwardForth: accelerating Uproot with an internal DSL

Jim Pivarski[1,*], *Ianna* Osborne[1,**], *Pratyush* Das[2,***], *David* Lange[1,****], and *Peter* Elmer[1,†]

[1]Princeton University, Princeton, New Jersey, United States.
[2]Institute of Engineering and Management, Kolkata, West Bengal, India.

Abstract. File formats for generic data structures, such as ROOT, Avro, and Parquet, pose a problem for deserialization: it must be fast, but its code depends on the type of the data structure, not known at compile-time. Just-in-time compilation can satisfy both constraints, but we propose a more portable solution: specialized virtual machines. AwkwardForth is a Forth-driven virtual machine for deserializing data into Awkward Arrays. As a language, it is not intended for humans to write, but it loosens the coupling between Uproot and Awkward Array. AwkwardForth programs for deserializing record-oriented formats (ROOT and Avro) are about as fast as C++ ROOT and 10–80× faster than fastavro. Columnar formats (simple TTrees, RNTuple, and Parquet) only require specialization to interpret metadata and are therefore faster with precompiled code.

1 Motivation

Despite being written in Python, Uproot [1] can read simple data types from ROOT TTrees [2] as fast as any precompiled code because the values are already contiguous in a raw view of the file. No computations are required to get a ROOT TBasket of numerical data from disk into memory, as an array, except to swap endianness if little-endian arrays are desired. Python's NumPy [3] library casts data from raw bytes as a metadata-only operation, and performs operations that scale as $O(n)$, where n is the length of the array, in precompiled code.

For more complex data types, however, the cost of computing in Python increases. Variable-length ("jagged") arrays are relatively quick, since ROOT's TTree format separates the numerical content of these arrays from the integers that define the starting positions of each entry's array. This is nearly the format required by Awkward Array [4], and only needs minor arithmetic transformations. Objects with fixed-size headers and consisting of fixed-size fields, such as `std::vector<TLorentzVector>`, can be extracted using NumPy tricks, though these tricks require more intermediate arrays, further slowing the transformation. Finally, general objects with nested, variable-length data, the simplest of which is `std::vector<std::vector<float>>`, require custom code to parse each data type. In Python, such code is hundreds of times slower than the equivalent C++.

Previously, we quantified this slow-down [5] using ROOT TTrees containing `float`, `std::vector<float>`, and vectors of vectors up to three levels deep, reading them with

[*]e-mail: pivarski@princeton.edu
[**]e-mail: ianna.osborne@cern.ch
[***]e-mail: reikdas@gmail.com
[****]e-mail: david.lange@cern.ch
[†]e-mail: peter.elmer@cern.ch

the then-current Python codebase and with custom C++ code, which represents what is possible now that Awkward Array is implemented in C++. The read performance of `float` data is identical for Python and C++, the C++ is several times faster for `std::vector<float>`, and the gap widens to factors of hundreds for the doubly-nested and triply-nested cases. Leveraging Awkward Array's C++ layer to accelerate Uproot is a worthwhile goal.

However, unless we limit our attention to special cases like nested vectors of numbers, this deserialization code is not known at compile-time. The byte-for-byte layout of a complex data type is expressed as data, in the TStreamerInfo of a ROOT file, and is therefore discovered at runtime. Moreover, the specifics of ROOT deserialization should not be spread between two packages, Uproot and Awkward Array: Uproot should focus entirely on ROOT I/O as Awkward Array focuses on array manipulation. The problem is to satisfy three constraints:

1. Deserialization code must not be hundreds of times slower than compiled code.
2. This code must be generated at runtime from TStreamerInfo.
3. All "knowledge" of ROOT I/O must be in Uproot, not Awkward Array.

The first two constraints could be satisfied by just-in-time (JIT) compilation. This is, in fact, what ROOT's Cling compiler [6] does. The third constraint could be satisfied by adding a C++ layer to Uproot. Both of these solutions, however, would significantly complicate the distribution of the Uproot or Awkward Array packages, and portability is a high priority.

This paper describes a different solution to the constraints listed above, which does not affect the portability of Uproot or Awkward Array. We introduce AwkwardForth, a domain-specific language (DSL) for deserializing arbitrary data into Awkward Arrays that is runtime-interpreted but nearly as fast as compiled code. Unlike DSLs intended for humans to read and write, this DSL is "internal," only used to communicate between software packages. Uproot's task becomes one of expressing ROOT I/O logic in AwkwardForth and Awkward Array executes it, returning filled arrays.

2 AwkwardForth

AwkwardForth is a member of the Forth family of languages, which includes Postscript [7], another internal DSL. AwkwardForth adheres to a subset of the ANSI Forth Standard [8] and has extensions for interpreting arbitrary input buffers and filling columns for Awkward Array.

Like all Forths, AwkwardForth is primarily concerned with stack manipulation. The runtime environment features a stack of integers and programs define "words" that manipulate the stack (as well as input and output buffers, in the case of AwkwardForth). Each word consumes and produces an arbitrary number of arguments and return values through this stack, and hence Forth words are more general than functions in a typical programming language. Whereas functional programming languages eliminate or minimize side-effects, Forth acts exclusively through side-effects. As such, it is more like an extensible assembly language.

The popularity of Forth peaked in the early 1980's because its interactive interpreter could fit within the tight resource constraints of early personal computers [9]. This same interactive minimalism makes it an ideal candidate for running fast programs that must be generated at runtime, such as deserializing ROOT files. On a 2.2 GHz CPU core, AwkwardForth takes about 5 ns to evaluate each word, compared to about 900 ns for Python to evaluate a bytecode.

Though motivated by the problem of ROOT deserialization, AwkwardForth is intended for the general problem of deserializing non-columnar data formats into columnar Awkward Arrays. Many file formats, such as ProtoBuf [10], Thrift [11], Avro [12], FlatBuffers [13], and JSON [14], describe data structures in a record-oriented layout, with all fields of one record contiguous with each other, while columnar formats like simple TTrees, RNTuple [15], Parquet [16], Arrow [17], and Awkward Array place all values of one field contiguous with each other before moving on to the next field. Conversions between columnar formats can be very fast, sometimes casting, rather than copying, the columns. Record-oriented formats, on the other hand, must be fully parsed. In this paper, we examine AwkwardForth's deserialization performance for ROOT TTrees (columnar and record-oriented), Avro, and Parquet.

Awkward Array has a tool for converting arbitrary record-oriented data into Awkward Arrays: `ArrayBuilder` constructs arrays in an append-only order, driven by commands such as `begin`/`end_record`, switch to `field`, append `integer`, and `begin`/`end_list`. A consequence of this generality is that `ArrayBuilder` discovers the array's data type at runtime, adding output buffers as the observed type gets more complex. While this is great for JSON, type discovery is unnecessarily slow for formats whose type is known in advance, though perhaps not as early as compile-time. (Uproot currently uses `ArrayBuilder` in Python.) In this paper, we also present the design of a `TypedArrayBuilder`, which generates AwkwardForth programs from data types, but is still driven by `ArrayBuilder`-like commands.

3 AwkwardForth virtual machine

AwkwardForth is implemented in C++ as a virtual machine with byte-compiled instructions. It is not interactive, unlike most Forths, since it is intended to be programmed algorithmically. Even the `TypedArrayBuilder` use-case works by "wiring" its fixed suite of commands to algorithmically generated Forth subroutines. Some `TypedArrayBuilder` commands must change the state of its finite-state machine: for instance, when filling an array of doubly nested lists of integers like `[[1, 2], [3]], [], [[4], [5]]`, the first `begin_list` (`[`) puts it into a state that expects another `begin_list` (`[`) or `end_list` (`]`); the second puts it into a state that expects `integer` or `end_list` (`]`). And yet, each of these commands must return control-flow to its caller and remember its state for the next call. `TypedArrayBuilder`'s AwkwardForth machine must be able to stop and resume with its state intact (unlike most Forths). AwkwardForth, therefore, has built-in words to control its own execution (**`pause`** and **`halt`**), and the execution may be resumed from outside the machine.

When an AwkwardForth machine is first constructed, it compiles its source code (text) into bytecode instructions (variable-length sequences of 32-bit integers—a jagged array), so that they can be interpreted more quickly. This is the same sense in which Python code is "compiled." Built-in words translate to 1–3 integer codes, the second and third being arguments that modify the first. User-defined words are separate sequences of instructions, called a "dictionary" in Forth. Control-flow structures, such as **`if`** and **`loop`**, are implemented as unnamed user-defined words so that their bodies are fixed-width "calls" into the dictionary, simplifying the logic of instruction pointer manipulation.

All errors are caught in the compilation phase except for 10 possible runtime errors: "user halt," "recursion depth exceeded," "stack underflow," "stack overflow," "division by zero," "read beyond," "seek beyond," "skip beyond," "rewind beyond," and "varint too big." The last 5 of these are specific to parsing input buffers.

All runtime execution is implemented in a single "**`noexcept`**" C++ function for speed.

An arbitrary number of named input buffers and named output buffers are associated with each AwkwardForth program. Inputs must be supplied at the beginning of a run; outputs are also created at this time. Input buffers, which are the data to be parsed, are viewed as untyped raw bytes, interpreted by the words of the program itself. Output buffers, which are columns to use in a new Awkward Array, have specific numerical types and can only be filled with that type. Inputs have a fixed size and are seekable; outputs grow in an append-only way.

4 AwkwardForth language

Like all Forth languages, AwkwardForth has an extremely simple syntax: whitespace delimits tokens and tokens are interpreted in reverse Polish order: "`3 4 +`" means the sum of `3` and `4` (the first two words put numbers on the stack and the third pops them and pushes their sum).

AwkwardForth has 57 standard built-in words and 61 extensions for input and output buffers, though 50 of the 61 are different ways of interpreting the bytes of an input buffer. Two special words control the state of the virtual machine: **`pause`** and **`halt`**, and three do not generate runtime code at all: they declare variables, inputs, and outputs.

```
variable variable_name                 ( Standard Forth )
input    input_name                    ( AwkwardForth extension )
output   output_name output_type       ( AwkwardForth extension )
```

As in Standard Forth, user-defined words are bracketed between **`:`** and **`;`** and the main code is anything outside these definitions. Control structures are pairs of words like **`if`**-**`then`** and **`do`**-**`loop`**, or triples like **`if`**-**`else`**-**`then`** and **`begin`**-**`while`**-**`repeat`**.

```
: fibonacci    ( pops n -- pushes nth-fibonacci-number )
  dup
  1 > if
    1- dup 1- fibonacci
    swap fibonacci
    +
  then
;

( pushes [0 1 1 2 3 5 8 13 21 34 55 89 144 233 377] onto the stack )
15 0 do
  i fibonacci
loop
```

The 50 special words for parsing are all arrows with a type code, an optional **`!`** for big-endian and an optional **`#`** to pull a number off the stack to determine how many times to repeat it. Thus, "`input_name i-> output_name`" reads 4 bytes from the input as an integer and writes it to the output, "`input_name !d-> output_name`" reads 8 bytes as a big-endian, double-precision float, and "`100 input_name #I-> output_name`" reads 100 unsigned 4-byte integers. In any case, the destination may be the stack: "`input_name i-> stack`".

The type codes are taken from Python's `struct` module, which AwkwardForth most closely resembles in purpose (but vastly exceeds in expressiveness). Two special codes, **`varint->`** and **`zigzag->`**, read variable-length unsigned integers and zig-zag encoded signed integers, which are used in many file formats, including Avro and Parquet. Also, Parquet needs a command to read *n*-bit unsigned integers, hence **`2bit->`**, **`3bit->`**, etc. for any *n*.

Outputs are similar, but less varied because they have preassigned types. They may be filled directly from an input (bypassing the stack for efficiency and to preserve the types of floating point numbers) or filled from the stack: "`output_name <- stack`". A shortcut for appending the last output plus a value from the stack is "`output_name +<- stack`".

More built-in words can be added to handle problems posed by input formats. Since each built-in word has a fixed 5 ns cost, specialized words result in faster AwkwardForth code.

5 AwkwardForth programs for ROOT, Avro, and Parquet

To read a TBasket of `std::vector<std::vector<float>>` from ROOT, we use the following AwkwardForth program:

```
input data                      ( ROOT TBaskets have a data buffer )
input byte_offsets              ( and a buffer of byte offsets )

output offsets0 int32           ( output Awkward Array offsets )
output offsets1 int32           ( ... )
output offsets2 int32           ( ... )
output content float32          ( and content )

0 offsets0 <- stack             ( offsets start at zero )
0 offsets1 <- stack
0 offsets2 <- stack

begin
  byte_offsets i-> stack        ( get a position from the byte offsets )
  6 + data seek                 ( seek to it plus a 6-byte header )
  data !i-> stack               ( get the std::vector size )
  dup offsets0 +<- stack        ( add it to the offsets )
  0 do                          ( and use it as the loop counter )
    data !i-> stack             ( same for the inner std::vector )
    dup offsets1 +<- stack
    0 do
      data !i-> stack           ( and the innermost std::vector )
      dup offsets2 +<- stack
      data #!f-> content        ( finally, the floating point values )
    loop
   loop
again                           ( ends with a "seek beyond" exception )
```

To read data with the same structure from Avro, we use the program below. It is applied to each data block of an Avro container file, with the AwkwardForth machine stack initialized with the number of entries in the block.

```
input data                      ( an Avro data block is a single buffer )

output offsets0 int32           ( output Awkward Array offsets )
output offsets1 int32           ( ... )
output offsets2 int32           ( ... )
output content float32          ( and content )

0 offsets0 <- stack             ( offsets start at zero )
0 offsets1 <- stack
0 offsets2 <- stack

0 do                            ( upper limit on the stack at startup )
  data zigzag-> stack
  dup offsets0 +<- stack        ( add it to the offsets )
  0 do                          ( and use it as the loop counter )
    data zigzag-> stack
    dup offsets1 +<- stack
    0 do
      data zigzag-> stack
      dup offsets2 +<- stack
      data #f-> content         ( finally, the floating point values )
      data b-> stack drop       ( ends with a zero-byte )
    loop
```

```
      data b-> stack drop         ( lists also end with a zero-byte )
    loop
    data b-> stack drop           ( lists also end with a zero-byte )
  loop
```

Parquet is a columnar file format, so the floating-point content comes in a form that's ready to use in Awkward Array, without even copying the uncompressed buffer. However, the nested list structure is in a highly packed form: repetition levels indicating the depth of each floating-point item, which are further run-length encoded or bit-packed in small groups.

We unpack them using two AwkwardForth programs: the first produces the repetition levels as unsigned 1-byte integers and the second converts them into three levels of offsets.

```
input data
output replevels uint8          ( output 1-byte integers )

data I-> stack                  ( get the number of bytes to read )
begin
  data varint-> stack           ( read a variable-length integer )
  dup 1 and                     ( its lowest bit selects encoding type )

  0= if                         ( zero means run-length encoding... )
    data B-> replevels          ( write the value to duplicate )
    1 rshift 1-                 ( determine how many times )
    replevels dup               ( duplicate it )

  else                          ( non-zero means bit-packed... )
    1 rshift 8 *                ( determine how many to read )
    data #2bit-> replevels      ( read all the 2-bit integers )
  then

  dup data pos 4 -              ( continue until end of input buffer )
until
```

This last program interprets the repetition levels as offsets. It uses variables to count the number of items at each level of list depth since all three need to be increased concurrently and using `swap` or `rot` to manage them on the stack would be unnecessarily complex. The Standard Forth words `@`, `!`, and `+!` read, write, and increment an off-stack variable. Variables have the same integer type as the stack and are initially zero.

```
input replevels
output offsets0 int32 output offsets1 int32 output offsets2 int32
variable count0 variable count1 variable count2

begin
  replevels b-> stack           ( get one repetition level )

  dup 3 = if                    ( 3 means deepest level of structure )
    1 count2 +!
  then
  dup 2 = if                    ( 2 means a new innermost list )
    1 count1 +!
    count2 @ offsets2 +<- stack 1 count2 !
  then
  dup 1 = if                    ( 1 means a new inner list )
    1 count0 +!
```

```
      count1 @ offsets1 +<- stack 1 count1 !
      count2 @ offsets2 +<- stack 1 count2 !
    then
    0 = if                    ( 0 means a new outer list )
      count0 @ offsets0 +<- stack 1 count0 !
      count1 @ offsets1 +<- stack 1 count1 !
      count2 @ offsets2 +<- stack 1 count2 !
    then

    replevels end             ( continue to the end of input )
  until

  count0 @ offsets0 +<- stack ( add the last counts for all three )
  count1 @ offsets1 +<- stack
  count2 @ offsets2 +<- stack
```

Programs such as these would not be written (and commented!) by hand, but generated by Uproot, other file-readers that produce Awkward Arrays, or `TypedArrayBuilder`.

6 Example of an AwkwardForth program for TypedArrayBuilder

`TypedArrayBuilder` was implemented using AwkwardForth. It takes an input type and generates a machine that fills that type, pausing before each input command. Commands are represented by enumeration constants, and the program flow depends on the sequence of commands it receives. The example program below accepts a triply nested list of `float`.

```
input data                  ( a single value: argument to append )
output offsets0 int32       ( output Awkward Array offsets )
output offsets1 int32       ( ... )
output offsets2 int32       ( ... )
output content float32      ( and content )

0 offsets0 <- stack         ( offsets start at zero )
0 offsets1 <- stack
0 offsets2 <- stack

: node3
  {float32-command} = if
    0 data seek             ( calling code puts the next value at )
    data d-> content        ( the beginning of the input buffer )
  else
    halt                    ( only the "float32" command is allowed )
  then
;
{node2} {node1} {node0}     ( see below )

0 begin
  pause node0               ( pause for input, run forever )
again
```

The words in curly brackets are strings to be replaced, such as the following for `{nodeN}`:

```
: {node_name}
  {begin_list-command} <> if  ( "begin_list" is required here )
    halt
  then
```

```
    0 begin
      pause dup {end_list-command} = if
        drop
        {offsets_name} +<- stack      ( update offsets for this array node )
        exit                          ( exit this subroutine's infinite loop )
      else
        {next_node_name}              ( another list node like this or node3 )
        1+
      then
    again
  ;
```

The `{*-command}` substitutions are the enumeration constants. The virtual machine waits at a `pause` word until `TypedArrayBuilder` puts a command number on the stack (and a value in the input buffer for the `float32` command), then resumes program flow, letting AwkwardForth format the output or `halt` if the command is not allowed. Thus, `TypedArrayBuilder` itself can be statically compiled but “wired” to different actions at runtime.

7 Performance

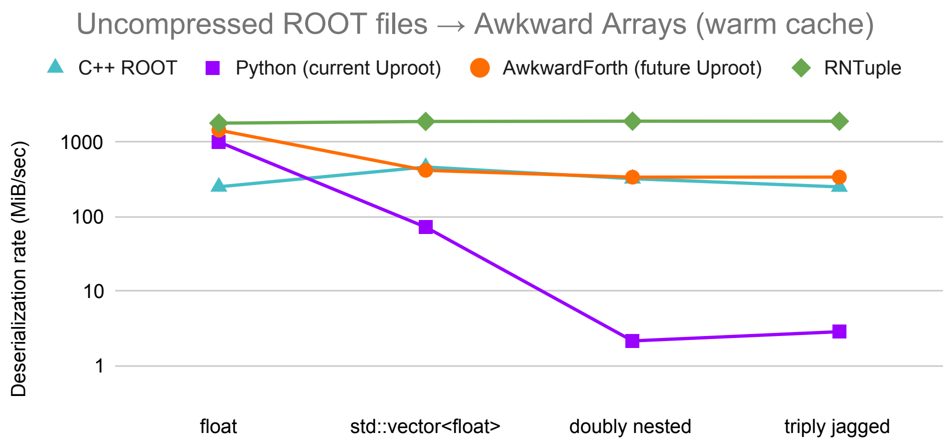
Figure 1 presents single-threaded deserialization rates of uncompressed ROOT, Avro, and Parquet files from a warmed filesystem cache on an Intel i7-8750H (2.2 GHz) processor. There are four test files for each format: 4-byte `float`, variable-length lists of `float`, doubly nested lists of lists of `float`, and triply nested lists of lists of lists of `float`. For ROOT files, variable-length lists are `std::vector`, but in Avro they are called “arrays” (despite being variable-length), and in Parquet, they are called “repeated groups.” All four types of files contain exactly 1 073 741 824 `float` values and are all approximately 4 GiB (large, but well within the 15.5 GiB of physical RAM). The lengths of the lists at each level of depth are Poisson-distributed with a mean of 8.0 items per list. ROOT’s TBasket size, Avro’s data block size, and Parquet’s row-group and page sizes were all fixed at 64 MiB.

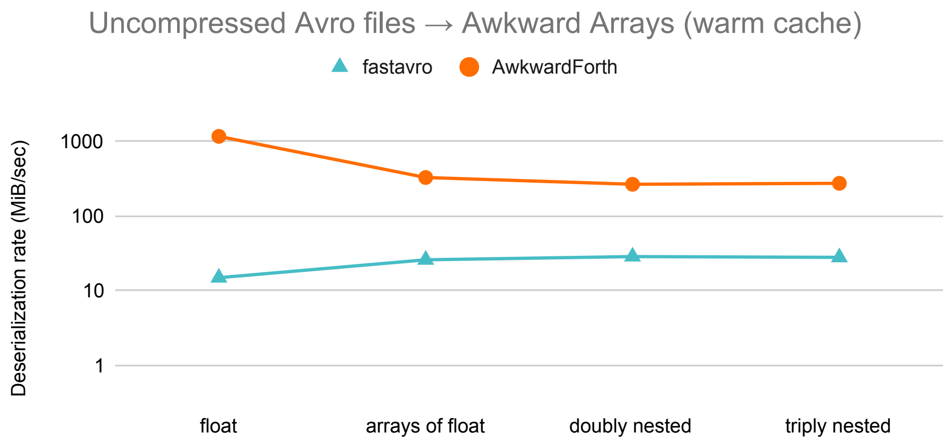
The ROOT files were read in three ways: (1) using C++ ROOT, loading data with `TBranch::GetEntry`) and copying it into `offsets0`, `offsets1`, `offsets2`, and `content` buffers for Awkward Array, (2) using Uproot 4.0.1’s Python code, and (3) using AwkwardForth (with Uproot to find the TBaskets within the file).

As an additional comparison, the same data were converted into ROOT’s future RNTuple format, which is truly columnar. As expected (reproducing our previous results [5]), `float` data are fastest to read, being essentially a memory-copy from TBaskets or RNTuple pages into the `content` buffer. (ROOT’s `TBranch::GetEntry` is not as fast as direct memory access, which could be enabled by switching to ROOT’s BulkIO feature, but we did not attempt that in this study.) RNTuple maintains this speed for all levels of nestedness for the same reason, while Uproot’s NumPy tricks slow down reading of `std::vector<float>` and Uproot’s pure Python is orders of magnitude slower for any deeper nesting. However, AwkwardForth keeps pace with ROOT’s `TBranch::GetEntry` at all levels of nestedness.

We next compared AwkwardForth with fastavro, the leading Python package for reading Avro files. fastavro is a Python extension library, using the C Avro implementation for speed. However, fastavro is compiled without knowledge of the schemas of the Avro files, which limits this advantage. AwkwardForth, on the other hand, has specialized Forth code for each data type and it’s fast enough to read Avro 10–80× faster than fastavro.

Finally, we compared AwkwardForth with pyarrow, a Python extension library for the C++ Arrow and Parquet projects. In this case, pyarrow outperforms AwkwardForth by factors of 1.5–8×. The AwkwardForth programs for Parquet in Section 5 show why: there’s nothing about them that specializes to the data type except for the `#2bit->` repetition level reader





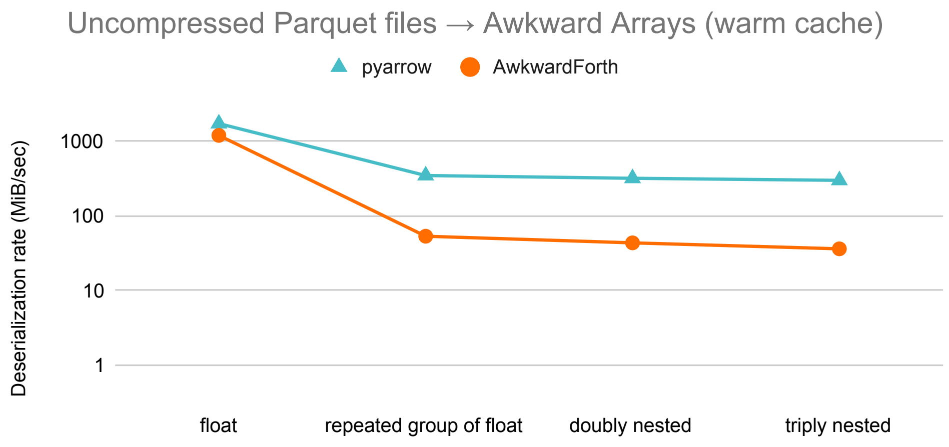


Figure 1. Deserialization rate for converting uncompressed ROOT, Avro, and Parquet files to Awkward Arrays from warm cache. C++ ROOT, fastavro, and pyarrow are the leading libraries for each file type.

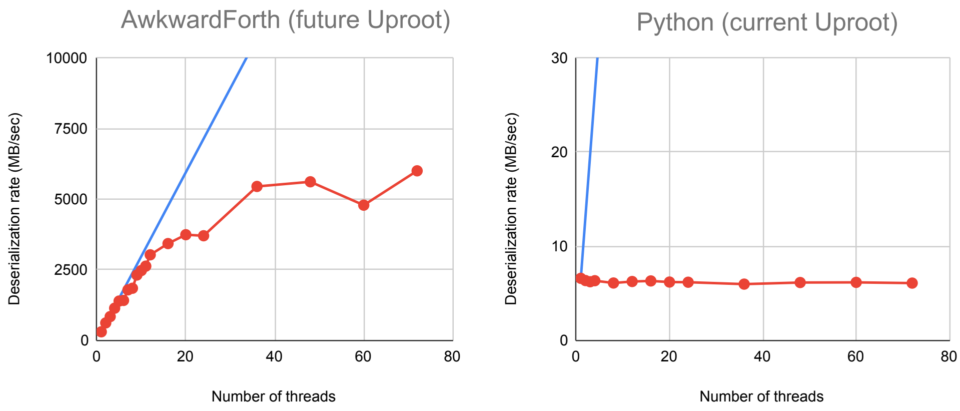


Figure 2. Scaling of deserialization by the number of threads. AwkwardForth (when called from Python) releases the Python GIL, which otherwise prevents any gains from parallel processing.

and the number of `count` variables and `offset` buffers. Any data type can be efficiently read with the same, unspecialized C++ code. This is true in general of columnar data formats, including RNTuple, so we have no plans of applying AwkwardForth to read columnar data.

For compressed data (not shown in Figure 1), all rates are suppressed by a constant for each algorithm. LZ4 is about 10% slower than the memory copy, but ZLIB is 10× slower than the memory copy, so AwkwardForth is not the bottleneck for ZLIB-compressed data.

We also studied the scaling of AwkwardForth with threads. The AwkwardForth virtual machine is stateful, but lightweight: many can be launched at once, one for each thread. Figure 2 shows deserialization rate of the ROOT files on an AWS `c5.18xlarge` instance, which has 72 cores. Linear scaling falls off at about 20 threads, or 4–5 GiB per second, which may be a limitation of memory bus supplying data from RAM. Python, by comparison, does not scale at all because of its Global Interpreter Lock (GIL) [18].

All scripts used to produce Figures 1 and 2 are available on GitHub [19].

8 Conclusions

AwkwardForth is a "lightweight" solution to the problem of generating fast deserialization code whose form is only known at runtime. In all, it consists of 3388 lines of C++ code in Awkward Array (version 1.1.0), with no dependencies. Unlike user-facing DSLs, its syntax is not subject to usability constraints, only ease of algorithmic generation and speed. As shown in Section 7, it keeps pace with ROOT's own TTree deserialization, exceeds fastavro for Avro, and is slower than but within an order of magnitude of pyarrow for Parquet.

Most importantly, AwkwardForth vastly improves upon the Python code in Uproot, speeding up `std::vector<std::vector<float>>` deserialization by factors of hundreds, and it makes parallel TBasket deserialization worthwhile. To integrate AwkwardForth into Uproot, every function that generates deserialization code in Python must be supplanted by a function to generate AwkwardForth. This can be a gradual transition, as missing Forth-generators can fall back to Python. We expect the majority of data types to be completed by the end of 2021.

9 Acknowledgements

This work was supported by the National Science Foundation under Cooperative Agreement OAC-1836650 (IRIS-HEP).

References

[1] J. Pivarski et al., *Uproot*, `https://doi.org/10.5281/zenodo.4543730`

[2] R. Brun, F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389**, 81 (1997)

[3] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., *Array programming with NumPy*, Nature **585**, 357 (2020)

[4] J. Pivarski et al., *Awkward Array*, `https://doi.org/10.5281/zenodo.4539721`

[5] J. Pivarski, P. Elmer, D. Lange, *Awkward Arrays in Python, C++, and Numba*, in *European Physical Journal Web of Conferences* (2020), Vol. 245 of *European Physical Journal Web of Conferences*, p. 05023, `2001.06307`

[6] V. Vasilev, P. Canal, A. Naumann, P. Russo, *Cling – The New Interactive Interpreter for ROOT 6*, Journal of Physics: Conference Series **396**, 052071 (2012)

[7] G. Reid, *Thinking in PostScript* (Addison-Wesley, 1990), ISBN 9780201523720, `https://books.google.com/books?id=ZeRXvgAACAAJ`

[8] Technical Committee X3J14, *ANSI Forth Standard* (1994), `https://www.taygeta.com/forth/dpans.html`

[9] E.D. Rather, D.R. Colburn, C.H. Moore, *The Evolution of Forth*, in *The Second ACM SIGPLAN Conference on History of Programming Languages* (Association for Computing Machinery, New York, NY, USA, 1993), HOPL-II, p. 177–199, ISBN 0897915704, `https://doi.org/10.1145/154766.155369`

[10] *Google ProtoBuf*, `https://developers.google.com/protocol-buffers/`

[11] *Apache Thrift*, `https://thrift.apache.org/`

[12] *Apache Avro*, `http://avro.apache.org/`

[13] *Google FlatBuffers*, `https://google.github.io/flatbuffers/`

[14] *JSON*, `https://www.json.org/`

[15] J. Blomer, P. Canal, A. Naumann, D. Piparo, *Evolution of the ROOT Tree I/O*, in *European Physical Journal Web of Conferences* (2020), Vol. 245 of *European Physical Journal Web of Conferences*, p. 02030, `2003.07669`

[16] *Apache Parquet*, `https://parquet.apache.org/`

[17] *Apache Arrow*, `https://arrow.apache.org/`

[18] D. Beazley, *Understanding the Python GIL*, in *PyCON Python Conference. Atlanta, Georgia* (2010)

[19] *Performance measurement code for this study*, `https://github.com/scikit-hep/awkward-1.0/tree/1.1.0/studies/awkward-forth-performance`