Acceleration of File De-Serialization with JIT Compilation

Alexander Muth

Dept. of Computer Science, TU Darmstadt
Laboratory for Parallel Programming

Abstract: This paper presents a Python code generator to improve the de-serialization process in the Extra-P project. Currently, the project relies on the marshmallow library, which can be slow and inefficient when processing large and complex data sets. To address this issue, the code generator uses the visitor pattern to traverse marshmallow objects and produce optimized Python code that maps input data directly to Python objects. By eliminating marshmallow's reflection-based de-serialization, the code generator significantly improves performance. The results confirm the effectiveness of the approach in enhancing the de-serialization process.

1 Introduction

In today's data-driven world, efficient data processing is essential for any application that deals with large data sets. In this context, de-serialization plays a crucial role in converting the data from its serialized form to an object that can be used by the application. However, de-serialization can be a performance bottleneck, especially when dealing with large and complex data structures.

1.1 Motivation

The motivation of this work is to improve the performance of de-serialization in the Extra-P project. The Extra-P project uses the marshmallow library for de-serialization. This is a popular Python library for converting complex data types, such as JSON or YAML, into Python objects. Because marshmallow is very slow, there is a need to explore alternative approaches to accelerate the de-serialization process in Extra-P.

1.2 The objective and scope of this work

The aim of this work is to develop a Python code generator that can speed up the file de-serialization in Extra-P by generating optimized Python code for each marshmallow object. This approach provides the opportunity to extend the code generator for custom data types or adapt the generated code of existing ones to specific needs. In addition, the generated code can be further analyzed to identify additional performance improvements. This work focuses on the development and evaluation of the code generator for Extra-P and it will not cover other potential use cases of the code generator.

2 Background

2.1 De-Serialization

Serialization is the process of converting objects into a format that can be stored. Deserialization is the reverse process of creating objects from the stored format. Both processes are critical in software development, as they enable objects to be shared and used between different applications or systems.

2.2 marshmallow

marshmallow is a popular library for object serialization and deserialization in Python. It uses a declarative schema to define how Python objects should be converted to and from JSON. However, the heavy use of reflection in marshmallow's implementation means that it effectively implements an interpreter on top of the Python interpreter. This leads to lower performance compared to other serialization libraries that generate code ahead of time.

2.3 Extra-P

Extra-P is a Python-based automated performance modeling tool, that is used to analyse scalability issues in High Performance Computing (HPC). It relies heavily on marshmallow to load and store the performance models. Therefore, Extra-P can be rather slow when dealing with large and complex data sets.

2.4 Requirements

To achieve the goal of speeding up the file de-serialization process in Extra-P, the following requirements were identified. They are sorted by priority:

1. **Compatibility with existing Extra-P project:** The solution must seamlessly integrate with the existing Extra-P project, which currently relies on marshmallow for de-serialization and is implemented in pure Python.
2. **Minimal changes to the existing codebase:** The solution should minimize the need for significant modifications to the existing Extra-P codebase. This approach ensures that the solution can be easily adopted without introducing new bugs or making the codebase more challenging to maintain.
3. **Improved performance over marshmallow:** The solution must be faster than the original marshmallow de-serialization process, as speeding up this process is the primary objective of this project.
4. **Flexibility for custom data types:** The solution should provide flexibility for users to adapt the code generation process to handle custom data types. This capability enables users to implement code generation for their specific use cases and data structures as needed.
5. **Ease of use and deployment:** The solution should be user-friendly, allowing for easy adoption and deployment. It should minimize the configuration requirements and provide straightforward integration with the existing Extra-P project.

3 Implementation

This chapter provides an overview of the implemented code generator, including the Application Programming Interface (API), code structure, and performance optimizations.

3.1 Compiler API

The code generator includes a class called *CompiledSchema* that serves as the main interface for utilizing the compiler. It offers simple methods for generating optimized executable code and executing it. Additionally, it provides an option to generate source code that can be independently executed, primarily for testing purposes.

To compile a schema, it needs to inherit from the *CompiledSchema* class. For schemas with nested schemas, only the top-level schema needs to inherit from *CompiledSchema*. The schema can be compiled using the *compile* method.

In addition to the standard deserialization (load, loads) and serialization (dump, dumps) methods provided by marshmallow schemas, the code generator introduces compiled counterparts denoted by the *_compiled* suffix. These compiled methods, such as *load_compiled*, allow data to be loaded and dumped using the optimized, compiled code. Prior to using the compiled methods, it is necessary to compile the schema.

A compilable schema and the process of compiling, loading and dumping data would look as follows:

```
class PluckableSchema(Schema):
    pluck_integer = Integer()

class CompilableSchema(CompiledSchema):
    constant = Constant(42)
    integer = Integer()
    plucked = Pluck(PluckableSchema, 'pluck_integer')

schema = CompilableSchema()
schema.compile(CompileFlags())
loaded = schema.load({'integer': 13, 'plucked': 42})
# loaded = {'constant': 42, 'integer': 13, 'plucked': {'pluck_integer': 42}}
dumped = schema.dump_compiled(loaded)
# dumped = {'constant': 42, 'integer': 13, 'plucked': 42}
```

The method *compile_to_string* can be used to obtain the generated source code as a string. This string includes all the necessary imports, local variables, and routines, allowing it to be executed independently of the *CompiledSchema* data. The generated source code can be easily debugged and profiled through independent execution.

3.2 Supported Features and Limitations

The implemented code generator focuses on accelerating the Extra-P project and, therefore, does not include all features of the marshmallow library.

The two main limitations are the absence of the *many* and *partial* parameters in the *load_compiled* and *loads_compiled* methods. Implementing the *many* parameter would require additional effort. However, the *partial* parameter can be supported with minor adjustments to the implementation.

Since not all marshmallow field types are necessary for the de-serialization in Extra-P, the code generator only includes the required field types for compilation. Certain field types, such as *AwareDateTime*, *Data*, *DateTime*, *Decimal*, *Dict*, *Email*, *Enum*, *Function*, *IP*, *IPInterface*, *IPv4*, *IPv4Interface*, *IPv6*, *IPv6Interface*, *Method*, *NaiveDateTime*, *Raw*, *Time*, *TimeDelta*, *Url*, and *UUID*, have not been implemented. The implemented field types include *Boolean*, *Constant*, *Field*, *Float*, *Integer*, *List*, *Mapping*, *Nested*, *Number*, *Pluck*, *String*, *Tuple*, as well as Extra-P's *ListOfPairs*, *ListToMapping*, *NumberField*, and *TupleKeyDict*.

The code generator supports the marshmallow *Schema*, as well as Extra-P schemas (*Schema*, *BaseSchema*) and schemas generated by the *make_value_schema* function.

Non-compilable fields are handled by the original marshmallow de-serialization methods. Therefore, non-compilable fields are still supported within a compiled schema, but they do not benefit from the performance improvement provided by the code generator. Further details will be provided in the next section.

The code generator fully supports marshmallow's pre- and post-processing functions, as well as recursive schemas. It performs both field-level and schema-level validation, including validations passed to the constructor. Excluding fields from the de-serialization process is also supported.

3.3 Code Generator

The code generator follows the visitor pattern and generates Python code for each individual field and schema in a marshmallow schema. These generated code fragments are then combined into a runnable Python script.

Each compilable field and schema is associated with an encoder, which generates the corresponding Python code for de-serializing the visited object. If a field or schema has child elements that require code generation, the visitor could be used to recursively visits them using the appropriate encoder and generates the corresponding code.

The encoder receives the visited object (field or schema) and the context of code generation. The context includes flags passed to the compiler and stacks that previous encoders can push data onto. The stacks can also be used to create scopes, ensuring that a variable is not accidentally overwritten in a generated code fragment that may be needed later.

Here's an example of the encoder for the Nested field:

```python
class NestedEncoder(FieldEncoder[Nested]):
    def _encode_deserialize(self, nested: Nested, context: CompileContext) -> EncodedReturn:
        with context.stacks.scope(DeserializeArgs(object=f'{context.stacks.object}.schema')):
            return visitor.deserialize(nested.schema, context)

    def _encode_serialize(self, nested: Nested, context: CompileContext) -> EncodedReturn:
        with context.stacks.scope(SerializeArgs(object=f'{context.stacks.object}.schema',
                                                 obj=context.stacks.value)):
            return visitor.serialize(nested.schema, context)

# Register the encoder in the visitor
visitor.register_encoder(NestedEncoder)
```

In this example, the *NestedEncoder* handles marshmallow's *Nested* field, used to nest schemas. To generate Python source code for a *Nested* field, the encoder generates code for the associated schema by visiting it using the visitor. Before visiting the schema, a new scope is created, and variables are pushed onto the stack. These variables are then popped from the stack when leaving the *with* block. The last line registers the encoder with the visitor.

By overriding the *field_ type* class method in an encoder, the type to be encoded can be customized. This can be seen in the example below. The *NumberFieldEncoder* overrides *field_ type* to specify the *NumberField* class as the type to be encoded, instead of the *Number* field class, which is encoded by the inherited *NumberEncoder*.

```python
class NumberFieldEncoder(NumberEncoder):
    @classmethod
    def field_type(cls) -> type:
        return NumberField
    ...
```

For fields, the visitor looks for the encoder that matches the exact type being visited. If no encoder is found for a field, a fallback encoder is used. The fallback encoder invokes the appropriate marshmallow de-serialization method, ensuring correct de-serialization even for fields without a specific encoder. The fallback encoder can be overridden using the *register_ field_ fallback_ encoder* method of the visitor.

Encoders typically use the *Template* class for generating code fragments. It provides functions for substituting keys in a string, similar to *string.Template*. However, the substitute functions updates the template and do not only return the processed string. The *substitute_ indented* method allows for the insertion of multiline strings with appropriate indentation into a template.

To support recursive schemas, the *SchemaEncoder* moves the generated code into a separate function and generates code with function calls to that function.

During code generation, an encoder can write variables to a dictionary representing local variables. This allows for evaluating for example default values during compilation, saving time during de-serialization.

3.4 Optimizations

To enhance both the compilation runtime and the efficiency of the generated source code, several optimizations were implemented based on the analysis of the program execution call graph using the profiling tool cProfile.

For example, in the *StringEncoder*, the use of *isinstance(value, str)* was replaced with *type(value) == str* to improve performance, as *isinstance* can be time-consuming. Additionally, the schemas are stored in local variables with unique keys to avoid accessing them through nested attributes when needed.

Significant runtime reduction was achieved by compiling the *create_object* methods and the *unpack_to_object* post-load routine of the Extra-P *Schema*. Instead of writing values into a dictionary during de-serialization and setting them as object attributes using a routine afterwards, the values are directly written to the appropriate location. As a result, the specific post-load routine is no longer executed, improving efficiency.

By utilizing the CompileFlags passed to the compiler, different features can be enabled or disabled based on specific use cases. For example, with the *validate* flag, types such as *Mapping* are checked, like it is done by marshmallow. It's important to note that modifying the flags can result in a different de-serialization process compared to marshmallow's default behavior. However, the de-serialization of the Extra-P *ExperimentSchema* is possible with any combination of flags, as long as the provided data is valid.

Through profiling and addressing major performance issues, the generated code has been optimized to achieve high efficiency during de-serialization.

4 Usage in Extra-P

To integrate the implemented compiler into the Extra-P project and accelerate the de-serialization process, the following steps can be followed:

1. The top-level schema used by Extra-P for loading experiments is the *ExperimentSchema* class, which currently inherits from *BaseSchema*. To enable compilation, *ExperimentSchema* can be modified to inherit from *CompiledSchema* in addition to its existing inheritance. This change allows the usage of the *compile* and *load_compiled*/*dump_compiled* methods for accelerated de-serialization.

2. Replace any instances where the marshmallow *load*/*dump* method of *ExperimentSchema* is used with *load_compiled*/*dump_compiled*. It's important to note that the instance of *ExperimentSchema* being used needs to be compiled once using the *compile* method. To optimize runtime performance, it's beneficial to keep a compiled instance throughout the program lifetime.

3. The encoder implementations provided for all necessary fields and schemas eliminate the need for additional modifications. You can find the required Python source code for these encoders in the *extrap.roastedmarshmallow* and *extrap.encoders* packages.

By following these steps, the Extra-P project can take advantage of the implemented compiler and achieve accelerated de-serialization.

5 Evaluation

To assess the effectiveness of the implemented code generator, three files were used for evaluation: *qs-cpu.extra-p*, *no_noise.extra-p*, and *with-noise.extra-p*.

To verify the correctness of the generated code, the same data was deserialized and serialized using both the marshmallow methods and the compiled methods. The results were compared for an exact match, and the implementation showed consistent agreement for all three files, both in the load and dump methods, confirming the correctness of the implementation

The evaluation was performed on an Ubuntu 22.04 (without desktop) system with Python 3.11.3 and marshmallow 3.19.0, utilizing an Intel Core i7-8750H processor.

To assess the runtime performance, serialization and deserialization were executed using marshmallow and the compiled counterparts, each performed 10 times. The average execution times in seconds were calculated to minimize the impact of external factors. The results for the average load time and speedup are presented in Table 1. The average dump time and speedup are presented in Table 2.

Table 1: Average runtime and speedup of *load* and *load_compiled*

file (size in MiB)	load (average in seconds)		speedup (average in times)
	marshmallow	compiled	
qs-cpu.extra-p (2,3)	2.096	0.292	7.178
no_noise.extra-p (20,6)	54.664	4.502	12.142
with-noise.extra-p (140,8)	274.562	58.566	4.688

Table 2: Average runtime and speedup of *dump* and *dump_compiled*

file (size in MiB)	dump (average in seconds)		speedup (average in times)
	marshmallow	compiled	
qs-cpu.extra-p (2,3)	1.121	0.270	4.152
no_noise.extra-p (20,6)	36.554	16.259	2.248
with-noise.extra-p (140,8)	115.197	38.350	3.004

The average execution time of the code generation itself, over 10 runs, was 0.228 seconds.

6 Conclusion and Outlook

In conclusion, the implemented code generator has successfully fulfilled the identified requirements and achieved the goal of significantly speeding up the file de-serialization process in the Extra-P Project. Through performance evaluation, the code generator has demonstrated remarkable improvements over the original marshmallow de-serialization process, resulting in substantial speed gains.

The compatibility of the code generator with the existing Extra-P project ensures seamless integration without the need for significant changes to the codebase. This compatibility allows Extra-P to leverage the benefits of the code generator without disrupting ongoing development and deployment.

Moreover, the code generator offers flexibility and extensibility, enabling easy adaptation to new use cases. It provides users with the ability to implement code generation for their own data types, allowing for customization and expanding the scope of applicability beyond the Extra-P project. The minimal configuration required for using the code generator enhances its accessibility and convenience, making it a valuable tool not only for Extra-P but also for any project utilizing the marshmallow library.

Moving forward, there are several potential ways for further development and improvement of the code generator. These include:

- Continuously monitoring and integrating updates from the marshmallow library to ensure compatibility and leverage new features or optimizations.
- Implement missing features of the marshmallow library like *many* and *partial* loading
- Conducting further performance optimizations to the code generation process itself by replacing the Templating mechanism which heavily relies on regex matching
- Further optimize the generated Python code to improve the performance of the compiled de-serialization even more
- Unexplored approaches such as multithreading could be utilized to further enhance the performance significantly.

By pursuing these options, the code generator can continue to evolve and serve as a valuable tool in the Extra-P project, enabling efficient and scalable file de-serialization while maintaining compatibility, ease of use, and the potential for ongoing optimization and improvement.