2018 Building Performance Modeling Conference and
SimBuild co-organized by ASHRAE and IBPSA-USA
Chicago, IL
September 26-28, 2018

ENERGYPLUS PERFORMANCE IMPROVEMENTS VIA JSON INPUT REFACTORING

Joshua New, Ph.D. and Mark B. Adams
Oak Ridge National Laboratory, Oak Ridge, TN

ABSTRACT

EnergyPlus is the flagship whole-building energy simulation program developed by the U.S. Department of Energy. This paper describes the refactoring efforts around the EnergyPlus input processor to natively support JavaScript Object Notation (JSON), allow validation through use of a JSON schema validator, convert and validate typical EnergyPlus input data file (*.idf) files to JSON files, and improve related metrics such as: read time for schema and input files, parse time for processing input, and time to query/return data from EnergyPlus internal data structure. Performance improvements on sample files result in 14% and 60% reduction in schema and input file reads respectively, 36-62% reduction to process input, and 24-99+% query reduction for functions that take advantage of the new format. For the files tested, overall speedup realized as part of this JSON refactor is 1.6x to 5.4x.

INTRODUCTION

Development of EnergyPlus began in 1995 to combine the most popular features and capabilities of Building Load Analysis and System Thermodynamics (BLAST) and DOE-2 to subsequently replace BLAST and DOE-2. EnergyPlus is currently the U.S. Department of Energy's (DOE) flagship whole-building energy simulation program (Crawley et al. 2001). Since that time, DOE has invested over $65 million in adding new building technologies and modern simulation capabilities. The primary users of EnergyPlus are architects and engineers assessing energy impacts in design, major software vendors offering simulation-based services for buildings, and agencies interested in code and policy impacts on building energy use. EnergyPlus is released under a commercial-friendly, open-source license on GitHub. Building Energy Modeling (BEM) has multiple use cases, both established and emerging:

- Design: architecture, HVAC system selection & sizing
- Operations: HVAC fault diagnosis, dynamic control, model predictive control, & demand response
- Market: code development & compliance, ratings, incentives, measurement & verification, policy, etc.

The input file used since 1995 was an input data file with the *.idf extension that explained the material properties, construction, building location, equipment, equipment schedules, and occupancy/internal load patterns for a building. This file was a structured text file with custom editors for allowing easier modification of the file. A conversion utility is provided with the simulation engine that allows the conversion from the *.idf's version number to the current version of EnergyPlus. Due to the step-by-step changes in subsequent EnergyPlus versions, this conversion utility necessarily converted subsequently version-by-version and could take one to three minutes to convert a reference building from v7.0 to v8.5. Using more modern techniques, such as schema enforcement and validation provided by JSON and JSON Schema, this conversion process could be orders of magnitude faster.

REFACTORING ENERGYPLUS

Javascript Object Notation

JSON (pronounced JAY-suhn) is an open-standard format that uses human-readable text to transmit data objects using key/value pairs (Crockford 2006). It is a language-independent format that is the most widely-used data format for asynchronous browser/server communication. JSON files typically use the extension *.json and are largely replacing eXtensible Markup Language (XML). There are existing tools for reading and manipulating JSON files in nearly all programming languages.

A literature review and comparative survey was conducted to cover modern JSON libraries, JSON Schema libraries, and YAML Ain't Markup Language (YAML). Three C++ JSON libraries were compared for characteristics important to the EnergyPlus development team including considerations such as performance, syntax, cross-platform, license, and flexibility as a header-only library. The first library is RapidJSON (Yip 2015), which is the fastest and most memory efficient C++ JSON library (Yip 2016). This library is cross-platform but is not header-only, meaning it needs to be compiled into a library. The license is a modified MIT license, which is a

```
BuildingSurface:Detailed,
  Building_Roof,            !- Name
  Roof,                     !- Surface Type
  IEAD Non-res Roof,        !- Construction Name
  TopFloor_Plenum,          !- Zone Name
  Outdoors,                 !- Outside Boundary Condition
  ,                         !- Outside Boundary Condition Object
  SunExposed,               !- Sun Exposure
  WindExposed,              !- Wind Exposure
  AutoCalculate,            !- View Factor to Ground
  4,                        !- Number of Vertices
  49.9110,0.0000,11.8872,   !- X,Y,Z ==> Vertex 1 {m}
  49.9110,33.2738,11.8872,  !- X,Y,Z ==> Vertex 2 {m}
  0.0000,33.2738,11.8872,   !- X,Y,Z ==> Vertex 3 {m}
  0.0000,0.0000,11.8872;    !- X,Y,Z ==> Vertex 4 {m}
```

Figure 1: Example legacy IDF format defines an object (top), followed by properties of that object in field names (including a unique name for later reference), and comments for those fields to the right.

```
{
    "BuildingSurface:Detailed": {
        "Building_Roof": {
            "construction_name": "IEAD Non-res Roof",
            "extensions": [
                {
                    "vertex_x_coordinate": 49.911,
                    "vertex_y_coordinate": 0.0,
                    "vertex_z_coordinate": 11.8872
                },
                {
                    "vertex_x_coordinate": 49.911,
                    "vertex_y_coordinate": 33.2738,
                    "vertex_z_coordinate": 11.8872
                },
                {
                    "vertex_x_coordinate": 0.0,
                    "vertex_y_coordinate": 33.2738,
                    "vertex_z_coordinate": 11.8872
                },
                {
                    "vertex_x_coordinate": 0.0,
                    "vertex_y_coordinate": 0.0,
                    "vertex_z_coordinate": 11.8872
                }
            ],
            "number_of_vertices": 4,
            "outside_boundary_condition": "Outdoors",
            "outside_boundary_condition_object": "",
            "sun_exposure": "SunExposed",
            "surface_type": "Roof",
            "view_factor_to_ground": "Autocalculate",
            "wind_exposure": "WindExposed",
            "zone_name": "TopFloor_Plenum"
        },
    }
}
```

Figure 2: Fast JDF format for rapid parsing, query, and validation. EnergyPlus objects are root keys followed by name keys and finally field name keys. The values are the same as in IDF.

permissive license compatible with EnergyPlus' license. The major negative of the RapidJSON library is an unintuitive syntax that is difficult to use and understand. The next JSON library surveyed was JSON for Modern C++ library, which is open-source and released under the permissive MIT license (Lohmann 2016). This library was designed to mimic Python's JSON library syntax, making the API easy to use and implement. Since Python is currently a developer dependency in EnergyPlus, this library's syntax should be familiar for many of the developers and require no additional packaging of dependencies. The library is header-only, allowing flexible implementation, and is cross-platform. Despite these advantages, this library neither has as fast a runtime nor the memory efficiency of the RapidJSON library. Another challenge is that this library requires recent releases of the major compilers for full support. Officially, JSON for Modern C++ supports GCC 4.9+, however, EnergyPlus currently supports GCC 4.8.4+. Although not officially supported, JSON for Modern C++ does compile on GCC 4.8.4. Finally, the last JSON library surveyed was JsonCpp (Lepilleur 2016). This library uses a Python script to generate an amalgamated source and header files which must then be compiled into a library. JsonCpp is the slowest and least memory efficient of the three JSON libraries studied. The syntax of JsonCpp is not as intuitive and easy to use as JSON for Modern C++ but is better than RapidJSON. After weighing the pros and cons of each, it was decided to use the JSON for Modern C++ library. The primary advantages of this library are its quick runtime performance, easy to use syntax, cross-platform capabilities, and flexibility as a header-only library.

YAML is a human friendly data serialization standard for all programming languages (Ben-Kiki 2005). It is a strict superset of JSON, meaning any valid JSON file can be parsed by YAML. However, YAML departs from JSON by using whitespace indention to denote structure and does not require quotation marks, brackets, braces, and open/close tags. It is also well suited to hierarchical data as wel as relational data. Unfortunately, there is only one C++ YAML library that adheres to the latest specification, yaml-cpp (Beder). This library has a dependency on Boost and is slower to parse a JSON file compared to the three JSON libraries studied. Another concern is the cost for maintenance and support of both JSON and YAML input file formats. By supporting both formats, all developers and user support personnel need to be fluent in both formats. This can lead to subtle bugs and differences since YAML has more features than JSON. For these reasons, it was decided to not pursue YAML as an input format for EnergyPlus.

A schema allows one to define the structure of a document that must be adhered to; in this case, the description of the building, material properties, construction, and equipment schedules in an EnergyPlus input file must follow certain rules. The creation of a schema also allows automated tools to validate a specific file and provide descriptive errors if the format, data ranges, or other properties violate the schema. A survey of C++ JSON Schema libraries found several limitations including poor error reporting (Galiegue 2013). There were only two C++ JSON Schema libraries with a license compatible with EnergyPlus - RapidJSON and Valijson. RapidJSON was the fastest schema validation library. Valijson took 2.2x longer to validate the Medium Office building (Table 1) and has a dependency on Boost, making this library not

feasible for EnergyPlus (Boost 2012). Neither library had acceptable error reporting, so it was decided that creating our own validation would allow use of callbacks during parsing to validate inputs as they are parsed in real-time. This allows for more localized error reporting, including line numbers, for more specific details such as specific violations of necessary physical relationships between objects within a building model.

Two JSON formats were investigated to identify a similar format to the legacy IDF format, but that also validates quickly using JSON Schema libraries in various programming languages. One format parses, queries, and validates quickly (Figure 2) while the other is very similar (Figure 3) to the legacy IDF format (Figure 1). These were tested on two building types - the standard Medium Office building (871 surfaces in 118 zones) and a "prj10" building with an extreme number of surfaces (45,382 in 80 zones). The time required to validate is shown in Table 1, highlighting the need to use a format that is well suited to JSON Schema and validates quickly.

Figure 3: Legacy-like JDF format in similar to legacy IDF.

EnergyPlus has an input data dictionary (IDD) which was translated into a epJSON schema. JSON Schema is used for the processing of the epJSON schema to enable users and 3rd-party developers to validate their *.epJSON file against the *.schema.epJSON using any JSON Schema validator in their programming language of choice. While this will catch any structural, data type, naming convention, or value-range issues, it will not capture relationships (e.g. a coil is part of a unitary system) that will be captured during validation when the *.epJSON is read by EnergyPlus.

Refactor Input Processor

As a necessary modification to changing the EnergyPlus input file type, a modified parser is required to translate the new input file to the building description stored in the computer's data structure as required to successfully run a simulation. EnergyPlus' input processor was restructured and optimized to increase EnergyPlus performance in this parsing task.

When EnergyPlus runs, there is a portion called the InputProcessor. Due to the modular nature of the EnergyPlus simulation engine, each module is responsible for getting the input data it needs to execute. This function is performed by the InputProcessor. In order to minimize the number and extent of modifications for the new input file format, this work uses the same InputProcessor function signatures as previous versions of EnergyPlus. Thus, the InputProcessor was modified only to read, parse, and query/return JSON data, but the data is provided to each of the EnergyPlus modules in the same format as previous versions. Within InputProcessor, a function known as GetObjectItem is responsible for searching EnergyPlus' internal data structure to return data relevant to a given object. Another function in InputProcessor, known as VerifyName, is responsible for verifying that each EnergyPlus object type or group of types have unique names. This function was rewritten to accommodate the additional structure provided by JSON's key/value pair store. The input file conversion was significantly assisted by the recent EnergyPlus development team's emphasis on unit testing, which allowed complete coverage testing of all modules. This provided necessary confidence that changes due to this refactoring did not have unintended consequences.

Performance Metrics

In the traditional *.idf, relationships between data fields are defined by position, shown in Figure 1. In the new *.epJSON, field level data within an EnergyPlus object is now key/value based as described in Figure 2. Key/value storage has several advantages: (1) version translation is easier, (2) allows for easier use of defaults as they don't have to be defined in the input file, and (3) is easier to query. There are tools for reading and parsing JSON files in almost all programming languages, resulting in a decreased maintenance cost for complex, custom parsing programs for reading and/or translating an EnergyPlus input file. Users can also add custom markup to the input file for use by 3rd-party or custom tools; EnergyPlus will ignore any data fields not defined by EnergyPlus' data dictionary.

Changing the order of objects in an EnergyPlus file should not break a simulation (i.e. an input file should not be

parse-order dependent), but this was never fully checked in previous EnergyPlus versions. Since the JSON library uses a C++ standard map to store the key/value pairs, the keys are sorted lexicographically. This caused several unit tests and integration tests to fail during development.

The theoretical complexity of the GetObjectItem and VerifyName were reduced from $O(n^2)$ to O(n) by using a C++ standard unordered map, which has amortized O(1) lookup. This can have a dramatic effect on runtime when parsing a building with many surfaces or other objects. Calls to VerifyName were eliminated since the function only checked that names were unique within the same EnergyPlus object type, and this capability is subsumed by the JSON standard which enforces unique keys. For the average EnergyPlus input file, using the OutPatient Reference Building as a surrogate, there was a marginal 1.6x performance gain due to the overhead of unordered map (Table 2). In the larger prj10 test file, which has an extreme number of surfaces (45,382 in 80 zones vs 871 in 118 zones for the Outpatient), shows larger performance gains of 5.4x. The large number of surfaces shows the advantage of algorithmic complexity changes.

FUTURE WORK

Unique strings/identifiers for each object could be used to provide a standard language for recognizing certain types of objects within or across different models. Also, advancing the refactor past the InputProcessor into the functions that it touches has the potential to speed-up simulation run-time.

Violating the assumption of order-dependence gave rise to unit tests and integration test failures that were fixed; however, some of the calculations during an EnergyPlus simulation may also be parse-order dependent. More work is needed to ensure that these expectedly-minor calculation differences are within acceptable tolerance.

As EnergyPlus continues to be refactored to be more object-oriented, individual objects can interact more directly with the JSON data structure instead of manipulating data in the old format. This will have a performance benefit not yet realized in the current work.

CONCLUSION

A major refactor of EnergyPlus' InputProcessor to use a new JSON input format yields an overall speedup of 1.6x to 5.4x. This large speedup comes from faster input parsing, reduced algorithmic complexity in InputProcessor functions, and greatly reduced need to verify unique names due to JSON standards compliance. This results in performance improvements on sample files result in 14% and 60% reduction in schema and input file reads respectively, 36-62% reduction to process input, and 24-99+% query reduction for functions that take advantage of the new format. In addition by using JSON and JSON Schema, users can use a large number of language agnostic tools to create, manipulate, and validate epJSON input files. By using a modern, standardized input format, EnergyPlus is well positioned for the next 20 years of development.

ACKNOWLEDGMENT

REFERENCES

Beder, J. "yaml-cpp, A YAML parser and emitter for C++,." *URL: https://github.com/jbeder/yaml-cpp*.

Ben-Kiki, Oren, et al. 2005. "Yaml ain't markup language (yaml) version 1.1." *URL: http://www.yaml.org/*.

Boost, C++. 2012. "Libraries." *URL: http://www.boost.org/*.

Crawley, Drury B, Linda K Lawrie, Frederick C Winkelmann, Walter F Buhl, Y Joe Huang, Curtis O Pedersen, Richard K Strand, Richard J Liesen, Daniel E Fisher, Michael J Witte, et al. 2001. "EnergyPlus: creating a new-generation building energy simulation program." *Energy and buildings* 33 (4): 319–331.

Crockford, Douglas. 2006. "The application/json media type for javascript object notation (json)."

Galiegue, et al. 2013. "Json schema: Core definitions and terminology." *Internet Engineering Task Force (IETF)*, p. 32.

Lepilleur, Baptiste. 2016. "JsonCpp." *URL: https://github.com/open-source-parsers/jsoncpp*.

Lohmann, Niels. 2016. "JSON for Modern C++." *URL: https://github.com/nlohmann/json*.

Yip, Milo. 2015. "RapidJSONa fast JSON parser/generator for C++ with both SAX/DOM style API." *THL A29.[Online]. Available: https://github.com/miloyip/rapidjson*.

Yip, Milo. 2016. "C/C++ JSON parser/generator benchmark." *Milo Yip.[Online]. Available: https://github.com/miloyip/nativejson-benchmark*.

NOMENCLATURE

BEM	Building Energy Modeling
BTO	Building Technologies Office
IDD	Input Data Dictionary
IDF	Input Data File
epJSON	JSON Data File
epJSON schema	JSON Data Dictionary
JSON	JavaScript Object Notation
XML	eXtensible Markup Language

Table 1: Comparison of the time between two different JSON formats and the time to validate their schemas shows a significant 20-40x speedup in input validation time for the Medium Office building and large, 45k-surface prj10 building.

	Validate Medium Office (milliseconds)	**Validate prj10 building (milliseconds)**		
	Python	Python	RapidJSON	Valijson / JsonCpp
Legacy-like Format	9.1	25.0	102.2	223.5
Fast Format	0.5 (20x)	6.7 (37x)	2.6 (40x)	9.6 (23x)

Table 2: Performance improvement realized from the refactor includes a 1.6x to 5.4x overall speedup for the buildings tested.

	Time to process Outpatient input file (milliseconds)			**Time to process prj10 input file (milliseconds)**		
Function or task	8.5 Release	Refactor - IDF input	Refactor - JSON input	8.5 Release	Refactor - IDF input	Refactor - JSON input
ProcessInput	366	300 (18%)	234 (36%)	4322	3,355 (22%)	1,637 (62%)
GetSurfaceData	28	13 (54%)	13 (54%)	72688	21,001 (71%)	21,001 (71%)
GetObjectItem	38	29 (24%)	29 (24%)	41617	333 (99.2%)	333 (99.2%)
VerifyName	2	0 (100%)	0 (100%)	11055	5 (99.9%)	5 (99.9%)
Parse input	174	135 (22%)	69 (60%)	4130	3,190 (23%)	1,472 (64%)
Parse schema	192	165 (14%)	165 (14%)	192	165 (14%)	165 (14%)