FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations

Authors removed for double-blind submission

ABSTRACT

JSON is one of the most popular data encoding formats, with wide adoption in Databases and BigData frameworks, and native support in popular programming languages such as JavaScript/Node.js, Python, and R.

Nevertheless, JSON data manipulation can easily become a performance bottleneck in modern language runtimes due to parsing and object materialization overheads. In this paper, we introduce FAD.JS, a runtime system for fast manipulation of JSON objects in data-intensive applications. FAD.JS is based on speculative just-in-time compilation and on direct access to raw data. Experiments show that applications using FAD.JS can achieve speedups up to 2.7x for encoding and 9.9x for decoding JSON data when compared to state-of-the art JSON manipulation libraries.

1. INTRODUCTION

The JavaScript Object Notation (JSON [5]) format is arguably one of the most popular data encoding formats, and has become a de-facto standard in several domains. Data-intensive systems and applications heavily rely on JSON. Notable examples are key-value databases (e.g., [9]) and BigData analytics frameworks (e.g., [25, 2]), where it is often used as the main encoding format to represent semi-structured data. Other relevant examples can be found in domains such as Cloud and Web technologies, where JSON is used as the main encoding format in client/server communications as well as in so-called microservices applications [18, 1].

In most of the scenarios where JSON data is employed, it is used at the boundary between a data source (e.g., a Database, a file system, or a memory-mapped TCP buffer) and a language runtime (e.g., a JavaScript/Node.js virtual machine). The interaction between the language runtime and the external data source can easily become a performance bottleneck for applications that need to produce or consume significant amounts of JSON objects. Such performance overhead is caused by two core characteristics of existing JSON parsing runtimes. First, the JSON object resides in a data source that is external to the memory space of the language runtime. As a consequence, the language runtime needs to *materialize* the data in its language-private heap memory space (using a primitive data type, e.g., a JavaScript string) before consuming it. Similarly but specularly, a language runtime producing a JSON-encoded string needs to allocate the string in its private memory space before externalizing it. A second source of performance overhead is that all the JSON encoding and decoding libraries in modern language runtimes rely on *general-purpose* techniques that do not take into account the structure of the data that they are manipulating. Decoding is often based on a LL parser [10], while encoding is implemented by performing a full walk of the object graph that is being converted to JSON. The adoption of such general-purpose libraries is mostly motivated by the fact that JSON is used in the context of dynamic languages such as JavaScript or Python, where it is not possible to know in advance (i.e., statically) the characteristics of the JSON data that will be processed by the application. In other words, such applications do not use a pre-defined *schema* (e.g., based on JSON schema [20]) that could be used to speed up data access. Interestingly, the lack of a pre-defined schema in many JSON-intensive applications does not necessarily imply that *some* form of structure could emerge in the way JSON data is created or accessed at runtime. As an example, many interactions with public Web APIs have some informally-defined structure. Similarly, when JSON data is produced after an interaction with a database, it often matches the structure of the database tables where data is stored. We argue that very often JSON-intensive applications present an *hidden schema* that is known only at runtime, and we belive that all such applications deserve specific optimizations.

In this paper, we introduce a new runtime system, called FAD.JS, that can significantly improve the performance of JSON operations in data-intensive applications. FAD.JS differs from existing encoding and decoding approaches for dynamically typed languages such as JavaScript in two runtime design aspects: (1) it can perform encoding and decoding operations on raw data, without materializing objects in the language memory space until they are used, and (2) rather than being based on general-purpose techniques, it is based on the notion of *specialization*, and relies on just-in-time compilation to optimize encoding and decoding operations for the specific characteristics of the JSON data being processed. Thanks to its design, FAD.JS performs extremely well in all cases where JSON operations have stable usage

Proceedings of the VLDB Endowment, Vol. 10, No. 7

patterns, outperforming general-purpose JSON libraries in all the considered benchmarks. This paper makes the following contributions:

1) We describe the design and the implementation of Fad.js, a runtime system for fast access to JSON data in dynamic languages. To the best of our knowledge, Fad.js is the first example of JIT compilation applied to JSON data access in a dynamic language runtime. We base our implementation on Graal.js [7], a state-of-the-art implementation of Node.js [6]. The Fad.js runtime techniques can be considered language-independent, and can be applied to other dynamic languages as well.
2) We describe a new JIT-based encoding technique, which we call *Constant structure encoding.* The technique can speed up the encoding of JSON data up to 2.7x on the considered benchmarks.
3) We describe a new JIT-based decoding technique, which we call *Direct structure decoding.* The technique can speed up the decoding of JSON data up to 9.9x in the considered benchmarks.

2. A MOTIVATING EXAMPLE

JSON is extensively used in data-intensive applications in combination with dynamic languages. Most of the times, the structure of the JSON objects being accessed by the language runtime is unknown until execution time, and common JSON encoding libraries do not make any a priori assumption on the structure of the data they access. Rather, they rely on well-known parsing and encoding techniques that are known to offer good performance for common usage patterns. Very often, however, JSON data manipulation could benefit from some *speculative* runtime assumptions. For example, a JSON encoder could speculate on some property names being constant: as long as the objects have the same set of properties (with constant types), the encoding of a JSON string could potentially be performed more efficiently. Consider the following example:

```
exports.handler = function(event, callback) {
  var result = {
           key: event.key,
           data: someFunctionOf(event)
         };
  // encode and forward to a data storage
      service
  callback.success(JSON.stringify(result));
}
```

The code snippet corresponds to an AWS Lambda function [1] consuming data from an external Web source (e.g., an HTTP connection). The code in the example produces a `result` object using the JSON `stringify` JavaScript built-in function. This function generates a JSON-formatted string by performing a full walk of the objects and values in the `result` object graph, reading all the names of the properties in the object, and traversing the graph (including the `data` object) while encoding the new JSON-formatted string. Intuitively, most operations could be avoided for this specific example: since the `result` object has a constant structure (i.e., it always has two properties named `key` and `data` of the same type), reading the names of its properties could be avoided (they are constant); similarly, traversing the full object graph could be avoided, too. The encoded JSON string could be created starting from some constant tokens (i.e., the pre-formatted property names), concatenated with the values of the `result` properties. Moreover, since the `result` object has a constant structure, reading the values from its properties (i.e., `key` and `data`) could be optimized, too. Rather than implementing the encoding operation using a general-purpose JSON encoder, this example suggests that the language runtime could specialize on the data being encoded, and benefit from some form of runtime knowledge.

Even more effective optimizations could be performed in the case of JSON data decoding. Consider the following example of an Apache Storm [2] stream processing component written in Node.js [1]:

```
Bolt.process = function(tuple, done) {
  var tweet = JSON.parse(tuple.value);
  if (tweet.user === "@userfoo") {
    // send to the next pipeline stage
    this.emit({
            value: tweet.body,
            anchorTupleId: tuple.id
          });
  }
  done();
}
```

The Storm component in this example selects a sequence of JSON-encoded tweets with a given username (`@userfoo`, in the example). Using the default JSON decoder of Node.js (i.e., `JSON.parse`), even the small code snippet in the example could result in significant overhead. For each tweet, the application allocates a UTF-8 JavaScript string in the Node.js' process heap space (from the raw binary data received from the socket), parses it (into `tuple.value`), materializes an object graph (the `tweet` object) in the Node.js heap space, accesses its `user` property, and – only if needed – reads a second property (i.e., `body`, which is potentially big). Intuitively, most of the operations could be avoided: each tweet that is received has several properties, but only two of them are actually accessed by the application. An ideal solution would avoid allocating a JavaScript string and an entire JavaScript object instance, and would rather read the content of the `value` property *directly* from the raw input data, materializing the `body` property only when (and if) it is read by the application. By materializing only what is really used, the performance of the application could be significantly improved. Nevertheless, performing such *partial* and *selective* materialization of the data into the Node.js' process heap would require non-trivial runtime information from the language execution runtime.

The Fad.js runtime is designed exactly for the encoding and decoding scenarios that we have described, which we informally call JSON access operations with an *hidden* JSON schema. Such operations share properties that can be commonly found in data-intensive scenarios where dynamic languages such as Node.js/JavaScript are used:

1) Objects often have properties with constant name and type (for example, objects have a property corresponding to a unique identifier or key.)
2) The JSON (encoding or decoding) operations are performed on more than a single JSON object or string. This is often the case for message-based applications or for data-intensive ones, where high volumes of JSON data are created or consumed (e.g., a file with one object per line).

---

[1] Example of a *Bolt* component extracted from the Apache Storm multi-language bindings for Node.js: `http://storm.apache.org/`

3) The JSON data are read only partially, and not all of the values are used by the application logic. Nevertheless, their usage presents a stable access pattern, that is, the application is very often accessing a similar subset of properties of the object graph.
4) The application manipulating JSON data always interacts with an external I/O data source (e.g., a Database, a TCP connection, or a file). The data are received in a binary format, which the language runtime has to convert to its native types (i.e., heap-allocated strings).

Typically, JSON data does not come with a schema. This is particularly true in the context of dynamic languages. Hence, it is not possible to make any static assumption on the structure or types of such data. In principle, any object graph could at runtime change all or a subset of its properties to have a different name or type (e.g., all JSON objects representing a tweet might have an `author` property, but only those corresponding to tweets generated using a mobile device might have a `gps_position` property, which could be accessed very rarely). As a consequence, the notion of hidden schema is not to be considered strict, as it cannot be formalized for the purposes of static or semi-static analysis. Conversely, we consider the hidden schema of a JSON-intensive application a pure runtime-only information that *could* emerge after observing JSON usage patterns. In other words, we consider it a runtime speculative assumption.

3. FAD.JS

FAD.JS is a JSON encoding and decoding runtime targeting the data-intensive workloads described in the previous section. Informally, FAD.JS attempts to identify a JSON hidden schema at runtime, and relies on its properties to access JSON data more efficiently. The FAD.JS runtime techniques are language-agnostic, and could potentially be applied to any managed language. In this paper, we focus on JavaScript and Node.js: in this context, FAD.JS can be considered a drop-in replacement for the built-in JSON libraries of JavaScript's core standard library. In addition to being fully compatible with the default Node.js' JSON library, FAD.JS features an additional API (detailed in section 3.3) that can be used to further improve the performance of JSON parsing under certain circumstances.

FAD.JS relies on runtime assumptions and the dynamic generation of efficient machine code that leverages such assumptions: as long as they hold, encoding and decoding operations can be performed more efficiently. FAD.JS is built on top of Oracle's Graal.js and Truffle technologies, which we describe in the following section.

3.1 Background: Truffle and Graal.js

Truffle [24] is a framework for the development of runtime systems that can be used to implement language execution engines (e.g., a JavaScript virtual machine) as well as JIT-enabled runtime libraries such as FAD.JS. A Truffle-based runtime is implemented in the form of a self-optimizing Abstract Syntax Tree (AST) interpreter [10]: each node in the AST corresponds to a single runtime operation (e.g., reading some bytes, performing a function call, etc.) which can be compiled to highly-optimized machine code by means of partial evaluation [15] by the Graal [21] dynamic compiler. At runtime, each AST node eagerly replaces itself with a specialized version that relies on some (runtime-only) speculative assumptions, leading to better performance. For example, node rewriting specializes the AST for the actual types used by an operation (e.g., short integers rather than double-prcecision numbers), and can result in the elision of unnecessary generality, e.g., boxing and complex dynamic dispatch mechanisms. As long as an assumption holds, the compiled machine code will benefit from it (e.g., by treating some object properties as short integers). Conversely, as soon as a runtime assumption is invalidated, the machine code and the corresponding AST node are *de-optimized* and replaced with new, more generic, versions that do not rely on the assumption anymore. Node rewriting and JIT compilation are handled automatically by the Graal [21] dynamic compiler, which transparently compiles AST nodes to machine code when needed, and replaces invalidated machine code with less-optimized one in case of speculation failures.

The FAD.JS runtime described in this paper has been designed to target the Oracle Graal.js JavaScript language runtime [7]. Graal.js is a high-performance JavaScript runtime running on the JVM; it is fully compatible with Node.js, and is developed using Truffle. Graal.js is a highly compliant implementation of JavaScript: as of today, it passes more than 99% of the ECMA language compliance tests, and is able to fully support Node.js workloads, with performance in line with state-of-the-art JavaScript runtimes such as Google V8 [6]. Since both FAD.JS and Graal.js are based on Truffle, their AST nodes are compatible, and can be freely combined. For example, the node implementing a JavaScript property lookup operation can be executed during a FAD.JS encoding operation. In this way, the machine code produced for the FAD.JS operation accessing JavaScript native objects (e.g., to read a property) will be compiled with the very same machine code of the JavaScript operation. This effectively means that *core* operations of the JavaScript runtime such as reading or writing properties are *directly* inlined in the FAD.JS runtime without any additional overhead.

3.2 Runtime Speculation in FAD.js

FAD.JS achieves high performance by means of two techniques that are based on speculative assumptions, JIT compilation, and direct access to raw data:

- *Constant structure encoding*: FAD.JS attempts to identify an object graph (or a subset of it) with constant structure, property names and types. When found, FAD.JS generates machine code that is specialized for such graph structure, and that can encode objects with higer efficiency.
- *Direct structure decoding*: FAD.JS attempts to identify a subset of properties that are frequently accessed of an object that has been generated from a JSON-formatted string. When found, the FAD.JS runtime generates machine code that is optimized for parsing only such properties and values. In this way, the FAD.JS runtime avoids materializing data in the JavaScript memory space unless it is used by the application. Depending on the API being used, parsing is performed eagerly or lazily.

Both techniques are implemented in FAD.JS at the VM level, meaning that they directly interact with the language execution runtime, and they leverage VM-level and per-object metadata.

FAD.JS can be considered an Active library [12], that is, a library with a given interface that can self-optimize and adapt its runtime behavior depending on its usage. The Fad.js runtime described in this paper targets the JavaScript language, with a special focus on Node.js applications. Targeting Node.js as the main scenario for Fad.js is motivated by the popularity of JavaScript in many JSON-intensive domains. Despite being Node.js the main target for the runtime described in this paper, the techniques that we describe are generic, and could easily be ported to other dynamic languages. In particular, porting Fad.js to other Truffle-based languages (e.g., Ruby or R) would require minimal engineering efforts.

3.3 FAD.js API

The Fad.js runtime is exposed to Node.js applications via a compact API designed to be very familiar to JavaScript developers, as it resembles the default general-purpose JSON API that is part of the JavaScript language specification [4]. Like with the built-in JSON runtime of Node.js, a JavaScript object graph can be converted to a JSON-formatted string using the `stringify` function:

```
var data = {an:{object:"graph"}};
// Encoding using the default Node.js API
var default = JSON.stringify(data);
// Encoding using FAD.js
var optimized = FADjs.stringify(data);
assert.equal(optimized, default); // true
```

The encoding operation has the same semantics of Node.js' default one, and the encoded string produced by Fad.js is identical to the one produced by the default JavaScript encoder.

A JSON-formatted string can be converted to a JavaScript object graph using two distinct APIs, namely, `parse` and `seek`:

```
var string = '{an:{object:"graph"}}';
// Decoding using the default Node.js API
var default = JSON.parse(string);
// Decoding using the two FAD.js APIs
var fullParsed = FADjs.parse(string);
assert.equal(fullParsed, default); // true
var fastParsed = FADjs.seek(string);
assert.equal(fastParsed, default); // false
```

The first function, `parse`, has the same semantics of the corresponding JavaScript built-in function, and can be used as a drop-in replacement for it: it produces an acyclic object graph corresponding to the input JSON data, and throws an exception in case of a malformed input string. At runtime, however, the `parse` function behaves differently, as it does not allocate any string nor any object graph in the heap space of the JavaScript application. Rather, it only ensures that the string is valid (and throws an exception in case of a validation failure), returning to the application a *proxy* object that corresponds to the actual object graph of the input data. In this way, no real allocation is performed on the JavaScript heap space. After the initial validation performed *in situ*, the actual object graph is populated lazily and selectively, that is, only for the values that the application will actually read.

The second Fad.js function, `seek`, is similar to the `parse` function, but does not perform full input data validation, and is designed to be used in all the contexts where the input data is expected to be already valid, for example because it is stored in a file managed by external data sources (e.g., a logging file produced by a trusted source) or it belongs to some memory-mapped files (for example to implement data exchanges between processes). Apart from the lack of the initial input correctness validation, `parse` and `seek` behave in the same way, and share all the runtime Fad.js optimizations.

Unlike built-in libraries in Node.js, the Fad.js parsing primitives can operate on raw data. This is described in the following code snippet:

```
fs.createStream('/path/to/some/file.json');
fs.on('data', function(chunk) {
  // chunk is a raw binary buffer with utf8 encoding
  Buffer.isBuffer(chunk, 'utf8'); // true
  // Node.js must allocate a JS string:
  var p = JSON.parse(chunk.toString('utf8'));
  // FAD.js can operate on the data, directly
  var p = FADjs.parse(chunk);
});
```

The code in the example corresponds to a small Node.js application reading a JSON file (e.g., a log file): while the default JavaScript JSON parser in Node.js always needs to convert raw binary data to a (heap-allocated) JavaScript string, the Fad.js runtime can operate on the raw data, directly, thus avoiding the materialization of the string in the Node.js heap space.

4. CONSTANT STRUCTURE ENCODING

In Fad.js, an object graph is encoded to a JSON-formatted string by speculating on the constant nature of the hidden schema of the input objects. As long as such assumption holds, the Fad.js runtime can avoid or optimize most of the expensive operations that are usually involved in the generation of the JSON string. Consider the following example Node.js application:

```
connection.on('data', function(data) {
  var entry = JSON.stringify(data) + '\n';
  fs.appendFileSync('/some/file', entry);
});
```

where the `entry` object corresponds to some data with the following informal (hidden) schema:

```
var entry = {
  id: /* any number, always present */,
  name: /* any string, always present */,
  value: { /* any object value, or empty */ }
}
```

The example corresponds to a logging application in which some user data is fetched from an external source (e.g., a database connection), and stored line-by-line in a file. The JSON encoding operation is performed on multiple object instances with a similar object graph: most of the structure of the JSON data is constant, with exceptions being the `value` field, which could be empty or have any other structure. As discussed earlier, a generic JSON encoding library would recursively walk the object graph of the `entry` object, initially reading each property name (i.e., `id`, `name`, and `value`), successively retrieving for each property name the value associated with it. In doing so, it would append property names and their values to the final JSON string, performing the necessary formatting associated with each value type (e.g., converting escape characters in strings).

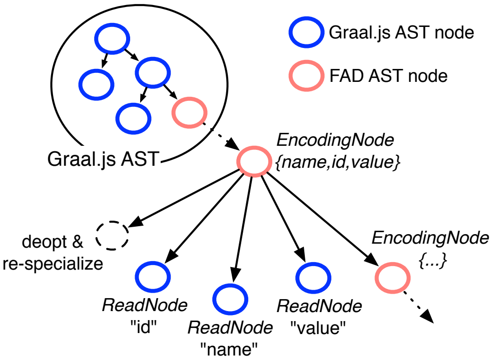


Figure 1: FAD.js encoding AST specialized for a given object shape. The FAD.js nodes perform the speculative encoding of the input object by leveraging Graal.js nodes for constant-lookup property reads. The FAD.js AST is itself inlined in the Graal.js AST calling the FAD.js encoder.

The FAD.js runtime implements the `stringify` operation in a different way, which does not require a full scan of the object properties and values for each new object instance with the same structure. Specifically, the FAD.js runtime generates a Truffle AST that resembles the structure of the object graph being encoded, and uses it to perform the encoding operation. The FAD.js runtime caches in the AST nodes all the values that are assumed compilation constant (e.g., property names and their type-related information): as long as the input objects have the schema that FAD.js expects (i.e., they have properties with the expected name and type), the FAD.js runtime avoids reading the property names of each object as well as their type, and performs the encoding operations by combining constant strings (the property names) with the runtime value of each property. The Truffle AST is built on-the-fly by FAD.js, and is specialized for the hidden JSON schema of the input object graph of the application. The generated machine code performing the encoding operation takes into account the dynamic nature of the object graph, that is, it can produce different strings depending on the presence of certain properties that are known to be *potentially* absent (e.g., `value` in the example), or that have a nature that is too heterogeneous for generating a constant compilation unit. For highly-polymorphic scenarios, i.e., when too many properties are absent or have a very heterogeneous data type, the FAD.js runtime rewrites its AST nodes and de-optimizes the compiled code to a generic version that does not rely on runtime speculation. FAD.js code generation operates as follows:

- A Truffle AST is built as a direct acyclic graph that matches the structure of the input object is created at runtime; the graph has a node for each of the object instances in the input graph (i.e., for the object `value` in the example) and edges correspond to object references. Since JSON does not allow cycles [4], FAD.js ensures that the graph is a tree.
- Each of the nodes of the AST stores a constant list of strings, which corresponds to the finite list of property names of each object instance.
- Each of the nodes also stores a constant list of *pre-formatted* strings that correspond to the formatted property names that will be used to generate the final JSON string. Such pre-formatted strings include the characters that have to be appended to generate the final encoding (e.g., the `":"` symbol, the proper string quotation characters, etc.)

The Truffle AST generated by FAD.js is effectively an executable representation of a JSON encoder that is tailored for the hidden JSON schema used by the application. It is specialized for objects with the given properties and types: as long as the input object graphs have the expected structure, executing the Truffle AST produces a valid JSON-formatted string. A Truffle AST specialized for the hidden graph of the example in the previous section is depicted in Figure 1, while Figure 2 presents the internal implementation of a specialized AST node for the same example hidden JSON schema. Encoding ASTs in FAD.js rely on runtime information provided by the Graal.js JavaScript engine, which we describe in the following section.

4.1 Object shapes in FAD.js

The FAD.js runtime operates on the JavaScript data types of Graal.js. One of the reasons behind Graal.js' performance is its dynamic object storage model [23], that is, a very efficient representation of objects in the language heap space with specialized AST nodes for fast read and write access to properties. Because JavaScript is a dynamic language, any object can have an arbitrary number of properties with arbitrary types, with any object being conceptually equivalent to a map. Property lookup operations on such dynamic objects can have a very high runtime overhead, as they might require to compute the hash of the property to be accessed for each operation. In order to reduce any hash-based lookup cost, modern dynamic languages (including Graal.js) rely on the notion of *object shape* [13, 16] (also called *Hidden classes*). An object shape is a runtime representation that encodes certain metadata regarding a specific object instance such as the number and the type of its properties. Shapes are used to perform constant-offset property lookups (rather than hash-based ones) where possible. Consider the following example:

```
// new object: empty shape
var obj = {};
// shape now contains [id:num]
obj.id = 3;
// shape now contains [id:num,name:str]
obj.name = "foo";
// shape now is [id:num,name:str,value:ref]
obj.value = {};
// both lookups can be performed with
    constant offset using the shape
var combined = obj.id + ":" + obj.name;
```

Shapes evolve at runtime, and encapsulate information about the internal structure of an object instance, that can be used later on by the language runtime to produce efficient machine code. For example, by knowing that `id` is the first property with a numeric type, the JIT compiler can generate machine code that performs the lookup operation in an efficient way (i.e., one memory load at constant offset from the base address of the object), rather than using expensive hash-based lookups (to compute the location of the selected property in the object memory layout).

```
1  public class EncodingNode extends ASTNode {
2    // pre-formatted values for this AST node
3    private final String encA = "{\'id\':";
4    private final String encB = ",\'name\':\'";
5    private final String encC = ",\'value\':";
6    // expected shape of the input object
7    private final Shape expectedShape;
8
9    /* Graal.js AST nodes used for fast
         constant-offset property lookups */
10   @Child private final ReadNode[] prop;
11   // Next encoding node in the AST
12   @Child private final EncodingNode next;
13
14   public void executeNode(JSObject input,
         StringBuilder result) {
15     // constant shape check
16     if (input.getShape() == expectedShape) {
17      /* the property name is a compilation
            constant, and the property reads
            will run a constant-offset lookup */
18       String valueA = prop[0].read("id");
19       String valueB = prop[1].read("name");
20       result.append(encA + valueA);
21       result.append(encB + valueB);
22       /* call the next AST node, potentially
           specialized for another object
           shape */
23       String valueC = next.executeNode(
24                       prop[2].read("value"));
25       result.append(encC + valueC + "}");
26     } else {
27       /* unexpected shape: rewrite */
28       throw new RewriteASTException();
29     }
30   }
31 }
```

Figure 2: A Fad.js Truffle AST node specialized to perform the encoding of an input object based on its shape. After a successful shape check, the node executes the encoding operation based on compilation-constant assumptions.

Object shapes are exploited in a similar way by the Fad.js runtime, as depicted in Figure 2. The figure describes the informal source code of a Truffle AST node generated by the Fad.js runtime to perform the encoding of an object with the same structure of the one in the example. The code in the figure corresponds to the Java code of a more complex AST node that Fad.js generates at runtime to encode the full object graph (corresponding to the AST depicted in Figure 1.) The node in the figure is specialized for the given object shape, and assumes that it will always have to encode objects with such shape. By exploiting this information, the Fad.js node can treat as compilation constants the names of the properties to be read. In this way, it can perform constant-time read of their values (whereas a general-purpose encoding library would have to list all the properties for each invocation of the parser). In addition to fast lookup of property values, the node can already make one more assumptions on the structure of the string that it will have to generate. In particular, it can treat as compilation constants some pre-initialized string tokens with pre-formatted JSON structure. When the type of an object to be parsed is not encoded in the AST node because it is a reference to another object, the AST node simply performs a call to another AST node which will specialize on the shape of the next object in the object graph (line 28 in the figure). Since all AST nodes are Truffle nodes, they are all inlined in a single compilation uint by the Graal compiler, and therefore they will execute without any dynamic dispatch overhead.

4.2 Impact on JSON encoding

Three key aspects make the Fad.js encoding approach faster than general-purpose approaches:

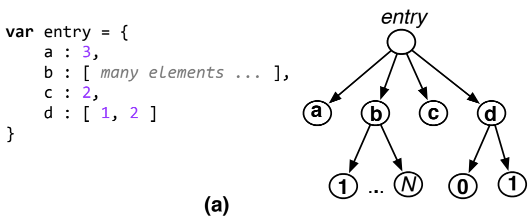
1) By assuming that property names are constant, the encoding step does not need to retrieve the list of properties from each object. Since an object instance in languages such as JavaScript can have any arbitrary number of property names, such operation can take a time proportional to the size of the object. In Fad.js this operation is constant.
2) After reading all the property names, a general-purpose encoder needs to retrieve the value of each property. Since objects in JavaScript can have any arbitrary number of properties of any arbitrary type, objects are usually implemented with a hash-based data structure (e.g., an hash map). As a consequence, reading each property value from an object corresponds to an hash-based lookup for each property name. In Fad.js such expensive hash-based lookup of property values is avoided: since property names are assumed constant, each value is resolved in the compiled code with a single constant-time memory load operation at a fixed offset in the JavaScript heap.
3) By assuming that the structure of the JSON object is a compilation constant, Fad.js does not perform a full recursive walk of the input object graph. Rather, it simply ensures that the input object has the same structure that the compiled code expects. This check can be performed very efficiently using object shapes.

5. DIRECT STRUCTURE DECODING

JSON parsing in Fad.js is implemented using a technique called direct structure decoding. The main peculiarity of this technique is that it enables the generation of efficient machine code specialized for accessing *only* the subset of the input JSON data that is used by the application, avoiding unnecessary parsing operations. Moreover, all accesses to data are performed *in situ*, without materializing in the JavaScript memory space values that are not explicitly used.

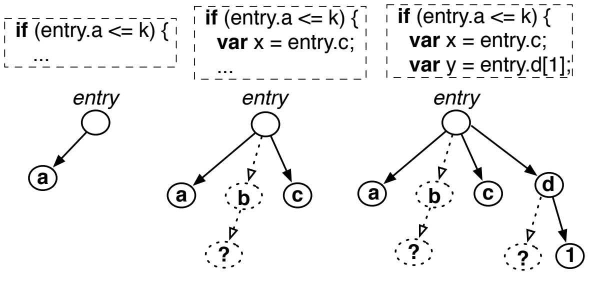
Unlike general-purpose JSON parsers, parsing in Fad.js is not performed at a single location in the code (that is, when `parse` or `seek` are called.) Rather, parsing is split into two separate operations, input data validation and selective parsing. The first operation is performed eagerly, while the latter is executed incrementally and lazily, and happens at property access time. Consider the following example:

```
// an array to store final result
var total = new Array();
// callback executed for each line
readFile.on('line', function(data) {
  var entry = FADjs.parse(data);
  if (matchCondition(entry.a)) {
    var x = entry.c;
    var y = entry.d[1];
    total.push([x,y]);
  }
});
```



```
var entry = {
    a : 3,
    b : [ many elements ... ],
    c : 2,
    d : [ 1, 2 ]
}
```

(a)
Initial (full) object graph



```
if (entry.a <= k) {
  ...
```

```
if (entry.a <= k) {
  var x = entry.c;
  ...
```

```
if (entry.a <= k) {
  var x = entry.c;
  var y = entry.d[1];
```

(b)
Runtime virtualized object graph after FAD parsing

Figure 3: String decoding (parsing) in FAD.JS. The full object graph (a) of a JSON string is not entirely materialized in the JavaScript memory by FAD.JS, and only the required subset is materialized after partial parsing (b).

The example corresponds to a data scraping application that scans a JSON file line-by-line, selecting the entries matching a specific condition (e.g., to retrieve all the log entries for a specific day). As the example suggests, the application does not need to parse the entire JSON data. A general purpose JSON parsing library, however, would always consume heap-allocated objects for each line of the file. The materialized full object graph for the JSON object in the example is depicted in Figure 3 (a). As the picture shows, the object includes two arrays of variable length (`b` and `c`) among other properties; parsing and materializing in the JavaScript heap such arrays would correspond to a considerable waste of resources. The JSON data in the example is parsed by FAD.JS using the following approach, which is also summarized in Figure 3 (b):

1) When `parse` is called (at line 5), no JSON object materialization is performed. Rather, an empty *proxy* object is created that holds a reference to the input data. We call this object the virtualized object graph of the JSON data. At this stage, no parsing operations have been performed yet.
2) After an (empty) virtual object is created, the input data is validated. This happens eagerly and *in situ*, i.e., without materializing its content in the Node.js heap space. In situ validation requires a full scan of the input JSON data, but does not require the allocation of the validated data in the JavaScript heap. During validation, the virtualized object graph corresponding to the data is populated with some minimal metadata that will be used to speed up the materialization of selected values at runtime. The metadata is called the JSON *parsing index*. Once the object has been validated, and no JSON syntax errors have been found, the virtual object is returned.
3) When a property of the virtualized object is read (i.e., the `entry.a` property in the example at line 6) the virtualized object materializes its value in the JavaScript memory space. To this end, the FAD.JS runtime parses only the subset of the input JSON data required to materialize the value of the property. Parsing is performed on the raw data, and the FAD.JS parser might start the parsing operation at any arbitrary position.
4) The virtualized object graph now stores the value that has been parsed. The next time the same property will be read by the application, its value will be read from the in-memory (materialized) representation, and no parsing operations will be performed anymore for that property on the raw data.
5) If the value of the property that has been parsed is of object reference type (e.g., `entry.d` at line 8) its value is not materialized, and another virtual object is created instead. When one of the values of the new virtualized object graph will be accessed (e.g., the `entry.d[1]` element), the FAD.JS parser will resolve the value by performing the correct incremental parsing operations.

In traditional parsers, any parsing operation starts from the beginning of the input data. The FAD.JS runtime can parse subsets of data beginning from any position. In order to speed up parsing operations, the FAD.JS runtime stores in its virtualized objects an auxiliary data index, called parsing index. Such index is used to keep track of the position of property values in the the input JSON data, and is used by FAD.JS to keep track of potential parsing positions. With the goal of saving memory space, the index does not contain the name of the properties: storing each property name would correspond to unnecessary string materializations in the JavaScript heap space, that the FAD.JS runtime would potentially not use. Rather, the index only contains an array of initial parsing positions (in the order they appear in the input data). When a property is accessed, it is responsibility of the FAD.JS runtime to chose from which index to start the parsing step. Therefore, a parsing operation may start from the first index in the parsing index, and then try all the successive ones until the required property (e.g., the `d` property in the example) is found. As a consequence of this approach, parsing indexes are not strictly required by the FAD.JS parser runtime: if no index is found, the parser would simply continue its parsing operations from the beginning of the string, or from a recent parsing position (if any). The parsing index is built while (eager) validating the input data (i.e., when the `parse` function is called), and is implemented using an array that occupies only a `int` Java value for each property in the input data. Moreover, its allocation is independent of the actual parsing operations in FAD.JS: for large JSON inputs the FAD.JS runtime can arbitrarily avoid the creation of indexes that are too big, and postpone the creation of fine-grained indexes at property access time. An exemplification of how indexes are used by the FAD.JS parser is depicted in Figure 5.

An important consideration about the FAD.JS parsing approach is that all parsing steps are performed lazily, when properties are read by the application. Beyond the obvious benefit of parsing only what is needed, the lazy nature

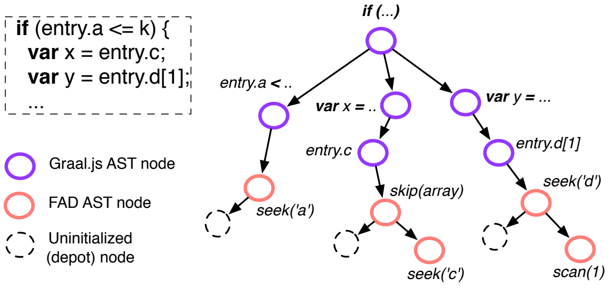


Figure 4: Lazy parsing of a JSON string in Fad.js. The Fad.js runtime is inlined in the JavaScript AST with nodes that drive the partial incremental parsing of the input string.

of the Fad.js parsing approach has another notable advantage: the parsing operations can be effectively inlined into the executable code, and can be specialized for every single access location. This has the advantage that the Fad.js parser can avoid unnecessary operations on the subset of the object graphs that it needs to materialize. For example, it can parse only the `entry.a` property when the threshold is not interesting for the application, avoiding parsing `entry.c` and other properties when they are not needed. The JavaScript source code with a Fad.js parsing operations embedded in its property access operations is depicted in Figure 4. Another important consideration about lazy parsing in the `parse` function is that the runtime semantics of the function is equivalent to the one of the default JavaScript JSON parser. In other words, lazy parsing happens transparently to the user, and the function can be used as a drop-in replacement for the JSON parsing runtime in existing applications.

5.1 Parsing using the Seek API

The example application in the previous section could also be implemented using the `seek` API introduced in Section 3.3. Using `seek` rather than `parse` would make the Fad.js runtime behave in the same way as described in the previous section, with the following notable differences:

1) The Fad.js runtime will not perform the initial eager validation of the input data. In case of a malformed input, calling `seek` will not throw an immediate exception.
2) Since validation is not performed eagerly, the Fad.js runtime will not populate the JSON parsing index when `seek` is called.
3) Validation and any updates to the parsing index are performed incrementally, when properties are accessed. In the example, this means that Fad.js validates the JSON data in three different moments, that is, each time one of the three properties is accessed by the application. This has the relevant consequence that Fad.js validates *only* the subset of the input JSON data that is required to ensure that the value can be materialized. In case of parsing errors an exception is thrown at property-access time.

The `seek` function can be considered an unsafe version of `parse` that can be used only when the input data is known to be valid, or when the application can tolerate a JSON parsing error by handling potential exceptions when properties are read (rather than when `parse` is invoked). Example scenarios where `seek` could be preferred over `parse` are all cases where the correctness of the data is guaranteed by the data source, for example if the data was produced by the same application in a previous step during some data conversion operation, or when it is received from a trusted network connection with consistency guarantees.

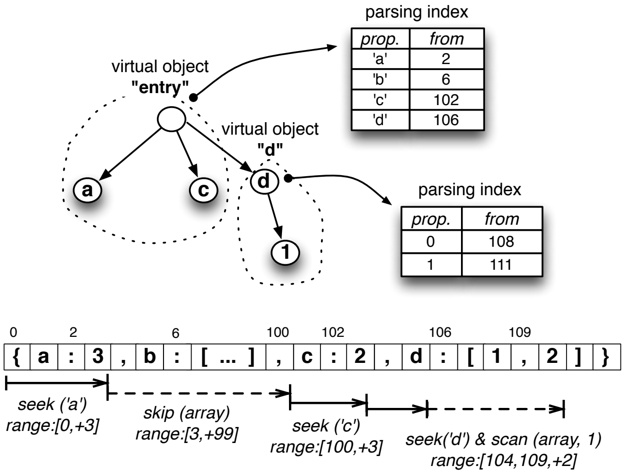


Figure 5: Parsing indexes. The Fad.js runtime builds an auxiliary index to be used to perform incremental parsing. Depending on the API used, the parsing indexes are populated eagerly during validation, or lazily while parsing a subset of the input data.

Thanks to its design, the `seek` function can effectively access only the very minimal subset of the data that is needed by the application, avoiding a full scan of the input JSON data at all when the application does not need to access the full JSON object. Using `seek` can lead to very significant speedups for applications that need to access a minimal part of large JSON objects with a complex graph structure.

5.2 Parser Specialization

Parsing operations in Fad.js are performed lazily, at property access time. For each property value to be accessed, parsing is done using a specialized JSON parser capable of retrieving the value of a single property using a special-purpose parser that can access only a subset of the entire JSON syntax.

Specialized parsers are implemented using Truffle ASTs, and are compiled to machine code via partial evaluation. Each specialized parser step in Fad.js corresponds to the combination of different, lightweight, JSON parsers that can recognize valid subset of the complete JSON syntax. For example, depending on the position of a property to be parsed, Fad.js may chose to parse only the N-th element of an array: in doing so, a specialized parser is used that can recognize only the subset of the JSON grammar for parsing array elements, and can safely *skip* the body of all the elements of the array (delimited by the comma "," symbol) that are not accessed by the application. Such specialized parser would potentially ignore the content of other array

Schema	*Objects/Values*	*Width/Length*	*Size*
*m*_books	1 / 5	5 / 50.5	482 B
*m*_catalog	2 / 4	2.5 / 8	98 B
*m*_google	17 / 54	4.12 / 32.46	3.5 kB
*m*_menu	9 / 12	2.22 / 5.18	310 B
*m*_react	8 / 30	4.63 / 10	656 B
*m*_sgml	7 / 11	2.43 / 43.1	1.09 kB
*m*_small	1 / 2	2 / 12	30 B
*m*_thumbnail	4 / 20	5.75 / 67.25	2.1 kB
*p*_avro	68 / 90	2.31 / 21.86	5.32 kB
*p*_fstab	3 / 5	2.33 / 993.2	10 kB
*p*_github	5 / 4	1.6 / 7.8	124 B
*p*_rpc-req	13 / 16	2.15 / 23.41	1.03 kB
*p*_rpc-res	30 / 28	1.9 / 22.76	1.84 kB
*p*_stat	7 / 11	2.43 / 14.33	436 B
*p*_twitter	15 / 51	4.33 / 39.29	3.4 kB
*p*_youtube	9 / 19	3 / 14.11	1.09 kB

Figure 6: Summary of the JSON schemas used by JSONBench. Schemas marked with p have a polymorphic nature, while schemas starting with m are monomorphic. All values are average values.

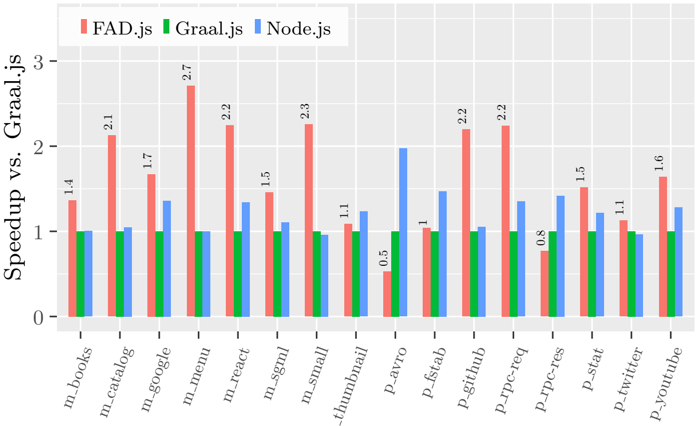


Figure 7: Performance of FAD.JS`stringify`. The FAD.JS JSON encoding runtime is consistently faster than the baseline for the monomorphic and polymorphic JSON schemas used by the benchmark.

elements and simply look for the comma separation symbol (still ensuring that strings and other values are escaped correctly); in this way, the parser can be considerably faster than a general-purpose JSON one, as it does not need to match all the possible symbols that a normal JSON parser would match. Moreover, such parser does not need to allocate and populate new objects in the JavaScript heap. Other specialized parsers that are used by FAD.JS cover all the possible parsing operations that a general-purpose JSON parse may perform, with an equivalent *skip* version that does not fully parse an object string but only validates it. Examples of such parsers are `skipObject` and `skipArray`, which skip (i.e., do not materialize) an entire JSON object (resp. array). Skipping is usually implemented with a fast linear scan of the substring corresponding to the object (resp. array). As discussed, each specialized parser is directly inlined in the Truffle AST node performing the property read operation. In this way, the FAD.JS parser is effectively able to generate highly-specialized machine code that can combine the property access operation with the other optimizations that the language runtime would already perform. In other words, the parsing step can also benefit from *all* the other optimizations that the language runtime is performing on the rest of the executed code. As an example, the language runtime can perform optimizations such as escape analysis, loop unrolling, or constant fold elimination on each of the values being parsed.

5.3 Impact on JSON decoding

The FAD.JS parser takes advantage of the following aspects that makes it more efficient than a general-purpose one:

1) By parsing only the properties that are actually used by the application, it can avoid traversing or materializing subtrees of the JSON graph that are not used.
2) By parsing only the properties that are accessed, FAD.JS performs fewer allocations of objects in the language runtime heap. Fewer allocations correspond to a lower memory footprint that can have an impact on the overall application performance (e.g., by reducing the overall pressure on garbage collection).
3) By performing the parsing operation together with property access, the specialized JSON parser can be inlined directly in the property lookup operation.
4) Since it can operate on off-heap raw data, the FAD.JS parser can be applied to any data-intensive application *before* any data is actually materialized in the Node.js heap space. Checking for the existence of a property (e.g., to perform filter operations) *without* reading its value can therefore happen in without *any* memory allocation in the language heap memory space.

6. EVALUATION

We have evaluated the performance of FAD.JS against state-of-the-art solutions such as the Node.js JSON parsing library (v6.7) and the default JSON parser in Graal.js. We consider Graal.js (v0.18) as the performance baseline for FAD.JS, as it shares with FAD.JS the JavaScript runtime environment. All the experiments were performed on a server-class machine (Intel Xeon 513 with 16 cores and 256GB RAM), with hyper-threading and turbo-mode disabled to ensure reproducibility. The standard deviation for each benchmark run is below 6%.

6.1 JSONBench

To assess the performance of FAD.JS on JSON-intensive workloads, we have designed a new benchmark, called *JSONbench*. The benchmark is aimed at measuring the performance of JSON operations in Node.js applications that make extensive usage of JSON, operating on raw data, and for which JSON encoding or decoding is the main performance bottleneck. The JSONBench benchmark consists of two JSON-intensive applications, namely, parsing and stringify. For each application, the benchmark evaluates the performance of a JSON runtime over a selection of a total of 16 different JSON object schemas that have been extracted from existing public data sources or Web services (such as a Google search result or a Twitter API response message). Each JSON object has different characteristics in terms of

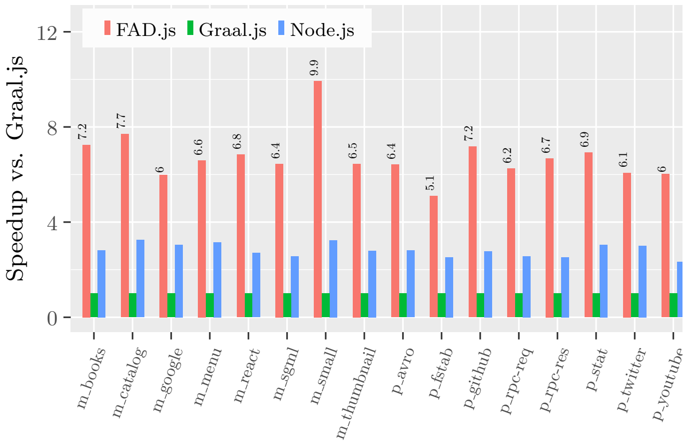


Figure 8: Performance of FAD.JS`parse` with applications accessing the first leaf property of the input object graph. The FAD.JS runtime has semantics equivalent to the default JavaScript library, and offers consistent significant speedups.

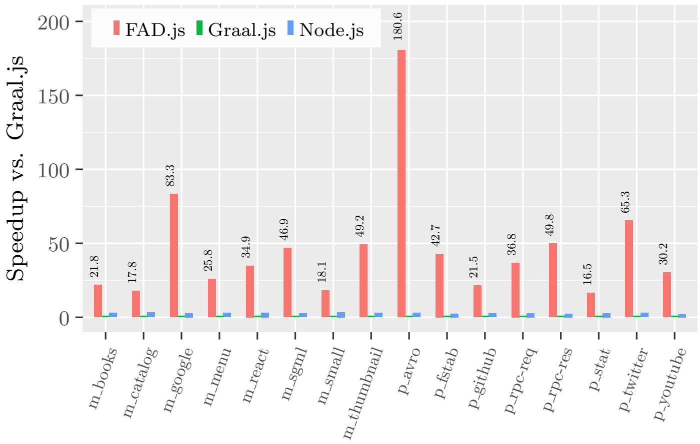


Figure 9: Performance of FAD.JS`seek` with applications accessing the first leaf property of the input object graph. The FAD.JS runtime can access only the minimal subset of the input data, with very high performance.

structure, number of elements, size, etc. An overview of the JSON objects used in the benchmark is depicted in Table 6. Each object is based on a JSON schema [20] object that is used by the benchmarking harness to generate pseudo-randomized data. After an initial data generation step which is common for both benchmarks, the two applications perform the following operations:

- The *stringify* benchmark simulates a Node.js microservice application (e.g., an Amazon Lambda function) that generates a stream of JSON objects. To this end, the application has to encode an high amount of JSON objects with a similar structure. The JSON schemas correspond to different types of messages produced by the benchmark (randomization ensures that each object is unique, and a fixed seed ensures reproducibility.) The benchmark measures the maximum throughput for the JSON encoding runtime to write the data to an in-memory data buffer.
- The *parsing* benchmark simulates a data scraping application processing JSON data in-memory. The benchmark first loads into a raw memory buffer a 1GB random-generated set of JSON objects (generated using the JSON schemas), and performs random accesses to the values of each object. The benchmark can be configured to change the number of properties that are read.

The JSON schemas used by the benchmark are divided in two main categories, namely, monomorphic and polymorphic schemas. The first category corresponds to JSON objects that always have all the property names and structure that the JSON schema prescribes. In other words, the benchmark random generator only ensures that each object has different values, but all objects always have the same fixed number of properties. The latter category corresponds to JSON objects that might also change some of their tree structure. For example, JSON objects with a same JSON schema may or may not have certain properties. The distinction between the two classes of randomized JSON data is aimed at simulating two different types of workloads, that is, workloads where JSON data is very homogeneous (e.g., when JSON is fetched from a database), and workloads where data is more dynamic, and its hidden JSON schema has only a subset of properties that are always constant (e.g., a Web service that can add or remove properties depending on invocation parameters).

The performance of the FAD.JS encoding runtime compared against Graal.js and Node.js are depicted in Figure 7. For all the considered JSON schemas, the FAD.JS runtime can effectively generate the JSON-encoded string in a time that is up to 2.7x faster than the state-of-the-art JSON runtime used by Graal.js. In general, FAD.JS performs better when the data to be encoded is monomorphic. This is expected, as the compiled code does not need to account for special cases and properties that might not exist. Still, also on polymorphic JSON schemas FAD.JS can achieve significant speedups for certain objects.

The performance of the FAD.JS decoding runtimes are depicted in Figure 8 and Figure 9. The FAD.JS `parse` and `seek` APIs offer different semantics and performance guarantees depending on the amount of properties that are accessed by the application and the type of input data validation that the application requires. In Figure 8 a first comparison of the FAD.JS`parse` runtime versus Node.js and Graal.js is depicted for accessing only the *first* property of the object graph. This benchmark is the ideal case for FAD.JS, as it requires the materialization in the JavaScript heap of one value only. The FAD.JS runtime can achieve average speedups up to 9.9x compared to the default Graal.js runtime. This is expected, and shows that performing validation of the input data on the raw memory can result in significant speedups without affecting the overall performance. The performance of the `seek` API for the same amount of property reads are depicted in Figure 9. Without validation (i.e., using `seek`), the FAD.JS runtime can access the JSON data with the best possible performance, and the decoding speedup does not depend on the size of the input JSON object, nor on its monomorphic or polymorphic nature. As a consequence, accessing complex JSON objects (e.g., `Avro`) can result in speedups up to 180x. This is expected, and shows that FAD.JS allows data-intensive applications to trade performance for correctness. Depending on the number of prop-

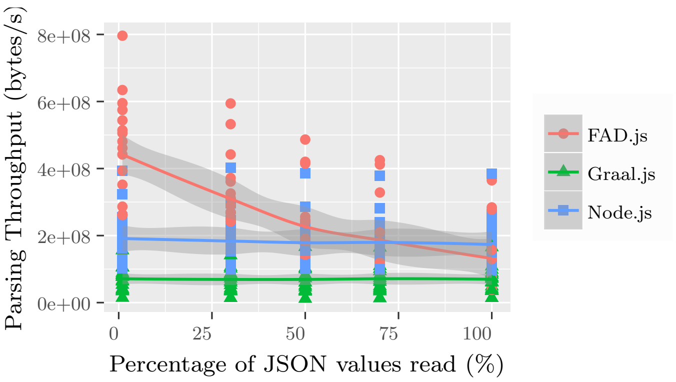


Figure 10: Performance of FAD.JSparse for an increasing number of property reads (for all the considered JSON schemas). Depending on the number of properties accessed by the application, the performance of FAD.JS tend to degrade. Still, even when the entire object graph is read, FAD.JS is faster than its Graal.js baseline.

erties that are being read, the FAD.JS performance are expected to degrade. This is depicted in Figure 10, where the benchmark is executed for an increasing number of property reads. As the picture shows, the FAD.JS runtime can effectively be faster than its state-of-the-art Graal.js baseline even when all the properties of the JSON object graph are read. This is because FAD.JS can operate on raw data, without an intermediate materialization. Nevertheless, the FAD.JS runtime is clearly preferable over the default JSON runtime when the number of properties read is small. Figure 12 describes the same scenario for `seek`. As expected, the performance of `seek` are considerably better when only a few properties are read, and degrade more quickly when the entire JSON object is accessed. Nevertheless, FAD.JS can be preferred over general-purpose parsers for JSON objects that have a simple structure (e.g., `books`) since the FAD.JS runtime can access raw data and can be inlined in the property-access operations.

6.2 Data-intensive applications

The JSONBench benchmark is designed to measure the performance of FAD.JS in applications where JSON operations are the main bottleneck. With the goal of highlighting the potential benefits of using FAD.JS in the context of more complex data processing applications, we have also measured the performance of FAD.JS when employed in combination with a popular data processing runtime. To this end, we have selected five existing benchmarks that rely on Node.js and on Apache Hadoop. All benchmarks perform some JSON operations to encode and decode data, but they also perform other operations that are not affected by JSON (for example, other performance bottlenecks could exist at the HDFS level, at the data distribution level, etc.) The goal of this benchmark is not to present an exhaustive selection of Hadoop benchmarks dominated by JSON, as writing a new benchmarking harness for Node.js and Hadoop is out of the scope of this paper. Rather, the goal is to highlight the potential benefit of FAD.JS when used in existing systems. For each of the benchmarks, we have replaced the default encoding operations with FAD.JS's `stringify` and `parse` and (or `skip`, if appropriate). The performance results are depicted in Figure 11. As the picture shows, all the applications benefit from the FAD.JS runtime, which can significantly increase the throughput of each data processing application up to 2.7x. This is expected, as JSON operations contribute significantly to the overhead of existing data-intensive applications.

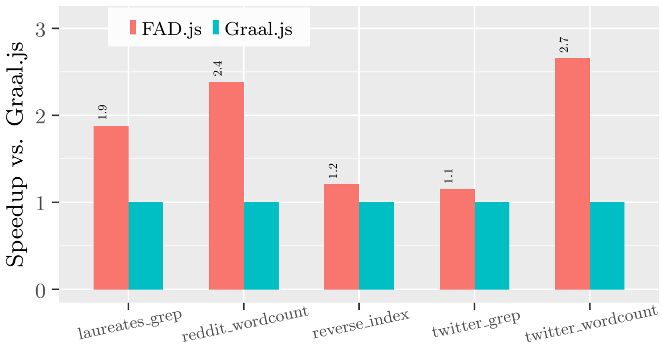


Figure 11: Performance of selected Apache Hadoop MapReduce jobs that use FAD.JS for encoding and decoding data.

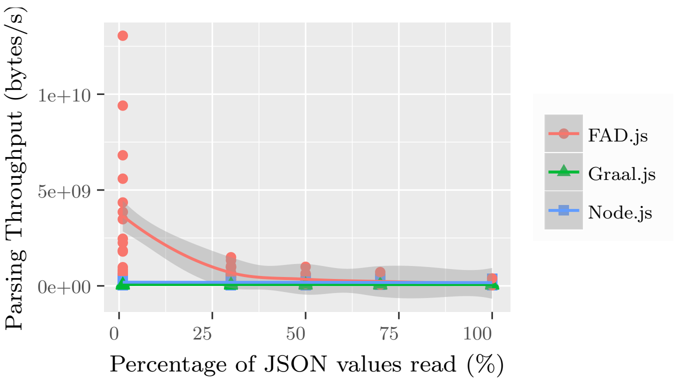


Figure 12: Performance of FAD.JSseek for an increasing number of property reads (for all the considered JSON schemas). Like with `parse`, FAD.JS is orders of magnitude faster than its baseline when a subset of the input data is accessed.

7. RELATED WORK

Lazy and incremental techniques have been used in several parsing runtimes and data formats (e.g., for XML data [19, 14]). We do not claim novelty for the incremental JSON parsing in FAD.JS, but we consider a novel contribution the integration of the parsing runtime with the language virtual machine, its just-in-time compiler, and the related techniques based on specialization, speculation, and direct access to raw data. Other relevant examples of lazy parsing approaches can be found in the domain of stream parsers (e.g., for JSON streams [3]). Such parsers usually operate on unbounded data sources, accessing only the subset of the data that the application needs. Unlike FAD.JS, all such parsers do not rely on VM-level support, and therefore cannot benefit from runtime-level optimizations. Moreover, they often require the user to program against a foreign API requiring to manually initialize and advance the parser.

The FAD.JS encoding and decoding runtimes generate machine code based on runtime knowledge of the hidden schema of the JSON data they access. To the best of our knowledge, FAD.JS is the first runtime that can optimize access to data without any static knowledge. Several examples of techniques that rely on static, *a priory*, knowledge exist. For instance, XML document projection [17], is a technique that is used to optimize XQuery operations on XML documents via static analysis. Another relevant example is the static generation of ad hoc parsing runtimes (e.g., for XML or Protocol Buffers [8]). When the schema of some data type is known at compilation time, a specialized parser can be created that can outperform a general-purpose one. All such approaches require some a priory knowledge (i.e., a schema) and cannot operate on data that is highly polymorphic. On the contrary, FAD.JS does not rely on any static knowledge.

Out of the realm of JSON and data encoding, other data-intensive systems leverage dynamic code generation and direct access to raw data. A relevant recent example of JIT-based optimizations can be found in Apache Spark [25], which relies on dynamic bytecode generation. The Spark approach shares with FAD.JS the intuition that data-intensive applications should be able to optimize certain operations to exploit the structure of the runtime data that they process. Differently from Spark, FAD.JS does not rely on bytecode generation, but rather uses runtime speculation and specialization. A relevant example of raw access to data is NoDB [11]. NoDB is a design paradigm (and a database system) designed to reduce the overhead of data accesses by exploiting in-memory indexes and direct access to raw data stored in plain text files. The NoDB approach shares with FAD.JS the vision that data-intensive applications should avoid materializing data as much as possible, and should instead rely on runtime knowledge. FAD.JS operates at a different level of abstraction than NoDB, but could potentially be adopted to speed up the access to raw data in any application that relies on JSON, including databases.

8. CONCLUSION

In this paper, we have presented FAD.JS, a runtime system for accessing JSON data using JIT compilation and direct off-heap data access. FAD.JS is based on Truffle ASTs and can offer speedups up to 2.7x for encoding and 9.9x for decoding JSON data. The FAD.JS runtime can effectively speed up existing data-intensive Node.js application, and can be used as a drop-in replacement of the default JavaScript JSON library. In the near future, we are adapting the FAD.JS runtime to operate on other Truffle-based languages (e.g., the R language [22]) and we plan to expand the techniques described in this paper to other data formats such as BSON.

Acknowledgment

We thank the VM Research Group at Oracle for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

9. REFERENCES

[1] Amazon Lambda. `https://aws.amazon.com/lambda/`.

[2] Apache Storm. `https://storm.apache.org/`.

[3] Clarinet: SAX-based Event Streaming JSON Parser. `https://github.com/dscape/clarinet`.

[4] ECMA Language Specification. `https://tc39.github.io/ecma262/`.

[5] JSON Object Notation Specification. `http://www.rfc-editor.org/info/rfc7159`.

[6] Node.js JavaScript Runtime. `https://nodejs.org/`.

[7] Oracle Graal.js. `http://labs.oracle.com`.

[8] Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[9] The Mongodb Database. `http://www.mongodb.org`.

[10] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[11] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB in Action: Adaptive Query Processing on Raw Data. *Proc. VLDB Endow.*, 5(12):1942–1945, Aug. 2012.

[12] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *ISGP*, pages 25–39. Springer, 2000.

[13] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. of POPL*, pages 297–302, 1984.

[14] F. Farfán, V. Hristidis, and R. Rangaswami. Beyond lazy xml parsing. In *Proc. of DEXA*, pages 75–86, 2007.

[15] Y. Futamura. Partial evaluation of computation process—anapproach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, Dec. 1999.

[16] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. of ECOOP*, pages 21–38, 1991.

[17] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. of VLDB*, pages 213–224, 2003.

[18] S. Newman. *Building Microservices.* O'Reilly Media, Inc., 1st edition, 2015.

[19] M. L. Noga, S. Schott, and W. Löwe. Lazy xml processing. In *Proc. of DocEng*, pages 88–94, 2002.

[20] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *Proc. of WWW*, pages 263–273, 2016.

[21] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. Snippets: Taking the high road to a low level. *ACM TECO*, 12(2):20:20:1–20:20:25, June 2015.

[22] L. Stadler, A. Welc, C. Humer, and M. Jordan. Optimizing R language execution via aggressive speculation. In *Proc. of DLS*, pages 84–95, 2016.

[23] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *Proc. of PPPJ*, pages 133–144. ACM, 2014.

[24] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proc. of ONWARD*, pages 187–204, 2013.

[25] M. Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. ACM, 2016.