Experimentation with WebAssembly: XML Parsing

Md Kamrul Hasan
Computer Science
TU Berlin
Berlin, Germany
hasan.1@campus.tu-berlin.de

Vedat Akdemir
Computer Science
TU Berlin
Berlin, Germany
akdemir@campus.tu-berlin.de

***Abstract*—In the modern world, data is a crucial factor for any organization, company or institute. eXtensible Markup Language (XML) plays a vital role in this regards. For server management, network management or for any e-commerce, XML is widely used. To work with the XML, XML parsing comes to mind. For accessing, reading or manipulating data, we need the XML parsers. There are lots of XML parsers are now existing and most of them are open source project. It is very important to choose right XML parser for the right situation for developers or IT admins. Some parsers provide the best performance but with small storage capacity and vice versa. The more users in a company are connected the bigger the XML file will be. When we try to parse an XML file with the web application, we usually do it with a JavaScript library. Sometimes for a big XML file, it is very time-consuming to load and parse the file. Often, it will block the web application. For making this efficiently, the WebAssembly concept has emerged. In this paper, we tried to show how fast and worthwhile to use a virtual machine like WebAssembly.**

***Index Terms*—WebAssembly, XML parser, JavaScript, SAX, SAX-JS, SAX-WASM**

I. Introduction

XML parsers can be divided into two independent dimensions: validating versus non-validating and another one is stream-based versus tree-based [6]. The validating parsers mainly used DTD to ensures the XML document is constructed properly whereas non-validating parsers only check that XML document is well-formed. So, we can say that non-validating is quite straightforward than validating parsers. For reading XML document with API, parsers used mainly two approaches: Stream-based and Tree-based [6]. Stream-based is also known as event-based parsers. For event-based, parsers read the chunks of a document and parse it whereas the tree based parser read the whole document. That is why, For a big size of the XML document, We will avoid the tree based parsers. Now, we try to describe the most common topics which are closely related to XML parsers.

A. XML

It stands for eXtensible Markup Language. It is a markup language. It is basically used for storing and transporting data. It is W3C recommended specifications and it has no predefined tags like HTML. So, the user can easily use their own tags and that is easy it is very easy to extend. It is both machine and human readable. XML has lots of characteristics like platform-independent, human-readable, extensible and own defined well data format. That is why it becomes crucial factors for application developers.

B. XML Parsing

Parsing is the way of splitting the information into different parts of the component. XML parser means reading XML structure and uses its application. There are lots of XML parsers. The widely used XML parsers are Document Object Models (DOM) and Simple API for XML (SAX). The XML parsing is implemented in many programming languages like Java, C++, C, VB, PHP, Perl, Python, Ruby and so on. For each of the XML parsing, we can somehow find a library in any platform.

C. DOM

It stands for Document Object Model. This present XML as a set of node objects. It is used for accessing and manipulating document. It can also add, edit, delete and get XML elements. It represents the XML as tree. The properties of DOM are ChildNodes, firstChild, nodeName, nodeValue and Attributes. The methods of DOM are getElementsByTagName(name), appendChild(node), removeChild(node) and getNamedItem(name).

D. SAX

It stands for Simple API for XML. It is an event-driven online algorithm for parsing XML. Whereas DOM parses the XML document as a whole, the SAX parses each part of XML document sequentially. By the definition of Oracle, It is serial access protocol that is ideal for stateless processing, where the handling of an element does not depend on any of the elements that came before. It requires less memory and has fast execution facilities[13].

E. WebAssembly

WebAssembly is a low-level binary format for web and it is mainly compiled from other languages like C, C++ or Rust. It is offered maximum performance like native speed. It is abbreviated as wasm which is a stack-based virtual machine[9]. It is mainly designed for the web application to work alongside with JavaScript not for replacing JavaScript. It uses WebAssembly JavaScript API for loading WebAssembly modules and then both of them will share functionalities. WebAssembly modules can be imported into a web(or

Node.js)app, exposing WebAssembly functions for usages via JavaScript[9]. The main goal of the WebAssembly is fast, be portable, keep secure, be readable and debuggable and don't break the web[9].

F. WebAssembly in web-platform

Here are several key concepts like module, memory, table and instance which are needed to understand how WebAssembly runs in the browser. The memory in WebAssembly is resizable ArrayBuffer. A module is a WebAssembly binary which is compiled by a browser to convert into executable machine code. Table is the resizable typed array of references. An instance is used at the run time with memory, table and set of imported values[9]. How WebAssembly interact with the web is showed in the image below [14].

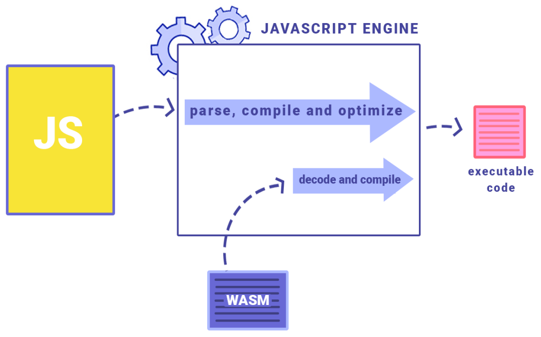


Fig. 1. Relative time spent processing WebAssembly in JavaScript engine [14].

G. WebAssembly in application

There are three main entry points[9]. Firstly, we have to compile a C/C++ application through Emscripten. Then we can write or generate WebAssembly directly at the assembly level. We can also use a Rust application and targeting WebAssembly as its output. These are great resources for people who are trying to figure out where to start, but they lack some of the tooling and optimizations of Emscripten. The Emscripten tool is able to take just about any C/C++ source code and compile it into a .wasm module and the necessary JavaScript code for loading and running the module and an HTML document to display the results of the code[9]. In this experimentation, we tried to find out the performance of WebAssembly by benchmarking with JavaScript library for XML parsing. We used SAX-based XML parser for our experimentation. Now, we try to discuss over both of XML parsers.

1) XML parsing with JavaScript: As we have to benchmark our result with JavaScript parser library. We chose SAX-JS[8] as the standard JavaScript package. It is a SAX-style parser for XML and HTML[8]. In the SAX-JS, There is a parser function and in that parser we can send argument like strict and opt. The default values for both of these arguments is false. Opt is used for string formatting whereas strict is used for checking whether the parser is jerk or not. Besides these, there are lots of event and method to use this library.

2) XML parsing with WebAssembly: For working and making compatible the WebAssembly in web application, firstly We have to make wasm binray file from C/C++/Rust. SAX-WASM mainly developed from the motivation from SAX-JS. To make the SAX-JS faster and low memory consumption, it is mainly designed. This is first streamable XML parser which is written in Rust. For our experimentation, we use this WebAssembly library as XML parser.

The rest of the paper is organised as follows. Section 2 describes the background studies and related works for our experimentation. After that, we show approaches for our experimentation. In the section 4, we discuss about the evaluation and benchmark result. In the final section, we give summary of our paper.

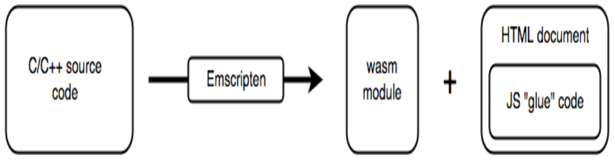


Fig. 2. Workflow of WebAssembly[9]

II. Related Work

In 2007, Wei Lu et al. implemented that a multicore system parallels XML serialization is better than XML serialization using parallel programming. There are numerous XML parsing available. They built for the same goal achievement but they have some significant differences in terms of performance, specifications and conformance to the standards[6]. For making the comparison, they used two most popular parsers: Xerces (a Java-based parser) and .NET parser (a Microsoft based parser)[6]. In their studies, they are able to show that Xerces Java is the best parser. But for a large data set, xParse is the best. When we try to make a decision on choosing XML parsers, It is not only performance which we should consider but also company needs, API support, and license fees.

In 2002, Srikanth Karre et al. used five validating XML parsers which are coded in Java. When they studied those parsers, three main features are kept in their minds. These three aspects are accuracy, speed and storage. Their main concerned is that how could a developer select a parser when he tries to work with any XML parsers. Their study is focused to find out how parsers are different from each other. Mainly three research questions are highlighted. First, How they are similar with respect to well-formedness, validity and namespace. Secondly, finding similarities according to acceptance and rejections of the XML document. Lastly, how they are related in terms of speed. In their papers, they used three independent (XML feature, Class of Document and File size) and three dependent (Acceptance, Rejections, and Speed) variables.

In 2003, Kai Ning et al. proposed that DTD-based XML parser which is the requirement of the network management. There are two types of structure in an XML document. One is physical and another one is a logical structure [2]. There are lots of storage units in the physical structure. Those units will constitute two ingredients: markup and character data. In the context of logical structure, the XML documents include declarations, elements, comments, character references and processing instructions. The logical and physical must fulfill the requirements of well-formedness and validity constraints[2]. Those are the two main factors for evaluating an XML document. For DTD-based XML parser, well-formedness checking and validity verify are two main features[2]. We can relate to the physical and logical structure. The markup in a physical structure is used for describing the storage and logical structure. Then the character data consists of different elements. Each of these element is started with start tag and end with an end tag. In this way, they make a hierarchy like a tree. There are four types of modules in DTD-based XML. These are Lexical Analysis, Error Handling Routine, Validity Verifying and Well-formedness [2]. The main problem in DTD-based XML is that it does not support all data types.

In 2007, Wei Lu et al. implemented that for a multicore system parallel XML serialization is better than XML serialization using parallel programming.

There are two wasm compiled XML parsers existing: Expat-WASM [10] and SAX-WASM[7]. Expat is actually a Mozilla project and a C library for XML parsing. It is actually stream-oriented parser [11]. Expat uses a callback function and also the event handler. Firstly, the callback functions will feed the whole XML document and then divide this document into pieces. After that, the event handler will take that part of the XML document and parse it. In that way, the expat can parse a huge document which is not actually fit into the memory. Expat-WASM is XML parser which is based on expat. Firstly, the expat is compiled into wasm and then move the wasm in the NPM packages. That is how it helps to real, battle-tested XML parser with zero run-time dependencies.

On the other hand, SAX-WASM is a SAX-style based XML parser which is designed from the concept of SAX-JS[7]. As it is working as a SAX basis, users do not have to concern about the size of XML files. It can work with any sizes of files. SAX-WASM mainly has written in Rust and compiled to wasm.

III. Implementation Approach

Our main goal is to show how and why the WebAssembly should be used for getting better performance than JavaScript. As the WebAssembly only works on C, C++ and Rust libraries. We found some existing C/C++ libraries. The most widely used libraries are Xerces, RapidXML, TinyXML, PugiXML. All of these libraries are C/C++ coded. We tried to use those libraries to compatible with the WebAssembly. For working with the WebAssembly, we have to compile the C/C++/Rust libraries into wasm binary through a compiler and then load it and use it via the JavaScript. But somehow, those libraries did not work out for our case. We then went for an alternative approach. We used Expat-WASM[10] which are wasm compiled. But this NPM package did not work with any browsers. After that, We chose SAX-WASM[7] which is now active. And it works well for our experimentation. So, for the side of WebAssembly, We selected SAX-WASM and for the JavaScript side, We chose SAX-JS. Since SAX-WASM is built from the inspiration of the SAX-JS. We built two different small projects for doing our experimentation: One is for testing SAX-JS and another one is for SAX-WASM. We used webpack for setting our project environment. For every project, We have one HTML and one JS file and the main XML parser library which on we work. After that, We tested different sizes of XML files for both libraries in different browsers. We then read the execution time for both of these project and compares them.

A. *How SAX-WASM works*

1) Pseudocode of SAX-WASM: We described the pseudocode for calling SAX-WASM library and its functionalities[7] below.

```
IMPORT SAX–WASM
async function loadAndPrepareWasm()
saxWasmResponse:=
await fetch("Directory of sax–wasm.wasm")
saxWasmbuffer:=await saxWasmResponse.arrayBuffer()
const parser = new SAXParser(
SaxEventType.Doctype |
SaxEventType.CloseTag |
SaxEventType.OpenTag |
SaxEventType.Text
)
//Instantiate and prepare the wasm for parsing
ready := await parser.prepareWasm(
new Uint8Array(saxWasmbuffer))
if ready then
RETURN parser

loadAndPrepareWasm().then(processDocument)
Procedure processDocument(parser)
currentTag := false
parser.eventHandler := (event, data) =>
    IF event is SaxEventType.CloseTag then
      currentTag := false
     IF event is SaxEventType.OpenTag then
       IF data.name is tagNameOfXML then
        currentTag := true
        ELIF event is SaxEventType.Text then
            IF currentTag then
             PRINT VALUE

startingTime := performance.now()
Load the XML file into xml1
```

```
length := xml1.length
chunkSize := 1000
StartchunkSize := 0
while StartchunkSize less than length then
parser.write(xml1.substr(StartchunkSize,
chunkSize))
StartchunkSize +:= chunkSize
endingTime := performance.now()
totalTime := endingTime − startingTime
PRINT the totalTime
parser.end()
```

As we used the SAX-WASM which is developed by Justin[7]. We try to explain the whole mechanism of SAX-WASM. The SAX-WASM parser is a WebAssembly executable and WebAssembly only allows working with number types (floating point integers, unsigned integers, hexadecimal, etc.). We first have to convert the original XML into utf8 code points. This becomes a Uint8Array where each byte represents a grapheme within the document. Then this Uint8Array should be written to the shared memory buffer and then pass pointer and length to SAX-WASM. The parser then looks at the significant bits in each byte within memory to determine how many bytes make up each character within the XML document. Once a utf8 character is extracted from the Uint8Array, the parser tokenizes the document looking for markers that determine the entities within it: open tags, attributes, comments, cdata which are aggregated. Once the entities are found, they are processed based on the SaxEventType passed in. This processing takes the aggregated entity and converts it into a utf8 encoded byte array which represents a valid JSON document. These bytes are then written to the shared memory with the pointer and length retained. The event type, pointer, and length are then passed back to JavaScript where they are converted back to a string and finally parsed into JSON using JSON.parse().

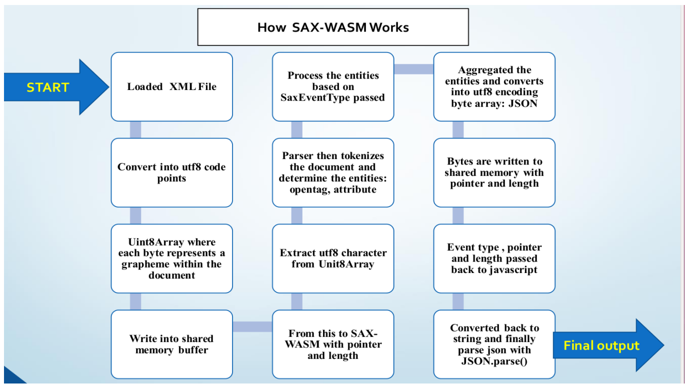
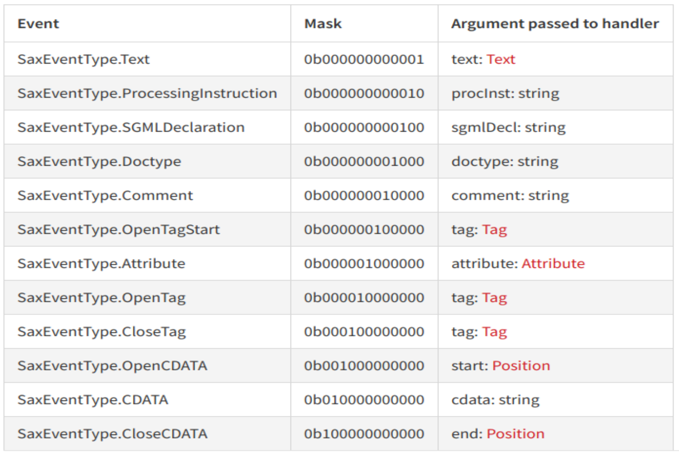

How SAX-WASM Works
START
Loaded XML File
Convert into utf8 code points
Uint8Array where each byte represents a grapheme within the document
Write into shared memory buffer
From this to SAX-WASM with pointer and length
Extract utf8 character from Unit8Array
Parser then tokenizes the document and determine the entities: opentag, attribute
Process the entities based on SaxEventType passed
Aggregated the entities and converts into utf8 encoding byte array: JSON
Bytes are written to shared memory with pointer and length
Event type , pointer and length passed back to javascript
Converted back to string and finally parse json with JSON.parse()
Final output

Fig. 3. Workflow of SAX-WASM

B. How SAX-JS works

1) Pseudocode of SAX-JS: We described the pseudocode for calling SAX-JS library and its functionalities[8] below.

```
IMPORT SAX library
```



Event	Mask	Argument passed to handler
SaxEventType.Text	0b000000000001	text: Text
SaxEventType.ProcessingInstruction	0b000000000010	procInst: string
SaxEventType.SGMLDeclaration	0b000000000100	sgmlDecl: string
SaxEventType.Doctype	0b000000001000	doctype: string
SaxEventType.Comment	0b000000010000	comment: string
SaxEventType.OpenTagStart	0b000000100000	tag: Tag
SaxEventType.Attribute	0b000001000000	attribute: Attribute
SaxEventType.OpenTag	0b000010000000	tag: Tag
SaxEventType.CloseTag	0b000100000000	tag: Tag
SaxEventType.OpenCDATA	0b001000000000	start: Position
SaxEventType.CDATA	0b010000000000	cdata: string
SaxEventType.CloseCDATA	0b100000000000	end: Position

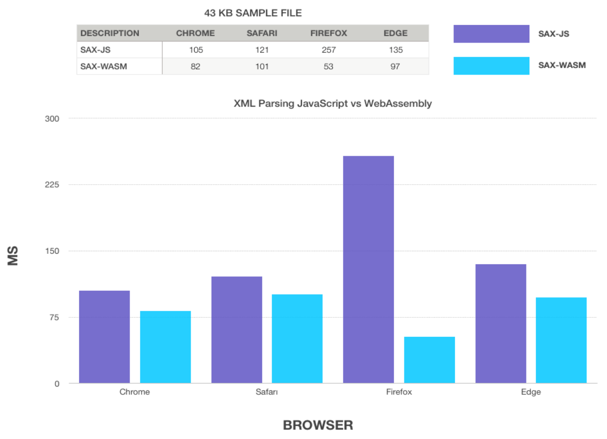
Fig. 4. Entities of SAX-WASM[7]

```
Call parser procedure with value TRUE
Set currentTag to FALSE
Call the onclosetag {
set the currentTag to FALSE
}
Call the onopentag
{
IF tag.name is tagnameOfXMLFile
set the currentTag to TRUE
}
Call the ontext event
{
IF currentTAG is TRUE
THEN
print or store the value
ENDIF
}
startingTime:=performance.now()
Load XML file
Call the the procedure with parser.write(XML file)
Call the procedure with parser.end()
endingTime:= performance.now()
ActualTime:=endingTime−sartingTime
```

For the JavaScript side, we used SAX-JS library. We will now give a brief description of this library. It is a SAX-style parser. The final goal for this library is to parse HTML and XML document. But it is now working only on XML for parsing. This parser works with XML entities in attribute and text nodes. In strict mode, unknown entities will be unsuccessful. There is a parser function in SAX-JS. It will receive different argument like strict, opt, trim, normalize, lowercase, xmlns, position and strictEntities. There are lots of built-in method such as write, close and resume etc. The write method is used for writing bytes on the stream and close method is for stopping the stream. To handle error, SAX-JS uses the resume method. The parsers object has some members: line, column, positions (Those are used for marking the parsing position in the XML documents), startTagPosition which indicates the current tag. As we already know that SAX-JS is event-driven XML parsers. Events are the important part of the SAX based XML parsers. The events which are used in SAX-JS are : error, text, doctype, processinginstruction, sgmldeclarartion, opentagstart, opentag, closetag, attribute, comment, opencdata, cdata, closecdata, opennamespace, closenamespace, end, ready and noscript.

IV. Evaluation

We tested SAX-WASM and SAX-JS with different size of XML files through different web browsers. We executed our project on Chrome, Safari, Firefox and Edge. For those browsers, the SAX-WASM is working fine. Firstly, we tried with the very small size (43KB) of XML files. After we analyze the benchmark result, we found out that SAX-WASM gave us better performance than the SAX-JS for all browsers. SAX-WASM gave the best result in Firefox among all browsers. Then we tried with a 2MB size of XML files. In that case, SAX-WASM still gave us a better performance than the SAX-JS. Chrome is the best in that sizes of the file.For the 20MB of XML file, SAX-JS and SAX-WASM gave the almost same execution time. Then we tried a big files(84MB). In that case, SAX-JS had better performance than SAX-WASM. Safari gave the best performance in the case of SAX-WASM. But we know that the SAX-WASM should be always better than the SAX-JS. That is the main reason for which the concept of WebAssembly is established. After long research on the SAX-WASM library and also consulting with the developers of SAX-WASM. We found out the problems behind the worst performance in the case of SAX-WASM for big files.

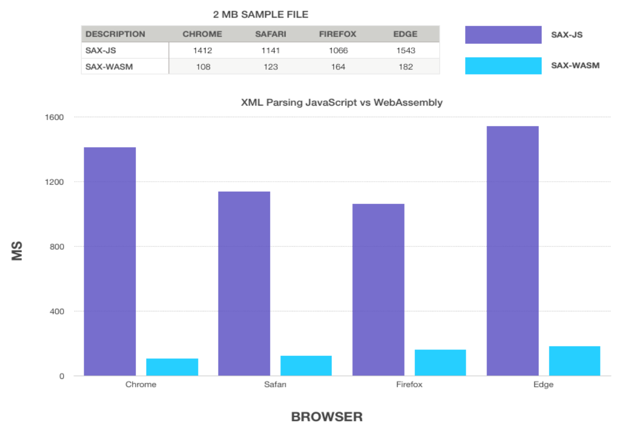

43 KB SAMPLE FILE

DESCRIPTION	CHROME	SAFARI	FIREFOX	EDGE
SAX-JS	105	121	257	135
SAX-WASM	82	101	53	97

Fig. 5. Benchmark result: File size: 43KB

A. *The reason behind the performance of SAX-WASM is less than SAX-JS*

The pointer represents the location within memory and it is the starting for reading. The length represents how many bytes to read. When the JavaScript side receives the data, performance becomes poor. Because it is converting utf8 bytes into


2 MB SAMPLE FILE

DESCRIPTION	CHROME	SAFARI	FIREFOX	EDGE
SAX-JS	1412	1141	1066	1543
SAX-WASM	108	123	164	182

Fig. 6. Benchmark result: File size: 2MB

a string and then parsing that string into JSON which can take up to 60 percent of the total time. For better performance, we have to subscribe to the fewest number of events as possible and We should not use SaxEventType.CloseTag, Since this can produce a very large JSON document which can make it slow to parse by JavaScript. We already know that SAX-WASM has a JavaScript counterpart that converts a buffer into utf8 chars and then those chars are parsed into JSON. About half of the time is spent in this operation and there is not a lot that can be done to speed this up. It's also dependent on the number and types of event which are asked for. The more events we deliver JSON data the more parsing on the JavaScript side which slow things down incrementally.For instance, just for a quick test on a 240k document with no JSON buffer conversion or JSON parsing and with attribute events turned on, It will take 25ms to parse the entire document. When buffer conversion takes place and JSON parsing is turned back on, this same document takes 43ms to process. When it adds CloseTagevents, this parse time creeps up to 70ms. With no events, the time of parsing is at 5ms.

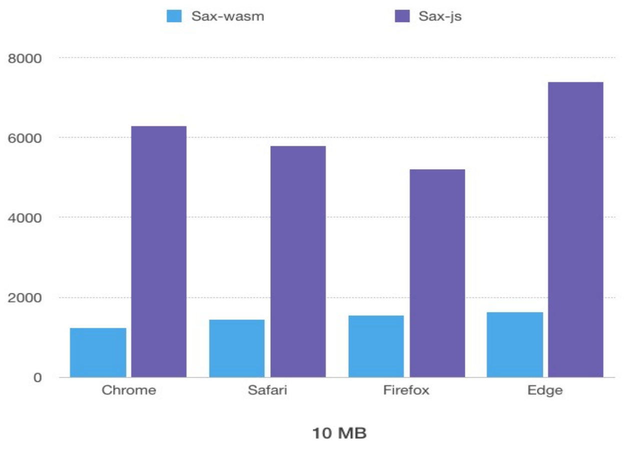


Fig. 7. Benchmark result: File size: 10MB

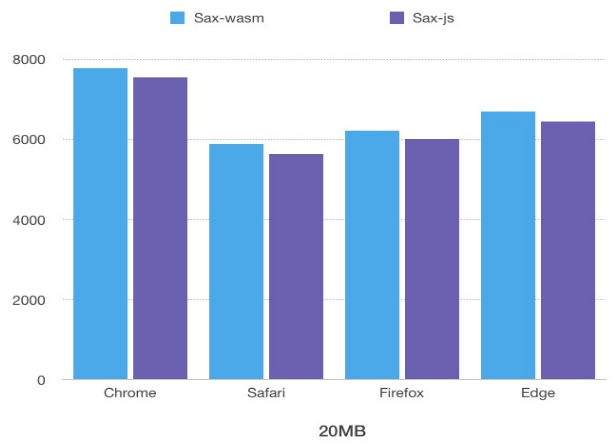
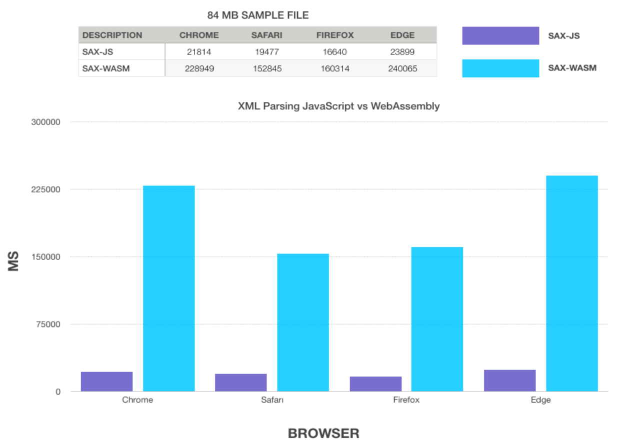


Fig. 8. Benchmark result: File size: 20MB



84 MB SAMPLE FILE

DESCRIPTION	CHROME	SAFARI	FIREFOX	EDGE
SAX-JS	21814	19477	16640	23899
SAX-WASM	228949	152845	160314	240065

Fig. 9. Benchmark result: File size: 84MB

B. Steps to make SAX-WASM faster

There are a number of optimizations that the SAX-WASM is now improving. The main problem is that the JavaScript side makes it slow. The EventReader which reads values directly from the buffer at the moment when they are accessed which is similar to how a Proxy works. This avoids a lot of overhead since there would be no need to parse an entire JSON string regardless of whether the subscriber is actually interested in the data.

V. Conclusion

An XML parser is a library which provides an environment for a client to access and read XML document. It will also examine the XML format and validate the XML document. Above all, all browsers have built-in XML parsers packages or libraries. In our experimentation, we used SAX-based XML parsers. For the web application, there are lots of JavaScript libraries for XML parsing. We used SAX-JS which is based on event-driven XML parsing. In some cases with JavaScript, we cannot get the result as our expectations for some cases. That is because of the way JavaScript interacts with the web. That is why WebAssembly has emerged. It is a kind of virtual machine which is supported by all major browsers. After hours of research on the implementation method, we came to the final decision to use SAX-WASM which is wasm compiled XML parsing library. As we mentioned in our previous sections that wasm is created by compiling the C/C++/RUST through emscripten compilers. It is a binary file which is loaded by JavaScript and then use its functionalities. It is acted like native speed when we execute the application which is developed by wasm. As WebAssembly support all widely used browsers, we do not have to worry about the cross-platform. We tested SAX-WASM and SAX-JS in different browsers and get different performance time. In some cases, SAX-JS gave better result but for most of the cases, SAX-WASM is better than SAX-JS.

Acknowledgment

We greatly thankful to the Fraunhofer Fokus and our supervisors Stefan Pham and Daniel Silhavy for their valuable suggestions and advice.

References

[1] Nicola, M. and John, J., XML Parsing: a Threat to Database Performance International Conference on Information and Knowledge Management,2003,pp.175-178.
[2] Kai Ning, Luoming Meng,Design and Implementation of DTD-based XML parser,proceedings of ICCT2003.
[3] Srikanth Karre and Sebastian Elbaum, An Empirical Assessment of XML Parsers,2002.
[4] Robert A. van Engelen, Constructing Finite State Automata for High-Performance XMLWeb Services,in the proceedings ofInternational Symposium onWeb Services and Applications (ISWS) 2004.
[5] Tong, T. et al,Rules about XML in XML, Expert Systems with Applications, Vol. 30,No.2,2006, pp. 397-411.
[6] Su Cheng Haw ,G. S. V. Radha Krishna Rao, A Comparative Study and Benchmarking on XML Parsers,Advanced Communication Technology, The 9th International Conference (Volume:1) ISSN:1738-9445 , 2-14 Feb. 2007 pp. 321 32.
[7] Justin, 'SAX (Simple API for XML) for WebAssembly', 2019. [Online]. Available:https://www.npmjs.com/package/sax-wasm
[8] Isaacs,'sax-js',2017.[Online].Available: https://www.npmjs.com/package/sax
[9] Mozilla,'WebAssembly',2019.[Online].Available:https://developer.mozilla.org/en-US/docs/WebAssembly
[10] Hildjj,'expat-wasm',2018.[Online].Available:https://github.com/hildjj/expat-wasm
[11] xml.com,'What is expat',2019.[Online]. Available:https://www.xml.com/pub/1999/09/expat/index.html
[12] University of Washington Repository,2019,[Online]. Available:http://www.cs.washington.edu/research/xrldatasets/
[13] Oracle,'SAX',2019,[Online]. Available:https://docs.oracle.com/javase/tutorial/jaxp/sax/index.html
[14] LogRocket,'WebAssembly',2019,[Online]. Available:https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71