Performance Analysis of XML APIs

Eric **Perkins**

Margaret **Kostoulas**

Abraham **Heifets**

Morris **Matsa**

Noah **Mendelsohn**

Abstract

XML, as a data interchange technology, delivers key advantages in interoperability due to its flexibility, expressiveness, and platform-neutrality. The broad range of applications and growing base of users for XML technologies has driven the development of common tooling, providing a consistent, robust infrastructure on which to build applications. These advantages have spurred widespread adoption of SOAP and web services, as a key component of the next-generation of business computing infrastructure. It is increasingly clear, however, that the advantages of XML result in a heavy performance penalty, and that current parsing technologies are unable to meet the performance demands of an XML-based computing infrastructure.

Current implementations of XML parsers use a variety of different APIs including DOM, SAX, and, in the web services world, JAX-RPC. Some progress has been made in recent years in improving XML parsing performance by replacing heavyweight APIs like DOM to transfer data from the parser to the application with lighter-weight methods, such as SAX, or application-specialized options, like JAX-RPC. While lighter-weight, the event-based SAX API is more difficult for application developers to program to than the straight-forward document model of DOM. Nonetheless, SAX is seen as a performance enabling API. Furthermore, even with lighter-weight APIs, performance remains a significant obstacle to many Web Services applications. This paper discusses various aspects of SAX and other current XML APIs, and analyzes aspects of their performance through the use of micro-benchmarks demonstrating how much of current XML parsing time is being lost to these inefficiencies. With better performance, and advantages in terms of usability and robustness, a new or modified API for XML would enable fast and easy XML parsing and validation across broad classes of XML applications.

Table of Contents

1. Introduction 3
2. Methodology 3
    2.1. Representative APIs 3
    2.2. Test Case 4
    2.3. Baseline 5
    2.4. Platform 5
3. Impact on Parsing Performance 5
    3.1. Transcoding 5
    3.2. Object Creation 6
    3.3. Measurements 6
    3.4. Discussion 7
4. Impact on Application Performance 8
    4.1. Test Case 8
    4.2. Measurements 9
    4.3. Discussion 10
5. Impact of Extraneous Data 10
    5.1. XPath API 11
    5.2. Discussion 11
6. Conclusions 11
Bibliography 13

1. Introduction

XML delivers key advantages in interoperability due to its flexibility, expressiveness, and platform-neutrality. The broad range of applications and growing base of users for XML technologies has driven the development of common tooling, providing a consistent, robust infrastructure on which to build applications. These advantages have spurred widespread adoption of SOAP and XML-based web services, as key components of the next generation of business computing infrastructure. It is increasingly clear, however, that with these advantages use of XML can also carry a heavy performance penalty, and that XML parsers currently in use are unable to meet the performance demands of an XML-based computing infrastructure.

With the increased use of XML in performance-critical scenarios, significant progress has been made in improving the performance of XML parsing [Chi04][Tak05][Eng04]. With these improvements in performance, however, parsers are operating in a regime in which seemingly ancillary considerations, such as choice of API, can have a significant effect on performance.

Current implementations of XML parsers use a variety of different APIs including DOM[DOM], SAX[SAX], and, especially in the web services world, custom-generated object trees, such as JAX-RPC[JAX-RPC]. That some heavy-weight APIs, such as DOM can have a significant impact on performance is qualitatively well known. Quantitatively, however, the impact that APIs have on parsing performance is not well understood.

API design can also have an effect on usability and application performance. Lower-level APIs often gain a performance advantage by requiring extra work of the application. While an overall win in a tightly written application, the increased application complexity has costs that go beyond performance.

In this paper we explore the performance characteristics of XML APIs through the use of micro-benchmarks. In this way, we propose to isolate costly aspects of APIs, so that their relative effects may be quantitatively compared. As a basis of comparison, we select three APIs: DOM, SAX, and JAX-RPC. These APIs are used as representatives of different approaches to API design, and are used to explore the impact that these approaches have. In Section 2, “Methodology”, we set out the benchmark methodology, discuss each of the representative APIs and introduce a baseline measure of parsing performance. In Section 3, “Impact on Parsing Performance”, we explore the penalty paid in the parser for the use of individual APIs. In Section 4, “Impact on Application Performance”, cost of API navigation is discussed. Finally, in Section 5, “Impact of Extraneous Data”, the costs associated with data that is not used by the specific application are explored through the use of a prototype API. Conclusions, and considerations regarding the design of future APIs are presented in Section 6, “Conclusions”.

2. Methodology

The effect of API design on XML parsing performance is not well understood. In the past, performance studies have focused on parsing as a whole, measuring the differences between various parsers implementing the same APIs. Where performance-aware APIs are used, they are typically tied to a particular parser implementation, and their performance advantage considered to be part a feature of that parser.

In this paper, we isolate the cost of APIs from parser implementations through the use of micro-benchmarks. Generated and hand-written code is used to simulate the overhead of particular computations in isolation from the rest of the system. Because the operations are very simple, the measurements may be considered to be representative of a highly optimized actual implementation. To the extent that operations are left out of the micro-benchmarks, the overall picture presented can be considered conservative.

2.1. Representative APIs

The experiments presented in this paper are used to evaluate the impact of API design on performance. In order to capture a variety of approaches, three representative APIs were chosen: DOM, SAX, and JAX-RPC. DOM is one of the original APIs for XML. It represents the document in an in-memory tree structure, where each attribute, element or character data value is a node. The DOM API is designed to be highly generic, and language-neutral. Additionally, DOM is designed to be modifiable, allowing applications to change the content and structure of XML documents. In order to support a wide range of uses DOM nodes carry a great deal of information, and the resulting data structure is heavy-weight.

SAX is a lightweight, event based API for XML. Designed for simplicity, SAX offers a lower-level interface to XML documents that can be used to build higher-level data structures. SAX is widely used, and is generally regarded as an efficient API for XML parsing (in contrast to DOM, which is understood to be slower, but more powerful).

JAX-RPC is an API designed for web service interactions in which an XML document is bound to a typed, Java object hierarchy. This binding is performed based on a schema for the input document. Each type in the schema becomes a named Java Bean, with appropriate accessors for each of its elements and attributes. This makes the API a natural choice for XML-based programming, since the XML data is rendered into the programming language natively. On the other hand, JAX-RPC is not suitable for many applications in that it does not preserve important aspects of the document. For example, the ordering of children is lost, as is intervening whitespace. Additionally, since all of the simple values are stored in native types, such as double and int, the lexical forms of these values are also lost—that is, the difference between `num1` and `num2`, below, is not preserved.

```
<num1>
3.14159 </num1> <num2>003.14159</num2>
```

Each of these APIs has individual strengths and weaknesses, and they represent three very different approaches to API design. In the analysis undertaken in this work, we focus on general implications for performance, rather than on the individual shortcomings of any API. Thus, for example, JAX-RPC may be taken as a general design approach, applicable to most application scenarios, individually, even though the actual API is suitable only to certain applications.

2.2. Test Case

To motivate the experiments, and to lend coherence to the various micro-benchmarks presented, all of the measurements are based on an actual XML instance. The instance is a short purchase order, taken from the XML Schema Primer[Schema0], with all insignificant whitespace removed. The size of the instance is less than a kilobyte, and is representative of a very simple XML message.

```

<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
```

```
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

2.3. Baseline

The measurements presented in the following sections are all given in microseconds per trial. In order to provide a baseline for this measurement, we compare the costs measured for API production and use to a baseline parsing measurement. For this measurement, we use a highly tuned parser that does all the work to parse an input document, and check well-formedness, but produces no API. The parser is used as the common base for optimized parsers with various APIs, and as such, provides a simple example of the baseline cost of parsing without API overhead. For the test input document, the time to parse a single instance is 35 microseconds. In the following sections, this number will be used to baseline all of the numbers presented, and will be referred to as *No API Parsing*.

2.4. Platform

All of the experiments presented were performed in Java, on a single machine. The IBM 1.4.1 Java virtual machine was used, on an 1.6 GHz Intel Centrino processor. To achieve the fine-grained measurements required for micro-benchmarking, we employed a native-code timer that makes use of the Pentium TSC register, which stores clock cycles. The native-call overhead of the timer was measured and removed, and the experiments were all warmed up to ensure full JIT compilation. Furthermore, each test was averaged over ten thousand iterations, to account for garbage collection times.

3. Impact on Parsing Performance

In this section we explore the impact that API design has on parser performance through the use of micro-benchmarks. By examining the performance characteristics of APIs in isolation, we determine conservative estimates of the performance cost to the parser that each API incurs. As discussed in Section 2.3, “Baseline”, the micro-benchmark results are compared with a baseline parsing measurement, to establish the scale of the performance penalty that each API introduces.

3.1. Transcoding

The syntax of XML is specified in Unicode. XML documents, however, are stored and transferred in an encoded form. Similarly, data is passed to the application in some encoded form. In practice, these two encodings are often not the same[1]. This means that APIs that pass character data to the application must often *transcode* this data. Transcoding is a relatively expensive operation, and may be expected to contribute significantly to the cost of text-oriented APIs like SAX and DOM.

[1]In Java, for example, the application almost always wants character data in UTF-16, the encoding of Java strings. Many XML documents, however, are stored and transmitted in UTF-8, since this is significantly more compact for western languages.

In the sections below, the overhead of transcoding is measured and presented separately from the rest of the overhead of API production. In all of the APIs measured, character data is passed as UTF-16. In our measurements, we assume that the input document is UTF-8 encoded. Transcoding micro-benchmarks are generated in which the relevant input byte sequences are transcoded according to the requirements of the API.

3.2. Object Creation

In addition to transcoding, API production can incur overhead in object creation. This includes the cost of memory allocation and data structure population. For tree-based APIs such as DOM and JAX-RPC, object creation is significant, and the performance of these APIs suffer accordingly. Object creation overhead is measured by producing the API, assuming complete knowledge of the input document and already transcoded values. In the case of SAX, the (minimal) overhead of handler method invocation is included in the object creation number.

3.3. Measurements

Transcoding and object-creation micro-benchmarks were run simulating the overhead of DOM, SAX, and JAX-RPC. The results are given in Table 1, “Cost of Transcoding and Object Creation.”.

	DOM	**SAX**	**JAX-RPC**
Transcoding	50.4	44.7	11
Object Creation	24.5	0.05	5.5

Table 1. Cost of Transcoding and Object Creation.

To produce the DOM tree for the input document, all of the character data and all of the element names and attribute names and values must be transcoded from UTF-8 to UTF-16. Additionally, all of these are wrapped in a `java.lang.String`, before being incorporated into the DOM tree. All of this overhead is measured as transcoding overhead. The total transcoding time for the DOM API is 50 microseconds. The additional overhead required to build the object tree using the transcoded strings is 24 microseconds. Thus, for DOM, the transcoding time alone is greater than the baseline parsing time (which was measured at 35 microseconds in Section 2.3, “Baseline”). When combined with object creation, the cost of the API is over twice the cost of No API Parsing.

As with DOM, the SAX API requires all of the character data and all of the element names and attribute names and values to be transcoded to UTF-16. In SAX, however, the character data is not wrapped in string objects, but is passed as raw character data. While somewhat difficult for applications to use, this does provide some savings in API production. The transcoding costs of SAX were measured at 45 microseconds, ten percent less than DOM. As opposed to DOM, however, SAX avoids a significant amount of object creation, which was seen to be a significant cost for DOM. In SAX, only attributes are added to a data structure before being passed to the application. This overhead, and the overhead of the handler method calls was simulated, and measured at only 0.05 microseconds. Thus, SAX provides significant savings in object allocation costs, but still incurs a significant transcoding cost. The total cost of the SAX API is significantly (29%) more than the cost of No API Parsing.

As with SAX and DOM, all of the character data and attribute values passed by JAX-RPC must first be transcoded to UTF-16[2]. In all cases, the character data is then wrapped in a `java.lang.String`. Because no element or attribute names are transcoded, however, the time required to transcode all of the values is significantly reduced. For JAX-RPC, the transcoding time was measured at 11 microseconds, less than a quarter of that for either SAX or DOM.

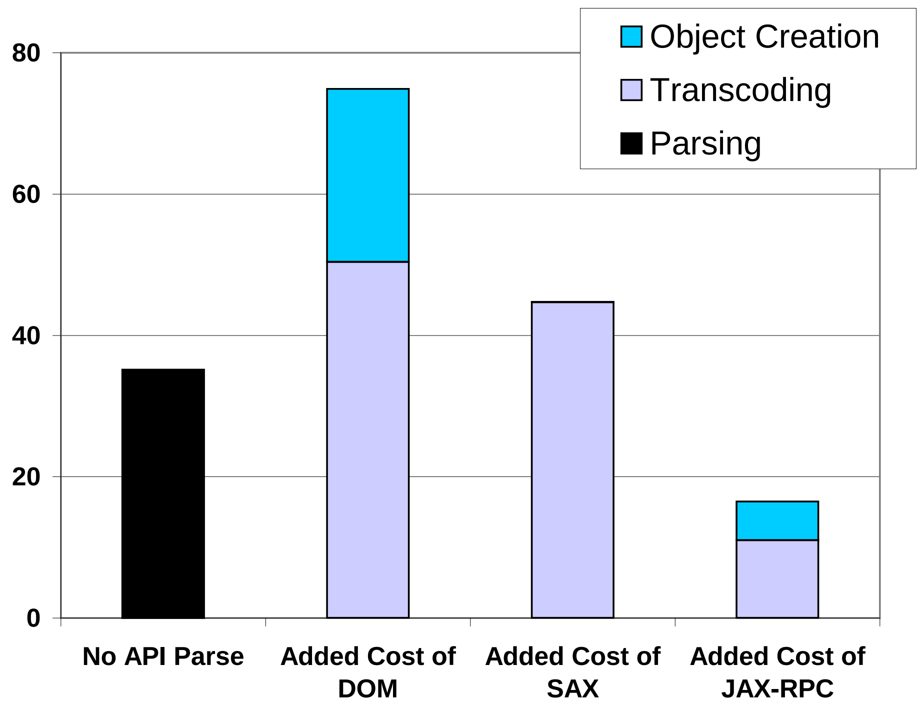
Building the JAX-RPC structure is, however, object intensive. Further, several of the fields, such as those of `int` type, must be translated from the string form to their value forms. Nonetheless, the additional cost is compensated for by

[2]Much of the data in JAX-RPC is, in fact, passed as value types such as `int`, or `BigDecimal`. In principle, a parser could process these fields directly from the UTF-8, if it was aware of the output form. For the purposes of these measurements, however, we chose to take the conservative route of transcoding all of the values, and then converting them to their value form.

the fact that JAX-RPC requires fewer objects than DOM. The measured object creation cost is 5.5 microseconds, nearly a quarter of the cost for DOM. When object creation is combined with the transcoding, the total cost of the JAX-RPC API is the smallest measured. Furthermore, it is the only API for which the additional cost of API production is less than the baseline cost of Parsing.

3.4. Discussion

The measurements presented in Figure 1, “Impact on Parsing Performance” show that the production of standard APIs has a significant impact on perceived parser performance. Even for relatively efficient APIs like SAX, the added cost of the API is greater than the cost of parsing alone. In heavier-weight APIs, like DOM, the costs are even greater.



Additional cost in microseconds of API production of DOM, SAX, and JAX-RPC, with a baseline measurement of parse time.

Figure 1. Impact on Parsing Performance

In the case of JAX-RPC, we have shown that the cost of the API itself is significantly less than the other APIs. This reduced cost is explained by the fact that the JAX-RPC tree reduces, somewhat, object creation and transcoding costs. The measurements however, are based on a theoretical performance limit. Normally, object-binding APIs are implemented as a binding layer on top of lower-level APIs such as SAX or DOM. This layering is obviously inefficient, since it simply adds the cost of the JAX-RPC API to the cost of the low-level API. Given an aggressively integrated approach to JAX-RPC, however, where parsing and object-binding are combined in a single pass, the result is an API that is significantly more efficient to build.[3]

[3]It should be noted that JAX-RPC is not a full representation of the infoset[InfoSet] for any document, since it does not preserve insignificant whitespace, or the lexical forms of many data values. For many applications, this is a good compromise. For other applications, however, more data will be required, with the consequent costs in API production.

4. Impact on Application Performance

In the last section, we discussed the impact of various API designs on parser performance. In this section, we discuss the impact that API design has on usability and application performance. Just as different APIs have different performance characteristics on the parsing side, they also have differing impacts on the performance and usability on the application side. Whereas the performance impact of API construction is often hidden from the user, and therefore seen as part of the "performance cost of XML", application-side impacts are visible to users, and therefore present a more direct challenge to users of XML tooling.

The APIs examined in this paper provide varying levels of abstraction to the user of the XML document. SAX provides a very low-level view of the input, whereas JAX-RPC provides a customized high-level view. On the parsing side, performance gains are sometimes seen for lower-level APIs, such as SAX. On the application side, however, these gains may be partially lost due to increased application complexity.

While not large, the impact on application performance of the APIs is not negligible. In the worst case, SAX, the cost of navigating the event stream is more than ten percent of the No API Parsing time. The costs of complex application code, however, go far beyond the measurable performance impact. The cost, in programming and maintenance difficulties, incurred by complex application code is only hinted at by the performance measures presented below. The complete set of results is presented in Figure 2, “Impact on Application Performance”

4.1. Test Case

In order to test application complexity, we simulate navigating the APIs for a theoretical use case. Using the same test instance purchase order, we assume an application in which the total cost of the purchase is calculated, including shipping and local tax. This requires the application to extract from the instance, the shipping city, state and zip-code, and the part-numbers, quantities, and prices of all of the items. The nine selected fields are shown below in bold. For each of the examples below, only the minimum work to locate the data values is performed. Because each API has a different data format, and so as not to over-emphasize numeric value conversion in the analysis, no conversions are measured.

```
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

4.2. Measurements

The micro-benchmarks for API navigation costs of DOM, SAX, and JAX_RPC are given in Table 2, “Cost of API Navigation”.

	DOM	**SAX**	**JAX-RPC**
API Navigation	2.1	4.1	0.07

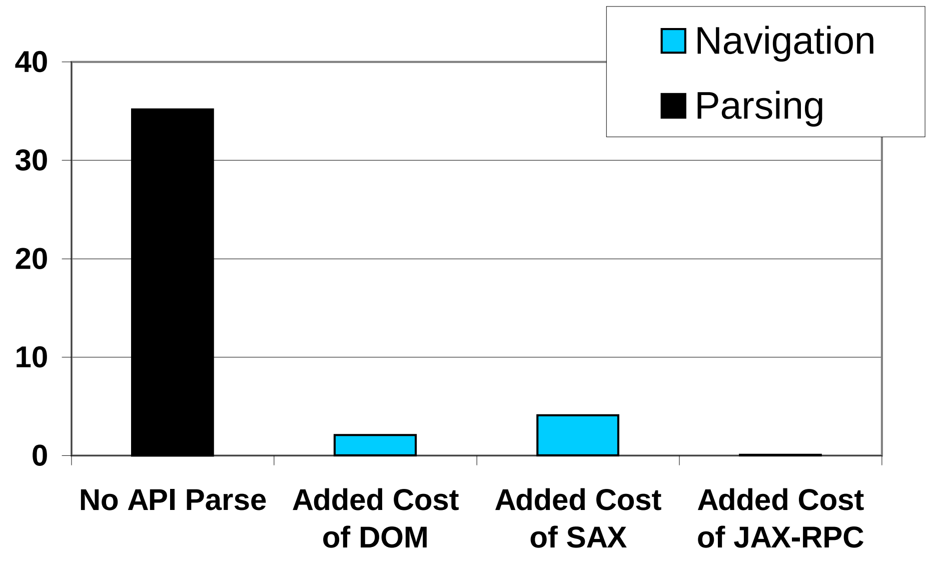
Table 2. Cost of API Navigation

With the use of helper functions, such as those provided by `org.apache.xerces.util.DOMUtil`, DOM navigation is fairly simple, if somewhat inefficient. The measured time to access the nine fields is 2.1 microseconds, or about six percent of the No API Parsing time.

The SAX API is much more difficult to navigate than DOM. At each start-element event, the application must compare the element name against known names to track the parser's position within the document. Furthermore this position must be kept as state, and updated with each event. The measured time for this code is 4.1 microseconds, or nearly twelve percent of the No API Parsing time. This is twice as expensive as DOM navigation and, at over ten percent, represents a significant cost that has been passed on to the application to pay for the improved performance of the parser.

Navigation of the JAX-RPC structure is extremely simple; this is the design point of the API. The fields of the structure are accessed as Java members and repeated fields are indexed as arrays. As a result, the navigation time is extremely small: 0.07 microseconds or 0.2 percent of the No API Parsing time. In this case, the increased complexity of the JAX-RPC API has payed off for the application in usability, and the results are clear even in the performance measurements. While JAX-RPC is not ideal for all use cases, the concept of high-level usable interfaces can be adapted for most use cases, and similar results may be expected.

4.3. Discussion



Cost in microseconds to Application of API Navigation for purchase order test case.

Figure 2. Impact on Application Performance

The navigation costs imposed on the application by the measured APIs are clearly much smaller than the costs associated with API production. As such, the application overhead does not contribute overwhelmingly to the so-called "XML Performance Penalty". In the worst cases, however, the costs are not negligible. Further, the cost of API navigation is directly visible to the application programmer, and as such contributes significantly to the perceived inefficiency of interacting with XML data. In as much as the costs are indicative of increased complexity of application code (as is clearly the case for SAX), the performance numbers also hint at a much larger problem of code production and maintenance.

5. Impact of Extraneous Data

In Section 4, “Impact on Application Performance”, the cost of API navigation was measured with respect to a motivating use case. Using the purchase order, a hypothetical billing application was described in which nine data items were retrieved from the input to calculate a total cost including local tax and shipping. The performance impact, while small compared to the cost of parsing, was in some cases still significant. This is because the APIs present much more information than the application is interested in. As a result, the application wastes time navigating the API, looking for the desired content.

In the case of JAX-RPC, which is well-suited to this particular access pattern, navigation is made easy with the typed tree. In general, however, extraneous data may present significant navigation overhead to the application. In addition to application impact, the unused data has an obvious impact on parsing performance. In the test application, only nine fields are used, but in every API, all twenty-four fields are passed to the application. In the text-oriented APIs, the situation is worse, since all of the element and attribute names are also passed. As demonstrated in Section 3, “Impact on Parsing Performance”, the cost of transcoding and object creation for all of this data is high. In this section we illustrate this cost through an example selective API based on XPath[XPath].

5.1. XPath API

As a demonstration of the impact of unused data on parsing and application performance, we experimented with a simple, example API that passes only requested data. In this API, the desired data is identified by XPath before parsing begins. During parsing, SAX-like events are thrown as the desired data is encountered, identifying which path was matched, and the character data that was found at that location. This API is overly simple for most real applications, but provides a simple measure of selective API performance. Setup for the parser is demonstrated in the example code below.

```
String[] paths = new String[] { "purchaseOrder/shipTo/city",
                                "purchaseOrder/shipTo/state",
                                "purchaseOrder/shipTo/zip",
                                "purchaseOrder/items/item/@partNum",
                                "purchaseOrder/items/item/quantity",
                                "purchaseOrder/items/item/USPrice", };
XPathHandler handler = new XPathHandler() {
    public void onPath(String p, char[] data, int start, int len) {
        System.out.println(p+" = '"+new String(data,start,len)+"'");
    }
};
XPathParser xpp = XPathParser.getParser(paths,handler);
xpp.parse(input);
```

To measure the performance of the XPath API, a custom parser was built using the baseline No API Parser, with XPath matching integrated into the parser, minimizing overhead. The result is a parser that accepts any number of simple paths and reports matching data, as described above. The performance of the XPath API parser was measured using the same timing mechanism as the micro-benchmarks of the previous sections. For the test example, the parse-time for the XPath Parser was 38.5 microseconds. The difference between the XPath API parsing time and the baseline No API parsing time of 35 microseconds is used as a cost measurement comparable to the micro-benchmarks already presented.

5.2. Discussion

The XPath API experiment demonstrates the power of selective reporting of data. By shifting nearly all navigation costs from the application to the parser, the API avoids any significant impact on application performance. Further, by avoiding API production costs, such as object creation and transcoding, for all of the unused data, the XPath API also minimizes the impact on parsing performance. As a result, the total cost of the XPath API is only 3.5 microseconds, or less than ten percent of the baseline parsing time.

While the XPath API provides a good example of the performance potential for selective APIs, it is not really a functional API. For example, application programmers might want to be able to retrieve whole subtrees of the input, specified by XPath. Native values, such as `double`, or `int` might also be desirable, and returning results in a table would be more usable than the unwieldy event-based API. As previous results with JAX-RPC indicate, however, none of these features presents significant performance obstacles. A carefully designed API could incorporate many such features without compromising the performance demonstrated here.

6. Conclusions

XML parsing has historically been a performance bottleneck for many applications. Recent developments in parsing performance, however, are changing this equation. Indeed, with a carefully tuned parser, the cost of actual parsing is often overwhelmed by the cost of producing APIs that are not designed for performance. Combined with the impact that these APIs have on application performance, the perceived cost of XML parsing is much larger than the actual cost of parsing the document.

In the previous sections we have presented a series of micro-benchmarks intended to measure the costs incurred by an API for a range of design choices. In Figure 3, “End to End Performance Estimates (in microseconds)” below, the results are combined to produce end-to-end performance estimates for parsing with each of the APIs tested. These data clearly show the significant overhead of the DOM API. Even SAX, which is widely perceived as an efficient API, has an overhead in excess of the total cost of parsing.

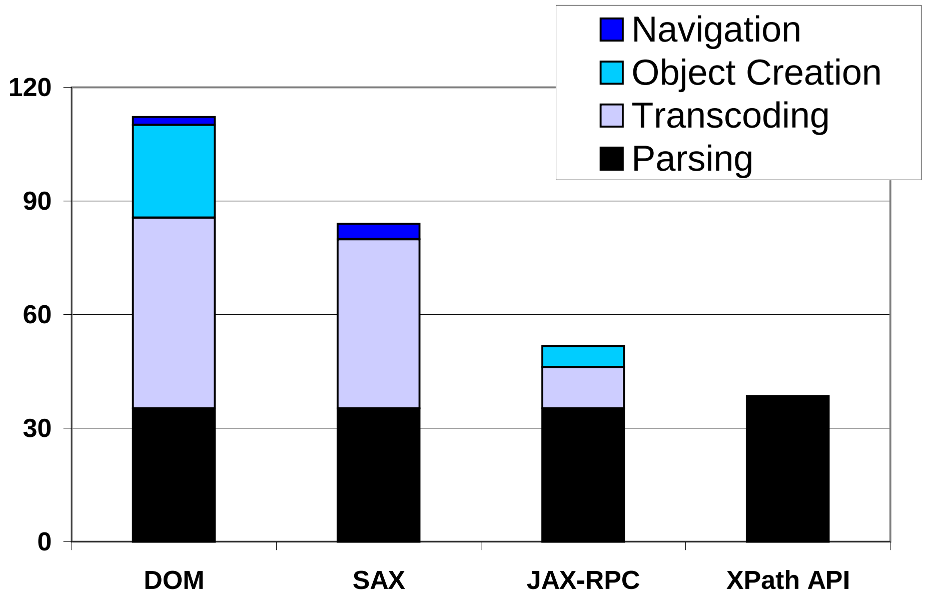


Figure 3. End to End Performance Estimates (in microseconds)

Specialized APIs, such as JAX-RPC have the potential for significantly better performance for at least some use cases if aggressively integrated with parsing. Typically, however these high-level APIs are built as layers on top of a generic, low-level API. This makes their costs additive with those of the low-level API, and removes any opportunity for efficiency.

With the XPath API experiment, we demonstrate an aggressive integration of parsing with a high-level API that limits transcoding and object creation and also minimizes navigation costs by only reporting requested data. The resulting overhead is less than ten percent of the baseline parse time. While only an experimental example, this demonstrates the power of an API that is designed for performance and ease of use.

As XML technology is used in more and more performance-critical contexts, increasing attention is being paid to XML parsing performance. While efforts to improve parsers have yielded good results, parsing efficiency is often held back by generic, low-level APIs. If XML tooling is to reach its full potential in terms of performance, new APIs that are designed from the outset for performance must be developed. Through our example XPath API, we show that high-performance is an achievable target for API design. Furthermore, the methodology used in the analysis of existing APIs demonstrates that, through the use of simple micro-benchmarks, performance limits can be determined for APIs based solely on their design. It is our belief that such micro-benchmarks can be used in the crucial design phase to inform API design, in order to develop standardized APIs that are both usable and performance-oriented, and that this method of design is key to the performance of future XML tooling.

Re-format page sizes

Bibliography

[XML] *Extensible Markup Language (XML) 1.0, Second Edition,* Tim Bray et al., eds., W3C, 6 October 2000. See http://www.w3.org/TR/2000/REC-xml-20001006.

[XML11] *XML 1.1, W3C Recommendation,* 4th February 2004, Francois Yergeau, John Cowan, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. See http://www.w3.org/TR/2004/REC-xml11-20040204/ .

[XMLNS] *Namespaces in XML,* Tim Bray et al., eds., W3C, 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114/ .

[InfoSet] *XML Information Set,* John Cowan and Richard Tobin, eds., W3C, 16 March 2001. See ht-tp://www.w3.org/TR/2001/WD-xml-infoset-20010316/ .

[Schema0] *XML Schema Part 0: Primer Second Edition* D. C. Fallside and P. Walmsley, World Wide Web Consortium, 28 October 2004. See http://www.w3.org/TR/xmlschema-0/

[Schema1] *XML Schema Part 1: Structures Second Edition,* H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, Editors. World Wide Web Consortium, 28 October 2004. See http://www.w3.org/TR/2004/REC-xmlschema-1-20041028 .

[Schema2] *XML Schema Part 2: Datatypes Second Edition,* P. Byron and A. Malhotra, Editors. World Wide Web Consortium, 28 October 2004. See http://www.w3.org/TR/2004/REC-xmlschema-2-20041028 .

[SAX] *SAX,* http://www.saxproject.org/ .

[Tak05] *An Adaptive, Fast, and Safe XML Parser Based on Byte Sequence Memorization,* Toshiro Takase, Hisashi Miyashita, Toyotaro Suzumura, Michiaki Tatsubori, WWW 2005, March 2005.

[Eng04] *Constructing Finite State Automata for High-Performance XML Web Services,* Robert van Engelen, International Conference on Internet Computing 2004: 975-981

[Chi04] *A Compiler-Based Approach to Schema-Specific XML Parsing,* Kenneth Chiu and Wei Lu, First International Workshop on High Performance XML Processing (Satellite of WWW2004).

[DOM] *Document Object Model (DOM) Level 1 Specification* V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood eds., W3C, 1 October, 1998. See ht-tp://www.w3.org/TR/REC-DOM-Level-1.

[JAX-RPC] *Java API for XML-Based RPC (JAX-RPC).* http://java.sun.com/webservices/jaxrpc/index.jsp.

[XPath] *XML Path Language (XPath),* J. Clark and S. DeRose, eds., World Wide Web Consortium, 16 November 1999, See http://www.w3.org/TR/xpath.

Biography

Eric **Perkins**
IBM Corporation [http://www.ibm.com]
Cambridge
Massachusetts
United States of America

Dr. Perkins is a Software Engineer working for IBM's T. J. Watson Research Center on next-generation Web technologies, focusing on high performance XML parsing, validation, and deserialization, and Web Services infrastructure. Before joining IBM, Eric received his Ph.D. in Information Technology from MIT in 2001 for research in algorithms for discrete element simulation. Previously Eric studied Engineering at Brown University (ScB. 1997), and MIT (SM. 1999).

Margaret **Kostoulas**
IBM Corporation [http://www.ibm.com]
Cambridge
Massachusetts
United States of America

Margaret Gaitatzes Kostoulas is a Software Engineer and has worked for IBM for eight years. She received a Masters degree in Computer Science from Purdue University. She is currently working on performance of the XML processing stack. In the past, she worked on Universal Usability projects and on cross-discipline projects designing virtual prototyping laboratories for computational science.

Abraham **Heifets**
IBM Corporation [http://www.ibm.com]
Cambridge
Massachusetts
United States of America

Mr. Heifets is a Software Engineer and has worked for IBM for two years. He received his Bachelors and Masters degrees from Cornell University. In the past, he has worked on a world champion robotic soccer team, a publish-subscribe system for location based services, and single-agent search algorithms. He currently works in the field of high-performance processing of the XML stack.

Morris **Matsa**
IBM Corporation [http://www.ibm.com]
Cambridge
Massachusetts
United States of America

Mr. Matsa has been a Software Engineer and a Researcher at IBM for eight years. In that time, he has developed many software prototypes, which have been integrated into five different IBM products. He also founded the IBM Extreme Blue internship program. He currently works in the field of high-performance processing of the XML stack. He received degrees in Mathematics and Computer Science from MIT, and then a Masters degree in Computer Science from MIT.

Noah **Mendelsohn**
IBM Corporation [http://www.ibm.com]
Cambridge
Massachusetts
United States of America

Mr. Mendelsohn is a Distinguished Engineer at IBM Research in Cambridge, MA. He has made significant contributions to the development of SOAP, the W3C XML Schema Language, and JavaBeans, and is a member of the World Wide Web Consortium Technical Architecture Group. During his career he has had extensive experience in the areas of operating systems, programming languages, and distributed systems. Mr. Mendelsohn has a Masters Degree in Computer Science from Stanford University and a Bachelors Degree in Physics from MIT.