XML Screamer:
Integrated, High-Performance XML Parsing, Validation and Deserialization

Margaret Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, Martha Mercaldi
Outline

- Introduction
- Why is XML Parsing Slow
- XML Screamer: Design
- XML Screamer: Performance
- Conclusion
XML Performance

- XML is everywhere
- Increasingly, XML is being used in processes that demand high-performance
- XML is widely seen as underperforming
- Validation is even worse
- Often, though, XML is used for exactly the kinds of things you want to validate
Why are traditional XML parsers slow?
How fast is your computer?

- An XML Parser must read through its input (and update the bytes of its output)

- A 1 GHz Pentium can read through an input buffer at approximately 100 Mbytes/sec which is approximately 10 cycles/byte.
How fast are traditional XML processors?

- Xerces-C 2.6, non-validating, using SAX: approximately 6 Mbytes/Sec/GHz, which is ~165 cycles/byte

- Expat version 1.95.8 for Windows using its UTF-8 API is about 12 Mbytes/sec/GHz (~80 cycles/byte)

- Remember: processor character scan rate = 100 MByte/sec/GHz (10 cycles/byte)

- What in the world are these XML processors doing for 80-160 cycles with each byte they pick up? That may be on the order of 300+ instructions per byte!
Input:

```xml
<inventoryItem>
  <quantity>10</quantity>
  ...
<inventoryItem>
```

Schema (abbreviated syntax):

```xml
<element name="inventoryItem">
  <sequence>
    <element name="quantity">
      <simpleType base="xsd:integer"/>
      <maxInclusive="10000"/>
      <minInclusive="0"/>
    </simpleType>
  </element>
  </sequence>
</element>
```

Output

```c
struct inventoryItem {
  int quantity;
};
```
A traditional XML Parser/Deserializer

```java
class inventoryItem {
    int quantity;
};
```

XML:

```
<inventoryItem>
    <quantity>10</quantity>
</inventoryItem>
```
A traditional XML Parser/Deserializer

**Input:**

```xml
<inventoryItem>
  <quantity>10</quantity>
  ...
</inventoryItem>
```

- Convert to UTF-16
- Validate against "inventoryItem"
- Throw Sax event
- Match against "inventoryItem" in deserializer
  - Discard Sax event
A traditional XML Parser/Deserializer

**Input:**

```xml
<inventoryItem>
  <quantity>10</quantity>
  ...
<inventoryItem>
```

- Validate against “quantity”
- Throw Sax Event for “quantity”
- Match against “quantity” in deserializer
- Convert “10” to UTF-16 and integer
- Validate as 0 < quantity < 10000
- *Discard Integer*

- Sax event for UTF-16 “10”
- Convert “10” to integer (deserializer)
- *Discard Sax event*
- Copy integer to “quantity” field in structure
Traditional parsers: performance issues

- Lots of expensive UTF-8 to UTF-16 conversions
- String compares done in UTF-16 (typically larger)
- Work duplicated between validator and deserializer
- Repeated data conversions (e.g. string/integer)
- Data copying
- Possible object & memory management overhead for SAX events
- Incremental reporting even when documents are small
XML Screamer

An Integrated Approach to High Performance XML Parsing, Validation & Deserialization
The XML Screamer Project

- **Goal:** show how fast XML and XML Schema Processing can be

- **Approach:** an XML Schema compiler that optimizes across layers that are traditionally separate -- scanning, parsing, validation & deserialization are integrated
What XML Screamer is…

- A compiler for XML Schemas
- Given a schema and a desired output API, generates a parser that:
  - Parses and validates against the schema
  - Populates the desired API
- Screamer compiler is in Java, produces parsers in C or Java
- C output much better tuned…much easier to study. All results reported here are for C.
Screamer Parser APIs

- **NoAPI:**
  - just reports well-formedness and root validity

- **Business object**
  - like gSOAP, or JAX-RPC, SDO, etc.

- **SAX**
  - knowledge of schema allows pre-computation of some SAX output
An XML Screamer Parser/Deserializer

```java
class inventoryItem {
    int quantity;
};
```

Output

XML Screamer Custom Generated Parser

XML

```xml
<inventoryItem>
  <quantity>10</quantity>
<inventoryItem>
```
An XML Screamer Parser/Deserializer

Input:

```xml
<inventoryItem>
  <quantity>10</quantity>
  ...
<inventoryItem>
```

Validate UTF-8 against "inventoryItem"
Validate UTF-8 against "quantity"
Convert "1" "0" from UTF-8 to integer
Make sure 0<integer<10000
Copy integer to deserialized structure
What makes XML Screamer fast?

- **Optimizing across layers**
- **Avoid intermediate forms**
  - Don’t use SAX if you don’t need it
- **Avoid format conversions**
  - Work in input encoding wherever possible
- **Attention to detail**
- **In short: think like a compiler writer!**
How fast is XML Screamer?
Benchmark Reporting

- **Test environment**
  - Other machines checked for consistency – see paper

- **Results normalized to 1 GHz Pentium Processor**
  - Therefore: throughput of 3.2 Mbytes/sec would be reported as 1 MByte/sec/GHz
  - Scales well across Pentiums, Xeons, etc. of various clock speeds (Centrino is bit faster per cycle)
Test cases

- Report on 10 separate tests over 6 schemas
- All UTF-8 instances, single buffer, fits in memory.
- Sizes range from 990 bytes through 116.5KBytes.
Comparison to Non-validating Parsers

- Screamer is validating, and report SAX events
- Median Performance improvement: \textbf{1.9x Expat 3.8x Xerces}
Business Object Creation

- Xerces and Expat included for reference
- Median Performance improvement: \(2.9x\) Expat \(5.9x\) Xerces
  * Business Objects not supported for tests 6 & 7
Validation Comparison

- **Median Performance improvement:**

  5.5x for Sax, 11.6x for Business Objects

  *Business Objects not supported for tests 6 & 7*
XML Performance Summary

- XML can be parsed, validated, and deserialized into high performance API at median throughput of about 35MBytes/sec. on a 1 GHz Pentium.
  - That is 35% of the speed of raw character scan rate.

- On modern 4GHz processor, that’s 140MBytes/sec, or 14,000 10K Byte msgs/sec.
  - If you want to devote only 10% of CPU to parsing, you can still do 14 Mbytes/sec, or 1,400 10Kbyte messages per second
Conclusions

- Parsing, validation, and deserialization can run at speeds within 20-40% of raw character scan rate.
  - Probably close to the true limits of XML performance.
- Validation can mean a net gain in performance, if you have the option to compile in advance.
- XML Stack is designed in layers, but in implementations, layers disrupt performance.
  - This has implications beyond just parsing and validation.
- API Choice can make a significant difference in performance.
Related publications


END
## Backup 1: Performance Measurements

<table>
<thead>
<tr>
<th>Test case</th>
<th>Throughput (Mytes/Sec/ProcessorGHz)</th>
<th>XML Screamer</th>
<th>Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xerces - SAX</td>
<td>Expat</td>
<td>Validating</td>
</tr>
<tr>
<td>ID</td>
<td>Schema Filename</td>
<td>Size</td>
<td>Non val</td>
</tr>
<tr>
<td>1 po</td>
<td>990.00</td>
<td>4.41</td>
<td>2.65</td>
</tr>
<tr>
<td>2ipo</td>
<td>1,406.00</td>
<td>4.24</td>
<td>2.51</td>
</tr>
<tr>
<td>3 MI_AUS_RESPONSE2_1</td>
<td>1,572.00</td>
<td>3.21</td>
<td>2.98</td>
</tr>
<tr>
<td>4po</td>
<td>8,062.00</td>
<td>6.79</td>
<td>3.01</td>
</tr>
<tr>
<td>5ipo</td>
<td>8,077.00</td>
<td>6.08</td>
<td>2.90</td>
</tr>
<tr>
<td>6bibteXML</td>
<td>8,609.00</td>
<td>8.28</td>
<td>5.58</td>
</tr>
<tr>
<td>7 MI_AUS_REQUEST2_1</td>
<td>9,429.00</td>
<td>4.06</td>
<td>3.16</td>
</tr>
<tr>
<td>8po</td>
<td>63,754.00</td>
<td>6.88</td>
<td>3.02</td>
</tr>
<tr>
<td>9ipo</td>
<td>64,233.00</td>
<td>5.68</td>
<td>2.85</td>
</tr>
<tr>
<td>10 periodic_table</td>
<td>116,506.00</td>
<td>6.03</td>
<td>3.99</td>
</tr>
</tbody>
</table>
Backup 2: SAX Pre-computation

Precomputation of SAX Events

- **Screamer: No precomputation (anyType)**
- **Screamer: Events precomputed**

<table>
<thead>
<tr>
<th>Test ID</th>
<th>MB/Sec/GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>