X-FETCH PERFORMER republica

X-FETCH PERFORMER™

Benchmark Documentation

2.2

X-FETCH PERFORMER™ BENCHMARK DOCUMENTATION

General

ABSTRACT

As XML[5] is applied on new fields of data processing, the performance and stability of XML tools must be considered more carefully with IT decisions.

This document displays a performance comparison between the most common XML processing techniques according to the benchmark package published in IBM Developerworks in September 2001[1].

BACKGROUND

There are two different approaches to XML handling:

1. Event-based string parsing (for example SAX[1]-parsing),
2. Using object models (like DOM[2]).

Both approaches have their pros and cons but none of them is universally (in all respects) better than the other. Republica's contribution to XML-based e-business is the combining of the best features of these techniques in the EJB[3]-compatible X-FETCH PERFORMER™.

The following paragraphs tell more about the two conventional approaches.

[1] See http://www-106.ibm.com/developerworks/java/library/x-injava/

Event-based parsing

Event-based parsing means that the XML-reading component (called XML parser) constructs an event queue out of the input XML document. This queue is then interpreted by the application (ie. the component that needs the information appearing in the document). This approach is fast and does not consume memory: even the largest documents and data streams can be fluently processed.

The main cons in event-based parsing are:

Code complexity, which leads to losses in design and implementation resources.
XML document cannot be modified or new document cannot be generated.

Object models

To be able to generate or modify an XML document, one has to build an *object representation* of the document. This means that all compounds appearing in XML (e.g. elements, attributes, processing instructions) are stored in a data structure (usually tree-form), and modifications of that are (eventually) rendered as modifications in the original XML document.

The process of forming the object representation out of a given XML document is called “parsing” and the operation of producing XML string out of an object representation is called “serialization”.

Object models provide better access to data and tools for manipulating XML. However, object models consume memory and they cannot operate on data streams.

X-FETCH PERFORMER™ provides access to data via both techniques, with the additional features:

- XML Parsing and Generation
- XML Validation (DTD and Schema)
- XML Filtering and Content-based Routing (patent-pending technology)
- Efficient Data Queries (XPath[4])
- EJB Compatibility
- Built-in Interfaces for SAX and DOM
- User Manuals (containing also tutorials and examples with full Java source code)
- On-line Helpdesk Support (contact email: helpdesk@republica.fi)

Benchmark Report

All results are average values of 10 separate tests. The XML document used in tests was *periodic.xml* (see benchmark package in the Developerworks' homepage[2]). The effects of external processes were minimized by closing other applications before running the tests.

PLATFORM

Hardware	Intel Pentium III, 500MHz processor with 128MB RAM
Operating System	Windows 2000 Professional Microsoft Corporation
Java	Java version 1.3.1 Java HotSpot™ Client VM 1.3.1-b24 Sun Microsystems Inc.

ACRONYMS USED IN BENCHMARK GRAPHS

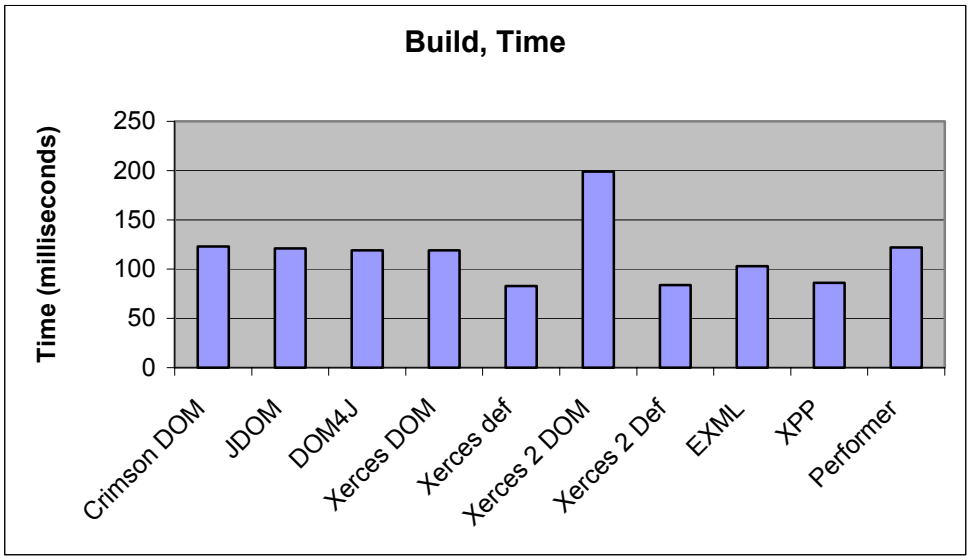
Crimson	Crimson DOM 1.1.1, see <http://xml.apache.org/crimson/index.html>
JDOM	JDOM β 0.7 (using Crimson for parsing), see <http://jdom.org/index.html>
Dom4j	Dom4j, see <http://dom4j.org/index.html>
Xerces DOM	Xerces 1.4.2 DOM, see <http://xml.apache.org/xerces-j/index.html>
Xerces Def	Xerces 1.4.2 DOM, Deferred Node Expansion, see <http://xml.apache.org/xerces-j/index.html>
Xerces 2 DOM	Xerces 2 DOM, see <http://xml.apache.org/xerces2-j/index.html>
Xerces 2 Def	Xerces 2 DOM, Deferred Node Expansion, see <http://xml.apache.org/xerces2-j/index.html>
EXML	Electric XML, see <http://www.themindelectric.com/exml/index.html>
XPP	XML Pull Parser, see <http://www.extreme.indiana.edu/xgws/xsoap/xpp/>
Performer	X-FETCH PERFORMER™ 2.2, see <http://www.x-fetch.com>

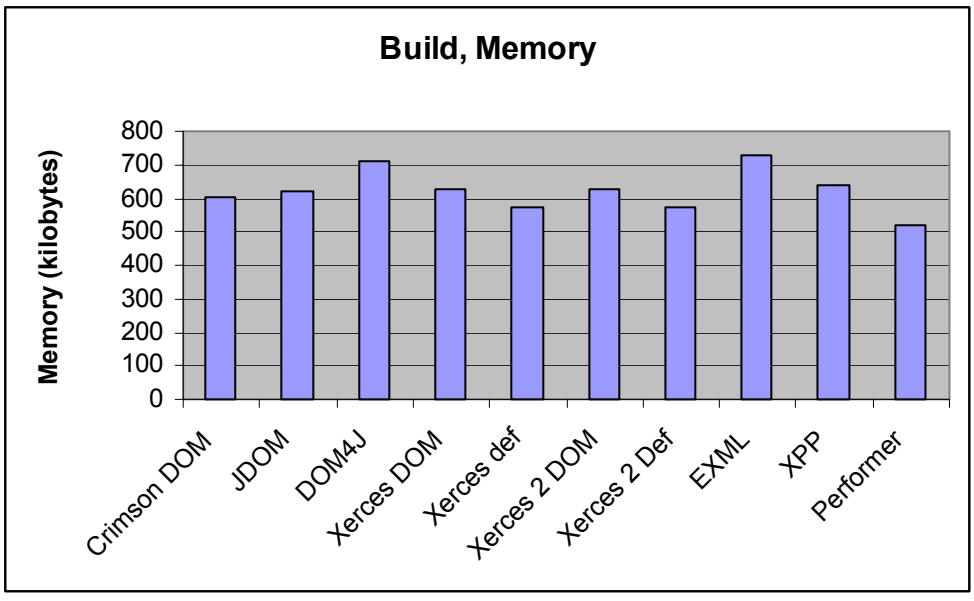
[2] See http://www-106.ibm.com/developerworks/java/library/x-injava/

BENCHMARK 1: BUILD AN OBJECT REPRESENTATION OUT OF XML DOCUMENT

In this test case, all products formed an object representation out of given XML file (*periodic.xml*). The average of 10 separate runs is used for the final comparison.

Both, elapsed time and memory consumption were measured.



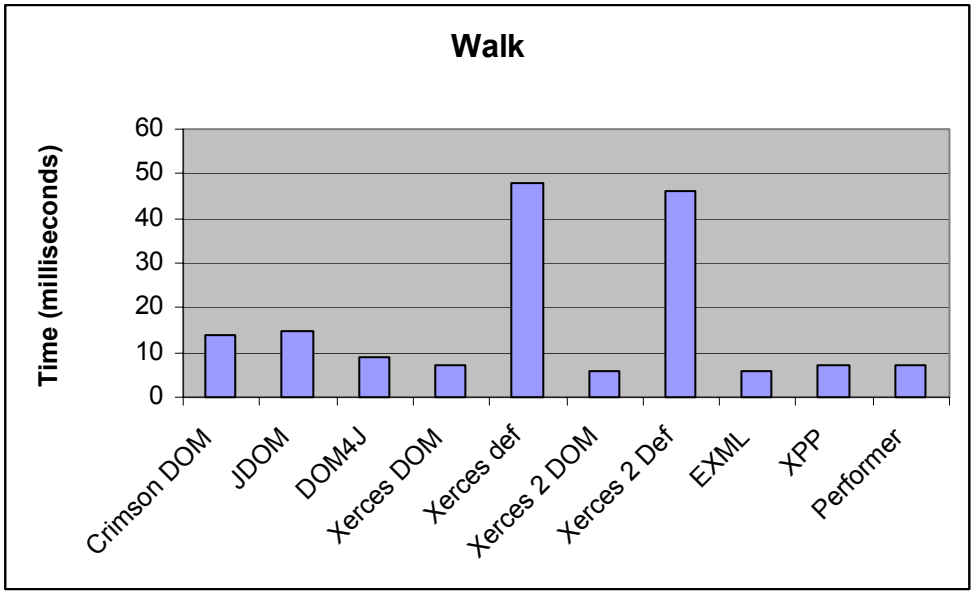


Build Results (sorted by time)	
Name	Time (milliseconds)
Xerces Def	83
Xerces 2 Def	84
XPP	86
EXML	103
DOM4J	119
Xerces DOM	119
JDOM	121
X-FETCH PERFORMER™	122
Crimson	123
Xerces 2 Dom	199

Build Results (sorted by memory consumption)	
Name	Memory (bytes)
X-FETCH PERFORMER™	517032
Xerces def	571544
Xerces 2 Def	571544
Crimson DOM	603256
JDOM	619880
Xerces DOM	627560
Xerces 2 DOM	627560
XPP	636304
DOM4J	708344
EXML	730696

BENCHMARK 2: WALK THROUGH AN OBJECT MODEL

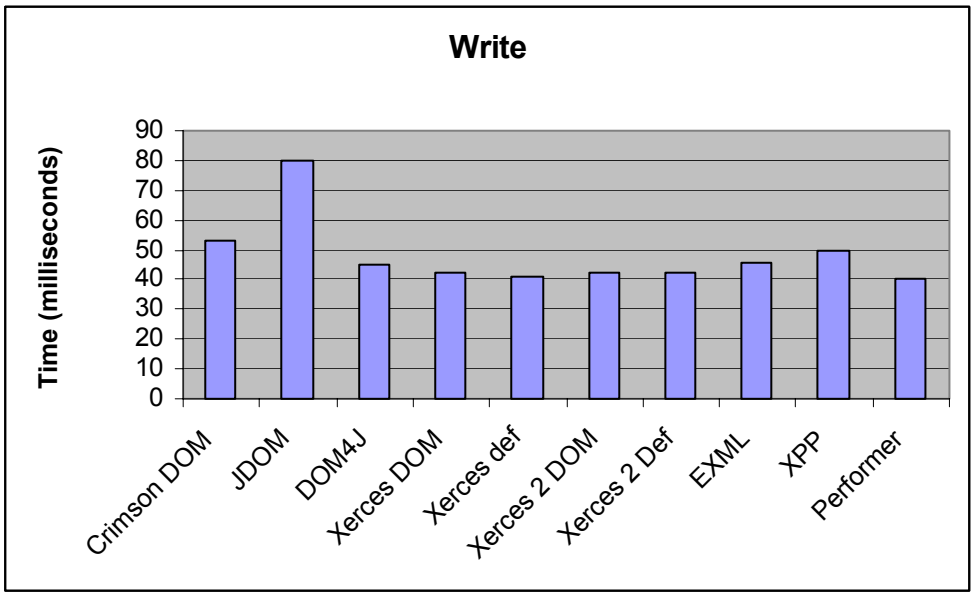
In this case, the parsed object model was traversed through. Again, the times are averages of 10 separate walks.



Walk Results (sorted by time)	
Name	Time (milliseconds)
Xerces 2 DOM	6
EXML	6
X-FETCH PERFORMER™	7
XPP	7
Xerces DOM	7
DOM4J	9
Crimson DOM	14
JDOM	15
Xerces 2 Def	46
Xerces Def	79

BENCHMARK 3: WRITE THE OBJECT MODEL BACK TO XML

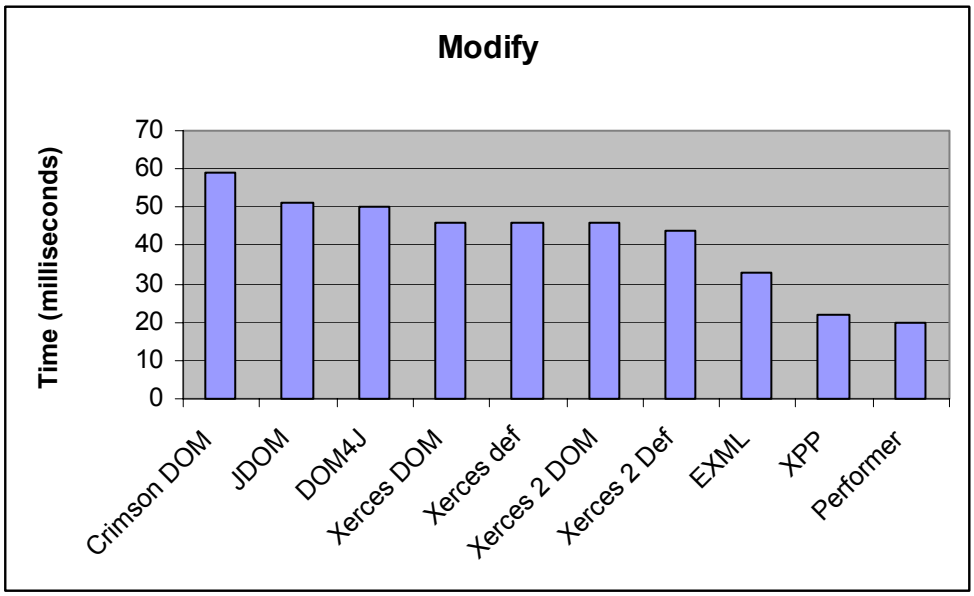
In this case, a once parsed object model was serialized back to XML string. Averages of 10 separate tests are displayed here.



Write (sorted by time)	
Name	Time (milliseconds)
X-FETCH PERFORMER™	40
Xerces def	41
Xerces DOM	42
Xerces 2 DOM	42
Xerces 2 Def	42
DOM4J	45
EXML	46
XPP	50
Crimson DOM	53
JDOM	80

BENCHMARK 4: MODIFY OBJECT MODEL

In this case, the object model was traversed through and modified. White-space characters were normalized and character data was wrapped into XML elements named **<text>**.

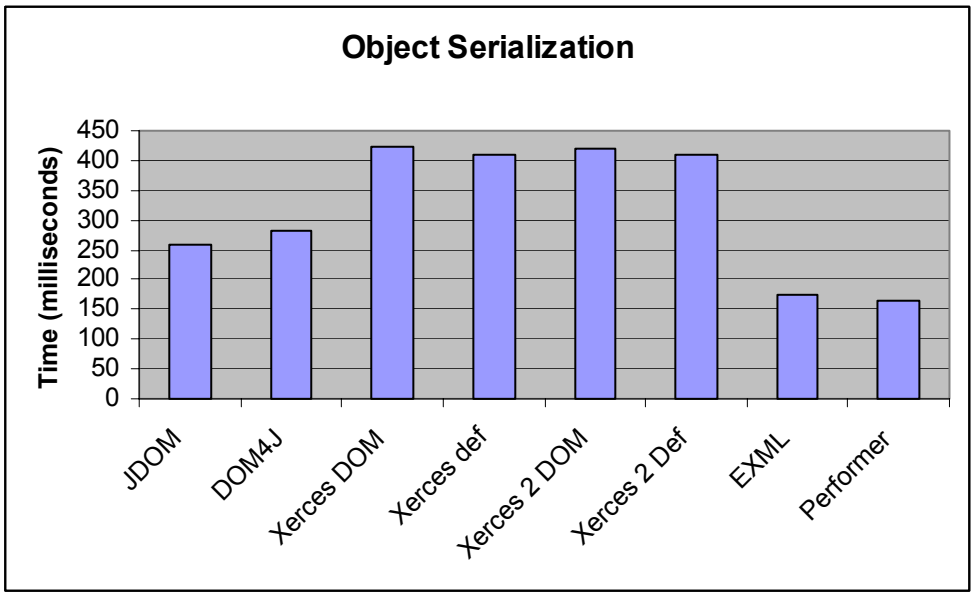


Modify (sorted by time)	
Name	Time (milliseconds)
X-FETCH PERFORMER™	20
XPP	22
EXML	33
Xerces 2 Def	44
Xerces DOM	46
Xerces def	46
Xerces 2 DOM	46
DOM4J	50
JDOM	51
Crimson DOM	59

BENCHMARK 5: JAVA OBJECT SERIALIZATION

Java language provides a way of writing an object into a stream (which can be directed to a file). This is often used in EJB-environments when computing time must be divided between processes and objects are "put-to-sleep" while waiting for other processes.

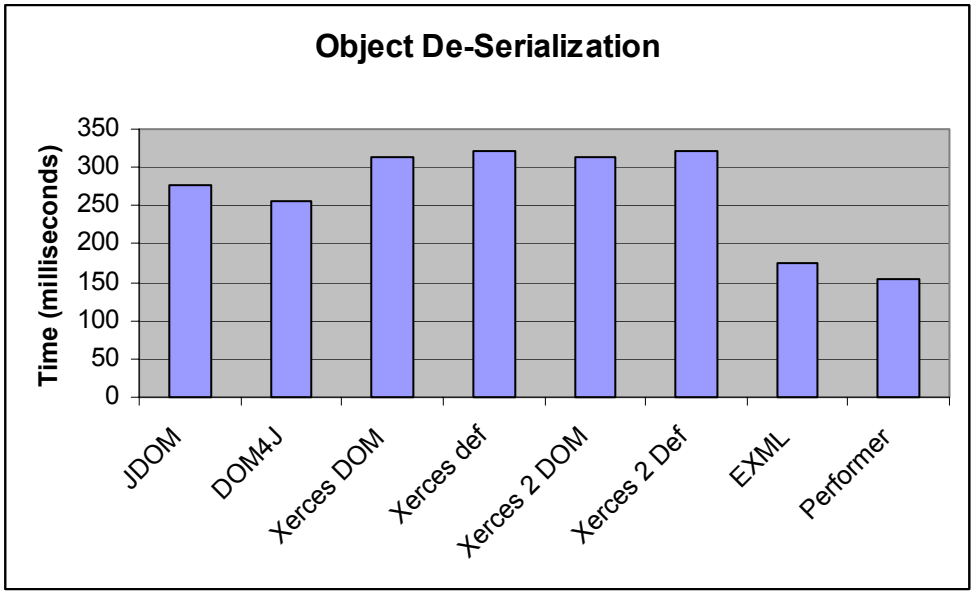
Note that only serializable document models can be used in EJB.



Object Serialization (sorted by time)	
Name	Time (milliseconds)
X-FETCH PERFORMER™	164
EXML	176
JDOM	258
DOM4J	282
Xerces def	410
Xerces 2 Def	410
Xerces 2 DOM	419
Xerces DOM	422

BENCHMARK 6: JAVA OBJECT DE-SERIALIZATION

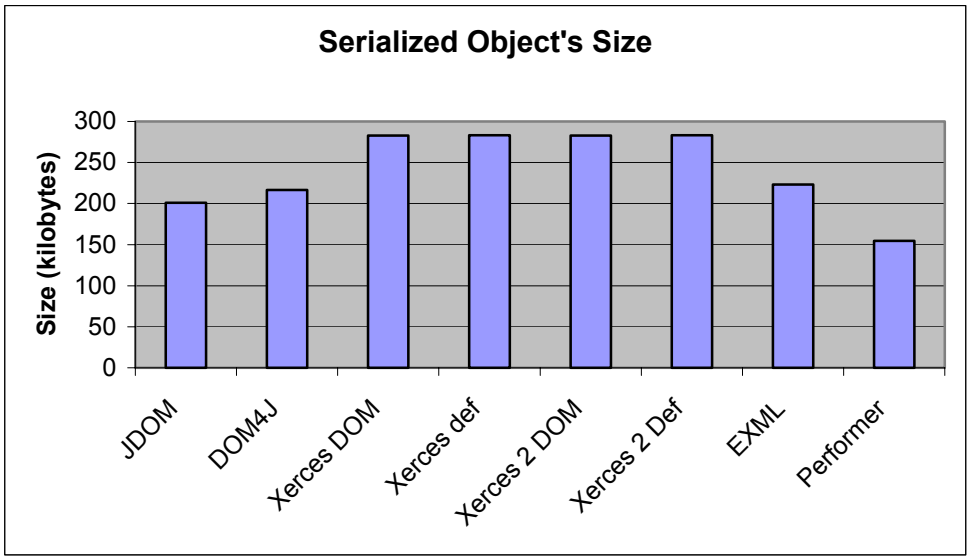
This test displays how fast a once serialized document object can be re-covered.



Object De-Serialization (sorted by time)	
Name	Time (milliseconds)
X-FETCH PERFORMER™	154
EXML	175
DOM4J	257
JDOM	277
Xerces DOM	313
Xerces 2 DOM	313
Xerces 2 Def	321
Xerces def	322

BENCHMARK 7: SERIALIZED OBJECT'S SIZE

This chart displays the size of the serialized Java object.

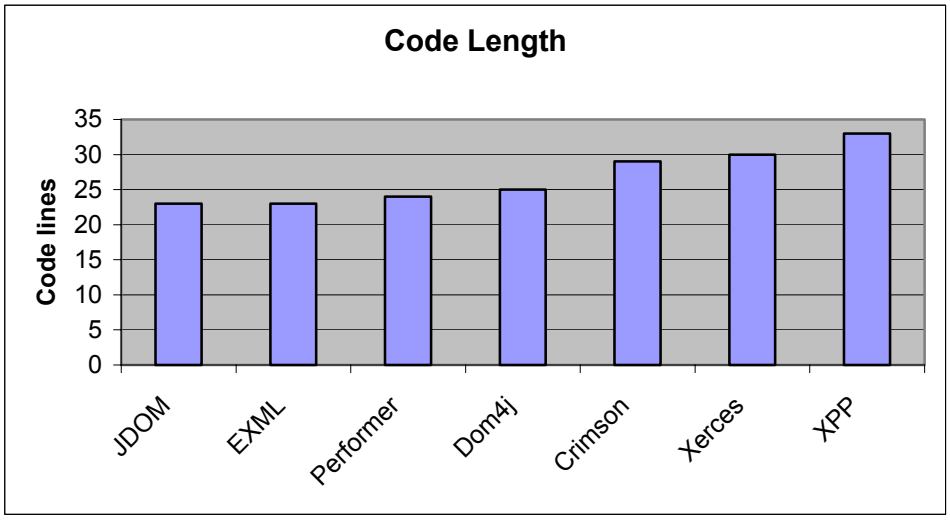


Serialized Object's Size (sorted by size)	
Name	Size (bytes)
X-FETCH PERFORMER™	154438
JDOM	200774
DOM4J	216682
EXML	223095
Xerces DOM	282897
Xerces 2 DOM	282897
Xerces def	283191
Xerces 2 Def	283191

BENCHMARK 8: CODE LENGTH COMPARISON

We compared the code lines used in benchmark implementations for parsing, modifying and serializing XML. As the graph shows, there were no significant differences between different object models.

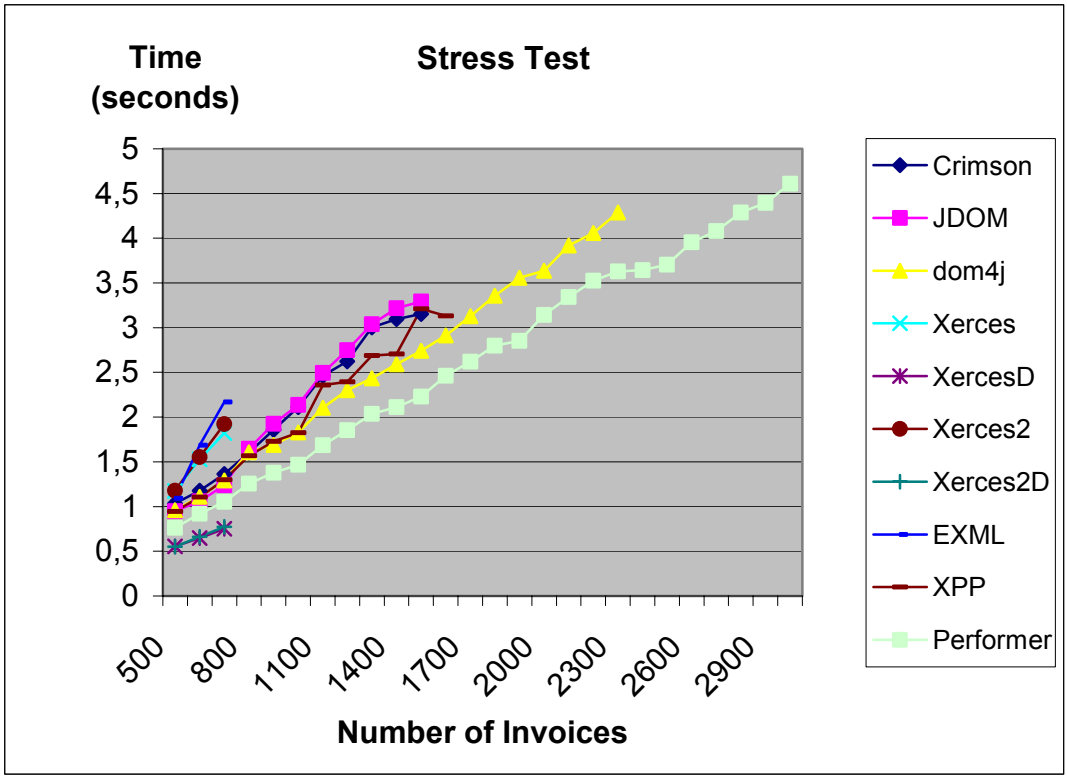
A corresponding implementation using some event-based parser (like SAX or XNI[6]) would consume hundreds of code lines.



Code length	
Name	Length (code lines)
JDOM	23
EXML	23
X-FETCH PERFORMER™	24
Dom4j	25
Crimson	29
Xerces	30
XPP	33

STRESS TESTS

Finally, we ran the tests with inputs of different sizes to see how the object models behave with large documents. In this case, the input material consisted of 500 up to 3000 invoices. X-FETCH PERFORMER™ was the only one capable of accomplishing the tests (other models ran out of memory).



Doubling the input up to 6000 invoices didn't affect on X-FETCH PERFORMER™'s performance: all material was processed in less than 10 seconds (on normal workstation) and memory curve stayed constant. This fact applies also on real world solutions: when the component is plugged into an existing XML invoicing system, all clients gain better and faster service.

This is why we've chosen X-FETCH PERFORMER™. By using X-FETCH PERFORMER™ in all XML projects, Republica builds platform independent XML software guaranteed with stability and optimal performance.

Conclusions

As we saw, X-FETCH PERFORMER™ finished first in most of the tests (in 7 out of 9). The reason for its superior performance is the XML filtering features applied in the MAP script of the X-FETCH PERFORMER™ benchmark: irrelevant information was filtered out at the building phase (which slowed down the build time but improved the manipulation and memory management).

Because of X-FETCH PERFORMER™'s ability of processing XML in small fragments, the stress test emphasizes its benefits compared to other techniques.

When comparing the code length, there were no significant differences between the object models. X-FETCH PERFORMER™ ranked 2nd with 24 code lines, just after the 23-lined JDOM and EXML implementations.

MORE INFORMATION

For more information about X-FETCH™ components and concepts please contact our sales department at sales@republica.fi

Republica Corp.
Elimäenkatu 12-16D, 6th Floor
FIN-00510 Helsinki
Finland

Tel. +358 (0)403 011 130
Fax. +358 (0)403 011 131
info@republica.fi
www.republica.fi
www.x-fetch.com

REFERENCES

[1] Simple API for XML, originally developed by David Megginson, see <http://www.saxproject.org>
[2] Document Object Model, W3C's specification, see <http://www.w3.org/DOM/>
[3] Enterprise Java Beans, see <http://java.sun.com/products/ejb/>
[4] XML Path Language, W3C's specification, see <http://www.w3.org/TR/xpath>
[5] Extensible Markup Language, W3C specification, see <http://www.w3.org/XML/>
[6] Xerces Native Interface, see <http://xml.apache.org/xerces2-j/xni.html>