Streaming APIs for XML Parsers

Java Web Services Performance Team
White Paper
August 2005

Table of Contents

Chapter 1

Introduction

Today, XML has emerged as a versatile and platform independent format for describing and delivering high-value solutions. Services using XML can be accessed from virtually any device, including cellular phones, PDAs, and desktops. Technologies like Web Services have been developed that can integrate existing business processes and resources and make them available over the Web by utilizing XML. To use this XML meaningfully in an application, it needs to be parsed and the relevant data extracted. There are a variety of ways to achieve this like Simple API for XML (SAX) and Document Object Model (DOM), but more recently a new breed of parsers based on pull-parsing techniques has emerged as the popular choice amongst developers.

This document describes the Sun Java Streaming XML Parser (SJSXP^{TM}) and some of its performance characteristics. SJSXP is an implementation of JSR-173. JSR-173 introduces new Streaming APIs for XML (StAX) which is a sandardized Java based API for pull-parsing XML. Pull parsing differs from the traditional SAX based iteration and DOM based tree model, in that it is optimized for speed and performance.

Chapter 2

StAX

From bootstrapping configuration files to deciphering large business documents received as messages, processing XML is ubiquitous. Most developers are familiar with two approaches for processing XML:

- Simple API for XML processing (SAX)
- Document Object Model (DOM)

SAX is a low-level API whose main benefit is efficiency. When parsing an XML document with SAX, events are generated and passed to the application using callbacks to handlers that implement the SAX handler APIs. The events are of a very low level, e.g. `startCDATA()` and `endCDATA()`. All of these low level events must be taken into account by the developer and it is also the developer's responsibility to maintain full document state information during the parse. SAX also requires the entire document to parsed.

DOM is a high-level parsing API whose main benefit is ease of use. DOM presents the application with an in-memory tree-like structure and provides for random-access. This simplicity comes at a high cost in performance penalties. DOM requires the entire document to be parsed and created as Objects before any part of the document can be processed or any actions taken.

Now imagine an API that has the efficiency of SAX yet is easy to use with higher level XML constructs. Enter the Streaming API for XML (StAX), a bi-directional API for reading and writing XML. It is formally specified by JSR 173, http://jcp.org/en/jsr/detail?id=173.

StAX is often referred to as "pull parsing." The developer uses a simple iterator based API to "pull" the next XML construct in the document. It is possible to skip ahead to areas of interest in the document, get only subsections of the document and arbitrarily stop or suspend processing at any time. This is different than SAX where the parser pushes the lower-level event at the application. With StAX, the application is in total control and drives the parser verses the parser driving the application.

StAX allows an application to process multiple XML sources simultaneously. For example: when one document includes or imports another document, the application can process the imported document while processing the original document. This use case is common when the application is reading documents such as XML Schemas or WSDL documents.

StAX has two basic functions: To allow users to read and write XML as efficiently as possible and be easy to use (cursor API), and be easy to extend and allow for easy pipelining (event iterator API). The event iterator API is intended to layer on top of the cursor API. The cursor API has two interfaces: `XMLStreamReader` and `XMLStreamWriter`. The event iterator API has two main interfaces: `XMLEventReader` and `XMLEventWriter`.

Events

Both APIs can be thought of as iterating over a set of events. In the cursor API the events may be unrealized; in the event iterator API the events are realized. An XML document is broken down into the following event granularity by both the cursor and event iterator API:

Interface javax.xml.stream.XMLStreamConstants

Field Summary	
`static int`	ATTRIBUTE Indicates an event is an attribute
`static int`	CDATA Indicates an event is a CDATA section
`static int`	CHARACTERS Indicates an event is characters
`static int`	COMMENT Indicates an event is a comment
`static int`	DTD Indicates an event is a DTD
`static int`	END_DOCUMENT Indicates an event is an end document
`static int`	END_ELEMENT Indicates an event is an end element
`static int`	ENTITY_DECLARATION Indicates a Entity Declaration
`static int`	ENTITY_REFERENCE Indicates an event is an entity reference
`static int`	NAMESPACE Indicates the event is a namespace declaration
`static int`	NOTATION_DECLARATION Indicates a Notation
`static int`	PROCESSING_INSTRUCTION Indicates an event is a processing instruction
`static int`	SPACE The characters are white space (see [XML], 2.10 "White Space Handling").
`static int`	START_DOCUMENT Indicates an event is a start document
`static int`	START_ELEMENT Indicates an event is a start element

Cursor API

The cursor API moves a virtual cursor across the underlying XML data and is the most efficient way to read XML data. A cursor can be thought of as an interface that moves over the underlying data and allows access to the underlying state through method calls. The cursor always moves forward. Events exist in the cursor (and event iterator) API as abstractions describing the XML Infoset. The cursor API is based on the "iterator" pattern:

- `hasNext()`
- `next()`

Interface javax.xml.stream.XMLStreamReader

The cursor model is supported by the `XMLStreamReader` interface:

Method Summary	
`void`	close`()` Frees any resources associated with this Reader.
`int`	getAttributeCount`()` Returns the count of attributes on this START_ELEMENT, this method is only valid on a START_ELEMENT or ATTRIBUTE.
`java.lang.String`	getAttributeLocalName`(int index)` Returns the localName of the attribute at the provided index
QName	getAttributeName`(int index)` Returns the qname of the attribute at the provided index
`java.lang.String`	getAttributeNamespace`(int index)` Returns the namespace of the attribute at the provided index
`java.lang.String`	getAttributePrefix`(int index)` Returns the prefix of this attribute at the provided index
`java.lang.String`	getAttributeType`(int index)` Returns the XML type of the attribute at the provided index
`java.lang.String`	getAttributeValue`(int index)` Returns the value of the attribute at the index
`java.lang.String`	getAttributeValue`(java.lang.String namespaceURI, java.lang.String localName)` Returns the normalized attribute value of the attribute with the namespace and localName If the namespaceURI is null the namespace is not checked for equality
`java.lang.String`	getCharacterEncodingScheme`()` Returns the character encoding declared on the xml declaration Returns null if none was declared
`java.lang.String`	getElementText`()` Reads the content of a text-only element, an exception is thrown if this is not a text-only element.
`java.lang.String`	getEncoding`()` Return input encoding if known or null if unknown.

`int`	getEventType`()` Returns an integer code that indicates the type of the event the cursor is pointing to.
`java.lang.String`	getLocalName`()` Returns the (local) name of the current event.
Location	getLocation`()` Return the current location of the processor.
QName	getName`()` Returns a QName for the current START_ELEMENT or END_ELEMENT event
NamespaceContext	getNamespaceContext`()` Returns a read only namespace context for the current position.
`int`	getNamespaceCount`()` Returns the count of namespaces declared on this START_ELEMENT or END_ELEMENT, this method is only valid on a START_ELEMENT, END_ELEMENT or NAMESPACE.
`java.lang.String`	getNamespacePrefix`(int index)` Returns the prefix for the namespace declared at the index.
`java.lang.String`	getNamespaceURI`()` If the current event is a START_ELEMENT or END_ELEMENT this method returns the URI of the prefix or the default namespace.
`java.lang.String`	getNamespaceURI`(int index)` Returns the uri for the namespace declared at the index.
`java.lang.String`	getNamespaceURI`(java.lang.String prefix)` Return the uri for the given prefix.
`java.lang.String`	getPIData`()` Get the data section of a processing instruction
`java.lang.String`	getPITarget`()` Get the target of a processing instruction
`java.lang.String`	getPrefix`()` Returns the prefix of the current event or null if the event does not have a prefix
`java.lang.Object`	getProperty`(java.lang.String name)` Get the value of a feature/property from the underlying implementation
`java.lang.String`	getText`()` Returns the current value of the parse event as a string, this returns the string value of a CHARACTERS event, returns the value of a COMMENT, the replacement value for an ENTITY_REFERENCE, the string value of a CDATA section, the string value for a SPACE event, or the String value of the internal subset of the DTD.

`char[]`	getTextCharacters`()` Returns an array which contains the characters from this event.
`int`	getTextCharacters`(int sourceStart, char[] target, int targetStart, int length)` Gets the the text associated with a CHARACTERS, SPACE or CDATA event.
`int`	getTextLength`()` Returns the length of the sequence of characters for this Text event within the text character array.
`int`	getTextStart`()` Returns the offset into the text character array where the first character (of this text event) is stored.
`java.lang.String`	getVersion`()` Get the xml version declared on the xml declaration Returns null if none was declared
`boolean`	hasName`()` returns true if the current event has a name (is a START_ELEMENT or END_ELEMENT) returns false otherwise
`boolean`	hasNext`()` Returns true if there are more parsing events and false if there are no more events.
`boolean`	hasText`()` Return true if the current event has text, false otherwise The following events have text: CHARACTERS,DTD ,ENTITY_REFERENCE, COMMENT, SPACE
`boolean`	isAttributeSpecified`(int index)` Returns a boolean which indicates if this attribute was created by default
`boolean`	isCharacters`()` Returns true if the cursor points to a character data event
`boolean`	isEndElement`()` Returns true if the cursor points to an end tag (otherwise false)
`boolean`	isStandalone`()` Get the standalone declaration from the xml declaration
`boolean`	isStartElement`()` Returns true if the cursor points to a start tag (otherwise false)
`boolean`	isWhiteSpace`()` Returns true if the cursor points to a character data event that consists of all whitespace

int	next() Get next parsing event - a processor may return all contiguous character data in a single chunk, or it may split it into several chunks.
int	nextTag() Skips any white space (isWhiteSpace() returns true), COMMENT, or PROCESSING_INSTRUCTION, until a START_ELEMENT or END_ELEMENT is reached.
void	require(int type, java.lang.String namespaceURI, java.lang.String localName) Test if the current event is of the given type and if the namespace and name match the current namespace and name of the current event.
boolean	standaloneSet() Checks if standalone was set in the document

Example:

```
// get a factory instance
XMLInputFactory myFactory = XMLInputFactory.newInstance();
// set error reporter (similar to setting ErrorReporter in SAX)
myFactory.setXMLReporter(myXMLReporter);
// set resolver (similar to setting EntityResolver in SAX)
myFactory.setXMLResolver(myXMLResolver);
// configure the factory, e.g. validating or non-validating
myFactory.setProperty(..., ...);
// create new XMLStreamReader
XMLStreamReader myReader = myFactory.createXMLStreamReader(...);
// document encoding from the XML declaration
String encoding = myReader.getEncoding();
// loop through document for XML constructs of interest
while(myReader.hasNext()) {
    int event = myReader.next();
    if (event == START_ELEMENT) {
        QName elementQName = myReader.getName();
       ...
    } else {
        ...
    }
}
```

Interface javax.xml.stream.XMLStreamWriter

The `XMLStreamWriter` interface specifies how to write XML. The XMLStreamWriter does not perform well formedness checking on its input. It does support Namespaces and even Namespace “correction” with the use of `java.xml.stream.isReparingNamespaces` on the `XMLOutputFactory`. The writing side of the API has methods that correspond to the reading side for event types:

Method Summary	
`void`	close`()` Close this writer and free any resources associated with the writer.
`void`	flush`()` Write any cached data to the underlying output mechanism.
NamespaceContext	getNamespaceContext`()` Returns the current namespace context.
`java.lang.String`	getPrefix`(java.lang.String uri)` Gets the prefix the uri is bound to
`java.lang.Object`	getProperty`(java.lang.String name)` Get the value of a feature/property from the underlying implementation
`void`	setDefaultNamespace`(java.lang.String uri)` Binds a URI to the default namespace This URI is bound in the scope of the current START_ELEMENT / END_ELEMENT pair.
`void`	setNamespaceContext`(`NamespaceContext` context)` Sets the current namespace context for prefix and uri bindings.
`void`	setPrefix`(java.lang.String prefix, java.lang.String uri)` Sets the prefix the uri is bound to.
`void`	writeAttribute`(java.lang.String localName, java.lang.String value)` Writes an attribute to the output stream without a prefix.
`void`	writeAttribute`(java.lang.String namespaceURI, java.lang.String localName, java.lang.String value)` Writes an attribute to the output stream
`void`	writeAttribute`(java.lang.String prefix, java.lang.String namespaceURI, java.lang.String localName, java.lang.String value)` Writes an attribute to the output stream
`void`	writeCData`(java.lang.String data)` Writes a CData section
`void`	writeCharacters`(char[] text, int start, int len)` Write text to the output

`void`	writeCharacters`(java.lang.String text)` Write text to the output
`void`	writeComment`(java.lang.String data)` Writes an xml comment with the data enclosed
`void`	writeDefaultNamespace`(java.lang.String namespaceURI)` Writes the default namespace to the stream
`void`	writeDTD`(java.lang.String dtd)` Write a DTD section.
`void`	writeEmptyElement`(java.lang.String localName)` Writes an empty element tag to the output
`void`	writeEmptyElement`(java.lang.String namespaceURI, java.lang.String localName)` Writes an empty element tag to the output
`void`	writeEmptyElement`(java.lang.String prefix, java.lang.String localName, java.lang.String namespaceURI)` Writes an empty element tag to the output
`void`	writeEndDocument`()` Closes any start tags and writes corresponding end tags.
`void`	writeEndElement`()` Writes an end tag to the output relying on the internal state of the writer to determine the prefix and local name of the event.
`void`	writeEntityRef`(java.lang.String name)` Writes an entity reference
`void`	writeNamespace`(java.lang.String prefix, java.lang.String namespaceURI)` Writes a namespace to the output stream If the prefix argument to this method is the empty string, "xmlns", or null this method will delegate to writeDefaultNamespace
`void`	writeProcessingInstruction`(java.lang.String target)` Writes a processing instruction
`void`	writeProcessingInstruction`(java.lang.String target, java.lang.String data)` Writes a processing instruction
`void`	writeStartDocument`()` Write the XML Declaration.
`void`	writeStartDocument`(java.lang.String version)` Write the XML Declaration.

`void`	writeStartDocument`(java.lang.String encoding, java.lang.String version)` Write the XML Declaration.
`void`	writeStartElement`(java.lang.String localName)` Writes a start tag to the output.
`void`	writeStartElement`(java.lang.String namespaceURI, java.lang.String localName)` Writes a start tag to the output
`void`	writeStartElement`(java.lang.String prefix, java.lang.String localName, java.lang.String namespaceURI)` Writes a start tag to the output

Example:

```
// Write the XML Declaration
myWriter.writeStartDocument("ISO-8859-1", "1.0");
// Writes a start tag to the output
myWriter.writeStartElement(“hello”);
// Write text to the output
myWriter.writeCharacters(“world”);
// Writes an end tag to the output relying on the internal state of the writer
// to determine the prefix and local name of the event
myWriter.writeEndElement();
// Closes any start tags and writes corresponding end tags
myWriter.writeEndDocument();
// Write any cached data to the underlying output mechanism
myWriter.flush();
// Close this writer and free any resources associated with the writer
myWriter.close();
```

Event Iterator API

The event iterator API introduces objects representing the events that one can probe for in cursor API. Events exist in the event iterator (and cursor) API as abstractions describing the XML Infoset. The event iterator API has an interface that is very easy to implement and use. The `nextEvent()` method returns an object that is immutable, can be cached or passed on to another component in the chain of processing.

The event iterator API is based on the “iterator” pattern:

- `hasNext()`
- `next()`
- `peek()` (ability to “peek” into the next event)

Interface javax.xml.stream.XMLEventReader

This is the top level interface for parsing XML Events. It provides the ability to peek at the next event and returns configuration information through the property interface.

Method Summary	
`void`	close`()` Frees any resources associated with this Reader.
`java.lang.String`	getElementText`()` Reads the content of a text-only element.
`java.lang.Object`	getProperty`(java.lang.String name)` Get the value of a feature/property from the underlying implementation
`boolean`	hasNext`()` Check if there are more events.
XMLEvent	nextEvent`()` Get the next XMLEvent
XMLEvent	nextTag`()` Skips any insignificant space events until a START_ELEMENT or END_ELEMENT is reached.
XMLEvent	peek`()` Check the next XMLEvent without reading it from the stream.

Example:

```
XMLInputFactory myFactory = XMLInputFactory.newInstance();
FileInputStream myFileInputStream = new FileInputStream(myFileName);
XMLEventReader myReader = myFactory.createXMLEventReader(myFileInputStream);
while(myReader.hasNext()) {
    XMLEvent myEvent = myReader.nextEvent();
    if (myEvent.isStartElement()) {
        ...
    } else {
        ...
    }
}
```

Interface javax.xml.stream.XMLEventWriter

The output side of the event iterator API is `XMLEventWriter`. This is the top level interface for writing XML documents. Instances of this interface are not required to validate the well formedness of the XML:

Method Summary	
`void`	add`(`XMLEvent` event)` Add an event to the output stream Adding a START_ELEMENT will open a new namespace scope that will be closed when the corresponding END_ELEMENT is written.

`void`	add(XMLEventReader reader) Adds an entire stream to an output stream, calls next() on the inputStream argument until hasNext() returns false This should be treated as a convenience method that will perform the following loop over all the events in an event reader and call add on each event.
`void`	close() Frees any resources associated with this stream
`void`	flush() Writes any cached events to the underlying output mechanism
NamespaceContext	getNamespaceContext() Returns the current namespace context.
`java.lang.String`	getPrefix(java.lang.String uri) Gets the prefix the uri is bound to
`void`	setDefaultNamespace(java.lang.String uri) Binds a URI to the default namespace This URI is bound in the scope of the current START_ELEMENT / END_ELEMENT pair.
`void`	setNamespaceContext(NamespaceContext context) Sets the current namespace context for prefix and uri bindings.
`void`	setPrefix(java.lang.String prefix, java.lang.String uri) Sets the prefix the uri is bound to.

Filters

It is possible to filter the input of both the Cursor and Event models. This is very efficient. Filters cannot modify the reader state, they can only skip events. For example, if an application only wants to see `START_ELEMENT` and `END_ELEMENT` events:

```
public class FilterImpl
    implements StreamFilter {
    public boolean accept(XMLStreamReader myReader) {
        if (myReader.isStartElement()
            || myReader.isEndElement()) {
            return true;
        } else {
            return false;
        }
    }
}
```

Resource Resolution

The `XMLResolver` interface provides a way to set the method that resolves resources during the processing of XML contents. The application sets the interface on the `XMLInputFactory`, which subsequently sets the interface on all processors that the instance of the factory creates.

Error Reporting and Exception handling

All fatal errors are reported as `javax.xml.stream.XMLStreamExceptions`. Nonfatal errors and warnings are reported using the `javax.xml.stream.XMLReporter` interface. The `Location` interface provides line/column/character offset information.

Chapter 3

Java Web Services Developer Pack

The Java^{TM} Web Services Developer Pack ("Java WSDP") is a free integrated toolkit you can use to build, test and deploy XML applications, Web services, and Web applications with the latest Web service technologies and standards implementations.
It provides developer choice and flexibility by supporting the Sun Java^{TM} System Application Server Platform Edition 8, the Sun Java System Web Server 6.1, and Tomcat 5.0 for Java WSDP 1.6 Web containers for Web services development.
With the newest release of the Java WSDP 1.6, developers will be able to:

- Develop and deploy using the latest XML and Web services technologies slated for inclusion into Sun's deployment platforms.
- Create XML and Web service-enabled applications that are secure, interoperable, and portable across different platforms and devices.
- Simplify and lower the cost of legacy application integration, data interchange, and publishing in a Web environment.

The Java WSDP 1.6 includes an EA implementation of Sun Java Streaming XML Parser Version 1.0, a high performance implementation of StAX, the Streaming API for XML. As described earlier, StAX is the standard Java based API for pull-parsing XML, which complements the existing SAX and DOM parsing models by allowing the programmer to explicitly ask for next events.

Sun Java Streaming XML Parser (SJSXP)

JSR-173 introduces new Streaming APIs for XML (StAX) which is a Java based API for pull-parsing XML. Sun Java Streaming XML Parser (SJSXP) is the Implementation of StAX and is extremely fast. SJSXP is a non-validating, XML 1.0 and Namespace 1.0 compliant XML parser. SJSXP has been written using the lower layer of Xerces2 responsible for reading and applying well formed rules of XML document. This layer responsible for reading various sections of XML document like `ELEMENT`, `ATTRIBUTES`, `CHARACTERS`, `PI`, `COMMENT` etc. is designed as per push model where events are pushed as they are encountered. However, in a pull model parser stops after parsing the next event on the input stream and control comes back to application. There has been some attempts where pull layer is built on top of push layer but this design is not efficient. It required buffering of events so that one event can be pulled at a time.

In SJSXP this problem was tackled in a different way and lower layers were re-designed and largely modified to behave in pull fashion. Internal state machine has been changed so that parser stops after parsing each event and has the capability to revive itself when instructed by the application to read next XML event. This design change is important because it is easy to give control than have it. A push layer (SAX) can be neatly built on top of pull layer without sacrificing performance.
While re-designing lot of other changes were done to make XML parsing more efficient and lot of other optimizations went into the code base. Result of these optimizations is visible in the benchmarks. Wherever possible, the parser maintains a snapshot of the underlying buffer without the extra overhead of copying data into buffer maintained by the parser. SJSXP also exploits the advantages given by the pull model. StAX Cursor model represents the state of the parser by an integer constant and various accessor functions to retrieve information related to that state. This design gives implementation vendors a chance to create objects lazily. SJSXP exploits this feature very well. Objects are not computed

unless requested by the application.
Compliance to the XML standards is of utmost importance, while optimizing XML parser, it has been taken care that compliance to the standards is maintained. SJSXP confirms to above mentioned XML standards. For example, it has come to notice that certain pull parsers doesn't check if each character in the XML document is a valid XML character. This is serious and will result in some of the non-XML files and fatal errors to escape through the parser.

SJSXP RoadMap

Going forward, XML 1.1 and DTD Validation support will be added to SJSXP. As StAX becomes part of Java platform in Java^{TM} SE 6.0, SJSXP will become the default implementation of StAX in Sun's JDK. As said earlier, push layer can be neatly built on top of pull layer. In JAXP 1.4, SJSXP and JAXP implementation are merged such that lower layers of JAXP implementation are replaced with SJSXP. Now in JAXP 1.4 there is a pull layer which sits at the bottom and a push layer is built on top of pull layer. From JAXP 1.4, single library will support 3 different XML parsing models viz. StAX (Pull Model), SAX (Push Model) and DOM (Object Model).

Chapter 4

XMLTest

What is XMLTest?

XMLTest [8] is an XML processing test developed at Sun Microsystems and released to the public in early 2004. Since then it has been adapted and used by other vendors to gauge XML performance. XMLTestis designed to mimic the processing that takes place in the lifecycle of an XML document. XMLTest simulates a multi-threaded server program that processes multiple XML documents in parallel. This is very similar to an application server that deploys web services and concurrently processes a number of XML documents that arrive in client requests. Since we wanted to concentrate on XML processing performance, rather than use some sort of web container, we designed a standalone multi-threaded program implemented in Java. To avoid the effect of file I/O, the documents are read from and written to memory streams.

XML Test measures the throughput of a system processing XML documents. For streaming parsers it just involves parsing through each document without any writing or serialization.   XML Test reports one metric: Throughput - Average number of XML transactions executed per second. It can be configured using the following parameters:

- Number of threads - This is tuned to maximize CPU utilization and system throughput.
- PullParserFactory - Implementation of parser used to parse through the document.
- StreamUsage – Whether stream parsers are being tested.
- RampUp - Time allotted for warm-up of the system.
- SteadyState - The interval when transaction throughput is measured.
- RampDown - Time allotted for ramp down, completing transactions in flight.
- XmlFiles - The actual XML documents used by XML Test.

XML Test reads these properties at initialization into an in-memory structure that is then accessed by each thread to initiate a transaction as per the defined mix. To keep things as simple as possible, XML Test is a single-tier system where the test driver that instantiates an XML transaction is part of each worker thread. A new transaction is started as soon as a prior transaction is completed (there is no think time). The number of transactions executed during the steady state period is measured. The throughput, transactions per second (TPS) is calculated by dividing the total number of transactions by the steady state duration. The average response time for each transaction is also calculated.

Running XMLTest

Different XML documents have different characteristics that are particular to the context in which they are used. For example some documents may have a high number of elements and others may have a higher number of attributes etc. The XMLTest runs performed try to take this characteristic into account. Since different XML document were used with differing number of elements, element names, number of attributes and number of nested elements, no access was performed by looking for specific element names or attributes since there were no common element names and attributes among all of the XML documents. Each document was parsed by instantiating an `XMLStreamReader`, and iterating through each `eventType`. The cursor APIs defined in JSR-173 are used for StAX parsing in XMLTest. For the case of XPP3 parser, the XMLPullParser class was instantiated instead of the `XMLStreamReader` to get the eventType. Each run was completed with two agents (two threads) to fully saturate the server's dual CPUs and the TPS was averaged over 10 iterations for each document.   The only runtime argument used was "`-server`".

Hardware and Software Configuration

Run Characteristics	Software	Machine Details
Ramp Up: 300sec	***JWSDP1.5***	***SunFire 280r machine***
Steady State: 600 sec	***JDK1.5***	***2 USIII+ 1015MHz***
Ramp Down: 30 sec	***Solaris 9***	***4096MB RAM***

Table 1: Hardware, Software, Benchmark configuration details

Other StAX Implementations

Implementation	JAR File / Release	Location
BEA	***wls_stax.jar***	***http://dev2dev.bea.com/technologies/stax/index.jsp***
Oracle	***xmlpull.jar, xmlparserv2.jar***	***http://www.oracle.com/technology/tech/xml/xdk/staxpreview.html***
RI	***stax-1.1-dev.jar***	***http://stax.codehaus.org/***
XPP3	***xpp3-1.1.3.4.M.jar***	***http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/index.html***
Woodstox	***wstx.jar***	***http://www.cowtowncoder.com/proj/woodstox/index.html***

Table 2. Other JSR-173 parsers

Performance Results

The performance of the five parsers was measured over a large number of XML documents. The results are presented in the following figures categorized on the basis of size.   Figures 1 and Figure 2 shows the parser results in documents sized from 500 KB to 4 MB, Figure 3 and 4 from 50-100K, and Figure 5 and 6 from 5K to 10K.

In Figure1, SJSXP (stippled bar) is shown to be faster than the BEA, Oracle, and RI implementations of StAX parser. SJSXP is as fast as XPP3 and Woodstox parsers except in the case of saml-500k and inv500k.xml. Closer inspection of these documents' structure does not reveal any factor that explains the difference in parser performance (see Table 3 for document characteristics).

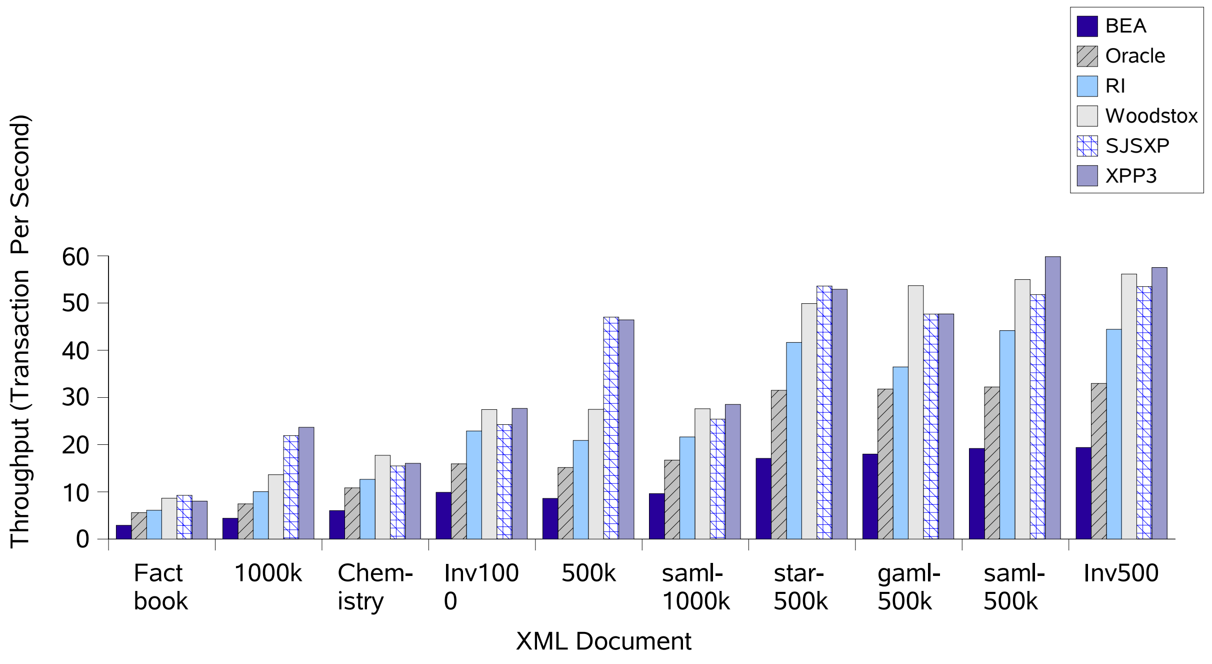

Figure 1. Throughput for parsers with documents sized from 500K to 4 MB

Document	Size (KB)	# Elements	# Attributes	Depth	# Unique elements	Avg Att/Element
factbook.xml	4140	55453	0	6	199	0.00
1000k.xml	996	14369	13737	6	51	0.96
Chemistry	1830	13082	35266	8	17	2.70
inv1000.xml	903	15075	14059	6	51	0.93
500k.xml	498	7364	6863	6	51	0.93
saml-1000k.xml	1017	13523	17486	6	8	1.29
star-500k.xml	490	11503	6	7	37	0.00
gaml-500k.xml	603	4222	11516	8	17	2.73

Table 3: Characteristics of XML Documents of 'Large' size

Figure 2 displays the ratio of the throughput of SJSXP divided by the throughput of the competitor parser. A throughput greater than 1 indicates a faster performance of SJSXP relative to the competitor, while a ratio of less than 1 indicates a slower one. For example, for the last document in the chart, inv500.xml, SJSXP is roughly 2.5X faster than BEA, but 0.93X that of XPP3. The XPP3 ratio is shown above the bars to help distinguish values close to 1 because of the small scale. To aid in determining values less than 1, the ratio is shown for the XPP3 parser.

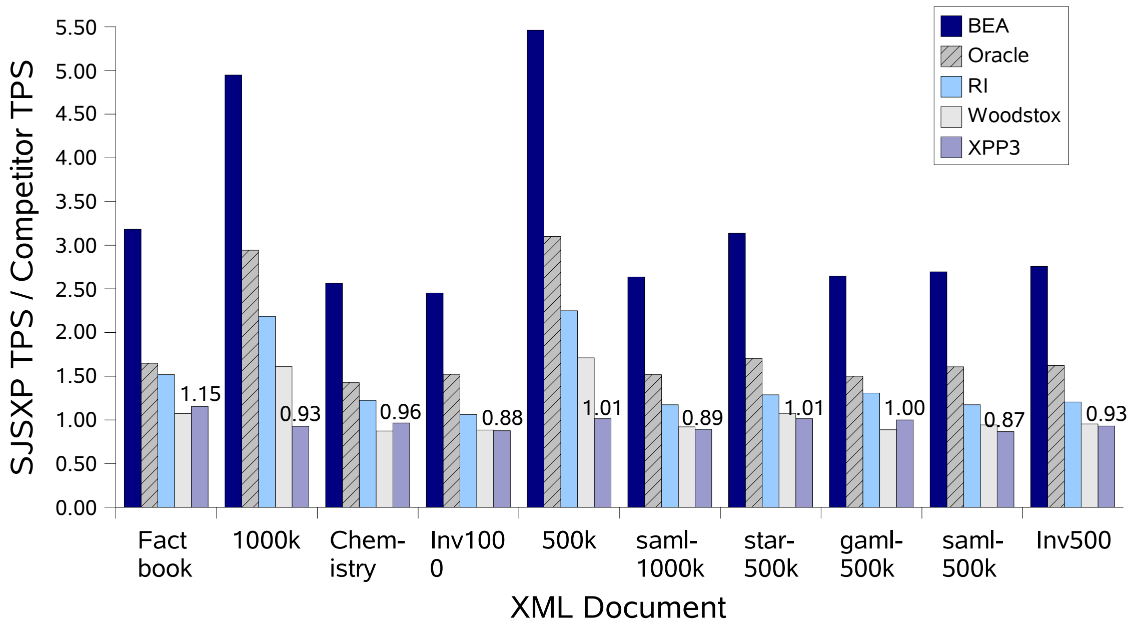

Figure 2: Ratio Bar Chart for 'Large' Document sizes

When testing documents of 'medium' size (50K to 100K) in Figure 3, SJSXP (stippled bar) continues to outperform BEA, Oracle and the Reference Implementations but again is behind XPP3 and Woodstox for certain documents (inv50.xml, saml-50k.xml, inv100.xml, periodic.xml). Table 4 reveals these documents' characteristics. Again, there seems to be no clear differentiator that explains the optimum class of document on which each parser performs optimally.

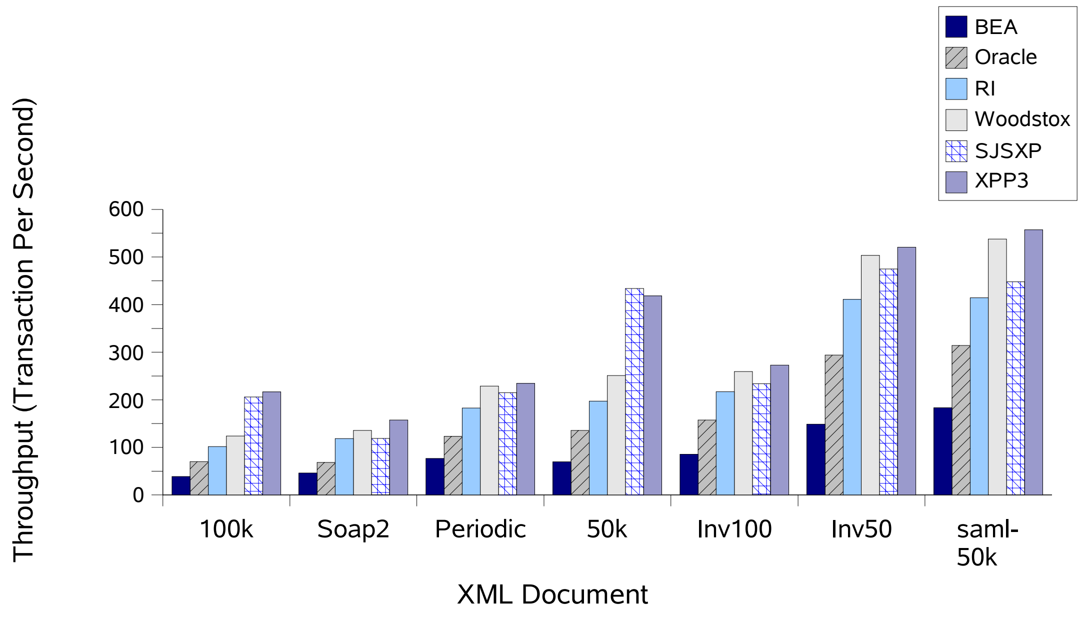

Figure 3. Throughput for parsers with document sized from 50K to 100K.

Document	Size (KB)	# Elements	# Attributes	Depth	# Unique elements	Avg Att/Element
100k.xml	101	1484	1375	6	51	0.93
soap2.xml	132	4501	999	9	10	0.22
50k.xml	51	749	689	6	51	0.92
inv100.xml	95	1575	1459	6	51	0.93
inv50.xml	50	825	759	6	51	0.92
saml-50k.xml	52	679	880	6	8	1.30

Table 4. Characteristics of XML Documents of 'Medium' Size

Figure 4 shows the ratio of the relative performance of SJSXP as compared to the competitors. The magnitude by which SJSXP outperforms BEA, Oracle and RI is easily visible here. However, it can be seen that SJSXP can lag behind as much as 0.75x compared to Woodstox and XPP3 for a given document size (soap2.xml). In Figure 4, a value greater than 1 signifies faster relative performance of SJSXP relative to the competitor, while value less than 1 signifies slower parser performance. To aid in determining values less than 1, the ratio is shown for the XPP3 parser

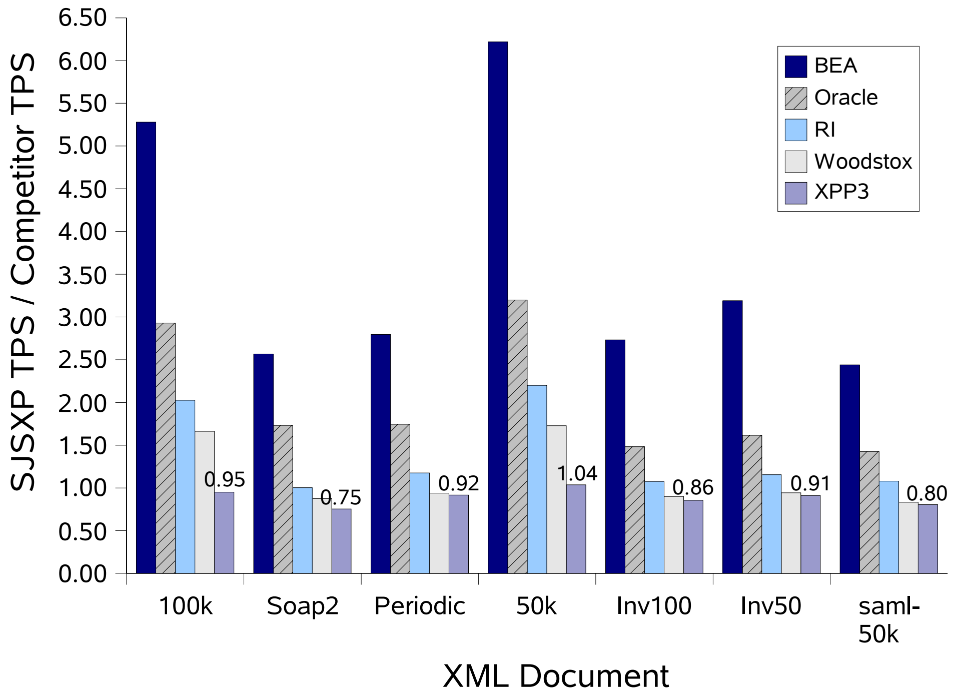

Figure 4 : Ratio Bar Chart for 'Medium' Document sizes

Finally, in the 'small' category of documents sized 5K-12K, Figure 5 shows the relative performance. SJSXP (stippled bar) again outperforms BEA, Oracle and RI. Woodstox and XPP3 parsers are faster in certain documents compared to SJSXP. Figure 6 shows the magnitude of the relative performance of the SJSXP compared to the competitors. SJSXP performance is slighly slower than woodstox implementation in this class of documents so the ratio of woodstox is shown above the bar in this figure. Table 5 reveals further document characteristics of the documents of the 'small' size but fails to identify the factor that explains the difference in XML parser performance.

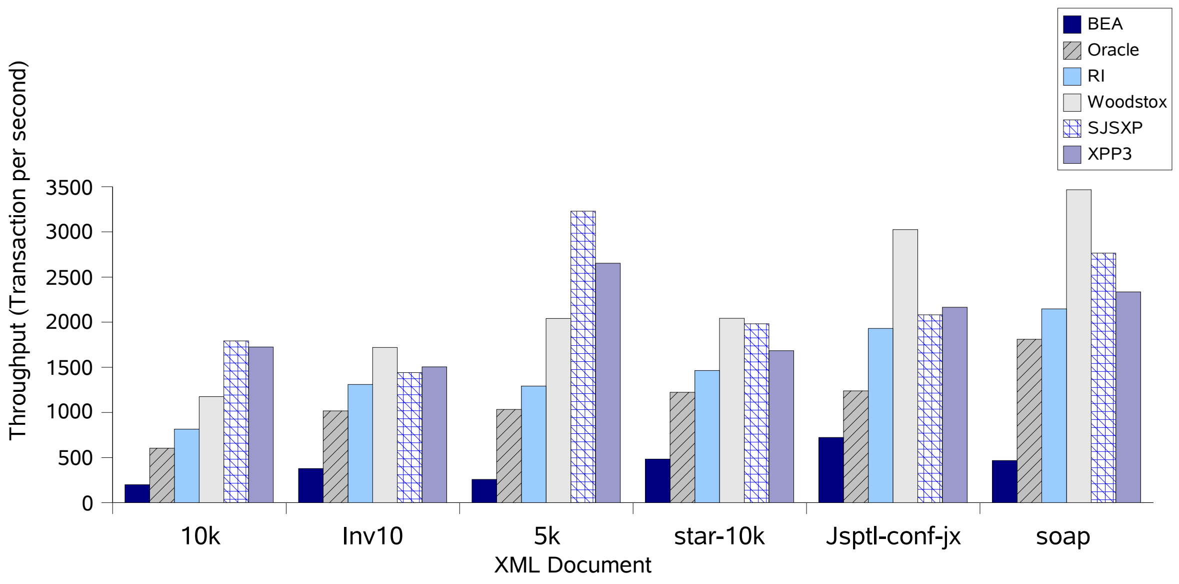

Figure 5. Throughput for parsers with documents sized from 5 to 12K.

Document	Size (KB)	# Elements	# Attributes	Depth	# Unique elements	Avg Att/Element
10k.xml	11	149	129	6	51	0.87
inv10.xml	14	225	199	9	51	0.88
5k.xml	6	89	73	6	51	0.82
star-10k.xml	12	255	6	7	37	0.02
jsptl-conf-jx.xml	11	218	0	6	21	0.00
soap.xml	7	92	73	9	54	0.79

Table 5. Characteristics of XML Documents of 'Small' Size.

In Figure 6, a value greater than 1 signifies faster relative performance of SJSXP relative to the competitor, while value less than 1 signifies slower parser performance. This time, the ratio is shown for Woodstox parser to aid in determining values less than 1

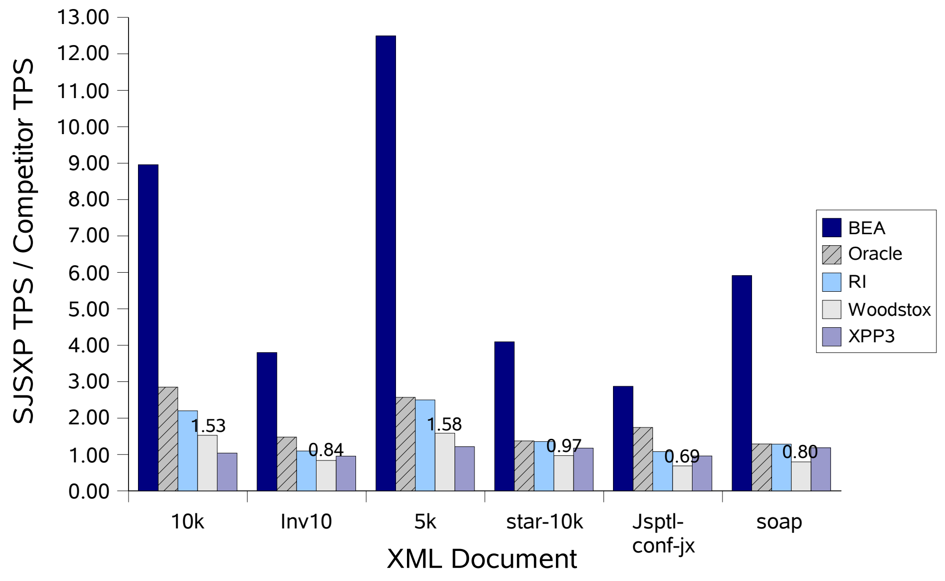

Figure 6. Ratio Bar Chart for 'Small' Document size

Chapter 5

Summary

SJSXP performance is consistently faster than BEA, Oracle and RI for all of the documents described here in this study. However, it lags behind Woodstox and XPP3 in some document sizes and in the best cases, exhibits similar performance compared to these two parsers. The total number of elements, attributes, and the maximum depth of nested elements, the number of unique elements and average number of attributes were also calculated for each XML document but there seemed to be no clear correlation between any of these characteristics and difference in relative parser performance.
It is worthy to note that XPP3 is based on XmlPullParser APIs and not JSR-173 compliant. XPP3 is a parsing API that will work with small devices (J2ME compatible). XmlPull defines only one interface to represent XML pull parser with one exception. It has a very small memory footprint and can be used as a building block for higher level APIs. It is designed to be a small lightweight parser for fast performance.

SJSXP on the other hand is a fully JSR-173 compliant streaming parser that has symmetrical bi-directional APIs that can both read and write XML documents using the same representation of XML. Its performance is only slightly behind that of a parser designed for speed even though it has more functionality. Specifically, it has two main styles of interfaces which aims to serve two basic functions: to allow users to read and write XML as efficiently as possible (cursor API) and to be easy to use, event based, easy to extend and allow easy pipelining (event iterator API). The cursor APIs for XML has two interfaces: XMLStreamReader and XMLStreamWriter while the event iterator APIs has two main interfaces: XMLEventReader and XMLEventWriter.

SJSXP introduces a new Streaming APIs for XML (StAX) which is a standardized Java based API for pull-parsing XML. StAX has two basic functions: to allow users to read and write XML as efficiently as possible and be easy to use (cursor API), and be easy to extend and allow for easy pipelining (event iterator API). Pull parsing differs from the traditional SAX based iteration and DOM based tree model, in that it is optimized for speed and performance. StAX is often referred to as “pull parsing.” The developer uses a simple iterator based API to “pull” the next XML construct in the document. The Java WSDP 1.6 includes an EA implementation of Sun Java Streaming XML Parser Version 1.0 but will become a part of Java platform in Java SE 6.0.

Readers are welcome to disucss and send their feedback about this paper on the Java Web Services performance community[11] online at http://performance.dev.java.net/

References

[1] Java WSDP
To access this resource online, go to http://java.sun.com/
[2] JSR 173: Streaming API for XML
To access this resource online, go to http://www.jcp.org/en/jsr/detail?id=173
[3] Reference implementation of the StAX API
To access this resource online, go to http://dev2dev.bea.com/technologies/stax/index.jsp
[4] Oracle StAX Pull Parser Preview
To access this resource online, go to http://www.oracle.com/technology/tech/xml/xdk/staxpreview.html
[5] Streaming API for XML (StAX) Reference Implementation homepage
To access this resource online, go to http://stax.codehaus.org/
[6] Xml Pull Parser 3rd Edition (XPP3)
To access this resource online, go to http://www.extreme.indiana.edu/xgws/xsoap/xpp/mxp1/index.html
[7] WoodStox
To access this resource online, go to http://www.cowtowncoder.com/proj/woodstox/index.html
[8] XMLTest 1.0, a White Paper
To access this resource online, go to http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf
[9] XMLTest 1.0 source code and samples
To access this resource online, go to http://java.sun.com/performance/reference/codesamples/
[10] XMLTest 2.0 source code and samples
To access this resource online, go to http://xmltest.dev.java.net/
[11] Java Web Services Performance Community forum
To access this resource online, go to http://performance.dev.java.net/