XBIS XML Infoset Encoding

Dennis M. Sosnoski
Sosnoski Software Solutions, Inc.
14618 NE 80th Pl.
Redmond, WA 98052

Abstract

XBIS is an encoding format for XML documents that is fully convertible to and from text, with Infoset equivalence between the original document text and regenerated document text. It's intended for use in transmitting XML documents between application components, and is therefore designed for processing speed. The current Java language implementation offers several times the performance of SAX2 parsers working from text documents across a wide range of document types and sizes, and across JVMs tested, while also providing a substantial reduction in document size for most types of XML documents.

Introduction

XBIS is the successor to the XML Stream (XMLS) project which was developed in 2000-2001 to address the issue of processing overhead when sending XML documents between application components. The XMLS implementation was designed for use with document models, so it took a document model as input and generated a document model as output. Even though it demonstrated much better performance than alternatives such as Java serialization, or conversion to and from text, it offered no direct performance comparison to parsing XML text because of its document-model focus.

XBIS uses essentially the same formats as XMLS, with only a couple of minor differences. The main difference from XMLS is in the form of input and output supported. XBIS uses a SAX2 event stream as input in order to generate the binary encoding of a document, and generates a SAX2 event stream as output when decoding the binary representation.

The main features that distinguish XBIS from other techniques for representing XML documents are:

1. Full convertibility of arbitrary XML documents to and from text representation (all Infoset information preserved),
2. On-the-fly conversion in both directions, so that output can begin as soon as the first input is received,
3. Very high performance in terms of CPU usage, along with a modest reduction in document sizes,
4. Streaming of documents with some state information retained between documents, allowing even further performance improvements and size reductions,
5. Potential for additional optimizations by the application generating XBIS which will work without any special programming for the application receiving XBIS.

Encoding Format

The XBIS encoding format mirrors the standard text form of an XML document in that all components of the document are present in the same order they'd appear in text. What differs is that XBIS uses a more compact representation of the components, and presents them in a more easily processed form.

The compactness comes mainly from taking advantage of the highly repetitive structure of a normal XML document, where the same element and attribute names are typically used many times over. XBIS defines each name as text only once, then uses a **handle** value to refer back to the name when it is repeated. This same approach is also taken with namespaces, so that even namespace prefixes are not repeated as text.

XBIS can also apply this approach to attribute values and character data, which often use the same text repeatedly. The extent to which this is done is an encoding option, but does not actually effect the format itself; a reader does not need to know what options were used to generate the encoding in order to build a document representation from the encoded form.

Besides the more compact representation, XBIS gains speed on the input side by presenting the document data in predigested form. This eliminates the need for any complex parsing of the input and allows the document to be reconstructed with minimal overhead.

Building Blocks

The XBIS encoding builds up from several simple types. These simple types are described in this section.

Integer Values

Positive integer values are used extensively in the encoding. The standard format of representing these values uses the low-order 7 bits of a byte for the actual value representation, with the high-order 8th bit used as a continuation flag - when the 8th bit is set, the next byte in the encoded stream contains another 7 bits of the value.

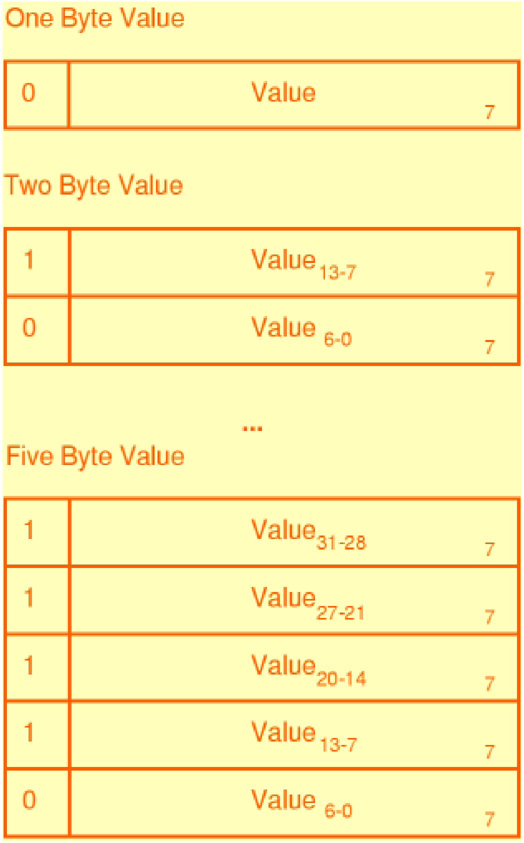


Figure 1. Integer value representations

Figure 1 shows how this looks when applied to values of various sizes. Values in the range of 0-127 can be represented in a single byte, as shown in the top image. Values of 128-16383 require two bytes, as shown in the second image (where the upper byte comes first in the encoded form). Larger values require more bytes, all the way to a maximum of five bytes to represent the maximum possible integer value. In actual use, the values being encoded can generally be represented in one or two bytes.

Quick Values

Quick values are a way of representing a limited range of positive integer values within a portion of a byte. This format is often used in combination with flags in a byte. When a value is to be encoded in this manner it is first incremented. If the incremented value fits within the portion of the byte allowed for the quick value, the value is stored directly within the byte. Otherwise, a `0` is stored within the byte and the incremented value is encoded in the following byte(s) using the normal integer value encoding defined above.

Strings

Strings are the basic building blocks of the serial form. The general string format uses a leading length value which gives the number of characters (**not** bytes) in the string, plus one. The value `0` is used for a `null` string, as opposed to the value `1` which represents a string of zero characters.

This length value is encoded as a normal integer value, as described above. It is followed immediately by the actual characters of the string. Each 16-bit Java character is also encoded as an integer value, so the length of the string data in bytes can potentially be up to three times the number of characters in the string.

String lengths can also be encoded as quick values in some cases. These work slightly differently in that the actual character length of the string is encoded as a quick value, rather than the length-plus-one value used in the general format. Since there is no way of representing a `null` string with this encoding, quick values are used for string lengths only when the string is required to be non-`null`.

Handles

Handle values are used to refer to previously defined items, which include element and attribute names, namespaces, entities, and optionally attribute value and character data strings. Each type of item listed uses a separate set of handles in order to keep the handle values as small as possible, giving the most efficient encoding. The context of a handle reference always determines which type of handle is being referenced.

Actual values start at `1`. Except for namespaces (which use a pair of predefined handles), a handle value of `1` will always represent the first item defined of that type, a handle of `2` the second, and so on.

A `0` in a handle value is used to indicate that a new item of the appropriate type is being defined. The new item is implicitly assigned the next handle value of that type and may then be referenced by that handle value later in the encoding.

When handles are encoded in quick value fields the actual value stored is one greater than the handle value, since as described above the quick value format makes special use of the 0 value. In this case a 1 indicates that a new item is being defined.

Names and Namespaces

Element and attribute name definitions use a common format. The first byte of the definition, shown in Figure 2, contains a quick value field for the namespace handle, along with a separate quick value field for the local name length.

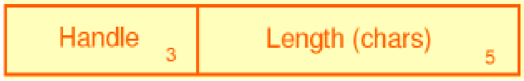


Figure 2. Name definition byte

Any additional data associated with the namespace immediately follows this definition byte. This additional data may take the form of a handle value larger than can be represented in the quick value field (indicated by a 0 value in the field) or a new namespace definition (indicated by a 1 value in the field).

New namespace definitions are encoded as a pair of strings, giving the prefix and the URI for the namespace. If multiple prefixes are defined for the same namespace URI a separate namespace definition is included in the serial form for each prefix.

Two namespace handles are predefined. Handle 1 is always assigned to the **no namespace** namespace, and handle 2 is always assigned to the **xml** namespace.

The additional information for the local name is encoded after any additional information for the namespace. If the name length quick value field in the name definition byte is too small to hold the length, the full length follows any namespace information. It is followed by the encoded characters of the local name.

Structure Encoding

XBIS is a stream encoding which is mainly intended for use with single documents. However, the format allows for encoding arbitrary combinations of elements and documents, and there are cases where this may be very useful to an application. Consider the case where multiple documents of the same type are being transferred from one program to another, for instance. The first document encoded would define most or all of the element and attribute names used in the entire series of documents, allowing the names to be referenced as simple handles in all the following encoded documents.

Each XBIS stream starts with four bytes reserved for XBIS itself. The first byte is a format identifier, which is set by the encoder to specify the format version used to encode the document and checked by the decoder to ensure that it is able to process that format. The only value used at present is 1, identifying the format defined by this document.

The second byte is an identifier for the adapter used to drive the encoding. This value is set by the encoder for information purposes only; the decoder reads this value and makes it available to the application but may not otherwise use it. This requirement is intended to preserve compatibility between all XBIS adapters. There are currently three values defined for this byte, 1 for the dom4j adapter, 2 for the JDOM adapter, and 3 for the SAX2 adapter.

The remaining two bytes of XBIS header are reserved for future use. They are currently written as 0 values and are ignored on input.

After the header the stream consists of one or more *nodes*. These are the primary document structure components, representing everything from a complete document down to a comment or character data string. Attributes are **not** considered nodes in the XBIS encoding, though, and are handled separately.

At the top level only two types of nodes are valid, element nodes and document nodes (when XBIS is used for complete documents, only the document nodes are valid at the top level). Each of these may in turn contain other nodes (including element nodes) as content. The content node definitions are nested within the definition of the containing node.

Each node begins with a node definition byte, which may be followed by additional information for the node. This node definition byte uses different formats for different types of nodes, with the high-order bits used as flags to identify the format.

Element Nodes

Element nodes use the format shown in Figure 3. The high-order bit of the node definition byte is always a 1 for an element node, and the next two bits are used as flags for whether the element has, respectively, attributes and content (0 if not, 1 if so). The remaining bits are a quick value for the element name handle, extended if necessary into the following byte(s). If the name has not previously been defined, the new name definition immediately follows the node definition byte.

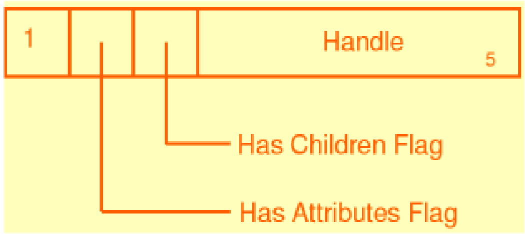


Figure 3. Node definition byte - Element

If the element has attributes, these are next. Attributes begin with an attribute definition byte, taking one of the forms shown in Figure 4. The top format is used for attributes with ordinary (unshared) values. The bottom format is used for attributes with shared values, which use handles to avoid encoding the same text repeatedly. Both ordinary and shared attribute values may be used in any combination.

Both attribute definition byte formats use the low-order bits of the byte for a quick value of the attribute name handle (extended, if necessary, to the following byte(s)). If the name has not previously been defined the name definition immediately follows the attribute definition byte.

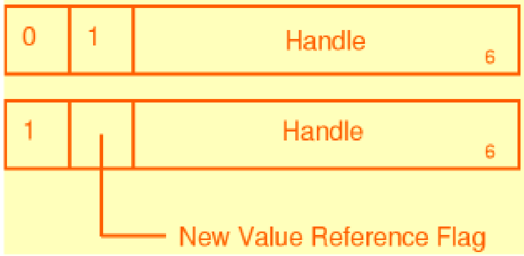


Figure 4. Attribute definition byte formats

The actual value of the attribute is next. For ordinary values, and for new shared values (as indicated by the flag in the attribute definition byte), these are strings in the general format. For previously-defined shared values the value is represented by a handle which identifies the value text.

The list of attributes for an element is terminated by a 0 value in place of an attribute definition byte (which can never be 0). If the node definition byte for the element does not indicate that attributes are present this 0 value is not included in the encoding.

If the element has content, the content nodes are next. The content nodes can be of any type (subject to XML structure concerns - a document as content of an element is obviously invalid, for instance). Each begins with a node definition byte, and as with the attributes the list of content nodes is terminated by a 0 byte in place of a node definition byte.

Text Nodes

Plain text (ordinary character data) nodes use the format shown in Figure 5. This gives the text length as a quick value in the low-order bits of the node definition byte (extended, if necessary, to the following byte(s)). It is followed by the actual encoded characters of text.

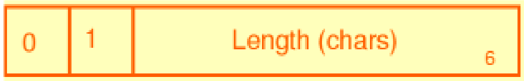


Figure 5. Node definition byte - Plain text

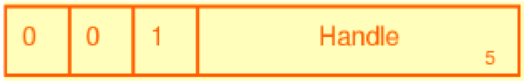


Figure 6. Node definition byte - Shared text

Shared text (ordinary character data) nodes use the format shown in Figure 6. This gives the handle for shared text in the low-order bits of the node definition byte (extended, if necessary, to the following byte(s)). If the text has not previously been defined (as indicated by a 0 value for the handle), the text definition immediately follows the node definition byte, as a string in the general format.

Both types of text nodes can be used within a single document, in any combination.

Namespace Nodes

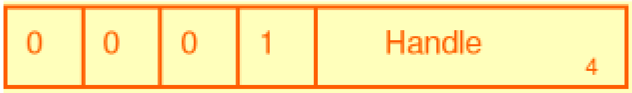


Figure 7. Node definition byte - Namespace

Namespace declaration nodes use the format shown in Figure 7. This gives the handle for the namespace in the low-order bits of the node definition byte (extended, if necessary, to the following byte(s)). If the namespace has not previously been defined (as indicated by a 0 value for the handle), the definition immediately follows the node definition byte. The namespace definition format is as described earlier in the **Names and Namespaces** section. The use of namespace declaration nodes is optional in XBIS.

Other Nodes

The other node types use a simple format in which the node definition byte just identifies the type of node, and any additional information for that node type is in the following bytes. These other node types are:

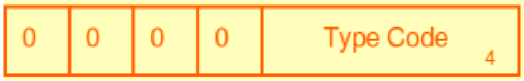


Figure 8. Node definition byte - Other

1. Document node: followed by content node list, as for element node
2. Comment node: followed by comment text, in the general format
3. CDATA node: followed by CDATA text, in the general format
4. Processing Instruction node: followed by a target and value, both as text in the general format
5. Notation node: followed by name, public id, and system id, all as text in the general format
6. Entity definition node: Not currently implemented, details TBD
7. Entity reference node: Not currently implemented, details TBD
8. Document Type node: followed by name, public id, and system id, all as text in the general format

The value `0` for a node definition byte is used to indicate the end of a list of node definitions. All other values are reserved and currently unused.

Performance

The current XBIS code includes a test program for comparing XBIS performance with SAX2 parsers. The test program uses one or more sets of test data, each consisting of either a single document or a collection of documents. The test program first reads the current test data set into memory as an array of bytes, then calls either a SAX2 text processor method or an XBIS processor method, as selected by a command line argument (separate executions are used to test XBIS vs. SAX2 performance). The called method performs the actual tests and fills measured time values into an array of results, which are accumulated by the test program across all test data sets and printed at the end.

Timing Methodology

The actual timing measurements are done using several passes over the test data set. Each pass may involve several repetitions of an operation with the test data set. The elapsed time for each pass is measured, and only the best time is saved and returned. This technique was adopted in order to obtain consistent timing results, since average times across even a larger number of passes showed much more variation than the best pass times, apparently due to JVM whims.

The actual sequence of operations is as follows for the SAX2 text processor method:

1. Parse document from the in-memory text (as a `ByteArrayInputStream`), with processing of SAX2 events by a simple handler that counts different types of items. This is measured as the input time for the SAX2 parser.
2. Parse document from the in-memory text, with processing of SAX2 events by an `XMLWriter` (from David Megginson) instance, generating text output to memory (using a `ByteArrayOutputStream`). This is measured as the parse+output time for the SAX2 parser, and the best time from (1) is subtracted from the best time for (2) to find the inferred output time for SAX2 text generation, while the actual output size

is saved as the text size for the document.

For the XBIS processor method the sequence is:

1. Parse document from the in-memory text, as in operation (1) for the SAX2 text processor method.
2. Parse document from the in-memory text, with processing of SAX2 events by the XBIS output generator, with output generated to memory. This is measured as the parse+output time for XBIS, and the best time from (1) is subtracted from the best time for (2) to find the inferred output time for XBIS generation. The actual output size is saved as the XBIS size for the document.
3. Process the XBIS output generated in (2) with an XBIS reader that generates SAX2 events, with the same simple handler as used for the SAX2 processing in (1) used to verify that the counts match the original SAX2 parse results. This is measured as the input time for XBIS.

Test Data

The test data sets are broken up into medium and large individual documents, along with several collections of smaller documents. The medium documents are:

- `periodic.xml`, periodic table of the elements in XML. Some attributes, fairly flat tree (114K bytes).
- `soap2.xml`, generated array of values in SOAP document form. Heavy on namespaces and attributes (131K bytes).
- `xml.xml`, the XML specification, with the DTD reference removed and all entities defined inline. Presentation-style markup with heavy mixed content, some attributes (156K bytes).

The large documents are:

- `weblog.xml`, a log of web page accesses reformatted as XML. Flat structure with no attributes and generally short character data sequences as content (2.9M bytes).
- `factbook.xml`, CIA World Factbook data reformatted as XML. Variable structure with no attributes and heavy character data content (4.0M bytes).

The collections of small documents are:

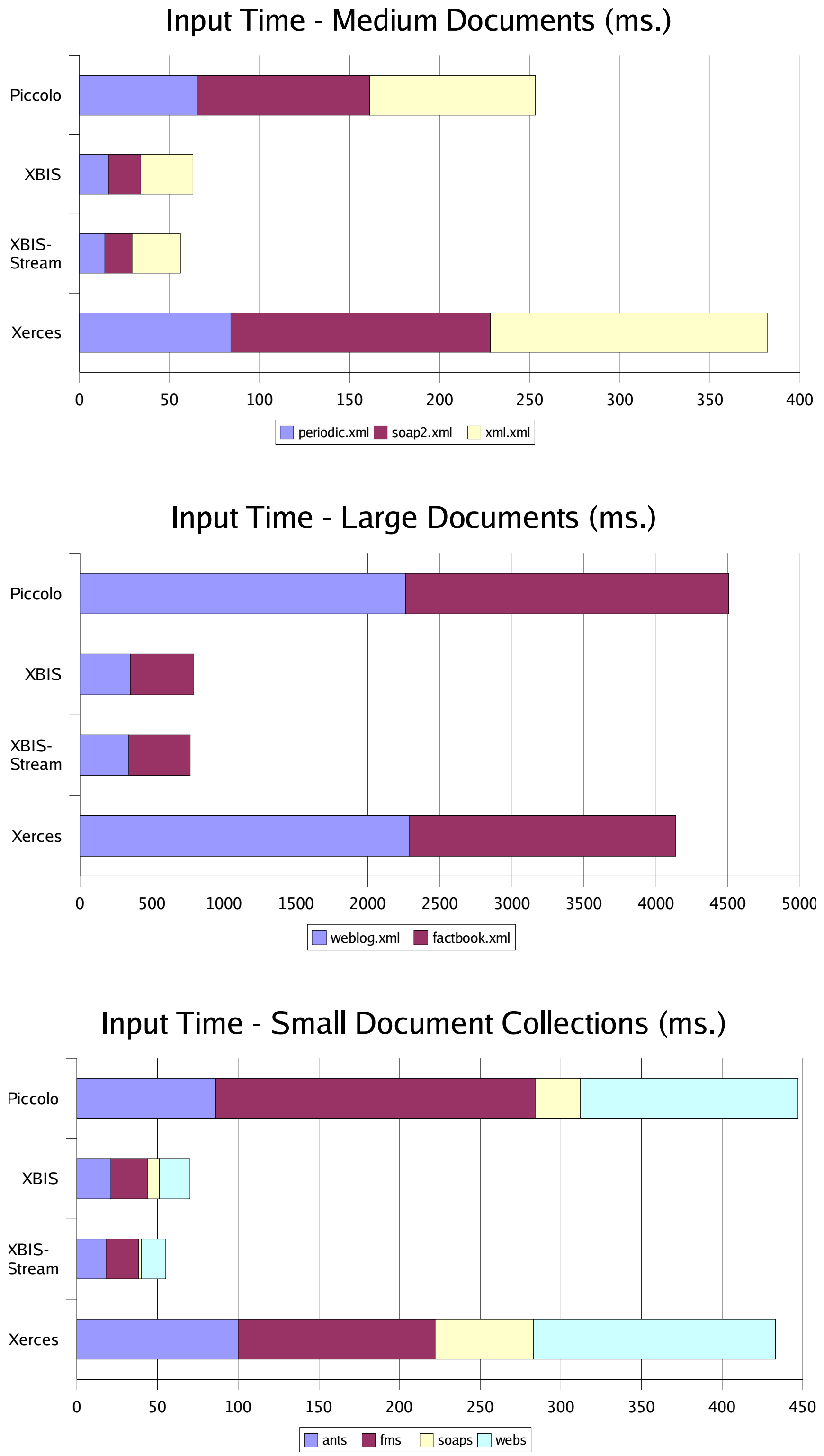
- `ants`, XML configuration files for the Ant build utility from a number of open source projects (18 documents, 100K bytes total).
- `fms`, RDF documents from Freshmeat.net (37 documents, 136K bytes total).
- `soaps`, SOAP request and response documents from an early version of the SOAP 1.2 specification and from an interoperability test set (42 documents, 30K bytes total).
- `webs`, web application configuration files from a number of open source projects (70 documents, 132K bytes total).

Test Results

Timing tests were conducted on an Athlon XP 2200+ system running Mandrake Linux 9.1, using both the Sun 1.4.2 JVM and the IBM 1.4.0 JVM and both the Piccolo and Xerces2 parsers. XBIS was tested with both normal document-at-a-time operation and streamed operation, where handle definitions were preserved across successive documents in a collection. Even with the streamed operation, definitions were preserved only within a set of documents and not across repetitions using the same set of documents.

The following diagrams show the results of the timing and memory usage tests. The sizes were the same across JVMs, and varied by less than one percent between parsers used, so only one set of sizes are shown.

The output timings are inherently less accurate than the input timings because they represent inferred values (obtained by subtracting the parse-only time from the parse-write time, on the assumption that the parse time is reasonably constant). The results are consistent across tests, so the basic timing assumption appears justified. However, the XMLWriter code used for SAX2 output has not been heavily optimized and probably suffers in the comparison with the XBIS code because of this. It's possible that other text generation code could come closer to matching XBIS performance.

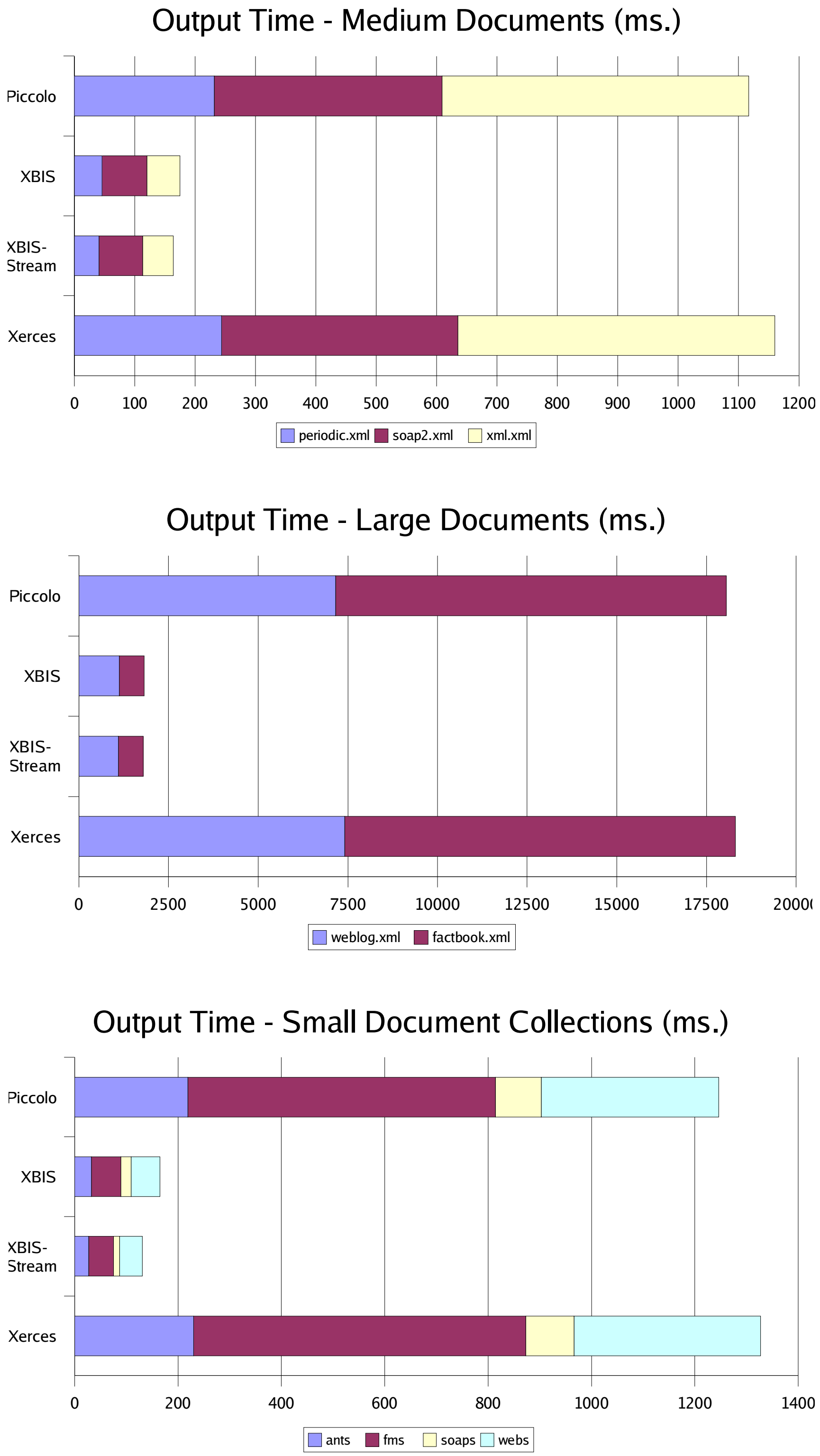


Input Time - Medium Documents (ms.)

Input Time - Large Documents (ms.)

Input Time - Small Document Collections (ms.)

Figure 9. Input times, using the Sun 1.4.2 JVM



Output Time - Medium Documents (ms.)

Output Time - Large Documents (ms.)

Output Time - Small Document Collections (ms.)

Figure 10. (Inferred) Output times, using the Sun 1.4.2 JVM

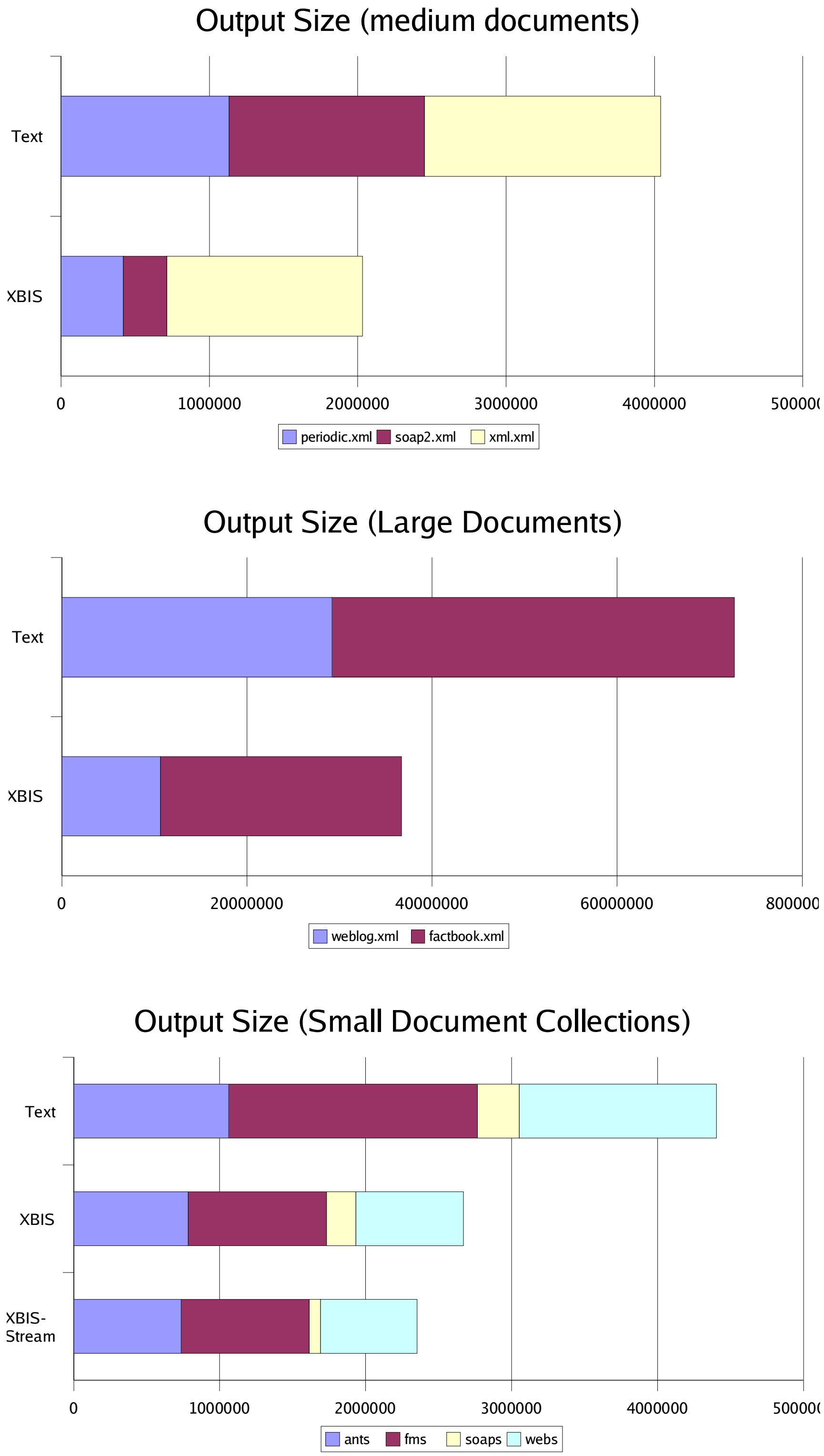


Figure 11. Output document size (bytes)

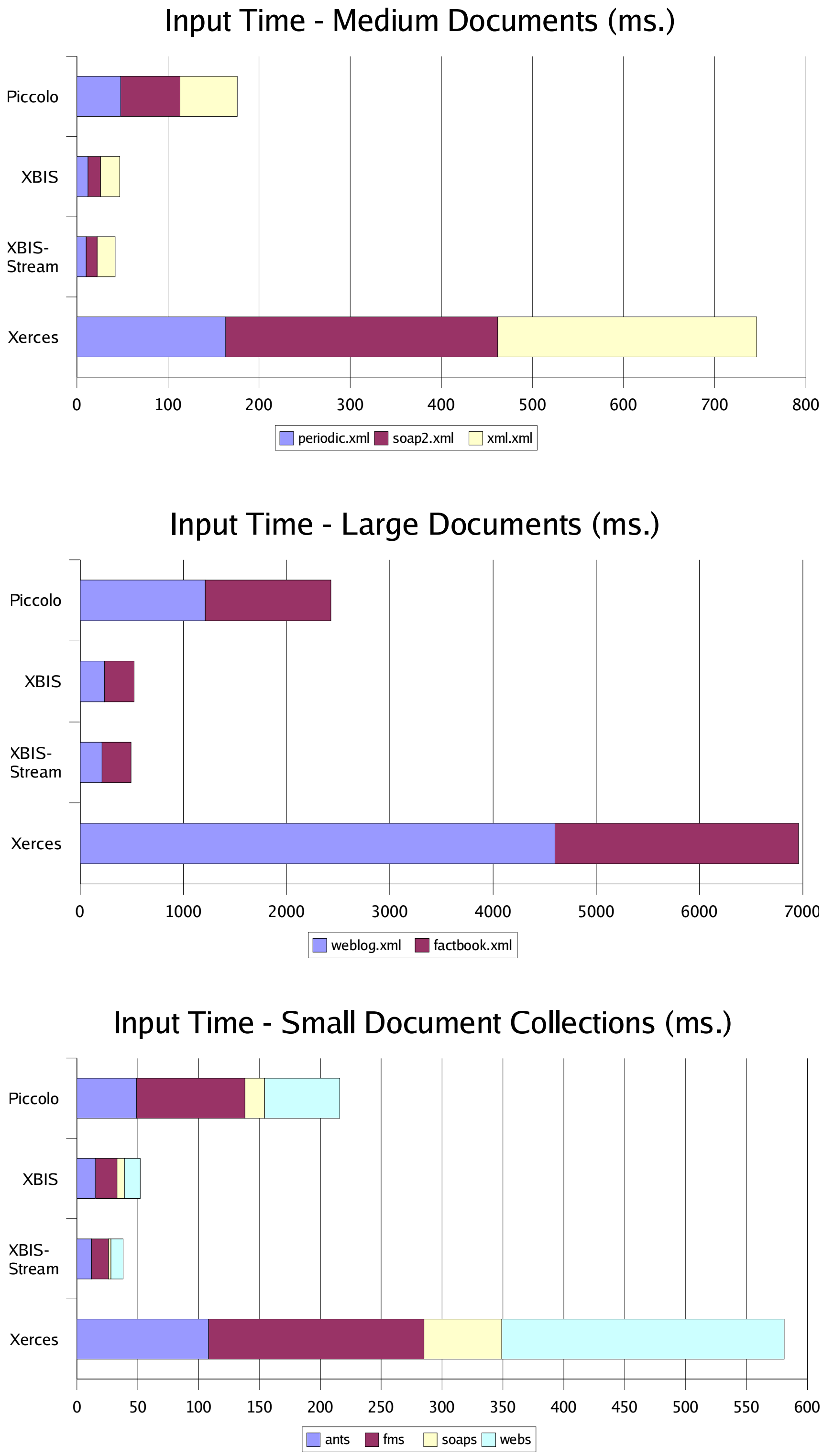
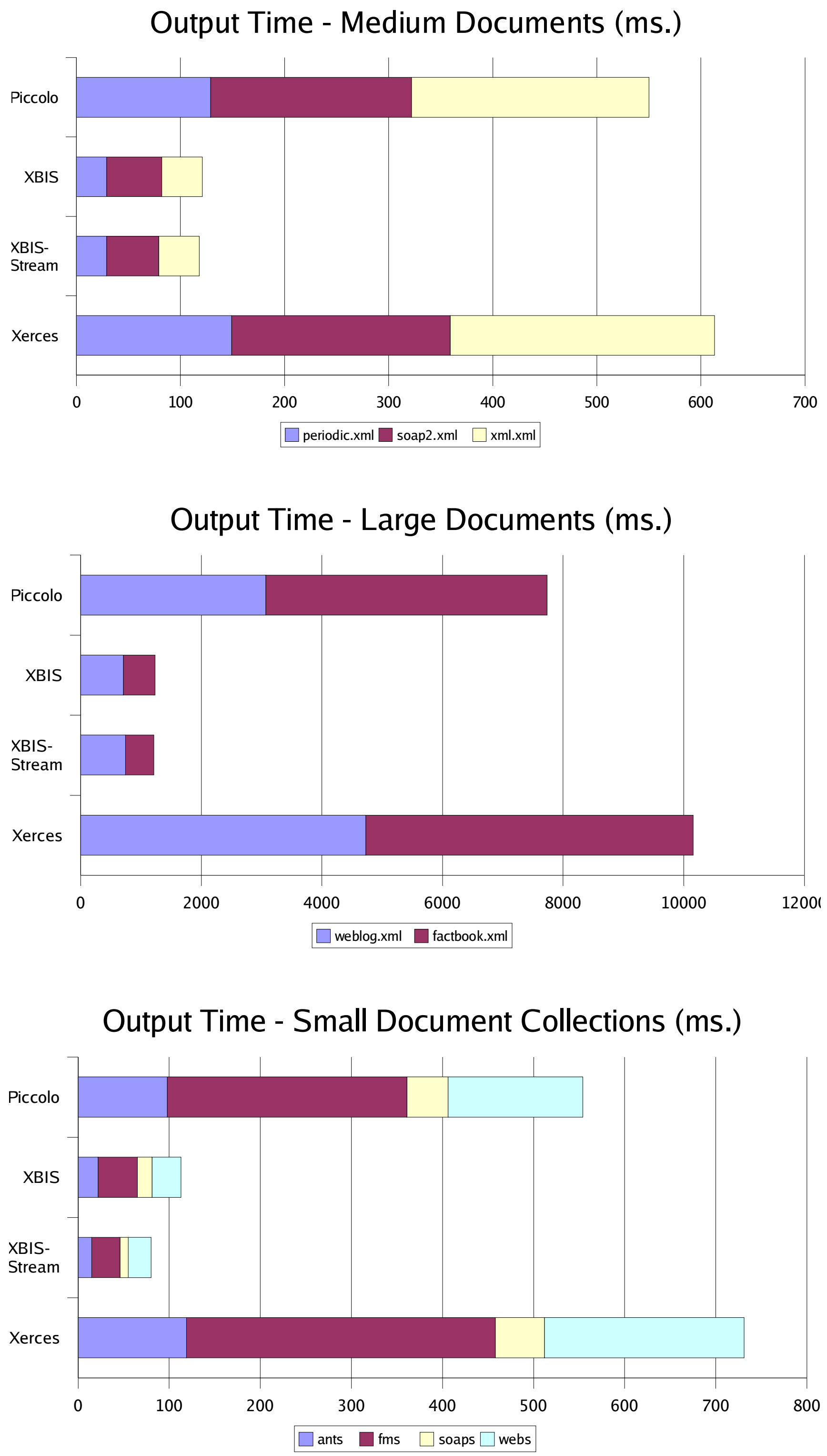


Figure 12. Input times, using the IBM 1.4.0 JVM



Output Time - Medium Documents (ms.)

Output Time - Large Documents (ms.)

Output Time - Small Document Collections (ms.)

Figure 13. (Inferred) Output times, using the IBM 1.4.0 JVM

As seen from the test results, XBIS offers several (generally 4 to 8) times the input performance of the SAX2 parsers tested for every type of document and both JVMs. On output, it provides even more of a performance advantage (generally 7 to 10 times for the Sun JVM, though less for the IBM JVM) as compared to a widely used (but not heavily optimized) SAX2 text output generator.

XBIS also provides a considerable size reduction as compared to XML text. The XBIS encoding mainly optimizes markup while leaving document character data content essentially unchanged, so the extent of the size reduction is heavily dependent on the type of document. This ranges from a relatively minor 17% in the case of the text-heavy XML specification to a maximum of 72% for the streamed SOAP document collection.

Size reductions can be even greater than this when XBIS is used for highly uniform document collections (as for SOAP documents involving a single Web service), or when the encoding application makes use of domain-specific knowledge to preconfigure common attribute value and character data content sequences.

Other Issues

As compared to a choice such as gzip on raw XML document text, XBIS delivers much less compression but much better CPU performance. It should not be considered as an alternative to gzip for cases where compression is the primary goal of a document encoding. Where performance is the main concern, however, XBIS excels.

The text encoding used by XBIS is essentially a variation on UTF-8, so XBIS provides full support for internationalization. XBIS also preserves all significant aspects of the text form of an XML document, so it allows for easy conversion to human-readable form (requiring only an interceptor for the translation). Since XBIS is a text-based transformation of XML documents it is not effected by choice of schema language and does not require that documents even follow any predefined schema.

Random access and dynamic update are not major design considerations for XBIS. Streaming is a major goal, and XBIS handles this very well. Both encoding and decoding are done using streamed data, so there's no need to have access to the entire document before beginning encoding or decoding. When multiple documents with the same schema are sent over a single stream XBIS is also able to retain information between documents in order to reduce redundancy.

XBIS has no known "intellectual property" encumbrances. The original XMLS encoding that XBIS is based on was developed in 2000-2001 and made public as open source software and documentation in 2001.