Compressing XML with Multiplexed Hierarchical PPM Models

James Cheney
Cornell University
Ithaca, NY 14850
(607) 255-1146
jcheney@cs.cornell.edu

1 Introduction

Extensible Markup Language (XML) is a standardized language that "describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them"[7]. According to Bosak and Bray [1], XML is the "next big thing" after HTML. It is gaining momentum in many areas of the computer industry; for example, Microsoft has announced plans to base future software systems on XML [2]. XML is not a specific markup language like HTML, but instead a meta-language for describing markup languages together with a strong standard for creating and parsing documents.

Whatever XML's advantages, it has one glaring disadvantage: document size. XML's parent standard, Standardized General Markup Language (SGML), made many provisions for minimizing document size. These options made SGML complex and difficult to implement, and they were omitted from XML. Indeed, the XML standard explicitly states that markup terseness was *not* a design goal. Consequently, XML is easy to process but verbose. XML documents can be many times larger than equivalent non-standardized text or binary formats, even if compressed. There is growing concern in the XML community that inefficiency arising from document size will hinder adoption and use of XML.

From an information-theoretic point of view, this situation is vexing because two sources carrying the same messages should have the same entropy and so compress to about the same size using any universal compressor. The problem is that XML documents may have nonlocal redundancy arising from XML's tree structure, which is difficult for text compressors to discover. Conversely, XML-conscious compression techniques might be able to compress XML documents as well as or better than other representations. Fortuitously, XML's simple design makes testing this hypothesis easier than in previous structured-data compression approaches such as syntax-based compression of program files [12, 3] or machine code compression[8, 16].

Liefke and Suciu [15] describe XMILL, an XML compressor that transforms documents to expose redundancy, then applies standard text compressors. XMILL combined with `gzip` compresses XML data about 10% better than `gzip` on equivalent non-XML forms; further improvement (up to 50%) is possible with user assistance in the form of complex command-line parameters. This work shows that XML-conscious compression can do better than text compression alone. However, XMILL's base transformation has several drawbacks: namely, it precludes incremental (online) processing of compressed documents, it actually hinders compressors other than `gzip`, and it requires user assistance to achieve the best compression.

```
<?xml version="1.0"?>
<doc>
 <title>The Great American Novel</title>
 <author>A. Nonymous</author>
 <copyright>Copyright 2000</copyright>
 <chapter id="1">
  <title>The Beginning</title>
  <body>
   ...body of chapter...
  </body>
 </chapter>
 ...other chapters...
</doc>
```

Figure 1: XML example

In this paper, we will describe alternative approaches to XML compression that illustrate other tradeoffs between speed and effectiveness. We describe experiments using several text compressors and XMILL to compress a variety of XML documents. Using these as a benchmark, we describe our two main results: an online binary encoding for XML called *Encoded SAX* (ESAX) that compresses better and faster than existing methods; and an online, adaptive, XML-conscious encoding based on Prediction by Partial Match (PPM) [5] called *Multiplexed Hierarchical Modeling* (MHM) that compresses up to 35% better than any existing method but is fairly slow. First, of course, we need to describe XML in more detail.

2 XML background

Superficially, XML documents looks a lot like HTML documents (see Figure 1). XML documents contain *element tags*, including start tags like `<title>` and end tags like `</title>`. Elements can contain other elements nested inside them, forming a tree structure. Elements can also contain plain text, comments, and special instructions for XML processors ("processing instructions"). Opening element tags can have associated *attributes* with *values*, such as `type` in `<list type="ordered">`. These are the most common constructs in XML; for more detail see the specification [7].

XML documents must be *well-formed*. Well-formed XML is more restrictive than HTML: elements must be nested, start tags must match corresponding end tags, and attribute names within start tags must be unique, among other things. XML documents can also include a *document type definition* (DTD). For example, the DTD statement `<!ELEMENT list (listitem*)>` specifies that `list` elements can contain sequences of `listitem` elements. *Validation*, or checking that an XML document follows the rules of a DTD, ensures that only meaningful data reaches an application.

The XML standard fixes many aspects of the behavior of compliant applications when processing XML, so there are many libraries that perform routine XML parsing tasks and report only important information to applications. Most XML parsing libraries use one of two interfaces, Simple API for XML (SAX) [17] and Document Object Model (DOM) [6]. SAX is an event-based API, suitable for one-pass algorithms such as search tools and filters. DOM provides an interface to XML data stored in memory as trees, and is better suited to multi-pass algorithms.

There are many additional tools, standards, and technologies in development by the W3C and Web community, but which are not relevant in this paper. For more information see the W3C web site's XML page, `http://www.w3c.org/XML`.

3 Compressing XML as text

XML is stored in plain text files, so the most obvious approach to XML compression is to use existing text compressors. To establish a benchmark against which

we can compare other methods of XML compression, we constructed a corpus containing a wide variety of forms of XML data. The corpus includes XML versions of formal proofs and safety-annotated assembly language (`proof`, `tal`), as well as Shakespearean plays (`play`), statistical and scientific databases (`stats`, `elts`), and XML versions of the XML specification (`spec1`, `spec2`). We also included some of the small XML examples (`sprot`, `tpc`, `treebank`, `weblog`) distributed with XMILL; unfortunately the full corpus was not available for these experiments. The "Size" and "M%" columns of Table 1(a) list the file sizes and markup percentages of the documents. The documents are listed in order from least to most markup.

The documents chosen for the corpus reflect a wide a variety of sources, in order to avoid skewing our results to a particular kind of data. It is possible that the real uses of XML data will be different from the somewhat artificial uses listed above. Thus, any conclusions drawn based on our corpus must be tentative. Nevertheless, because XML is not yet a mature, everyday technology, this would likely be true of any alternative proposed corpus.

We compressed each document using `gzip`[10], `bzip2`[19], `ppmd+`[5, 21] (with a context bound of 5), and `ppm*`[4, 21]. The compression results, reported in bits per original XML document character, are shown in Table 1(b). For each file, the best compression in a table is in **bold**, and the best compression overall (including Tables 1 and 2) is in ***bold italic***. At the bottom of each table is listed the average bit rate, weighted average (total compressed bits/total bytes), and compression time.

Markup tends to be more redundant and compressible than text. *Textual* documents (less than 50% markup) tend to have high bit rates of around 1.5–2.0. This is below the 2.1–2.2 bpc usually seen with English text, and reflects an averaging of very redundant markup and less redundant text. *Structured* documents (more than 75% markup) compress up to several times smaller than text documents. Documents mixing text and structure (50%–75% markup) tend to fall in between these extremes.

There is no clear winner. The `bzip2` compressor is best overall; it does 20–30% better than `gzip` and about as well as `ppm*`. However, each compressor performs poorly on at least one document. For structured documents, there is a large gap between `gzip` and `ppm` and the other compressors (as much as a factor of 4), but otherwise there are no obvious *a priori* reasons to believe a particular compressor will be effective on an unknown document.

Because all XML documents are generated by a fairly restrictive context-free grammar, perhaps grammar-based compression techniques such as Nevill-Manning and Witten's SEQUITUR [18] or Kieffer and Yang's grammar-based codes [13] would do better. We have experimented with a version of YK compression and found that it compressed about as well as `bzip2` and `ppm*` on our corpus. Nevertheless other forms of grammar-based compression, perhaps tailored to XML, might offer improvements.

Plain-text XML compression is easy, fairly fast, and relies on existing, robust compressors. On the other hand, good compression is either very slow (using `ppm*`), or fairly slow and off-line (using `bzip2`). Off-line compression is undesirable for XML because it forces a long wait before document parsing and processing can begin. Moreover, there is no reason to believe that text compressors can take full advantage of the redundancies present in structured XML data, especially in light of the high

(a) Documents			(b) Raw XML text (bpc)				(c) XMILL transformation (bpc)				
Name	M%	Size(B)	GZIP	BZIP2	PPM	PPM*	T_{XMILL}	GZIP	BZIP2	PPM	PPM*
play	39%	251898	2.160	1.549	1.576	**1.542**	5.365	2.053	1.669	**1.625**	1.707
treebank	41%	6298	1.782	1.524	**1.336**	1.434	2.267	1.285	1.367	**1.172**	1.222
spec2	42%	201918	2.139	1.722	**1.653**	1.663	6.656	2.128	2.190	**1.752**	1.829
spec1	43%	196308	1.948	1.539	1.517	**1.511**	6.619	1.942	1.893	**1.631**	1.697
weblog	57%	2244	2.053	2.389	**1.925**	1.986	4.078	2.246	2.642	**2.078**	2.132
tpc	58%	287906	1.475	**1.100**	1.391	1.279	3.718	1.208	***1.064***	1.141	1.224
sprot	67%	10248	2.033	2.020	**1.899**	1.991	4.613	1.998	2.158	**1.943**	2.083
elts	77%	113135	0.620	**0.415**	0.597	0.443	2.683	0.440	0.408	0.408	**0.405**
stats	89%	669309	0.798	**0.368**	0.591	0.378	2.145	0.446	**0.352**	0.360	0.387
proof	99%	252682	0.311	0.166	0.481	**0.156**	2.846	0.171	0.142	0.283	**0.134**
tal	99%	734535	0.312	**0.121**	0.467	0.127	2.167	0.168	**0.131**	0.223	0.139
average			1.420	1.175	1.223	**1.137**	3.924	1.282	1.274	**1.143**	1.178
weighted			1.001	**0.667**	0.877	0.680	3.369	0.817	0.729	**0.719**	0.720
time(s)			2.98	16.55	62.23	399.16	0.82	2.88	7.65	40.38	189.92

Table 1: (a) Document statistics, (b) Text and (c) XMILL compression results

variation in bit rates among the compressors.

4 Existing XML-conscious compressors

We are aware of two existing XML-conscious compression techniques: XMLZIP, by XML Solutions [20], and Liefke and Suciu's XMILL[15].

XMLZIP parses XML data and breaks the structural tree into many components: a "root" component containing all data up to depth d from the root, and one component for each of the remaining subtrees starting at depth d. The root component is modified by adding references to the remaining subtrees, and the components are compressed by Java's built-in ZIP/DEFLATE archive library. This does not improve compression over compressing the whole document: in fact, using XMLZIP with $d = 2$, the example XML file that comes with the XMLZIP distribution compresses to 225512 bytes, whereas ordinary ZIP compresses it to only 118848 bytes. Increasing d tends to decrease compression performance, since redundancies across separated subtrees cannot be used in compression. XMLZIP's chief benefit that it allows limited random access to XML documents without storing the whole document uncompressed or in memory.

XMILL parses the XML data, transforms it, and then compresses the result using an ordinary text compressor. The XMILL transformation (henceforth referred to as T_{XMILL}) splits the data into three components: one for element and attribute symbol names, one for plain text, and one for the document tree structure. The XML fragment `<elt att="abcd">XYZ</elt>` might be transformed to `[elt,att];` `[abcd, XYZ];` `[S0 S1 T X T X]`, where `Si` refers to symbol table entry `i`, `T` refers to the next unused text string, and `X` terminates an element or attribute list. In fact, XMILL allows multiple text containers within the text component, in which case a reference `Ti` refers to the next unused string in container `i`. By default, XMILL uses a container-building heuristic of "group by enclosing structure", so for example all text immediately enclosed in elements `<elt>...</elt>` is put in one container. Users can provide other heuristics using a *path expression/user compressor* language of command line options. This facility makes better compression possible, but requires user expertise and effort.

Table 1(c) illustrates XMILL's compression performance on our corpus using sev-

eral underlying compressors. The "T_{XMILL}" column shows the bit rate of the base XMILL transformation without any further compression, and the GZIP column shows the results of compressing using XMILL's default configuration, T_{XMILL} followed by GZIP. As noted in [15], it is possible to use other compressors besides GZIP with XMILL. The XMILL source distribution comes with a version that uses BZIP2. We have also used PPM and PPM* with XMILL. The remaining three columns show the results of these experiments. Note that this is *not* the same as running XMILL to get GZIPped, transformed data and then compressing that with other compressors. Doing so would give misleading results because compressed data usually doesn't compress further. Instead, we have shown the results of replacing GZIP with other compressors, which is a fair way to compare XMILL's approach with other attempts that use more advanced encoders like BZIP2 and PPM.

T_{XMILL} improves overall compression for `gzip` and `ppm` by about 18%. Structured document compression with `gzip` and `ppm` also improves by 20–40%. Surprisingly, T_{XMILL} with `bzip2` and `ppm*` compresses 5–10% *worse* than the raw text does. We speculate the reason is that T_{XMILL} attempts to do something that these compressors already do better. The `bzip2` and `ppm*` compressors can take advantage of contexts much longer than the immediately enclosing XML element or attribute name. T_{XMILL} disrupts these contexts, cutting off opportunities for compression.

T_{XMILL} is itself fast, and speeds up subsequent compression by as much as 50% because compressors ultimately process fewer symbols. On the other hand, XMILL scatters parts of the document, making incremental processing impossible. Also, although the path-expression/user-compressor language is powerful, there is no guarantee that users will be able to put it to its best use. In the next section we will describe an alternative transformation that is also fast, compresses almost as well with `gzip`, compresses better with other compressors, and works incrementally.

5 SAX event encoding

Simple API for XML (SAX) is an event-driven interface in which an application supplies the parser with "callback" event handlers that are invoked when certain parsing events occur. For example, in parsing the fragment `<elt att="abcd">XYZ</elt>`, a SAX parser would report events

```
startElement("elt",("att","abcd"))
characters("XYZ")
endElement("elt")
```

The event sequences provide all the information an XML-compliant application needs. We can leverage the work a SAX parser does by encoding the sequence of events. A decoder can decode these events, and reconstitute an XML document equivalent to the original. Alternatively, the decoder could act as a SAX parser, parsing encoded event sequences instead of text, and sending these events directly to the application.

We chose an event encoding that uses single bytes to encode element start tags, end tags, and attribute names and to indicate events such as "begin/end characters", "begin/end comment", and so on. The encoder and decoder maintain consistent symbol tables; whenever a new symbol is encountered, the encoder sends the symbol name and the decoder enters it into the table. We omit the (largely irrelevant) details

	(a) Encoded SAX (bpc)					(b) New Models (bpc)			(c) Comparison	
name	ESAX	GZIP	BZIP2	PPM	PPM*	MM	MHM	MHM*	XML/BZ2	% imp
play	6.329	2.068	1.532	**1.504**	1.535	1.517	***1.489***	1.499	1.549	3.2%
treebank	7.272	1.678	1.406	**1.305**	1.393	1.246	1.171	***1.164***	1.524	23.7%
spec2	6.738	2.018	1.683	**1.596**	1.630	1.653	***1.592***	1.603	1.722	6.9%
spec1	6.826	1.840	1.507	**1.470**	1.473	1.515	***1.441***	1.446	1.539	6.1%
weblog	5.030	2.068	2.267	**1.897**	1.936	1.783	***1.765***	1.772	2.389	25.8%
tpc	5.687	1.438	**1.081**	1.290	1.215	1.236	**1.092**	1.096	1.100	0.4%
sprot	5.541	2.009	1.954	**1.848**	1.906	1.841	***1.778***	1.816	2.020	10.1%
elts	4.081	0.577	**0.392**	0.473	0.420	0.441	0.381	***0.378***	0.415	8.9%
stats	2.566	0.593	**0.352**	0.464	0.364	0.356	0.314	***0.311***	0.368	15.7%
proof	2.517	0.198	0.150	0.264	**0.140**	0.273	0.168	***0.106***	0.166	36.3%
tal	2.452	0.194	**0.119**	0.223	0.121	0.223	0.139	***0.093***	0.121	23.3%
average	5.005	1.335	1.131	1.120	**1.103**	1.098	1.029	***1.026***	1.175	12.6%
weighted	3.912	0.877	**0.649**	0.726	0.660	0.704	0.630	***0.615***	0.667	7.8%
time(s)	2.56	5.14	8.41	42.32	74.13	39.18	46.97	314.94		

Table 2: (a) ESAX and (b) MHM compression results, (c) comparison

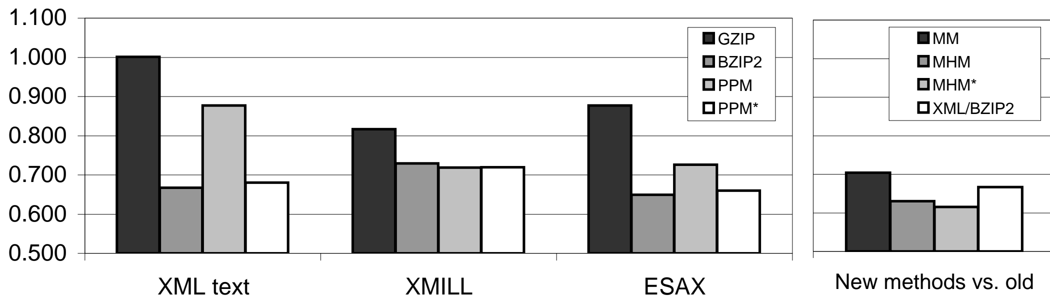

Table 3: Weighted average over entire corpus (bits per symbol)

of the encoding. As an example, assuming the tag `elt` has been seen before, and is represented by byte `10`, but attribute name `att` has not, and the next available byte for attribute names is `0D`, our running XML fragment would be encoded as

<elt	att=	"asdf"	>	XYZ	</elt>
10	0D a t t 00	a s d f 00	FF	FE X Y Z 00	FF

This encoding is implemented using the `expat` XML parser, version 1.95. In principle, the encoding can be implemented using any SAX parser. Table 2(a) shows the results of compressing the encoded SAX events; the “ESAX” column lists the bits of encoded SAX per original symbol.

Using `gzip` or `ppm`, ESAX is only 7% or 1% worse than XMILL, respectively. With the other compressors , ESAX is 8–10% better than XMILL and even slightly (3%) better than `bzip2/ppm*` on the raw XML. Although ESAX is slower than XMILL, overall compression time is only substantially worse for `gzip` and is actually considerably better (60%) than XMILL for `ppm*`. Moreover, using `bzip2`, ESAX combines XMILL's speedup factor of 2 with an improvement in compression. Furthermore, ESAX, unlike XMILL, can be encoded and decoded online, so that the XML data can be processed incrementally.

6 Multiplexed hierarchical modeling

We now describe a modeling technique called multiplexed hierarchical modeling (MHM), based on the SAX encoding from the previous section and on PPM modeling. The technique employs two basic ideas: *multiplexing* several text compression models

based on XML's syntactic structure (one model for element structure, one for attributes, and so on), and *injecting* hierarchical element structure symbols into the multiplexed models.

First, we must briefly review the PPM text compression model. PPM models maintain statistics concerning which symbols have been seen in which contexts of preceding symbols. For each symbol, the model is used to estimate a probability range. This probability range is used to transmit the symbol using arithmetic coding. Then, the model is updated to indicate the symbol has been seen in the context. Probability estimation works as follows: If the symbol has been seen in the longest matching context, then the probability is the relative frequency in the context. Otherwise, an "escape symbol" is encoded, and the next longest context is tried, and so on. The decoder maintains the same model and uses symbols seen so far and escape symbols to decode incoming symbols and update its model.

Based on our observations and on the behavior of XMILL, we conjectured that in XML, hierarchical contexts based on the path from the root to the symbol might be as accurate or more accurate than preceding-symbol contexts. To test this hypothesis, we modified the PPMD+ sequential model to allow backtracking based on the hierarchical structure. For example, in encoding `<a><b>X</b>Y</a>`, hierarchical backtracking modeling (HBM) uses `<a><b>` as a context for `X` but uses `<a>` and not `X</b>` as a context for `Y`. We implemented HBM in PPMD+ and found that it only improves compression for a few documents and worsens compression for others. Comparing the predictions made by sequential modeling and hierarchical backtracking revealed that hierarchical contexts are used in several different ways depending on the syntactic class (element, attribute, or string) of the next symbol. Since HBM is unaware of syntactic context, its estimates are averages of several distinct distributions. Consequently hierarchical backtracking predictions are often poorer than the predictions made by the sequential model.

In order to avoid this averaging phenomenon, we decided to multiplex several models, switching among them based on syntactic context supplied by the parser. We used four models: one for element and attribute names, one for element structure, one for attributes, and one for strings. Each model maintains its own state but all share access to one underlying arithmetic coder; encoding and decoding still run incrementally. Our running example gets encoded as:

Model	<elt	att=	"asdf"	>	XYZ	</elt>
Elts:	10				FE	FF
Atts:		0D	a s d f 00	FF		
Chars:					X Y Z 00	
Syms:		a t t 00				

Each row represents the symbols seen by a model. The following table breaks down the composition of the ESAX versions of the test files by class:

	play	treebank	spec2	spec1	weblog	tpc	sprot	elts	stats	proof	tal
Elements	12%	18%	7%	9%	13%	23%	15%	14%	40%	45%	44%
Attributes	3%	5%	17%	17%	3%	6%	27%	31%	13%	55%	55%
Characters	84%	74%	76%	73%	75%	71%	55%	54%	47%	0%	0%
Symbols	0.1%	2.1%	0.4%	0.2%	9%	0.1%	3.5%	0.5%	0.2%	0.4%	0.2%

If neighboring symbols from different syntactic classes are drawn from distinct independent sources, the multiplexed models adapt to these distributions better than a single model can and prediction improves. However, sometimes symbols are strongly correlated with nearby symbols in different syntactic classes. In this case, multiplexing harms compression by breaking up dependences. On average, these factors balance out, resulting in compression slightly better than sequential modeling (see Table 2, "MM" column).

Multiplexing enables more effective hierarchical structure modeling. Model multiplexing breaks existing cross-class sequential dependencies; if we can restore them, we can improve prediction. A common case for these dependencies is for the enclosing element tag to be strongly correlated with enclosed data (recall that this is the motivation for XMILL's text container grouping). The multiplexed hierarchical model (MHM) exploits this by by injecting the enclosing tag symbol into the element, attribute, or string model immediately before an element, attribute, or string is encoded. "Injecting" a symbol means telling the model that it has been seen but not explicitly encoding or decoding it. For lossless decoding, the decoder must be able to infer when a symbol has been injected by the encoder, and also infer the symbol value. Fortunately, enclosing element tag symbols satisfy this property. In our running example <elt att="asdf">XYZ</elt>, the models see:

```
        | <elt | att=             | "asdf"     | >       | XYZ                | </elt>
Elts:   | 10   |                  |            |         | FE                 | FF
Atts:   |      | <10> 0D          | a s d f 00 | <10> FF |                    |
Chars:  |      |                  |            |         |     <10> X Y Z 00  |
Syms:   |      |         a t t 00 |            |         |                    |
```

where <nn> represents reporting a symbol to the model without explicitly encoding or decoding it.

We modified the multiplexed model to perform element symbol injection. We built two versions: "MHM", which used the PPMD+ model with 5 context symbols for all syntax classes, and "MHM*", which used the PPM* model for the element and attribute contexts instead. The results are summarized in Table 2(b).

The MM compressor is about 2% better than ESAX with ppm; MHM is 13% better, and MHM* is 15% better. MHM and MHM* also compare favorably with ESAX/BZ2 and XML/BZ2, ranging from 3–8% better. MHM's chief weakness is on structured documents, where it can compress 20–40% worse than MHM*; on the other hand, MHM* does only 1–4% worse on textual documents. In terms of speed, MM is about the same as ESAX/PPM, MHM is 20% slower, and MHM* is about eight times slower. The slowdown occurs because the compressors process many injected symbols as well as the symbols that are actually transmitted. All three compressors, MM, MHM, and MHM*, run in one pass.

The MHM and MHM* models compress the XML files (individually, and as a whole) better than any other methods of which we are aware. The MHM compressor seems to work best for textual XML with sparse markup, whereas the MHM* compressor does much better on structured documents and slightly worse on textual documents. Table 2(c) compares MHM* with XML compressed with bzip2, the

best previously existing compressor. Table 3 summarizes overall compression results. MHM* compresses better on all documents tested, ranging from a few percent better on textual documents to 10–35% better on structured documents. The biggest problem with both compressors is speed. Since MHM* is only about 5% better overall than ESAX compressed with `bzip2`, it is hard to justify the factor-of-40 slowdown. On the other hand, MHM compresses textual documents better and is only six times slower than BZIP2, but doesn't compress structured documents as well as MHM*.

7 Related and future work

XMLZIP and XMILL are the only XML-specific compressors of which we are aware; undoubtedly others are in development. In the Wireless Access Protocol standard group's Binary XML Content Format (WBXML) [9], XML elements and attribute tags are tokenized with respect to a fixed symbol table. WBXML is similar to ESAX, but packs more information into some bytes, which may hinder compression. Kanne and Moerkotte [11] have addressed storing XML efficiently for database querying.

Both grammar-conscious and grammar-inferring text compression might apply to XML compression, since XML has context-free structure containing unstructured text. Katajainen et al. [12] and Cameron [3] were the first to investigate grammar-based compression; more recently, Lake [14] combined PPM and grammar modeling. Nevill-Manning and Witten [18] and Yang and Kieffer [13] have investigated text compression using grammars learned from the text. our model multiplexing approach resembles stream splitting techniques in machine-code compression (Ernst et al. [8], Lucco[16]).

The MHM model we used was limited to one level of hierarchical context. We observed that more element context helps considerably in compressing structured data, but is harmful in compressing textual data and is slow. Furthermore, compressing structured data well required using the considerably slower PPM*. We believe that it would be worthwhile to redesign existing compressors with both hierarchical and sequential structure in mind, in order to get better compression without sacrificing performance.

Inspired by XMILL's path expression/user compressor language for guiding compression, we speculate that user assistance can also improve our approach. Such assistance might take the form of XMILL-style path expressions or constraints such as DTDs. Knowing constraints enables the model to infer many element or attribute symbols, which then need not be transmitted or received. Conversely, it may be helpful to have "data mining" tools which assist the user in constructing a set of constraints that characterize an XML source. These tools could be used to find constraints that help compress the data; indeed, that could be one measure of how accurate proposed constraints are. This is an area in which we believe there is much room for improvement in XML compression and data mining.

8 Conclusion

We established a working XML compression benchmark based on text compression, and found that `bzip2` compresses XML best, albeit more slowly than `gzip`. Our experiments verified that T_{XMILL} speeds up and improves compression using `gzip`

and bounded-context ppm by up to 15%, but found that it worsens compression for bzip2 and ppm*. We presented an alternative transformation, ESAX, which speeds up and improves compression for all the text compressors we used, compresses 2–4% better than text XML using bzip2, and which allows online processing. Finally, we described a new technique called multiplexed hierarchical modeling that combines existing text compressors and knowledge of XML structure. Using the PPMD+ and PPM* models as components, our MHM and MHM* models compress textual XML data about 5% better and structured data from 10–35% better than the best existing method.

References

[1] J. Bosak and T. Bray. XML and the Second-Generation Web. *Scientific American*, May 1999.

[2] A. Bosworth. Microsoft's Vision for XML. In *SGML/XML Europe*, 1998. http://www.oasis-open.org/cover/bosworthXML98.html.

[3] R. D. Cameron. Source encoding using syntactic information source models. *IEEE Trans. Inform. Theory*, 34(4):843–850, July 1988.

[4] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997.

[5] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm.*, COM-32(4):396–402, 1984.

[6] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification (Second Edition), Version 1.0. http://www.w3.org/TR/REC-DOM-Level-1/.

[7] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006.

[8] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, pages 358–365, June 1997.

[9] Wireless Access Protocol Forum. Binary XML Content Format Specification. http://www.wapforum.org/.

[10] J.-L. Gailly. gzip, version 1.2.4. http://www.gzip.org/.

[11] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering*. IEEE Computer Society Press, March 2000.

[12] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software—Practice and Experience*, 16(3):269–276, March 1986.

[13] J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inform. Theory*, IT-46(3):737–754, 2000.

[14] J. M. Lake. Prediction by grammatical match. In *Proceedings of the 2000 IEEE Data Compression Conference*, pages 153–162. IEEE Computer Society, 2000.

[15] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.

[16] Steven Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 27–34, Vancouver, British Columbia, June 18–21, 2000.

[17] D. Megginson. SAX: A Simple API for XML. http://www.megginson.com/SAX/.

[18] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40(2/3):103–116, 1997.

[19] J. Seward. bzip2, version 0.9.5d. http://sourceware.cygnus.com/bzip2/.

[20] XML Solutions. XMLZIP. http://www.xmls.com/.

[21] W. J. Teahan. PPMD+, PPM* source code. http://www.cs.waikato.ac.nz/~wjt/.