An Empirical Evaluation of Simple DTD-Conscious Compression Techniques

James Cheney
University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

1. INTRODUCTION

The term "XML compression" has been used to describe techniques addressing several different (though related) problems, all relevant to Web data management:

1. *minimum-length coding for efficient XML document storage and transmission* [13, 5, 10, 1];
2. *compact binary formats for efficient (streaming) XML message processing and transmission* [8, 9]; and
3. *storage techniques for efficient XML database query processing* [11, 17, 3, 2].

To avoid ambiguity, in this paper, the term "XML compression" is used in the first (and, we believe, original and most accurate) sense exclusively. We will compare our proposed techniques only with other approaches that address problem (1), not problems (2) or (3).

Since XML markup often displays a high degree of redundancy, ordinary text compressors (`gzip` [7], `bzip2` [15], etc.) are frequently used for XML storage and transmission. Text compressors perform adequately for archiving XML files in many situations; however, they are blind to the underlying structure of the XML document so may miss compression opportunities. Because of this, researchers have studied, and companies have marketed, XML compression tools.

In previous work [5], we developed a streaming XML-conscious compressor `xmlppm`, and showed that it provides compression superior to other contemporary text and XML-conscious compression techniques (including XMill [13]).

The purpose of this paper is to investigate whether DTD information can be used to improve compression in `xmlppm` enough to justify the added implementation effort. We consider the *minimum-length coding problem for valid XML*: Given a data source producing XML conforming to a DTD, find the smallest possible encoding. We assume both sender and receiver have access to identical copies of the DTD.

It appears to be common sense that a DTD, XML Schema, or RELAX/NG schema should be useful in helping to compress conforming XML documents, and many of the above approaches exploit or require an XML Schema. However, we are aware of no rigorous experimental validation of DTD or schema-conscious XML compression in comparison with competitive DTD-unconscious XML compression techniques.

We choose to focus on DTDs exclusively (rather than XML Schema or RELAX/NG schemas) for several reasons: DTDs are simpler, more established, and more widely adopted; DTD parsing is built-in to most XML parsers; DTD validation is easier to implement than for the other approaches; and substantial compression improvements turn out to be possible using DTDs only. Nevertheless, XML Schema and RELAX/NG schemas can provide much more detailed information about documents, especially about their text content. We view generalizing our results to more powerful schema systems as an important future direction.

Because `xmlppm` already compresses both XML structure and text very well, and because of the complexity of the underlying PPM algorithms, it is easy to generate ideas for DTD-based compression that work well "on paper" but are either incompatible with `xmlppm` or do not improve compression relative to `xmlppm`. In this paper we describe `dtdppm`, a version of `xmlppm` that simultaneously validates and compresses XML relative to a DTD. Our main contribution is the development of four simple DTD-based optimizations that techniques that *do* work well with `xmlppm`: *ignorable whitespace stripping, symbol table reuse, element symbol prediction*, and *bitmap-based attribute list coding*. These simple techniques are validated by experiments showing substantial compression benefits for a variety of real data sources.

The structure of the rest of the paper is as follows. Section 2 reviews `xmlppm`. Section 3 presents the DTD-conscious compression techniques used in `dtdppm`. Section 4 presents experimental results, and Section 5 discusses the results. Section 6 concludes.

2. BACKGROUND

In previous work, we developed `xmlppm` [5, 4], an algorithm for XML compression based on Prediction by Partial Match (PPM) [6], one of the most advanced known text compression techniques. PPM compression builds a statistical model of the data seen so far, and uses it to generate a probability distribution predicting the next symbol; the actual symbol is transmitted using arithmetic coding relative to this distribution. In `xmlppm`, an XML file is first parsed using a SAX parser to generate a stream of SAX events. Each event is encoded using a bytecode representation called ESAX. The ESAX bytecodes are encoded using one of several "multiplexed" PPM compressors, for elements, charac-

Eighth International Workshop on the Web and Databases (WebDB 2005)
June 16–17, 2005, Baltimore, Maryland, USA

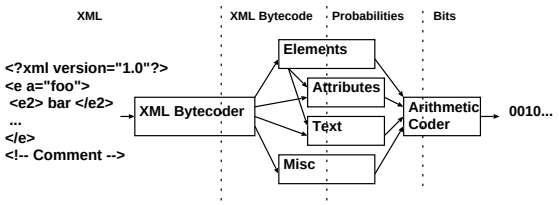

Figure 1: `xmlppm` architecture

ters, attributes, and miscellaneous symbols. The encoder and decoder states are in lockstep so that the decoder always knows which model to use for the next symbol. The architecture of `xmlppm` is shown in Figure 1.

This XML-conscious, multiplexed modeling approach offers several benefits over simply using PPM compression directly on XML text. First, `xmlppm` tokenizes element and attribute names, so less time is spent on low-level PPM compression. Second, text, element, and attribute content in XML have different statistical characteristics, so using different models for each kind of data improves compression. Third, `xmlppm` uses its knowledge of the structure of XML documents to influence the underlying statistical models and improve compression. Specifically, the element, attribute and text PPM models are given "hints" about the surrounding XML element context. (The paper [5] gives full details.)

PPM techniques are among the most advanced known text compression techniques, and many XML documents consist primarily of unstructured text, so it is not surprising that using PPM leads to improved XML compression. However, this level of performance comes at a cost. In the original reported experiments [5], the implementation of `xmlppm` was very slow (ranging from 5–40 times slower than `bzip2`). Since then, the speed of `xmlppm` has been improved considerably by incorporating Shkarin's highly optimized PPMII implementation of PPM [16]. The most recent version of `xmlppm` [4] is still about a factor of 5–10 slower than `gzip`, but is generally as fast as or faster than `bzip2`, while providing better compression.

Table 1 compares `gzip`, `bzip2`, the current implementation of `xmlppm`, and `dtdppm` on the corpus used in [5]. The experimental setup is described in Section 4. The XML column lists the size of the input files in bytes; the other columns measure compression rate in bits per input character (bpc) for each compressor. Times are in seconds. The final column % shows the percentage change in compression rate or execution time for `dtdppm` vs. `xmlppm` (that is, $((X-D)/X)\%$, where X is the rate/time for `xmlppm` and D the rate/time for `dtdppm`). The results are discussed further in Section 4.1.

3. COMPRESSION TECHNIQUES

We modified the current version of `xmlppm` to read in and validate its input relative to a DTD: the resulting DTD-conscious compressor is called `dtdppm`. In `dtdppm`, the DTD validation state is available for use during compression, so DTD-specific optimizations are possible. We have implemented four DTD-based optimizations: ignorable whitespace stripping, symbol table reuse, element symbol prediction, and bitmap-based attribute list coding.

	XML (bytes)	`gzip` (bpc)	`bzip2` (bpc)	`xmlppm` (bpc)	`dtdppm` (bpc)	change %
elts	113181	0.622	0.416	0.375	0.327	13%
pcc1	51647	0.557	0.444	0.301	0.229	24%
pcc2	262669	0.320	0.183	0.145	0.123	15%
pcc3	186857	0.374	0.227	0.167	0.144	14%
play1	251898	2.160	1.549	1.449	1.453	−0.21%
play2	136841	2.141	1.630	1.473	1.464	0.62%
play3	279703	2.267	1.647	1.569	1.566	0.18%
sprot	10268	2.056	2.048	1.692	1.430	16%
stats1	669347	0.798	0.369	0.294	0.282	4.2%
stats2	616094	0.750	0.338	0.272	0.258	5.3%
tal1	734590	0.313	0.123	0.117	0.102	13%
tal2	510111	0.322	0.151	0.127	0.107	16%
tal3	251698	0.330	0.198	0.152	0.125	18%
tpc	287992	1.476	1.101	1.007	0.980	2.6%
tree	6704	1.734	1.494	1.104	0.715	35%
w3c1	220794	1.888	1.464	1.302	1.275	2.0%
w3c2	196233	1.947	1.534	1.365	1.338	2.0%
w3c3	201849	2.139	1.722	1.530	1.471	3.8%
w3c4	104938	1.804	1.521	1.333	1.280	4.0%
w3c5	247465	1.823	1.435	1.336	1.287	3.7%
weblog	2304	2.160	2.559	1.747	1.420	18%
total	5343183	1.035	0.701	0.625	0.603	3.6%
time(s)		0.61	2.38	1.89	2.50	−32%

Table 1: XMLPPM corpus

3.1 Ignorable whitespace stripping

Much of the whitespace in an XML document is irrelevant to the data being represented: for example, whitespace is often used only as a visual cue to the document's hierarchical structure. We call this whitespace *ignorable*. Our implementation attempts to drop ignorable whitespace whenever possible. PPM algorithms typically use up to 10 of the most recent characters as context to predict the next symbol. Compressing long sequences of whitespace flushes the context, so the model is unprepared for whatever comes next. Therefore, though it may seem trivial, whitespace stripping is crucial for good PPM compression performance because it helps prevent PPM models from losing track of context.

It is not generally safe to drop whitespace in the absence of a DTD, since whitespace is significant in some elements (e.g., `<xsl:text>`). However, in the presence of a DTD, whitespace can usually be safely ignored whenever it occurs inside an element whose content model does not mention `#PCDATA`. `dtdppm` tests for this whenever character data is encountered, and ignorable whitespace is dropped. When documents with ignored whitespace are decompressed, `dtdppm` optionally inserts newlines and indentation so that the resulting document will be human-readable rather than one long line.

Nevertheless, there may be documents with whitespace that is ignorable by our definition but which users wish to preserve, so whitespace stripping is optional.

3.2 Symbol table reuse

In `xmlppm`, element and attribute tags (and some other kinds of symbols) are replaced by symbol table references. However, in the absence of a DTD, the symbol table needs to be built dynamically by the encoder and decoder, so the text for each symbol is sent when it is first encountered in the document. In `dtdppm`, the DTD is available to both encoder and decoder, and it is not necessary to transmit the

symbols inline. Instead, both encoder and decoder can refer to a symbol table built from the DTD.

The savings from this optimization are not dramatic, since the DTD may be much smaller then the document; however, symbol table reuse is important in a situation in which many small documents are to be compressed, especially since ordinary compression techniques are less effective for smaller documents.

3.3 Element symbol prediction

In the absence of a DTD, any element tag can, in principle, occur anywhere in the document. However, in valid XML, elements may have regular expression content models

```
<!ELEMENT foo (bar,(baz|bar)*)>
```

that constrain the children of the element. For highly structured data, frequently there is only one possible next element symbol; this can be determined by inspecting the state of the DTD validator. When this is the case, the `dtdppm` encoder omits the element bytecode since the decoder can infer the next element symbol from context. Similarly, it is possible to test whether the remaining content model is empty. In this case, the "end-element" bytecode that would ordinarily be sent is omitted because the decoder can infer it from context.

This technique may seem trivial since it does not do anything special other than to omit symbols that can be predicted from context. However, more sophisticated techniques seem to interact badly with PPM. For example, in approaches like those of Levene and Wood [12] or XComprez [10], element content sequences are encoded in a more sophisticated way that is dependent on the remaining regular expression content model. We experimented with a simple form of this approach in `dtdppm`, but found that it does not help much relative to ordinary `xmlppm` compression. This is because PPM compresses byte-aligned text, so using non-byte-aligned encodings for element symbols confuses the underlying PPM model. Nevertheless, this is definitely an area where improvement may be possible. Combining sophisticated regular expression coding techniques with PPM compression is a challenge left for future work.

3.4 Bitmap-based attribute list encoding

Attribute list declarations

```
<!ATTLIST elt att1 TYPE1 DFLT1 att2 TYPE2 DFLT2 ...>
```

are one of the most complicated features of DTDs. Attribute values can have one of several types `TYPE`:

`CDATA`	Arbitrary text
`ID`	Globally unique, can't be `FIXED`
`IDREF(S)`	Must refer to an `ID`
`NMTOKEN(S)`	Must be a name token
`ENTITY(IES)`	Must be a declared `ENTITY`
`NOTATION` $(v_1\|\cdots\|v_n)$	Enumerated, declared `NOTATION`
$(v_1\|\cdots\|v_n)$	Enumerated type

Attribute types `IDREF`, `NMTOKEN`, and `ENTITY` can be plural. Attributes can also have several default specifications `DFLT`:

`"dflt"`	Default value is `"dflt"`
`#FIXED "dflt"`	Must be present and equal `"dflt"`
`#REQUIRED`	Value must be present
`#IMPLIED`	May be absent, no default

In XML, the order of attribute-value pairs is irrelevant; thus, we may rearrange the attribute list if doing so improves compression. In `dtdppm`, we encode attribute lists by sending a (byte-aligned) bitmap indicating which attributes are present, then sending the attribute values. Default and type constraints are used to avoid sending redundant information.

The details of the encoding are as follows. First, the attribute list is scanned in order to build a bit vector. This bit vector says which attribute values are going to be sent next. `#FIXED` and `#REQUIRED` attributes are not included, since they must be present in a valid document. For `#IMPLIED` attributes, 1 indicates present, 0 absent. For attributes with a default value, 1 indicates that the value is non-default, 0 otherwise. Subsequently, the values of `#REQUIRED` attributes and other attributes whose bitmap value is 1 are transmitted. The values of attributes with enumerated types $(v_0|\cdots|v_n)$ are encoded as bytecodes $0,\ldots,n$.

For example, given

```
<!ATTLIST elt att1 CDATA #FIXED "foo"
              att2 (x|y|z) #REQUIRED
              att3 CDATA #IMPLIED
              att4 NMTOKEN "bar">
```

the encoding of the attribute list of

```
<elt att1="foo" att2="y" att4="baz">
```

is `40 01 'b' 'a' 'z' 00`. The top two bits of the first byte ($40_{16} = 01000000_2$) code the absence of `att3` and (non-default) presence of `att4`; `01` codes enumerated value `y`; and the non-default value `baz` of `att4` is transmitted as a null-terminated string.

There are many possible variations on this theme. We initially tried two simpler ideas, based on adjacency lists and vector representations of attribute lists, each of which worked well for some examples but not for others; the bitmap approach combines the advantages of the two approaches.

4. EVALUATION

The design of a corpus for testing compression techniques can be a subtle issue, because of the possibility of accidental bias towards one or another kind of data. So far, no standard corpus for XML compression (let alone DTD/schema-conscious compression) has emerged. Desirable characteristics of test data include that the data (and DTDs) be freely available online, that there are nontrivial amounts of data (whole documents instead of short examples), that the data is actually valid relative to the DTD, and finally, that the data be "realistic" (i.e., not random or arbitrary). Obviously many of these criteria are subjective. Unfortunately, it is not easy to find data sources having all these characteristics.

We have evaluated `dtdppm` on five corpora:

- The XMLPPM corpus [5]
- Short documents (NewsML)
- Medium structured application data (MusicXML)
- Medium flat datasets (UW XML repository)
- Large datasets (DBLP, Medline, PSD, XMark)

We are aware of other collections of valid XML, such as the Niagara experimental data[1] but have not had time to experiment with these other sources.

[1] `http://www.cs.wisc.edu/niagara/data.html`

Our experiments were performed on an AMD Athlon 64 3000+ (1.8Ghz clock speed) with 512MB RAM, running Red Hat Fedora Core 3. We report compression in bits per character (relative to the original XML input) and total compression time for `gzip`, `bzip2`, `xmlppm`, and `dtdppm` for each data source. (For PPM techniques decompression takes the same time as compression.) The PPM models used by `xmlppm` and `dtdppm` are order 5 models with 1MB of working memory per model (for a total of 4MB). We also benchmarked the current version of XMill[2], and found that, as in [5], it compresses no better than `bzip2` but runs up to three times faster. Because of limited space, these results are omitted.

In addition, we compared the effectiveness of the individual compression techniques, and found that no single technique was dominant. Space limits preclude a full discussion.

4.1 The XMLPPM corpus

For comparison with previous work, we evaluated the performance of `dtdppm` using the same data[3] used by [5]. This corpus contains XML files ranging from small (1KB) to large (700KB), and including both highly textual data and highly structured data. DTDs for each file were either constructed by hand or obtained online. Some errors and inaccuracies in existing DTDs were corrected.

The realism of this benchmark is debatable; its chief virtue is variety. Also, results for this benchmark may be skewed since we constructed some of the DTDs ourselves (with compression in mind), rather than using given DTDs.

Nevertheless, the results (Table 1) do indicate that DTD-conscious compression can be worthwhile for a variety of kinds of XML. In particular, small XML fragments (`sprot`, from SwissPROT; `tree`, from Penn Treebank; and `weblog`, a web log excerpt) exhibit substantial improvements of 16–35%, and large, highly-structured files (`elts`, periodic table data; `pcc1-3`, formal proofs; `tal1-3`, typed assembly language files) improve 13–24%. On the other hand, examples with a lot of text or very regular structure (`play1-3`, Shakespeare plays; `stats1-2`, baseball statistics; `tpc`, TPC benchmark data; `w3c1-5`, W3C standards) did not compress significantly better (0-5% improvement); one example compresses 0.14% worse. For these examples, `xmlppm` already compresses regular structure well and the DTDs provide no information that would help improve text compression. As with most of our examples, `dtdppm` ran slightly slower than `xmlppm`.

4.2 Short documents

NewsML[4] is an XML dialect designed for news articles from press services (e.g. Reuters). The 80KB NewsML DTD defines a NewsML document as some metadata and uses XHTML for the article content (another 56KB). We obtained the DTD and a collection of 246 example NewsML articles, ranging from 6.5–18.2KB (average size 11.2KB). The compression results for the NewsML data are summarized in Table 2. The "NewsML" line shows the compression rates over the entire corpus; the "time" line shows the total compression time.

These results suggest that NewsML documents benefit substantially from DTD-conscious compression, largely, we believe, due to symbol table reuse. Both `xmlppm` and `dtdppm` are considerably slower than `bzip2` in this case; reparsing the 136KB of DTD files accounts for roughly 55% of `dtdppm` running time. This overhead could be alleviated by specializing the compressor to the DTD.

	`gzip` (bpc)	`bzip2` (bpc)	`xmlppm` (bpc)	`dtdppm` (bpc)	change %
NewsML	2.292	2.241	1.982	1.484	25%
time(s)	0.38	2.19	1.54	6.11	−300%
MusicXML	0.304	0.216	0.223	0.127	43%
time(s)	0.10	1.78	0.57	0.77	−35%

Table 2: NewsML and MusicXML results

4.3 Medium structured application data

XML is becoming a widespread format for storing application data: for example, recent versions of popular office suites either store application data as XML directly, or offer the ability to export data in XML. However, standard DTDs for such data are not always available, stable, or heeded.

MusicXML[5] is an XML dialect for representing music. MusicXML documents can be translated to a sheet music PDF file of either all parts or a single part, as well as to a MIDI file that can be played directly on a synthesizer or further processed using sequencing software. We obtained the MusicXML DTD files (106KB total) and 18 example MusicXML files ranging from 8.8–230KB (101KB average), each corresponding to one or two sheets of a musical score. The compression results for MusicXML are shown in Table 2. The best compression is obtained by `dtdppm`; the average improvement is 43%. Note that plain `xmlppm` generally compresses MusicXML slightly worse than `bzip2`, but both `xmlppm` and `dtdppm` are slightly faster.

The MusicXML web page claims that `gzip`-compressed MusicXML documents are only about twice as large as equivalent documents in MuseData, a custom format. Since `dtdppm` compresses MusicXML 58% better than `gzip` on average, this suggests `dtdppm` is competitive with a hand-coded binary format.

4.4 Medium flat datasets

XML is sometimes used to export, or *publish*, the data in a relational table or database, often with some added structure. The UW XML repository[6] includes several example XML data sources, many of which consist of a flat sequence of elements with identical structure. Unfortunately, many of these examples do not possess DTDs, or are not valid. We chose several medium-sized examples that do have DTDs and are valid to evaluate `dtdppm` for such data.

This situation seems to offer great promise, since the DTD tells us almost everything we need to know about the structure of the data: only a few details (such as the number of rows) need to be filled in. However, data with very regular structure already compresses very well using plain XML-conscious compression techniques, because the DTD only tells the compressor things it learns quickly for itself. As a result, the amount of improvement that can be expected for such data is limited. On the other hand, many of the medium-sized files make liberal use of whitespace for readability. As a result, some improvement to compression re-

[2] `http://www.cs.washington.edu/homes/suciu/XMILL/`
[3] available at `http://xmlppm.sourceforge.net/`
[4] `http://www.newsml.org`
[5] `http://www.recordare.com/xml.html`
[6] `http://www.cs.washington.edu/research/xmldatasets/`

	XML (bytes)	gzip (bpc)	bzip2 (bpc)	xmlppm (bpc)	dtdppm (bpc)	change %
321gone	24442	2.213	2.228	1.884	1.741	7.5%
cornell	30979	1.026	0.954	0.880	0.732	17%
ebay	35472	2.480	2.580	2.189	2.103	3.9%
reed	283582	0.533	0.332	0.327	0.274	16%
SigRec	478337	1.363	0.812	0.802	0.745	7.1%
ubid	20246	1.494	1.515	1.296	1.114	14%
wash	3068693	0.525	0.317	0.390	0.275	30%
yahoo	25347	1.971	1.903	1.620	1.470	9.3%
total	3967098	0.673	0.431	0.477	0.372	22%
time(s)		0.28	3.21	1.17	1.65	−41%

Table 3: Medium flat file benchmark results

sulting from whitespace stripping is to be expected.

The experimental results are shown in Table 3. In a few examples (reed, wash, course information; cornell, personnel records; ubid, auction data), dtdppm achieves a substantial improvement because of whitespace stripping, improving compression substantially (14–30%). For the other examples (321gone, ebay, yahoo, auction data; SigRec, bibliographic records) improvement was in the more modest 4–9% range. Overall compression improved 22% relative to xmlppm (mostly because of wash). For this dataset, bzip2 compressed better than xmlppm, but dtdppm performed best overall. Perhaps surprisingly, both xmlppm and dtdppm were 2–2.5 times as fast as bzip2.

4.5 Large datasets

Another increasingly common scenario is the use of XML as a format for serializing large databases. Examples include scientific databases like SwissPROT/UniPROT and the Georgetown Protein Sequence Database and bibliographic databases like DBLP and Medline. These databases are typically made available on the Web and updated at intervals ranging from daily to yearly. Because of their size, scalable and effective compression is very important.

Another large dataset example is the data generated by XMark. The XMark benchmark [14] has been proposed as a means for comparing the performance of XML databases. It consists of a DTD for auction data and a data generator which generates a random valid document of size proportional to a given "scaling factor".

In Table 4, we present the results of compressing four large datasets: xmark is an example XMark file[7], medline is one part of the PubMed database[8], psd is a file from the Georgetown Protein Sequence Database, and dblp is an XML serialization of the DBLP database. The psd and dblp examples were obtained from the UW XML repository.

The results vary. For xmark, dtdppm and xmlppm compress 7.7% worse than bzip2. This is the only example in the paper for which dtdppm is not competitive (i.e., within 1% of the best). For other examples, dtdppm's compression is competitive or best. However, xmark data may not be a realistic compression benchmark because it is randomly generated. The DTDs for these documents do not provide many opportunities to predict a unique next symbol, so the behavior of dtdppm is essentially the same as xmlppm. Also, all of these documents use whitespace only trivially (i.e., each element tag is on its own line, but there is no indenting), so

[7] http://www.xml-benchmark.org

[8] http://www.ncbi.nlm.nih.gov/

	XML (bytes)	gzip (bpc)	bzip2 (bpc)	xmlppm (bpc)	dtdppm (bpc)	change %
xmark	116MB	2.616	1.754	1.888	1.889	−0.02%
time(s)		13.4	46.3	39.1	39.5	−0.8%
medline	127MB	1.278	0.888	0.841	0.838	0.4%
time		7.5	65.1	32.3	33.0	−2.2%
psd	717MB	1.209	0.857	0.867	0.846	2.5%
time		33.9	389.7	169.3	170.0	−0.4%
dblp	103MB	1.479	0.963	0.940	0.947	−0.8%
time(s)		6.9	48.8	27.3	28.2	−3.2%

Table 4: Large benchmark results

whitespace stripping has little effect. Compression time is also similar to xmlppm in most cases, although interestingly dtdppm and xmlppm are generally 15–60% faster than bzip2.

5. DISCUSSION

There are several lessons that can be learned from our experiments. DTD-conscious compression (as embodied in dtdppm) is very effective for small messages, highly-structured documents, or documents with large amounts of formatting whitespace. For large datasets, dtdppm does not compress significantly better than xmlppm; however, both xmlppm and dtdppm compress as well as or better than bzip2 but 30-50% faster. xmlppm will probably never be as fast as gzip, but xmlppm and dtdppm compress significantly better than gzip while staying within an order of magnitude of gzip's speed.

Another observation is that DTDs can be well- or ill-suited for compression. For example, in stats2 (baseball statistics), the DTD says that each player element has a sequence of optional sub-elements $(e_1^?, \ldots, e_n^?)$, but the actual data exhibits only two instances of this content model. Similarly, common content models like $(e_1|\cdots|e_n)^*$ do not provide any information that helps dtdppm. Finding ways to take advantage of such content models is an important area for future work, especially since the same kind of techniques may be useful for compressing regular expression-typed text in XML Schema.

Another problem is that XML's data model is ordered, whereas many data sources (e.g. relation fields, semistructured data trees, or BibTeX records) are conceptually unordered. DTDs cannot express unordered content models efficiently so the content model $(e_1|\cdots|e_n)^*$ is often used as an approximation. Other schema systems such as ASN.1, RELAX/NG and XML Schema do provide unordered content models, but it is not obvious how to compress with respect to such content models effectively. One possibility is to sort content in unordered content models so as to place it in a normal form, as with attribute lists; however, this transformation is non-streaming.

XML encourages a structured approach to data management, but this approach is usually followed only up to a point. A typical example of the use of low-level character data formats is the use of date strings Mar 15 17:55 instead of XML markup

```
<date><month>Mar</month><day>15</day>
      <time><hour>17</hour><min>55</min></date>
```

The former representation is briefer and more human-readable, but xmlppm will likely compress the latter much better. XML Schema's datatypes (especially dates) may be useful for improving compression for this kind of data.

6. RELATED WORK

Liefke and Suciu's XMill [13] is probably the best known XML compressor. One interesting aspect of XMill is that it allows user-defined container specifications using XPath expressions to define containers and to specify datatype-specific compressors. This can significantly improve compression, but may require nontrivial user effort. It is possible that XMill could be made schema-conscious by automatically generating specifications from schemas.

Levene and Wood [12] propose DTD-based encodings for XML data in which the encoding is dependent on the current content model. For example, content matching $r|s$ is encoded by sending 0 if the content matches r, or 1 if it matches s, then encoding the content relative to r or s respectively. This encoding has not been implemented as far as we know; also, although Levene and Wood prove an optimality result, it rests on very strong assumptions (data must conform to a nonrecursive DTD and be generated by independent random choices). This is a step in the right direction, but more theoretical understanding is needed.

Jeuring and Hagg [10] have developed XComprez, which compresses valid XML using an encoding similar to that of Levene and Wood. They use a powerful experimental programming language called Generic Haskell in which the compressor constitutes approximately 650 lines of code (in contrast to 4300 lines of C++ code for `xmlppm` and 9000 lines for `dtdppm`). It is not yet clear whether this approach scales to large XML documents, but advanced programming tools like Generic Haskell may make it easier to rapidly prototype compression techniques prior to full-scale implementation.

SCMPPM [1] is an XMLPPM variant that uses a separate PPM model to compress the text content under each element. It achieves reported improvements of 20% over plain XMLPPM when compressing large TREC datasets. However, it has not been evaluated on other data, so this result must be taken with a grain of salt.

7. FUTURE WORK

As stated in the introduction, we view `dtdppm` as the first step in a logical progression to RELAX/NG- and XML schema-conscious compression tools. In particular, RELAX/NG seems like a logical next step because it is not much more complicated than DTD yet supports datatypes for text content. As for XML Schema, we are intrigued by the possibility of compressing text relative to arbitrary regular expressions. However, we suspect that obvious ways of doing this will not be as effective as plain PPM or `xmlppm`; instead, we intend to find a way to combine PPM and regular expression-based modeling. If such an approach can be found, we believe it will also help with element content compression.

Finally, the prototype implementation[9] has a few bugs that need to be fixed, and several worthwhile optimizations appear possible (particularly pre-compiling or specializing `dtdppm` to a DTD).

8. CONCLUSIONS

The purpose of this paper was to determine whether DTD-conscious compression techniques offer enough benefits, relative to state-or-the-art XML compression, to be worth the (nontrivial) implementation effort needed. We implemented a validating compressor, `dtdppm`, which reads in a DTD and XML document and simultaneously validates and compresses it. In addition, `dtdppm` performs several optimizations on the encoding which are only possible in the presence of a DTD. Put together, these optimizations can improve compression by up to 43% over `xmlppm`. While `dtdppm` can be very effective for small or highly-structured documents, it may not compress unstructured, mostly-text, or large documents significantly better than `xmlppm`. Nevertheless, we found `xmlppm` and `dtdppm` compress large documents as well as `bzip2` but significantly (15-60%) faster.

We believe that this is the first comprehensive assessment of a DTD-conscious XML compression tool.

[9] `http://xmlppm.sourceforge.net/dtdppm`

9. REFERENCES

[1] J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Proc. 2004 IEEE Data Compression Conference (DCC'04)*, page 522, 2004.

[2] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis Viglas. Vectorizing and querying large XML repositories. In *Proc. 21st Int. Conference on Data Engineering (ICDE 2005)*, 2005. To appear.

[3] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Int. Conference on Very Large Data Bases (VLDB'03)*, pages 141–152, 2003.

[4] J. Cheney. `xmlppm`, version 0.98.2. `http://xmlppm.sourceforge.net/`.

[5] James Cheney. Compressing XML with multiplexed hierarchical models. In *Proc. 2001 IEEE Data Compression Conference (DCC 2001)*, pages 163–172. IEEE, 2001.

[6] 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.

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

[8] Marc Girardot and Neel Sundaresan. Millau: An encoding format for efficient representation and exchange of XML over the web. *Computer Networks*, 33(1–6):747–765, 2000.

[9] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *Transactions on Database Systems*, 29(4), December 2004.

[10] Johan Jeuring and Paul Hagg. Generic programming for XML tools. Technical Report UU-CS-2002-023, Utrecht University, 2002.

[11] W.Y. Lam, W. Ng, P.T. Wood, and M. Levene. XCQ: XML compression and querying system. In *Proc. 12th Int. Conference on the World Wide Web (WWW 2003)*, 2003.

[12] M. Levene and P. T. Wood. XML structure compression. In *Proc. 2nd Int. Workshop on Web Dynamics*, 2002.

[13] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *SIGMOD '00: Proc. 2000 ACM SIGMOD international conference on management of data*, pages 153–164. ACM Press, 2000.

[14] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, 2001.

[15] J. Seward. `bzip2`, version 0.9.5d. `http://sources.redhat.com/bzip2/`.

[16] Dmitry Shkarin. PPM: One step to practicality. In *Proc. 12th IEEE Data Compression Conference*, pages 202–211, 2002.

[17] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. 18th Int. Conference on Data Engineering (ICDE'02)*, pages 225–234. IEEE, 2002.