An Extensible Compressor for XML Data

Hartmut Liefke*
Univ. of Pennsylvania
liefke@seas.upenn.edu

Dan Suciu
AT&T Labs
suciu@research.att.com

1 Introduction

We have implemented a compressor (`XMill`) and decompressor (`XDemill`) for XML data, to be used in data exchange and archiving, which can be downloaded from http://www.research.att.com/sw/tools/xmill. `XMill` compresses about twice as good as `gzip`, at about the same speed. It does not need a DTD in order to compress, and preserves the input XML file faithfully, including element order, attributes order, PI's, comments, the DTD, etc. A novelty in `XMill` is that it allows users to combine existing compressors in order to compress heterogeneous XML data: by default it uses `zlib`, a library function implementing `gzip`'s functionality, and includes some standard compression techniques for simple data types. Further, `XMill` is extensible with user-defined compressors for complex data types, such as DNA sequences or images. This paper describes `XMill`'s extensible architecture; it also summarizes some other aspects of `XMill` from [3], such as container expressions and an experimental evaluation.

There are three principles in `XMill`:

Separate structure from data The *structure* consists of XML tags and attributes; it forms a tree. The *data* consists of a sequence of items (strings) representing element text contents and attribute values. The structure and the data are compressed separately.

Group data items with related meaning Data items are grouped into *containers*, and each container is compressed separately. By exploiting similarities between the values in a container, the compression improves substantially. By default, `XMill` groups data items based on the element type, but users can override that through *container expressions*.

Apply semantic compressors to containers Some data items are text, others are numbers, others are dates, etc. `XMill` applies different specialized compressors (*semantic compressors*) to different containers.

While experimenting with `XMill` we made an interesting discovery. A lot of data is available today in ASCII files, in various application specific data format: Web logs, IP traffic logs, biological data, linguistic data, etc. These data instances are good candidates to be migrated to XML, in order to gain flexibility and to benefit from the large number of available XML tools. But the size of the XML file is larger (we observed increases from 30% to 200%), and remains still larger when we compare the `gzip`-ed files. This is a known fact, and a common objection to migrating these formats to XML. We discovered however that `XMill` shrinks the XML file to less than `gzip` reduces the original file (up to half the size in some cases). Thus, `XMill` offers an incentive to migrate data from application specific formats to XML.

The rest of the paper is organized as follows: Sec. 2 describes an example of migrating data to XML, Sec. 3 describes `XMill`'s architecture, showing how the structure is stored, how the data items are separated, describing the semantic compressors. Sec. 4 describes how `XMill` can be extended by users with new semantic compressors. Sec. 5 is a brief overview of several experiments on real data sets.

2 Migrating Data to XML: An Example

We will illustrate with Web server log data, which we will use as a running example. Virtually every Web server logs its traffic, for security purposes, and this data can be (and often is) analyzed. The log file is an ASCII file in which each line represents an HTTP request, e.g.[1]:

```
202.239.238.16|GET / HTTP/1.0|text/html|200|
1997/10/01-00:00:02|-|4478|-|-|
http://www.so-net.jp/|Mozilla/3.0 [ja]
```

*This work was done while the author was visiting AT&T Labs.

[1] This is one line in the log file.

Different formats are currently in use: in our example we use a variation on Apache's *Custom Log Format*[2]. Each line is a record with eleven variable-length fields delimited by |: host, request line, content type, etc. Missing values are indicated by -. Different web servers however can be configured to log different sets of fields, or in a different format. This, of course, makes applications processing Web log data brittle and non-portable.

The same data can be XML-ized, e.g. like that:

```
<apache:entry>
 <apache:host> 202.239.238.16 </host>
 <apache:requestLine>GET / HTTP/1.0</apache:requestLine>
 <apache:contentType> text/html </apache:contentType>
 <apache:statusCode> 200 </apache:statusCode>
 <apache:date> 1997/10/01-00:00:02 </apache:date>
 <apache:byteCount> 4478 </apache:byteCount>
 <apache:referer>http://www.so-net.jp/</apache:referer>
 <apache:userAgent> Mozilla/3.0 [ja] </apache:userAgent>
</apache:entry>
```

The data is now self-descriptive, and missing values are simply dropped (only nine fields are present). Fields can be added or removed at will, and applications consuming such data are more robust. The size of the log, however, increases: we observed a 200% increase from the ASCII file to the XML file and a 40% increase from the `gzip`-ed ASCII file to the `gzip`-ed XML file. This is an argument against migrating Web log data into an XML format. With `XMill` however, one can compress the XML file to about half the size of the `gzip`-ed ASCII file (Sec. 5).

3 The Architecture of `XMill`

`XMill` compresses XML data by applying the three principles in Sec. 1. Its architecture is shown in Fig. 1. The XML file is parsed by an SAX[3] parser that sends tokens to the *path processor*, the main module in `XMill`. The purpose of the path processor is to separate the structure from the data, and to further separate the data items according to their semantics (Principles I and II).

To illustrate with our example, the *structure* consists of the XML tags only (details in Sec. 3.1), while the *data items* are the text inside the XML tags: all host values (203.237.165.15, 203.172.22.2, ...), all requestLine values, etc. The structure will be stored in one container, while the data items are stored in separate data containers, one for each element type.

Next, the structure container and all data containers are compressed with `gzip`[4] before being sent to disk. In addition, each data item may be compressed with a semantic compressor (discussed in Sec. 3.3), before being stored in its container (Principle III). These semantic compressors are specified by the user on the command line: by default, no semantic compressor is applied (the container will be compressed with `gzip` anyway). For example, the user may know that most values of the host element are IP addresses, and instruct `XMill` to compress them as four unsigned bytes.

In XML, elements may be nested to arbitrary depth. By default, `XMill` groups data items based on their innermost element type. Users however can override this, by providing *container expressions* on the command line (Sec. 3.2). The path processor uses these expressions to determine in which container to store each data item. Path expressions also determine which semantic compressor to apply (if any).

The amount of main memory holding all containers is fixed to 8MB.[5] After this limit is exceeded by storing the next data item, the containers are `gzip`-ed, sent to disk, then the compression resumes. In effect, this partitions the input XML file into blocks which are compressed independently.

The decompressor, `XDemill`, is similar, but proceeds in reverse. It reads one block at a time in main memory, decompresses every container, then merges the XML tags with the data values to produce the XML output.

3.1 Separating the Structure

The *structure* is defined to be the XML file with all text values and attribute values removed, and replaced with their container number. Start-tags are dictionary-encoded, i.e. assigned an integer value, while all end-tags are replaced by the same, unique token. For illustration purposes, we denote start-tags with T1, T2, ..., the unique end-tag with /, and container numbers with C1, C2, ... For our running example the structure of the first entry is:

```
T1 T2 C1 / T3 C2 / T4 C3 / T5 C4 / T6 C5 / T8 C7 /
T11 C10 / T12 C11 / /
```

Here T1= apache:entry, T2= apache:host, and so on, while / represents any end tag. Internally, each token is encoded as an integer: tags are positive, container numbers are negative, and \ is 0. Numbers between

[2] `http://www.apache.org/docs/mod/mod_log_config.html`
[3] SAX stands for *Simple API for XML*, `http://www.megginson.com/SAX/`.

[4] In fact with the `zlib` library function, but we will refer to it as `gzip` throughout the paper.
[5] The limit is adjustable by the user. The effect of varying window sizes on compression rate is studied in [3].

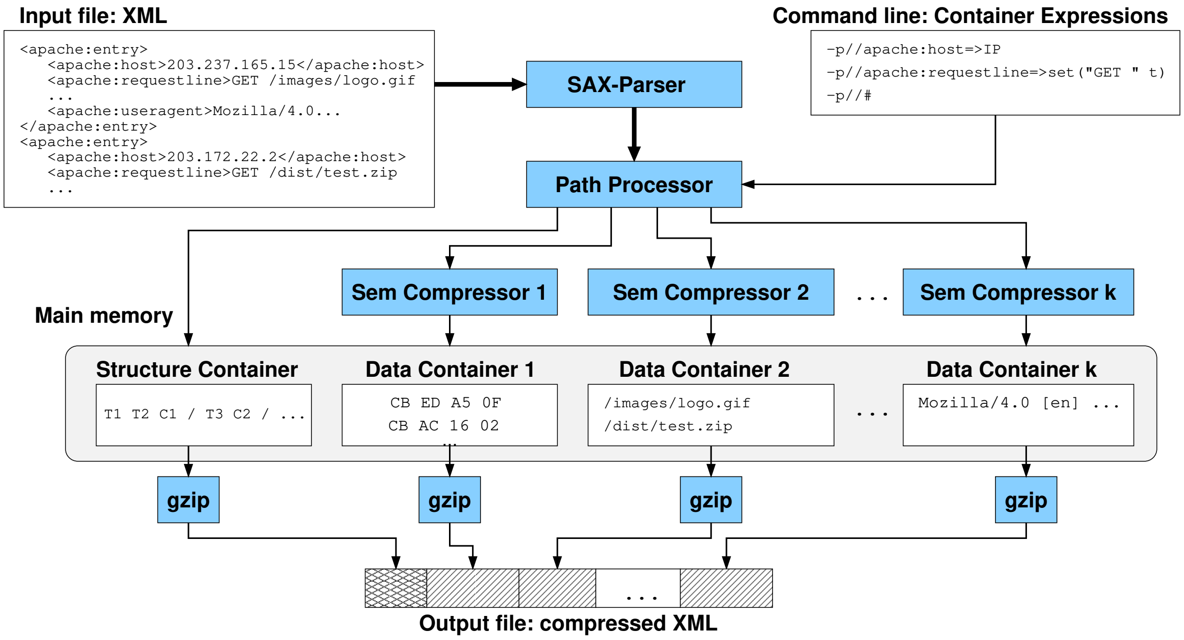

Figure 1: Architecture of the Compressor

-64, 63 take one byte, others take two or four bytes: the string above takes 26 bytes.

By default white spaces are ignored, and the decompressor will produce some standard formatting. The user can instruct `XMill` to treat spaces as normal text, so that the decompressor can reconstruct an identical XML output. The size of the compressed file typically increases only slightly when white spaces are preserved: around 4% in our running example. In the rest of the paper we will assume that white spaces are ignored.

Regular XML data tends to repeat items with similar structure. `gzip`, which is based on the Ziv-Lempel algorithm LZ77[5], compresses such repeated sequences in the structure container extremely well. The compressed structure usually amounts to 1%-3% of the compressed file.

3.2 Grouping Data Items

We describe now how data items are mapped to containers, before being compressed. By default `XMill` creates one container for every tag (element type) in the XML file, and stores each data item according to its element type. The user can override this default and describe a different mapping of data items to containers by specifying *container expressions* in the command line: this may result in better compression. For example consider an XML file about products whose produce description (subelement `<descr>`) is either in English or German, depending whether the product is under an `<usa>` element or a `<germany>` element. One improves compression by mapping the descriptions in two distinct containers. The command line for that contains two container expressions:

```
xmill -p //usa//descr -p //germany//descr file.xml
```

Container expressions are based on XPath [1]: `//usa//descr` denotes any `<descr>` element contained in some `<usa>` element. `XMill` creates two distinguished containers corresponding to the two command line expressions, and stores there each data item matching that expression. All other data items are stored according to their element tag.[6]

The container expressions only need to be specified to the compressor, not to the decompressor.

Overriding the default grouping of data items usually results in only modest improvements in the compression ratio. Much better improvements are achieved with semantic compressors, described next.

3.3 Semantic Compressors

XML files encoding data, as opposed to structured documents, often contain strings representing certain simple data types: e.g. IP addresses, integers, ennumeration values (like US states). For these, simple semantic compressors can be applied in `XMill`, and results in significant improvements in compression.

[6] This is the default and described by container expression //#. Each substitution of # represents a different container. A complete description of container expressions is in [3].

Semantic compressors are specified on the command line with the container expressions. For our Web log example in Sec. 2, the command line:

```
xmill -p //apache:code=>u  file.xml
```

says that data items under `<apache:code>` are to be stored as unsigned integers (u is the name of the unsigned integer compressor): only one byte is used for numbers less than 128 and two bytes or four bytes for larger numbers, and all are stored in a separate container. If some `<apache:code>` data item does not parse as an unsigned integer, then it will be stored in the default text container for this tag. Thus, no XML file is ever rejected, but the user's hint is only useful if many `<apache:code>` values are unsigned integers. XMill predefines a few simple semantic compressors, such as: integer encoder (name i), dictionary encoder (e - for enumeration), run-length (rl) and delta (di) encoder. New ones can be defined by users, as we describe shortly.

Some XML elements contain structured data types. For example `<apache:host>` contains four dot-separated unsigned integers. XMill provides compressor combinators to handle such data values. The *sequence combinator* seq(...) divides the string into substrings and compresses them separately. For example, seq(u8 "." u8 "." u8 "." u8) compresses an IP address as four integers.[7] The *alternate combinator* or(s1 s2 ...) tries each of the compressors s1, s2, ..., until one of them accepts the string. Finally, the *repetition combinator* rep(d s) compresses substrings separated by delimiter string d using compressor s.

4 Extending XMill

Users can specify their own, application-specific compressors (and compressor combinators!) in XMill through a special interface called SCAPI (Semantic Compressor API). Both XMill and XDemill need to be recompiled, resulting in an extended compressor/decompressor pair. This is very useful in applications where complex domain-specific data, such as DNA sequences [2], needs to be exchanged between two partners. Data compressed with an extended compressor can only be decompressed by the corresponding decompressor: this is a limitation, and raises the interesting question of code migration, but it goes beyond the scope of our work.

SCAPI Fig. 2 shows the (simplified) core classes of SCAPI. To define a new (de)compressor, the user must extend the classes and overload their methods.

[7] u8 is the predefined compressor for integers in the byte range ($0 \leq i < 256$).

A UserCompressor-object compresses text items in two phases:[8] First, the text item is parsed (method ParseString). If the text item has the wrong format and cannot be compressed, then ParseString returns 0, otherwise 1. After successfully parsing the string, the method CompressString is called to store the item.

A UserDecompressor-object decompresses a given data from source containers and writes the decoded string to the output stream.

An Example We illustrate with the delta compressor (which is already defined in XMill under the name di). This compressor stores a sequence of data items like $9012, 9008, 9008, 9015, \ldots$ as $9012, -4, 0, +7, \ldots$: the idea is that the original numbers are large, but differences are small, and can be stored in less space. Such sequences occur e.g. in temperature measurements. The delta compressor is:[9]

```
class DeltaCompr : public UserCompressor  {
 struct DeltaState {
   int oldval;   // The previous value
   int curval;   // The new (currently parsed) value
 };
 char ParseString(char *str,int len, void *state)  {
   return(ParseInt(str,len,
                  &(((DeltaState *)state)->curval)));
 }
 void CompressString(char *str, int len,
                  Container *container,void *state)  {
   container->StoreCompressedInt(
        ((DeltaState *)state)->curval
       -((DeltaState *)state)->oldval);
   ((DeltaState *)state)->oldval=
     ((DeltaState *)state)->curval;
 }
};
```

DeltaState represents the *compressor state* and keeps the last integer value. To avoid parsing the same string again, CompressString keeps the last parsed value in DeltaState.curval. The function ParseInt(str,len,&state->curval) parses string str with length len as an integer and, if successful, stores the result in state->curval and returns 1. Otherwise, it returns 0. The method StoreCompressInt stores an integer in the container, using the compressed integer format described in Sec. 3.1.

The decompressor class DeltaDecompr is simpler:

```
class DeltaDecompr : public UserDecompressor  {
  void Decompress(DecomprCont *container,
                  XMLOutput *output, void *state)  {
    ((DeltaState *)state)->oldval
```

[8] The two-phase approach is necessary to ignore and "undo" previous parses. This is important for combinators, such as seq(u8 "." u8 "." u8 "." u8), when the last u8 rejects its input string.

[9] We omit several details, such as the initialization of the state with oldval=0

```
class UserCompressor {
   virtual char     ParseString   (char *str, int len, void *state)=0;
   virtual void     CompressString(char *str, int len, Container *container, void *state)=0;
};
class UserDecompressor {
   virtual void     Decompress(DecomprCont *container, XMLOutput *output, void *state)=0;
};
```

Figure 2: The SCAPI-interface (fragment) for implementing semantic compressors

```
      += container->GetCompressedInt();
    output->OutputInt(((DeltaState *)state)->oldval);
  }
};
```

The method GetCompressedInt reads the next (compressed) integer from the container and adds its value to the previous value oldval. Then the value is printed using the method OutputInt(oldval).

Additional SCAPI details have been left out from this presentation. For example an optimization is that some compressors never reject their input strings: in that case XMill does a single pass, and avoids calling the method ParseString at all. SCAPI defines a third class, called UserCompressorFactory, used in identifying and instantiating the (de)compressors. Additional methods deal with the initialization/termination process. The interface is further complicated by the fact that some semantic compressors store data into more than one container. Finally, SCAPI allows users to define new compressor combinators (like seq, or, rep), which have parameters. All this adds complexity to the interface. A complete description of SCAPI is in XMill's user manual, http://www.research.att.com/sw/tools/xmill.

5 Experimental Evaluation

We give here a brief overview of XMill's performance, emphasizing the improvements achieved by semantic compressors. More experimental results are in [3]. The Weblog data was described in Sec. 2. SwissProt is a biological database storing meta-data about DNA sequences. Treebank [4] is a large collection of parsed English sentences from the Wall Street Journal stored in a Lisp like notation, which we converted to XML. TPC-D(XML) is an XML representation of the TPC-D benchmark database, using two levels of nesting[10]. We deleted from the TPC-D data the Comment field, which takes about 30% of the space, and consists of randomly generated characters. DBLP is the popular database bibliography database[11], and is stored in a large collection of small XML files, which we concatenated into one large file. Finally, Shakespeare is a corpus of marked-up Shakespeare plays.

The original size of the first four data sources (Weblog, SwissProt, Treebank, TPC-D) is between 35MB and 99MB, while the XML-ized versions have a size between 54MB and 172MB. DBLP and Shakespeare are stored directly in XML and have a size of 47MB and 7.3MB, respectively.

We ran the experiments on a Windows NT machine with a 300MHz Pentium Processor and 128MB main memory. The compression ratio is expressed as "bits per bytes". For example, 2 bits/bytes means that the compressed file size is 25% of the uncompressed file size (lower is better).

Compression Ratio Fig. 3 shows the compression ratios for gzip and XMill under various settings. For each data set, the four connected bars represent gzip, and XMill run with three settings: no grouping (XMill //), grouping based on parent tag (XMill //#; this is the default setting), and user-defined grouping with semantic compression (abbreviated XMill <u>). In XMill <u> we used the best combination of container expressions we could find for each particular data set. As expected, better settings for XMill always produced better compression. For the first four data sets (which had more data and less text), XMill compressed under the default setting to 45%-60% the size of gzip: using semantic compressors, XMill reduced the size to 35%-47% of gzip's. For the more text-like data sets, XMill still performed better than gzip, but less spectacularly. For the first four data sets, the bar on the left represents the size of the gzipped original file (i.e. the height of the bar is size(gzip(orig))/(8∗size(XML))).

Compression Time v.s. Ratio There is a classical trade-off in general-purpose compression: compressors either achieve high compression ratios or are fast. For example, compress is faster than gzip but achieves worse compression rates, while bzip (http://www.bzip2.org) achieves better compression rates but is excessively slow.

We compared XMill against a few standard compressors (gzip, compress, and bzip) and the com-

[10] We tried other XML representations too, and observed no significant change in the experimental results.

[11] http://www.informatik.uni-trier.de/~ley/db/index.html

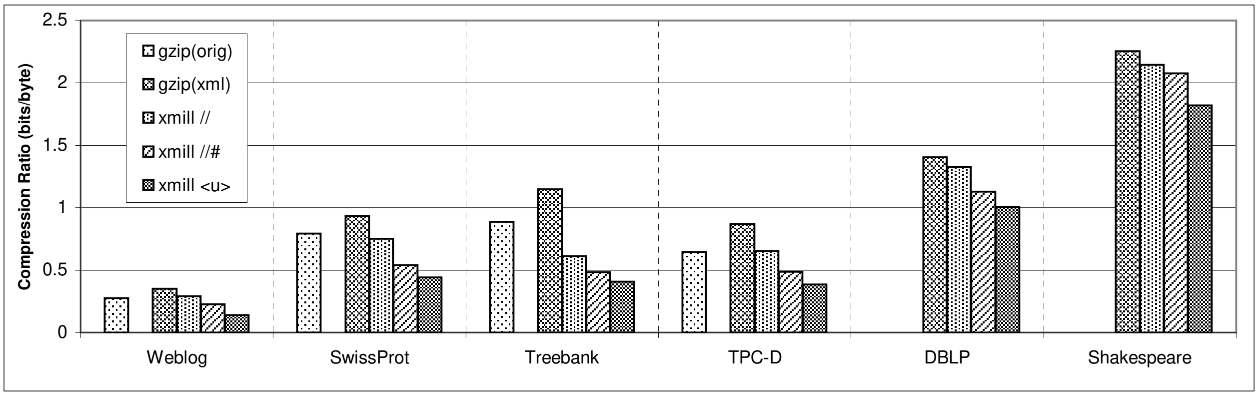

Figure 3: Compression Results

pression time and ratio results are shown in Fig. 4. All compression rates and compression times are normalized w.r.t. that of `gzip`.[12]

The diagram shows clearly that `XMill` offers the best overall time/space tradeoff for XML data. The blobs highlight the "data-like" XML data sets (Weblog, SwissProt, Treebank, and TPC-D). Given `bzip`'s impressive performance, we tried to replace `gzip` with `bzip` in our compressor `XMill`. As expected, the resulting compressor (called XBMill) compresses better (but slower) than `XMill`.

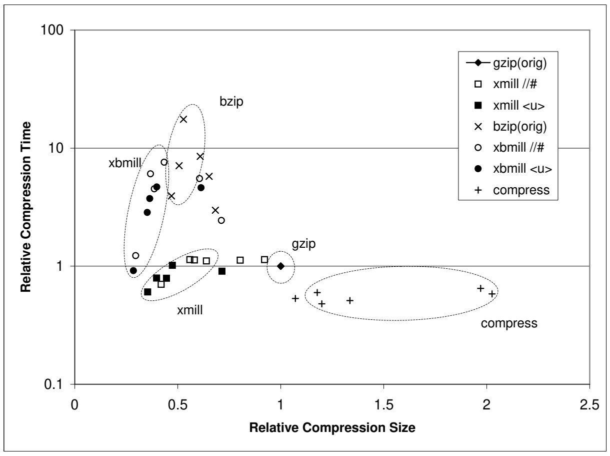

Figure 4: Compression Rate v.s. Time

6 Conclusions

`XMill` and `XDemill` are implemented in C++, and have about 18,000 lines of code. They use `zlib`, the library function variant of `gzip`. `XMill` produces consistently better compression ratios than `gzip`, even under the default settings. We noticed that many XML instances contain simple data types, such as integers, IP addresses, enumeration values. `XMill` provides a few simple semantic compressors for such data types that improve the compression ratio significantly. For XML application containing more complex data types, `XMill` can be extended with new user defined compressors. We believe this to be a powerful feature.

[12] In our configuration, the absolute compression time of `gzip` was between about 30s and 70s.

References

[1] J. Clark and S. DeRose. XML path language (XPath), version 1.0. *W3C Working Draft*, August 1999. Available as `http://www.w3.org/TR/xpath`.

[2] S. Grumbach and F. Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing and Management*, 30(6):875–886, 1994.

[3] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *ACM SIGMOD Conference on Management of Data*, 2000. (to appear).

[4] M.P. Marcus, B. Santorini, and M. Marcinkiewicz. Building a large annotated corpus of english: the Penn Treebank. *Computational Linguistics*, 19, 1993.

[5] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.