XMill: an Efficient Compressor for XML Data

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

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

Abstract

We describe a tool for compressing XML data, with applications in data exchange and archiving, which usually achieves about twice the compression ratio of gzip at roughly the same speed. The compressor, called XMill, incorporates and combines existing compressors in order to apply them to heterogeneous XML data: it uses zlib, the library function for gzip, a collection of datatype specific compressors for simple data types, and, possibly, user defined compressors for application specific data types.

1 Introduction

We have implemented a compressor/decompressor for XML data, to be used in data exchange and archiving, that achieves about twice the compression rate of general-purpose compressors (gzip), at about the same speed. The tool can be downloaded from www.research.att.com/sw/tools/xmill/.

XML is now being adopted by many organizations and industry groups, like the healthcare, banking, chemical, and telecommunications industries. The attraction in XML is that it is a self-describing data format, using tags to mark individual data items. However, there are some serious concerns about exporting one's data into XML. Since XML data is irregular and verbose, it can impact both query processing and data exchange. Many applications (e.g. Web logs, biological data, etc) use other, specialized data formats to archive and exchange data, which are much more economical than XML. As a self-describing format XML brings flexibility, but compromises efficiency.

In this paper we show how to exploit XML's self describing nature to gain in compression. We describe a compressor (XMill) and a decompressor (XDemill) whose architecture leverages existing compressing algorithms and tools to XML data: XMill uses zlib (the library function version of gzip), a few simple, data type specific compressors, and can be further extended with user-defined compressors for complex, application specific data types. The idea in XMill is that it uses the XML tags to decide which compression algorithm to apply.

While experimenting with XMill we made a striking discovery. By migrating data from other, more space-efficient formats to XML, the size of the compressed data decreases. Many such formats are in use today, for biological data, for Web logs, etc. In each case the data is stored in a simple (but application specific) format, usually designed to be reasonably space-efficient for the application at hand. When translated into XML the data expands, mainly because XML tags are verbose and must be repeated; gzip compresses the XML data pretty well, but it is still larger than the original gzipped data. With XMill however, the XML data is compressed better than the original gzipped data, almost to half the size. Thus, by making the data self-describing, one improves compression. Of course, the same kind of compression could be applied to the original format, but one has to write a specific compressor for each format. In summary, by converting to XML, one gains both flexibility and efficiency (when compression is used).

Our compressor, XMill, applies **three principles** to compress XML data:

Separate structure from data The *structure*, consisting of XML tags and attributes, is compressed separately from the *data*, which consists of a sequence of data items (strings) representing element contents and attribute values.

Group related data items Data items are grouped into *containers*, and each container is compressed separately. For example, all <name> data items form one container, while all <phone> items form a second

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

container. This is a generalization of *column-wise compression* in relational databases (see e.g. [10]).

Apply semantic compressors Some data items are text or numbers, while others may be DNA sequences. XMill applies specialized compressors (*semantic compressors*) to different containers.

An original component of XMill are the *container expressions*, a concise language used for grouping data items in containers, and for choosing the right combination of semantic compressors.

Applicability and limitations The compressor described here has two limitations. The first is that it is not designed to work in conjunction with a query processor. Our targeted applications are data exchange, where compression is used to better utilize network bandwidth, and data archiving, where compression is used to reduce space requirement. A second limitation of XMill is that it wins over existing techniques only if the data set is large, typically over 20KB, because of the additional bookkeeping overhead and the fact that small data containers are poorly compressed by gzip. Hence it is of limited or no use in XML messaging, where many small-sized XML messages are exchanged between applications.

Contributions In this paper, we make the following contributions.

- We describe an extensible architecture for an XML compressor that leverages existing compression techniques and semantic compressors to XML data.
- We describe container expressions, a brief yet powerful language for grouping data items according to their semantics, and specifying combined semantic compressors. We present an efficient implementation technique for the path language, which dramatically improves performance for deeply nested data.
- We evaluate XMill on several real data sets and show that it achieves best overall compression rates among several popular compressors. Furthermore, we show that by using XMill one decreases the size of the compressed data by migrating from other data formats to XML.

The paper is organized as follows. Sec. 2 describes two motivating examples. Sec. 3 provides background about compression techniques and gives an information-theoretic justification for our approach to XML compression. The architecture of XMill, the container expression language and semantic compressors are described in Sec. 4. In Sec. 5 we show how to make XMill scalable and to achieve compression/decompression times competitive with gzip. Sec. 6 describes experimental results, which we discuss in Sec. 7. We describe related work in Sec. 8 and conclude in Sec. 9.

2 Motivating Example

We start by illustrating with a very simple, but quite useful example: Web Log files. Virtually every Web server logs its traffic, for security purposes, and this data can be (and often is) analyzed. Each line in the log file represents an HTTP request. A typical entry in such a log file is[1]:

```
202.239.238.16|GET / HTTP/1.0|text/html|200|
1997/10/01-00:00:02|-|4478|-|-|http://www.net.jp/|
Mozilla/3.1[ja](I)
```

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 fields delimited by |: host, request line, content type, etc. Hence, the file's structure is very simple, with records with a fixed number of variable-length fields[3].

Collected over long periods of time, Web logs can take huge amounts of space. In our example we only considered a file with 100000 entries as the one above. Its size is almost 16MB, and gzip shrinks it to 1.6MB:

weblog.dat: **15.9MB** weblog.dat.gz: **1.6MB**

Applications processing such Web logs are brittle, and in general not portable, since different vendors use different formats. To gain flexibility, we may consider converting the Web log into XML with the following format:

```
<apache:entry>
 <apache:host>202.239.238.16</apache: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.net.jp/</apache:referer>
 <apache:userAgent>Mozilla/3.1[ja](I)</apache:userAgent>
</apache:entry>
```

Applications are now easier to write. However the size increases substantially:

weblog.xml: **24.2MB** weblog.xml.gz: **2.1MB**

Our goal is to gain from XML's flexibility without using more space. An obvious idea is to assign integer codes (1, 2, 3, ...) to the XML tags, and use a single character for closing tags. A more interesting idea is to separate the XML tags (encoded as numbers) from the data values, and compress with gzip independently the tags and the data values. We save space, because the XML tags are the same for each record, and gzip can encode this very efficiently (see Sec. 3.2). With XMill this effect is accomplished by command line:[4] `xmill -p // weblog.xml`. This brings the size down to:

[1]This is one line in the log file.
[2]http://www.apache.org/docs/mod/mod_log_config.html
[3]Missing values are common and are indicated by -.
[4]Sec. 4.2 describes XMill's command line.

```
              ...
-p//apache:host=>seq(u8 "." u8 "." u8 "." u8)
-p//apache:byteCount=>u
-p//apache:contentType=>e
-p//apache:requestLine=>seq("GET " rep("/" e) " HTTP/1.0")
              ...
```

Figure 1: Semantic compressor settings `settings.pz`.

weblog1.xmi: **1.75MB**

The next idea is to compress data values separately, based on their tags: that is, all host values are compressed together, all request lines are compressed together, etc.

This behavior is the default and is achieved using the ommand line `xmill weblog.xml`. Since `gzip` achieves better compression when applied to values of similar types, this reduces the size even further:

weblog2.xmi: **1.33MB**

We now use less space than the original gzipped file.

We can do quite a lot better than that. The idea is to inspect carefully each field and use a specialized compressor for it. For example the `<apache:host>` is usually (or always) an IP address, hence can be stored as four unsigned bytes; most entries in `<apache:requestLine>` start with `GET` and end in `HTTP/1.0` (some in `HTTP/1.1`): these substrings can be factored out. Other improvements are also possible. We analyzed eight of the eleven fields and applied specialized compressors available in `XMill`. The corresponding `XMill` command line is:

```
xmill  -f settings.pz  weblog.xml
```

where some parts of file `settings.pz` are shown in Fig. 1 (specialized compressors are described in Sec. 4.3). This reduces the compressed size to:

weblog3.xmi: **0.82MB**

Note that this is about half the original gzipped file. This achieves our goal: the compressed XML-ized data can be stored in less space than the compressed original data, while applications gain in flexibility[5].

The Web log is a simple example illustrating columnwise compression applied to XML. The second example is much more complex. SwissProt is a well-maintained database for representing protein structure [6]. It uses a specific data format, called EMBL [8], for representing information about genes and proteins (not shown here for lack of space). We converted the original EMBL data into XML as shown in Fig. 2.

We repeated the experiments above on a fragment of the SwissProt data[7]. The original file had 98MB and the XML-ized version had 165MB. `gzip` reduces the files to 16MB and 19MB, respectively:

```
<Entry id="108_LYCES" mtype="PRT" seqlen="102">
 <AC>Q43495</AC>
 <Mod dat="15-JUL-1999" Rel="38" typ="Created"></Mod>
 <Mod dat="15-JUL-1999" Rel="38" typ="Last SeqUpd"></Mod>
 <Mod dat="15-JUL-1999" Rel="38" typ="Last AnnUpd"></Mod>
 <Descr>PROTEIN 108 PRECURSOR</Descr>
 <Species>Lycopersicon esculentum (Tomato)</Species>
 <Org>Eukaryota</Org> ... <Org>Solanum</Org>
 <Ref num="1" pos="SEQUENCE FROM N.A">
  <Comment>STRAIN=CV. VF36</Comment>
  <MedlineID>94143497</MedlineID>
  <Author>CHEN R</Author> <Author>SMITH A.G</Author>
  <Cite>Plant Physiol. 101:1413-1413(1993)</Cite>
 </Ref>
 ...
 <EMBL prim_id="Z14088" sec_id="CAA78466"></EMBL>
 <MENDEL prim_id="8853" sec_id="LYCes"></MENDEL>
 <Keyword>Signal</Keyword>
 <Features>
  <SIGNAL from="1" to="30">  <Descr>POTENTIAL</Descr>
  </SIGNAL>
  <CHAIN from="31" to="102"> <Descr>PROTEIN 108</Descr>
  </CHAIN>
   ...
 </Features>
</Entry>
```

Figure 2: XML Representation of SwissProt entry

sprot.dat:	**98MB**	sprot.xml:	**165MB**
sprot.dat.gz:	**16MB**	sprot.xml.gz:	**19MB**

Repeating the three steps above we obtained the following improvements in size:

sprot1.xmi:	**15MB**
sprot2.xmi:	**11MB**
sprot3.xmi:	**8.6MB**

Note that the last file is obtained after fine-tuning `XMill` on the SwissProt data.

In both examples the three steps correspond precisely to the compression principles spelled out in Sec. 1. As the examples suggests, each principle contributes with a significant improvement.

3 Background

3.1 XML

For the purpose of this paper, an XML document consists of three kinds of tokens: tags, attributes, and data values. As usual we model an XML document as a tree: nodes are labeled with tags or attributes, and leaves are labeled with data values. The *path* to a data value is the sequence of tags (and, possible one attribute) from the root to the data value node.

3.2 Compressors

General Purpose Compressors Most practical dictionary compressors are derived from the LZ (Ziv and Lempel) family of compressors. The idea in the original

[5]Of course, an application has to decompress the data first.

[6]`http://www.expasy.ch/sprot/`

[7]We omitted comments and the actual DNA sequence, which can be compressed using specialized compressors.

LZ77 [20] is to replace repeating sequences in the input text with a pointer to a previous occurrence. We refer the reader to [2] for a good introduction, but only mention here one important property of LZ77 that we exploited in `XMill`. Namely a large number of repetitions of the same sequence, like `A B C A B C ...A B C` are compressed extremely well, essentially as a run length encoding storing only one copy of `A B C`, its length, and a repetition count.

The popular general-purpose compression tool `gzip` uses LZ77 in combination with other techniques. A function library, `zlib`, makes its functionality available to applications. We used `zlib` in `XMill`, and will refer to `zlib` and `gzip` interchangeably in the paper.

Special Purpose Compressors A variety of special-purpose compressors exists, ranging from ad-hoc to highly complex ones [2, 16]. Special data types can be encoded in binary, e.g. *integer* or *date*. A *dictionary encoding* assigns an integer to each new word in the input, and stores the mapping from codes to strings in a dictionary. Specialized compressors exist for a variety of data types, e.g. images, sound or DNA sequences [2, 7].

3.3 Information Theory

In his classic paper [18] introducing information theory Claude Elwood Shannon describes an *information source*, a *channel*, and a *destination*, and studies how much information can be sent by the source to the destination. This is given precisely by how well the source can be compressed. A source S generates a message $x_1, x_2, \ldots, x_m$, symbol by symbol, with each symbol drawn from a fixed, finite alphabet $A = \{a_1, \ldots, a_n\}$. Shannon modeled a source as a Markov Process, and defined its *entropy*, H. The most popular formula for the entropy is for the special case of order 0 Markov Processes, where each symbol a_i has a fixed probability p_i:

$$H \stackrel{\text{def}}{=} p_1 \log \frac{1}{p_1} + \ldots + p_n \log \frac{1}{p_n}$$

Shannon proved in his paper the *fundamental theorem for a noiseless channel*, which essentially says that a message of m symbols cannot be compressed to less than mH bits on average, and that almost optimal compressors exists. Dictionary compressors, discussed at the beginning of this section, have been shown to achieve almost optimal compression [2].

Optimal compression of heterogeneous sources Unlike Shannon's information sources, XML data is heterogeneous. We define a *heterogeneous information source* S to be a collection of $k+1$ sources $S_0, S_1, \ldots, S_k$, over alphabets $A, B_1, \ldots, B_k$. The first alphabet has k symbols, $A = \{a_1, \ldots, a_k\}$, called *tags*, while the others can have an arbitrary number of symbols. The heterogeneous source emits messages of the following shape:

$$x_1, y_1, x_2, y_2, \ldots, x_m, y_m \qquad (1)$$

where $x_1, \ldots, x_m \in A$, and, whenever $x_j = a_i$, then the next symbol y_j belongs to B_i.

We prove that the three compression principles in Sec. 1 lead to an optimal compression for heterogeneous sources. Heterogeneous sources are a simplification of XML since they don't model nesting: nesting can be modeled by probabilistic grammars [2].

If all $k+1$ sources are of order 0, then the heterogeneous source S is equivalent to a (homogeneous) source modeled by a Markov Process with $k+1$ states over the alphabet $A \cup B_1 \cup \ldots \cup B_k$ (details omitted).

Consider the following compression of a heterogeneous source in three steps: (1) separate the tags $x_1, x_2, \ldots$ from the data items $y_1, y_2, \ldots$, (2) further separate the data items according to their source $S_i, i = 1, k$, (3) apply an optimal compressor for each source $S_0, \ldots, S_k$. Let $H_0, H_1, \ldots, H_k$ be the entropies of the $k+1$ sources, and let $p_1, \ldots, p_k$ be the probabilities of source S_0. Then the compression just described uses:

$$mH_0 + mp_1H_1 + mp_2H_2 + \ldots mp_kH_k \qquad (2)$$

bits for the message (1) of length $2m$. This is because it needs mH_0 bits for $x_1, x_2, \ldots, x_m$; then there are, on average mp_1 symbols from source S_1, etc. Our theorem below proves that this is optimal:

Theorem 3.1 *The entropy of the heterogeneous source S is: $\frac{1}{2}(H_0 + p_1H_1 + \ldots + p_kH_k)$. Hence the number of bits used in (2) is optimal on average.*

4 The Architecture of `XMill`

The architecture of `XMill` is based on the three principles described in Sec. 1 and is shown in Fig. 3. The XML file is parsed by a SAX[8] parser that sends tokens to the path processor. Every XML token (tag, attribute, or data value) is assigned to a container. Tags and attributes, forming the XML structure, are sent to the structure container. Data values are sent to various data containers, according to the container expressions, and containers are compressed independently.

The core of `XMill` is the *path processor* that determines how to map data values to containers. The user can control this mapping by providing a series of *container expressions* on the command line. For each XML data value the path processor checks its *path* against each container expression, and determines either that the value has to be stored in an existing container, or creates a new container for that value.

[8]Simple API for XML, `http://www.megginson.com/SAX/`.

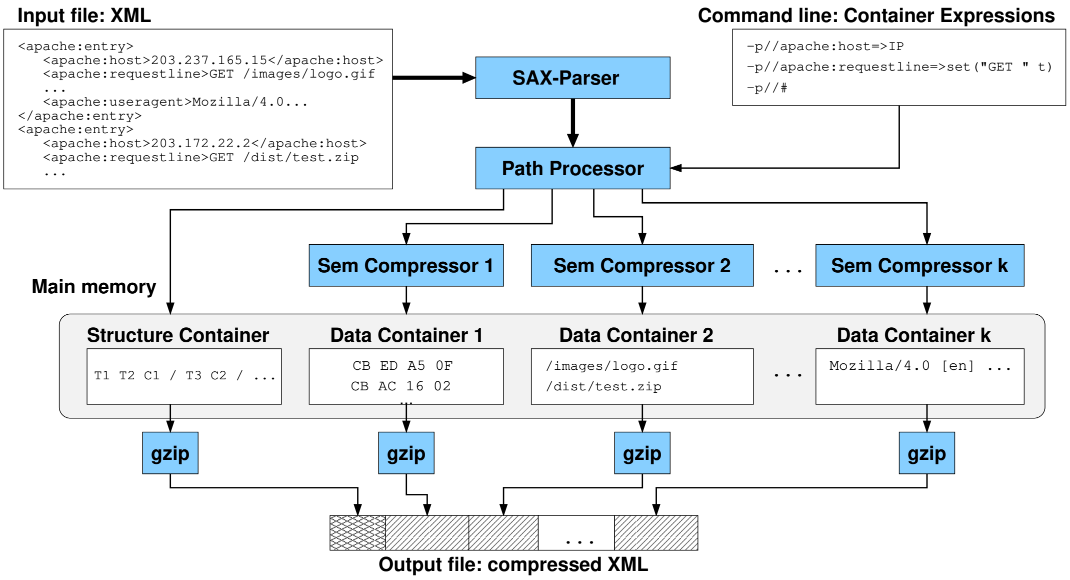


Figure 3: Architecture of the Compressor

Users can associate semantic compressors with containers. A few *atomic* semantic compressors are predefined in `XMill`, like binary encoding of integers, differential compressors, etc. Users can also *combine* semantic compressors into more complex ones or can write new semantic compressors and *link* them into `XMill`. The default *text* "compressor" simply copies its input to the container, without any semantic compression.

Containers are kept in a main memory window of fixed size (the default is 8MB). When the window is filled, all containers are gzipped, stored on disk, and the compression resumes. In effect this splits the input file into independently compressed blocks.

The decompressor `XDemill` is simpler, and its architecture is not shown. After loading and unzipping the containers, the decompressor parses the structure container, invokes the corresponding semantic decompressor for the data items and generates the output.

4.1 Separating Structure from Content

The *structure* of an XML file consists of its tags and attributes, and is tokenized in `XMill` as follows. Start-tags are dictionary-encoded, i.e. assigned an integer value, while all end-tags are replaced by the token `/`. Data values are replaced with their container number. To illustrate, consider the following small XML file:

```
<Book> <Title lang="English"> Views </Title>
       <Author> Miller </Author>
       <Author> Tai </Author>
</Book>
```

Tags, such as `Book`, `Title`, ... are dictionary encoded as `T1`, `T2`, etc. Data values (e.g. `English`, `Views`, `Miller`, and `Tai`) are assigned containers `C3`, `C4`, and `C5` depending on their parent tag:

```
Book = T1, Title = T2, @lang = T3, Author = T4
Structure = T1 T2 T3 C3 / C4 / T4 C5 / T4 C5 / /
```

In practice all tokens are encoded as integers (with 1, 2, or 4 bytes, see Sec. 4.3): tags/attributes are positive integers, `/` is 0, and container numbers are negative integers. The structure above needs 14 bytes.

So far we have ignored white spaces between tags, e.g. between `<Book>` and `<Title>`, and the decompressor produces a canonical indentation. Optionally, `XMill` can preserve white spaces: in that case it stores them in container[9] 1. In our example, the structure becomes `T1 C1 T2 C1 T3 C3 / C4 / C1 T4 ....`

The size of the compressed file typically increases only slightly when white spaces are preserved: around 4%. We observed a higher increase (30%) only for Treebank, a linguistic database (see Sec. 6), because of its deeply nested structure. In the rest of the paper we will assume that white spaces are ignored.

We observed that, in practice, our simple encoding scheme compresses extremely well. Since many data sources tend to have repeated or similar structures (e.g. many books with one Title, one `@lang` attribute and two Authors), `gzip`'s algorithm (Sec. 3.2) can reduce the size dramatically. In our experiments the compressed structure was typically around 1%-3% of the compressed file for data with regular structure and 20% for data with highly irregular structure (Treebank).

4.2 Grouping Data Values

Each data value is uniquely assigned to one data container. The mapping from data values to containers

[9] Container 0 holds the structure while container 2 holds the PI's, DTD's, and comments.

is determined by the following information: (1) the data value's path, and (2) the user-specified container expressions. We describe them next, using the following running example:

```
<Doc> <Book> <Title lang="English"> Views </Title>
      </Book>
      <Person> <Name>  Peter </Name>
               <Title> Mr. </Title>
               <Child> Karen </Child>
      </Person>
</Doc>
```

Recall that the *path* to a data value is the sequence of tags from the root to that value (Sec. 3.1): e.g. the path to `Mr.` is `/Doc/Person/Title`, while the path to `"English"` is `/Doc/Book/Title/@lang`.

Container Expressions A natural idea is to create one container for each tag or attribute. For example all `Title` data values go to one container, all `@lang` attribute values go to a different container, etc.

This simple mapping typically performs well in practice, but sometimes it is too restrictive. The context may change the tag's semantics: `/Doc/Book/Title` has a different meaning from `/Doc/Person/Title`, hence the two `Title`'s are best compressed separately. Conversely, different tags may have the same meaning, like `Name` and `Child`.

Our approach is to describe mappings from paths to containers with container expressions. Consider the following regular expressions derived from XPath [4]:

$$e \ ::= \ label \mid * \mid \# \mid e_1/e_2 \mid e_1//e_2 \mid (e_1|e_2) \mid (e)+$$

Except for $(e)+$ and $\#$, all are XPath constructs: *label* is either a *tag* or an @*attribute*, $*$ denotes any tag or attribute, e_1/e_2 is concatenation, $e_1//e_2$ is concatenation with any path in between, and $(e_1|e_2)$ is alternation. To these constructs we added $(e)+$, the strict Kleene closure.

The interesting novel construct is $\#$. It stands for any tag or attribute (much like `*`), but each match of $\#$ will determine a new container. The formal semantics of container expressions is described in [12].

A *container expression* has the form $c \ ::= \ /e \mid //e$, where $/e$ matches e starting from the root of the XML tree while $//e$ matches e at arbitrary depth in the tree. We abbreviate $//*$ with $//$.

Example 4.1 `//Name` creates one container for all data values whose path ends in `Name`. `//Person/Title` creates a container for all `Person`'s titles. `//` places all data items into a single container.

Example 4.2 `//#` creates a family of containers: one for each ending tag or attribute. It is a concise way to express a whole collection of container expressions: `//Title`, `//@lang`, etc. (one for each tag in the XML file). `//Person/#` creates a distinct container for each tag under `Person`, and `(#)+` creates a distinct container for every path.

Container expressions `c1, ..., cn` are given in the command line, with the `p` switch:

```
xmill  -p c1  -p c2  ...  -p cn  file.xml
```

For each data value, the path processor matches its path against $\mathtt{c}_1, \mathtt{c}_2, \ldots$, in that order. Assuming the first match is found at $\mathtt{c_i}$, the processor computes the "values" of the `#`'s in $\mathtt{c_i}$ which made the match possible. These values, together with `i`, uniquely determine the data value's container.

Example 4.3 Consider the following command line:

```
xmill  -p //Person/Title  -p //(Name|Child)
       -p //#  file.xml
```

This command line compresses all `Person`'s titles together, all `Name`s and `Child`s together. All other data values are compressed based on their ending tag. In particular `.../Book/Title` will be compressed separately from `.../Person/Title`.

Default Behavior The expression `-p //#` is always inserted at the end of the command line. In particular, the command line `xmill file.xml` is equivalent to `xmill -p //# file.xml`. This ensures that every data value is stored in at least one container and it provides a reasonable default behavior.

4.3 Semantic Compressors

XML data often comes with a variety of specialized data types like integers, dates, US states, airport codes, which are best compressed by specialized semantic compressors. `XMill` supports three kinds of semantic compressors: atomic, combined, and user-defined.

Compressor	Description
t	default text compressor
u	compressor for positive integers
i	compressor for integers
u8	compressor for pos. integers < 256
di	delta compressor for integers
rl	run-length encoder
e	enumeration (dictionary) encoder
"..."	constant compressor

Table 1: Atomic Semantic Compressors

Atomic semantic compressors: There are eight such compressors in `XMill`, shown in Table 1. We explain here just a few and refer the reader to [16] or standard textbooks [17] for a general discussion.

The text compressor `t` does not compress, but rather copies the string to the container unchanged (it will be compressed later by `gzip`). Positive integers (compressor `u`) are binary encoded as follows: numbers less than 128 use one byte, those less than 16384 use two bytes, otherwise they use four bytes. The most significant one or two bits determine the length. The last entry is a constant compressor that does not produce any output (the best compression of all !), but checks that the input is the given constant. Some semantic compressor-decompressor pairs may be lossy, e.g. `u, u8, i` do not preserve leading zeros.

Semantic compressors are optionally specified on the command line using the syntax `c=>s` where `c` is a container expression (Sec. 4.2) and `s` is a semantic compressor. When missing, the default semantic compressor is text. For a simple illustration, consider the example:

```
xmill -p //price=>i  -p //state=>e  file.xml
```

The `price` data items are compressed as integers, `states` as enumeration values, and all remaining data items are grouped based on their tag (recall that the default `-p //#` is added at the end), with no semantic compression.

A semantic compressor may reject its input string. In the example above, a `price` value which does not parse as an integer will be rejected by the `i` compressor. In that case `XMill` tries the next path expression: eventually, the last `-p //#` will match. One can exploit this behavior by specifying alternative compressors, like in `xmill -p //price=>i -p //price=>e ...` to capture price values like `1450, low, 55, high, ...`.

Combined compressors: Often data values have structure. For example an IP address consists of four integers separated by dots (e.g. `104.44.29.21`); a `request` value (Sec. 2) consists of `GET` followed by a variable string. `XMill` has three compressor combinators for compressing such values:

- *Sequence Compressor* `seq(s1 s2 ...)`. For example, `seq(u8 "." u8 "." u8 "."  u8)` compresses an IP address as four integers.

- *Alternate Compressor* `or(s1 s2 ...)`. For example, consider page references in a bibliography file. These can be either like `145-199`, or single pages like `145`. The composite compressor is `or(seq(u "-" u) u)`.

- *Repetition Compressor* `rep(d s)`. Here `d` is the delimiter and `s` another semantic compressor. For example, a sequence of comma separated keywords can be compressed by `rep("," e)`.

Fig. 1 illustrates the use of combined semantic compressors for the Weblog data.

User-defined Compressors Some applications require highly specialized compressors, like for DNA sequences [7]. Users can write their own compressors/decompressors and link them into `XMill` and `XDemill`, conforming to a specified API, called SCAPI (Semantic Compressor API [11]). Semantic compressors can be used in the command line, like in `xmill -p //DNAseq=>dna file.xml`, where `dna` is the compressor's name. The extended `XMill` becomes application specific, since a file compressed with such an extended `XMill` can only be decompressed by an `XDemill` with the corresponding decompressor.

5 Implementation

`XMill` and `XDemill` are implemented in C++, and have together about 18,000 of code. We wrote our own SAX parser for XML, which parses the XML file and translates it into a stream of events: one event for each start-tag, end-tag, data value, etc. Every XML event (token) is sent to the path processor, which is described next.

5.1 Path Processor

The path processor keeps track of the current path for each data value and evaluates successively each container expression on the path: the latter involves evaluating a regular expression, and, if successful, evaluating the semantic compressor on that data value. This is the most time-critical piece in the compressor and we tried three different evaluation methods. They are described in detail in [12].

Direct Evaluation of Regular Expressions Each container expression is preprocessed into a minimized, deterministic automaton (DFA)[9], and for each of them we maintain its current state while parsing the XML file. This method becomes inefficient when we have more than one container expression, since we need to evaluate several DFAs for each XML tag.

Evaluation using DataGuides Here we use a cache: if $p_1, p_2, \ldots$ are all the XML paths seen so far, then the cache consists of a trie for $p_1, p_2, \ldots$. This trie becomes equivalent to a DataGuide [5] at the end of the XML document. We keep a list of corresponding DFA states at each DataGuide node, and only need to advance a single pointer in the DataGuide while parsing XML tags. An exception is when we encounter a new path p: then we expand the DataGuide with a new node and we need to compute its associated DFA states. The size of the DataGuides ranges from a few nodes (for regular data) to very large (for irregular data). We found DataGuides efficient except for the most irregular and deeply nested data.

Evaluation using Reversed DataGuides Irregular and deeply nested data causes the DataGuide to grow out of proportions. An example of such data is the XML-ized *TreeBank* linguistic database[10] [13], which contains annotated sentences from the Wall Street Journal. The DataGuide had 340000 nodes, which translated into about 16MB of main memory (depending on the number of DFAs), far exceeding our 8MB memory window.

Our third strategy uses a *reversed DataGuides*, which is just the DataGuide structure for the *reverse* paths, and working in conjunction with reversed DFA's. The reversed DataGuide in our example has 1.1 million nodes (in contrast to 340000 nodes). However it is possible to *prune* the reversed DataGuide much better than the direct DataGuide, because container expressions usually discriminate based on the last few tags in the path: e.g. `//Person/Name` and `//#` only look at the last one or two tags. In all our examples the reversed DataGuide were pruned after one or two tags. For the Treebank data, pruning was done after one tag, reducing the reversed DataGuide to approx. 250 nodes (the number of distinct leaf tags).

6 Experimental Evaluation

We evaluated `XMill` on several data sets. Our goal was to validate `XMill` for XML data archiving and data exchange and to test `XMill` as a compensatory tool for migrating other data formats to XML.

Data sources We report the evaluation of `XMill` on six data sources, whose characteristics are shown in Fig. 6. The Weblog and the SwissProt data were described in Sec. 2. Treebank [13] 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[11]. 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[12]. Finally, Shakespeare is a corpus of marked-up Shakespeare plays, and it is stored directly in XML.

Fig. 6 shows the size of the original data sources, the size of their XML representation, and four characteristic measures: our assessment of the data's regularity (yes/no), the maximal depth of the XML tree, the number of distinct tags, and the number of nodes in the DataGuide (another measure of (i)regularity).

Classes of experiments We performed three classes of experiments. First, we compared the compression ratios of `gzip` and `XMill` under various settings. We also tested the variation of the compression ratio as a function of the data size, and its sensitivity to the memory window. Second, we measured the compression and decompression times of `XMill` and `gzip`. Third, we measured the total effect of `XMill` in an XML data exchange application over the network.

Platform We ran the first two sets of experiments on a Windows NT, 300Mhz PII machine with 128MB main memory. The data exchange experiment was performed by sending data from AT&T Labs, running an SGI Challenge L (4 x 270MHz MIPS R12000, Irix 6.5.5m) to two places: the University of Pennsylvania, running a Sun Enterprise 3000 (4 x 250Mhz UltraSPARC) with 1024MB of memory, and a home PC (100MHz, Linux, 32MB) connected to a cable modem. We transfered files with `rcp`, for which we measured a transfer rate of 8.08MBits/s (AT&T to Penn) and 1.25MBits/s (AT&T to home PC via cable modem).

Experimental Methodology The *compression ratio* is expressed as "bits per bytes", e.g. 2 bits/bytes means 25% of the uncompressed file size. The *running time* represents the elapsed time in seconds. We run each experiment eight times and take the average of the last five runs. For the data exchange experiment, we measured the compression and decompression times separately (at AT&T) from the data transfer; each was executed eight times, as explained.

In comparing the running time of `XMill` with `gzip` we noticed differences in the (time and space) efficiency of `gzip` (the stand-alone tool) and `zlib` (the library used in `XMill`). For meaningful comparisons, we replaced `gzip` with `minigzip`, a stand-alone program in `zlib`, and compiled it with the same options as `XMill`. In all experiments below, "`gzip`" actually means `minigzip`.

6.1 Compression Ratio

Fig. 5 shows the compression ratios for different data sources and compressors. For each data set, the four connected bars represent `gzip`, and `XMill` run with three settings (as in Sec. 2): no grouping (`XMill //`), grouping based on parent tag (`XMill //#`; the default), and user-defined grouping with semantic compression (abbreviated `XMill` <u>). In `XMill` <u> we used the best combination container expressions we could find for each data set. For the first four data sets, the bar on the left represents the relative size of the gzipped original file (i.e. the height of the bar is `size(gzip(orig))/(8*size(XML))`).

For the first four data sets (which had more data and less text), `XMill`'s 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` performs only slightly better than `gzip`. Note that with the

[10] More information about TreeBank is available under `http://www.cis.upenn.edu/~treebank/`.

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

[12] `http://www.informatik.uni-trier.de/~ley/db`

Data Source	Original Size	Size in XML	Regular?	Depth	Tags	DataGuide Size
Weblog Data	57.7MB	172MB	yes	1	10	11
SwissProt	98.5MB	158MB	yes	3	92	58
Treebank	39.6MB	53.8MB	no	35	251	339920
TPC-D	34.6MB	119MB	yes	2	43	60
DBLP	-	47.2MB	yes	3	10	145
Shakespeare	-	7.3MB	no	5	21	58

Figure 4: Data sources for performance evaluation

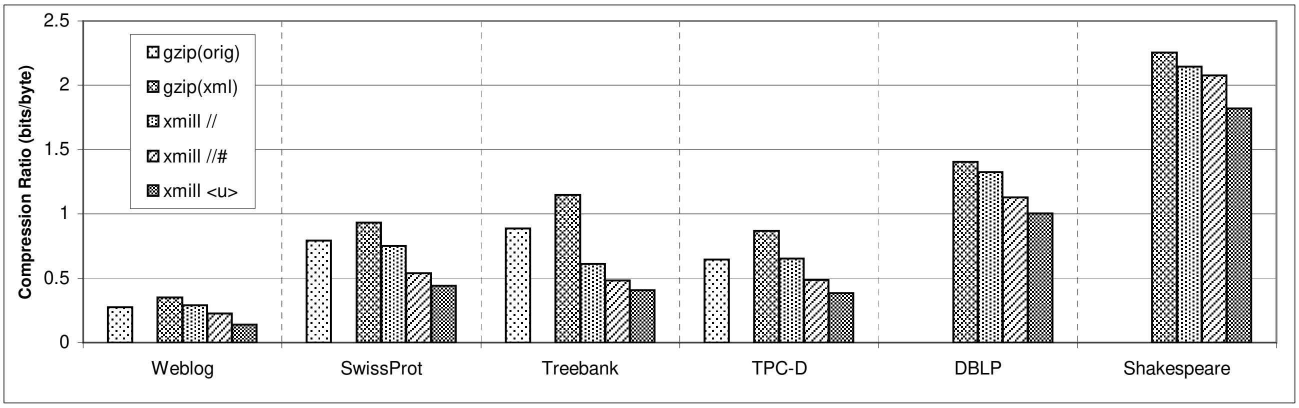


Figure 5: Compression Results

default setting, XMill already compressed the XML file better than gzip compressed the original file.

Fig. 6 shows the compression ratio as a function of the XML data size for Weblog and SwissProt. On small files, XMill performs worse than gzip because it splits the data into too many small containers. The crossing point for both data sets was at about 20KB.

Fig. 7(a) shows the sensitivity of XMill's compression ratio on the memory window size. The compression ratio is normalized with respect to the default of 8MB. The results show that a smaller memory window size substantially degrades the compression rate, again because a small window implies small containers, on which the compression rate is poor. Beyond 8MB, both data sets were compressed in only a few blocks, and the compression rate did not improve too much.

6.2 Compression/Decompression Time

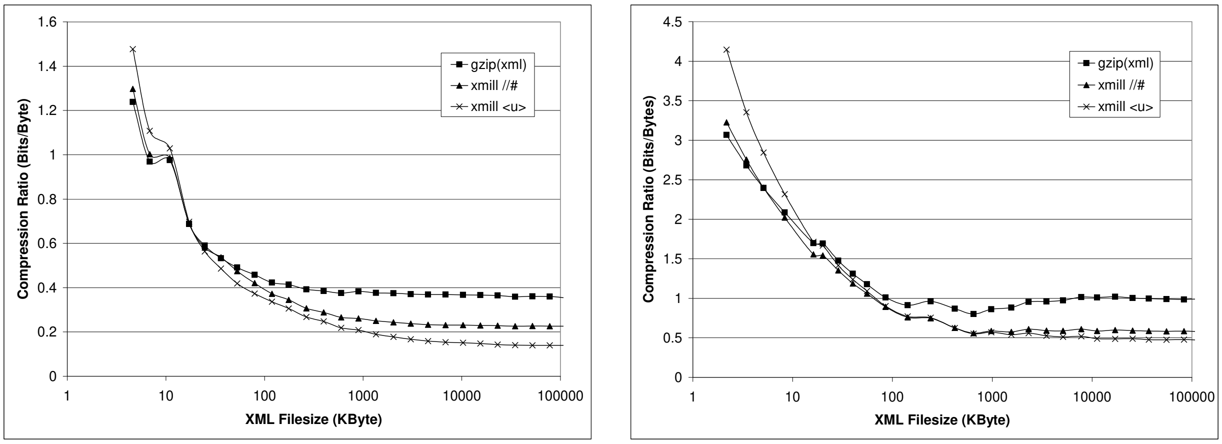
We measured the (de)compression time for Weblog, SwissProt, and Treebank and considered three compression strategies: gzip, XMill //#, and XMill <u>. Fig. 8(a) shows the compression time for each data source and compressor. For XMill, the time is split into two parts: (1) parsing and applying semantic compressors, and (2) applying gzip. XMill is generally as fast as gzip, since XMill saves time by applying gzip to a smaller data size and spends the time on regrouping. For the same reason, XMill <u> is faster than XMill //#, because the semantic compressors precompress the data, hence gzip spends less time.

Fig. 8(b) shows the decompression time for each of the data sources and compressors, broken down according to the decompression step. There are four such steps: (1) gunzip the containers, (2) interpret the XML structure and merge the data values, applying the appropriate semantic decompressors: this results in a stream of SAX events, (3) generate the XML string (start-tags, end-tags, data values, etc), (4) output the file. For gunzip we only have steps (1) and (4). Note that the time fragmentation into parts (1)-(4) is not completely accurate, because of caching interferences.

For a complete decompression (written to a file) XDemill's speed is comparable to gunzip. If we remove the output step (4) for on-the-fly decompression, then XDemill is about twice slower than gunzip. We pay here the price of having to merge data from different containers. However, an application could do even better by consuming SAX events directly, rather than having to re-parse the XML string: such applications only need XMill to perform steps (1) and (2) (gunzip's output always needs to be parsed).

6.3 Data Exchange

Fig. 9 shows the results of exchanging Weblog data from AT&T to the Univ. of Pennsylvania (a) and to a home computer via a cable modem (b). The



(a) Weblog (b) SwissProt

Figure 6: Compression Ratio under Different Sizes of Weblog and SwissProt

bars are split into compression time (lower bar), and transmission+decompression (upper bar). The end-to-end transfer time (from XML file to XML file) is dominated by the compression time: here there are no significant differences between `XMill` and `gzip` (with some slight advantage for `XMill` on the slow network). If the file is already compressed, then `XMill` shows detectable improvements over `gzip`; again, better so in the case of a slow network. Furthermore, if the decompressed data is used directly in an application (via a SAX interface), then only the transfer and decompress+decode time matters, and `XMill` takes only about 60% of the time of `gzip`. We obtained similar results for SwissProt, which are omitted.

7 Discussion and Future Work

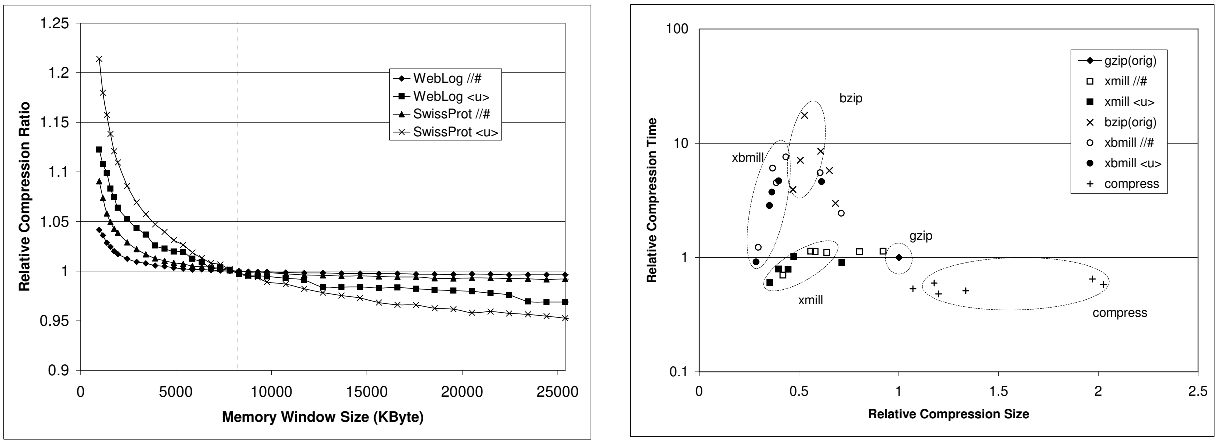
Benefits of `XMill` The experiments show that `XMill` clearly achieves better compression rates than `gzip` (around a factor of 2, for data-like XML, less for text-like XML), without sacrificing speed. This makes `XMill` a clear winner for data archiving. For data exchange however, the improvement depends on two factors: the type of exchange application, and the relative processor v.s. network speed. For a slow network, `XMill`'s improvements are always detectable, because of its better compression rate. For a fast network, one has to look at all three exchange steps: compression, network transfer, and decompression. Compression is consistently the most expensive, and is about the same in `gzip` and `XMill`. Hence, the relative advantage depends on the type of application. For an end-to-end file transfer, there is no clear winner. In XML publishing, the file is compressed only once, and only network transfer and decompression matters: `XMill` is consistently, but only modestly faster than `gzip`. If, moreover, the data is imported directly into applications, then the decompression does not need to produce an output XML file: here `XMill` can become significantly better than `gzip`.

Time/Space Tradeoff Different general-purpose compressors offer a variety of time/space tradeoffs. We tried a few of them on our six data sets (Fig. 6): `gzip`, `compress`, and `bzip`, where `compress` is faster than `gzip` but achieves worse compression rates, while `bzip` achieves better compression rates but is excessively slow. The results are shown in Fig. 7(b), where all compression rates and compression times are normalized with respect to that of `gzip`. The blobs highlight the "data-like" XML data sets (Weblog, SwissProt, Treebank, and TPC-D). The diagram shows that `XMill` offers the best overall time/space tradeoff for XML data. Given `bzip`'s impressive performance, we replaced `gzip` with `bzip` in `XMill`. Interestingly, while the resulting compressor (called `xbmill`) compresses better than `XMill`, the compression times did not increase as badly as between `gzip` and `bzip`: this is because `xbmill` applies `bzip` to a smaller amount of data.

Schema Extraction All container expressions in `XMill` have to be specified manually. They were designed keeping the XML-Schema in mind [19], and it is relatively straightforward to generate them from a given XML-Schema. However, it would be more useful to extract them automatically from a given XML data set. Unlike previous work on schema extraction for semistructured data [14], the critical part is choosing the right semantic compressor for each container. An automatic datamining tool must recognize integers, dates or structured fields and cluster data correspondingly.

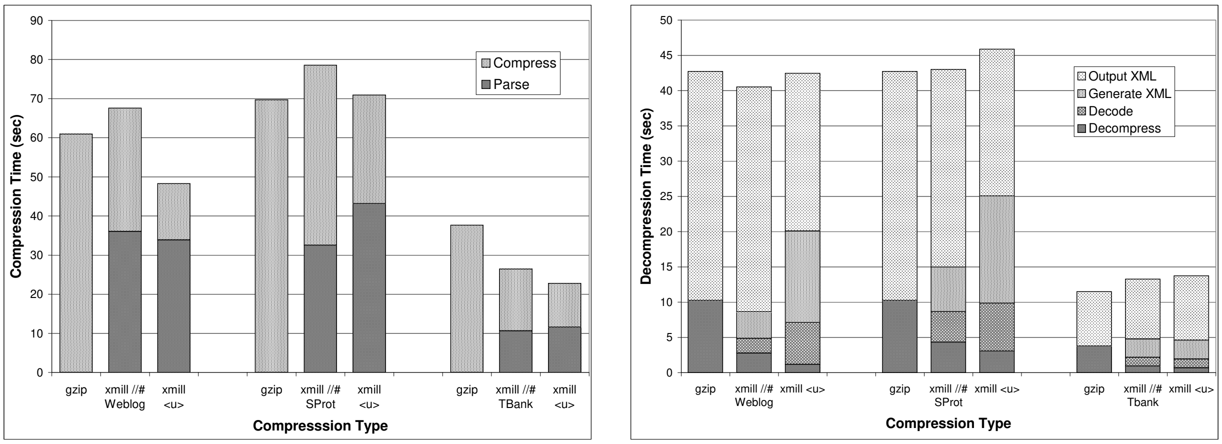
8 Related Work

General Compression General compression methods are described in textbooks [2, 17]. A more recent method is block-sorting compression described in [3], which is used in `bzip`. It sorts the characters in a block first before applying other compression.



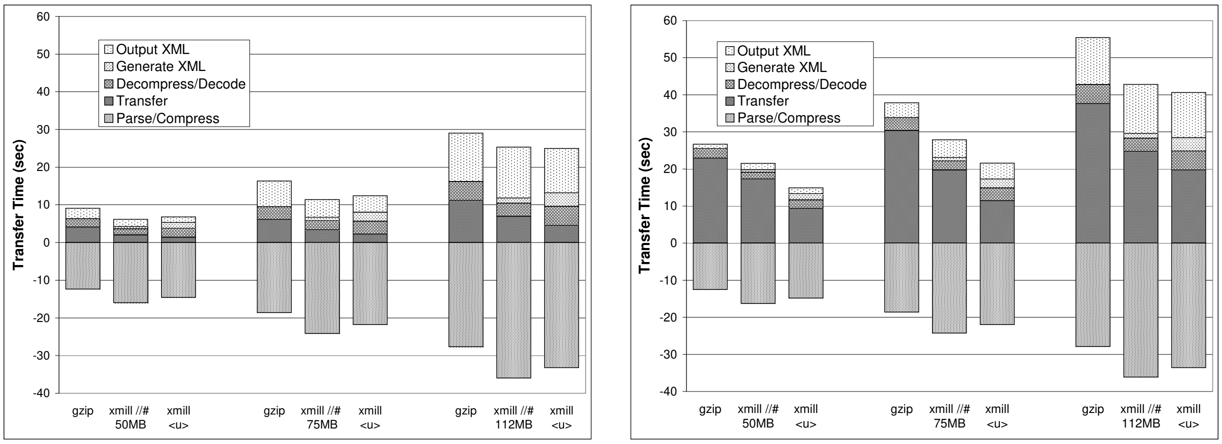
(a) (b)

Figure 7: (a) Compression under Varying Memory Windows, (b) Compression Rate vs. Time



(a) (b)

Figure 8: Compression (a) and Decompression (b) Time



(a) (b)

Figure 9: Network Transfer Time from AT&T Labs to Penn (a) and to a home PC (b)

Database Compression In databases, compression has been advocated as a method for cost reduction: to save storage space and improve processing time, based on the observation that much of the query processing time is due to I/O. A survey of database compression techniques can be found in [16]. The more recent work in [10, 6, 15] proposes techniques that allow the query processor to decompress a small unit of data at a time: one column value in the table, or one row.

Two features distinguish `XMill` from this work: `XMill` is not designed to be used in a query processor, and we do not propose a new compression algorithm, but rather offer a framework in which existing algorithm can be leveraged to compress XML data.

An interesting tool which influenced us during this project is `pzip` [1]. It compresses data files with fixed-length records very efficiently by first applying run-length encoding on mostly blank character columns and by gouping the remaining columns using its schema extraction tool before submitting it to `zlib`.

Other XML Compressors At the time of writing, a single product has been announced, by XML Solutions, called `xmlzip` (`www.xmlzip.com`). Implemented in Java, `xmlzip` cuts the XML tree at a certain depth and compresses the upper part separately from the lower part, both using `gzip`. Tested on our data sets (Fig 6), it ran out of memory on all sets except Shakespeare. There, it achieves a compression ratio between that of `gzip`'s and `XMill`, but at much lower speed.

9 Conclusions

We have described a compressor for XML data called `XMill`, which is an extensible tool for applying existing compressors to XML data. Its main engine is `zlib`, the library function variant for `gzip`. One of our targeted applications is XML data archiving, where compression rate counts alone. Here `XMill` achieves about twice the compression rate of `gzip`, at roughly the same speed, and is generally ranked best among other compressors we compared it against: `compress`, `bzip`, `xmlzip`. A second application we target is data exchange, where both compression ratio and compression/decompression time count. While `XMill` never looses to `gzip`, the size of its improvements depends on a variety of factors (type of application and relative processor/network speed), and range from none to almost a factor of 2.

Acknowledgments We would like to thank Ken Church, Mary Fernandez, Glenn Fowler, and Val Tannen for their helpful comments. Special thanks to Peter Buneman who suggested to us the name `XMill`.

References

[1] D. Belanger and K. Church. Data flows with examples from telecommunications. In *Proceedings of 1999 Workshop on Databases in Telecommunication*, Edinburgh, UK, September 1999.

[2] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, May 1994.

[4] 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`.

[5] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[6] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. IEEE Conf on Data Engineering*, 1998.

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

[8] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[9] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[10] B.R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, pages 695–704, Santiago de Chile, Chile, September 1994.

[11] H. Liefke and S.B. Davidson. An extensible compressor for XML data. *SIGMOD Record*, 29(1), March 2000.

[12] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. Technical Report MS-CIS-98-06, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, October 1999.

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

[14] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997. Available from `http://www.research.att.com/~suciu/workshop-papers.html`.

[15] W.K. Ng and C.V. Ravishankar. Block-oriented compression techniques for large statistical databases. *TKDE*, 9(2):314–328, 1997.

[16] M. A. Roth and S. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, Sept. 1993.

[17] D. Salomon. *Data Compression. The Complete Reference*. Springer, New York, 1997.

[18] C.E. Shannon. A mathematica theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. Also available in *Claude Elwood Shannon, Collected Papers*, N.J.A.Sloane and A.D.Wyner eds, IEEE Press, 1993.

[19] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. *W3C Working Draft*, September 1999. Available as `http://www.w3.org/TR/xmlschema-1`.

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