EFFECTIVE ASYMMETRIC XML COMPRESSION

Przemysław Skibiński[1], Szymon Grabowski[2], Jakub Swacha[3]

February 1, 2007

[1] University of Wrocław, Institute of Computer Science,
Joliot-Curie 15, 50-383 Wrocław, Poland
inikep@ii.uni.wroc.pl
[2] Technical University of Łódź, Computer Engineering Department,
Politechniki 11, 90-924 Łódź, Poland
sgrabow@kis.p.lodz.pl
[3] Szczecin University, Institute of Information Technology in Management,
Mickiewicza 64, 71-101 Szczecin, Poland
jakubs@uoo.univ.szczecin.pl

This is a preprint of an article accepted for publication in
Software—Practice and Experience
Copyright © 2005 John Wiley & Sons, Ltd.
http://www.interscience.wiley.com

Abstract

The innate verbosity of the Extensible Markup Language remains one of its main weaknesses, especially when large XML documents are concerned. This problem can be solved with the aid of XML-specialized compression algorithms.

In this work, we describe a fast and fully reversible XML transform which, combined with generally used LZ77-style compression algorithms, allows to attain high compression ratios, comparable to those achieved by the current state-of-the-art XML compressors. The resulting compression scheme is asymmetric in the sense that its decoder is much faster than the coder. This is a desirable practical property as in case of many XML applications data are read much more often than written. The key features of the transform are dictionary-based encoding of both document structure and content, separation of different content types into multiple streams, and dedicated encoding of numbers and dates.

The test results show the proposed transform to improve the XML compression efficiency of general purpose compressors on average by 35% in case of gzip and 17% in case of LZMA. Compared to the current state-of-the-art SCMPPM algorithm, XWRT with LZMA attains over 2% better compression ratio, being 55% faster.

KEY WORDS: Extensible Markup Language, XML compression, XML encoding, text transform.

1. Introduction

Since the advent of the Extensible Markup Language (XML) its verbosity has been viewed repeatedly as a misfeature. Notwithstanding that a big part of critique is due to hapless attempts of applying the XML where it certainly should not have been applied, the XML verbosity is a fact, a crucial one when huge data collections are concerned.

Nowadays the XML is a *real* standard, not only officially defined and endorsed, but actually used. Even though proprietary binary formats could help save time and memory, the XML is simply more convenient. And alternative formats, such as YAML [1], boasted for its simplicity, seem to be in no position to threaten this popularity.

The primary objective of our research was to design an efficient way of compressing XML documents. High compression ratio and speed were considered equally important. As we have observed that archived XML documents are more often decompressed than recompressed, we focus on attaining quick decompression rather than compression. We also require the transform to be fully reversible, so that the decompressed document is a mirror image of the original. Neglected by existing XML compressors, this property has weighty practical implications, to be discussed later.

The map of the paper is as follows. Section 2 contains a short review of existing XML compression methods, thus putting our work in a proper context. Section 3 discusses the sources of redundancy in XML documents. In Section 4 we present our transform step-by-step. The implementation details are a subject of the next section. Section 6 contains thorough experimental results and related discussion. We measure not only compression ratios, but also compression and decompression times, and simulate file transfers over a network in a range of realistic bandwidths. Section 7 gives our conclusions and points out several issues for further study.

We start, however, with pointing some remarkable features of the format we tackle with. XML is a metalanguage: the set of tags used for marking up the data is chosen by the author of a given document. In this way, various entities from the real world can be described naturally with XML tag names. XML represents a tree structure: its elements can be uniquely located by their path starting at the document root. Each element is delimited with a start tag and an end tag, between which data are usually stored in plain text or structured form. XML documents have hierarchical structure: start/end tag pairs are nested so that an outer start tag can have its ending counterpart only after all its inner tags have been closed. Note that in a file an XML document is stored as a "flattened" tree obtained from depth-first traversal.

Elements may have one or more attributes, for example (*SwissProt.xml*):

```
<Entry id="100K_RAT" class="STANDARD" mtype="PRT" seqlen="889">
 <AC>Q62671</AC>
 <Mod date="01-NOV-1997" Rel="35" type="Created"></Mod>
 <Mod date="01-NOV-1997" Rel="35" type="Last sequence update"></Mod>
 <Mod date="15-JUL-1999" Rel="38" type="Last annotation update"></Mod>
 <Descr>100 KDA PROTEIN (EC 6.3.2.-)</Descr>
 <Species>Rattus norvegicus (Rat)</Species>
```

where e.g. `Mod` is a tag and `date`, `Rel` and `type` are its attributes.

After a tag, some data and/or other tags may be located, but eventually we should come across its matching end tag. Pairless tags are possible too, they are signaled with the `/` symbol just before the closing bracket `>`. Example (*Enwikinews.xml*):

`<namespace key="0" />`.

Also, comments are possible, delimited by `<!--` and `-->` sequences.

If an XML document adheres to those simple syntactical rules, it is called well-formed. Usually there are additional constraints imposed upon XML documents. These can be defined in Document Type Definition (DTD) or XML Schema, specifying the allowed tags, attributes and their values, and the structure. We say that a document is valid according to a given DTD, if it is well-formed and conforms to the specified DTD.

The aforementioned verbosity, and hence redundancy, of many XML documents (especially large ones) is caused by highly repetitive (sub)structures of those documents and often long element and attribute names. Therefore, a need to compress XML, both efficiently and conveniently to use, has been identified nowadays as a burning research issue in the scientific community.

2. Review of existing XML compression methods

One of the first XML-oriented compressors was XMill [2] presented in 2000. It parses the XML data and splits them into three components: element and attribute symbol names, plain text and the document tree structure. As those components are typically vastly different, it pays to compress them as separate streams. XMill component streams have been originally compressed with gzip, and then [3] also with bzip2, PPMD+ and PPM*.[1] With gzip and order-5 PPMD+ the XMill transform improves compression by about 18% [3], but once higher order contexts come to play (bzip2, PPM*), the gains disappear, and it even compresses worse than the respective compressors on non-preprocessed documents. The supposed reason is that high order PPM compressors already handle the different contexts well enough, so the XMill transform helps little if at all, and on the other hand breaking the original structure makes it impossible to exploit the redundancy across components.

Although XMill makes it possible to encode each container using a dedicated method (exploiting the type of data stored within it, e.g., numbers or dates), this feature is not practical as it requires the user to choose the encoding for specific containers.

Cheney's XMLPPM [3] is a streaming compressor which uses a technique named multiplexed hierarchical modeling (MHM). It switches between four models: one for element and attribute names, one for element structure, one for attributes, one for strings, and encodes them in one stream using PPMD+ or, in newer implementations, Shkarin's PPMd. The tag and attribute names are replaced with shorter codes. An important idea in Cheney's algorithm is injecting the previous symbol from another model into the current symbol's context. Injecting means that both the encoder and decoder assume there is such a symbol in the context of the current symbol but do not explicitly encode nor decode it.

---

[1] References to all the general purpose compressors mentioned in this work can be found at `http://www.maximumcompression.com`.

The idea of symbol injection is to preserve (at least to some degree) contextual dependencies across different structural models, which was totally lost in XMill.

SCMPPM [4] can be seen as an extreme case of XMLPPM. Instead of using only few models, it maintains a separate model for each element class. Every class contains elements having the same name and the same path from the document root. This technique, called Structure Context Modeling (SCM), wins over XMLPPM on large documents (tens of megabytes), but loses on smaller files. Also, SCMPPM requires lots of memory for maintaining multiple statistical models and under limited memory scenarios it may lose significantly, even compared to pure PPMd [5].

In a recent work [5] Cheney proposed a hybrid solution (Hybrid Context Modeling, HCM), trying to combine the best features of MHM and SCM. In this algorithm initially a single model for each structural class is used, with symbol injection, i.e., it starts exactly as MHM. The novelty is to count occurrences of each element. Once the counter exceeds a predefined threshold, the given element gets its own model space, so is separated from the other elements. HCM also imposes a limit on the maximum number of models to confine memory usage on very large files. Both these parameters are chosen experimentally. Albeit sound, the HCM algorithm rarely dominates both SCM and MHM.

Several proposals (see e.g., [6] and references therein) make use of the observation that a valid XML structure can be described by context-free grammar, and grammar based compression techniques can be applied then. Grammar-based compression can be seen as generalization of dictionary-based compression, as it can identify and succinctly encode potentially complex patterns in the text. Still, this approach, albeit promising, so far has not yielded compressors competitive e.g. to XMLPPM in the compression ratio.

A recent trend in XML compression is to support queries directly in the compressed representation. The pioneering work in this domain was XGRind [7], while probably the most advanced solution at the moment is XBzip [8]. Although this scheme is quite impressive in both compression ratio and search/navigation capabilities, it loses to SCMPPM in compression ratio even if no support for queries is implemented. Together with auxiliary structures for searching, it sometimes needs even more space than a respective gzip archive, at least with the default settings (cf. Table 2, XBzipIndex column, and Fig. 1 (top) in [8]).

Yet another line of research is to construct DTD- or Schema-aware compressors (XCQ, ICT XML-Xpress™). Taking into account that the syntax of the document is already stored in a DTD, impressive compression ratio can be obtained, provided a DTD for a given XML document is available for both compressor and decompressor, and the given document fully conforms to it. Theoretically, this could be the best way to handle XML compression, but in practice XML documents with unavailable (or even undefined) DTD are often used, and document structure can be rearranged, raising the issue of incompatibility between compressor and decompressor.

In this paper we shall address neither the problem of making the compressed XML queriable, nor using DTD in the process. Instead, we shall focus at devising a method to store XML in a very compact form, and in a way as simple and fast as possible.

3. Handling the redundancy of XML documents

The initial thought that moved our attention to the preprocessing approach was that if XML is a kind of text, then a word-replacing scheme using a static dictionary of the most frequent English words should improve compression ratio more or less as much as it does with plain text [9]. Indeed, the scheme worked nicely with XML-marked textual documents. As expected, it did not help with compressing XML documents dominated by numerical data. The real disappointment was that it did not help much compressing most XML documents, that is those with a mixed textual-numerical content.

A simple investigation revealed that in fact, in most XML documents, there is a short list of words which appear throughout the document with extremely high frequency. This is particularly the case with tag and attribute names, but attribute values or some of the element content words can also appear many times.

The problem is that the lists of frequent words vary greatly between XML documents with different content type. This is why the static dictionary turned out to be quite a failure. Yet this redundancy can be exploited with the help of a semi-dynamic dictionary. The frequent words can be extracted in a preliminary scan over the document to form the dictionary. Then, every time a dictionary word is found in the document, it can be replaced by its dictionary index. Notice that this happens very often, as the dynamic dictionary contains words that are actually frequent in the document, not words that could potentially be frequent, as was the case with a static dictionary. If encoded properly – and we shall show later what we mean here – dictionary indices are always shorter than the words they reference. The only drawback of a semi-dynamic dictionary is that it must be stored explicitly within the preprocessor output. Still, as the words included in the dictionary are highly frequent, the size (in bytes) of the stored dictionary is relatively small compared to the total encoded length of the document.

Another issue to solve is how to encode numbers. They appear in the great bulk of XML documents, as many fields in real-world databases are numeric. Storing numbers as text is, roughly spoken, ineffective. Numbers can be encoded more efficiently using a numerical system with base higher than 10. Moreover, we noticed that numbers – slightly different among themselves – were often the only non-repetitive element within a very repetitive context. We can eliminate this unpredictability by removing the numbers to another data stream, marking only the place of their appearance in the original XML document. Now, the document content becomes more repetitive, and the numbers in the separate stream can be encoded as densely as possible, using a numerical system with base 256.

Universal compression algorithms treat an XML document as a stream of homogenous data. The compression model in universal algorithms is built on the correlation between adjoining symbols. Such approach is far from optimal though, as XML is actually a mix of several data streams, each one with structural properties of its own, and there is a lot of correlation between symbols which are distant but syntactically related.

Knowing the difficulty that universal compression algorithms have coping with this situation, an easy solution would be to separate the streams. This can be done by moving the contents of each XML element class to a specific place in the output file. Thus, the

contents of the same element class are kept together, away from the contents of different elements.

In a well-formed XML document, every end tag must match the corresponding start tag. This can hardly be exploited by general-purpose compression algorithms, as they maintain a linear, not stack-alike data model. The compression ratio can then be increased by replacing every matching end tag with merely an element closing flag.

The following two simple techniques deal with redundancy in the layout of XML documents. Although they are not relevant for all XML documents, they provide noticeable gains when compressing documents they were aimed at.

The first one makes use of structural indentation by efficiently encoding the leading blanks in lines. This kind of redundancy, typical to XML documents created with editors caring about the output format, cannot be well exploited by general-purpose compression models. The second technique works for those documents in which an end tag is usually followed with a newline character. Such a concatenation can be reduced to a single symbol.

4. XML Word Replacing Transform

In this section we introduce XML Word Replacing Transform (XML-WRT, or XWRT for short) through detailed description of its main constituents.

XWRT is lossless. Perhaps surprisingly, few of the presented XML compressors in the literature are truly lossless. This is because the document layout (e.g., trailing spaces) is often ignored as it carries no meaning. Still, preserving the layout may be useful for human editors of a document. The exact fidelity of the decompressed document with the original is also required in order to verify the document integrity using a cyclic redundancy check or hash functions. These are the reasons for which we have developed a truly lossless approach, following a long tradition of text compression.

Our transform is designed primarily for LZ77-style [10] compression. We have chosen LZ77 to keep the decompression fast. There are context-aware algorithms, PPM [11] in particular, that achieve higher compression efficiency, but the price is much slower decompression and much higher memory requirement. Compressors from the LZ77 family differ significantly from the context-aware schemes. Basically, they parse input data into a stream of matching sequences and single characters. The matches are encoded by specifying their offsets, i.e., distances to the previous sequence occurrence in a limited past buffer, and lengths, while the literals are encoded directly, in most variants with Huffman coding. There have been many refinements since the seminal works of Ziv and Lempel [10] and Storer and Szymanski [12], but the overall picture has not changed. LZ77 compressors do not predict characters on the basis of their context, or, in some modern variations, they do it only in a very low order rudimentary manner. The strength of LZ77 lies in succinct encoding of long matching sequences and high decoding speed. In consequence, an XML preprocessor adapted for LZ77 compressors should attempt to:

- reduce the number of characters to encode;
- decrease the offset (in bytes) of the matching sequences;

- decrease the length (in bytes) of the matching sequence;
- virtually increase the sliding window, i.e., the past buffer in which matching sequences are looked for.

It appears that in case of LZ77 (but not necessarily PPM or BWT), shortening the output of the preprocessor has clear positive effect on the final compression ratio.

The architecture of XWRT implementation is portrayed in Fig. 1.

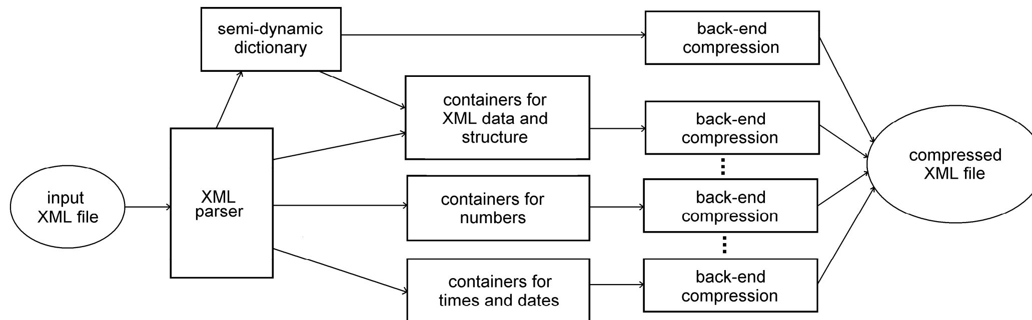

Figure 1: The general XWRT operating scheme

The dictionary

The backbone of our transform is to replace the most frequent words with references to a dictionary. Our notion of a "word" is broader than its common meaning, and will be described below. The XWRT dictionary is obtained in a preliminary pass over the data, and contains sequences of length at least $l_{\min}$ characters that appear at least $f_{\min}$ times in the document. The dictionary is sorted according to the frequency of occurrence of words.

A dynamic dictionary was previously used in XMill [2], but XMill allows only the element and attribute names as dictionary entries, while the XWRT dictionary contains items from the following classes:

- ordinary words – sequences of lowercase and uppercase letters (a-z, A-Z) only,
- start tags – sequences of characters that start with <, contain letters, digits, underscores, colons, dashes, or dots, and end with >. Start tags can be preceded by one or more spaces as XML documents usually have regular arrangements of the lines, in which individual tags very often begin in the same column, preceded with the same number of spaces,
- URL address prefixes – sequences of the form `http://domain/`, where `domain` is any combination of letters, digits, dots, and dashes,
- e-mails – patterns of the form `login@domain`, where `login` and `domain` are any combination of letters, digits, dots, and dashes,

- words in form `"&data;"`, where `data` is any combination of letters, representing XML entities,
- special digrams – sequences `="` and `">`, which appear very frequently with attribute values,
- runs of spaces – sequences of spaces that are not followed by start tags (for documents with regular layouts).

As noted in the previous section, using a static dictionary is problematic because of the hardness to select a proper set of words, relevant across a wide range of real-world XML documents. To give an idea of this difficulty, we note that many tags are word shortcuts or short word phrases glued according to various conventions (e.g., lastname, lastName, LastName, last_name, LAST_NAME, etc). It is practically impossible to foresee and accommodate them all in a predefined dictionary of limited size.

The words selected for the dictionary are written explicitly, with separators, at the beginning of the output file. For most documents the dictionary contains not more than several hundred items, hence the codewords have one or two bytes.

Dictionary references are encoded using a byte-oriented prefix code. Although it produces slightly longer output than, for instance, Huffman coding, the resulting data can be easily compressed further, which is not the case with the latter.

Actually, XWRT uses two dictionary encoding schemes: one optimized for traditional LZ77 compressors (e.g., gzip) and a less dense variant, with non-intersecting ranges for different codeword bytes, for modern LZ77 compressors (e.g., LZMA). Both schemes employ three kinds of codewords: 1, 2, and 3 bytes long. In the gzip-optimized scheme, the first byte of the codeword can belong to one of three disjoint ranges:

(*i*) C_1 if it is a one-byte long codeword; there are $|C_1|$ such codewords available,

(*ii*) C_2 if it is a prefix of two-bytes long codeword, followed by a single byte in the full possible value range; there are $|C_2| * 256$ such codewords available,

(*iii*) C_3 if it is a prefix of three-bytes long codeword, followed by two bytes in the full possible value range; there are $|C_3| * 256 * 256$ such codewords available.

In this way, we obtain $|C_1| + |C_2| * 256 + |C_3| * 256 * 256$ codewords in total. As this is a kind of prefix code, all the codewords are immediately decodeable. The size of ranges C_1, C_2, and C_3 are set according to the size of the document to compress and the resulting dictionary size.

In the LZMA-optimized scheme we use only two disjoint ranges of bytes, C_1 and C_2, but the codeword lengths still span from 1 to 3 bytes (as in the code described in [9]). Any codeword byte from the range C_1 is unambiguously recognized as the suffix byte. In this way, we have $|C_1|$ one-byte codewords, $|C_2| * |C_1|$ two-byte codewords, and $|C_2| * |C_2| * |C_1|$ three-byte codewords. Such a reversed byte order was found to improve compression ratio.

As a side benefit, our dictionary encoding, with separate alphabets for original words and codewords, makes it straightforward to encode only a prefix of a word, if the prefix matches some word in a dictionary but the whole word does not. Still, the gain we achieve in this way is insubstantial.

Our scheme applies the spaceless word model [13], in which single spaces before encoded words are omitted, as they can be automatically inserted on decoding.

Element containers

Another important concept is separating the XML document structure from the content [2]. Contents of the same element should be grouped together, as this implies better LZ77 compression, since the container contents may form long repetitive patterns, the sliding window contains more matches to be found, and match offsets are reduced. Sometimes, element contents may have more in common with contents of the different neighboring elements than with contents of similar elements elsewhere in the document. However, in practice, cases of XML documents subject to LZ77-type compression, which show no gain after such treatment, are rare.

In the current implementation containers are identified only by their element names, not by the whole path. For example, the contents of `root/help` and `root/extra/help` elements will be moved to the same container. This mode of operation was found to produce higher compression ratios than the one using containers identified by full paths (as in SCMPPM).

The container streams are arranged in lexicographical order of the names of their elements. It is conceivable that ordering them according to the characteristics of their contents would yield some compression gain with small containers, but mostly with back-end compressors like gzip, suffering from a seriously limited history buffer, and thus sensitive to spatial location of similar data in their input. Therefore, we may change this design concept in the future XWRT releases.

In the gzip mode, the containers are compressed separately, one after another, whereas in the LZMA mode they are all concatenated and then compressed as a single data stream. Both choices were based on empirical observations showing improved compression ratios.

Number encoding

Another idea in XWRT is the compact encoding of sequences of digits. Any sequence of digits (even of length 1) is replaced with a flag, which remains in the main output stream, whereas the actual value is put into a separate container, encoded as a base-256 number. Although we have observed that for some datasets slightly better results were obtained with other radix bases, such as 64 or 100, using the densest possible encoding seems the best choice on average.

The flag is a single character from range '1' to '4', which identifies a digit sequence and tells the length (in bytes) of its encoded value. As only 256^4 integers can be stored on four bytes, sequences representing larger numbers are simply split into several shorter ones (a rare case in practice). If the digit sequence starts with one or more zeroes, the initial zeroes are left intact in the text.

This technique was found to improve compression ratio for each tested compression scheme. A positive side-effect of this encoding is that the symbols '5'…'9' are no longer reserved for the input alphabet in the main output stream, and thus can be used for other purposes, to be described below.

Some numerical data represent specific information types, such as dates. One of the most common date formats, which can also be found in XML documents, is YYYY-MM-DD (Y for year, M for month, D for day). The aforedescribed "general" integer encoding would require five bytes in the main stream (including two separating hyphens) plus four bytes in the number stream, to encode any such date.

Assuming for simplicity that each month can have up to 31 days, all the dates in the interval from 1977-01-01 to 2153-02-26 can be represented as an integer from the range 0-65535. Therefore, a date can be encoded on just three bytes: one flag (we chose the unused symbol '5') in the main stream, and a two bytes long integer in a separate stream.

Other date formats sometimes also appear, or only the year is given. This case is handled separately, namely any integer number in the range from 1900 to 2155 is considered to be a year, and encoded appropriately. The flag in the main stream is '6', while a single byte in a separate container tells the difference between the number and 1900.

A similar idea is utilized to encode the time. XWRT recognizes numbers from 1 to 12 followed by the suffix "am" or "pm" (e.g., "11:30pm"), and encodes them as pairs of bytes: the first byte is the hour in the 24-hour convention, the other specifies the minutes. The two bytes are saved in two separate containers, while in the main stream, as usual, a unique flag signals the encoding done.

Some documents, e.g. containing bibliographical information, are flooded with page ranges. They are usually in the format "x–y" (e.g., "120–148"). A simple observation is that usually $0 < y - x < 256$. If this happens, the value of x, encoded in two bytes, is sent to one stream, and the 1-byte difference $y - x$ is sent to another stream. The last numerical category which, in our opinion, deserves special treatment, are fractional numbers. If we stayed only with the number encoding described above, a fractional number of the form, e.g., *dddd.ddd* (where *d*'s stand for any digit) would be encoded as two integers (*dddd* and *ddd*), with the separator (dot) in between. Now, we handle separately *some* fractional numbers: those with exactly two digits after the decimal point and some of those with one digit after the point. Namely, numbers from 0.0 to 24.9 (one fractional digit) are replaced with a flag, and their value, encoded on a single byte, is sent to a separate stream. For the case of any number with two fractional digits (e.g., "102.00", "95.01"), its suffix, starting from the decimal point, is replaced with another flag, and those two last digits are encoded on a single byte and sent to yet another stream.

5. Implementation details

The input data are processed by a finite state automaton (FSA), which accepts proper words and numerical (including date and time) expressions. The crucial operation in the encoding is dictionary search. A search function is called twice for each word in the document: first time during the dynamic dictionary buildup, second time during the actual parsing and word encoding. The choice of a dictionary data structure can seriously affect the overall XWRT performance. We have decided to use a fixed-size (4 MB) array with chained hashing for search, which we previously tested in our work on WRT [9]. Its advantages are simplicity, moderate memory usage, and $O(1)$ search time (assuming that a single word is read in constant time).

For the semi-dynamic dictionary, we allocate 8 MB of memory. If the dictionary reaches that limit, it is frozen, i.e., the counters of already included words can be incremented but no new word can be added. Still, in practice we rarely get close to the assumed limit (which can also be changed with a program switch).

The codeword space should not be wasted for very short or very rare words. Therefore, building the dictionary depends on two parameters: the minimum word length l_{min} , and the minimum number of word (i.e., dictionary entry) occurrences in the text, f_{min}. Only those words which are not shorter than l_{min} and have at least f_{min} occurrences in the text are included in the dictionary. Empirical observations led us to set $l_{min} = 2$ and $f_{min} = 6$, as these values gave good results for most files.

As explained in the previous section, apart from the element containers, the XWRT transform can produce up to 26 data streams (for storing numbers etc.), buffered in memory. When the buffer gets full, it is flushed, with its contents saved either uncompressed (but transformed), or with prior zlib or LZMA compression, depending on the processing mode used. The buffer size is 8 MB by default but can be changed with a program switch.

The reverse XWRT is simpler. Again we use an FSA, which now recognizes XWRT flags and codewords, and transforms them to the original form. Obviously, there is no real search in the dictionary, only lookups in $O(1)$ time per codeword.

The XWRT transform was implemented in C++ and compiled with MS Visual C++ 2005.

6. Experimental results

In order to compare the performance of our algorithm to the existing XML compressors, as well as several popular or otherwise interesting general-purpose compressors, a set of experiments has been run. In compression benchmarking, proper selection of datasets used in experiments is essential. To the best of our knowledge, there is no publicly available and widely respected XML dataset corpus to this date. We could not use any published corpus in an exact form due to unavailability of some datasets [14], and presence of datasets producing completely unreliable compression results, such as the randomly generated XMark [14] (allowing XWRT to attain astonishingly high compression ratios due to its small vocabulary of words) and the encrypted Treebank [15] (thwarting XWRT effectiveness due to its lack of repeatable words). We have based our corpus on datasets from [14], [15], adding two Wikipedia database files from [16].

As a result, our experimental corpus consists of:

- *DBLP*, bibliographic information on major computer science journals and proceedings,
- *Enwikibooks*, a collection of free content textbooks,
- *Enwikinews*, a news collection,
- *Lineitem*, business order line items from the 10 MB version of the TPC-H benchmark,
- *Shakespeare*, a corpus of marked-up Shakespeare plays,
- *SwissProt*, a curated protein sequence database,
- *UWM*, university courses.

We dubbed our file collection *Wratislavia XML Corpus*. The references to all its files are given at [17]. Table 1 presents detailed information for each dataset: its size, the

number of elements, the number of attributes, and the maximum depth. Note that the total collection size is almost half a gigabyte.

Table 1. Basic characteristics for the XML datasets used in the tests

Dataset	File size	Elements	Attributes	Max. depth
DBLP	133 862 735	3 332 130	404 276	6
Enwikibooks	156 300 597	533 698	49 115	5
Enwikinews	46 418 850	278 670	24 607	5
Lineitem	32 295 475	1 022 976	1	3
Shakespeare	7 894 983	179 690	0	7
SwissProt	114 820 211	2 977 031	2 189 859	5
UWM	2 337 522	66 729	6	5
Sum	493 930 373	–	–	–

The test machine was an Intel Core 2 Duo E6600 2.40 GHz system with 1 GB memory and two Seagate 250 GB SATA drives in RAID mode 1, running Windows XP 64-bit edition. We tested the current version of software implementing our transform, XWRT 2.1, available from [18].

For a reference, we also present results of the preliminary version of XWRT (1.0) [19]. Apart from many minor differences between those two versions, the main distinction lies in the lack of element containers and embedded compression engines in the old version.

In the first experiment we tested the impact of the individual XWRT (2.1) transform components onto the compression ratio in comparison with a single compressor – gzip 1.2.4 working with default settings. Table 2 contains the results. The first row is just for giving an idea to what degree the complete transform (without applying a back-end compressor) shortens the input XML documents. As we see, the "compression" is on the order of 70%, but varies significantly from file to file. The second row shows the results of the gzip compression, and the following rows present the improvement to gzip compression after adding the XWRT components, one by one. The percentages express the differences to the previous row.

Table 2. XWRT transform, step-by-step compression improvement

File →	*DBLP*	*Enwiki-books*	*Enwiki-news*	*Lineitem*	*Shakes-peare*	*Swiss-Prot*	*UWM*	Average
XWRT	69.09%	58.09%	59.17%	85.89%	71.06%	65.62%	81.67%	70.08%
gzip	81.66%	70.76%	71.84%	90.91%	72.64%	87.65%	93.07%	81.22%
+ dynamic dict.	31.80%	23.14%	26.13%	41.54%	28.61%	33.02%	30.66%	30.70%
+ EOLs and spaces	0.00%	0.00%	0.05%	0.00%	0.00%	0.14%	2.07%	0.32%
+ numbers	4.33%	1.91%	3.19%	26.72%	0.02%	13.90%	7.02%	8.16%
+ letter sequences	1.83%	1.38%	1.73%	0.99%	1.38%	3.23%	3.39%	1.99%
+ containers	8.01%	0.33%	-0.51%	10.41%	3.88%	13.71%	6.80%	6.09%
+ static dict.	-0.03%	0.16%	0.60%	-0.03%	1.57%	-0.48%	3.07%	0.69%
Together, relative to gzip	41.07%	26.00%	29.83%	61.99%	33.41%	51.68%	44.90%	41.27%

A few observations can be done. Although the input files are already well compressible with general-purpose compressors, our transform yields further significant gains on all the files. In two cases (*Lineitem* and *SwissProt*) the gzip file after the XWRT transform was

more than halved than without it! From the transform components, the most important is the dynamic dictionary responsible for over 70% of the total gain, on average. Special handling of End-of-Line symbols and trailing spaces had a negligible effect: only 0.32% on average and no gain at all on as many as four files. Succinct encoding of numbers was the second most successful idea, with gains over 8% on average and over 26% in the best case. It was of no use only for *Shakespeare*, a purely textual document. Sending non-dictionary-encoded sequences of letters to another stream had mediocre but quite stable positive effect over our collection. Finally, grouping the data into containers gives an average extra gain of 6%. The mentioned components constitute the XWRT transform. However, we were also curious if using an external dictionary of English words could significantly boost the compression. The answer is negative: the average gain was very little, below 1%, and in the light of the obvious drawbacks of a static dictionary, we decided to leave it apart.

In the following experiments, data transformed with XWRT were passed to several general-purpose compression tools: gzip, LZMA, bzip2, and grzip2. First we are going to present briefly those compressors and justify our choice. With one exception (pointed below), default settings were always used.

Table 3. Compression ratios in bits per characters

	gzip	XWRT1 +gzip	XWRT2 -l2 (gzip)	bzip2	XWRT2 +bzip2	grzip2 -m4	XWRT2 +grzip2
DBLP	1.463	1.029	0.864	0.955	0.789	0.861	0.750
Enwikibooks	2.339	1.795	1.734	1.891	1.702	1.724	1.583
Enwikinews	2.248	1.660	1.590	1.675	1.463	1.500	1.321
Lineitem	0.721	0.488	0.276	0.335	0.269	0.419	0.268
Shakespeare	2.182	1.560	1.481	1.491	1.350	1.423	1.288
SwissProt	0.985	0.675	0.475	0.608	0.430	0.516	0.433
UWM	0.553	0.383	0.315	0.321	0.305	0.346	0.291
Average	1.499	1.084	0.962	1.039	0.901	0.970	0.848

	LZMA -a1	XWRT2 -l6 (LZMA)	XMill 0.9 zip	XMill 0.9 bzip2	XMill 0.9 ppm	XMLPPM -l9	SCMPPM -l9
DBLP	0.943	0.747	1.250	0.955	0.940	0.802	0.693
Enwikibooks	1.686	1.505	2.295	1.914	1.838	1.621	1.621
Enwikinews	1.462	1.300	2.198	1.722	1.746	1.379	1.398
Lineitem	0.421	0.243	0.380	0.264	0.270	0.261	0.242
Shakespeare	1.646	1.348	2.044	1.575	1.584	1.295	1.293
SwissProt	0.478	0.388	0.619	0.489	0.477	0.416	0.417
UWM	0.368	0.278	0.382	0.317	0.310	0.259	0.274
Average	1.001	0.830	1.310	1.034	1.024	0.862	0.848

Gzip is based on Deflate, the most widely-used compression algorithm, known for its fast compression and very fast decompression, but limited efficiency. Deflate uses Huffman coding for literals, match lengths and match offsets. The buffer (*sliding window*) for finding matches has only 32 KB, which is mostly responsible for both very high compression speed and mediocre compression ratios. We used gzip 1.2.4 in the tests. Another representative of the LZ77 compression family is LZMA (version 4.43). It uses a proprietary compression method, also implemented in the better known 7zip compression

utility, respected for its high efficiency and fast decompression, but slow compression. Some of the major traits of LZMA are sophisticated match parsing, working with large buffers, and low order contextual encoding of literals.

Bzip2 (version 1.0.2) is certainly the only compressor based on the Burrows–Wheeler transform which gained a wide acceptance outside the compression community. It is characterized by high text compression ratios and quite fast decompression. The compression speed varies, but is typically acceptable. Grzip2, version 2.4, is a newer piece of software, avoiding some of bad design decisions of bzip2. By default it submits the input data to a fast LZ preprocessor, and then runs block sorting compression. In the tests, its fastest mode, -m4, was used, in which Schindler's Sort Transform [20] is performed instead of BWT after the LZ preprocessing; we also set the maximum block size (8 MB).

In Table 3, the compression results obtained for XWRT-transformed datasets are compared to those achieved by the same compression algorithms on the datasets in their original form. Existing XML-aware compressors are represented in the results with the fast XMill 0.9, and the current state-of-the-art XML compressors, XMLPPM 0.98.2 and SCMPPM 0.93.3. We were not able to test XBzip, as it crashes on files larger than 100 MB on our test machine.

It was found during the tests that XMLPPM is not truly lossless (it did not reproduce exactly any of the seven test files in the decompression), and XMill fails to exactly reproduce DBLP file.

Table 4. Compression times in seconds

	gzip	XWRT1 +gzip	XWRT2 -l2 (gzip)	bzip2	XWRT2 +bzip2	grzip2 -m4	XWRT2 +grzip2
DBLP	6.05	11.89	13.64	31.66	22.51	5.92	14.20
Enwikibooks	11.55	15.58	17.15	29.52	26.79	11.14	21.07
Enwikinews	3.02	4.45	4.95	9.27	8.43	2.94	5.87
Lineitem	1.05	2.05	1.90	9.52	3.89	1.09	2.21
Shakespeare	0.64	0.86	0.90	1.77	1.26	0.59	1.07
SwissProt	3.67	9.63	10.48	27.66	24.64	3.28	11.65
UWM	0.06	0.16	0.17	0.77	0.37	0.06	0.20
Average	26.04	44.61	49.22	110.15	87.92	25.03	56.31

	LZMA -a1	XWRT2 -l6 (LZMA)	XMill 0.9 zip	XMill 0.9 bzip2	XMill 0.9 ppm	XMLPPM -l9	SCMPPM -l9
DBLP	120.49	33.31	10.63	26.26	17.18	23.13	52.13
Enwikibooks	152.06	50.53	16.75	27.06	32.73	38.55	54.36
Enwikinews	43.47	15.25	4.77	8.37	9.39	10.41	16.17
Lineitem	40.77	3.90	2.34	3.47	1.63	2.66	13.75
Shakespeare	8.00	6.78	0.94	1.91	1.50	2.02	3.55
SwissProt	82.83	31.28	7.94	31.74	9.91	13.60	39.21
UWM	1.61	0.40	0.13	0.52	0.16	0.19	0.84
Average	449.23	141.54	43.49	99.32	72.49	90.54	180.01

It is apparent from the results that the compression ratio has been improved by transforming XML data, on average, by over 35% in case of gzip, 17% for LZMA, and 13% for non-LZ77-based compressors (bzip2 and grzip2). Compared to the existing XML-

specialized compressors, XWRT combined with gzip is on average 27% better than XMill's zip mode, whereas XWRT combined with LZMA is on average 20% better than XMill's bzip2 mode. As for the so far best available XML compressors, XWRT with LZMA beats XMLPPM by 3.7% and SCMPPM by 2.1%, on average.

Compression and decompression times are presented in Table 4 and 5, respectively. The decompression times usually grow if XWRT is applied, but we think that the compression gains justify it. When XWRT is used with LZMA, the compression time is greatly reduced, threefold on average, and by over 10 times in the extreme case of the *Lineitem* file.

Looking at the combined compression and decompression times, XWRT with gzip is on average 20% slower than XMill's zip mode (remember the 27% superiority in compression ratio), whereas XWRT with LZMA is on average 35% slower than XMill's bzip2 mode (for 20% better compression ratio). XWRT with LZMA is faster than SCMPPM by 55% on average (beating it by 2.1% in compression ratio). For the biggest file that XMLPPM was able to decompress, *DBLP*, XWRT with LZMA is faster by 27.8% (attaining 6.5% better compression ratio).

Table 5. Decompression times in seconds

	gzip	XWRT1 +gzip	XWRT2 -l2 (gzip)	bzip2	XWRT2 +bzip2	grzip2 -m4	XWRT2 +grzip2
DBLP	3.08	7.25	4.46	6.44	6.90	9.50	9.89
Enwikibooks	1.64	8.91	4.70	9.75	9.61	17.27	17.53
Enwikinews	0.50	2.50	1.50	2.78	2.70	4.78	4.79
Lineitem	0.24	1.77	0.96	1.48	1.28	1.72	1.47
Shakespeare	0.09	0.31	0.26	0.50	0.46	0.81	0.78
SwissProt	2.31	6.27	3.89	5.03	5.45	5.49	6.68
UWM	0.03	0.09	0.07	0.09	0.10	0.08	0.11
Average	7.89	27.10	15.87	26.08	26.53	39.64	41.27

	LZMA -a1	XWRT2 -l6 (LZMA)	XMill 0.9 zip	XMill 0.9 bzip2	XMill 0.9 ppm	XMLPPM -l9	SCMPPM -l9
DBLP	4.31	3.79	2.94	5.41	17.16	28.25	50.83
Enwikibooks	6.47	7.82	3.08	7.83	37.17	–	58.57
Enwikinews	1.74	2.18	0.49	2.27	10.44	–	17.27
Lineitem	0.80	1.04	0.39	0.72	1.55	3.87	13.03
Shakespeare	0.34	0.45	0.11	0.44	1.70	2.51	3.61
SwissProt	2.92	4.64	3.50	4.05	9.98	18.38	35.25
UWM	0.06	0.09	0.06	0.08	0.16	0.28	0.81
Average	16.64	20.04	10.56	20.78	78.15	–	179.37

Finally, we examined the practical impact of applying compression to large XML documents published on the Internet. To an end user, a practical measure of compression "usability" can be the total time it takes to fetch a compressed document, and then decompress it. Of course, this measure, let us call it *document delivery time*, depends on two factors: computer processing speed and available network bandwidth. Below, we present the document delivery time calculated for a set of five of the seven test files (*Enwikibooks* and *Enwikinews* were omitted due to problems with XMLPPM). The results were obtained by adding transmission and decompression times of every file; they do not

include lesser real-world factors such as connection setup time, transfer protocol overheads, etc. The decompression times were measured on the same machine as in the remaining tests (Intel Core 2 Duo E6600 2.40 GHz). Fig. 2 shows results simulated for a 8 Mb/s network connection, Fig. 3 for a 512 kb/s network connection.

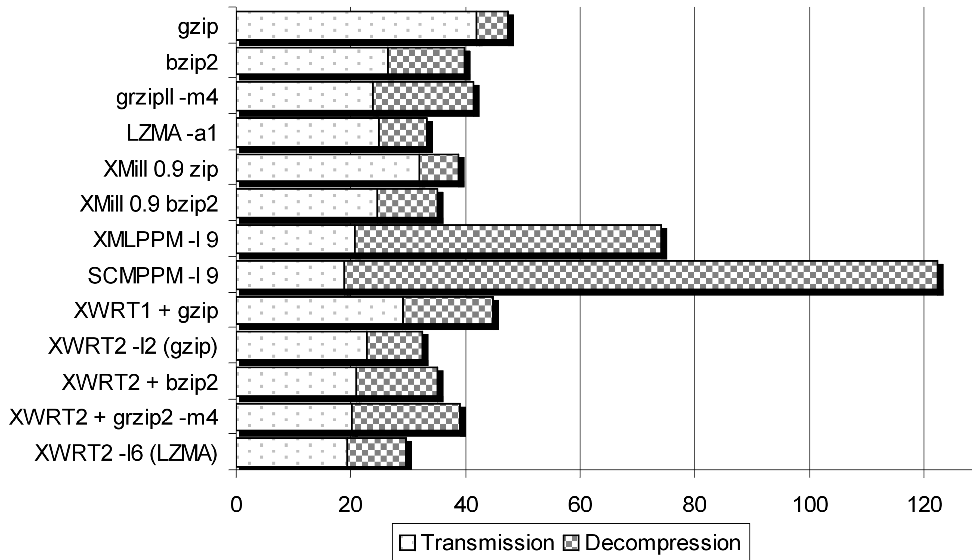

Figure 2: The average speedup for the bandwidth of 8 Mbit/s

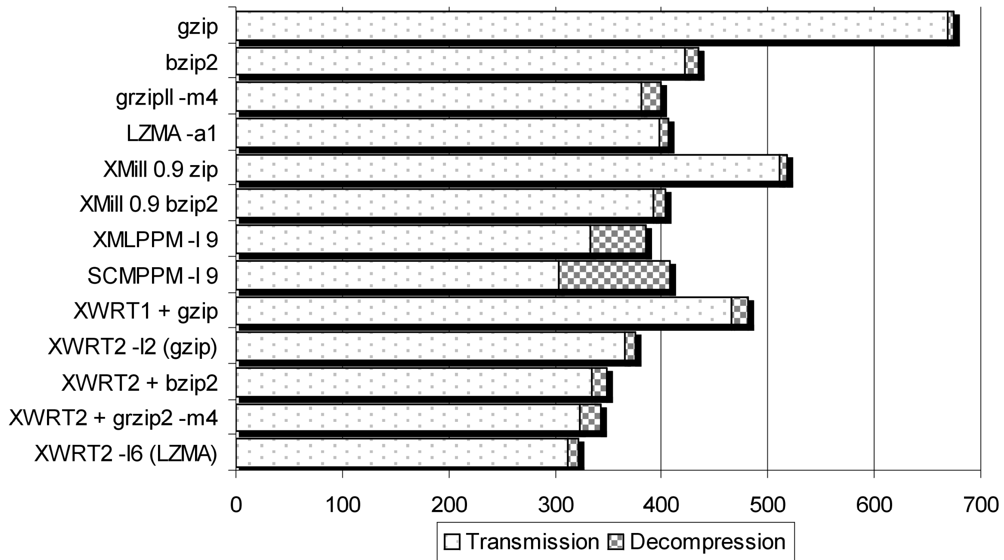

Figure 3: The average speedup for the bandwidth of 512 kbit/s

We could not show that on the graphs (and keep a readable scale), but most large XML databases are so redundant that even the simplest compression using the bare gzip results in about seven-fold document delivery speedup compared to transmission without compression.

Still, specific compression algorithms fare even better. For both of the investigated network throughputs, the fastest scheme is XWRT 2.1 -l6, i.e., the proposed transform with LZMA as the back-end compressor. As it can be easily noticed, the lower is the transmission speed, the more important is the compression ratio, and the higher is the transmission speed, the more important is the processing time. XWRT (either with gzip or LZMA) is the clear winner for the entire range of typical Internet bandwidths. It yields to the efficient SCMPPM only for speeds lower than about 128 kb/s, and to the fast gzip for speeds higher than about 100 Mb/s.

For the two investigated network throughputs, applying the proposed transform was found to be helpful for each of the four investigated back-end compressors, attaining average relative improvement of, respectively, 38% for gzip, 17% for bzip2, 16% for LZMA, and 13% for grzip2. The highest relative improvement measured for gzip is good news, considering gzip's widespread popularity and low computational and memory requirements.

7. Conclusions and future work

We have presented a fast XML transform aiming to improve lossless XML compression in combination with existing general purpose compressors. We focused on fast decoding of a compressed document, i.e. the reverse transform is not only fast, but it is also optimized for the LZ77-style compression characterized by very fast decompression. The main components of our algorithms are: a semi-dynamic dictionary of frequent alphanumerical phrases (not limited to “words” in a conventional sense), splitting document contents into many dedicated containers, and binary encoding of numbers and dates.

Thanks to the proposed transform, the XML compression ratio of a widely-used LZ77-type algorithm, Deflate (used by default in gzip and zip), can be improved by as much as 35%. Although the decoding speed gets twice worse, the longer decompression time is more than compensated by shorter transmission time for typical Internet transfer rates. To this end, XWRT with LZMA as the back-end compressor achieves the shortest document delivery time across a vast range of realistic bandwidths.

It appears from the results that the main advantage of our algorithm over the competitors comes from applying the dictionary encoding not only for structural elements (tags, attributes) but also to the textual content. We expect that relaxing the rules for the items in our dictionary (e.g., accepting pairs of words, full URL paths, more timestamp formats, etc.) could produce even better results for the price of further complication of the transform.

There are a couple of ways likely to increase the compression further. One is using container-related dictionaries. Another is order-0 entropy encoding of the contents of those elements which have few distinct values in the document. Preliminary experiments suggest that for some databases, like *Lineitem*, the gain can be very significant. Some little gain can also be obtained by more specific encoding of the list of words in the (dynamically built)

dictionary. Another line of our research is to optimize the transform for PPM compression, to increase its advantage over XMLPPM and SCMPPM even more.

REFERENCES

[1] YAML home page. http://www.yaml.org.

[2] Liefke H, Suciu D. XMill: an efficient compressor for XML data. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, USA, 2000, 153–164.

[3] Cheney J. Compressing XML with multiplexed hierarchical PPM models. In Storer JA, Cohn M, editors, *Proceedings of the 2001 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2001; 163–172.

[4] Adiego J, Navarro G, de la Fuente P. Using Structural Contexts to Compress Semistructured Text Collections. *Information Processing and Management* 2007; 43:769–790.

[5] Cheney J. Tradeoffs in XML Database Compression. In Storer JA, Cohn M, editors, *Proceedings of the 2006 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 2006; 392–401.

[6] Leighton G. Two New Approaches for Compressing XML. M.Sc. Thesis. Acadia University, Wolfville, Nova Scotia, 2005. (Also available at http://cs.acadiau.ca/~005985l/MThesis.zip.)

[7] Tolani P, Haritsa J. XGRIND: a query-friendly XML compressor. *Proceedings of the 2002 International Conference on Database Engineering*, San Jose, CA, USA, 2002, 225–234.

[8] Ferragina P, Luccio F, Manzini G, Muthukrishnan S. Compressing and Searching XML Data Via Two Zips. *Proceedings of the 2006 International World Wide Web Conference (WWW)*, Edinburgh, Scotland, 2006, 751–760.

[9] Skibiński P, Grabowski Sz, Deorowicz S. Revisiting dictionary-based compression. *Software–Practice and Experience* 2005; 35(15):1455–1476.

[10] Ziv J, Lempel A. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory 1977; IT-23:337–342.

[11] Cleary JG, Witten IH. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 1984; COM-32(4):396–402.

[12] Storer J, Szymanski TG. Data compression via textual substitution. *Journal of the ACM* 1982; 29:928–951.

[13] Moura ES, Navarro G, Ziviani N. Indexing Compressed Text. In Baeza-Yates R, editor, *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, Valparaiso, Carleton University Press, 1997; 95–111.

[14] Ng W, Lam W-Y, Cheng J. Comparative Analysis of XML Compression Technologies. *World Wide Web* 2005; 9(1):5–33.

[15] Miklau G. XML Data Repository, University of Washington, 2004. http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

[16] Wikipedia databases. http://download.wikipedia.org/backup-index.html [November 2006].

[17] Wratislavia XML Corpus. http://www.ii.uni.wroc.pl/~inikep/research/Wratislavia/index.htm.

[18] Skibiński P. XWRT program version 2.1. http://www.ii.uni.wroc.pl/~inikep/research/XML/XML-WRT21.zip.

[19] Skibiński P, Grabowski Sz, Swacha J. Fast transform for effective XML compression. CADSM conf., accepted, Lviv, Ukraine, 2007.

[20] Schindler M. A Fast Block-Sorting Algorithm for Lossless Data Compression. In Storer JA, Cohn M, editors, *Proceedings of the 1997 IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, California, 1997; 469.