# Create and Evaluate an Efficient Implemtnataion of Bzet Bitstring Compression

Robert Uzgalis
Tigertail Associates

29 September 2011

# What's a Bitstring

A bitstring is a sequential series of bits,
that is 1s and 0s or True and False marks.

Position in a bitstring is important. That is
1010 is different from 0011 even though they both
have two True bits and two False bits.

A good example of a bitstring is a computer register.

# Bitstrings are Information

Bits in a bitstring represent information. Sometimes numeric, positional, or just symbolic.

- Numerical ... each bit is a power of 2 and the sum represents the integer.
- Positional ... each bit corresponds to a position in something -- like a record in a data base.
- Symbolic ... each bit means something. Like the first bit means: is a person. The second bit means: is female...

# Big Bitstrings

Bitstrings tend to get larger with time.

- In numeric representations, from overflow
- In computer design registers increase in size for increased performance and to make easy use of parallelism.
- In database retrieval because data bases tend to get larger with time.

  Bitstring is a bit of a mouthful, in the future just bitsets, which is shorter and easier to say.

# Sparse Bitsets

A generalization: As bitsets get larger they tend to get sparse.

- For example most algorithms use relatively small numbers, far less than the size of machine register.
- In databases rare properties are by definition sparse, thus bitstrings which locate these records are also sparse.

# Taking advantage of Sparseness

- The usual way of taking advantage of sparseness in large bitsets is to compress them with run-length encoding.
- Run length encoding is a linear compression technique that replaces a run of zeros or ones with a length and an attribute

  So for example: `(128)0 (128)0 (1)1 (127)0 (128)0` could be a run length encoding of a 512 bit bitset with a 1 bit near the middle. With a little trickery this could be packed into 5 bytes or 40 bits.

# A New Way to Compress

I invented a positional, varying length, exponential, encoding, as a general, more useful, way of doing compression. Call this new representation a Bzet*.

Details of the method are contained in the power point slides from a presentation I gave here at UCLA on May 5th and in a Python program that implements the method. I won't go into the details here.

* patent pending.

# Bzets

Bzets come in multiple flavors. They are a tree based representation and the degree of branching in the tree determines the flavor. A Binary Bzet has two way branching; An Octal Bzet has eight way branching and so on. Multi-flavor Bzets are also possible, where different levels of the tree have different degrees of branching.

- For software implementation on current computer architectures Octal Bzets seem optimal.
- For hardware implementation a Binary Bzet seems more appropriate.

# Bzets

One of the important properties of Bzets is that operations on bitsets like AND, OR, and XOR are done on the compressed bitsets. This means that one NEVER unpacks and repacks the Bzet to do operations. And thus the time to do operations is compressed as well as the space saved. The Bzet software includes many operations. And others could be added.

# The Software Implementation

Currently there is a proof-of-concept software implementation of Octal Bzets written in Python 3. This project would replace this software with a C or C++ efficient version of the software and provide an API for Python 2/3, C, and maybe some other high level languages to allow free non-profit use of the Bzet idea.

# Software

If the BZET software gets done quickly or if there is enough man-power in the group then a matching efficient run-length encoding API might also be built to allow speed and storage comparisons to be done in an application.

# So the Project consists of:

I. Understanding the Bzet compression technique.

II. Designing a Bzet API for C and Python 2/3.

III. Creation of some new algorithms for Bzet processing (e.g. right and left shifting of bitsets).

IV. Implementing the Bzet algorithms in efficient C or C++ code.

V. Finding or writing RLE code and building a compatible RLE API.

VI. Running some comparison performance tests with the RLE and Bzet compressions.

VII. Documenting what you have done and reporting the results from the Bzet/RLE comparison tests.