IBM z Systems

zlib Update

Jim Johnston, TPF Development Lab

March 23, 2015

TPFUG – Dallas, TX

IBM

Presentation Summary

zlib General Information

- What is zlib?
  - General purpose software compression library which provides both in-memory and to-file system compression and de-compression.
  - Ubiquitous (used by JDK/SDK, MySQL, Curl, Apache, etc.)
- Home Page: http://zlib.net/
- Authors: Jean-Loup Gailly & Mark Adler
- RFCs Implemented by zlib library (compression data formats supported)
  - RFC 1951 DEFLATE Compressed Data Format
    - Underlying Compression Format
  - RFC 1950 ZLIB Compressed Data Format
    - wrapper around DEFLATE stream
    - default in-memory compression format
  - RFC 1952 GZIP file format
    - Also a wrapper around the DEFLATE stream
    - For compressing files, but can be done in memory as well

zlib General Information

- Useful Tidbits from FAQ
  - zlib is Thread Safe
    - Multiple threads can work with different data streams at the same time.
    - For compression to file system uses existing thread concurrency support.
  - 64-bit compatible
  - Maximum Amount of data that can be compressed is only limited by memory available to process.
    - Multiple calls can be used to compress or uncompress data in chunks.
    - Data must be in contiguous storage (no scatter/gather interfaces)
  - zlib does not handle *.ZIP files directly (like WinZip, 7-Zip)
    - Requires some extra code to access the underlying DEFLATE stream
  - No accessing data randomly in a compressed stream without special preparation
    - stream maintain indexes of where decompression can begin.

z/TPF Specific zlib Information

- zlib v1.2.8 - PJ42410 (PUT11)
  - opensource co-req PJ42641 (PUT 11)
- zlib v1.2.3 – PJ32283 (PUT 4)
- Version upgrade is a replacement of previous version for z/TPF
  - same library, CZCM
- No application migration required, backward/forward compatible
- Mostly fixes between v1.2.8 and v1.2.3
- Interesting new API provided between the two versions inflateGetDictionary()
- zlib is SCRT discounted

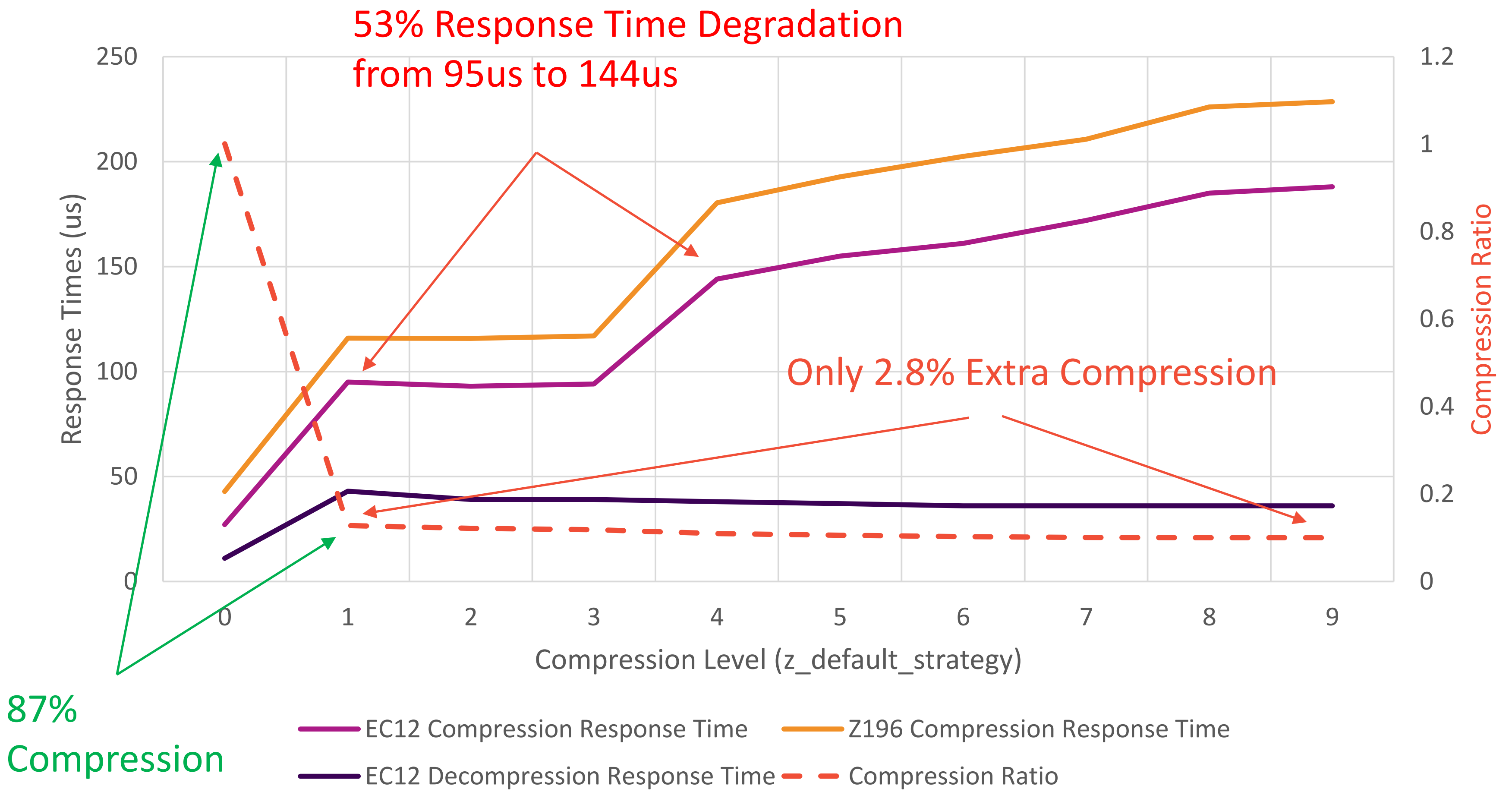
zlib 'Knobs'

- Compression Level (0-9)
  - 1 = Best Speed, 9 = Best Compression, 6 = Z_DEFAULT_COMPRESSION
- Compression Strategy
  - Z_DEFAULT_STRATEGY
    - String Matching Optimization (LZ77 algorithm) + Huffman algorithm
  - Z_HUFFMAN_ONLY
    - Force Huffman Encoding (skip LZ77 processing)
  - Z_FILTERED
    - Somewhere between Huffman and Default Strategy
      - Limits LZ77 encoding by having min criteria for matching length
  - Z_RLE
    - PNG Data (graphics compression format)
  - Z_FIXED
    - Disables a specific component of Huffman Encoding

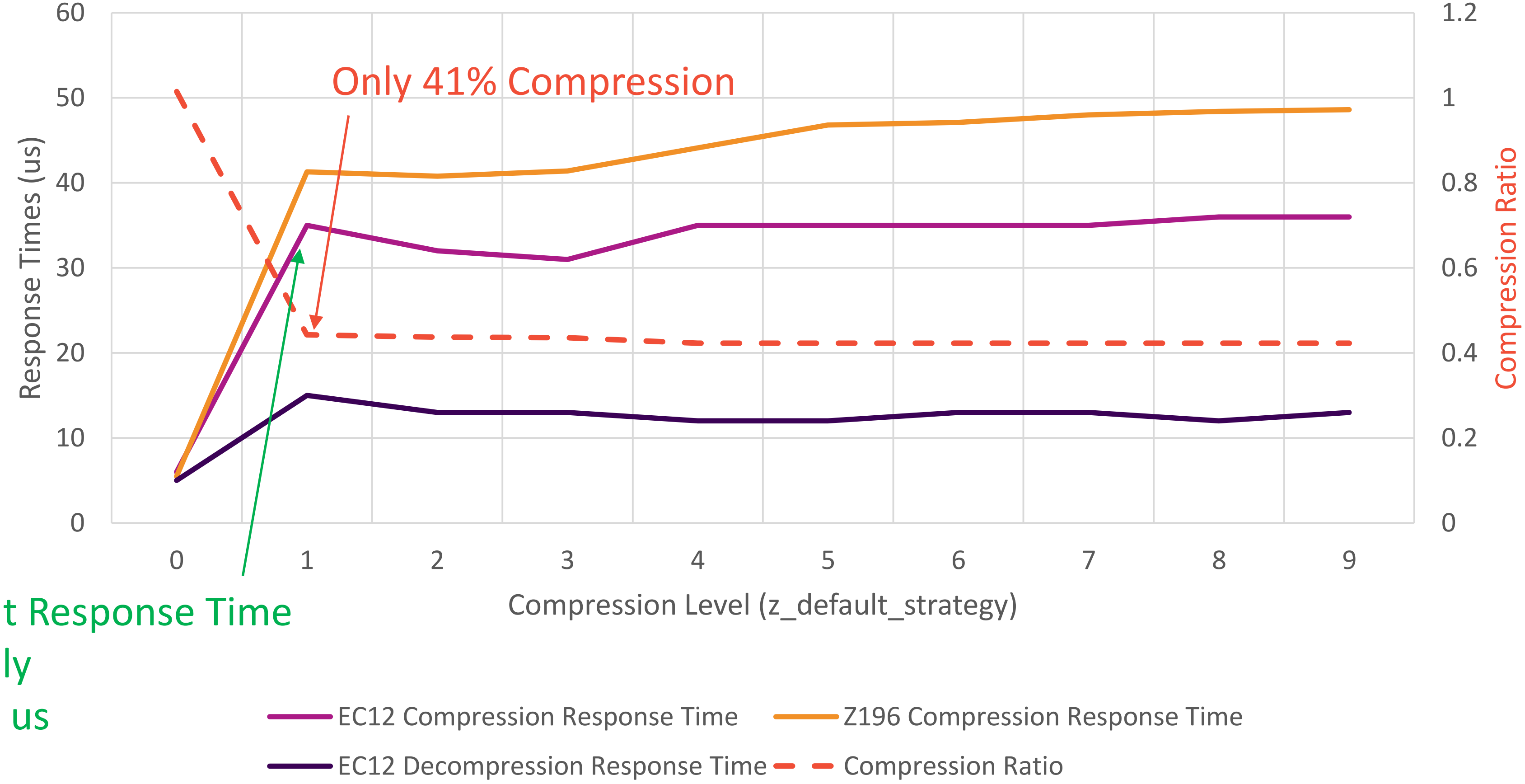
zlib ‘Knobs’

- WindowsBit (8-15)
  - History window size (aka dictionary buffer)
  - Default setting is 15, base 2 (e.g., size would be 2^15)
- memLevel (1-9)
  - Internal memory use setting. 1 = minimum memory 9 = uses maximum memory
  - Default setting is 8
- Default WindowsBits and memLevel requires a minimum of 256k Bytes per compression stream

12kB XML document compression performance

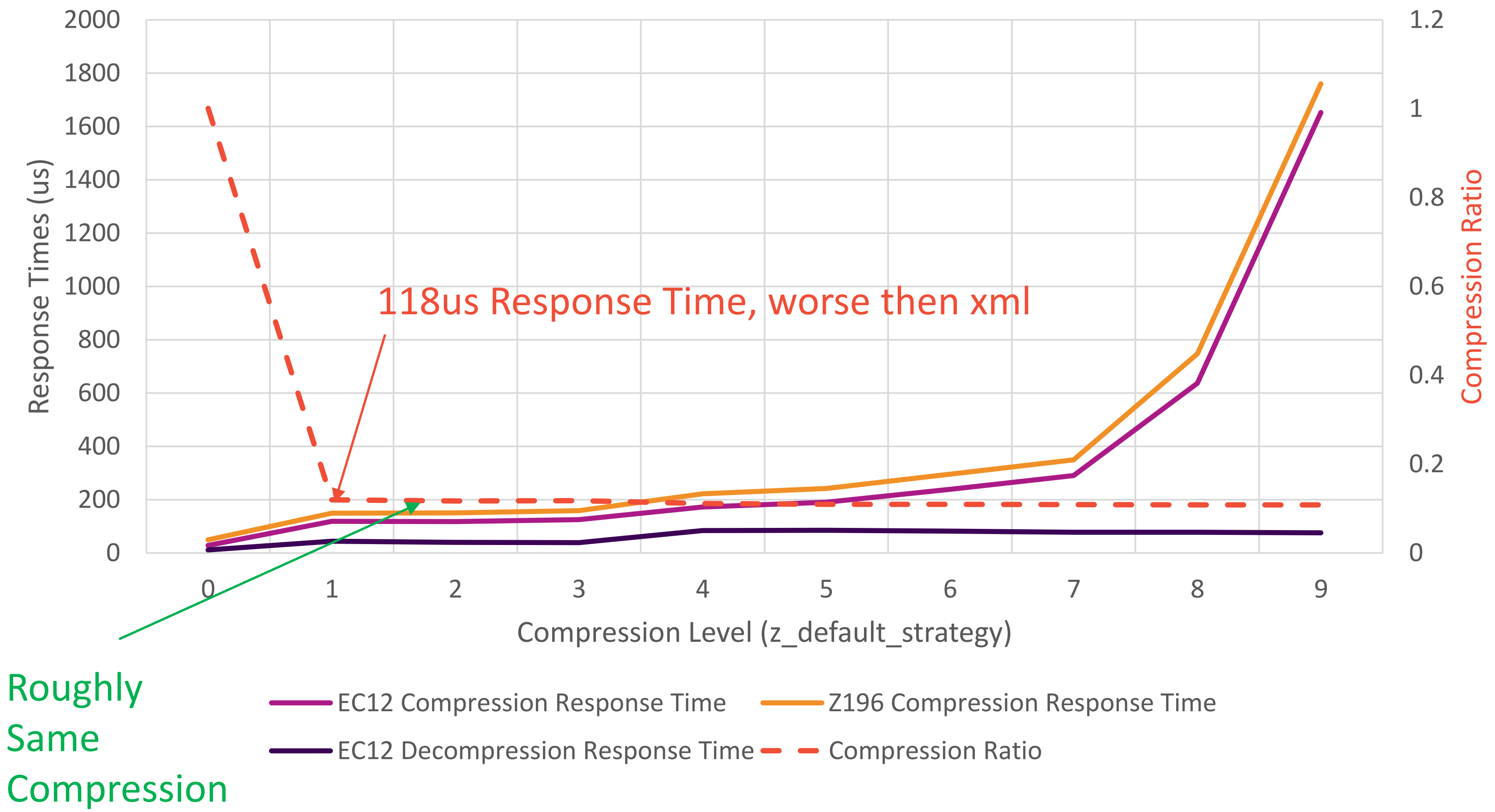


759 byte XML document compression performance

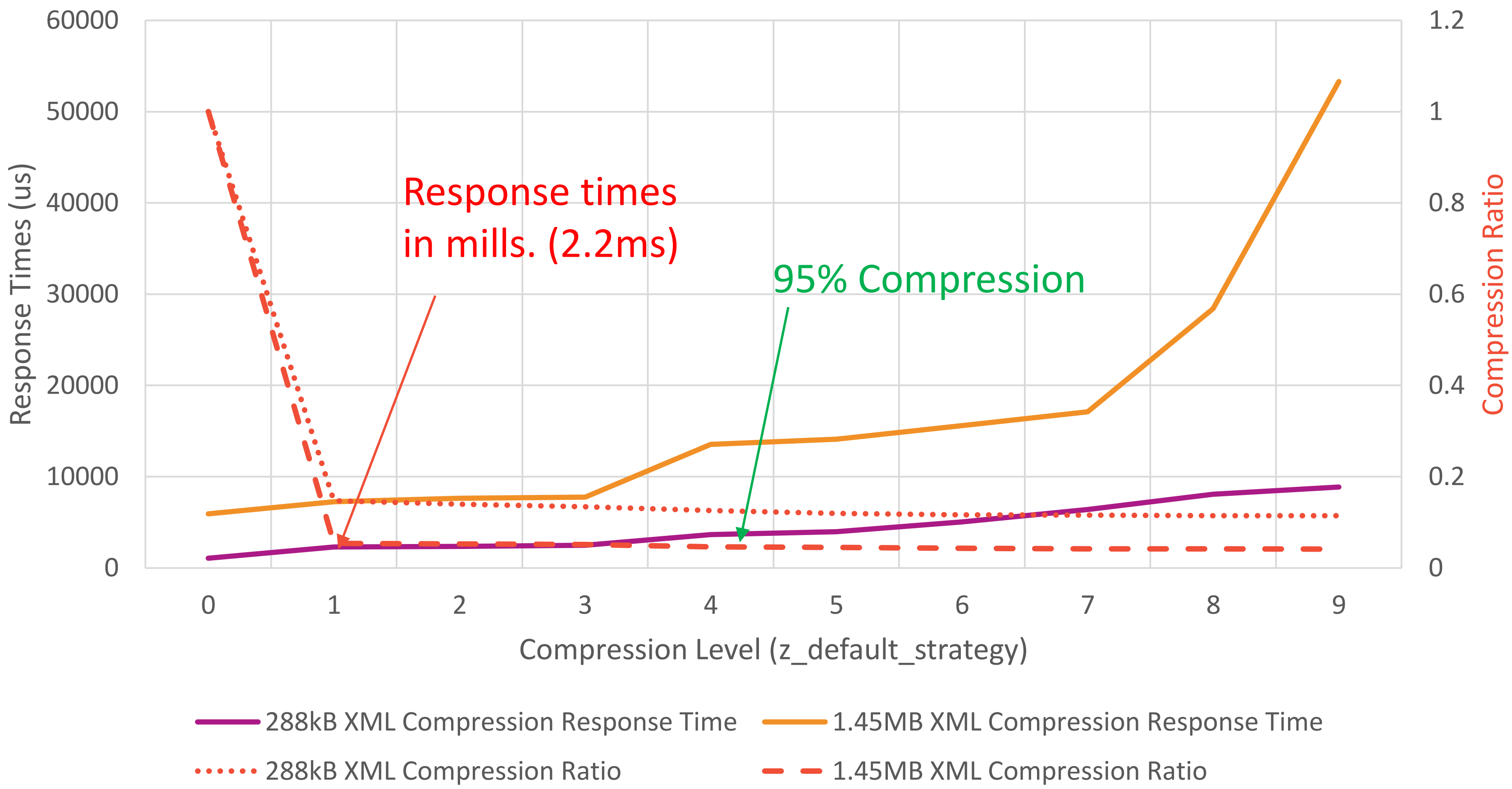


But Response Time only 35 us

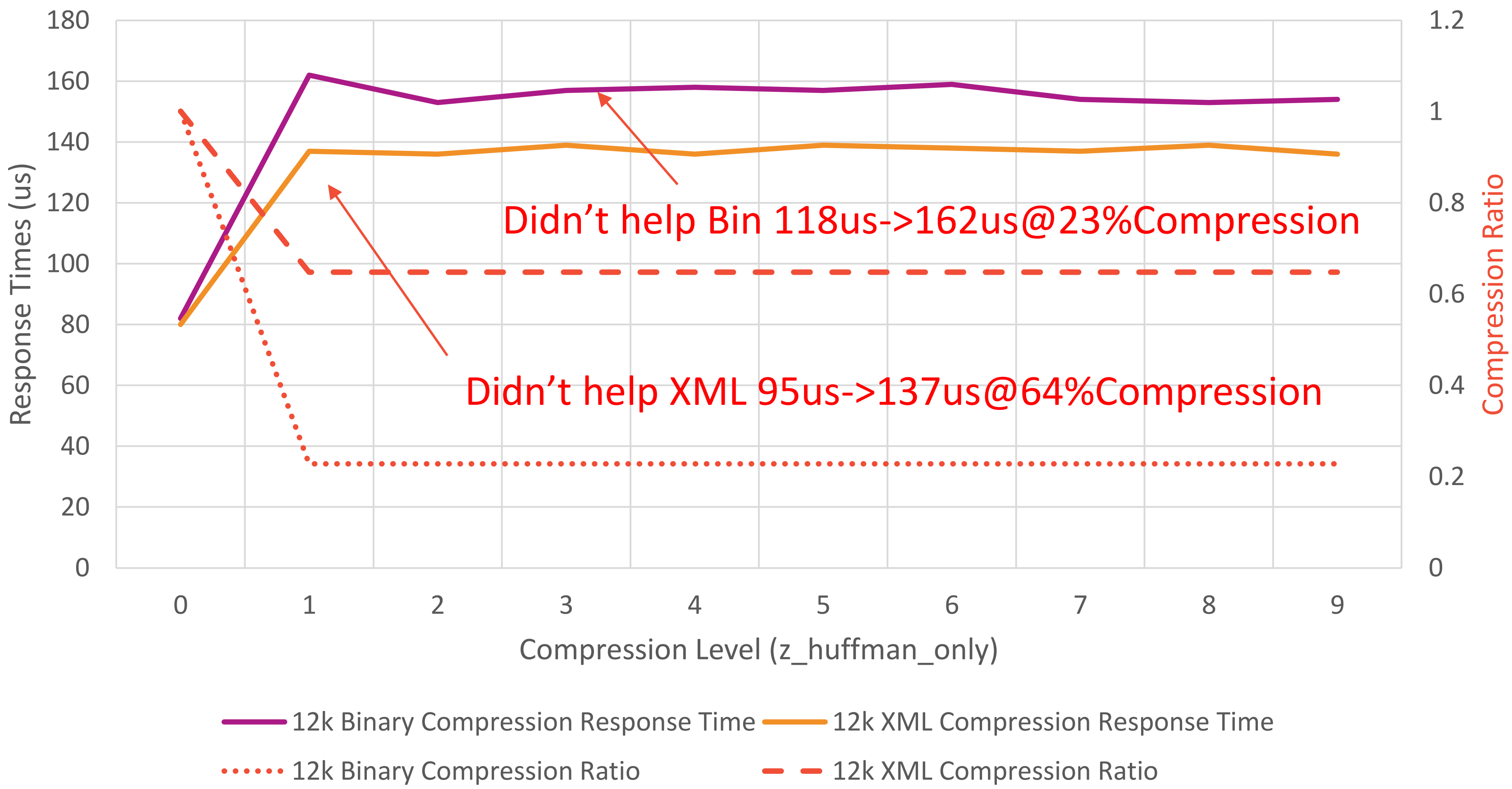
12kB binary file compression performance



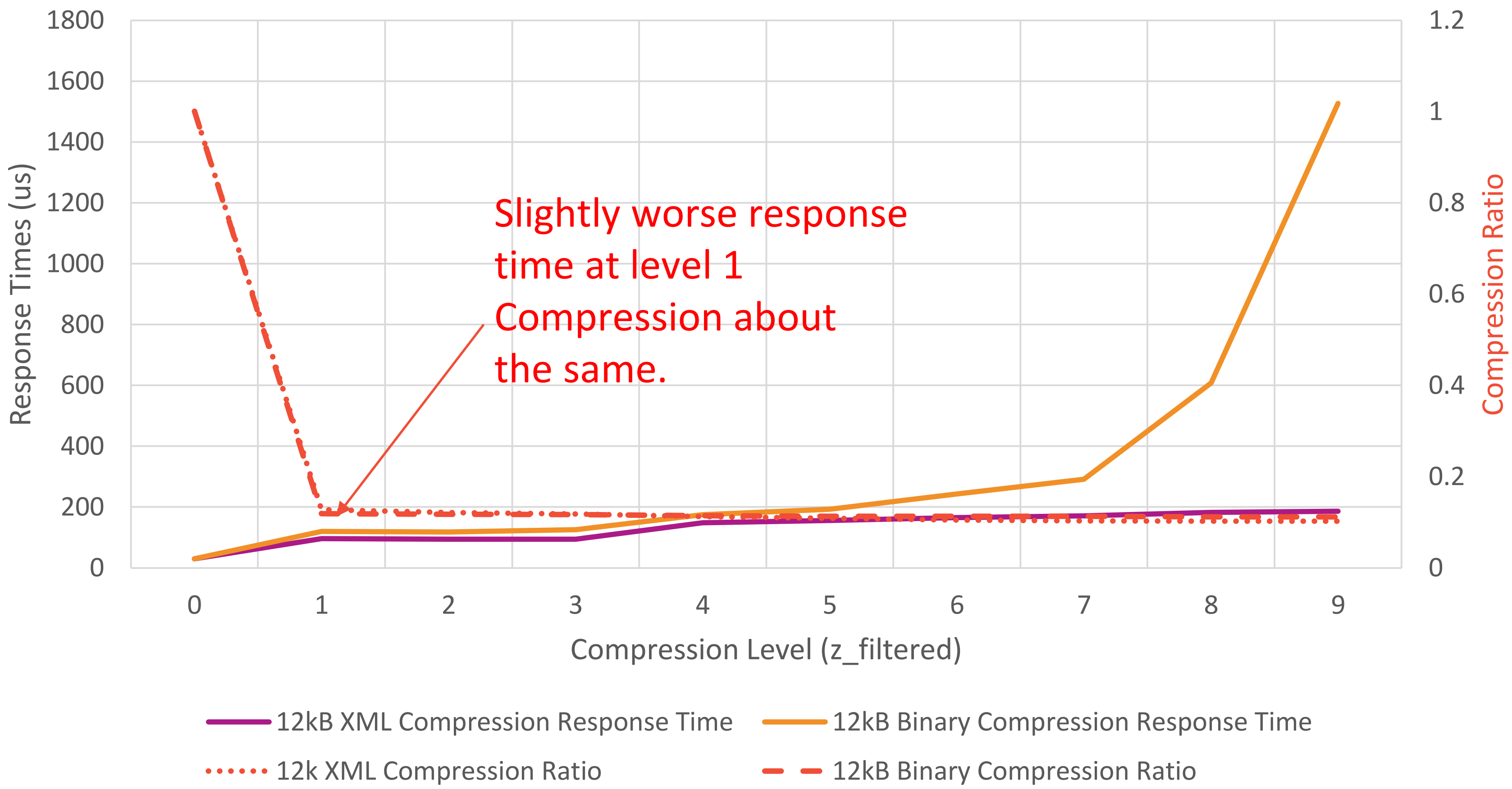
Larger File Compression (EC12)



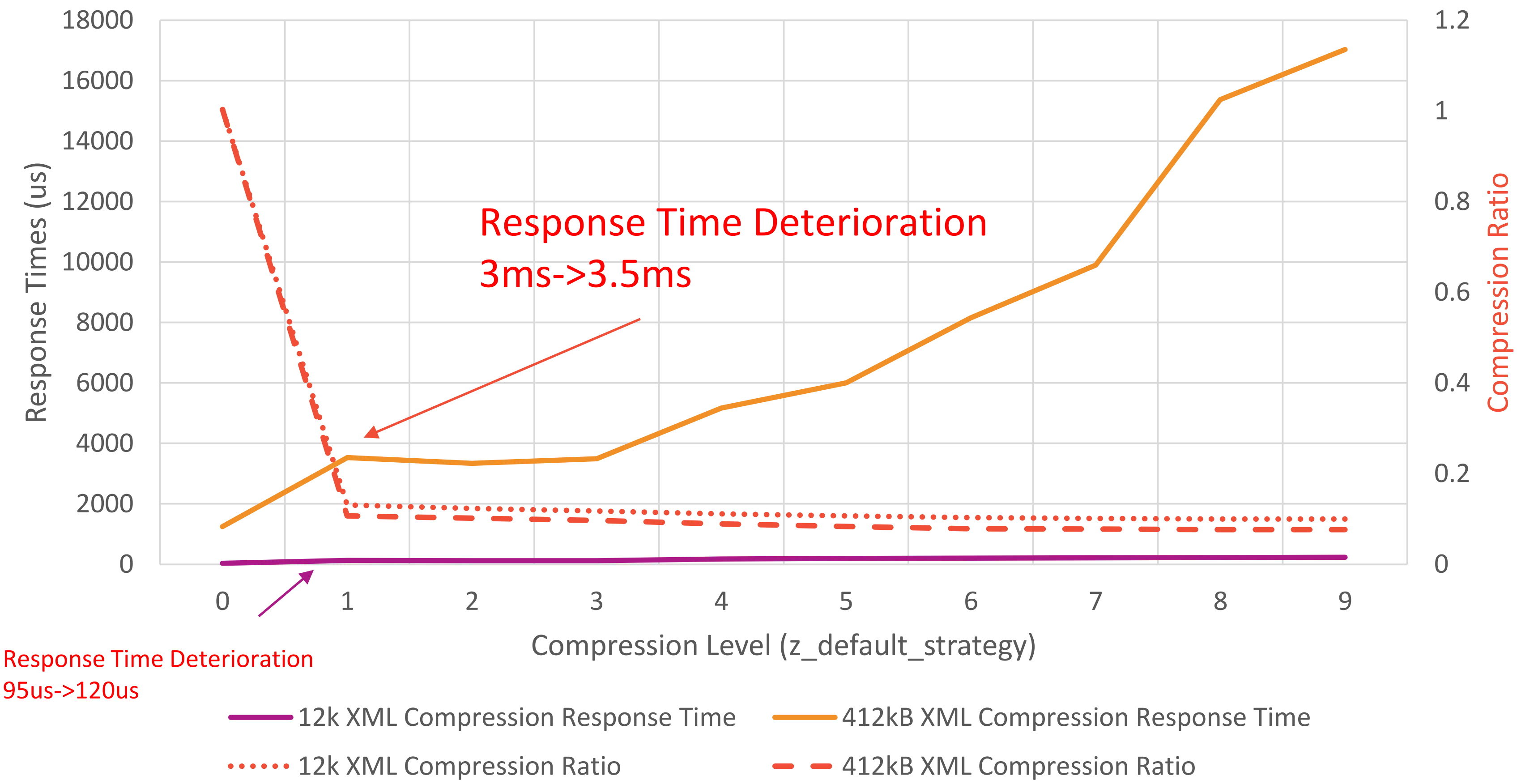
12kB Compression Using Huffman Only (EC12)



Compression Using Filter Strategy (EC12)

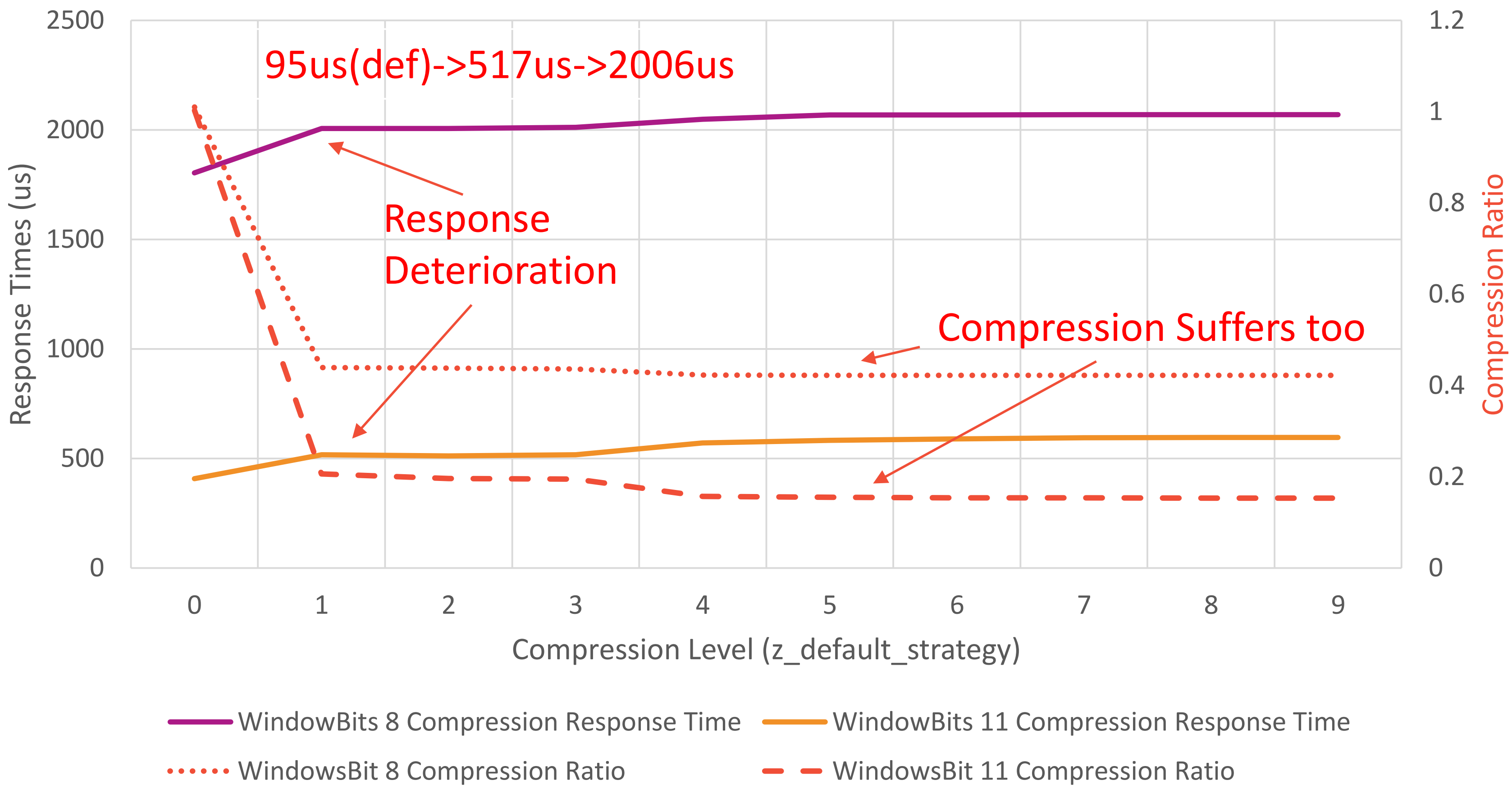


Compression using MemLevel 3 (EC12)



Response Time Deterioration
95us->120us

12kB XML Compression adjusting WindowBits (EC12)



Deflate Interfaces

Simplest way is to use compress and compress2 APIs (looks like memcpy).

```
int compress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);
int compress2 (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen, int level);
int uncompress (Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);
uLong compressBound (uLong sourceLen);
```

Things to note when using these APIs:

- Not stream based, the destination buffer must contain enough space or the call fails.
  - Use *compressBound()* to determine destination buffer size.
  - Save and pass *sourceLen* size to *uncompress()* for its destination buffer size.
- Recommend using *compress2()* with compression levels 1-3.
- Actual compressed/uncompressed bytes is returned and replaces original *destLen.*
- *Can't specify a dictionary.*

<u>Example:</u>

```
size_t compressedSize = compressBound(incomingBuffer);
Bytef out[compressedSize];
int level = 3;
ret = compress2(out, &compressedSize, (const Bytef *)incomingBuffer, (size_t) incomingSizeParm, level);
```

Deflate Interfaces (continued)

Stream-based zlib APIs to compress and uncompress data, more complex.

```
int deflateInit (z_streamp strm, int level);
int deflateInit2(z_streamp strm, int level, int method, int windowBits, int memLevel, int strategy);
int deflate (z_streamp strm, int flush);
int deflateEnd(z_streamp strm);
int inflateInit (z_streamp strm);
int inflateInit2(z_streamp strm, int level, int windowBits);
int inflate(z_streamp strm, int flush);
int inflateEnd(z_streamp strm);
```

Things to note when using these APIs:

- Stream based, useful when many messages coming into system must be filed down in order to receive entire message.
- Complexity arises from filling input buffers & emptying output buffers at different rates.

Example:

```
z_stream strm;
flushflag = Z_NO_FLUSH;
deflateInit(&strm, level);
do {
    //Get more input data to compress into inputBuffer
    // if no more then set flushflag to Z_FINISH.
    strm.avail_in = numberOfInputBytes;  //# of Bytes
    strm.next_in = inputBuffer[];              //Buffer
    do  {
     strm.avail_out = numberOfOutputBytes;
     strm.next_out = outputBuffer[];
     deflate(strm, flushflag);
     //consume compressed data here, file down, etc.
    }  while (strm.next_out == 0);
 }  while (flushflag != Z_FINISH);
deflateEnd(&strm);
```

GZIP Interfaces

ZLIB APIs for compressing to file system, just like FSTREAM APIs (e.g., fopen, fread…)

```
gzFile gzopen(const char *path, const char *mode);
int gzread (gzFile file, voidp buf, unsigned len);
int gzwrite (gzFile file, voidpc buf, unsigned len);
int gzprintf(gzFile file, const char *format, ...);
z_off_t gzseek(gzFile file, z_off_t offset, int whence);
int gzrewind(gzFile file);
z_off_t gztell(gzFile file);
int gzeof(gzFile file);
int gzclose(gzFile file);
```

Example:

```
gzFile mygzFile;
char buffer[size];
mygzFile = gzopen("testfile", "rb3f");
bytesRead = gzread(mygzFile, &buffer, size);
vel);
gzclose(mygzFile);
```

Things to note when using these APIs:

- `gzopen()` uses mode similar to fopen but with level & strategy specification "wb3h" (e.g. compress level 3, using Huffman-only).
- `gzread()` and `gzwrite()` will uncompress data from a file and compress data to a file respectively.
- Multiple gzip streams can be in the same file separated by other bytes.
- `gzseek()` `offset` represents byte position of uncompressed data, when file opened for write only forward seeks are supported.

Dictionary Use

- Dictionary is optional for zlib.
- Dictionary is limited to window size specified in deflateInit
  - ~32k for default WindowsBit setting
- Dictionary simply a character buffer ordered by higher frequency character sequences at the end of the buffer.
- Dictionary set with API after deflateInit, inflateInit calls
- Inflate call returns error if Dict was used during deflate but not set before inflate.
- Dictionary use didn’t reduce the response time.
- inflateGetDictionary() retrieves the dictionary zlib built as part of inflate
  - No need to have compression dictionary specified during deflate to get dictionary.

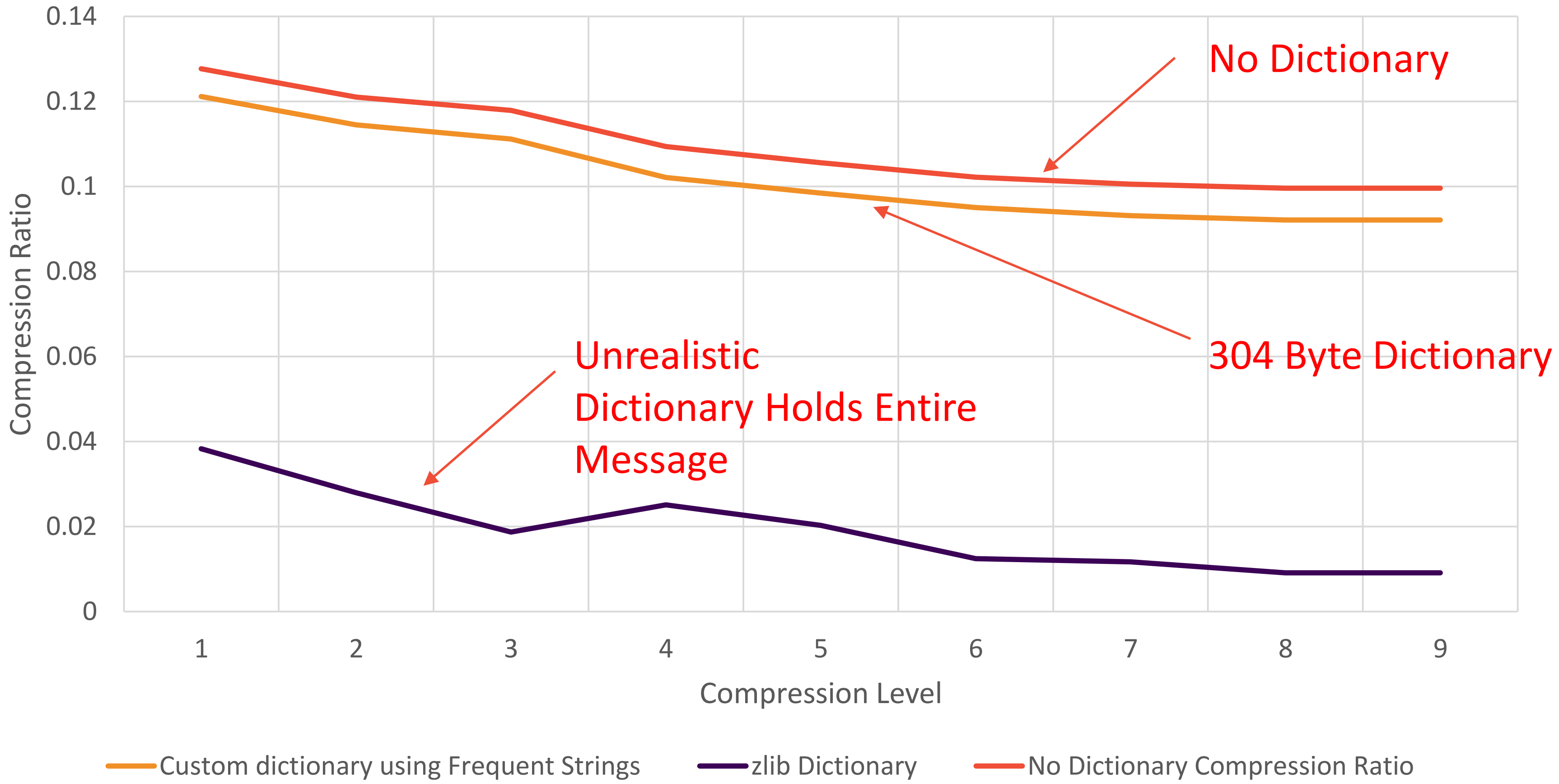
Example:

```
z_stream strm;
deflateInit(&strm, level);
ret = deflateSetDictionary(strm, (const Bytef*)
                   dictionaryBuf,  dictLen);
deflate(strm, flushflag);
```

Example:

```
 dictionaryBuf = malloc(32768);
      ret = inflateGetDictionary(&strm, (Bytef *)dictionaryBuf, &dictLen);
//Dictionary fetched with length of: dictLen
//Can write buffer out to file, for re-use later on
inflateEnd(&strm);
```

Dictionary Compression (12k XML Document)



zlib Tuning Recommendations

- Stick with Z_DEFAULT_STRATEGY, other algorithms seem to compress worse and cost more in terms of Utilization/Response Time
  - There may be very specific scenarios where the other algorithms shine…
  - Stick with compression levels 1-3
  - At level 4 the response time takes a hit as high as 50%
    - From compression level 1 to level 9 there appears to be a gain of 3% in compression
- Keep WindowsBit & MemLevel setting to Default
  - Response Time deteriorates rapidly when lowering setting on WindowsBit
  - There might be use case for lowering this if you are running multiple compressions under the same process and the messages are small.
- Use compress2 API whenever you can if you have the memory and can compress the message in one call.
  - Very easy to code, allows to set lower compression level
- Initial measurements show little benefit for dictionary use given potential problems of maintenance across a distributed environment.
  - Needs further investigation.

Questions/Comments