RJ10508 (ALM1303-015) March 15, 2013
Computer Science

IBM Research Report

Random Read Speedup in Compression Enabled Storage Systems

Cornel Constantinescu, Joseph Glider, David Chambliss
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Dilip Simha
Stony Brook University

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Random Read Speedup in Compression Enabled Storage Systems

Cornel Constantinescu, Joseph Glider, David Chambliss and Dilip Simha
{cornel, gliderj, chamb}@us.ibm.com, dnsimha@cs.sunysb.edu
IBM Almaden Research Center and Stony Brook University

Abstract

Modern primary storage systems are increasingly supporting real time compression, often based on the LZ77 and/or Huffman algorithm. There is a tension in such systems between achieving compression effectiveness along with performance and scalability; best compression effectiveness is thought to require more data to be compressed in a unit, but decompressing those large units to extract small blocks cause higher latencies and consume much more system resource. We show that, starting from the `zlib` base, both goals can be achieved, resulting in `zlib` -compatible highly compressed code streams that support low latency extraction of small blocks.

1 Introduction and Related Work

In today's compression-enabled primary storage systems, the *compression unit* of raw data compressed as a group is generally tens or hundreds of kilobytes (KB). This helps achieve good compression and high throughput for sequential or large random reads. However, the need to decompress a full compression unit to obtain a small piece of it degrades latency and I/O rate for small random reads (such as 4 KB pages). The goal in this work was to improve `zlib` (which combines LZ77 and Huffman coding) latency and CPU utilization for extracting arbitrary parts of files or storage blocks while retaining compression effectiveness and memory footprint, and while also retaining `zlib` codestream compatibility.

One option is to manage the overhead by using smaller compression units. That choice, however, entails impact on other system attributes, for example the amount of metadata needed to manage the compression units. A second approach is to devise an efficient method of extracting small sections of a compression unit. Kreft's and Navaro's work on LZ-end [4] provides for extraction of sections, but their specific encoding cannot be built from the results of `zlib` parsing, and requires significantly more resources (memory and processing). We must also concern ourselves with the layer of Huffman coding used in `zlib`. Jacobson [2] developed a method for decoding a Huffman coded stream from a random location. However, LZ77 decoding must pull in delocalized data from the code stream anyway, so localized Huffman decoding is of limited benefit for our objective.

In the rest of the paper, we will describe a series of explorations we conducted, by which we arrived at a surprisingly simple `zlib`-based approach to compression and decompression that retains compression effectiveness while dramatically improving decompression latency and CPU utilization. In Section 2 we discuss the implementation and performance of an efficient *recursive extraction* algorithm. In Section 3 we evaluate the performance and compression consequences effects of *partitioned extraction* which is a means to use smaller, flexible compression units. Finally in Section 4 we offer some observations based on these analyses.

2 Recursive Extraction

For our initial explorations, we bypassed the Huffman coding in `zlib` and operated on the `zlib`/LZ77 encoder output, using `zlib` [1] code base version 1.2.5. Note that this technique can be also applied to other LZ77 based algorithms (like LZ4 [5] or Snappy [6] that don't use Huffman coding as a second stage).

2.1 Recursive extraction algorithm

An LZ77 code-stream has all the information needed to start extracting data from any desired raw byte position *a* to byte position *b*. The LZ77 parsing [8] factors a string into a sequence of phrases. Each phrase in the `zlib`/LZ77 parsing corresponds to a token in the code stream, either a literal or a (back_pointer, length) tuple representing a match. By preprocessing the code stream we quickly reconstruct the sequence of phrases

and build auxiliary succinct data structures, along the lines described in [3, 7], for fast random access into the phrases. (One can also reduce the preprocessing work in the decoder by recording additional metadata in the code stream.) The succinct data structures support two functions: One maps from the byte domain to the LZ77 phrase domain: phraseID = phraseOf(byte_address). Its inverse maps from phrases to bytes: phrase_start_address = startOf(phraseID).

The *recursive extraction* algorithm to decode from an arbitrary starting point a (Figure 1) identifies the phrase containing that a. If that phrase is a literal it is extracted, and the starting point advances. If the phrase is a match then its back_pointer leads to another starting point in another phrase. The back_pointers are followed until literals are reached. While this is compactly shown as a recursion, the implementation we evaluated actually used iteration instead of recursive calls, and memoized extracted phrases so that back-tracking to literals is not repeated each time.

```
Input: Phrases[], random access metadata, a, b, len = b–a+1
Output: bytes in [a, b]
1. extractSegment(start_addr, len)
2.    if (len == 0) return;
3.    Pa = phraseOf(start_addr);
4.    if (Phrases[Pa] == literal)
5.       output literal; start_addr++; len– –; return;
6.    offset = start_addr – startOf(Pa);
7.    back_pointer = Phrases[Pa].distance;
8.    Pa_length = Phrases[Pa].length;
9.    leftOver_Pa = Pa_length – offset;
10.   pas = startOf(Pa) – back_pointer; /*start addr. of source of Pa */
11.   match_addr = pas + offset;
12.   if(leftOver_Pa ≥ len)
13.      extractSegment(match_addr, len);
14.   else
15.      extractSegment(match_addr, leftOver_Pa);
16.      extractSegment(start_addr + leftOver_Pa, len – leftOver_Pa);
```

Figure 1: The Recursive Extraction Algorithm

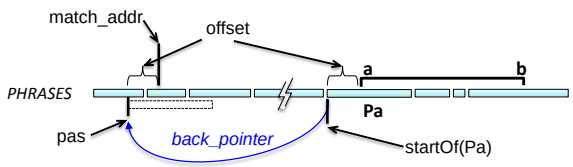

Figure 2: Operation of the algorithm. The dashed box shows the overlap of phrase Pa with prior phrases with the same content.

2.2 Performance evaluation

With a feasible technique in hand to decompress arbitrary sections without an end-to-end decompression, we need to determine its performance. Each iteration of the algorithm is simple and quick, but extraction of a segment $[a,b]$ can be slow if long back_pointer chains must be followed to reach literals. We compare the time to extract $[a,b]$ recursively with standard, sequential decoding from the start to point b, denoted $[0,b]$. While the latter generally accesses much more of the input compressed stream, it does so progressively which is well suited to processor caches and prefetching, whereas recursive extraction touches bytes across scattered cache lines with additional references to control metadata. When extracting $[a,b]$, the starting offset a and the extraction length $len = b - a + 1$ are both strongly correlated to performance. The question is for which extraction lengths and offsets, if any, the recursive approach is generally faster.

In the discussion, LZ77 refers specifically to the LZ77 implementation in `zlib`, and the times do not include the very short preprocessing time to build the random access data structures. Experiments were performed on a SUSE Linux 10 system with a single-core Intel Xeon 3.2 GHz processor having 3 GB of memory. We evaluated the performance of random extraction methods on a set of 34 files representative of the data relevant in the context of a compression-enabled primary storage system. The files ranged from very compressible to uncompressible. In each figure they are placed along the x-axis in order of increasing compression ratio (CR = compressedSize/originalSize). The raw size of each file is 250 KB = 256000 bytes, because this approximates a typical uncompressed amount of data compressed together.

We chose a spread of 6 offset values in each file: 32 KB, 64 KB, 100 KB, 140 KB 180 KB and 215 KB. Extraction with a smaller offset is generally faster, since back_pointer chains reach at most to offset 0.

We evaluated against two sets of extraction lengths. For block-oriented storage systems, the lengths we consider *large amounts* are most applicable: 4 KB, 8 KB 16 KB and 32 KB. Thus with 6 offsets and 4 extraction lengths as described, we performed 24 extraction measurements for each of the 34 files. In Figure 3 we show the 24 large-amounts extraction times for one sample file. Some applications like indexing or searching in compressed data may make smaller extractions, so we also evaluated with *small amounts*: 8, 16, 32, 64, 128, 256, and 512 bytes, and 1 KB, 2 KB, and 4 KB. With 6 offsets and 10 lengths we performed 60 extraction measurements for each of the 34 files. In the next sections when comparing extraction speeds of random extraction methods we usually aggregate the measurements in two ways: (1) aggregated per file, so we display 34 time points, or (2) aggregated per offset (across files and extraction sizes) so we have 6 points, one for each offset.

	R	S	R	S	R	S	R	S	R	S	R	S	
Extracted Length	32K	32K	64K	64K	100K	100K	140K	140K	180K	180K	215K	215K	Full Decode
4096	32	85	35	148	44	221	56	302	68	383	73	453	564
8192	36	124	35	156	43	227	56	308	67	389	73	459	564
16384	37	106	41	171	49	243	64	324	73	404	78	477	564
32768	50	138	52	203	61	323	73	356	86	439	175	509	564
sum	155	453	163	678	197	1014	249	1290	294	1615	399	1898	2256

Figure 3: The 24 measured extraction times (6 offsets x 4 extraction lengths "large amounts") for one of the 34 files. Times are in microseconds. Recursive extraction times $[a,b]$ (tinted columns) are labeled *R*. Sequential extraction $[0,b]$ is labeled *S*.

For the smaller sizes (from 8 bytes to 4 KB), recursive extraction of $[a,b]$ was generally faster than sequential $[0,b]$ decompression (Figure 4). This held true for the full range of offsets (Figure 5). The algorithm could be effectively used in applications needing to extract small fields at random from large compressed datasets. For general-purpose storage servers, however, the predominant I/Os will have sizes 4 KB or larger.

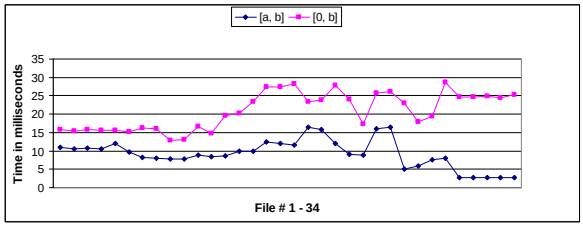

Figure 4: Aggregate extraction times for extracting small amounts $[a,b]$ (sizes from 8 bytes to 4KB) vs $[0,b]$.

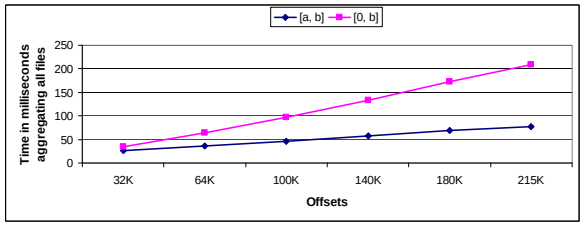

Figure 5: Extracting small amounts $[a,b]$ vs. $[0,b]$ from various offsets: $[a,b]$ gets faster relative to $[0,b]$ as offset increases.

The results averaged over larger extraction sizes, 4 KB to 32 KB, were significantly different (Figures 6 to 7). For most files in Figure 6 it is better to decompress $[0,b]$ sequentially than to extract $[a,b]$ recursively. The main exception is nearly uncompressible files (rightmost in the figure) for which the reference chains are few and short, but it is not our main goal to improve speed in cases where bypassing compression entirely might be an even better choice.

Figure 7 shows that for higher starting offsets, the aggregate time for extraction across all the data sets is higher. This highlights the overhead generated by the recursive backward references; the larger the offset, the higher the probability that more and longer recursive backward reference chains will need to be followed.

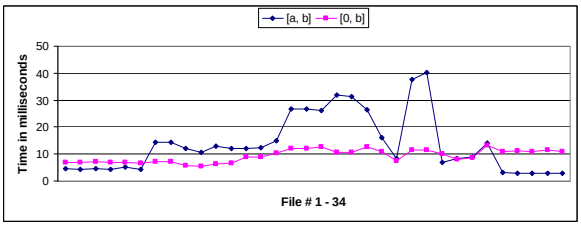

Figure 6: Extracting $[a,b]$ vs. $[0,b]$ for 4 KB and larger: $[0,b]$ is generally faster. Due to the potential for long chains of back references, the method can be slow when extracting segments larger than few hundreds bytes.

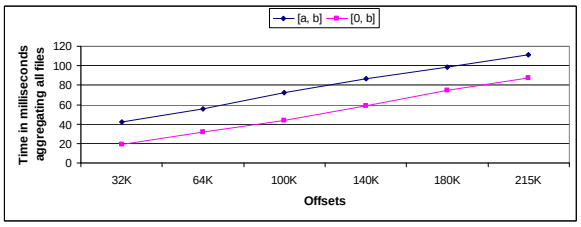

Figure 7: Extracting $[a,b]$ vs. $[0,b]$ from various offsets, for large amounts. Both methods get slower as offset increases, and $[0,b]$ is generally faster.

3 Smaller and Flexible Compression Units

The overhead of $[0,b]$ compression arises from decompressing $[0,a-1]$ and then discarding it (after possibly using pieces of it to produce the content $[a,b]$). With smaller compression units the discarded content is smaller. We evaluated the benefit of switching from 250 KB compression units to 32 KB by flushing compressor output every 32 KB, thus starting a new compression unit or *partition*, and extracting $[a,b]$ with sequential decompression starting at the nearest flush point prior to a. This *partition-aware* decoding is much faster than decoding $[0,b]$. Figure 8 shows that extraction times for this method hold relatively constant between 2 and 4 msec.

Using smaller compression units also provides very significant improvements for the performance of recursive extraction, because it reduces the lengths of back_pointer chains. Figure 9 shows that, recursive extraction from files split into 32 KB compression units took significantly less time than extracting from the beginning of the file, reversing the result from experiments without flushing. For the larger extraction sizes we focus

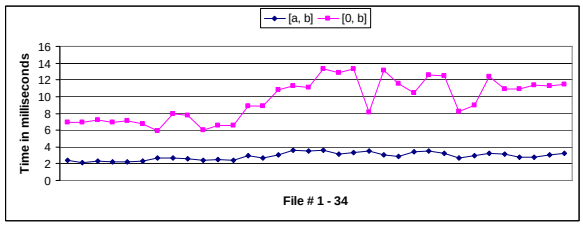

Figure 8: Reduced compression units (32 KB): sequentially uncompressing only the 32K partitions containing $[a,b]$ is much faster than $[0,b]$.

on, though, the better performance comes from decompressing sequentially from the start of a compression unit to the endpoint b, as shown in Figure 8.

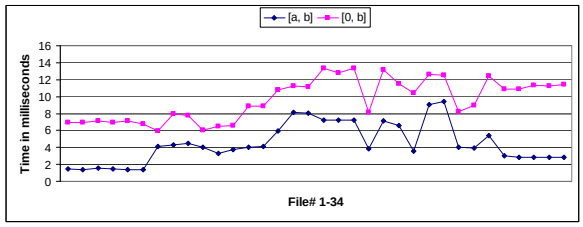

Figure 9: Recursive extraction with 32k compression units: Large-amount extraction times for $[a,b]$ are $2\times$ to $5\times$ better than $[0,b]$.

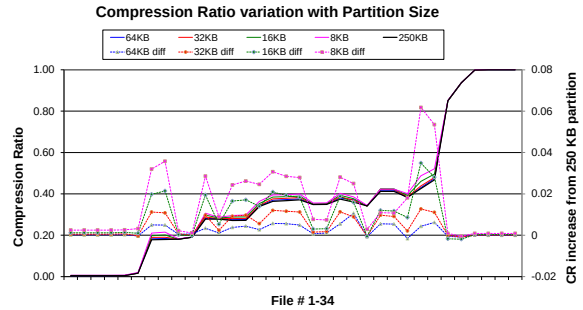

Figure 10: Compression ratio variation with partition size. For each reduced partition size, the CR difference against the full 250 KB compression unit is shown at enlarged scale.

The choice of compression unit is coupled to other aspects of system design beyond data extraction efficiency, such as preferred I/O sizes for disk drives, the block sizes for space management, and the management of placement metadata. The choices can be made independent by packing multiple compression units as partitions within a data block generally handled as a unit for placement and I/O operations. This not only allows compression units to be smaller, but also permits variable size without alignment constraints, so odd lengths of compressed output need not be padded. But smaller compression units limit compression effectiveness by narrowing the range over which repeats may be detected and removed. Is that price small enough to allow for the performance improvements of smaller compression units?

For our data set (and for other larger data sets that we looked at as well), we found the compression ratio degradation caused by reducing compression units from 250 KB to 32 KB to be surprisingly small. Figure 10 shows that over our data set, for partition sizes between 250 KB and 8 KB, where the `FASTEST` mode of `zlib` was employed, the absolute differences in compression ratio varied from zero to about 6%, about 2% on average, and the maximum relative difference was 25%. We chose the `FASTEST` mode as being typical for realtime applications, but found that even with the default zlib mode, the difference was on average about 5%. This was a surprising but very encouraging result. We note that the reduced LZ77 effectiveness might have been compensated for by an increased Huffman effectiveness, since Huffman tables for smaller partitions may better fit each partition's content.

An important additional benefit of partitioning (using smaller compression units) is the proportional reduction in the volume of data that must be Huffman decoded. Thus for block storage systems our preferred solution to the random-extraction performance problem is PEX: partition-aware extraction implemented with the combined LZ77 and Huffman coding as packaged in `zlib`.

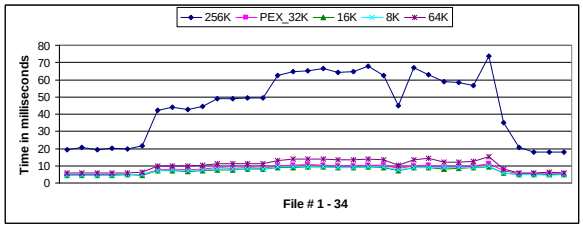

Figure 11: Aggregated extraction times for PEX: partition-aware extraction using `zlib`.

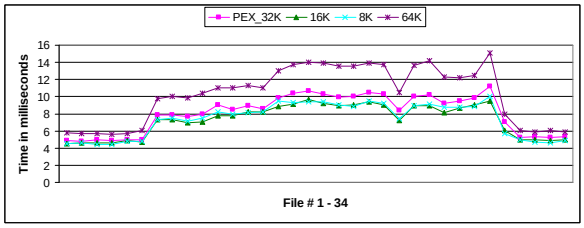

Figure 12: PEX relative speeds for partitions 8K to 32K

Figure 11 shows the PEX performance benefit, by comparing the time to extract the larger-size segments (4 KB to 32 KB) using PEX partition sizes from 8 KB to 64 KB, compared with decompressing the full 250 KB file (the starting point for this investigation). Partitioning is

highly effective for a wide range of partition sizes. Figure 12 focuses on variation with partition size; smaller partition sizes generally improve performance but the performance increase diminishes as the partition size decreases.

Finally, given that Huffman decoding is actually a large percentage of the decoding time, we wondered whether we could do without it. Figure 13 shows that removing Huffman coding from `zlib` results in dramatically better performance relative to partitioned extraction. However, Figure 14 shows that compression ratio degradation after removing Huffman is quite large. We conclude that while Huffman coding is expensive for performance, it contributes too much to the compression effectiveness to take away.

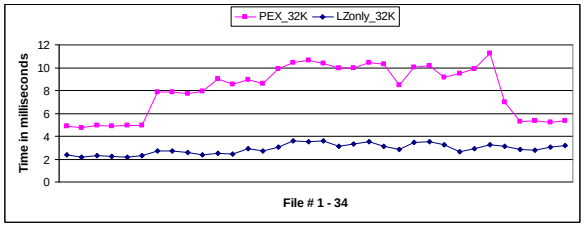

Figure 13: PEX decoding is more than 2× slower than the LZ77 decoding without Huffman coding.

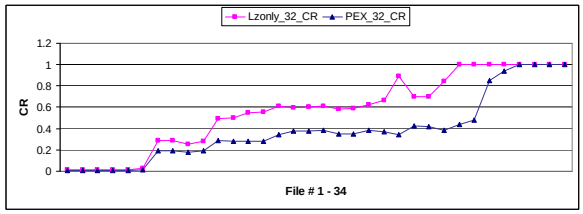

Figure 14: PEX has much better compression than LZ77 without Huffman coding.

4 Conclusions and Future Work

Prior to this work, it was fairly clear that a form of recursive decoding should be feasible, but not whether it would be competitive in performance. It is intuitively appealing to decode only the desired range and its antecedent phrases, thereby touching less of the total data payload. However, the cost per byte touched is much higher in the recursion than in the mature and streamlined `zlib` code. We found that recursive extraction is very effective for small extraction sizes. However, it is not very beneficial in the size range of most importance to block-oriented storage devices.

The surprising result from our investigations was that so much performance improvement was achievable from a simpler change to current techniques, with very little impact on compression effectiveness. We found partitioned extraction to be the approach of choice for larger extraction sizes. Another advantage of using smaller, flexible size partitions as units of compression, is that they can be re-packaged into other compressed blocks (at clean-up, for example) without going through uncompression and recompression. These are practical results which can be leveraged to provide dramatic improvements in realtime processing of compressed data in primary and archival storage systems along with many other applications.

Future research building on these investigations may include hardware implementations of these algorithms along with explorations into encoding schemes that enhance random extraction performance. In addition, there are significantly faster compression techniques emerging that will benefit from random extraction techniques.

References

[1] DEUTSCH, P., AND GAILLY, J.-L. Zlib compressed data format specification version 3.3, 1996.

[2] JACOBSON, G. Random access in huffman-coded files. In *Proceedings of the 1992 Data Compression Conference* (Washington, DC, USA, 1992), IEEE Computer Society, pp. 368–377.

[3] KIM, D. K., NA, J. C., KIM, J. E., AND PARK, K. Efficient implementation of rank and select functions for succinct representation. In *Proceedings of the 4th international conference on Experimental and Efficient Algorithms* (Berlin, Heidelberg, 2005), WEA'05, Springer-Verlag, pp. 315–327.

[4] KREFT, S., AND NAVARO, G. Lz77-like compression with fast random access. In *Proceedings of the 2010 Data Compression Conference* (Washington, DC, USA, 2010), IEEE Computer Society, pp. 239–248.

[5] LZ4. Lz4 compression software. http://code.google.com/p/lz4/, 2011.

[6] SNAPPY. Snappy compression software. Wikipedia: http://en.wikipedia.org/wiki/Snappy_(software), 2011.

[7] VIGNA, S. Broadword implementation of rank/select queries. In *Proceedings of the 7th international conference on Experimental algorithms* (Berlin, Heidelberg, 2008), WEA'08, Springer-Verlag, pp. 154–168.

[8] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory 23*, 3 (1977), 337–343.