Non-Linear Compression: Gzip Me Not!

Michael F. Nowlan, Bryan Ford, and Ramakrishna Gummadi
Yale University

Abstract

Most compression algorithms used in storage systems today are based on an increasingly outmoded sequential processing model. Systems wishing to decompress blocks out-of-order or in parallel must reset the compressor's state before each block, reducing adaptiveness and limiting compression ratios. To remedy this situation, we present Non-Linear Compression, a novel compression model enabling systems to impose an arbitrary partial order on inter-block dependencies. Mutually unordered blocks may be compressed and decompressed out-of-order or in parallel, and a compressor can adaptively compress each block based on all causally prior blocks. This graph structure captures the system's data dependencies explicitly and completely, enabling the compressor to adapt using long-lived state without the constraint of sequential processing. Preliminary experiences with a simple Huffman compressor suggest that non-linear compression fits a diverse set of storage applications.

1 Introduction

Data compression [17] algorithms have improved over the years [14, 26], and are now an integral part of storage systems. Examples include data compression and deduplication for virtual machines, snapshots, backups and archivals [16,22] and remote access, synchronization and version control [12,18,25,27]. Other application domains such as audio/video storage and playback [2] and network protocols [3, 8] use compression extensively.

Modern compression schemes are adaptive, and, hence, inherently stateful. Most algorithms assume their state evolution is linear: the compressor may use any information derived from bytes 1 through n to compress byte $n + 1$; the decompressor must typically also process bytes 1 through n before it can decode byte $n + 1$. An application can reset the compressor's state completely at application-defined block boundaries to make blocks independently decompressible, but the compressor is then unable to build or utilize *any* long-lived state across block boundaries. Many modern applications, however, require parallel or out-of-order compression or decompression of limited-size blocks, such as: versioned or deduplicating file systems that must support random block access [16, 22]; distributed revision control systems that must compress and merge often small deltas from independent sources [12, 18]; collaborative editing systems needing to compress and interchange many small but mutually-interacting document "transforms" [9,21]; and network protocols desiring both data/header compression [3, 8] and out-of-order delivery [11,15]. Even in storage applications for which large blocks are suitable, linear compression can limit the system's ability to exploit multicore CPUs or new parallel I/O devices [6] and media [24].

To address this significant constraint, we propose a new Non-Linear Compressor (NLC) abstraction. It structures the complete compression state as a Directed Acyclic Graph (DAG) of individual state nodes. An NLC state node supports three operations: forking a child node, compressing a block of data, and merging with another node. Forking copies compression *state*, while merging joins two compression states into one. This fork/merge model enables the application to express a dependency graph structured as an arbitrary partial order, thereby allowing the compression algorithm to build and adapt using long-term state across blocks, while allowing sibling nodes anywhere in the DAG to be processed independently (Section 3.3).

We developed a proof-of-concept prototype implementing this abstraction, which supports compression using Adaptive Huffman coding. Our early experiences indicate that NLC should be attractive to current and emerging storage systems and architectures.

2 Motivation

We motivate NLC by exploring popular storage systems currently using compression. The nature and requirements of these systems suggest they could benefit from a non-linear model of compression.

2.1 Deduplicating File Systems

Deduplicating file systems identify redundant chunks of data, replacing identical instances with a single copy [22], and often delta-encoding similar, but not identical, blocks [16]. This delta-encoding produces small compressed blocks (e.g., <1KB) that are logically paired to much larger blocks of data. Naturally, a set of similar files forms a cluster of compressed blocks and a file system may contain any number of such logical "partitions". Lastly, these systems must support random access to compressed blocks. The properties of small blocks and random access make these systems ill-suited for traditional linear compressors.

A deduplicating file system using a linear compressor is forced to tradeoff access time with compression ratio. The system can either use a single compressor, compressing and decompressing all data for each request, or reset compression state for each block. The first option increases processing overhead while the second option hurts compression ratios, especially for small block sizes (Section 3.1).

Our insight here is that often blocks are related to *some* other blocks, but not usually all. By leveraging a more selective (i.e., shorter) dependency graph, NLC aims to enable random access for small block sizes without sacrificing the compression ratio within related blocks.

2.2 Other Applications

There are many applications that process data with distinct logical boundaries that could benefit from (or require) random access.

Distributed SCM. Distributed Source Control Management systems [12, 18] are conceptually similar to deduplication systems. SCM users often proactively fork and merge development branches, and groups of users may collaborate from geographically diverse areas. These systems also use delta-encoding compression to improve ratios for a file's versioned history. The potential for small deltas between versions and the requirement for independent processing by edge users suggest that an adaptive compressor supporting random access could be very useful.

Collaborative Editing. Operational Transform Collaboration systems such as Google Wave [1] and others [9, 21] use a DAG to represent disjoint state transitions between a server and a client in a shared statespace. Local actions are transmitted to the remote destination, where the receiver "transforms", or modifies, the action to account for its own local actions that have occurred in the interim, and then applies the received action locally. For high-traffic collaborations, compression can significantly lower the overall bandwidth use. Specifically, incremental updates are often small and may come from many disjoint users, suggesting predictable long-lived state with random access.

Network Protocols. Google's SPDY [3] protocol compresses HTTP headers and multiplexes multiple streams onto a single TCP connection. Although TCP does not deliver data out-of-order (i.e., random access), other protocols such as uTCP [15] and UDP [23] can deliver datagrams out-of-order. Unfortunately, pairing SPDY with one of these unordered protocols would not offer lower latency due to the in-order requirement of SPDY's linear compressor (i.e., gzip).

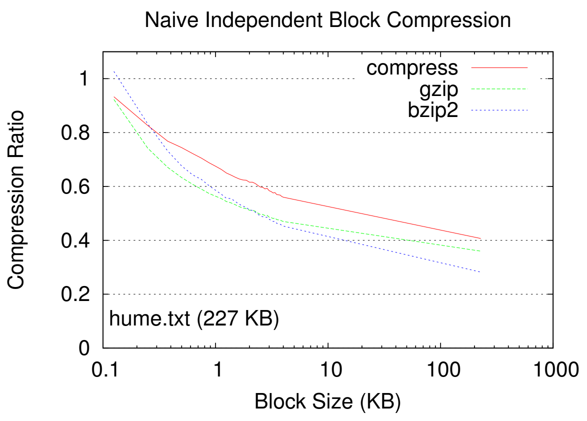

Figure 1: Resetting compression state to enable random access of data blocks limits the compression ratio for small block sizes.

3 Towards Non-Linear Compression

We present Non-Linear Compression as a system abstraction and API, independent of any particular compression scheme or implementation. We first motivate NLC by exploring the limitations of linear schemes in modern systems.

3.1 Linear Compression Limitations

Storage and transmission systems compressing data and/or metadata, typically face a tradeoff between compressor effectiveness and the granularity at which data is independently decompressible. There are two main compression modes: stream mode and block mode, each of which has significant limitations, as explained below.

In stream-oriented use, an application treats the compressor as a sequential filter, feeding in an arbitrarily large file or other data object incrementally and storing compressed output as the compressor produces it. This mode enables the compressor to build up state gradually over the entire input and eliminate redundancy at large scales, but requires decompression to proceed in the same fashion and limits potential processing parallelism at the compressor or decompressor.

In block-oriented use, in contrast, an application processes data one block or "application data unit" [7] at a time, resetting the compressor to a fresh initial state before each block. Each block can then be decompressed independently of other blocks, providing random-access or out-of-order decompression and benefiting more fully from parallelism on modern multicore hardware [10].

Block-oriented use limits the effectiveness of advanced compression algorithms depending on block size, however. Figure 1 illustrates this limitation by comparing the effectiveness of popular compressors when run with a variety of block sizes on a large text file. As the block size decreases, the compression ratio, defined as output size to input size, becomes higher. The loss of compression history at each block reduces compression effectiveness significantly because the compressor's history identifies redundancy only within the given block.

Linear compressors excel at compressing huge amounts of data, but for systems wishing to exploit random or unordered access without compromising the compression ratio, a new abstraction is needed.

3.2 Non-Linear Compression Overview

The key idea underlying non-linear compression (NLC) is to eliminate the current "all-or-nothing" choice between deriving a given data block's compression state from *all* prior data in a linear sequence, or *no* prior information. Instead, the application specifies to an NLC compressor explicitly, via an arbitrary directed acyclic graph (DAG), which previously-compressed blocks a given compressed block depends on. The NLC decompressor likewise assumes that to decompress a given block, the application will have already decompressed the "prior" blocks it specified as dependencies. As a result, decompression of incomparable blocks in the DAG are independent and fully parallelizable.

The DAG thus imposes a partial ordering relation on data blocks compressed by the DAG nodes. Two nodes in the graph are either ordered according to an ancestor-descendant relationship, or not. Unordered nodes and associated data can be processed completely independently, during both compression and decompression.

This DAG representation provides three main benefits. First, a non-linear compressor offers a single abstraction to applications with multiple logical streams, avoiding the burden of maintaining multiple compressors. Second, independent branch paths can be processed concurrently, parallelizing I/O operations reading and writing compressed data, or processing packets from unordered network protocols [15]. Thus, NLC represents a natural progression in compression in line with the industry's overall "serial-to-parallel" shift. Third, adaptive techniques can localize compression state within DAG branches, allowing logically distinct branches to have different probability distributions.

Although nominally similar, graph or tree-based compression schemes [4, 5, 13], are fundamentally different from NLC. Whereas these approaches compress *data* structured as a tree (e.g., web graphs) using linear compression, NLC structures the actual *compression state* as a graph.

3.3 A Non-Linear Compression API

Applications will often compress data as an "application data unit" (ADU), which is a logically contiguous chunk of data. NLC gives applications fine-grained control of how to structure dependencies between consecutive ADUs, whether linear or not. The NLC API provides four main operations. The first operation creates a node, while the other operate on a node.

Initialize. Initialization creates a single state node with fresh internal compression state and no dependencies (i.e., no parent nodes). The state node represents an independent compression point at the top of DAG. The application can create multiple such nodes.

Compress. A state node compresses variable-length ADUs. NLC guarantees that a given ADU will be decompressible "as a unit" once all causually prior ADUs in the DAG have been processed. NLC compressors may—but are not required to—support stream-oriented, incremental or partial decompression *within* ADUs. Naturally, two independent state nodes in the DAG can compress or decompress in parallel. Repeatedly compressing ADUs with only a single state node forms a linear chain of dependent compressed ADUs, much like regular sequential compressors. A state node can compress zero or more ADUs. For adaptive schemes, compressing an ADU modifies a node's internal state.

Fork. Forking enables a state node to create a child node with identical internal compression state. The parent copies its internal state at the *time of forking*, which may or may not be the same as the state the parent initially possessed. This enables nodes to compress an ADU and then pass the resulting state on to a child.

A child maintains a dependency on its parent and all of its parent's ancestors, but not on any siblings (i.e., other children of the same parent). A parent that forks two children in succession creates a divergent path in the DAG similar to logically distinct streams. Forking a child node *locks* the parent state node, preventing future calls to "Compress", but the locked node can still fork children.

Merge. Merging combines the internal state of two nodes and returns a new node with this combined state. The new node has a dependency on both parent nodes, as well as the union of their ancestors. Merging creates a new node, and *locks* each parent state node (as in Fork). Merging effectively aggregates the compression state, such as frequency tables or dictionaries, accumulated along all paths from the root to these nodes.

4 An NLC Prototype

The previous section outlines our general NLC framework, but many specific compression schemes could potentially implement this API. We now present one simple proof-of-concept implementation based on Huffman coding, and explore several alternative heuristics to improve NLC's unique *Merge* behavior. The techniques below are simplistic and merely intended as starting points for designing future, more mature compression algorithms in the NLC framework.

4.1 Adaptive Huffman Compression

Our current early prototype builds on simple adaptive Huffman coding. The compressor's inter-ADU state consists of a 256-entry frequency table, describing the number of times a given input byte ("symbol") has appeared in causally "prior" ADUs in the DAG. For each ADU, the compressor first builds a Huffman coding tree based on the initial frequency table summarizing all prior ADUs. The compressor then encodes each symbol in the current block using the Huffman tree. As the compressor processes each symbol, it updates its internal frequency count to reflect this input, while leaving the Huffman code unmodified throughout the block. The compressor thus adapts at block boundaries, enabling better compression of future ADUs—although not the current ADU—using shorter codes for more common symbols.

To enable the encoding of all input bytes/symbols a priori using Huffman coding, the frequency count for each input symbol must be greater than zero. Blocks with no predecessors in the DAG are "compressed" with trivial frequency tables in which all symbols have a count of 1, yielding no compression in such "initial" blocks. Thus, *all* compression in our current prototype derives from inter-block adaptivity. The current prototype also makes no provisions for compressing repeated byte sequences or context-sensitive frequency modeling; hence we have no expectation that this scheme would compete "head-to-head" with a mature linear compressor in typical (e.g., large-file) scenarios.

The *Fork* operation in our prototype simply copies the frequency counts of data objects from the parent to the new child state. *Merge*, in contrast, can be done in many ways and involves tradeoffs discussed next.

4.2 Merging Behavior

Merging combines the frequency counts of two nodes. Early indications are that the Merge operation improves adaption of compression state and offers a convenient synchronization mechanism for applications. We continue to investigate the usefulness of various merge heuristics such as: complete history traversal to accrue frequency counts; simple adding, or compounding, of frequency counts; and taking the maximum count between two children. Frequency counts directly influence the construction of the Huffman coding tree, thus, more accurate frequency counts produce better compression. In practice, the forking and merging behavior of the application determines whether a complete heuristic is needed, or whether a simpler approach (i.e., addition) is accurate "enough". We hope that more experience using the Merge operation will better elucidate its utility.

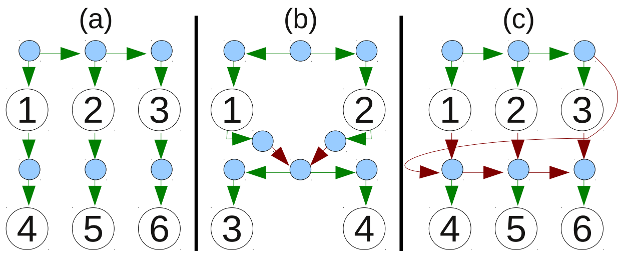

Figure 2: NLC state growth for various applications: (a) Simple parallel compression, (b) Version control with individual branches for files or users, (c) Parallel windowing for out-of-order network protocols.

4.3 Preliminary Results

Using our first-cut NLC implementation, we ran several experiments to see how NLC behaves with different application behaviors. We make no claim that our design is optimal, or even that our Adaptive Huffman implementation is competitive with real-world compressors such as gzip that combine Adaptive Huffman coding with dictionary methods. Rather, we explore different behaviors and present these results as suggestive of the outcomes when using NLC.

Figure 2 shows three different application behaviors using green and red arrows to represent Fork and Merge operations, respectively. Blue circles and numbered circles represent compression state and compressed data blocks, respectively. Part (a) exploits NLC for simple parallel compression. This behavior initializes a window, w, of base state nodes (or a single base state and its $w - 1$ children). Treating this window of nodes as individual compressors, each node compresses a logical stream of ADUs independently. This behavior does not use the API's merge functionality.

Behavior (b) models a potential version control system, where the base state forks children to be used in compressing different files. Periodically, the children nodes merge back with the base state, collecting compression state. The base state can then fork new children with adapted compression state for each file.

Lastly, behavior (c) models how a network application might use a sliding window of state nodes to compress ADUs for use with an out-of-order protocol. The receiver of a compressed ADU can always decompress it, provided that the compressed ADU is no more than w ADUs beyond the last successful decompression.

We wrote a simple application to test (a) and (c) above by compressing a contiguous data stream into independent ADUs of 128 bytes. Figure 3 shows the average compression rate for different stream sizes. As expected, the overall compression rates are worse than those in Figure 1 because our NLC prototype currently uses only Huffman coding (without a deduplication algorithm).

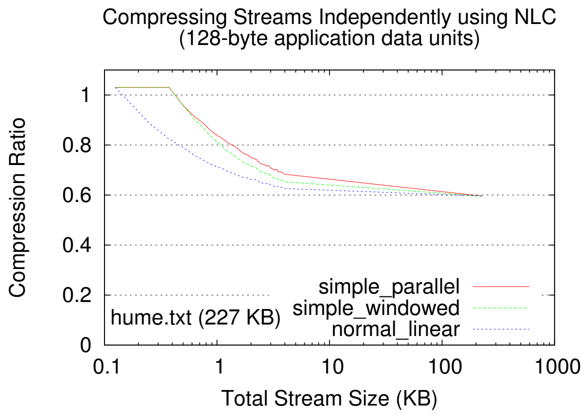

Figure 3: NLC compresses small ADUs independently based on the forking and merging strategy.

The "simple_parallel" and "simple_windowed" data series use a window of $w = 3$. Note that streams less than or equal to $3x$ ADU experience no compression, since the first w ADUs are compressed independently before any adaptation takes place. The figure also shows "normal_linear", which uses a single state node to repeatedly compress ADUs. Each of these ADUs is *dependent* on the previous ADU.

4.4 Next Steps

Experiences with our early prototype suggest the importance of several design decisions that we plan to explore further. For compression, our prototype uses Adaptive Huffman coding, which operates by assigning shorter code words to more frequently occurring data objects (e.g., bytes). Adaptive Arithmetic coding [26] also uses frequency counts and should be an easy addition. LZW-style [19] deduplication uses word dictionaries, but is conceptually similar for Fork and Merge operations.

Other considerations include "code spaces", or multi-byte data objects, as our current prototype only operates on a byte granularity. Furthermore, probabilistic models [20] for code spaces introduce new possibilities for automatically detecting application-specific data objects. We also plan to investigate a "decay" model, phasing-out frequency counts and/or code spaces in order to react more quickly to a changing source. Related to decay is garbage collecting obsolete state nodes; our prototype performs no special garbage collection. Lastly, our prototype assumes the application names or identifies compression state nodes itself, but could, in the future, support some naming scheme.

5 Conclusion

Storage systems today often use linear compression algorithms that cannot fully exploit modern parallel processors and protocols. To alleviate this tension, Non-Linear Compression allows applications to structure data dependencies in an arbitrary, hierarchical graph.

Acknowledgments

We thank Jana Iyengar, Avi Silberschatz, Rob Ross and the reviewers for their valuable feedback. This research was sponsored by NSF grants CNS-0916413 and CNS-0916678 and the ASEE-NDSEG program.

References

[1] Google wave operational transform. `http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform`.

[2] The moving picture experts group. `http://mpeg.chiariglione.org/`.

[3] SPDY: An Experimental Protocol For a Faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[4] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference, 2001*, pages 203–212, 2001.

[5] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 679–688. 2003.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192. 2009.

[7] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[8] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[10] J. G. Elytra. Parallel data compression with bzip2. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.9891`.

[11] B. Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, Aug. 2007.

[12] git: the fast version control system. `http://git-scm.com/`.

[13] X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM J. Comput.*, 30(3):838–846, 2000.

[14] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept. 1952.

[15] J. Iyengar, S. O. Amin, M. F. Nowlan, N. Tiwari, and B. Ford. Minion: Unordered delivery wire-compatible with TCP and TLS. Apr. 2012. To appear. `arXiv:1103.0463`.

[16] P. Kulkarni et al. Redundancy elimination within large collections of files. In *USENIX*, June 2004.

[17] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19:261–296, September 1987.

[18] mercurial: distributed source control management. `http://mercurial.selenic.com/`.

[19] M. Nelson. LZW data compression. *Dr. Dobb's Journal*, Oct. 1989.

[20] M. Nelson. Arithmetic coding + statistical modeling = data compression. Feb. 1991. `http://marknelson.us/1991/02/01/arithmetic-coding-statistical-modeling-data-compression/`.

[21] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120, 1995.

[22] C. Policroniades et al. Alternatives for detecting redundancy in storage systems data. In *USENIX*, June 2004.

[23] J. Postel. User datagram protocol, Aug. 1980. RFC 768.

[24] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. ISCA '09, pages 24–33. 2009.

[25] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Apr. 2000.

[26] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30:520–540, June 1987.

[27] T. Ylonen and C. Lonvick, Ed. The secure shell (SSH) authentication protocol, Jan. 2006. RFC 4252.