Bicriteria data compression: efficient and usable*

Andrea Farruggia, Paolo Ferragina, and Rossano Venturini

Dipartimento di Informatica, University of Pisa, Pisa, Italy
{farruggi,ferragina,rossano}@di.unipi.it

Abstract Lempel-Ziv's LZ77 algorithm is the *de facto* choice for compressing massive datasets (see e.g., Snappy in BigTable, Lz4 in Cassandra) because its algorithmic structure is flexible enough to guarantee very fast decompression speed at reasonable compressed-space occupancy. Recent theoretical results have shown how to design a bit-optimal LZ77-compressor which minimizes the compress size and how to deploy it in order to design a *bicriteria data compressor*, namely an LZ77-compressor which trades compressed-space occupancy versus its decompression time in a smoothed and principled way. Preliminary experiments were promising but raised many algorithmic and engineering questions which have to be addressed in order to turn these algorithmic results into an effective and practical tool. In this paper we address these issues by first designing a novel bit-optimal LZ77-compressor which is simple, cache-aware and asymptotically optimal. We benchmark our approach by investigating several algorithmic and implementation issues over many dataset types and sizes, and against an ample class of classic (LZ-based, PPM-based and BWT-based) as well as engineered compressors (Snappy, Lz4, and Lzma2). We conclude noticing how our novel bicriteria LZ77-compressor improves the state-of-the-art of fast (de)compressors Snappy and Lz4.

1 Introduction

The design of high-performing distributed storage systems — such as BigTable by Google [6], Cassandra by Facebook [3], Hadoop by Apache — requires the design of lossless data compressors which achieve effective compression ratio and very efficient decompression speed. The scientific literature abounds of solutions for this problem, named "compress once, decompress many times", but compressors running behind those large-scale storage systems are highly engineered solutions which only merely resemble the scientific results from which they are derived. The reason relies in the fact that theoretically efficient compressors are designed and analyzed in the RAM model, while their performance in practice is significantly conditioned by the numerous cache/IO misses induced by their decompression algorithms. This poor behavior is most prominent in the BWT-based compressors, such as Bzip2 and its derivatives [1, 5], and it is not negligible in the LZ-based approaches (dating back to [19, 20]).

* The Research was partially supported by the MIUR PRIN grant ARS-Technomedia

This motivated the software engineers to devise variants of Lempel-Ziv's original proposal (e.g., Snappy by Google, Lz4) which inject several software tricks having beneficial effects on memory-access locality at the cost of, however, increasing the compressed size. These compressors expanded further the known jungle of space/time trade-offs,[1] thus posing the software engineers in front of a choice: either achieve effective compression-ratios, possibly sacrificing the decompression speed (as it occurs in the theory-based results [8, 10, 11]); or try to trade compressed space by decompression time by adopting a plethora of programming tricks, which nonetheless waive any mathematical guarantees on their final performance (as it occurs in Snappy and Lz4).

Recently, it has been shown [7] that it is possible to design a bicriteria LZ77-compressor which allows to trade in a smoothed and principled way both the space occupancy (in bits) of the compressed file and the time cost of its decompression, by taking into account the underlying memory hierarchy. The key result was to design an algorithm that determines efficiently an LZ77-parsing of the input file $\mathcal{S}$ which minimizes the compressed-space occupancy (in bits), provided that its decompression time is bounded by a value T (in seconds) fixed in advance. Symmetrically, it is possible to exchange the role of the two computational resources. This problem has been solved by rephrasing the *bicriteria LZ77-parsing* problem into the well-known *Weight-constrained shortest path problem* (WCSPP) over a weighted DAG, where the goal is to search for a path whose decompression-time is at most T and whose compressed-space is minimized. This allowed to design an algorithm which solves the problem in $O(n \log^2 n)$ time and $O(n)$ working space, thus improving significantly all previously known results for the general version of WCSPP, which require $\Omega(n^2)$ time.

Very preliminary experiments [7] have shown the potential of the bicriteria LZ77-compressor (shortly, Bc-Zip) whose decompression speed is close to those one of Snappy and Lz4 (i.e., the fastest ones) and compression ratio is close to those of BWT-based and LZMA compressors (i.e., the most succinct ones). In this paper we address the following issues, which prevent the bicriteria strategy to be successful in practical settings:

- Bc-Zip deploys as a subroutine the bit-optimal LZ77-compressor devised in [11], which finds a LZ77-parsing that minimizes the compressed output (cfr. [12, 16]). Unfortunately the compressor implemented in [11], and used in [7], was slow and not optimal in asymptotic sense, though it was superior to the heuristics introduced in [2, 13, 17].
- the decompression efficiency of Bc-Zip relies heavily on the estimation of LZ77 decompression time. This was addressed in [11] by proposing an interpolation approach which required a "training" dataset and deployed many parameters, losing accuracy and generalization.

The main contributions of this paper are the following:

- we propose a novel bit-optimal LZ77-compressor which is simpler, cache-aware and asymptotically optimal, thus resulting faster in practice (see Section 3).

[1] See e.g., http://mattmahoney.net/dc/text.html.

This represents an important step in closing the compression time gap with the widely known compressors Gzip, Bzip2 and Lzma2 .
- we introduce a new model for estimating the decompression time. This model is based on few measurable parameters that depend only on the underlying machine and, thus, result independent of the file to be compressed/decompressed. Given this model we design a *calibration tool* which automatically derives the model, achieving an average error of $\approx 5.6\%$; this is quite satisfactory according to [14]. Due to space limitations, its technical discussion is deferred to the journal version of this paper.
- we finally evaluate the novel bit-optimal LZ77-compressor and Bc-Zip by investigating many algorithmic and implementation issues (see Section 4): integer encoders, block lengths, dataset types, ample set of classic (LZ-based, PPM-based and BWT-based) as well as engineered compressors (Snappy, Lz4, Lzma2 and Ppmd). We perform many experiments aimed at measuring the impact of those features, so leading to the design of a compressor that surpasses the decompression performance of well engineered and widely used compressor Lz4 on three out of four datasets.

The ultimate result achieved by this paper is a deep and variegate understanding of the novel bicriteria compression technology both in terms of efficacy and efficiency issues under various experimental scenarios. We will make available to the scientific community this large implementation effort by providing the datasets, the whole experimental setting and the C++ code of Bc-Zip.

2 Background

The *bit-optimal LZ77-parsing problem* asks for a LZ77 parsing of a text $\mathcal{S}[1, n-1]$ whose compressed representation requires minimum space (in bits).

A LZ77-parsing of a text $\mathcal{S}$ is a decomposition of $\mathcal{S}$ in m substrings (phrases) of the form $p = \mathcal{S}[s, s+\ell-1]$ such that either $p = \mathcal{S}[s]$ is a single *character* (hence $\ell = 1$), or it is $\ell > 1$ and thus $\mathcal{S}[s, s+\ell-1] = \mathcal{S}[s-d, s-d+\ell-1]$ is a text substring of length ℓ *copied* from d positions before in $\mathcal{S}$. Clearly, many candidate copies might occur in $\mathcal{S}$, each having a different length and distance, so the possible LZ77-parsings of $\mathcal{S}$ may be numerous. Each of these LZ77-parsings induces a compressed version of $\mathcal{S}$ which is obtained by, first, substituting each phrase p with the pair $\langle 0, \mathcal{S}[s]\rangle$, if p is a single character, and with $\langle d, \ell\rangle$, otherwise; and then encoding each of those pairs with a pair of variable-length binary codewords which are computed by means of two (possibly different) integer encoders enc_d and enc_ℓ. For the sake of clarity, we drop the subscripts whenever the argument, either distance or length, allows us to disambiguate the encoder in use.

An important assumption of the bit-optimal approach, as of [9, 11], is that the integer encoders satisfy the so-called *non-decreasing cost property*, which is satisfied by most encoders adopted in modern compressors. An integer encoder enc satisfies the non-decreasing cost property if $|\mathsf{enc}(n)| \leq |\mathsf{enc}(n')|$ for all $n \leq n'$. Moreover, these encoders must be *stateless*, that is, they must always encode the same integer with the same bit-sequence.

More formally, given a text $\mathcal{S}$ and a pair of encoders enc_ℓ and enc_d, the bit-optimal LZ77-parsing problem asks thus for a LZ77 parsing of $\mathcal{S}$ which minimizes the compressed size when using enc_ℓ and enc_d as integer encoders. Authors of [9,11] modeled the *bit-optimal LZ77-parsing problem* as a *single-source shortest path* problem over a graph $\mathcal{G}$, consisting of $n = |\mathcal{S}| + 1$ nodes (one per $\mathcal{S}$'s character, plus a sink node) labeled with the integers $\{1, \ldots, n\}$. In particular, there is (i) a node i associated to each character $\mathcal{S}[i]$; (ii) an edge $(i, i+1)$ for every $i < n$, and (iii) an edge $(i, j+1)$ iff the substring $\mathcal{S}[i, j]$ occurs earlier in the text. It follows that each edge (i, j) is in bijective correspondence with a candidate phrase of the LZ77-parsing of $\mathcal{S}$.

This graph has a number of properties: (i) it is directed and acyclic, and (ii) there is a bijection between LZ77-parsings of $\mathcal{S}$ and paths from 1 to n in $\mathcal{G}$. Since each edge is associated to a phrase, it can be weighted with the length, in bits, of its codeword. In particular, edges $(i, i+1)$ are assumed to have constant weight, since they correspond to the single-character phrase $\langle 0, \mathcal{S}[i] \rangle$; while edges $(i, j+1)$ are weighted with the value $|\mathsf{enc}(d)| + |\mathsf{enc}(\ell)|$ provided that $\langle d, \ell \rangle$ is the associated codeword. Given that $\mathcal{G}$ is a DAG, computing a shortest path from 1 to n is simple and takes $O(m)$ time and space. But there are strings for which $m = \Theta(n^2)$, so this algorithm is not practical even for files of a few tens of MiBs.

Starting from these premises, this problem was attacked in [9, 11] by introducing two main ideas: (i) prune $\mathcal{G}$ to a significantly smaller subgraph which preserves the shortest path of $\mathcal{G}$ from nodes 1 to n; (ii) generate on-the-fly this subgraph, thus minimizing the working space of the shortest-path computation.

The *pruning strategy* consists of retaining, for each node, only the *maximal edges*, that is, edges of maximum length among those with equal cost (in bits, according to enc). It has been shown [11] that the number of maximal edges depends on the structure of enc_d and enc_ℓ, but it is $O(n \log n)$ for the vast majority of encoders. The key algorithmic issue was then to show how to generate the maximal edges outgoing from a given node i, incrementally along with the shortest-path computation, taking $O(1)$ amortized time per edge and only $O(n)$ auxiliary space. This task is called *Forward Star Generation* (shortly, FSG).

The algorithm originally described in [11] involves the construction of suffix arrays and compact tries of several substrings of $\mathcal{S}$ (possibly transformed in proper ways) so that, although optimal asymptotically, this algorithm is not practical. In the next section we show a new algorithm which is optimal asymptotically and much simpler than the algorithm proposed in [11], since it is based solely on lists and their sequential scans.

3 Bit-optimal Compression: Faster and Practical

The Forward Star Generation task asks to compute all maximal edges spurring from a node i only when needed, and discarded afterwards. An edge is *maximal* if it is either d-maximal or ℓ-maximal (or both). An edge spurring from vertex i and represented by a LZ77 phrase $\langle d, \ell \rangle$ is *d-maximal* if it is the longest LZ77 phrase taking at most $|\mathsf{enc}(d)|$ bits for representing its distance component; ℓ-maximality

is defined similarly. Finding ℓ-maximal edges is easy once d-maximal edges are known, since the strategy consists on "splitting" d-maximal phrases according to the cost classes of enc_ℓ, so here we concentrate on finding those d-maximal edges.

Let us now consider a *cost class* of enc_d, that is, the maximal sub-range $[l, r]$ of $[1, n]$ such that each integer between l and r takes exactly c bits by using encoder enc_d, for some c. There is one d-maximal edge for each cost class.

Let us take the d-maximal edge, say $(i, i + \ell)$, for the cost class $[l, r]$. We can infer that the substring $\mathcal{S}[i, i + \ell]$ is the longest substring starting at i and having a copy at distance within $[l, r]$ because the subsequent edge $(i, i + \ell + 1)$ denotes a longer substring whose copy-distance must therefore occur in a farther back subrange (because of d-maximality).

Maximal edges leaving from i can be found by considering, for each cost class, the suffix array of $\mathcal{S}$ restricted to positions i and $[i - r, i - l]$, which we denote as Rsa. In fact, the d-maximal edge can be found by looking for the lexicographic *predecessor* and *successor* of $\mathcal{S}[i, n]$ in Rsa and taking the one with the longest common prefix to $\mathcal{S}[i, n]$. The selected suffix thus is the copy-reference of the corresponding d-maximal edge. This strategy, however, is inefficient, because Rsa cannot be computed in less than $\Omega(r - l) = \Omega(n/\log n)$ time when the number of d-maximal edges is $O(\log n)$. We overcome efficiency problems related to building indexing data structures (like the Rsa) by computing *en ensemble* the d-maximal edges of $O(r - l)$ vertexes. To do that, we extend the simple strategy outlined above by looking simultaneously for predecessors/successors of a set of suffixes.

More precisely, let us denote $B = [i, i + r - l]$ as the range of positions for which we would like to determine the d-maximal edges, and $W = [i - r, i + r - 2l]$ the set of potential *back-references*. Notice that $|W| = 2(r - l)$, so $|B \cup W|$ is at most $3(r - l)$ (less if they overlap). Let us then denote Rsa as the suffix array restricted to positions in $B \cup W$. The main idea is to find all successors of suffixes starting in B with a left-to-right scan of Rsa, and all predecessors with a right-to-left scan. During the scan, we keep a queue $\mathcal{Q}$ of positions in B for which we did not have found yet their matching predecessor/successor. The queue is kept sorted in ascending order. So, let us assume the element of Rsa currently examined is in W but not in B. This means that it may be a successor for *some* of the positions in $\mathcal{Q}$, and so we have to determine those positions and remove them from the queue. We underline that not every position in $\mathcal{Q}$ may apply, because the distance between the current element in W and the element in the queue may be greater than r. However, since positions in $\mathcal{Q}$ are sorted by increasing position, those can be found in optimal $O(1)$ time per match by examining the queue starting from the first element, and stopping whenever the current element in the queue has distance greater than r.

Let us now consider the case when the currently examined element in Rsa is a position j in B. This implies that we have to insert j in $\mathcal{Q}$ while maintaining it sorted. This operation cannot be performed in constant time, since the element may be inserted in the middle of $\mathcal{Q}$. However, since distance between positions in B cannot be greater than r, positions in $\mathcal{Q}$ greater than j can be matched with j itself and removed from $\mathcal{Q}$, so j can be appended at the bottom of the

queue in constant time. It is clear that the time complexity is proportional to the number of examined/discarded elements in/from $\mathcal{Q}$, which is $|B|$, plus the cost of scanning Rsa, which is at most $|W| + |B|$. This implies that each maximal edge is found in amortized $O(1)$ time.

An important part is the efficient and on-the-fly generation of the Rsa of suffixes starting in $W \cup B$. Due to space limitations, we only sketch two optimal solutions, which will be illustrated in the journal version of this paper. The first one, general but less practical, is based on the *Sorted Range Reporting* data structure [4]. The other solution makes some (generally satisfied) assumptions about the integer encoders in use, and it is yet optimal but more practical because it is based only on lists scanning. The last one is at the core of our implementation of the Bc-Zip compressor tested in Section 4.

4 Experiments

In this section we show the effectiveness of this novel "optimization approach" over LZ77-based compression in a throughout and conclusive way. Due to space limitations, here we will only highlight the most important results, omitting many figures and technical details. A more thorough illustration will be available in the journal version of this paper.

In the first part of this section we will evaluate the advantage of the bit-optimal parsing against Greedy, the most popular LZ77 parsing strategy, and many high-performance compressors. We will show that the bit-optimal parsing has a clear advantage over heuristically highly engineered compressors, thus justifying the interest in the technology. We will also show that the novel Fast-FSG algorithm exposed in Section 3 helps to bring down considerably the compression time, thus making bit-optimal parsing a solid and practical technology.

In the second part we will compare bicriteria data compression against the most common approach of trading decompression time for compression ratio. In fact, many practical LZ77 implementations (e.g., Gzip, Snappy, Lz4 etc.) employ the bucketing strategy (that is, splitting the file in blocks (buckets) of equal size which are individually compressed and then concatenated to produce the compressed output) or a *moving window* to (hopefully) lower decompression time by limiting the maximum distance at which a phrase may be copied, thus forcing spatial locality. Interestingly enough, we will show that this approach is not the best one to speed up the file decompression because basically it takes into account neither integer decoding time nor the length of the copied string, which may be relevant in some cases and could amortize the cost of long but far copies. We will validate this argument by also introducing a *decompression time model* which properly infers the decompression time of a LZ77-parsing from a small set of features (such as number of copies, distances distribution, etc). This model will be used in order to efficiently determine proper edge-costs in the graph over which the WCSPP is solved.

We will finally show the vast time/space trade-off achievable with the bicriteria strategy, which improves *simultaneously* both the most succinct (like Bzip2) and the fastest (like Lz4) compressors.

Experimental settings: We implemented the compressor in C++11, and we compiled it with Intel C++ Compiler 14 with flags `-O3 -DNDEBUG -march=native`. According to the applicative scenario we have in mind, we used two machines to carry out the experiments. The first machine, used in compression, is equipped with AMD Opteron 6276 processors, with 128GiB of memory; the second machine, used in decompression, is equipped with an Intel Core i5-2500, with 8GiB of DDR3 1333MHz memory. Both machines run Ubuntu 12.04.

Experiments were executed over 1GiB-long (2^{30} bytes) datasets of different types: (i) Census: U.S. demographic informations in tabular format (type: database); (ii) Dna: collection of families of genomes (type: highly repetitive biological data); (iii) Mingw: archive containing the whole mingw software distribution (type: mix of source codes and binaries)[2]; (iv) Wikipedia: dump of English Wikipedia (type: natural language). Each dataset have been obtained by taking a random chunk of 1GiB from the complete files. The whole experimental setting (datasets and C++ code) is available at `http://acube.di.unipi.it/bc-zip/`.

We experimented various integer encoders for the LZ77 phrases: Variable Byte (VByte), 4-Nibble (Nibble), and Elias' γ (Gamma) and δ (Delta) [15, 18]. We also introduce two variants of those encoders, called VByte-Fast and T-Nibble, which perform particularly well on LZ77 phrases.

In the design of our compressor we modified the LZ77 scheme to allow the encoding of runs of literals in just one phrase. This twist has beneficial effects on both decompression speed and compression ratio on incompressible files when using the bit-optimal strategy, and introduces a very effective way of controlling the space/time trade-off when using the Bicriteria strategy.

Speed improvements over the novel bit-optimal compressor: In Figure 1 we compare the running time of the bit-optimal LZ77 algorithm when employing either the original subroutine for generating maximal edges (shortly FSG), as proposed in [11], or our novel Fast-FSG algorithm, described in Section 3. Figure 1 shows the results only for dataset Wikipedia, as the figures do not change significantly on the other datasets. We compared the compression ratios produced by two integer encoders — namely, Gamma and VByte-Fast — which are the ones that yield the lowest and highest performance gaps.

In the plots, Fast-FSG and FSG significantly diverge in running time, reflecting the different time complexities (constant vs $O(\log b)$ per edge, where b is the size of the bucket). In the Gamma case, the running time for the 1 MiB bucket-size are nearly the same for FSG and Fast-FSG, while the gap is already $\approx$ 4x for a 1GiB bucket-size. In the VByte-Fast case, gap ranges from $\approx$ 1.3x to $\approx$ 2.5x.

The improvements introduced by Fast-FSG make the bit-optimal LZ77-compressor much closer to the widely used and top performing compressors in com-

[2] Thanks to Matt Mahoney – `http://mattmahoney.net/dc/mingw.html`.

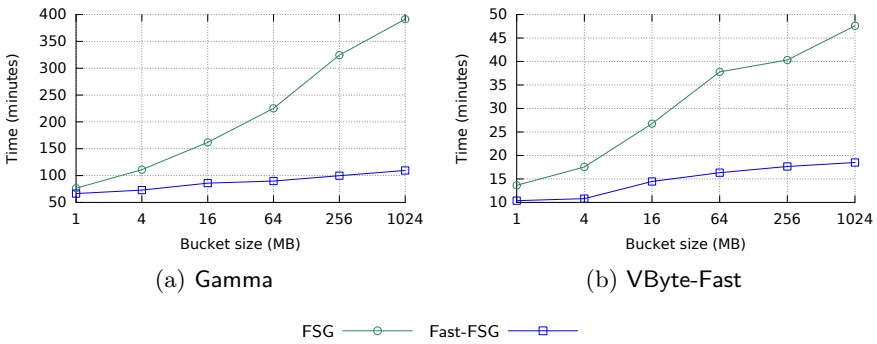

Figure 1. Comparison between the novel Fast-FSG and the previously known FSG in parsing the dataset Wikipedia by using VByte-Fast and Gamma as integer encoders. The construction time is reported by varying the bucket size.

pression time. In fact, our bit-optimal construction is on-par or faster than Lzma2.

Bit-optimal performance: According to our experiments, integer encoders T-Nibble and VByte-Fast are the most interesting in terms of compression ratio and decompression speed (respectively, the most succinct and the fastest). For this reason we restrict our attention to these two encoders in the next experiments. In our tests, the bit-optimal strategy produces parsing which are more than 10% smaller on average than greedy ($\approx 11.5\%$ with VByte-Fast, $\approx 14.6\%$ with T-Nibble), in which the rightmost longest match is always selected, while being $\approx 15\%$ faster at decompression. Working space was $\approx$ 59GiB of main memory.

Using bit-optimal in lieu of greedy means that we can use a faster encoder without sacrificing compression ratios. Bit-optimal achieves this result by "adapting" parsing choices to the ideal symbol probability distribution of the underlying integer encoders.

Table 1 compares the bit-optimal strategy (called `LzOpt`) against the best known compressors to date. With respect to the most space-efficient compressors (Lzma2, Bzip2, Ppmd, and BigBzip), compression ratio is, overall, only slightly worse: the gap with Lzma2, the most succinct compressor, ranges from 15% (Census) to 25% (Dna), with Mingw representing a situation in which the combination of bit-optimal parsing plus literal encodings let `LzOpt` be the most succinct compressor. Notice that Lzma2 reports better compression ratios than `LzOpt` due to its choice of a different, *statistical* encoder, whereas `LzOpt` is restricted to the use of *stateless* ones (see discussion on Section 2). On the other hand, `LzOpt` decompression time is order of magnitudes better than these approaches.

Comparing with the fastest compressors (Snappy and Lz4), parsings obtained with `LzOpt` are way more succinct: the relative gap of those compressors in compressed space ranges from $\approx 60\%$ (Census) to over $1,300\%$ (Dna). Decompression speed is already very competitive, especially if the slightly less succinct VByte-Fast

Table 1. Comparison between bit-optimal compressor (LzOpt), bicriteria compressor (Bc-Zip) and state-of-the-art data compressors. For each dataset we highlight the parsing having the closest decompression time to Lz4.

Dataset	Compressor	Compressed size (MBytes)	Decompression time (msecs)
Census	LzOpt (T-Nibble)	38.08	776
	LzOpt (VByte-Fast)	40.19	572
	Bc-Zip (VByte-Fast, 556 ms)	40.38	549
	Bc-Zip (VByte-Fast, 494 ms)	41.63	506
	Bc-Zip (VByte-Fast, 454 ms)	44.42	462
	Gzip	48.23	2,472
	Lzma2	33.03	2,652
	Snappy	123.68	634
	Lz4	61.82	454
	Bzip2	39.96	15,054
	BigBzip	33.28	71,000
	Ppmd	38.70	38,000
Mingw	LzOpt (T-Nibble)	179.01	1,586
	LzOpt (VByte-Fast)	192.34	954
	Bc-Zip (VByte-Fast, 920 ms)	193.77	845
	Bc-Zip (VByte-Fast, 726 ms)	205.56	695
	Bc-Zip (VByte-Fast, 461 ms)	293.62	472
	Gzip	344.47	5,534
	Lzma2	187.68	8,323
	Snappy	461.00	891
	Lz4	384.67	726
	Bzip2	317.96	32,469
	BigBzip	222.22	152,000
	Ppmd	245.54	414,000

Dataset	Compressor	Compressed size (MBytes)	Decompression time (msecs)
Dna	LzOpt (T-Nibble)	23.78	598
	LzOpt (VByte-Fast)	25.14	482
	Bc-Zip (VByte-Fast, 455 ms)	27.97	468
	Bc-Zip (VByte-Fast, 418 ms)	47.59	432
	Bc-Zip (VByte-Fast, 381 ms)	75.08	395
	Gzip	245.25	5,815
	Lzma2	17.62	1,681
	Snappy	448.67	1,301
	Lz4	333.74	1,007
	Bzip2	45.79	34,157
	BigBzip	42.02	152,000
	Ppmd	196.36	129,000
Wikipedia	LzOpt (T-Nibble)	175.86	3,080
	LzOpt (VByte-Fast)	191.19	1,748
	Bc-Zip (VByte-Fast, 1306 ms)	205.89	1460
	Bc-Zip (VByte-Fast, 973 ms)	270.35	1106
	Bc-Zip (VByte-Fast, 862 ms)	316.18	986
	Gzip	269.36	6,154
	Lzma2	166.16	9,871
	Snappy	422.80	1,093
	Lz4	309.51	862
	Bzip2	214.65	29,037
	BigBzip	150.88	151,000
	Ppmd	148.27	283,000

encoder is taken into account. We will close the speed gap w.r.t. Snappy and Lz4 with the Bicriteria Data Compression scheme.

Effectiveness of the bucketing strategy: Overall, experiments confirm that compression ratio does improve with a longer bucket size, but the exact improvement does depend on the peculiarities of the data being compressed. This implies that trading decompression time vs compression ratio via the choice of a proper bucket size requires a deep understanding of the data being compressed. In Figure 2 we show the decompression time when varying the bucket size. We only plot the results for Wikipedia and Dna because they suffice to capture the range of behaviors shown by the bit-optimal LZ77-compressor over our four datasets. In particular, we mention that the behavior on Census and Mingw is similar to that on Wikipedia, with the former reaching with VByte-Fast a decompression speed up to 2,200 MiB/sec with 4-MiB buckets that decreases to 1,800MiB/sec with 1-GiB buckets, while over Mingw the decompression speed is about 1,100MiB/sec for any bucket size. For both datasets, the speed is roughly halved when using T-Nibble, the reason being the word-boundary alignment of VByte-Fast's codewords which removes the need of bit-shifting them when reading from memory.

Figure 2 also shows that the dependency between the bucket size and the decompression speed of the bit-optimal LZ77-output highly depends on the characteristics of the data being compressed, but (possibly) in a counter-intuitive way. In Wikipedia, it generally decreases with larger bucket sizes, with a peek somewhere near 4MiB, instead of 1MiB; in Dna the decompression speed *improves* with larger bucket sizes. In Dna, which is highly repetitive, there are far back-

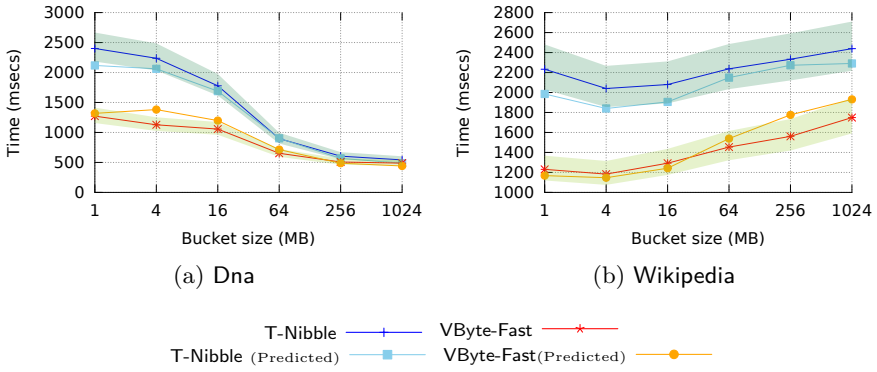

Figure 2. Decompression time by varying the bucket size on Dna and Wikipedia. The plot reports also the time predicted by our decompression-time model and a band around the decompression time capturing a relative error of 10%.

references which copy long portions of a genome which compensate the cache miss penalty induced by the copy (fewer phrases). On the other hand, Wikipedia is less repetitive and so far back-references added by larger windows are not much long, save little space, and thus they do not compensate the miss penalty incurred by their decompression. It is evident now that if we want to trade in a principled way decoding time *versus* compressed space, and thus ultimately improve the design of the bicriteria compressor Bc-Zip, we need to precisely explain and, thus, predict these phenomena. We designed a time model (full description in the journal version) which is capable of predicting decompression time with an average precision of $\approx 5.6\%$, which is a remarkable achievement as accurately predicting running times (that is, achieving an average precision of 10% or better) is notoriously an hard task [14]. In Figure 2 we plotted the predicted decompression time alongside actual decompression times. This model takes into account the cache miss latency to access distant substrings, the phrase decoding and the copying time. Thanks to this time model, Bc-Zip is capable of trading decompression time for compression ratio in a smooth and *consistent* way, as shown in the next paragraph.

Bicriteria compressor: Our implementation of the Bc-Zip compressor largely follows the scheme exposed in [7], using the Fast-FSG algorithm exposed in Section 3 and some minor algorithmic twists to accelerate compression. In our tests we compressed each dataset several times, for both VByte-Fast and T-Nibble, with time bounds ranging from the decompression time of the time-optimal parsing to the decompression time of the space-optimal one. In this way we can determine the whole range of trade-offs offered by Bc-Zip. Moreover, in order to directly compare Bc-Zip against the state-of-the-art compressors adopted in storage systems (such as Hadoop and BigTable), we compressed each dataset by setting its decompression-time bound (or compressed-space bound) as the

decompression time of the parsings generated by Lz4 (highlighted entries in Table 1). The average time model accuracy is $\approx 4.5\%$ (VByte-Fast $\approx 5.4\%$, T-Nibble $\approx 3.7\%$).

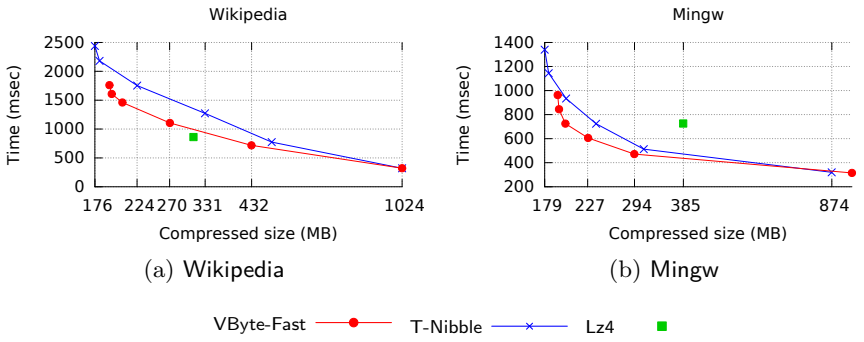

Figure 3. Space/time trade-off curve obtained with Bc-Zip, by varying the decompression time bound, and Lz4.

Table 1 and Figure 3 show the large range of trade-offs obtained by Bc-Zip. For instance, in Mingw spans from $\approx$ 300 msec to $\approx 1,400$ msec time-wise, and from $\approx$ 976MB to $\approx$ 179MB space-wise. Another interesting aspect is that T-Nibble is competitive against VByte-Fast only when maximum compression is required, otherwise the latter delivers more succinct parsings for the same decompression time. This is due to T-Nibble's relatively slow decoding, which forces the compressor to trade LZ77 copies for literals in order to meet the decompression time budget. This is in contrast with VByte-Fast's fast decoder, which does not impact much on decompression time and thus the compressor only cares about the cache behavior by substituting cache miss-inducing copies for a sequence of miss-free ones, a more succinct time-saving strategy.

Another interesting observation is that varying the decompression time impacts little on compressed size when more succinct parsings are considered, while it may impact considerably when less space-efficient parsings are taken into account (with varying degree: more accentuate with Mingw, less with Wikipedia). This provides a quantitative explanation of the natural question that motivated the work in [7], namely *"who cares whether the compressed file is slightly longer if this allows to improve significantly the decompression speed?"*. The present paper shows that the space/time trade-offs do not change linearly but, instead, a small change in one resource may induce a significant change in the other, unpredictably.

Moreover, Bc-Zip is extremely competitive with Lz4, since it clearly dominates it in three out of four datasets (Census, Dna, Mingw), while in Wikipedia it is very close, being only $\approx 12\%$ slower and $\approx 2\%$ less succinct than Lz4. Overall, Bc-Zip

performs more consistently thanks to its well-principled design, and surpasses the performance of well engineered, and widely used, compressor Lz4 on three out four datasets. We therefore believe that this is a nice success case of a win-win situation between algorithmic theory and engineering.

References

1. D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.
2. J. Békési, G. Galambos, U. Pferschy, and G. J. Woeginger. Greedy algorithms for on-line data compression. *J. Algorithms*, 25(2):274–289, 1997.
3. D. Borthakur *et alii*. Apache Hadoop goes realtime at Facebook. In *SIGMOD*, pages 1071–1080, 2011.
4. G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *ISAAC*, pages 173–182, 2009.
5. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *Tech. Rep. Digital*, 1994.
6. F. Chang *et alii*. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.
7. A. Farruggia, P. Ferragina, A. Frangioni, and R. Venturini. Bicriteria data compression. In *SODA*, pages 1582–1595, 2014.
8. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.
9. P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. In *SODA*, pages 768–777, 2009.
10. P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, 61(1):51–74, 2011.
11. P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM Journal on Computing (SICOMP)*, 42(4):1521–1541, 2013.
12. J. Katajainen and T. Raita. An analysis of the longest match and the greedy heuristics in text encoding. *Journal of the ACM*, 39(2):281–294, 1992.
13. S. T. Klein. Efficient optimal recompression. *Computer Journal*, 40(2/3):117–126, 1997.
14. L. Huang, J. Jia, B. Yu, B. Chun, P. Maniatis and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *NIPS*, pages 883–891, 2010.
15. D. Salomon. *Data Compression: the Complete Reference, 4th Edition*. Springer Verlag, 2006.
16. E. J. Schuegraf and H. S. Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval*, 10(9-10):309–319, 1974.
17. M. E. G. Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, 32(2):344–373, 1985.
18. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.
19. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23:337–343, 1977.
20. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.