Gipfeli - High Speed Compression Algorithm

Rastislav Lenhardt[I, II] and Jyrki Alakuijala[II]

[I]University of Oxford
United Kingdom
rastislav.lenhardt@cs.ox.ac.uk

[II]Google
Switzerland GmbH
jyrki@google.com

Abstract. Gipfeli is a high-speed compression algorithm that uses backward references with a 16-bit sliding window, based on 1977 paper by Lempel and Ziv, enriched with an ad-hoc entropy coding for both literals and backward references. We have implemented it in C++ and fine-tuned for very high performance. The compression ratio is similar to Zlib in the fastest mode, but Gipfeli is more than three times faster. This positions it as an ideal solution for many bandwidth-bound systems, intermediate data storage and parallel computations.

1 Introduction

Over recent years improvements in memory bandwidth have lagged behind advances in CPU performance. This puts other parts of the system under pressure and often makes I/O operations a bottleneck. Gipfeli[1] is a new high-speed compression algorithm which tries to address this issue by trading CPU performance used in data compression for improvements in I/O throughput by reducing the amount of data transferred. Its goal is to decrease both running-time and memory usage. An overview of throughput of different I/O components of a computer is given in Table 1.

There are several other high-speed compression algorithms. The main competitors are Snappy [1] (currently being the key part of the Google infrastructure and sometimes referred to also as Zippy), QuickLZ [2] and FastLZ [3]. The goal of this work is to have the algorithm with the best compression ratio in this category.

Recent trends and developments in computing often require that we reconsider past paradigms. Frequency increase of single CPU cores is hindered by physical limitations and demand for lower energy consumption. However, the overall CPU performance is boosted by using multi-core CPUs; Intel predicts a many-core era with massive parallel applications in the near future [4]. This trend is reinforced by significant growth in data usage by individuals and organisations.

A typical application of massively parallel computation might involve an individual system with four CPUs, each having six cores, with the shared network storage among many individual systems. In this scenario, since Gipfeli is designed to work well in

[1] Gipfeli is a Swiss name for croissant. We chose this name, because croissants can be compressed well and quickly.

SATA HDD 7.2k rpm read / write average speed	80 MB/s
Solid State Drives read / write average speed	180 MB/s
Ethernet 1 Gigabit/s maximum speed	128 MB/s
Gipfeli compression speed	$>$ 220 MB/s
Gipfeli decompression speed	$>$ 350 MB/s

Table 1. Throughput

parallel (see Section 3), the data in Table 1 suggest that speed of network I/O storage would need to be over 40 Gigabits/s to make Gipfeli compression a bottleneck.

Gipfeli uses backward references, as introduced by Lempel and Ziv in their famous 1977 paper [5], with very light-weight hashing and a 16-bit sliding window. This has already been suggested in the previous work of Williams [6], Fiala and Greene [7]. The algorithm consists of two main parts. The first part, LZ77, builds on the implementation in Snappy with several improvements discussed in Section 3. The main difference is the second part, which is rare in the area of high-speed compression algorithms: it is an application of entropy coding. Gipfeli uses a static entropy code for backward references and an ad-hoc entropy code for literals based on sampling the input. Sampling is neccessary, because there is not enough time and memory to read the whole input in order to gather all the statistics and build a conversion table. We also could not use Huffman codes [8] or arithmetic coding [9], because of their slow performance.

Gipfeli is open-source and available at *http://code.google.com/p/gipfeli/*.

Organisation. The paper starts with an overview of the algorithm and a description of the compression format in Section 2. It is followed in Section 3 by the key features and tricks we implemented and explanations of decisions we made in order to accelarate the algorithm and make it more useable. In our benchmarks in Section 4, we compare Gipfeli with other high-speed compression algorithms and consider also Zlib [10], the industry standard for compression. We run it with the compile-time option *FASTEST* switched on to make Zlib run even faster than in the fastest mode choosable at run-time. In the last section, Section 5, we reflect on the results of benchmarks and argue why Gipfeli is the best trade-off. Moreover we discuss our real-world experiment and possible applications.

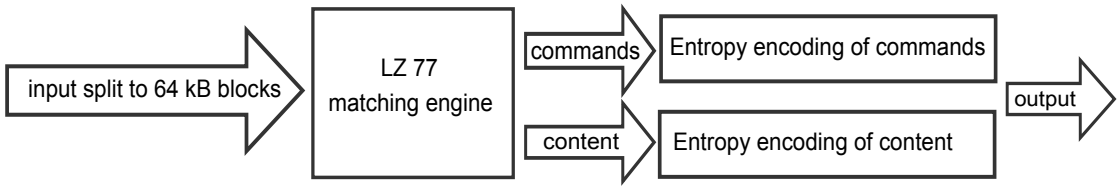

Fig. 1. Diagram of the algorithm

2 Compression Format

Figure 1 shows that the input string is processed in blocks of size 64 kB. The first part of the algorithm is the LZ77 matching engine, which either outputs parts of the string or notices that the same part has already been output and references it. In the next stage, we apply entropy coding to this output.

Example. Input *abcdeeabcdef* can be represented by the LZ77 matching engine as: (i) output *abcdee*; (ii) copy string of length 5 from the position 6 symbols backwards; (iii) output *f*.

The final compressed string has the form $ABBB...B$, where A contains the length of the uncompressed string and is followed by blocks B, where each block B is the result of compressing an input block of length at most 2^{16}.

Each block B consists of two parts: the first part contains *commands* and starts with their count; the second part contains *content*.

There are two types of *commands*:

- The *emit literals* command has a parameter *length*, which takes the next *length* symbols from the *content* and moves them to the output.
- The *emit copy* command represents a backward reference and has parameters *length* and *distance*. The command goes *distance* symbols back from the end of the output produced so far and copies from there *length* symbols to the end of the output.

2.1 Encoding Emit Literal Command

The whole compression is performed in blocks of size 2^{16}, allowing *length*, the parameter of the *emit literal* command, to have values between 1 and 65536. Therefore we can encode value $length - 1$, using 16 bits.

Since small lengths occur much more often in all reasonably compressible files (see Table 2 for statistics about text and html input), we use a shorter code in this case.

length	proportion	*length*	proportion
1	44.32%	6	3.16%
2	20.67%	7	1.88%
3	11.59%	8	1.38%
4	7.63%	9	1.00%
5	5.01%	≥ 10	3.40%

Table 2. *Length* in Emit literal command for text / html input

If *length* is at most 53 then the prefix 00 (representing the *emit literal* command) is followed by 6 bits representing the value. Otherwise we use values 53 to 63 to

specify the bit length (6 to 16) of the value $length - 1$, followed by this value using the specified number of bits.

2.2 Encoding Emit Copy Command

The command starts with a prefix which determines the number of bits used to represent *length* and number of bits used to represent *distance*.

length	*distance*	prefix	length bits	distance bits
4 - 7	1 - 1024	010	2	10
4 - 7	1025 - 8192	011	2	13
4 - 7	8193 - 65536	100	2	16
8 - 15	1 - 1024	101	3	10
8 - 15	1025 - 65536	110	3	16
16 - 67	1 - 65536	111	6	16

Table 3. Static entropy code used to encode backward references.

The entropy code for backward references is based on the gathered statistics for text and html input data (see Table 4 and Table 5).

length	proportion
4 - 7	81%
8 - 15	15%
16 - 67	4%

Table 4. Distribution of *length* of backward references

bits representing *distance*	proportion	bits representing *distance*	proportion
1	0.2%	9	7.4%
2	0.1%	10	8.9%
3	0.2%	11	10.5%
4	0.7%	12	11.8%
5	1.5%	13	13.3%
6	3.0%	14	14.8%
7	4.3%	15	10.8%
8	5.8%	16	6.6%

Table 5. Distribution of *distance* of backward references

2.3 Entropy Code for Content

The entropy code for the *content* of a particular block is used only when it is advantageous to do so. We discuss later how we decide whether to use it.

We order 8-bit symbols according to their frequency and split them into three groups:

- for the most frequent 32 symbols, we use 6 bits (prefix 0 + *index*)
- for the next 64 symbols, we use 8 bits (prefix 10 + *index*)
- for the last 160 symbols, we use 10 bits (prefix 11 + *value*)

where *index* is the alphabetical order of the symbol in the group, and *value* is the 8-bit char value of the symbol.

We need to communicate which symbols are in which group. We use a two-level bitmask to specify which 96 of the 256 symbols are in the first two groups. In the top level we specify for every block of 8 symbols whether one or more of them are in the first or second group; in this case, in the second level we specify exactly which of these 8 symbols are present. Finally, we use 12 bytes (a bitmask of length 96) to separate symbols of the first and the second group. If these 96 symbols are not specified in the first place, then no entropy coding is performed and each symbol is represented by its *value*, i.e. by 8 bits.

Sampling input to build the entropy code. For performance reasons, we do not read the whole *content* extracted from input block to determine which symbols occur most often and whether to perform entropy encoding. We use five samplers, which remember for each symbol if they have seen it or not. One small sampler looks at one symbol from each 43, three medium samplers look each at two symbols from each 43 and the large one looks at three symbols from each 43.

This information allows us to estimate the frequency of each symbol. It helps us to determine whether entropy coding makes sense, as we save two bits on the first 32 symbols, but we lose two bits on the last 160 symbols. It also induces a natural order on symbols, so we can split them into the groups in this order. This method, in which we use the estimated numbers from Table 6 (which need to be adjusted depending on the number of symbols in the last *SMMML* category, i.e. symbols spotted by all five samplers), proved to work very well in practice for identifying the split.

3 The Key Implementation Features

Limited memory usage, optimal parallel performance. A fast and useful algorithm must necessarily use as few resources as possible. Therefore we implemented Gipfeli such that its working memory, not taking into account input and output, fits comfortably in the Level-1 and Level-2 caches of a single CPU core. The implementation uses only slightly more than 64 kB. This is crucial, because accessing Level-1 and Level-2 caches is an order of magnitude faster than accessing RAM, which is still

samplers	proportion	samplers	proportion
$\emptyset$	$< 0.08\%$	SMM	0.55%
S	0.19%	MMM	0.62%
M	0.20%	SML	0.63%
L	0.22%	MML	0.72%
SM	0.34%	SMMM	0.9%
MM	0.36%	SMML	1.14%
SL	0.37%	MMML	1.67%
ML	0.41%	SMMML	$> 4\%$

Table 6. Relative proportion of symbols given the combination of samplers that have seen them.

much faster than accessing a hard drive. A direct consequence is that when we run six parallel compressions on CPU with six cores, there is no slow down. In other words, six cores of the same CPU can compress six times as much in the same time as a single core.

Very light-weight hashing in the LZ77 part. We allocate a large amount of memory for the hash table alone, in order to discover backward references in the LZ77 part of the algorithm. In our implementation the best performance trade-off is to allow 2^{15} hash table entries. We always read four consecutive symbols as an unsigned 32-bit integer and hash it to a value less than 2^{15}. Each entry of the hash table is a 16-bit unsigned integer, which allows us to reference backwards as far as the size of the block. This has already been implemented in Snappy.

References to the previous blocks and no need to reset the hash table. Unlike Snappy, we have added to Gipfeli support for backward references to the previous block. Each 16-bit *value* of a hash table entry represents how far from the beginning of the block are the four symbols that were hashed to this entry. If this *value* is smaller than the current distance from the beginning of the block then it represents a position within the current block. Otherwise it represents a position from the previous block.

The size of the block and the size of the entry value is 16-bit, therefore each entry of hash table reference to the valid position with the exception when we are in the first block. Therefore we initially reset all hash table entries to 0 to obtain this property. There is still one special case, and it is position 0 of the first block, which we can solve simply by not looking for backward reference in this case.

As we move to the next block, we adjust the pointer to the start of this block. Observe that our semantics for the values in hash table gives automatically support for backward reference to the previous block without any need to adjust the values in the hash table.

High performance for incompressible input. Most of the algorithms become much slower for non-compressible inputs. Snappy and Gipfeli are exceptions (see jpeg in Table 7) thanks to an idea of Steinar Gunderson, who suggested we keep increasing the size of steps when looking for the next backward reference if we are not successful in finding one for some time.

Writing and reading bits. The static entropy code for backward references and the ad-hoc entropy code for literals are bit-based. That brings new challenges, because just writing bits to the output in the compression phase and reading bits from the input in the decompression phase constitutes a significant portion of the running time of the whole algorithm (20 - 30%).

Our experiments have shown that the best way to manipulate bits is to use an unsigned 64-bit integer as a buffer and then always write the whole buffer at once to the output. The next improvement is used mainly after applying the ad-hoc entropy code to literals: since each symbol can be represented by at most 10 bits, we can always safely pack six of them to one 64-bit integer and write it at once.

The other improvement concerns backward references. There are six different prefix codes (see Table 3), and it helps siginificantly if we avoid branching. We can achieve this by noticing that the bit length of *distance* and *length* determine in which case we are. So we can have static tables of constants that determine how to build the bit value for the given backward reference. This also explains why we tried to keep the entropy code simple and did not allow higher variability.

Fast construction of entropy code for *content*. Recall the sampling technique explained in the previous section. Despite its simplicity, it works very well and achieves over 20% savings for the *content* from text files. That is very close to the theoretical optimal 25% for this code, which we get if we save two bits for every 8-bit symbol.

Even though the main savings (over 75%) comparing to Snappy are from entropy code used for *commands*, it still makes sense to do entropy code for the *content*. It is computationally cheap and decreases the number of bits to be written to the output, so it makes the final phase of the algorithm faster. It is a little counter-intuitive, but when writing the output is an expensive operation, improvement in the compression ratio also often leads to an improvement in performance.

Unaligned stores are much better than *memcpy* for short lengths. As the statistics show (Table 2 and Table 4) we often need to copy only a few bytes (either to the output or during the creation of intermediate *content* from the input). A much faster solution than using *memcpy* is to perform unaligned stores of unsigned 32- or 64-bit integers at the pointer position.

File description	Compression ratio			Speed		
	Snappy	QuickLZ	Gipfeli	Snappy	QuickLZ	Gipfeli
text 1	59.8%	54.9%	46.4%	311 MB/s	266 MB/s	220 MB/s
text 2	56.2%	49.3%	40.6%	341 MB/s	271 MB/s	236 MB/s
text 3	57.1%	51.9%	43.6%	327 MB/s	277 MB/s	229 MB/s
html	23.6%	19.4%	19.7%	780 MB/s	466 MB/s	528 MB/s
url addresses	50.9%	43.4%	39.1%	428 MB/s	307 MB/s	272 MB/s
protocol buffer	23.2%	15.8%	17.5%	870 MB/s	512 MB/s	713 MB/s
jpeg	99.9%	100%	99.9%	5 GB/s	374 MB/s	2 GB/s
pdf	82.1%	100%	81.5%	1500 MB/s	382 MB/s	909 MB/s
C source code	42.4%	42.3%	37.9%	431 MB/s	279 MB/s	273 MB/s
LSP source code	48.4%	47.7%	42.9%	425 MB/s	237 MB/s	246 MB/s
executable	51.1%	45.7%	44.7%	397 MB/s	313 MB/s	253 MB/s

Table 7. Comparison of Snappy, QuickLZ and Gipfeli

4 Benchmarks

In this section we compare the speed and compression ratio of Gipfeli and its competitors. All files in our benchmarks are from previous benchmarks of either QuickLZ or Snappy. These files are originally from the Calgary and the Canterbury corpus [11]. We decided to pick several of them representing the overall performance of the algorithms and putting more emphasis on text and html content. The benchmarks were performed on an Intel Xeon W3690 CPU using only one of its cores, where each core has its own 32 kB L1 instruction cache, 32 kB L1 data cache and 256 kB L2 cache.

The results are presented in Table 7 and plotted for one text file in Fig. 2. To put the performance in context with Zlib (in the default and the fastest compression mode), we performed a standard comparison on the first 1 GB of data from English Wikipedia (see Table 8).

Similarly to most of the compression algorithms, the decompression phase of Gipfeli is faster than the compression phase. In all cases, it is at least 60% faster.

Program	Compression ratio	Time	Speed
Snappy	53%	2.8 s	354 MB/s
QuickLZ	46%	3.5 s	284 MB/s
Zlib fastest	43%	13.5 s	74 MB/s
Gipfeli	41%	4.3 s	232 MB/s
Zlib default	32%	41.7 s	24 MB/s

Table 8. First 1 GB of English Wikipedia

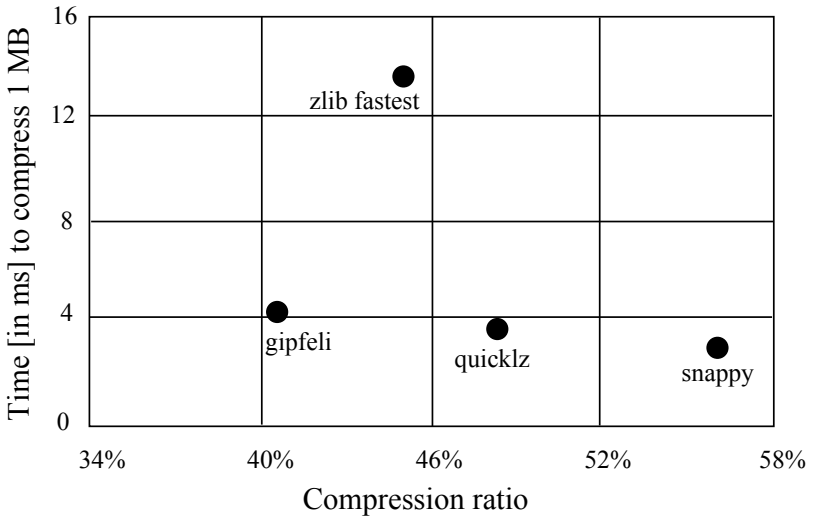

Fig. 2. Benchmark for plaintext.txt

5 Applications and Conclusion

The benchmarks in this paper show that Gipfeli can achieve compression ratios that are 30% better than Snappy with slow-down being only around 30%. Gipfeli achieves even higher speed for html content and remote procedure calls (protocol buffer in Table 7) than for text content. We argue, using the I/O data from the introduction, that once the compression algorithm is fast enough, i.e. computations are bound by external I/O costs, we need to compress as densely as possible. At that point, improvements in the compression ratio are more important than running time.

The encouraging outcome of the benchmarks led us to test Gipfeli in a real-world setting to support our theoretical assumptions with practical experiment. Our case study was MapReduce technology [12] used inside Google to run distributed computations. A typical computation processes terabytes of data and runs on thousands of machines. In short, MapReduce consists of two phases: the first phase, Map, applies some computation in parallel to all the input items to produce the intermediate items, which are then merged in the second phase, Reduce.

Currently, Snappy and Zlib are both options used inside MapReduce. We replaced Snappy by Gipfeli and our experiment confirmed our expectations: the computation was faster (up to 10%) and led to lower RAM usage. Since Gipfeli can compress better, it is now a candidate replacement for both Snappy and Zlib (which is currently used in the situations where a better compression ratio is needed) in MapReduce. Other plausible applications are: the replacement of Snappy in Bigtable technology [13], which is used to store large amounts of data; and in Google's internal remote procedure calls.

Gipfeli is currently in the alpha phase. The algorithm has not gone through complete testing, but a few terabytes of data have been successfully pushed through it. Compared to Snappy, QuickLZ and FastLZ, which have been tuned for several years,

Gipfeli is still young. We have open sourced it to allow additional improvements and optimisations by the community.

Acknowledgement. We would like to thank Steinar Gunderson for several optimisations of the code and for well-written and well-optimised code for matching backward references in Snappy, which we partly reuse.

References

1. S. Gunderson, "Snappy." http://code.google.com/p/snappy/.
2. L. M. Reinhold, "Quicklz." http://www.quicklz.com.
3. A. Hidayat, "Fastlz." http://www.fastlz.org.
4. S. Borkar, "Platform 2015 : Intel processor and platform evolution for the next decade," *Systems Technology*, 2006.
5. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
6. R. Williams, "An extremely fast ziv-lempel data compression algorithm," in *Data Compression Conference, 1991. DCC '91.*, pp. 362 –371, apr 1991.
7. E. R. Fiala and D. H. Greene, "Data compression with finite windows," *Commun. ACM*, vol. 32, pp. 490–505, April 1989.
8. D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098 –1101, sept. 1952.
9. I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, pp. 520–540, June 1987.
10. P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3." RFC 1950 (Informational), May 1996.
11. R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Data Compression Conference, 1997. DCC '97. Proceedings*, pp. 201 –210, mar 1997.
12. J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.
13. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design And Implementation*, pp. 205–218, 2006.