



A.S. Bugaev, ORCID: 0009-0003-3153-2819 <ax.bugaev@gmail.com>
G.K. Sichkar, ORCID: 0009-0005-4684-9610 <georgy.sichkar@yandex.ru>
A.I. Vasenina, ORCID: 0000-0003-3457-0610 <a.vasenina@spbu.ru>

Acknowledgements. The experimental results were obtained using the hardware generously provided by Shvacher LLC.

A.C. Бугаев, ORCID: 0009-0003-3153-2819 <ax.bugaev@gmail.com>
Г.К. Сичкар, ORCID: 0009-0005-4684-9610 <georgy.sichkar@yandex.ru>
А.И. Васенина, ORCID: 0000-0003-3457-0610 <a.vasenina@spbu.ru>

Lossless compression is a type of compression that does not permit any data loss during the compression/decompression process [1, 2]. Data storage systems apply lossless compression as a means of utilizing the disk space more efficiently [3-5]. One of the features commonly seen in such systems is transparent compression – lossless compression handled by one of the kernel-level subsystems, which does not affect the way user interacts with data [6]. The focus of this paper is the use of compression on logical volumes, which leads to application of compression algorithms at the block layer of the operating system kernel [7]. Viability of the aforementioned data compression heavily depends on the choice of the algorithm, which has to be lightweight enough for compression

to run in-flight, as well as on the way the algorithm is applied. For in-flight (or online) compression, the main priority is low latency, achieved with high throughput [6]. In contrast, offline compression, which runs outside of the critical I/O path, focuses on compression ratio [8].

The disk throughput is generally higher with larger I/O requests due to the decreased overhead that comes with handling a single I/O request at the block layer [9]. However, allocating a single large buffer to hold data for each large request is problematic due to memory fragmentation. Hence, the data of an individual I/O request in the block layer of the Linux kernel is rarely stored as a single contiguous chunk: instead, the data is split into smaller contiguous chunks, usually by one page, which is 4 KiB in size on x86_64 architectures [7]. Such a chunk can be called a vector, and a sequence of vectors is called a scatter-gather buffer.

LZ4 is a lossless, dictionary-based [10] compression algorithm, implementation of which is included in the Linux kernel, making it available for use by external kernel modules. According to several benchmarks and papers [11, 12], it has one of the highest compression and decompression speeds available, while commonly reducing the size of data up to 2 times. Hence, it is well-suited for in-flight compression in the Linux kernel block layer. Dictionary-based compression algorithms reduce data size by finding matches in previously processed data, usually limited by some fixed-size sliding window [10]. Therefore, for compression to be stronger, the algorithm has to be able to access as much preceding data as fits into that window. Consequently, in order to guarantee the highest compression ratio on data of a single block layer I/O, it is necessary to allocate a contiguous buffer that fits all of this data, as well as to copy the data into this buffer. As an alternative, it is possible to run compression separately on each of the vectors in a scatter-gather buffer, while using a single previously processed one as a dictionary. The dictionary is used for finding matches, as if the current chunk directly follows it [13]. This way, however, the compression ratio depends on the layout of the scatter-gather buffer and is generally lower in comparison, since using only a single vector as the dictionary limits the possible back reference for match searching.

The standard implementation of LZ4 is based on pointer arithmetic [14] and only handles contiguous buffers [15]. In this paper we propose a modification of LZ4 algorithm based on common block layer abstractions – scatter-gather buffer iterators [16]. Our modification further allows compression to be run on whole scatter-gather buffers at once, improving compression ratio, while not requiring any contiguous buffer allocation and data copying. Eliminating buffer allocation leads to reduced memory usage, which is valuable for high-load systems. We hypothesized that eliminating data copying would provide sufficient throughput for online compression. If successful, the results can be applied to similar compression algorithms, such as Zstandard [12, 17].

We aimed to compare the performance of the proposed solution, which we called LZ4E, to the mentioned applications of LZ4 in the block layer, including copy-based approach and separate compression of I/O vectors. The research questions that we address in this paper are as follows.

RQ1. What approaches achieve the best compression ratio?

RQ2. How much memory does LZ4E save compared to the contiguous approach?

RQ3. What is the throughput of LZ4E algorithm and how does it compare to the throughput of other compression approaches?

The rest of the paper is organized as follows. Section 2 gives a brief overview of existing accomplishments in the field, as well as presents a work that is similar to ours. Section 3 provides the information necessary for a better understanding of our work, describing LZ4 algorithm and its existing implementation in the kernel, as well as the existing API of the Linux kernel block layer. Section 4 describes the general idea and design behind our LZ4 modification, while examining key details of the implementation and explaining the choices that were made. Section 5 describes the way our LZ4 extension was tested and compared to the original implementation, as well as provides the obtained data, interprets the comparison results, and answers the formulated research questions. Finally, Sections 6 and 7 propose plans and ideas for further investigation on this topic, and provide a brief conclusion to this paper.

2. Related works

B. Warnke in 2017 proposed an extension of LZ4 for page-based buffers, comprised of pages scattered in memory, as well as presented a new algorithm, called BeWalgo, designed specifically for processing such buffers [18]. The main restriction of the algorithm is that the buffers must be composed of whole pages and cannot have fragmentation that is possible in block layer vectors, which have variable length and might not be aligned to the page boundaries. As stated in the paper, such an extension was implemented by replacing pointer operations with calls of functions, which handle work with page-buffers internally, using some helper structure that describes current position into the buffer while storing page addresses in an array. As a result, the extended version of LZ4 was evaluated to be slightly slower than the original one, taking data copying into account, while providing the same compression ratio. The suggested reason for this performance degradation is the increased complexity of simple read, write and copy operations, caused by necessity of handling page boundary crossing. The proposed BeWalgo algorithm, however, seems to be significantly faster on page-based buffers, while providing weaker compression on some of the datasets.

Additionally, of high relevance to our work is research on transparent compression in the Linux kernel block layer, performed in years 2010–2012 by scientists of Foundation for Research & Technology – Hellas [6, 19, 20]. At the time, this series of works outlined the potential of block-level compression, as compared to that on the filesystem level, by presenting a block-storage layer for transparent compression, called ZBD. Later works expanded on this topic, combining ZBD layer with use of SSDs based I/O caches. As a result, the block-level compression was shown to be a viable option for compressing data in-flight, while requiring additional optimizations and careful metadata management.

3. Background

This section is organized as follows. Firstly, we describe the format of generic LZ4 block. Secondly, we describe an implementation of LZ4 presented in the Linux kernel source tree. Lastly, we describe some core structures of the Linux kernel block layer.

3.1 LZ4 block format

An LZ4 block [21] is composed of sequences. Each sequence is comprised of a string of literals, or uncompressed bytes, and some variable-length metadata. Some part of this metadata provides information about the contained literal string. The remaining part encodes the match found in data, that precedes current position during compression. In more detail, an LZ4 sequence is laid out in the following order, also shown in Fig. 1:

- 1) 1 byte token – first 4 bits of literal length, last 4 bits of match length;
- 2) 0 or more bytes of literal length – length of the literal string in bytes;
- 3) 0 or more bytes of literals – contiguous string of uncompressed bytes;
- 4) 2 bytes of offset – offset from current position to find match in the preceding data;
- 5) 0 or more bytes of match length – length of the found match in bytes.



Fig. 1. Layout of generic LZ4 sequence.

The length of the literal string can also be zero, meaning that sequence contains only a match. For terminating an LZ4 block, each of the following conditions must be met:

- the last sequence contains only literals, meaning no match;

- the last 5 bytes of input are always literals;
- the last match must start at least 12 bytes before the end of block.

The process of decompressing an LZ4 block is quite straightforward. Decompression of a single sequence in the block starts by copying the literals first, using encoded literal length, followed by copying the match, which is found in previously decompressed data using the encoded offset from current position and the match length.

3.2 Linux kernel LZ4 implementation

This subsection provides a general overview on LZ4 algorithm presented in the kernel, as well as some implementation details. Most commonly used compression functions in the kernel are `LZ4_compress_fast` and `LZ4_compress_default` [15]. `LZ4_compress_fast` is more generic, and has the following arguments:

- `source` – pointer to the start of the source buffer;
- `dest` – pointer to the start of the destination buffer;
- `inputSize` – size of the data to compress;
- `maxOutputSize` – maximum allowed size for the compressed block;
- `acceleration` – factor for acceleration of the compression;
- `wrkmem` – pointer to the working memory required for compression.

The signature for `LZ4_compress_default` is identical, except that the acceleration factor is omitted, with the default value of 1 being used. Compression requires preallocated working memory of `LZ4_MEM_COMPRESS` [15], which is around 16 KiB. This memory is used for an internal dictionary, represented by a hash table that stores source data byte positions relative to the start of the buffer. To insert a position into the hash table, the algorithm reads 4 or 8 bytes from that position and hashes the read data. The kernel implementation of LZ4 supports the variable size of the hash table entry. Entries are stored as 2 bytes if size of the input is less than or equal to 64 KiB. Otherwise, hash table entries are 4 bytes in size. Smaller size of entries makes it possible to store 8192 positions instead of 4096, reducing the number of collisions and thus improving compression.

The sliding window for match searching has a fixed size of 64 KiB [15] that allows the offsets to be encoded by 2 bytes. The algorithm finds matches by iterating through the source buffer. At each step it obtains the potential match positions from the dictionary and adds current position to the hash table. The initial iteration step depends on the acceleration parameter and is incremented each 64 steps.

Before the start of the main loop, the first byte of source data is processed and its corresponding position is added to the hash table. After that, an anchor is set to the position of the first byte in the source buffer that has not been encoded yet. The main loop can be split into the following steps [22].

- 1) Iterate through source buffer until a dictionary match is found within the sliding window. Add entries to the hash table along the way.
- 2) Roll back both the source and the match positions as far as possible to get the maximum match length.
- 3) Calculate the literal length as the distance from the anchor to the current source buffer position and write it to the token and the following bytes.
- 4) Copy bytes starting from anchor up to the current source buffer position into the destination buffer.
- 5) Write the offset as the distance between source and match positions.

- 6) Calculate the length of the match by comparing the bytes starting from the match position and the current source position. Write it to the token and the following bytes after offset.
- 7) Advance source position to first byte after the match and update the anchor.
- 8) Check the updated source position for a match. If the match is found then jump to step 5, otherwise jump to step 1.

Fig. 2 illustrates how a resulting LZ4 sequence is formed after the match is found. The main loop terminates when the block end conditions are met during the match search. To complete the compression, the remaining literals from anchor are copied to the destination buffer. The loop also terminates in case of destination buffer overflow resulting in a compression error. The overflow check is skipped if the compressed block is guaranteed to fit within the destination buffer even if the source data is incompressible. The exact upper bound of the size of the compressed block can be obtained by `LZ4_COMPRESSBOUND(source_data_size)` macro [15]. The resulting value is slightly larger than size of uncompressed data. Skipping the overflow checks allows the compression to run slightly faster.

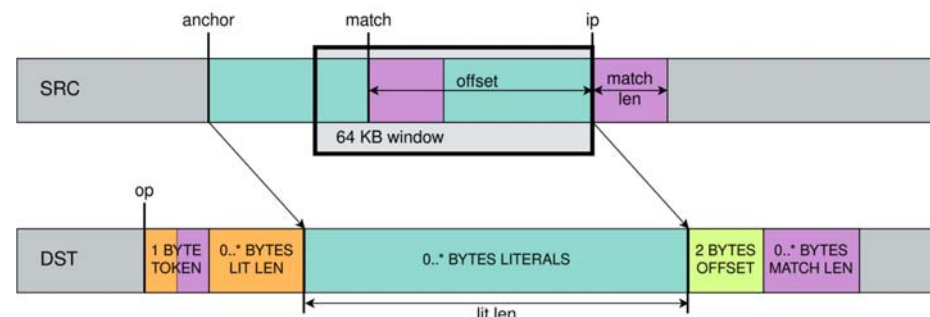


Fig. 2. Creation of LZ4 sequence after finding a match.

Compression functions described above work on contiguous buffers, and do not save context, such as hash table entries, between individual function calls. Therefore, using this API for running compression on individual vectors of a scatter-gather buffer results in a significantly lower compression ratio. In order to achieve the highest compression ratio, one has to copy the data into a single preallocated contiguous buffer. However, it is possible to save some of the compression context by using the Streaming API of LZ4 [13]. Essentially, when using this API, we are not resetting the working memory between calls, while also using a single previously compressed contiguous chunk as a dictionary for match searching. Here, the dictionary is a chunk of already processed literals, not to be confused with the hash table that stores the byte positions. The dictionary to be used is selected as the largest contiguous chunk of preceding data fitting into 64 KiB window. This selection happens internally using pointer arithmetic [14]. Depending on location of the dictionary in memory, it is configured for one of two modes [22]:

- `prefix mode` – the input data immediately follows the dictionary;
- `external mode` – the dictionary is stored at an arbitrary location, which does not overlap with the input data.

The streaming approach compresses the data in the current buffer as if the dictionary directly precedes it. Fig. 3 illustrates the difference between contiguous and streaming compression approaches. Currently the only streaming compression function exported for kernel use is `LZ4_compress_fast_continue` [15]. Its signature is almost identical to that of `LZ4_compress_fast` except for the `LZ4_stream_t` argument replacing `wrkmem` for keeping track of the context of the compression.

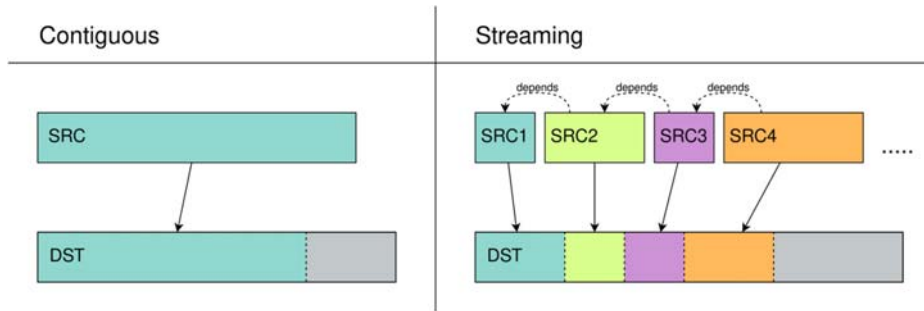


Fig. 3. Regular (left) and Streaming (right) LZ4 APIs.

3.3 Linux kernel block layer API

This subsection describes data structures used in the block layer of the Linux kernel for representing and handling scatter-gather buffers. A single contiguous data buffer (vector) is represented in the block I/O by struct `bio_vec` [7], which has the following fields:

- `bv_page` – pointer to the first page on which this vector lies;
- `bv_offset` – offset in bytes from page start to the start of this vector;
- `bv_len` – length of this vector in bytes.

The `bio_vec` is often shortened to `bvec`. Data held by a single vector is not necessarily mapped into the virtual address space: rather than storing a pointer to the data, `bvec` contains an address of struct page, using which this pointer has to be obtained. On systems with high memory [23, 24], or memory that is not permanently mapped into virtual space, that might require creating a temporary mapping, which can be quite costly. On 64-bit systems, however, virtual address space is large enough to map all physical addresses directly, and an address of the data can be obtained by a simple bit of pointer arithmetic. A single `bvec` only holds a pointer to the first struct page. However, the data held by `bvec` can span multiple contiguous pages. This concept is called multi-page `bvec`, as opposed to single-page `bvec`, and was introduced into the kernel for improving block I/O efficiency [25]. On another note, `bvecs` have an arbitrary offset and length and might not be aligned to the page boundaries. Therefore, generally it is not possible to avoid contiguous buffer allocation and data copying by mapping these pages into a single contiguous virtual space using `vmap` function [26].

A scatter-gather buffer is represented as a sequence of `bvecs`, commonly stored as an array [7]. Fig. 4 illustrates the structure of an arbitrary scatter-gather buffer. The iteration through a scatter-gather buffer is done with an iterator, represented by struct `bvec_iter` [16]. Among its fields we mark out the following:

- `bi_idx` – index of current `bvec` in the array;
- `bi_bvec_done` – number of bytes processed in current `bvec`, or offset from start of `bvec`;
- `bi_size` – residual length of the buffer.

There are a number of helper functions and macros for this type of iterators. For instance, to advance iterator by a specified number of bytes, one can use `bvec_iter_advance` or `bvec_iter_advance_single` [27]. The latter is used for iteration within a single `bvec`, while the former can be used for the general buffer iteration. These functions require the underlying `bvec` array to be provided alongside the iterator. It is possible to obtain `bvecs` in-flight during iteration by using `bvec_iter_bvec` and `mp_bvec_iter_bvec` [27] for single-page and multi-page `bvecs`, respectively.

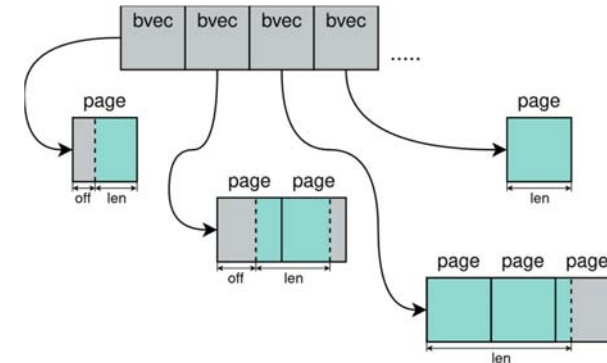


Fig. 4. Generic structure of a scatter-gather buffer.

Finally, a single I/O in the block layer of the Linux kernel is represented by struct `bio` [7], containing a scatter-gather buffer as an array of `bvecs`, as well as a start iterator. The size of the I/O is stored within the iterator. Number of `bvecs` in a single block-level I/O is limited, and the current limit is equal to 256, provided by macro `BIO_MAX_VECS` [28].

4. Implementation

This section outlines a general design and idea of our LZ4 extension for scatter-gather buffers and provides a more detailed implementation description.

4.1 General design

Instead of writing an algorithm from scratch, we directly modify the standard version of the algorithm, taken from the Linux kernel source tree. While the essence of the algorithm stays the same, our modification is based on abstracting the buffer iteration, for the purpose of replacing the uses of pointer arithmetic. The algorithm was adapted for handling `bvecs` using `bvec` iterators, which required modification of existing helper functions for read-write-copy operations, similarly to what was done in [18]. Additional helpers were added for these operations on a single `bvec`, as well as on the whole scatter-gather buffer starting from given position, which is represented by a `bvec` iterator. As a baseline for the operations on a contiguous chunk, the existing LZ4 helper functions were used. Positions in the buffer are also represented by iterators, along with regular integer values, which are used for calculation of lengths and offsets. Addresses stored in the hash table are replaced with separate minimal structures encoding the position inside the scatter-gather buffer. These structures are comprised of index in the `bvec` array, relative to the index of some base iterator, and an offset from start of the corresponding `bvec`. The API of extended LZ4 compression, which we called LZ4E, accepts minimal required representations of scatter-gather buffers. For instance, the arguments of the extended fast compression function, `LZ4E_compress_fast`, are as follows:

- `src` – source buffer as an array of `bvecs`;
- `dst` – destination buffer as an array of `bvecs`;
- `srcIter` – pointer to iterator at the start of data to compress;
- `dstIter` – pointer to iterator at the start of compressed block to be written;
- `acceleration` – LZ4 acceleration factor;
- `wrkmem` – pointer to preallocated working memory.

As well as the standard LZ4, LZ4E compression function returns the number of bytes written to the destination buffer or zero in case of an error. The size of the data to be compressed, as well as the maximum allowed size of output, can be specified using `bi_size` field of the iterators of the source and destination buffers. These iterators are advanced inside the compression function, so by the end of the function call they are pointing to the first bytes following the processed data in their respective buffers. This API was designed for usage in the block layer. Therefore, we decided to limit the number of bvecs that can be processed by compression function to `BIO_MAX_VECS`, or 256, so that the indices of the bvec array can be encoded by a single byte.

4.2 Implementation details

Similarly, to the original version, which uses different hash table address lengths depending on the input size, our modification supports different addresses chosen based on the layout of the provided source buffer. There are three types of addresses that can be used, depending on number of given bvecs, as well as the maximum size of bvec, defined as follows:

byU16:

- 4 bits of `idx`, encoding up to 16 bvecs;
- 12 bits of offset, encoding bvecs up to 4 KiB, which is the common size of a single page.

byU32:

- 8 bits of `idx`, encoding up to 256 bvecs, which is equal to `BIO_MAX_VECS`;
- 24 bits of offset, encoding bvecs up to 16 MiB.

byU64:

- 8 bits of `idx`, encoding up to 256 bvecs as well;
- 32 bits of offset, supporting bvecs of maximum size theoretically possible at the moment;
- 24 bits reserved, available for future use.

These configurations were chosen based on a possible entry length (2, 4 or 8 bytes) and common bvec size limit (4 KiB). The address type is decided at runtime, as a part of initialization before the main loop. To make this decision, we iterate over each multi-page bvec of the source buffer using the provided start iterator.

All the data processing in the extended algorithm, including read-write operations and iteration through buffers, requires a respective iterator. Therefore, we need to construct an actual iterator after obtaining an address from the hash table, in order to check for potential matches. While a single address can encode a position inside the scatter-gather buffer, it is not sufficient for creating an iterator for this position. The missing piece for the iterator is `bi_size`, or the number of bytes left to process in the whole buffer. For this reason, we save `bi_size` of iterator at the start of each bvec into a separate array. For obtaining this value, index from the address structure is also used as the index into the array. This array requires 1 KiB of additional working memory: 4 bytes for each of 256 possible bvecs. The actual values are saved to array during initialization, while iterating over multi-page bvecs of the source buffer. Using this array, called `bvIterSize`, along with `baseIter` iterator, which points to the start of the respective buffer, a given address `addr` can be converted to an actual iterator `iter` using the following calculations.

```
iter.bi_idx = addr.idx + baseIter.bi_idx
iter.bi_bvec_done = addr.offset
iter.bi_size = bvIterSize[addr.idx] - addr.offset
```

And for the opposite conversion, in order to obtain address from `iter` and `baseIter`, the following equations are used.

```
addr.idx = iter.bi_idx - baseIter.bi_idx
addr.offset = iter.bi_bvec_done
```

For obtaining length of the literal sequence after finding a match, we need a regular integer position `pos`, which can be calculated using base iterator and the current iterator as follows.

```
pos = baseIter.bi_size - iter.bi_size
```

Fig. 5 provides a visual overview of the transformations above.

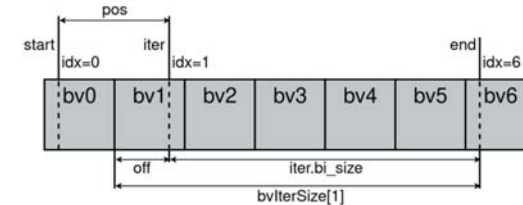


Fig. 5. Calculations made using proposed hash table addresses and the additional `bvIterSize` array.

As a base for scatter-gather buffer iteration, existing API for bvec iterators was used directly or slightly modified. For small iterations by one byte, more optimized versions were added. Missing functions for rolling an iterator back, required for finding the largest match, were implemented in a similar fashion. Helper functions for read-write-copy operations, as well as higher level functions for finding the length of the match, were modified using this API.

On level of the whole scatter-gather buffer, these helpers carefully handle the fragmentation by constructing bvecs in-flight and performing memory access based on how many bytes are left in current bvec. On systems without high memory, we are able to use multi-page bvecs, since all the contiguous pages of a single bvec are mapped directly into a contiguous virtual space. This allows us to handle whole chunks at once, thus improving throughput. On systems with high memory, however, some pages might not be mapped, or scattered over virtual memory. This is handled by processing each page individually, using in-flight construction of single-page bvecs. The utilization of multi-page bvecs is decided at compile time, based on macro `LZ4E_MULTIPAGE`, that is set by default if kernel option `CONFIG_HIGHMEM` [24] is disabled.

For processing a single bvec, we operate similarly to how `memcpy_from_bvec` and `memcpy_to_bvec` [27] work. To access the data, we have to obtain a pointer to the data from the page, after mapping this page into virtual memory, if necessary, on systems with high memory. After obtaining necessary addresses, this data can be copied between the corresponding contiguous buffers. Finally, mapped pages have to be unmapped by the end of the procedure. Instead of using the bitwise `memcpy`, we use a custom copying function, that is based on unaligned loads/stores using the existing LZ4 helpers, while not overwriting any data after the target copy length. This can improve performance for small reads and writes, which are especially common during match searching. As an optimization, that can be enabled statically at compile time, the algorithm supports obtaining pointers to the starts of each bvec in advance, so that these addresses are not required to be calculated in-flight when performing read-write operations on the data of bvecs. The pointers are saved to arrays in working memory when iterating through multi-page bvecs as part of initialization. These arrays, both for source and destination buffers, each require 2 KiB of additional memory on 64-bit systems, since we support 256 bvecs at most while a single pointer has length of 8 bytes. As with `LZ4E_MULTIPAGE`, this optimization is not guaranteed to work on systems with high memory, since we are not saving pointers for individual pages. The optimization can be enabled during compilation using macro `LZ4E_PREMAP`.

The complete implementation of the extended algorithm can be found on GitHub [29]. It is presented as an external kernel module, which exports functions of the described API, thus making LZ4E compression available for kernel use.

5. Evaluation

This section provides an insight on how the extended LZ4 algorithm was tested and compared to the original using different APIs, along with the results that were obtained in the process. Here we answer the research questions posed in Section 1.

5.1 Comparison approaches

We aimed to compare our LZ4 modification in the block layer to other possible approaches of I/O request compression. We selected four targets for comparison, which are described as follows.

- Contiguous – data of I/Os is copied into a preallocated contiguous buffer and compressed using regular LZ4 API.
- Vectorized – data of each bvec is compressed individually, using the regular API as well. No additional source buffers are allocated.
- Streaming – same as Vectorized, but the LZ4 Streaming API is used to improve compression ratio.
- Extended – data of I/Os is compressed using the proposed extension of LZ4 for scatter-gather buffers.

To estimate the proposed approach, we carried out the experiments with the collection of the following metrics:

- Compression ratio, as ;
- Compression throughput, as ;
- Memory usage, as amount of the memory required for running compression.

Memory usage is calculated as a sum of destination buffer sizes and the size of the working memory for compression, except for the Contiguous case, in which the source buffer size is added as well. Additionally, we measure the time spent for data copying, which is relevant only to the Contiguous method. This allows us to examine how much of an overhead data copying actually brings.

We aimed to compare all the methods in equal conditions. Therefore, in order to maximize performance of compression for each variant, destination buffers are allocated using the upper size bound for compressed block. This includes Vectorized and Streaming cases, where destination buffer is allocated for each bvec. For the same reasons, all of the buffers are allocated as physically contiguous, and not only virtually contiguous. In order to minimize variance, which might occur due to emergence of multi-page bvecs during I/O generation, all of the I/Os are additionally normalized to having equal bvec lengths. Furthermore, we specify to the kernel to disable preemption during time measurement, to avoid any interference with running timers.

5.2 Testing framework

To test compression on I/O requests we implemented a driver for a virtual block device, which includes LZ4 compression into I/O processing using described methods. From user's point of view, this proxy is transparent, meaning that the read data is the same as the written data. Such a block device must be created over some underlying block device, which is specified through parameters found in the sysfs virtual filesystem [30]. The compression method to be used can be selected via sysfs as well. For processing a single block-level I/O, its data is compressed and decompressed back using the standard LZ4 decompression, after which I/O is remapped to the underlying device. This allows us to test the correctness of our extended LZ4 implementation. Additionally, this block device

supports various statistics, also available as read-only sysfs parameters. The most relevant statistics are:

- `decomp_size` – size of data before compression, in bytes;
- `comp_size` – size of data after compression, in bytes;
- `comp_ns` – time elapsed during compression, in nanoseconds;
- `copy_ns` – time spent on data copying, in nanoseconds;
- `mem_usage` – memory used for compression, in bytes.

For measuring time intervals, we used `ktime` interface [31], which provides access to high-resolution timers. We used this API by a custom macro, that wraps timer usage around exactly one function call, while disabling preemption for the corresponding duration using `preempt_disable` [32].

The testing framework consists of described proxy block device, along with some parametrized scripts that run compression on files of the dataset using this device, saving statistics as intermediate results along the way. Essentially, for each compression method we write the file to the proxy device, read it from device into some temporary file, save compression statistics to a separate file and reset the statistics afterwards. All the block I/O is performed using the `dd` utility [33] with `direct` flag specified. The complete testing framework, including the described proxy block device and additional scripts, is available on GitHub [29].

5.3 Experimental setup

The following datasets were used for the experiment:

- The Canterbury corpus [34] – 2.7 MiB;
- The Calgary corpus [34] – 3.2 MiB;
- The Large corpus [34] – 10.6 MiB;
- Silesia corpus [35] – 202.1 MiB.

All of the corpora above are designed specifically for testing data compression methods, and are regarded as classical datasets in the field. Combined, these datasets cover a wide range of file types and extensions. For testing, each collection of files was concatenated into a single large one, resulting in one file per dataset.

The test platform used for the experiment has the following specifications.

Hardware:

- CPU: Intel(R) Xeon(R) Gold 6338 @ 2.00 GHz (32 cores/socket, 2 sockets, 2 threads per core, total 128 CPUs);
- RAM: 496.6 GiB available (503.5 GiB total) – DDR4 Synchronous Registered (Buffered) 3200 MHz, Samsung M393A4K40EB3-CWE 32 GiB DIMMs;
- Storage: SAMSUNG MZWLJ1T9HBJR-00007 1.92 TB NVMe SSD (NVMe version 1.3, firmware EPK9GB5Q);
- NUMA: 2 nodes (node0: CPUs 0–31,64–95; node1: CPUs 32–63,96–127);
- Caches: L1d 3 MiB, L1i 2 MiB, L2 80 MiB, L3 96 MiB;
- Hrtimers: 1 nsecs resolution.

Software:

- OS: Linux kernel 6.18.5-200.fc43.x86_64 (Fedora 43);
- gcc version: 15.2.1 20260123 (Red Hat 15.2.1-7);
- dd version: coreutils 9.7.

A single 57.3 GB NVMe namespace was selected as the underlying block device. For each of four compression methods chosen for the experiment, the test, consisting of a single write-read sequence,

was run 15 times. Block size for dd was set to 1 MiB, as a size large enough to be a viable use case for our extension, as well as size that perfectly fits 256 single-page bvecs. The extended compression library was compiled using the -O3 optimization level of gcc [36].

5.4 Evaluation results

Here we give answers to the research questions formulated in Section 1. For compression ratio and memory usage, the relative error of obtained results is negligible. For the mean values of compression throughput, the relative error averages to 0.66%, peaking at 4.47%.

RQ1. What approaches achieve the best compression ratio? Fig. 6 shows the obtained compression ratio for all chosen approaches. As expected, compression on contiguous buffers reduces data size the best. Compressing each bvec individually (Vectorized) has the lowest compression ratio. The use of the Streaming API makes data compression slightly stronger, giving 9–17% boost in compression ratio compared to the Vectorized approach. The extended algorithm achieves the same best compression ratio as the Contiguous approach.

RQ2. How much memory does LZ4E save compared to the contiguous approach? Fig. 7 summarizes the memory usage of all chosen approaches. The extended algorithm reduces memory usage by 49.47% compared to the Contiguous approach. This saving comes from eliminating the need to allocate a separate contiguous source buffer and copy all request data into it. In contrast, the Vectorized and Streaming approaches do not reduce memory usage further, as they already avoid the additional source buffer allocation.

RQ3. What is the throughput of LZ4E algorithm and how does it compare to the throughput of other compression approaches? Fig. 8 presents the compression throughput for all chosen approaches. The extended algorithm suffers a significant throughput degradation, being 5–6 times slower than both the Contiguous and Vectorized approaches, and 2–3 times slower than the Streaming approach. The Contiguous approach, despite requiring data copying, does not experience a large throughput penalty from the copy operation itself, as shown by the “copy loss” graph legend entry.

Additionally, we measured performance of LZ4E with optimization of in-advance bvec mapping, enabled by macro LZ4E_PREMAP. This optimization, however, provides only an insignificant throughput increase within 1.43%. The trade-off is the increased size of the required working memory, which grows by 4 KiB, or 23.53%.

As a result of the experiment, our extended LZ4 compression provides either stronger compression or significantly lower memory usage, as compared to the rest of the methods selected. The compression speed, however, becomes substantially lower. The evaluation results for all three metrics are summarized in Table 1.

Table 1. Summary of evaluation results for all compression methods. The values are per-dataset means across four datasets that were used. For memory usage, the values show required memory for a single I/O, which is constant across datasets. For other metrics, ranges from minimum to maximum values are shown. Throughput is reported in decimal units (MB/s), while memory usage is in binary units (KiB).

Compression method	Compression ratio	Compression throughput, MB/s	Memory usage, KiB per I/O
Contiguous	1.905–2.099	385.7–515.7	2068.1
Vectorized	1.474–1.767	362.0–516.4	1048.0
Streaming	1.718–1.930	209.0–264.6	1048.0
Extended	1.905–2.099	72.8–88.2	1045.0

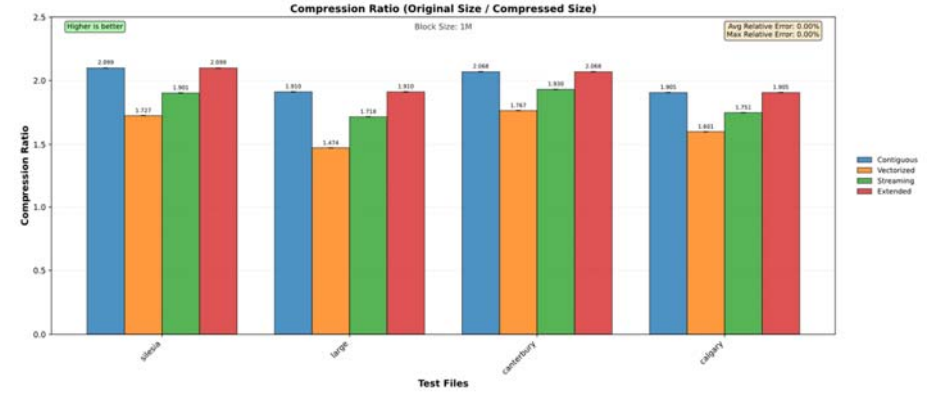


Fig. 6. Evaluated compression ratio of methods selected for comparison.

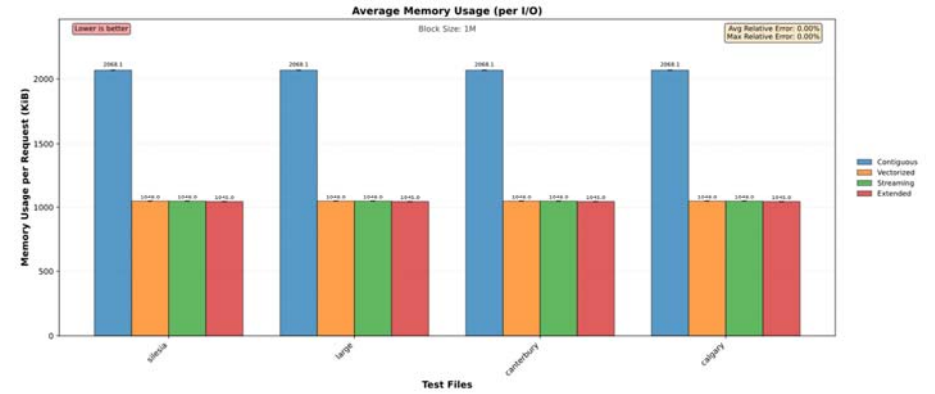


Fig. 7. Evaluated memory usage of methods selected for comparison.

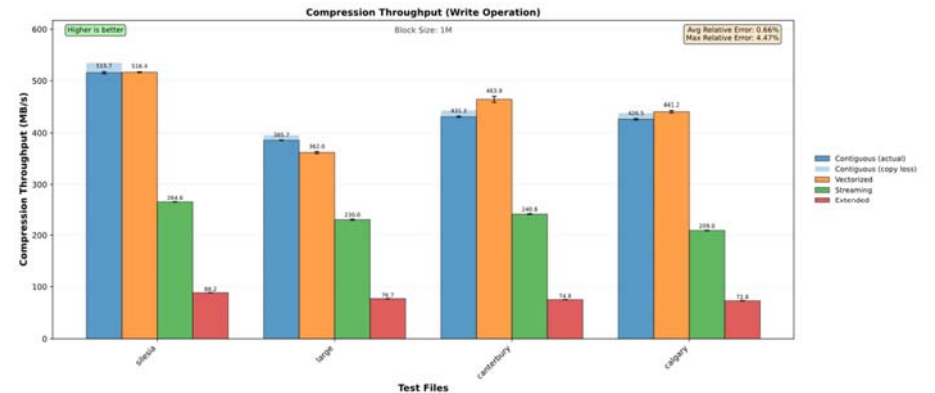


Fig. 8. Evaluated compression throughput of methods selected for comparison.

5.5 Performance analysis

The suspected causes for such significant throughput degradation, encountered after extension for scatter-gather buffer support, are as follows.

- 1) Replacing simple pointer arithmetic with iterator handling introduces substantially more complexity throughout the algorithm.
- 2) Simple read-write-copy operations are also affected, requiring considerably more branches for careful memory handling. This makes a substantial difference specifically during match searching, as a large portion of the algorithm's runtime is devoted to short reads.
- 3) Dictionary handling becomes more complex, as converting positions to hash table addresses and back requires considerably more calculations. This is particularly significant because match searching, which accounts for the majority of the runtime, heavily relies on hash table lookups.
- 4) Locality is also worse, since the memory in which matches are searched is more scattered than in the other methods. This may lead to an increased number of cache misses.

To investigate these hypotheses, we collected profiling metrics using the perf utility [37]. The following perf events were examined:

- instructions, cycles – to examine computational complexity and calculate instructions per cycle (IPC);
- branches, branch-misses – to examine branch prediction;
- cache-references, cache-misses – to examine last-level cache (LLC) performance;
- L1-dcache-load-misses – to examine L1 cache performance.

To evaluate branch prediction, we measured the total number of branch instructions (BRA) and the branch miss rate (BMR). Similarly, for LLC performance, we used the total number of LLC references (CRE) and the LLC miss rate (CMR). For evaluating L1 cache performance, only the number of L1 data cache misses (L1M) was utilized. The total number of instructions (INS) served to evaluate computational complexity, while IPC was applied to diagnose CPU pipeline stalls. For all raw metrics (BRA, CRE, L1M, INS), the Contiguous approach served as the baseline. The derived metrics (BMR, CMR, IPC) are reported as absolute values, as they already represent percentages or ratios.

For counting events, the perf stat command was used. This command was called over dd invocation within the same testing framework, described in Subsection 5.2. To collect the metrics, we performed an additional experiment consisting of 15 test runs using the same experimental setup, described in Subsection 5.3.

The summary of profiling results is presented in Table 2. Average relative errors for individual metrics, except CRE and CMR, are in 0.05–1.10% range, and the maximum relative errors are in 0.15–4.79% range. For CRE and CMR combined, relative error averages to 7.73%, peaking at 72.01%. Detailed profiling results are presented in Figures 9–12. Specifically: branch metrics (BRA and BMR) are shown in Fig. 9, LLC metrics (CRE and CMR) in Fig. 10, instruction count (INS) in Fig. 11, and IPC in Fig. 12.

Addressing the hypotheses formulated earlier, the following conclusions about LZ4E performance can be drawn from the profiling results.

- 1) The total number of branches increased by 6.4 times compared to the Contiguous approach, correlating with the observed throughput penalty. However, the absolute number of branch misses did not grow proportionally to the number of branches, increasing by only 15.04%. The proposed method achieves a 5.5 times lower miss rate than the Contiguous approach, indicating that branch prediction is not the primary concern.

Table 2. Summary of profiling results for all compression methods. The values are averages of per-dataset means across four datasets that were used. For metrics BRA, CRE, L1M, and INS, the values are reported relative to the Contiguous approach (baseline). For BMR, CMR, and IPC, the absolute values are presented.

Compression method	BRA	BMR	CRE	CMR	L1M	INS	IPC
Contiguous	1.000	4.45%	1.000	10.25%	1.000	1.000	1.621
Vectorized	1.085	3.80%	0.892	10.45%	0.646	1.113	1.790
Streaming	1.497	9.25%	1.105	14.00%	0.784	1.352	1.279
Extended	6.399	0.80%	1.029	11.33%	1.059	9.952	3.719

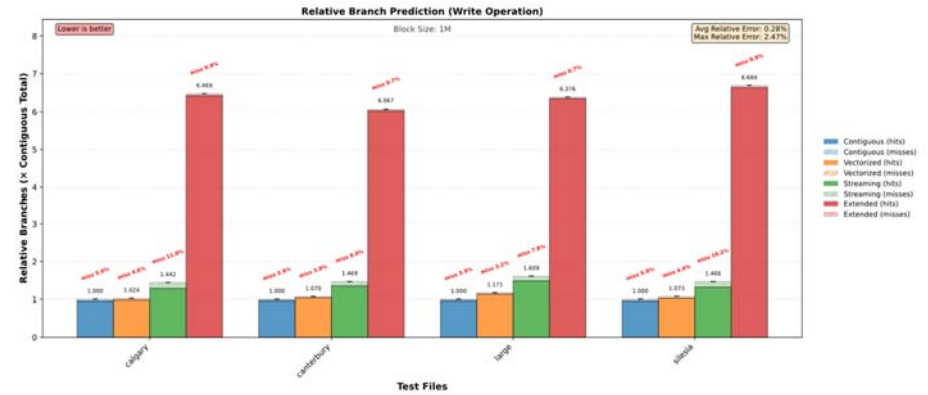


Fig. 9. Evaluated branch prediction metrics (BRA, BMR) for all compression methods.

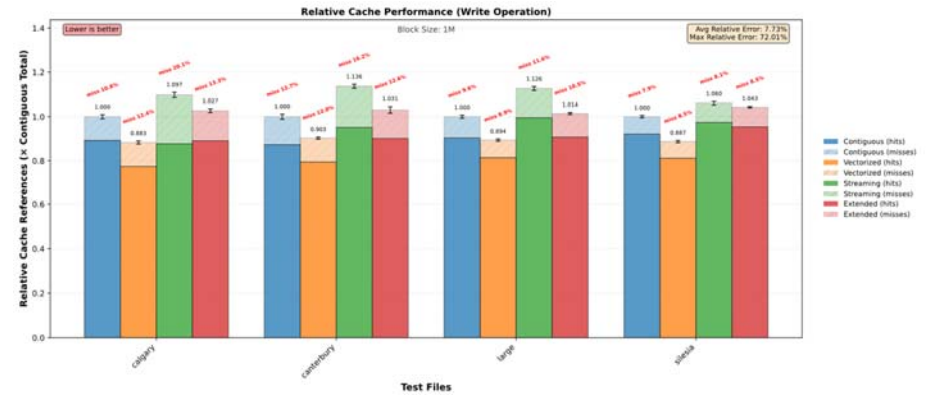


Fig. 10. Evaluated last-level cache performance metrics (CRE, CMR) for all compression methods.

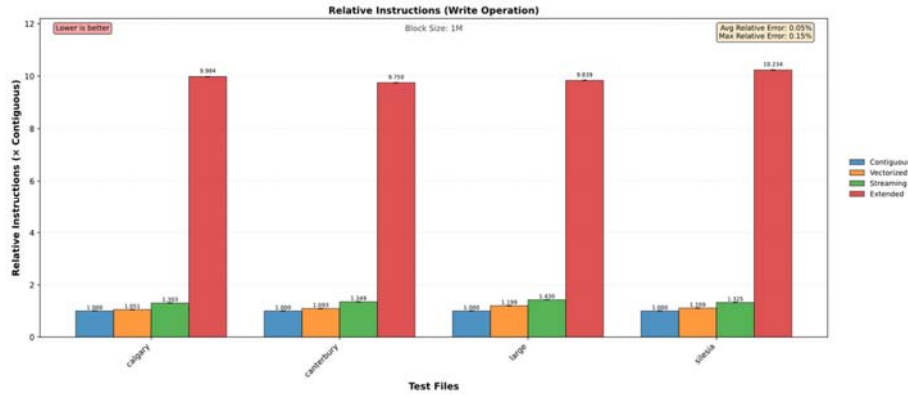


Fig. 11. Evaluated instruction count (INS) for all compression methods.

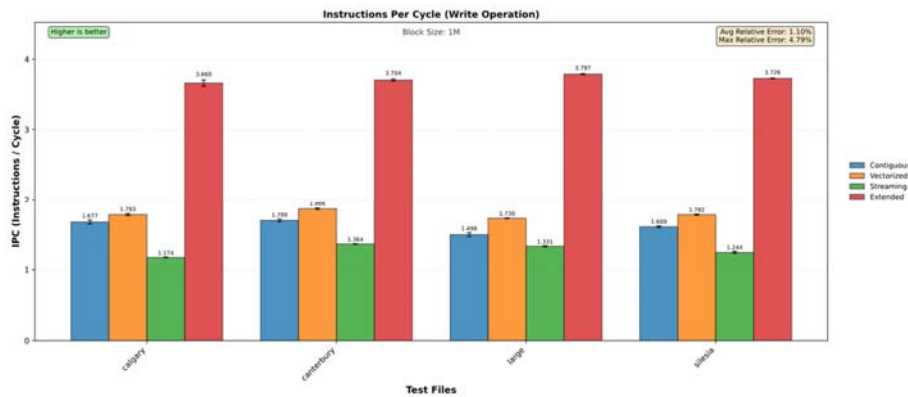


Fig. 12. Evaluated instructions per cycle (IPC) for all compression methods.

- 2) The number of LLC references increased by 2.90%, while LLC misses and L1 data cache misses increased by 13.74% and 5.90%, respectively. This supports the claim of worse cache locality, but the increase in cache misses is not substantial enough to explain the throughput drop. Therefore, cache performance is not the bottleneck.
- 3) The total number of executed instructions increased substantially – by almost 10 times. This result supports the hypothesis of increased computational complexity and is the most likely cause of the observed throughput degradation.
- 4) The proposed approach achieves 2.3 times higher IPC compared to the Contiguous method. Combined with the instruction count results, this indicates that while the extended algorithm executes approximately 10 times more instructions, it executes them with 2.3 times better efficiency.

To summarize our performance analysis, we have identified that the performance issues of the extended algorithm stem from significantly higher computational complexity. However, the specific bottleneck of the algorithm, if one exists, remains to be determined.

6. Future work

This work can be extended in several directions. First, the throughput issue must be addressed by identifying the exact bottleneck and simplifying the required calculations. This may involve designing a new algorithm specifically adapted for the new data processing approach, similarly to what was done in [18]. Second, we plan to implement the extended LZ4 decompression, which would be straightforward using the proposed design of the extension. Finally, our approach can be extended to other dictionary-based compression algorithms commonly used in the block layer of the Linux kernel, such as Zstandard [12, 17], which is a more heavyweight algorithm as opposed to LZ4. It would be valuable to investigate how this approach affects performance for such algorithms.

7. Conclusion

As a result of this work, we propose an extension of LZ4 algorithm that can directly handle data of scatter-gather I/Os in the Linux kernel block layer, not requiring additional source buffer allocation and consequent data copying while providing the exact same compression ratio, as when running standard compression on data as a single contiguous chunk. We also compared its performance to the case of running standard compression individually for each of the vectors of a scatter-gather buffer, using both the regular and Streaming APIs. To summarize, our modification provides a major memory usage reduction compared to the contiguous case, and a significant increase in compression ratio as compared to vectorized cases. However, this comes at a cost of drastic compression throughput loss, reducing speed up to 5–6 times. Presented profiling results show that the throughput drop is caused by a substantial increase in computational complexity – the number of executed instructions increases by almost 10 times. The exact bottleneck responsible for this instruction increase remains to be identified. Eliminating the throughput issue is necessary to make our solution a viable choice for online compression. As an alternative, observed characteristics of the proposed modification show that it is better suited for offline compression, which runs outside of the critical I/O path and prioritizes compression ratio over throughput. Reduced memory usage further allows to reduce the overall memory pressure, which is valuable for high-load systems. Additionally, our solution is well-suited for memory-constrained environments and can serve as a fallback when the faster copy-based approach fails due to insufficient memory.

References

- [1] Gupta A., Bansal A., Khanduja V. Modern lossless compression techniques: Review, comparison and analysis. In 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, February 2017, pp. 1-8, DOI: 10.1109/ICECCT.2017.8117850.
- [2] Salomon D. Data Compression: The Complete Reference, 4 ed., G. Motta and D. Bryant, Eds., New York, NY: Springer Science+Business Media, LLC, 2007, 1093 p.
- [3] dm-vdo. Available at: <https://docs.kernel.org/admin-guide/device-mapper/vdo.html>, accessed 27.09.2026.
- [4] ZFS: Workload Tuning. Available at: <https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/Workload%20Tuning.html>, accessed 27.09.2026.
- [5] zram: Compressed RAM-based block devices. Available at: <https://docs.kernel.org/admin-guide/blockdev/zram.html>, accessed 27.09.2026.
- [6] Klonatos Y., Makatos T., Marazakis M., Flouris M., Bilas A. Transparent Online Storage Compression at the Block-Level. ACM Transactions on Storage, May 2012, vol. 8, pp. 1-33, DOI: 10.1145/2180905.2180906.
- [7] Love R. The Block I/O Layer. In Linux Kernel Development, 3 ed., Upper Saddle River, NJ: Addison-Wesley, 2010, pp. 289-304.
- [8] Yao X., Li Q., Lin K., Gan X., Zhang J., Gao C., Shen Z., Xu Q., Yang C., Xue J. Extremely-Compressed SSDs with I/O Behavior Prediction. ACM Transactions on Storage, August 2024, vol. 20, no. 4, pp. 1-38, DOI: 10.1145/3677044.
- [9] Yu Y.J., Shin D.I., Shin W., Song N.Y., Choi J.W., Kim H.S., Eom H., Yeom H.Y. Optimizing the Block I/O Subsystem for Fast Storage Devices. ACM Transactions on Computer Systems, June 2014, vol. 32, pp. 1-48, DOI: 10.1145/2619092.

- [10]. Salomon D. Dictionary Methods. In *Data Compression: The Complete Reference*, 4 ed., G. a. B. D. Motta, Ed., New York, NY: Springer Science+Business Media, LLC, 2007, pp. 171-261.
- [11]. Izbench Compression Benchmark. Available at: <https://morotti.github.io/Izbench-web/>, accessed 27.09.2026.
- [12]. Maulidina A.P., Wijaya R.A., Mazel K., Astriani M.S. Comparative Study of Data Compression Algorithms: Zstandard, zlib & LZ4. In *Science, Engineering Management and Information Technology*, Ankara, September 2023, DOI: 10.1007/978-3-031-72284-4.
- [13]. Matsuoka T. LZ4 Streaming API Basics. Available at: https://github.com/lz4/lz4/blob/dev/examples/streaming_api_basics.md, accessed 27.09.2026.
- [14]. Pointer Arithmetic. Available at: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Pointer-Arithmetic.html, accessed 27.09.2026.
- [15]. LZ4 API in the Linux kernel. Available at: <https://elixir.bootlin.com/linux/v6.19.12/source/include/linux/lz4.h>, accessed 27.09.2026.
- [16]. Overstreet K. Immutable biovecs and biovec iterators. Available at: <https://docs.kernel.org/block/biovecs.html>, accessed 27.09.2026.
- [17]. Zstandard API in the Linux kernel. Available at: <https://elixir.bootlin.com/linux/v6.19.12/source/include/linux/zstd.h>, accessed 27.09.2026.
- [18]. Warnke B. Page-Based Compression in the Linux Kernel, Bachelor's Thesis, University of Hamburg, Hamburg, August 2017, 59 p. Available at: https://wr.informatik.uni-hamburg.de/media/research/theses/benjamin_warnke_page_based_compression_in_the_linux_kernel.pdf, accessed 27.09.2026.
- [19]. Makatos T., Klonatos Y., Marazakis M., Flouris M., Bilas A. Using transparent compression to improve SSD-based I/O caches. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Paris, April 2010, pp. 1-14, DOI: 10.1145/1755913.1755915.
- [20]. Makatos T., Klonatos Y., Marazakis M., Flouris M., Bilas A. ZBD: Using Transparent Compression at the Block Level to Increase Storage Space Efficiency. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, Incline Village, NV, May 2010, pp. 61-70, DOI: 10.1109/SNAPI.2010.15.
- [21]. Collet Y. LZ4 Block Format Description. Available at: https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md, accessed 27.09.2026.
- [22]. Implementation of LZ4 compression in the Linux kernel. Available at: https://elixir.bootlin.com/linux/v6.19.12/source/lib/lz4/lz4_compress.c, accessed 27.09.2026.
- [23]. Love R., Memory Management. In *Linux Kernel Development*, 3 ed., Upper Saddle River, NJ: Addison-Wesley, 2010, pp. 231-260.
- [24]. Zijlstra P. High Memory Handling. Available at: <https://docs.kernel.org/mm/highmem.html>, accessed 27.09.2026.
- [25]. Lei M. block: support multi-page bvec. Available at: <https://lwn.net/Articles/779903/>, accessed 27.09.2026.
- [26]. Memory Management APIs. Available at: <https://www.kernel.org/doc/html/v6.19/core-api/mm-api.html>, accessed 27.09.2026.
- [27]. API for the Linux kernel block I/O vectors. Available at: <https://elixir.bootlin.com/linux/v6.19.12/source/include/linux/bvec.h>, accessed 27.09.2026.
- [28]. BIO_MAX_VECS identifier. Available at: https://elixir.bootlin.com/linux/v6.19.12/A/ident/BIO_MAX_VECS, accessed 27.09.2026.
- [29]. Extended LZ4 library with testing framework. Available: <https://github.com/ItIsMrLaG/lz4-sgori>, accessed 27.09.2026.
- [30]. Mochel P., Murphy M. sysfs – The filesystem for exporting kernel objects. Available at: <https://docs.kernel.org/filesystems/sysfs.html>, accessed 27.09.2026.
- [31]. ktime accessors. Available at: <https://docs.kernel.org/core-api/timekeeping.html>, accessed 27.09.2026.
- [32]. Love R. Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe. Available at: <https://www.kernel.org/doc/Documentation/preempt-locking.txt>, accessed 27.09.2026.
- [33]. dd: Convert and copy a file. Available at: https://www.gnu.org/savannah-checkouts/gnu/coreutils/manual/html_node/dd-invocation.html, accessed 27.09.2026.
- [34]. The Canterbury corpus: Descriptions of the corpora. Available at: <https://corpus.canterbury.ac.nz/descriptions/>, accessed 27.09.2026.
- [35]. Silesia compression corpus. Available at: <https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>, accessed 27.09.2026.

- [36]. gcc: Options That Control Optimization. Available at: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>, accessed 27.09.2026.
- [37]. perf: Linux profiling with performance counters. Perf Wiki. Available at: <https://perfwiki.github.io/main/>, accessed 27.09.2026.
- [38]. Love R. *Linux Kernel Development*, 3 ed., Upper Saddle River, NJ: Addison-Wesley, 2010, 440 p.

Информация об авторах / Information about authors

Александр Сергеевич БУГАЕВ является студентом четвертого курса бакалавриата Математико-механического факультета Санкт-Петербургского государственного университета, направление «Технологии программирования», младшим инженером-разработчиком ООО «Швачер». Его научные интересы включают сжатие данных, операционные системы, системы хранения данных.

Alexander Sergeevich BUGAEV – fourth-year undergraduate student at the Faculty of Mathematics and Mechanics, Saint Petersburg State University, majoring in Programming Technologies; junior software engineer at Shvacher LLC. His research interests include data compression, operating systems, storage systems.

Георгий Константинович СИЧКАР является выпускником Математико-механического факультета Санкт-Петербургского государственного университета, направление «Программная инженерия», младшим инженером-программистом ООО «Швачер». Его научные интересы включают сжатие данных, системы хранения данных, операционные системы, виртуализацию.

Georgy Konstantinovich SICHKAR – graduate of the Faculty of Mathematics and Mechanics, Saint Petersburg State University, majoring in Software Engineering; junior software engineer at Shvacher LLC. His research interests include data compression, storage systems, operating systems, virtualization.

Анна Игоревна ВАСЕНИНА является главным инженером-программистом ООО «Швачер», инженером-исследователем лаборатории технологий программирования инфраструктурных решений ("Лаборатория YADRO") Санкт-Петербургского государственного университета. Её научные интересы включают операционные системы, системы хранения данных, файловые системы и менеджеры томов.

Anna Igorevna VASENINA – principal software engineer at Shvacher LLC, research engineer at Laboratory of Programming Technologies for Infrastructure Solutions ("YADRO Laboratory") of Saint Petersburg State University. Her research interests include operating systems, storage systems, file systems and volume managers.