Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Seitenbasierte Komprimierung im Linux-Kernel - Page-Based Compression in the Linux Kernel

vorgelegt von

Benjamin Warnke

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6676867
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Anna Fuchs
Betreuer:	Dr. Michael Kuhn
Betreuerin:	Anna Fuchs

Hamburg, 18.08.2017

Abstract

Since computers exist, the performance of processors increases faster than the throughput of networks and permanent storage. As a result, the performance bottleneck is the throughput between the processor the other system components. To improve overall cluster performance, the processor can be used to compress the data before sending it to a slower component.

In order to apply the compression efficiently to all kind of data intensive applications, compression can be performed in an underlying file system which is widely used in clusters. One of these file systems is Lustre.

LZ4 is focused on speed and is currently one of the fastest lossless compression algorithms. The algorithm can compress up to 4.51 GB/s while decompression reaches about 9.14 GB/s. All available *LZ4* implementations use continuous buffers. Since Lustre is a Linux kernel module, the memory is accessed via page arrays. To be able to use *LZ4* the data has to be copied into a continuous buffer and back later. The aim of this thesis to improve the throughput of the compression by reducing the memory utilization and the number of copying processes. Therefore a modified version of *LZ4* is introduced, which works directly on page-based buffers. After the implementation of the *LZ4* algorithm with page-based buffers, it became clear that the performance was not sufficient. Therefore, a new algorithm called *BeWalgo* is designed that doubles the compression throughput when page arrays are used as buffers. The drawback of the *BeWalgo* algorithm is that the compression ratio may be lower in comparison to *LZ4*.

Contents

1. Introduction

Processor performance increases by about 60% per year; At the same time, the speed of the memory increases by only 10% [Car02, p. 1] [nvi17]. The performance gap between the processor and the other system components increases exponentially. As a result, the processors are currently wasting a lot of time by waiting for the slow components. To reduce the waiting time, applications can overlap the time-consuming read and write (r/w) operations with additional calculations. But overlapping is not always possible and requires additional code optimizations for each application.

Nodes in clusters and supercomputers are connected with very fast network connections with large data rates and low latencies. InfiniBand interconnects are often used there [Mel15]. If the data are compressed more quickly and with a better compression rate, it would be possible to transfer more data over the same network. Since compression takes time, the permanent storage of the data is delayed. Normally large simulations do not write data for permanent storage and read it back immediately. Most often the data is written once and later read only for a restart or for evaluation. The restart of a simulation normally occurs only once every few hours or days. Post-processing of the generated data is usually done by another application, which does not have the data in the cache anyway. In both cases, the increased delay is not important because both events are relatively rare and the latency is added only once to the total run time.

Clusters and supercomputers require large storage systems. These large storage systems are mostly, but not exclusively, based on hard disk drives (HDDs) because of the relatively low price per volume. HDDs can write a few hundred megabytes per second, which is much slower than the random access memory (RAM). When the data stream is sent compressed to the storage nodes, the reduced data size results in a reduced waiting time for the calculation nodes because the required data is available more quickly. When compression occurs in the file system, each application on the system that stores a lot of data gets additional performance without the need for individual compression algorithms.

No user application that produces a lot of data must worry about compression as performance improvement when the file system transparently compresses all files. There are a lot of file systems, but there are only a few that are widely used in clusters and supercomputers [Int14]. One of these file systems is Lustre [Lus17]. It is possible to add compression to Lustre because it is open source. Lustre is a distributed file system, which implies that multiple computation nodes can simultaneously write to multiple dedicated storage nodes. These storage nodes write the data to their attached storage devices. The Lustre file system is completely written in C and runs in the Linux kernel.

Adding compression support to a file system requires several components. When different compression algorithms are available, the file system must be able to choose the best compression algorithm for the current data. For later decompression, the file system must store the size of compressed and uncompressed data. The work on all components needed to add compression to the file system is already in progress while this thesis is being written [Fuc16].

Benchmarks show that *LZ4* [Col17] is currently one of the fastest compression algorithms. *LZ4* is a lossless compression algorithm with a focus on speed. The algorithm can compress data with up to 4.51 GB/s while the decompressor reaches 9.14 GB/s. To compare the speed of different compression algorithms, the function *memcpy* is normally used as a reference. The currently available *LZ4* versions require at least virtual continuous buffers for input and output. The allocation of virtual continuous memory requires the Linux kernel to modify the page tables, resulting in an overhead. On the other hand, the allocation of large physically continuous buffers within the kernel leads to serious challenges because the kernel can not guarantee that the requested memory is available. To solve this problem, a modified *LZ4* algorithm is implemented which operates directly on page-based buffers.

Benchmarks in Chapter5.3 show that the *LZ4* compression speed on page-based buffers is poor because the throughput reaches only 2.99 GB/s while achieving the same compression ratio as the continuous buffer version. Therefore, this thesis presents a new compression algorithm called *BeWalgo* with focus on speed in the context of page-based buffers. With the page-alignment of r/w operations, it is possible to double the achieved throughput in comparison to *LZ4*. Depending on the structure of the data to be compressed, the *BeWalgo* algorithm can lead to poorer compression ratios compared to *LZ4*. For example, if text is to be compressed, *BeWalgo* achieves a compression ratio of 1.9 while *LZ4* reduces the same data with a ratio of 4.3. When scientific data are to be compressed, both algorithms achieve nearly the same compression ratio as shown in Chapter 5.

1.1. Structure of this Thesis

Chapter 2 describes the basics needed for fast compression and programming in the Linux kernel. Chapter 3 describes the design of the various algorithms, including some problems that arose during the creation process. Section 3.2 explains how the various algorithms have been implemented. In addition, the key elements are displayed which make the code as fast as possible. Chapter 4 shows some compression algorithms and helper algorithms. These algorithms could be useful after they have been adapted for the Linux kernel because they either improve the compression ratio or represent completely new algorithms. Chapter 5 evaluates the the differences between the continuous and page buffer based algorithms as well as the effect of different data-sets to the compression ratio and speed.

2. State of the Art

This chapter shows the basic concepts of the compression algorithm *LZ4* as well as some differences between user space and kernel space code.

2.1. Compression

Compression algorithms are designed to reduce the size of data by converting data to another format. This is only possible if the data contains repeating patterns. There are two large classes of compression algorithms: Lossless and lossy [Seb03]. Lossless algorithms guarantee that all reconstructed data exactly match the corresponding original data. Lossy algorithms may loose some information to reduce the size even more. Lossy algorithms are not further explained, since these should not be used in the context of file systems. For compression algorithms preallocated input and output buffers are required. Most algorithms, including *LZ4*, require these buffers to be at least virtually continuous. Currently, *LZ4* is the fastest compression algorithm for continuous buffers. To modify the algorithm and accept page-based buffers, it is necessary to analyze *LZ4* to understand how the algorithm works and why it is so fast.

2.1.1. *LZ4*

The *LZ4* algorithm belongs to the family of the LZ* algorithms. In LZ* the uncompressed data are divided into literals and matches. A literal is a group of bytes that are first seen by the compressor and can not be compressed. The literal is stored in the compressed data stream as it is. A match is a group of bytes that the compressor has already seen before. To find matches the compressor uses a sliding window. The compressor therefore only needs to store the information where these bytes, relative to the current position, have already been seen. The matches are the interesting components because they can be used to reduce the data size. Matches can be optimized in count and length.

Figure 2.1 visualizes the pattern of the *LZ4* compressed data stream. This pattern is repeated until all the data are encoded. The last pattern ends directly after a literal.

Token		Literal length	Literals	Offset	Match length
Literal length	Match length				
4 bits	4 bits	$[0, N]$ bytes	$[0, L]$ bytes	2 byte	$[0, M]$ bytes

Figure 2.1.: *LZ4* pattern of compressed data

2.1.1.1. Length Encoding

To store the information how long a literal or match is *LZ4* needs to encode lengths. To store this lengths efficiently *LZ4* can not use the default two's complement. Therefore, Figure 2.2 shows the decoding of such an *LZ4* length.

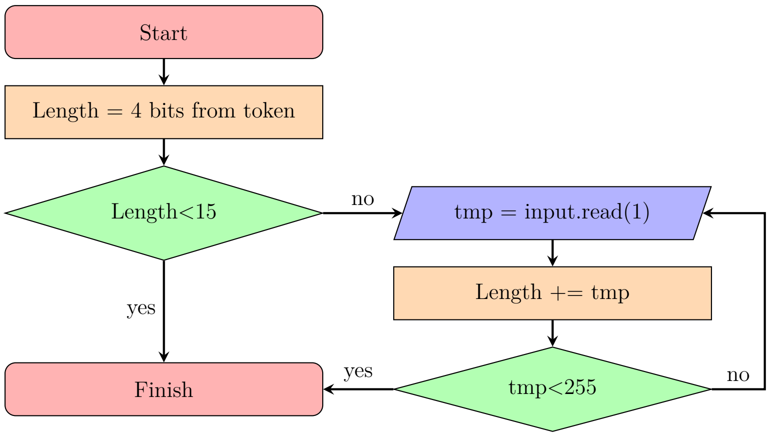
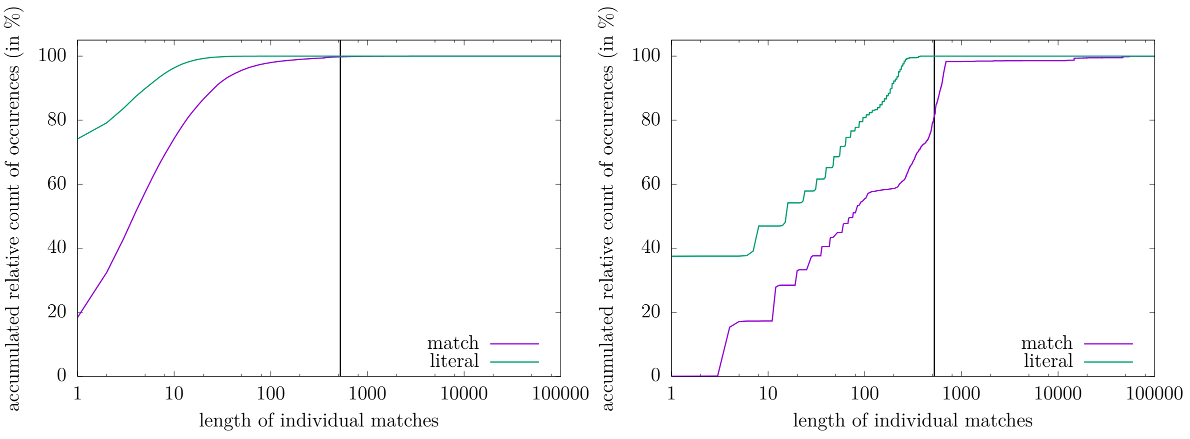


Figure 2.2.: *LZ4* decode length

The first component of an encoded length is stored in four bits, which are part of the token. If the four bits represent a value less than 15, then the length is decoded. Otherwise, the following bytes must be read and summed up until a byte with a value less than 255 is read. When a match is decoded, it is necessary to add the value min-match to the decoded length.

To understand why *LZ4* uses this length encoding, it is best to show analyzes of different datasets. These datasets are explained in detail in Chapter 5.



(a) 'text' dataset (b) 'hdf5' dataset

Figure 2.3.: Length distribution of literals and matches with the *LZ4* compressor

Figure 2.3a shows the analysis results of the 'text' dataset. An analysis of the 'silesia corpus' dataset resembles the 'text' dataset and can therefore be found in the appendices. Figure 2.3 shows the distribution of literal-lengths and match-lengths. The vertical line indicates the point at which more than two bytes are required to encode a length with the *LZ4* encoding.

If the compression ratio is very high as in the 'hdf5' dataset, then only 80% of the data can be encoded with matches of at most two bytes, as shown in Figure 2.3b. Otherwise, as in the 'text' dataset, almost all matches and literals can be encoded in at most two bytes. Even if only 80% of the encoded lengths are at most two bytes long, this is still the majority. If the length encoding would use groups of two bytes or more, then the compressed data stream would require more space.

2.1.1.2. Search for a Match

The *LZ4* compressor spends between 91% and 98% of the total time only to find matches. The process of finding matches is shown in Figure 2.4.

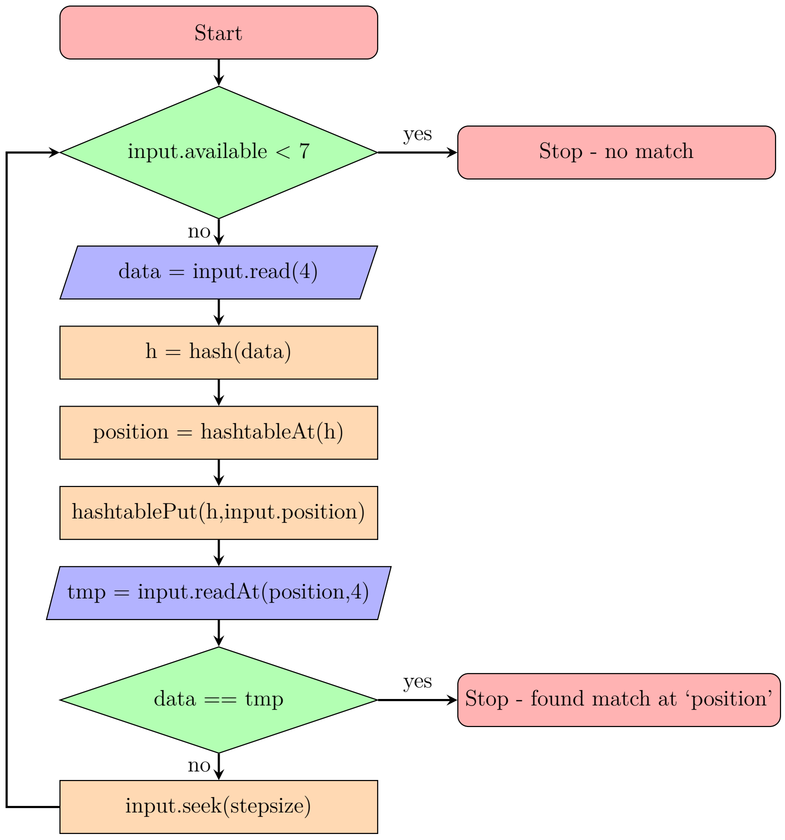

Start
input.available < 7
yes
Stop - no match
no
data = input.read(4)
h = hash(data)
position = hashtableAt(h)
hashtablePut(h,input.position)
tmp = input.readAt(position,4)
data == tmp
yes
Stop - found match at 'position'
no
input.seek(stepsize)

Figure 2.4.: *LZ4* search for a match

The *LZ4* compressor requires a hash table in order to be able to locate matches in the data. When searching for a match, the compressor must ensure that no data is touched beyond the input buffer. In addition, the compressed data must end with a literal.

Therefore, the process of searching for a match is aborted when only a few literals are left in the input buffer. If there is enough data, the current data is hashed and used to find a location in the hash table. If the data at the obtained position matches the data at the current position, then the compressor has found a match. Otherwise, the compressor must search further. In order to improve the speed when the data is poorly compressible, the step size to be skipped can be increased after each non-coincidence. In order to influence the compression speed, the compression function accepts an acceleration factor as a parameter. This parameter is used to change the step size to gain more speed at the expense of a poorer compression ratio.

2.1.1.3. Compression

The *LZ4* compressor must convert the original data into the compressed data format described in Figure 2.1. Figure 2.5 shows the general principle how the *LZ4* compressor works.

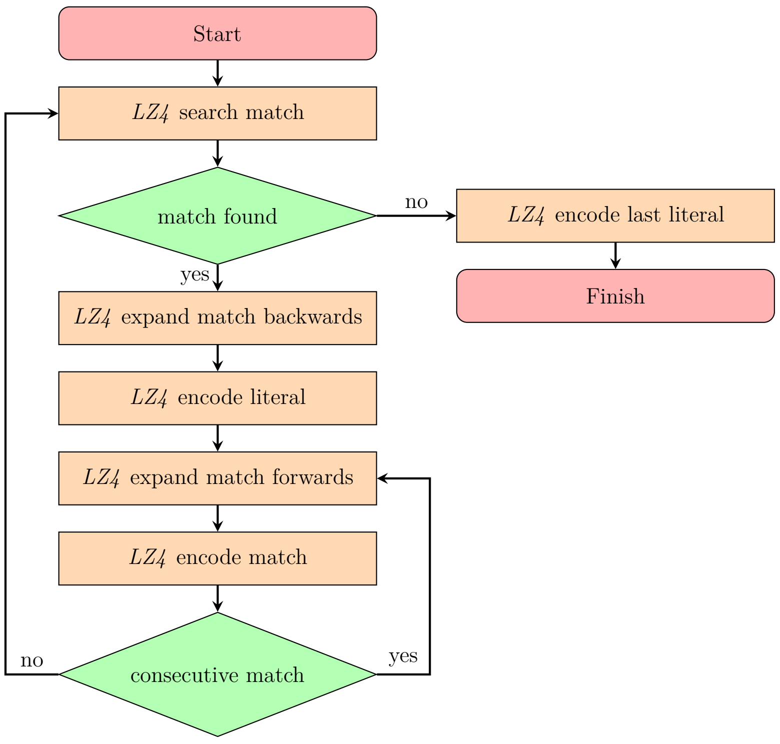


Figure 2.5.: *LZ4* compression overview

Now, the detailed description for the compression process follows:

- **search for match**: The LZ4 compression process starts with searching for a match.

- **match found**: If there is no match or no other data is present, the last literal is encoded. If there is a match, the algorithm tries to expand the match as far as possible to the beginning.

- ***LZ4* encode literal**: If it is known where the next match begins, the data between the last encoded match and the current match must be stored as a literal. It is possible that the literal-length is zero.

- ***LZ4* encode last literal**: It is necessary that the last pattern of the compressed data ends with a literal because there is no guarantee that there is any match at all. Since matches must be at least four bytes long, they can not be encoded at the end. In addition, the performance can be increased by reducing the branches. The token of the last pattern is the only one in which no implicit min-match length is encoded.

- ***LZ4* expand match to end**: After the literal is encoded, the compressor expands the match as far as possible. During expansion, the compressor must ensure that it does not expand beyond the limits of the input buffer.

- ***LZ4* encode match**: If the position and the length of the match are known, the compressor can append this information to the compressed data stream.

- **consecutive match**: To optimize the speed, *LZ4* can skip a large portion of its code if there are successive matches. To skip large parts of the code, GOTOs are used. While GOTOs are rarely used in the user space, they are often used in Linux kernel code.

2.1.1.4. Decompression

The decompression of a compressed *LZ4* data stream is simple and therefore particularly fast. To decompress a stream in *LZ4* format, it is sufficient to read the bytes from the input stream only once. Figure 2.6 shows the general method of decompression.

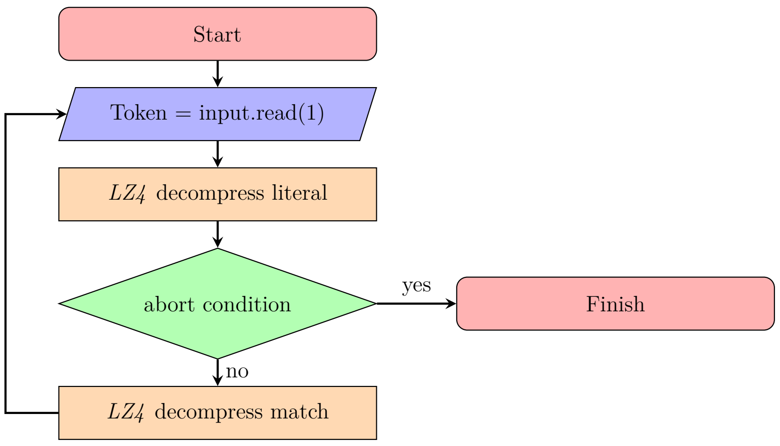


Figure 2.6.: *LZ4* decompression overview

The first byte in the stream is a token. This token contains the first part of the lengths for the following literal as well as the following match. After the token has been read, the decompressor continues to decompress a literal.

To decompress a literal, it is necessary to first decode the literal-length, as described in Subsubsection 2.1.1.1. Now that the literal-length is known, the literal from the input buffer is copied to the output buffer. The only position at which an *LZ4* decompressor can successfully terminate is directly after decompressing a literal. If the data is not fully recovered, the decompressor continues to decompress a match.

Decoding a match is similar to decoding a literal. However, since matches contain a reference to previously decoded data, it is necessary to first decode this reference by reading an offset. With this offset and the current position in the decoded data stream, it is possible to calculate the origin of the new data. According to the origin of the data, the literal-length is decoded. In order to reduce the required storage space for the compressed data by the *LZ4* algorithm, the match-length must be increased by the min-match value. After the position and the length are known, the match can be copied to the current position in the target buffer. Since this copying process copies between two points in the same buffer, it is possible that the source area and the target area overlap. If these areas overlap, the copy function must ensure that only data that already exists is read. This process is repeated until all data are restored.

2.1.1.5. Memcpy vs. Custom wild-copy Function

There are several ways to copy data from one memory location to another. The default *memcpy* routine is provided by the compiler. This function can be used if there are large memory blocks to be copied. But that is not enough for the needs of a compressor or decompressor. Combining many small copies with the overhead of a function call results in inefficient code. A function call forces the processor to dump active registers to memory, execute the called function, and then restore all registers. When for example eight bytes should be copied, then it is obvious that dumping of all registers to the memory is an overkill for this task. *LZ4* therefore has a custom *wild-copy* function that copies eight bytes at once from a given source to a particular destination. The reason for copying eight bytes at once is to reduce the loop overhead. In addition, the overhead for the function call is removed. To achieve this advantage, the custom *wild-copy* function must be inlined into the source code, otherwise there is no advantage. The disadvantage of copying eight bytes at once is that this custom *wild-copy* function can not be called at the end for the last literal. A further disadvantage is that the size of the binary code increases by approximately 3% only for the copying code due to the inlining at different locations in the code. As the binary kernel module grows larger, the processor can no longer keep the entire code in the fastest cache.

2.2. Linux kernel

The gcc compiler is used for the compilation of the Linux kernel [Sve17, p.2]. Even if the same compiler is used for the user space code as well as for the Linux kernel, there are some differences.

2.2.1. Missing Features for Linux Kernel Code

Many libraries exist for anything in the user space, but none of them can be used within the Linux kernel. Floating-point hardware and vector registers are disabled in Linux kernel modules, to improve overall system performance, especially in the context of *context switches.* This is possible since large parts of the Linux kernel do not need these features.

In the user space, each application can be started as often as needed. In the Linux kernel, each module can only be loaded once. This allows a Linux kernel module to merge tasks from different users and has the ability to change the order of tasks to improve performance. For example, if several things are compressed in parallel, the module can guarantee that there are at most as many threads as there are processor cores. On the other hand, when the module has to process tasks from different applications, internal locking and queuing mechanisms may lead to a new kind of trouble.

2.2.2. Memory

2.2.2.1. Page-Based Buffers

In Linux, memory is organized in pages. The *struct page* contains all information that the Linux kernel has to know about a piece of memory. Since this structure has a wide range of attributes, it is not explained further in this thesis. All attributes of the *struct page* and their meanings are described in [Gor04, p.118].

```
struct page *page;
```

Listing 2.1: Datatype of a page in the Linux Kernel

Normally a single page represents 4096 bytes. In the Linux kernel, the *PAGE_SIZE* can be adjusted to any power of two. To allocate a single page, the *alloc_page* function can be used. How this function works in detail is found at [Gor03, p.117].

If large continuous memory is needs to be allocated as a single page the function *alloc_pages* can be used. This function can allocate a page containing powers of two as the number of child pages.

```
struct page *alloc_page(unsigned int gfp_mask);
struct page *alloc_pages(unsigned int gfp_mask, int n);
```

Listing 2.2: Allocate a page in the Linux Kernel

The function *__free_page* can be used to release a single page. How this function works in detail is found at [Gor03, p.119].

```
void __free_page(struct page *page);
```

Listing 2.3: Free a page in the Linux kernel

To access page content, the page needs to be mapped first. The function *kmap* [Gor04, p.147] is used to map a page. When the module currently does not need access to the page content, the mapping can be dismissed. This means that only the mapping is invalid, the data itself is still in memory, so that swapping can occur. To dismiss such a mapping, the *kunmap* function is used.

```
void *kmap(struct page *page);
void kunmap(struct page *page);
```

Listing 2.4: Map a page in the Linux kernel

2.2.2.2. Continuous Buffers

To allocate small memory blocks in the Linux kernel, the function *kmalloc* can be used. The function requires the target size for the allocation as a common *malloc*. The *kmalloc* function also requires some flags that describe how and where the memory is to be allocated. If the module no longer needs the memory, then *kfree* should be called as soon as possible to reduce memory fragmentation. Once the memory is allocated, the pointer can be used just as any pointer in the user space can be used. The *kmalloc* function always allocates physical continuous memory. Because physically continuous memory can not be swapped or moved, the size of such allocation is limited by software and hardware. The upper limit for *kmalloc* allocations is dependent on the architecture. But regardless of the architecture, the upper bound for a single allocation is defined by *kmalloc* in *"linux/slap.h"* as 2^{25}, which is 32 MiB. In addition, when the random access memory (RAM) is highly fragmented, the maximum available size at runtime can be additionally reduced. How these functions work in detail can be found at [Gor04, p.118].

```c
void *kmalloc(size_t size, int flags);
void kfree(const void *objp);
```

Listing 2.5: Allocate small memory blocks in the Linux kernel

If the Linux kernel module requires a memory block that is too large, *kmalloc* can not be used because there is not enough physically continuous memory. Large storage areas can only be allocated virtually. Virtual continuous memory is allocated physically scattered, then the small blocks are blended side by side by software. Blending allows parts or even the entire memory area to be swapped to a connected hard disk. The function to allocate virtual continuous memory in the Linux kernel is called *vmalloc*. If the module no longer needs the memory, the function *vfree* must be called to release the memory. To allocate virtual memory, the Linux kernel must modify the page tables. In the page table is coded, which pages are present and where they are located. Changing the page table takes some time, so physically continuous memory should be preferred in the kernel space. How these functions work in detail can be found at [Gor04, p.112-113].

```c
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

Listing 2.6: Allocate large memory blocks in the Linux kernel

If a kernel module allocates memory, then it must ensure that the returned pointer is not NULL. If a user space application detects that there is no memory, then this application will normally crash and be terminated. In the kernel, each module must handle situations where there is no memory without terminating itself or crashing the system.

2.2.3. Scatter-Lists

Scatter-lists are a construct in the Linux kernel to define an interface for copying data between a page-based buffer and a continuous buffer. *Scatter-lists* can be initialized with an array of pages. Thereafter, it is easy to access any data sequence in the page-based buffer. *Scatter-lists* support missing pages. To achieve these gaps between the pages, *Scatter-lists* contain a linked list that stores the offsets of each page along with the pointers to the pages. The disadvantage of being able to have gaps between the pages is, even if there are no gaps, that the internal linked list must be traversed for each memory access. If there are no gaps, the corresponding pages can be calculated in a constant time. Therefore, the overhead of *Scatter-lists* is only acceptable if there are only a few copy operations or when there are gaps between the pages.

2.2.4. Errors & Debug

Bugs and errors in the core space are particularly critical. If an error occurs in the kernel space, it is possible that the whole system crashes and not just the one module that actually causes the error. Therefore kernel modules should detect errors and try to handle them gracefully. In addition, if the Linux kernel module requests more memory than is available, some random user space applications are killed to free memory. This should be avoided whenever possible.

Debugging kernel space modules is more difficult than debugging user space applications. If the kernel crashes, then the whole system exits. As a result, each debugger would terminate immediately too, without having a chance to analyze why the system crashed. To debug kernel code, it is common practice to log some messages with information about the values of some variables. If the system crashes, one can hopefully look into the logfile and maybe it is possible to see what exactly happened. There is a big problem with log messages: Log messages may not be printed or saved to the disk if the system hangs or crashes. Therefore, it is possible to add manual delays after log messages to give the system some time to display the message before doing anything critical. Delays in the code have the disadvantage that the debugging of high-frequent loops with explicit delays requires a lot of time, even if only the last iterations contain delays.

2.2.5. Kthreads

kthreads in the Linux kernel space are similar to processes in the user space. Every Linux kernel thread runs in its own context. Therefore, *context switches* are required when *kthreads* are involved. Linux kernel modules that do not call *msleep* either directly or indirectly, regardless of the elapsed time can not be preempted by the scheduler. Multiple measurements with the *perf* tool proof, that in the case of sequential compression, there are at most two *context switches* - one at the beginning and one at the end. When *kthreads* are used, the number of *context switches* linearly scales with the number of *kthreads* started.

2.2.6. Work_Queue

A *work_queue* is a special queue type defined by the Linux kernel. Once the *work_queue* is initialized, at least one thread is waiting for work to be appended to the queue. The advantage of the *work_queue* is that the complete locking and synchronizing mechanism is already implemented. If the module is not dependent on other modules, then a global *work_queue* can be used that is shared among all other modules. The global *work_queue* reduces the number of concurrent threads. This leads to the advantage that cache misses are reduced after a preemptive interrupt. If the *work_queue* is initialized at the module start time, then this can be used to limit the number of simultaneous threads by processing only the desired number of data records simultaneously.

3. Design and Implementation

3.1. Design of page based algorithms

The first approach for compressing page-based buffers is to compress the pages independently. If each page is compressed independently into another page, it is not necessary to check whether the data are overlapping two pages because only one page is viewed at a time. Unfortunately, when trying to compress the pages, the problem sometimes occurs that some pages might be compressed as desired, while others might increase in size because they are not compressible. If the compressor does not use the entropy of previous pages, the entire compression became useless because the data normally does not contain patterns in that small ranges.

If only some of the pages are compressed, a different array of data are required to store the information, which pages are compressed, and which ones are not. This approach has the advantage that parallelization can be applied perfectly, since all pages are independent of each other.

When a page is fully filled during compression, the compression can continue automatically on the next page. If it is possible to switch to another page, this check must be performed before each r/w operation at the input and output buffer. The problem becomes even greater when more than one byte belongs to a single r/w operation. For example when the offset of a match is decoded, two bytes needs to be read. If the offset on the page is badly chosen, than both bytes are located at different pages. In such a case, the r/w operation itself must verify that these bytes are read from the correct page. This verification results in a large number of branches in the code. Each branch leads to the trashing of the processor pipeline and therefore takes a lot of time. This approach retains the same compression ratio as before, but the throughput is reduced compared to the continuous buffers compression as can be seen in the Chapter 5.

The first subsection evaluates how the *LZ4* algorithm is to be adapted to accept page-based buffers. Then the *BeWalgo* algorithm is designed in both subversions to solve the problems that *LZ4* had. After the introduction of all serial algorithms, the last subsection shows the intended advantages of parallel compression.

3.1.1. *LZ4*

The *LZ4* algorithm itself has already been explained in Subsection 2.1.1. Like any other compression algorithm, *LZ4* treats patterns that are aligned to page boundary the same way as any other pattern. As a result, each r/w operation may be partial on two sides. To access data that is distributed over two pages, several instructions are required. Since these overlapping r/w operations can appear at any position during both encoding and decoding, this takes a lot of time. When it is required to copy data from one page-based buffer to another, it gets worse. Page overlapping r/w operations can now occur on input and output pages at different times, which will increase the required branches in the code to about 580%. In addition, the length coding used by *LZ4* is unfavorable in connection with pages: To read a number, the *LZ4* decompressor must read one byte after another until a byte is not equal to 255. To read the data byte by byte, the decompressor must verify before each read operation that the pointer still points to a valid address on one of the pages.

The performance would increase drastically if it would be possible to rewrite the compressor so that data are never stored in places where they overlap two sides. The compressor would need to split some matches into different partitions to align the r/w operations so that all data are stored either completely on one page or the other. The problem here is the *LZ4* data format, where each match-length is implicitly increased by four. This implicit four makes it impossible to guarantee that all page overlapping r/w operations are eliminated since, in the worst case, the compressed data only consist of four byte matches, which can not be separated at any position. The second problem is that even if such a modified compressor exists, this compressor must eliminate all overlapping r/w operations at both the input and output buffers. This compressor would produce a larger output when trying to split all the matches so that the compression loses effectiveness.

3.1.2. *BeWalgo*64

The compression and decompression performance can be drastically improved if the algorithms can work directly with the page aligned memory. With memory aligned r/w operations, it is possible to read several bytes at the same time without having to access several pages. Known compression algorithms treat aligned patterns just like any other pattern. To improve the speed at the expense of the compression ratio, this thesis introduces a new compression algorithm that uses r/w operations which are aligned to eight byte boundaries. This alignment is possible because the Linux kernel code guarantees that each configurable *PAGE_SIZE* is a power of two. Additionally it can be assumed that the size of a page is larger than eight bytes. The eight byte alignment is selected because most current CPUs support 64-bit architectures and therefore the register word size is eight bytes. Since everything is aligned to eight bytes, each encoded length can hide an implicit scale factor of eight. Figure 3.1 shows the pattern for the compressed data stream.

Control-block						Literals$_1$	Literals$_2$
Literal length$_1$	Match length$_1$	Offset$_1$	Literal length$_2$	Match length$_2$	Offset$_2$		
1 byte	1 byte	2 byte	1 byte	1 byte	2 byte	$[0, N] \cdot 8$ bytes	$[0, M] \cdot 8$ bytes

Figure 3.1.: 64-bit *BeWalgo*64 pattern of compressed data

As can be seen, the control block and the literals have been aligned to an eight byte boundary. The control block contains two sets of literals and matches. Since this is known at compile time, the compiler is expected to optimize the code to access the two portions of the control block as efficiently as possible. However the compiler does not optimize the code as expected. Therefore, the developer must manually duplicate the code to reduce the branches needed for decompression.

The eight byte alignment, of course, has some drawbacks to the compression ratio. If the data to be compressed does not contain patterns that are aligned to eight byte boundaries, the number of matches found is drastically reduced. This can lead to a poor compression ratio. If the matches and the literals do not alternate in the given dataset, then there are numerous bytes that are wasted to encode zero-length literals or matches.

If the data to be compressed contain patterns that are mostly aligned at a four or eight byte boundary, the compression ratio is almost the same as for *LZ4*.

3.1.3. *BeWalgo*32

This compression algorithm is a variant of the 64-bit subversion of *BeWalgo* compression, as described in Subsection 3.1.2. This algorithm substantially reduces the r/w alignment from eight to four bytes. The intended advantage is that even smaller matches can be found without introducing too many branches. Since the alignment of four bytes is still large enough to cover a complete set of literal-length, match-length and offset, a similar pattern can be used for the compressed data. Since the alignment is reduced to four bytes, the implicit scale factor must also be reduced to four for all lengths.

Control-block			Literals
Literal length	Match length	Offset	
1 byte	1 byte	2 byte	$[0, N] \cdot 4$ bytes

Figure 3.2.: 32-bit *BeWalgo*32 pattern of compressed data

3.1.4. Parallelization

Current processors do not show their full performance until tasks are executed in parallel. If the compression algorithm is executed in parallel, then the throughput can be theoretically multiplied. To compress the data in parallel, the original data can be divided into chunks. These chunks can then be compressed independently. The problem with independent chunks is that the compressed data stream should store the data in the right order without gaps. Gaps occur automatically when there are multiple chunks because it is unlikely that every compressed page is filled completely. Gaps between the compressed data would result in a lower compression ratio and are therefore unwanted.

3.2. Implementation

This section describes how the algorithm is implemented and the problems on the way. The first subsection explains what *Scatter-lists* are and why they can not be used. Then it is explained how the *LZ4* algorithm is converted to accept page-based buffers, and why this implementation loses so much performance. Subsection 3.2.3 explains how the *BeWalgo* subversions are implemented and why these implementations are faster than the *LZ4* implementation. The last subsection shows how to optimize the code in the kernel space to get a greater acceleration.

The entire compression source code written for this thesis is available at GitHub [Ben17].

3.2.1. Scatter-Lists

Scatter-lists are a construct defined in the Linux kernel. The idea is that *Scatter-lists* should define an interface for the transfer of data between a page-based buffer and a continuous buffer.

By viewing the Linux source code, you can see that *Scatter-lists* use linked lists. These lists are traversed for each copy process. The compressor spends most of its time reading data from the input buffer. The compressor reads a maximum of eight bytes at the same time. When searching for a match, some positions in the source are even read even several.

If the compressor would use *Scatter-lists*, then the number of branches would increase with the length of the data to be compressed for the search for the correct page to read from. It is obvious that this is not a good idea. During compression, it is previously known that there are no gaps between the pages. Therefore, it is not necessary to have such a linked list. Since each page has the same size, the page index and the position on a page can be calculated in a constant time. All pages are passed to the compressor in the form of an array. This array provides random access to each page as needed.

Another strategy to reduce the number of list iterations is that the *Scatter-lists* copy large memory blocks between the page-based buffers and local continuous buffers. For this, it would be necessary to allocate such additional buffers that would otherwise not be required. This idea has been discarded since memory in the kernel space is limited and the copying process needs additional time.

3.2.2. *LZ4*

Direct writing of page-based code is difficult because of the increased error potential. Therefore, the continuous buffer code is used as the baseline.

First, all pointer operations such as read, write, step forward, and distance calculations are replaced by function calls. These function calls can now be changed to work with page buffers. Due to the structure of the compressed data, the newly introduced branches in the assembler code are required and can not be removed. These helper functions are forced to be inlineed by the compiler to achieve a speed increase of up to 80%. The continuous buffer algorithm has been highly optimized. Some of these optimizations have been removed during conversion to page-based buffers.

In addition, some simple operations are much more complicated than before. For example, while the compressor is searching for a match, it is necessary to compare four byte sections with each other in the input buffer. In the original code for continuous buffers, there is only the comparison of the data from two integer pointers, as shown in Listing 3.1.

```c
match_found = (*a == *b);
```

Listing 3.1: LZ4 Identify a match

When using page-based buffers, one must take care to ensure that each read operation reads all data from the correct page. Listing 3.2 shows only the instructions for a single read-32-bit-operation that automatically moves the pointer forward when the data have been read. Even if there are enough data on the current page, then at least one jump as well as some arithmetic calculations are performed which were not needed for the continuous buffers.

```c
static FORCE_INLINE void LZ4_pages_forward
   ↪ (page_access_helper* p) {
    /* this function contains a lot of arithmetic and is
       ↪ therefore critical for the total execution time */
    p->idx = p->idx + 1; /*number of the page*/
    p->ptr = p->pages[p->idx]; /*pointer to the nth page*/
    p->ptr_end_byte_t = p->pages[p->idx] + PAGE_SIZE;
    p->ptr_end_size_t = p->pages[p->idx] + PAGE_SIZE -
       ↪ sizeof (size_t);
}
static FORCE_INLINE U32 LZ4_pages_read32
   ↪ (page_access_helper* p) {
    U32 res;
    switch (p->ptr_end_byte_t - p->ptr) {
    case 0: /* there are no bytes left on the current
       ↪ page.*/
        /* each 'case' results in an additional jump in
           ↪ assembler code */
        /* lots of arithmetic operations are slowing down the whole process */
        LZ4_pages_forward (p);/* switch to the next page */
        res = *((U32*) p->ptr); /* read the data */
        p->ptr += 4; /* increment the pointer */
        break;
    case 1:/* there is one byte left on the current page. */
        res = (U32) (*(p->ptr));/* read last byte of page */
        LZ4_pages_forward (p); /* continue reading on next page */
        res |= ((U32) (*((U16*) p->ptr)) << 8) | ((U32) (*(p->ptr + 2)) << 24);
        p->ptr += 3;
        break;
    case 2:
        res = (U32) (*((U16*) p->ptr));
        LZ4_pages_forward (p);
        res |= ((U32) (*((U16*) p->ptr)) << 16);
        p->ptr += 2;
        break;
    case 3:
        res = (U32) (*(p->ptr)) | ((U32) (*((U16*) (p->ptr + 1))) << 8);
        LZ4_pages_forward (p);
        res |= ((U32) (*(p->ptr)) << 24);
        p->ptr += 1;
        break;
    default:/* there are enough bytes on the current page */
        res = *((U32*) p->ptr); /* just read all bytes */
        p->ptr += 4;
        break;
    }
    p->offset += 4;/* absolute position is always increased by 4 regarless of a page switch */
    return res;
}
```

Listing 3.2: LZ4 read 4 bytes with page based buffers

3.2.3. *BeWalgo*32 & *BeWalgo*64

Both *BeWalgo* subversions have a very similar pattern for the compressed data. To reduce redundant code, the two subversions share the same source code. The key parts in which the subversions differ are implemented with macros to give the compiler the best starting point for optimization.

The general code structure was inspired by *LZ4*, but since the r/w operations are aligned to four or eight byte boundaries, the implementation is rewritten from scratch, and therefore has several differences to the *LZ4* implementation. The most obvious components are the compressor and the decompressor. The implementation of the algorithm has several aspects. The *BeWalgo* compressor is divided into several code segments, which are usually divided by labels. These labels are used as jump marks to jump directly to any point in the code without having to use redundant branches. The following sections show the most interesting code sections and optimizations in the compressor code.

3.2.3.1. Function Prototype

To avoid a redundant code, only a generic compressor is implemented. To remove the overhead introduced by the generic components, the function must be inlined. The function attribute *no_instrument_function* tells the compiler to omit an empty function call at the beginning of the function. This empty function call would otherwise have been used to allow the trace of an error in the stack. This empty function is particularly problematic when only small files are compressed and the compression algorithm is therefore often called. The *___always_inline* attribute forces the compiler to inline the function. Additionally all compile time constants are forced to be evaluated. If the compiler detects a condition that is always true or false, the corresponding dead branch is removed. The code in Listing 3.3 shows the function prototype for the generic compression function.

```
_attribute__ ((no_instrument_function)) static
  ↪ __always_inline int BeWalgo_compress_generic (
    BeWalgo_compress_internal* wrkmem,
    const BEWALGO_COMPRESS_DATA_TYPE* const source,
    BEWALGO_COMPRESS_DATA_TYPE* const dest,
    const int source_length,
    const int dest_length,
    int acceleration,
    const BeWalgo_safety_mode safe_mode,
    const BeWalgo_offset_check offset_check);
```

Listing 3.3: BeWalgo function prototype for compression

For the later code snippets, it is useful to explain the parameters of the function. Most parameters are similar to the generic *LZ4* compression function.

- The **wrkmem** pointer points to a memory area that the compressor can use during compression. This memory must have a predefined size that is preset by default to four pages. The reason for only a small wrkmem pointer is that this memory area should remain in the fastest cache level as possible. The memory contains only one hash table. This hash table is used to find the next match. This procedure is described in detail in Subsubsection 3.2.3.2.

- The pointer **source** points to the input buffer that is to be compressed. The size of the memory referenced is defined by the parameter **source_length**. The BEWALGO_COMPRESS_DATA_TYPE is used to merge the 32-bit and 64-bit subversion into the same source code.
- The **dest** pointer points to the output where the compressed data is to be written. The maximum length is specified by the parameter **dest_length**.
- The parameter **acceleration** is used to increase speed at the cost of a poorer compression factor to search for the next match.
- The variable **safe_mode** is an internal variable. This variable describes whether the output buffer is large enough even in the worst case if the data is uncompressible. If the memory is large enough, many branches can be omitted, resulting in a performance boost.
- The variable **offset_check** is also an internal variable. This variable defines whether the source data is small enough to prevent the offset from overflowing. If the offset can not overflow, some branches can be eliminated, which in turn increases the throughput.

3.2.3.2. Find Next Match

Listing 3.4 shows the source code for the match searching process.

```
 1 do {
 2     if (unlikely (ip >= source_end_ptr)) {
 3         goto _encode_last_literal;
 4     }
 5     h = b_compress_hash (*ip);
 6     match = source + wrkmem->table[h];
 7     wrkmem->table[h] = ip - source;
 8     if ((*match == *ip)
 9         && ((offset_check == B_NO_OFFSET_CHECK)
10         || (ip - match <= B_OFFSET_MAX))) {
11             goto _find_match_left;
12     }
13     ip += (length++ >> B_SKIPTRIGGER);
14 } while (1);
```

Listing 3.4: BeWalgo Search next match

Figure 3.3 shows a visual representation of the code shown in 3.4. The numbers in the nodes correspond to the line numbers of the Listing.

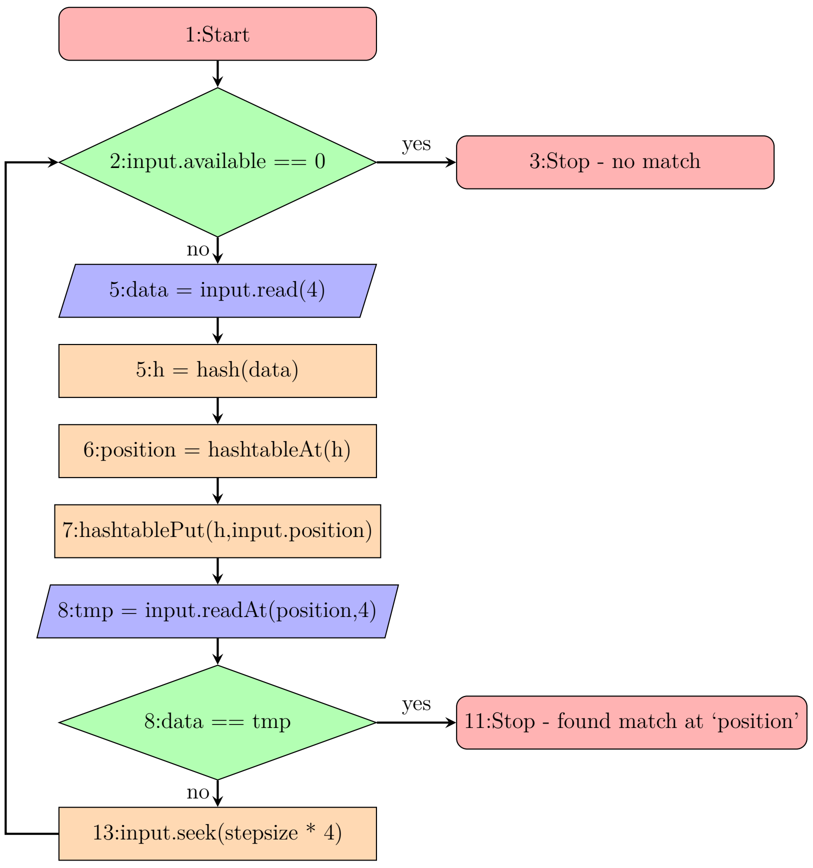

1:Start
2:input.available == 0
yes
3:Stop - no match
no
5:data = input.read(4)
5:h = hash(data)
6:position = hashtableAt(h)
7:hashtablePut(h,input.position)
8:tmp = input.readAt(position,4)
8:data == tmp
yes
11:Stop - found match at ‘position’
no
13:input.seek(stepsize * 4)

Figure 3.3.: *BeWalgo* search for a match

The branch in line 2 checks for the end condition. If the current position in the input array is equal to the first position after the input array, then there is no more match in the input. In this case, the remaining literals must be encoded. The ‘unlikely’ macro tells the compiler that this condition is rarely true. The compiler can use this information to write the corresponding code to the end of the binary file. As an effect, the performance is increased when the prediction is true. If the prediction is incorrect, the code must probably be loaded from cold storage. This results in longer execution times. If you give the compiler such a hint, then this should be true, otherwise it is better not to give the compiler any hints.

Line 5 calculates the hash value for the current couple of bytes. Line 6 uses the calculated hash value to extract the last position which produced the same hash and store it in the ‘match’ variable. Line 7 finally updates the position in the hash table to point to the current position. Calculating the hash requires several operations including multiplication. To force the compiler to produce better code, this hash is stored in a temporary variable. Otherwise as detected in all of my test cases the compiler inserts code which calculates that value twice.

The branch in line 8-10 detects whether a match has been found. Since the hash table is quite small, there are a lot of hash collisions. These hash collisions lead to many false match predictions. The condition in line 9 can be evaluated at compile time. If this expression is true, the compiler will completely omit the source-code in rows 9 and 10. Due to the reduced number of operations, the performance is increased by about 5%. If the condition can not be simplified, the offset must be calculated and compared with the maximum allowable offset. If there is no match, the current position on line 13 is increased. The 'ip' pointer is incremented in incremental steps. In the case of non-compressible data, this improves the performance significantly since most literals are not even touched.

3.2.3.3. Consecutive Matches

If the data are well compressible, there are many consecutive matches. If the whole code for searching a match would be executed, this would consume a considerable amount of time. Therefore, there is a single check for a consecutive match immediately after the encoding of a match. The code that checks this match is identical to the loop content in Listing 3.4. If there is such a consecutive match, execution jumps directly to the code where the end of a match is searched for. Since much code is not executed, the compression gains a lot of performance when there are numerous consecutive matches. The number of consecutive matches increases for well compressible data.

3.2.3.4. Encode Match

In a single control segment, it is possible to store matches up to a length of 255 multiplied by the implicit factor of four or eight. If the matches are much longer, then it takes a lot of time to encode each match individually. To improve performance, only the first matches are encoded by setting the offset and the length individually. The remaining match-length is encoded by repetitively copying the first truly encoded match. Because the number of write operations is halved, the performance increases especially when a file contains large trivial sequences such as zeros.

3.2.3.5. Page-Based Code

Direct writing of page-based buffer code is difficult because of the increased error potential. Therefore, the continuous buffer code is used as a baseline as described above for the *LZ4* code.

The next step is to replace all pointer operations such as read, write, step forward and distance calculations with function calls. These function calls can now be changed to work with a page-based buffer.

After all bugs have been fixed, the next step is to inline all of these page-based functions in order to make all branches visible. Because all page-based functions must check whether the pointer points to a valid page, many branches are nearly exact duplicates. From the developer's point of view, it is possible to remove these branches while the compiler can not remove them. In contrast to *LZ4*, the *BeWalgo* algorithm uses blocks that are aligned to four or eight bytes. Because of the increased alignment for the r/w operations, the number of checks that the page has enough data has been significantly reduced. This is one of the most important differences to the page-based *LZ4* code, as this was not possible there.

In the final code, it is possible to map all pages beforehand and check whether they are randomly physically continuous. This check requires only one addition and one comparison per page. Therefore this check is very cheap. If the pages are continuous, it would be possible to apply the continuous buffers algorithm directly to the page-based buffers to improve throughput. During the speed measurements, this type of improvement is not useful since it introduces a test dependency which can not be controlled and affects the speed significantly.

3.2.4. Optimization

3.2.4.1. Function Calls

Each function call causes the 'call' command in the assembler code. If there is a function call, all registers must be pushed to the stack. In addition, all parameters for the called function must also be put into the stack. The function is executed and then all registers are restored and the stack is popped. If the function is only a small helper, then this overhead is not acceptable. As my benchmarks have shown the performance for the page-based code would be halved.

Therefore, the next step is to provide these helper with the attribute *___always_inline* for the compiler. Just using the *O3* compiler flag is not sufficient. When the compiler detects such a forced inline request, the function is compiled once and the assembler code is then copied to the surrounding function. As a result, there is no 'call' command for this function in the assembler, and the stack operations are omitted.

But the called function is compiled differently than the code that is programmed directly in a single function. If the code is larger, the compiler can reorder the statements more easily if the code is written in a single function. It is not possible to write a trivial example to demonstrate this behavior, since any trivial example would be trivial for the compiler, and would therefore be pointless.

The popular function *memcpy* is one of these short frequent helper functions. The worst thing with *memcpy* is that the gcc compiler does not inline this function in any of the test cases. Since most matches are small, the overhead for *memcpy* is one of the worst cases in the compressor. To eliminate this overhead, a custom version of *memcpy* is implemented where 32 bytes are copied at once. The last bytes are then copied in blocks of eight or four bytes respectively.

3.2.4.2. ?:-operator against if

The *?:-operator* should be preferred over an 'if'. Although both are logically the same, the *?:-operator* is definitely compiled without a branch in assembler code. Since each branch results in trashing in the processor pipeline, a reduced branch count is preferred. In small functions, the compiler can translate transparently between the 'if' and the *?:-operator*. But the compiler does not recognize this in large functions like the compressor code.

Therefore, it is not possible to give a trivial function example to prove this behavior. Benchmark tests from the compressor show that the *?:-operator* increases performance by up to 5%.

3.2.4.3. Vectorization

Most Linux kernel modules do not need vector registers. To improve the overall system performance, these vector registers are not stored before kernel space code is executed. This means that while the kernel code is running, there might be some unsaved user data in the vector registers. Since these data are not stored, it is not allowed to write data to such registers within the kernel module. Since the speed improvements for the general compression code would be small anyway, this optimization strategy is not applied to the compression modules.

3.2.4.4. Parallelization

The compression code itself can hardly be parallelized because the logic of the compression functions is sequential. To apply multiple processor cores for compression, the input data must be split into several chunks. These chunks can then be compressed independently. Completely independent compression tasks can then be performed by different threads. Because of the newly introduced independent compression, the compression ratio is reduced since the entropy of earlier chunks is thrown away. Tests on the 'text' dataset show that the compressed data compared to the sequential compression is increased by 1.6% when a chunk size of 128 KiB is used.

In the Linux kernel, there are two ways to run code in parallel. The first way is to use *kthreads*. The *kthreads* are used similar to the threads in the user space.

Another way to run code parallel in the Linux kernel is to use a *work_queue*. A *work_queue* is initialized with a preferred number of *kthreads*. After initialization, the desired amount of *kthreads* will wait for the work to be placed in that queue. When tasks are appended to the queue, the *kthreads* take those tasks and execute them. The *work_queue* is easy to use and includes all the synchronization and locking mechanisms required to distribute tasks across multiple *kthreads*.

4. Related Work

This chapter presents some related work. This work aims to improve the performance of other algorithms, combine several algorithms for easy exchange, or introduce other compression algorithms.

4.1. Bitshuffle

Bitshuffle [Kiy15] restructures the data before it is compressed. Bitshuffle itself has no influence on the data size. The idea behind Bitshuffle is that even if the data does not contain repeating byte groups, the probability that some bits are equal over several bytes is nevertheless very high. For example, ASCII text that contains a sequence of randomly distributed characters uses only about 32 different characters, while a byte can hold 256 different states. When Bitshuffle is applied to such data, a matrix transpose operation is applied at the bit level. After this matrix transposition, the first bits of successive bytes are merged into one byte. Since ASCII text always sets some bits to zero, the compression algorithm must now compress data with large sequences of zero bytes. This results in a much higher compression ratio even though the characters are randomly distributed. An advantage of Bitshuffle is that the data size remains unchanged. This leads to many possibilities such as vectorization and parallelization. For example, if threads are used, the operations can be perfectly parallelized since the threads access completely independent data segments with exactly the same workload. The same is true for vectorization.

Since Bitshuffle can not be used without a compression algorithm, it is not useful to give an throughput for Bitshuffle alone. Multiple Benchmarks [FA15] show, that the throughput of many different compression algorithms increases, if these are used together with Bitshuffle. In the average case the compression ratio of various compression algorithms is at least doubled.

The disadvantage is that the Bitshuffle algorithm can not achieve full throughput in the Linux kernel because the vector registers are disabled there. But if the code were adjusted to work in the Linux kernel, the Bitshuffle algorithm could work perfectly on page-based buffers without losing performance. Due to the perfect alignment to the page boundaries, the performance for transposing the bit matrix could increase.

4.2. Blosc

Blosc [Alt15] is a user space compression library that is meant to accelerate memory-bound computations. To reduce the memory bus activity Blosc divides the data into small blocks which fit completely in the fastest cache level. To increase the processor throughput SIMD instructions are used whenever possible. To actually compress data several fast compression algorithms like *LZ4* are bundled with the source code. If the *LZ4* compressor is used together with Bitshuffle the throughput can reach up to 22.58 GB/s at a compression ratio of 68 [FA15].

4.3. QuickAssist

Intel® QuickAssist uses special hardware for compression. If this hardware is present, data can be compressed without load on the CPU. This makes it possible to overlay the compression time perfectly with additional calculations. QuickAssist supports two compression algorithms, deflate and LZS, both based on Huffman coding. The disadvantage is that the special hardware must be present. One of the chipsets compatible with compression is the Intel® ColetoCreek 8950 chip [Sil16]. This hardware can reach compression speeds up to 24 GB/s.

4.4. Zstandard

Zstandard [Cha16] or zstd is another compression algorithm. Similar to *LZ4* and *BeWalgo*, this algorithm accepts a parameter to manipulate the compression rate and the ratio. In contrast to *LZ4*, an increase in the parameter leads to a stronger compression. If a very high value is selected, the temporary memory required for compression is increased in addition to the increased time. This algorithm searches and encodes matches like *LZ4*. In contrast to *LZ4*, the literals between the matches are additionally compressed with different algorithms. While the compression rate is thereby improved, the performance suffers. If the standard acceleration is used, then zstd reaches up to 100 MB/s during compression and up to 459 MB/s during decompression.

4.5. LZ4m

LZ4m [Se-17] is a variant of *LZ4*, which is focused on the compression of in-memory data. This algorithm is to be used for data that is going to be swapped from the main memory. As a result, this algorithm must be very fast in order not to negatively affect overall system performance. Since the goal seems to be to compress exactly one page at once, the algorithm can make some assumptions to meet these requirements while reducing the data size. LZ4m is about 25% faster than the original *LZ4* algorithm. The main difference between LZ4m and the *BeWalgo* algorithm is that LZ4m uses less bits to encode lengths and offsets of literals and matches. Due to the reduced number of bits it seems so that LZ4m can not use the entropy of previous pages.

5. Evaluation

During this thesis, three different algorithms were evaluated with continuous buffers and page-based buffers.

Three different datasets were used for the throughput tests. Since the throughput itself is not meaningful, the compression ratio is also given close to all measurements.

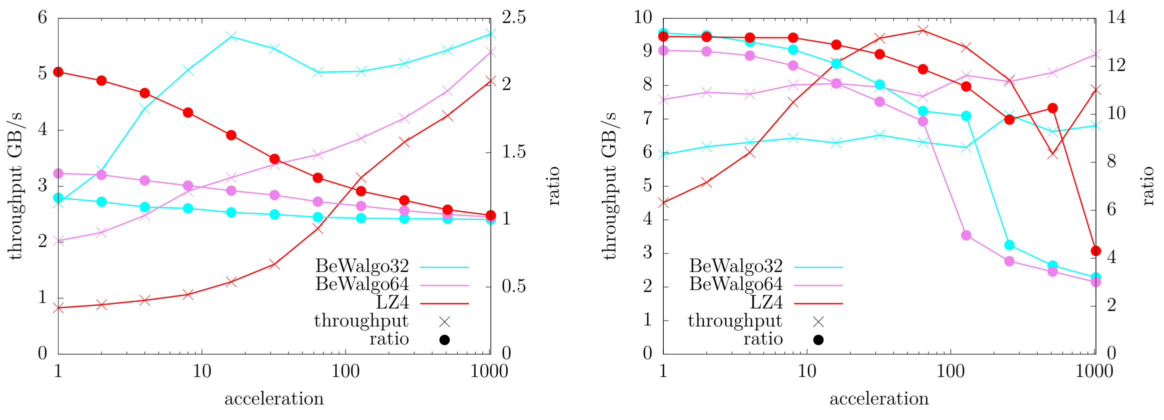
- The first dataset, called ‘silesia corpus’ dataset, contains a collection of files that can not be compressed with lossy algorithms. This dataset was first used by Sebastian Deorowicz in his PHD thesis [Seb03]. The collection consists of medical images, executable files, text, and more. In total, this dataset comprises 212 MB. The data contain many patterns, which are usually very short. The goal of this dataset is to be able to compare lossless compression algorithms with each other.
- The second dataset, called ‘text’ dataset, contains a collection of 300 MB of source code and configuration files found on my system. These numerous files were linked together to form a single large dataset. Since a large portion of the dataset is source code, there are many similar patterns in the data. From the viewpoint of the compressor, this dataset contains many mostly short patterns and is structured similarly to the ‘silesia corpus’ dataset.
- The third dataset, called ‘hdf5’ dataset, contains 468 MB of tbnt data. This dataset contains the relative occurrence of different chemicals in German waters with latitude, longitude and depth. These data are used to simulate and predict how and where chemicals flow through rivers into the ocean. This dataset is closest to the real application because it is data that is actually produced and used on clusters. The compression algorithms can benefit from larger patterns in the file that are likely to be aligned to data types such as integer, float, or double.

The *LZ4* algorithm in the Linux kernel with continuous buffers has been updated by Sven Schmidt as described in his paper [Sve17]. This implementation is used without any change to compare the speeds.

For comparison purposes, the throughput of a custom *wild-copy* function is displayed. This function copies 32 bytes in each loop run until all data has been copied. Within the compression algorithms, this *wild-copy* function is faster than the default *memcpy*. The *wild-copy* function can be applied either directly to pages or to continuous buffers without differences in throughput. Another reason for the comparison with the *wild-copy* function instead of *memcpy* is to ensure that the data are actually copied by the processor, and therefore no hardware acceleration in the background can be used. If hardware accelerated copy would be used, the throughput should not be compared with compression code since the data never reaches the processor while it is copied.

5.1. Acceleration

The *LZ4* algorithm accepts an acceleration parameter to adjust the speed and the ratio as needed. The *BeWalgo* algorithm accepts a parameter to produce the same effects. Figure 5.1 shows that the acceleration factor actually has an effect on speed and the compression rate for all the algorithms presented. For this comparison, the compression throughput is measured only with continuous buffers since the effect is similar for the algorithms with page-based buffers.



(a) 'silesia corpus' dataset (b) 'hdf5' dataset

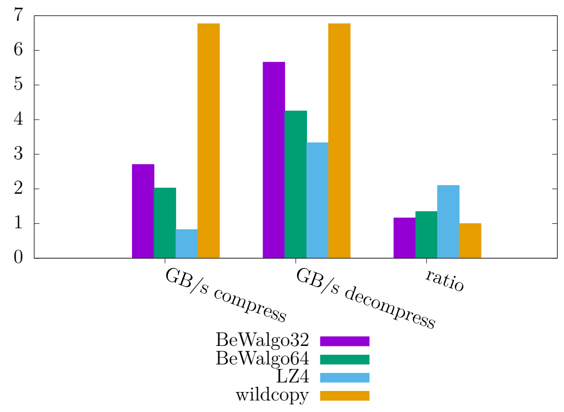
Figure 5.1.: Acceleration effect on speed and ratio

Figure 5.1a shows that the effect of the acceleration parameter leads to an improvement in throughput, while at the same time the compression ratio decreases uniformly. This is the intended and expected behavior. Because the similarities to the 'text' dataset are extreme, that graphic can be found in the attachments.

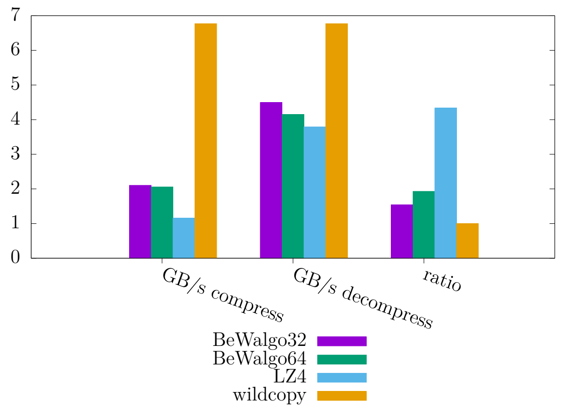
On the other hand, Figure 5.1b shows that if the data is well compressible, the acceleration parameter may lead to negative effects. With increasing acceleration, the *BeWalgo* subversions produce poorer compression ratios for this dataset, while the speed improvement is minimal. In this test case, *LZ4* leads to poorer results. While the compression ratio of *LZ4* decreases with higher acceleration, the size of compressed data increases. As a result there is more data to copy which in turn decreases the compression throughput.

5.2. Continuous Buffer Based Algorithms

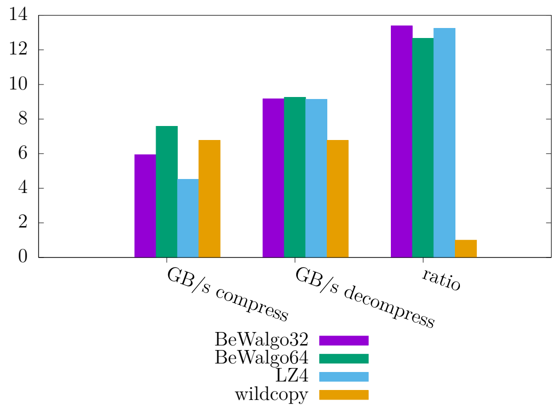
First, the continuous buffer based algorithms are compared. For the tests, each dataset is loaded once into the memory and then compressed directly in the continuous buffers several times. For this comparison, the acceleration factor is set to one, the default value for each algorithm.



(a) ‘silesia corpus’ dataset



(b) ‘text’ dataset



(c) ‘hdf5’ dataset

Figure 5.2.: Comparison of the algorithms using continuous buffers

Figure 5.2a shows that the *LZ4* algorithm can reduce the size of the ‘silesia corpus’ dataset to a half. The *BeWalgo* subversions also reduce the size of this data, but the ratio is negligibly small. Since the *LZ4* algorithm results in a much better ratio in this case, the throughput can not be compared well.

Compressing the ‘text’ dataset as shown in Figure 5.2b results in similar results. In this case, the compression ratio can be doubled compared to the ‘silesia corpus’ dataset, while the throughput remains similar.

On the other hand, the ‘hdf5’ dataset shown in Figure 5.2c results in completely different results. Since the data contains large repeated patterns, all algorithms achieve similar very good compression ratios. The decompression rates are almost the same for all algorithms. When compressing the data, the *BeWalgo* subversions have a minimal better throughput.

Due to the strong compression ratio of the ‘hdf5’ dataset, the compression and decompression algorithms can be faster than the *wild-copy* function. The strong compression ratio of 13 means for the decompressor that on average every byte is read once and written 13 times. This results in very good caching effects that do not occur when the data is only copied from one memory location to another.

5.3. Page Based Algorithms

The goal of this thesis is to compress the data with page arrays as buffers, which are to be used in the Lustre file system. The acceleration factor for all algorithms in the following figures is set to the default value as before. Regardless of the use of page-based or continuous buffers, the compression ratio is the same. The compressed data from the *BeWalgo* algorithms are even binary-identical between the continuous and page-based buffer versions. The throughput measurements marked with 'copy' contain the necessary operations to copy the page-based buffer into the continuous buffer and back. The algorithms that natively accept page-based buffers are not affected by this flag.

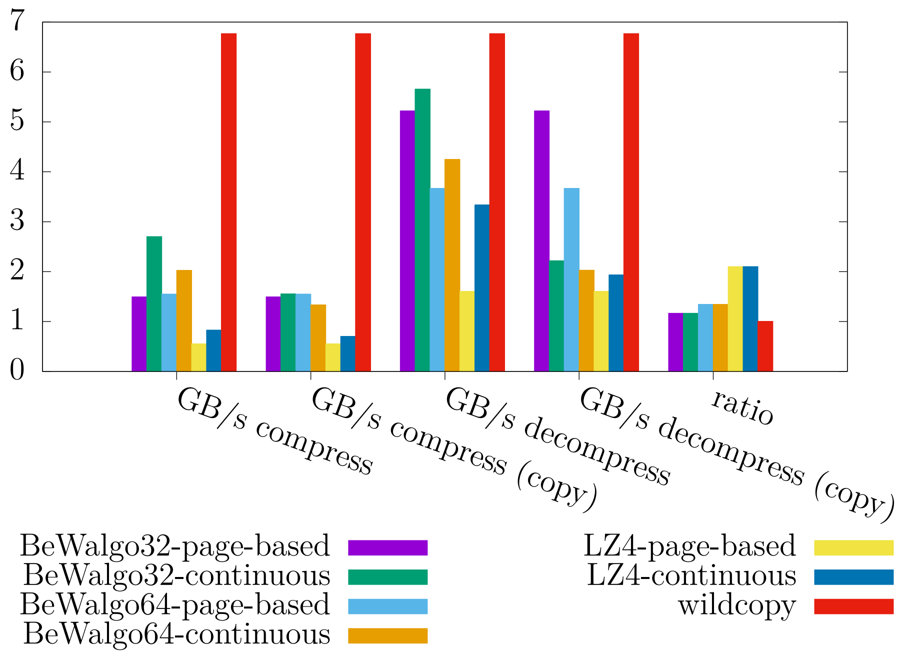


Figure 5.3.: 'silesia corpus' dataset

As is known from the previous chapter, the *BeWalgo* algorithm can not very well compress the 'silesia corpus' dataset while the *LZ4* algorithm can reduce the same data to about half. Figure 5.3 shows that the time required for copying to continuous buffer is a massive impact on overall performance. In the case of *BeWalgo*, the throughput can be doubled if the data are compressed natively in page-based buffers. The *LZ4* algorithm shows a surprising effect: if the compression - especially the page-based buffer version - is extremely slow, then it is faster to copy the data into continuous buffer, compress them and then copy them back. One reason for this effect is the nature of the *LZ4* algorithm, which completely ignores the alignment of the patterns. The continuous buffer version of the *LZ4* algorithm can perform some good optimizations that are not applicable in a page-based environment. In this way, the continuous buffer *LZ4* compressor achieves 1.5 times the throughput of the page-based version. In the context of decompression, the page-based *LZ4* reaches only half the throughput of the continuous buffer version.

In contrast to the 'silesia corpus' dataset, the 'hdf5' dataset can be compressed very well with all selected algorithms. Figure 5.4 shows that the compression ratios are almost identical for all compared algorithms.

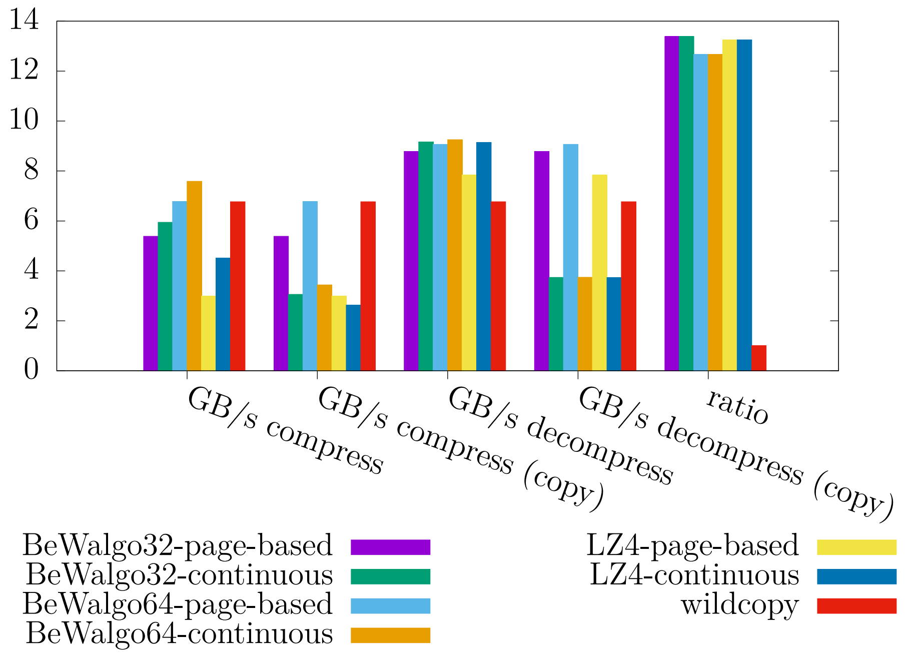


Figure 5.4.: 'hdf5' dataset

The comparison of the throughput without the 'copy' flag shows that the continuous buffer algorithms are always at least slightly better than their page-based counterparts. In the case of *LZ4*, the continuous buffer algorithms are far better than their page-based counterparts due to the nature of the algorithm.

Data transfer between page-based and continuous buffers reduces the throughput of the affected algorithms by half. As a result, the algorithms that accept page-based buffers natively are drastically faster in the Lustre context. As a bonus, the page-based algorithms do not even require a temporary continuous buffer in the kernel space.

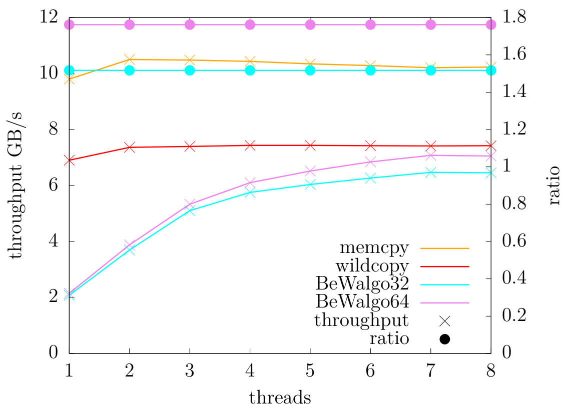
The *LZ4* compression algorithm compared to the *BeWalgo* algorithm results in only half of the throughput. At this point, the nature of *LZ4*, which ignores the pattern alignment, leads to a huge disadvantage.

5.4. Parallel Compression

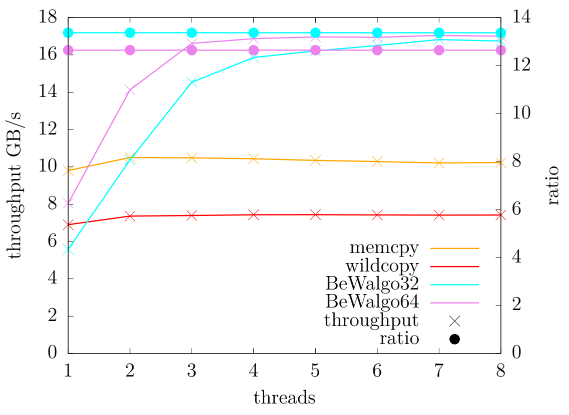
Since the compression consumes significantly more time than the decompression, only the compressor is initially parallelized. To parallelize the compression, the data is divided into chunks of 128 KiB, which are then compressed in parallel. The constant chunk size independent from the number of threads leads to the same compression ratio regardless of the number of threads. In the following subsections, only the *BeWalgo* algorithm is displayed because the *LZ4* data format does not allow the concatenation of the compressed data streams. *LZ4* compressed data can not be concatenated because it is impossible to define match-lengths of zero between two literals.

5.4.1. Continuous Buffer Based Algorithms

First, the continuous buffer algorithms are compared in terms of acceleration gained by the use of more processor cores.



(a) 'text' dataset



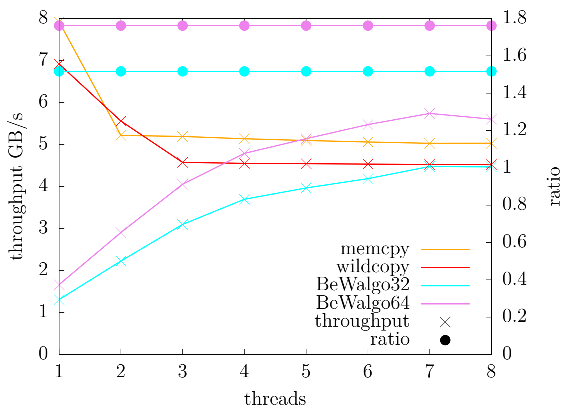
(b) 'hdf5' dataset

Figure 5.5.: continuous buffer based algorithm throughput by thread count

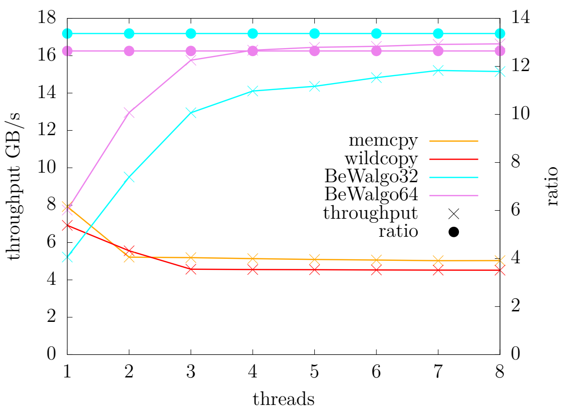
Parallel compression of the 'text' dataset shows that the algorithm scales well with the number of processor cores, as shown in Figure 5.5a. If well compressible data such as the 'hdf5' dataset is compressed in parallel, the algorithm scales up to a maximum of 17 Gb/s. This seems to be a hardware limitation of the main memory bandwidth. Surprisingly, the compression of the 'hdf5' dataset is much faster than the parallel *wild-copy* as well as the parallel *memcpy*, where the data are only copied. Because of the compression, the *BeWalgo* algorithm requires a much smaller output buffer. Therefore, the cache is used more efficiently, enabling higher throughput.

5.4.2. Page Based Algorithms

Finally, the page-based compression algorithms are compared with respect to the acceleration obtained with more processor cores.



(a) 'text' dataset



(b) 'hdf5' dataset

Figure 5.6.: page-based algorithm throughput by thread count

Figure 5.6 shows that the throughput and compression ratio of the *BeWalgo* algorithms remain nearly the same when using page-based buffers instead of continuous ones. Thus, it would be obvious that if the continuous buffer algorithms were used in Lustre, the throughput would definitely decrease as the extra time to copy the data to continuous buffers would cause a large overhead.

When the page-based *wild-copy* algorithm is run with multiple threads, the throughput decreases with increasing thread numbers. A possible reason for this behavior might be that the cache gets invalidated more often when multiple cores concurrently map and unmap a lot of pages. The page-based *memcpy* algorithm is a bit faster than the *wild-copy*, and scales similar.

6. Conclusion and Future Work

6.1. Conclusion

The processor performance is increasing faster than the performance of any storage related component. To reduce this gap, data can be compressed before it is stored or transmitted. With compression the size of data can be decreased, while at the same time the latency is increased. The increasing latency is acceptable as long as the data is accessed in big bulks. To improve the performance of many applications at once the compression is applied in the underlying file system. The file system of choice is the distributed kernel file system Lustre.

There exist many different compression algorithms. To improve the performance in this context the compression algorithm must be very fast and lossless. The chosen algorithm is *LZ4* since it is currently the fastest. All available compression algorithms use continuous buffers for input and output. To maximize the performance in the kernel the compression algorithms must directly operate on the page based buffers.

There are multiple possible strategies how to handle the knowledge that there is an array of pages in the background. The most efficient strategy is to compress the whole page array at once and be careful to write the data always to the correct page. The original *LZ4* algorithm looses a lot of performance if the data is stored in page arrays.

The *BeWalgo* solves the performance problems of *LZ4* with page buffers at the expense of a potentially poorer compression ratio by viewing the input data in an 8 byte raster. Additionally the *BeWalgo* algorithms open the possibility to concatenate the parallel compressed binary chunk data without storing the individual compressed chunk sizes. By concatenating the binary compressed data the compression ratio can be increased while being able to compress parallel. *LZ4* requires these individual chunk sizes because zero length matches in the middle are not encodable.

6.2. Future Work

Compression in the Linux kernel is still under development. Therefore, there will be a lot of work in the future until compression is used in productive environments.

Many different compression algorithms exist. Each compression algorithm has different advantages and disadvantages. For example, if the data is stored for a long time, it might be useful to use very strong compression to reduce the disk space required. If the data is produced very quickly, it would be better to use a weak and fast compression algorithm to make better use of the hardware. To use the individual advantages a lot of algorithms needs to be adapted to work in the Linux kernel.

Bibliography

[Alt15] Francesc Alted. Blosc: A blocking, shuffling and lossless compression library. *blosc.org*, 2015. [Online; accessed July 28, 2017].

[Ben17] Benjamin Warnke. PageBasedCompressionInTheLinuxKernel. `https://github.com/Qualenritter/PageBasedCompressionInTheLinuxKernel.git`, 2017. [Online; accessed July 14, 2017].

[Car02] Carlos Carvalho. The Gap between Processor and Memory Speeds. Technical report, Universidade do Minho, 2002. [Online; accessed June 21, 2017].

[Cha16] Charles Vandevoorde. Analysis of Zstandard, a fast and modern compression algorithm. Technical report, ECAM, December 2016. [Online; accessed Juli 2, 2017].

[Col17] Yann Collet. LZ4 - Extremely fast compression. *github.com*, 2017. [Online; accessed July 13, 2017].

[FA15] Valentin Hänel Francesc Alted. python-blosc: a Python wrapper for the extremely fast Blosc compression library. *github.com*, 2015. [Online; accessed August 11, 2017].

[Fuc16] Anna Fuchs. Client-Side Data Transformation in Lustre. Master's thesis, Universität Hamburg, Mai 2016.

[Gor03] Mel Gorman. An Investigation into the Theoretical Foundations and Implementation of the Linux Virtual Memory Manager. Master's thesis, University of Limerick, Limerick, 2003.

[Gor04] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open source series. Prentice Hall, 2004.

[Int14] Intel. Lustre: The Most Used HPC File System. `https://image.slidesharecdn.com/austincherian-bigdataandhpctechnologies-intel-141105191352-conversion-gate01/95/austin-cherian-big-data-and-hpc-technologies-intel-32-638.jpg?cb=1415216084`, 2014. [Online; accessed June 9, 2017].

[Kiy15] Kiyoshi Masui, Mandana Amiri, Liam Connor, Meiling Deng, Mateus Fandino, Carolin Höfer, Mark Halpern, David Hanna, Adam D. Hincks, Gary Hinshaw, Juan Mena Parra, Laura B. Newburgh, J. Richard Shaw, Keith Vanderlinde. A compression scheme for radio data in high performance computing. *Astronomy and Computing*, 12:181–190, September 2015. [Online; accessed Mai 31, 2017].

[Lus17] Lustre. Lustre. `http://lustre.org/`, 2017. [Online; accessed Mai 16, 2017].

[Mel15] Mellanox Technologies. InfiniBand Now Connecting More than 50 Percent of the TOP500 Supercomputing List. `https://www.scientificcomputing.com/news/2015/07/infiniband-now-connecting-more-50-percent-top500-supercomputing-list`, 2015. [Online; accessed June 21, 2017].

[nvi17] nvidia. CPU and GPU Trends. `http://slideplayer.com/slide/8113417/25/images/4/CPU+and+GPU+Trends.jpg`, 2017. [Online; accessed July 13, 2017].

[Se-17] Se-Jun Kwon, Sang-Hoon Kim, Hyeong-Jun Kim, and Jin-Soo Kim. LZ4m: A Fast Compression Algorithm for In-Memory Data. In *IEEE International Conference on Consumer Electronics (ICCE)*, 2017. [Online; accessed Juli 2, 2017].

[Seb03] Sebastian Deorowicz. *Universal lossless data compression algorithms*. PhD thesis, Silesian University of Technology, Gliwice, 2003. [Online; accessed Juli 2, 2017].

[Sil16] Silicom Ltd. Silicom Enhances Intel QuickAssist Technology for Compression. `http://www.silicom-usa.com/blog-5-test/`, 2016. [Online; accessed June 10, 2017].

[Sve17] Sven Schmidt. Implementierung von LZ4Fast im Linux-Kernel. [Online; accessed Mai 31, 2017], 2017.

Appendices

A. Test-System

Table A.1 shows the relevant hardware properties of the test-system used for the measurements.

Architecture	x86_64
Endian	Little
Processor	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
CPU max MHz	4400
CPU min MHz	800
L1d cache (per core)	32K
L1i cache (per core)	32K
L2 cache (per core)	256K
L3 cache (total)	8192K
Memory-clock	1600MHz
Memory-type	DDR3
Memory-size	16GB
OS	Ubuntu 17.04
kernel	ubuntu 4.10.0-21-generic
Compiler	gcc version 6.3.0 20170406

Table A.1.: Test-System

B. Chapter

This chapter contains graphs similar to the illustrations in the text. For completeness, these graphs are presented in this chapter.

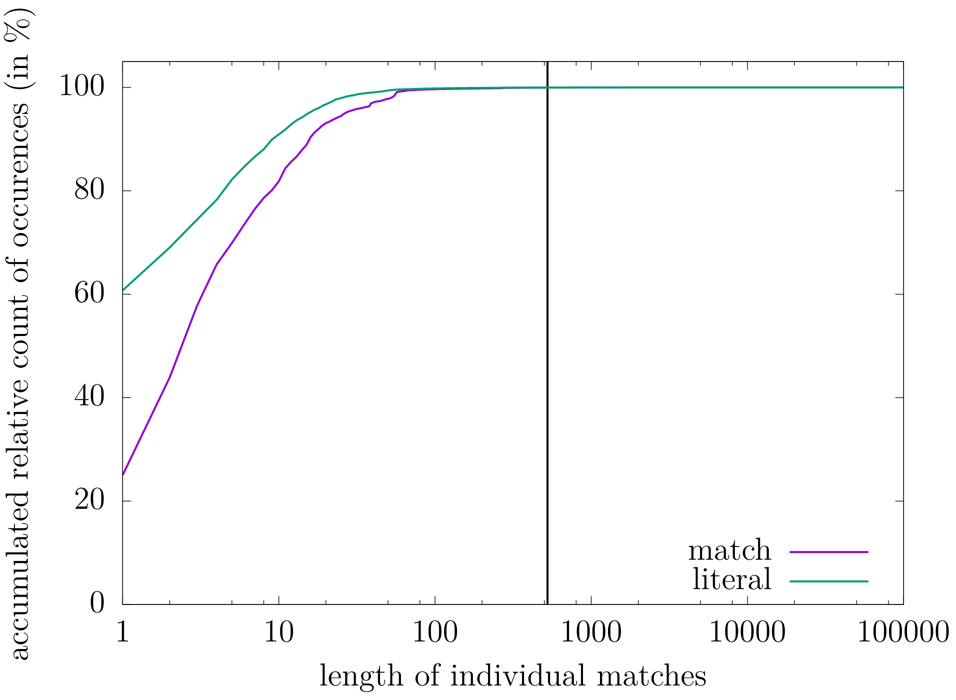


Figure B.1.: Length distribution of literals and matches with the *LZ4* compressor - 'silesia corpus' dataset

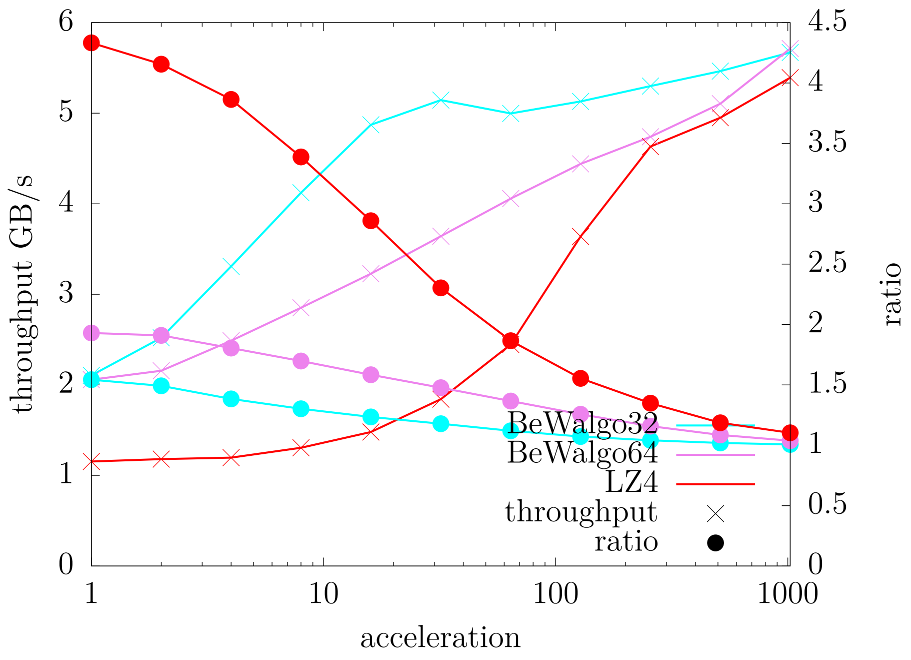


Figure B.2.: Acceleration effect on speed and ratio - ‘text’ dataset

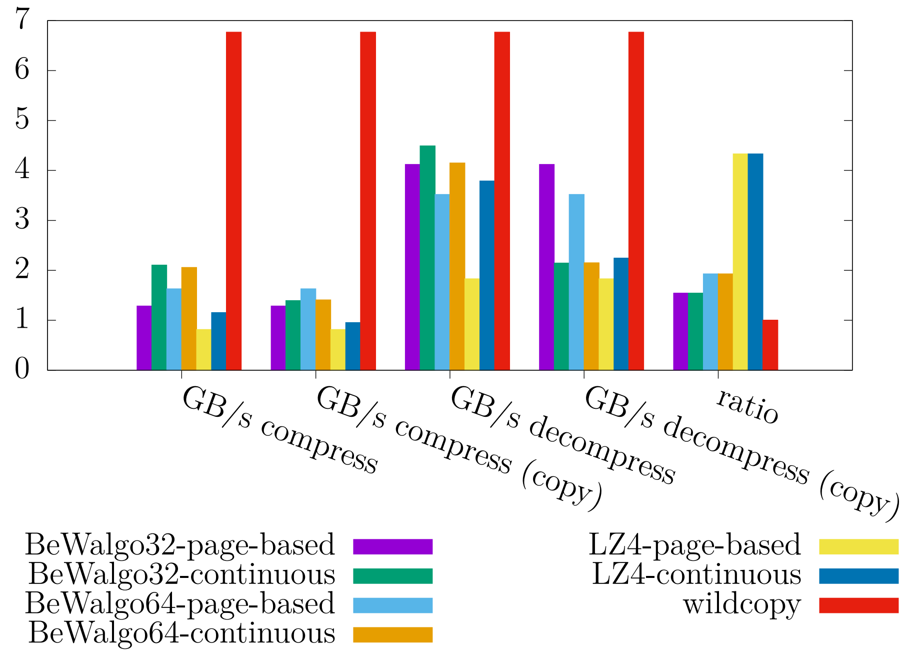


Figure B.3.: compressing the ‘text’ dataset using page based buffers

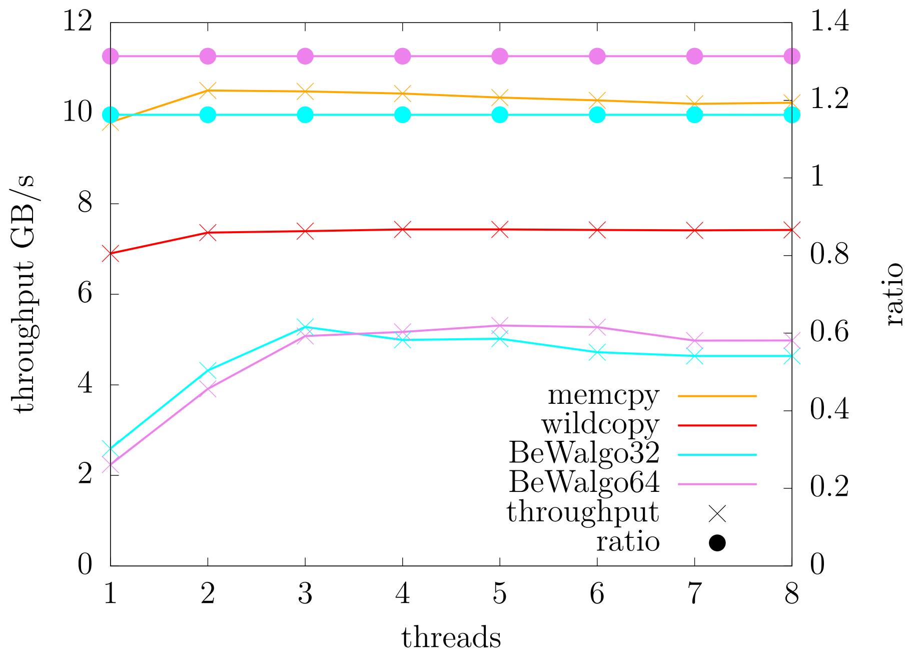


Figure B.4.: continuous buffers algorithm throughput by thread count for the 'silesia corpus' dataset

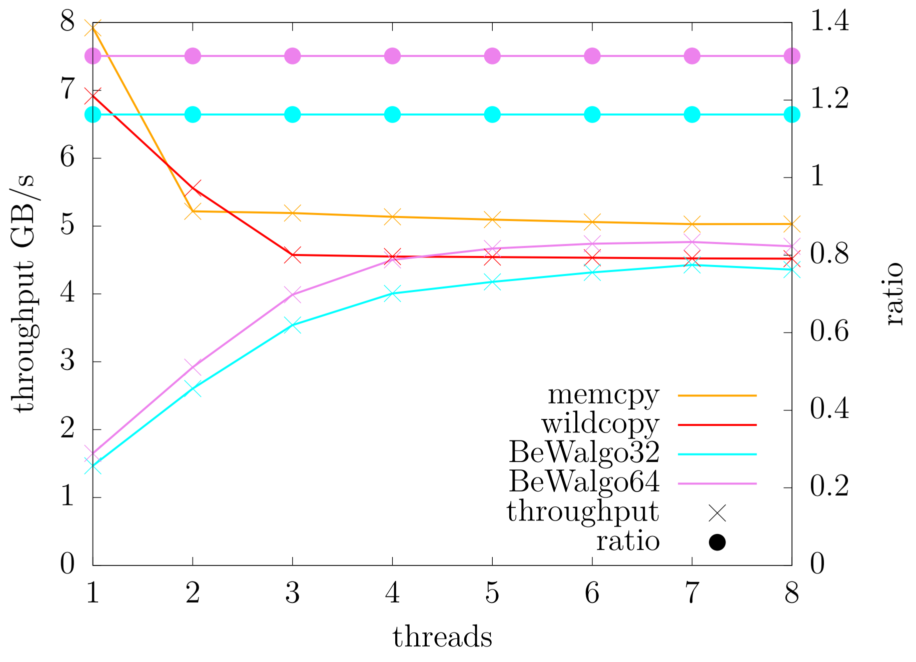


Figure B.5.: page-based buffers algorithm throughput by thread count for the 'silesia corpus' dataset

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Bachelor Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift