Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

# Implementierung von LZ4Fast im Linux-Kernel

vorgelegt von

Sven Schmidt

Projekt „Parallelrechnerevaluation“
Arbeitsgruppe Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften

| | |
|---|---|
| Matrikelnummer: | 6647018 |
| Studiengang: | Informatik |
| Betreuer: | Anna Fuchs; Dr. Michael Kuhn |
| Abgabedatum: | 17.03.2017 |

# Abstract

*Ziel der vorliegenden Arbeit und des zugehörigen Projekts ist es, den Kompressions-Algorithmus LZ4 im Linux-Kernel durch eine aktuelle Version zu ersetzen. Die Idee ist, langfristig Kompression über LZ4 in Lustre zu ermöglichen. Dafür wurde der Upstream-LZ4-Code zunächst als Kernel-Modul realisiert und in Lustre eingebaut. Danach wurde der Code für die Integration in den Linux-Kernel vorbereitet und der entsprechende Submitting-Prozess durchlaufen. Der Fokus der Arbeit liegt auf notwendigen Anpassungen des User-Space-LZ4 im Hinblick auf Besonderheiten des Kernels sowie auf Funktionstests und Benchmarks im Vergleich zur vorherigen Version. Diese Arbeit richtet sich an Studierende und Entwickler, für die ein Einblick in die Linux-Kernel-Entwicklung interessant ist.*

# Inhalt

# 1 | Einleitung

Durch immer leistungsfähigere Rechner können Jahr für Jahr mehr und mehr Daten in der gleichen Zeit erfasst, berechnet, aggregiert oder verarbeitet werden. Zur Weitervarbeitung solcher Datenmengen müssen diese zunächst gespeichert werden. Allerdings wächst Speicherkapazität nicht in dem gleichen Maße, wie Rechenleistung. Es liegt daher nahe, die Daten vor der Speicherung zu komprimieren. Viele Dateisysteme sind heute dazu in der Lage, Kompression automatisiert zu leisten. Diese Menge umfasst derzeit allerdings keine *verteilten Dateisysteme*[1].
Zu den bekanntesten, verteilten Dateisystemen zählt **Lustre**, ein linux-basiertes Dateisystem, das permanent auf der Hälfte der Top zehn der schnellsten Supercomputer der Welt zum Einsatz kommt[2]. Bei solchen verteilten Dateisystemen kommt noch ein zweiter, wichtiger Faktor zum Tragen: Die Netzwerkgeschwindigkeit, denn der Client sowie Lustres Server und Speicher-Knoten können an verschiedenen Enden der Welt stehen, sodass Daten über das Netzwerk versendet werden müssen. Auch hier ist Kompression ein wichtiger Faktor.

Die Realisierung von Kompression in Lustre ist ein Projekt am Arbeitsbereich Wissenschaftliches Rechnen der Uni Hamburg[3]. Der genutzte Kompressionsalgorithmus ist ***LZ4***, der sowohl verlustlos als auch sehr schnell arbeitet: Der Kompressor leistet pro Kern **400 Megabyte pro Sekunde**, während der Dekompressor sogar die **Geschwindigkeit des Arbeitsspeichers** erreichen kann[4]. Interessant an LZ4 ist der Ansatz, dem Kompressor keinen Parameter für die Kompressionsrate (Grad der Kompression) mitzugeben, sondern für die *Kompressionsgeschwindigkeit*: Je höher diese *acceleration*, desto niedriger die Kompressionsrate und umgekehrt. Beispielsweise können Daten für die Übertragung über das Netzwerk mit hoher acceleration und folglich hoher Geschwindigkeit komprimiert werden, für die persistente Speicherung aber mit niedrigerer acceleration, um höhere Kompression im Austausch für eine längere Laufzeit zu erhalten.

Zu Beginn des Projekts existiert LZ4 seit Version 3.11 (2013) im Linux-Kernel. Allerdings ist die benutzte Version stark veraltet; sie bietet für die Kompression nur die Funktion `lz4_compress`. Der acceleration-Parameter kam erst in einer späteren Version[5] mit der Funktion `LZ4_compress_fast`. Diese Variante von LZ4 nennen wir im Folgenden auch *LZ4Fast*.

[1] Wikipedia, *Category:Compression file systems — Wikipedia, The Free Encyclopedia*.
[2] Wikipedia, *Lustre (file system) — Wikipedia, The Free Encyclopedia*.
[3] https://wr.informatik.uni-hamburg.de/research/projects/ipcc-l/start
[4] Colett, *LZ4 github repository*.
[5] Cyan, *Sampling, or a faster LZ4*.

# 2 | Vorbereitungen

## 2.1 Installation eines aktuellen Kernels

Während der Entwicklung des 3.10-Kernels (2013) gab es 9,02 Änderungen pro Stunde[1]. Folgerichtig sollte Kernel-Entwicklung immer auf Grundlage des aktuellen *Mainline-Kernels* erfolgen; gemeint ist die Version, die gerade entwickelt wird.
Jedes Linux-Derivat, wie zum Beispiel Debian, kommt mit einem so genannten *Distributions-Kernel*, mit der es ausgeliefert wird. Welcher Kernel genutzt wird, verrät `uname -r`:

```
$ uname -r
3.16.0-4-amd64
```

Listing 2.1: Die Ausgabe von `uname -r` verrät die Kernel-Version

Der aktuelle[2] Mainline-Kernel ist 4.10-rc8. Im Folgenden wird es daher zunächst darum gehen, wie ein aktueller Kernel gebaut und installiert wird. Dadurch werden die Grundlagen dafür vermittelt, wie Änderungen am Kernel getestet werden können.

Ein gutes Referenz-Werk stellt

*Kroah-Hartman, Greg (2007). Linux kernel in a nutshell. Farnham: O'Reilly.*

dar. Das Werk kann kostenlos bezogen werden (siehe Anhang B). Greg Kroah-Hartman ist der Verantwortliche für die Stable-Releases des Kernels. Im Vergleich zu den Mainline-Versionen, die Linus Torvalds rausgibt (etwa 4.10), sind Stable-Releases an dem Schema x.y.z zu erkennen (4.10.1). Damit ist er, nach Linus Torvalds, einer der wichtigsten Kernel-Verantwortlichen.

### 2.1.1 Voraussetzungen

Um einen Kernel zu bauen, werden nicht viele Werkzeuge benötigt: Ein *Compiler*, ein *Linker* und das Tool ***make***. Der Linux-Kernel kann nur mit dem C-Compiler aus der *GNU Compiler Collection (gcc)* kompiliert werden, die auch den Linker mitbringt. gcc ist in der Regel vorinstalliert. Für die Entwicklung brauchen wir außerdem das Versionskontrollsystem ***git***. `make` ist

[1] Kroah-Hartman, *3.10 Linux Kernel Development Rate*.
[2] 17.02.2017

im Paket *build-essential* enthalten. Unter Debian-basierten Systemen genügt daher ein Aufruf von `sudo apt-get update && sudo apt-get install -y git build-essential`.

### 2.1.2 Beziehen des Kernel-Sourcecodes

Grundsätzlich existieren verschiedene Möglichkeiten, den Sourcecode für den Linux-Kernel zu beziehen. Fertige Abbilder des Kernels stehen zum Beispiel auf kernel.org zur Verfügung. Da wir die Mainline-Version benötigen, nutzen wir `git`, um den Sourcecode direkt aus einem Mainline-Repository zu klonen. Listing 2.2 zeigt ein Beispiel.

```
$ git clone http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux
    .git/ linux-source
$ cd linux-source
```

Listing 2.2: Klonen des Kernel-Trees aus einem git-Repository

Die Nutzung von `git` hat außerdem den Vorteil, dass zu jedem Zeitpunkt über `git pull` der aktuelle Stand aus dem Repository abgerufen werden kann, sodass das eigene Kernel-Abbild immer dem Stand der Entwicklung entspricht.

### 2.1.3 Konfiguration des Kernels

Der Linux-Kernel ist modular aufgebaut. Das bedeutet, die einzelnen Komponenten des Kernels sind größtenteils sauber voneinander getrennte, selbstständige Einheiten. Damit ist es nicht obligatorisch, beispielsweise Gerätetreiber in das Kernel-Image zu integrieren, die von der genutzten Hardware gar nicht benötigt werden (falls kein CD-ROM-Laufwerk vorhanden ist, muss auch kein entsprechender Treiber verfügbar sein). Bei der Konfiguration des Kernels wird der User vor eine Reihe von Fragen gestellt, die nicht alle einfach zu beantworten sind. Falsche Antworten können entsprechend zu Systemen führen, die nicht lauffähig sind, da Komponenten fehlen. Umgekehrt ist ein kompakter Kernel performanter. Das ist auch unter dem Aspekt wichtig, dass der Kernel als fundamentaler Teil des Betriebssystems permanent im Arbeitsspeicher liegt und der für ihn reservierte Speicher erst beim Herunterfahren freigegeben wird. *Linux kernel in a nutshell* enthält daher ein komplettes Kapitel über die Konfiguration eines Kernels.

```
$ cd linux-source
$ make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/x86 4.10.0-rc8 Kernel Configuration
*
...
Deadline I/O scheduler (IOSCHED_DEADLINE) [Y/n/m/?]
```

Listing 2.3: `make config` ist der Basis-Befehl für die Konfiguration eines Kernels

Der grundlegende Befehl ist `make config`. Ein Beispiel umfasst Listing 2.3. Mit dieser Methode wird der Benutzer Schritt für Schritt nach jeder verfügbaren Option gefragt. In der Ausgabe sehen wir die Frage nach dem *Deadline I/O Scheduler*. Das Token [Y/n/m/?] verrät, welche Antwort-Möglichkeiten bestehen:

- **y**: Ja: Integration in das Kernel-Abbild
- **n**: Nein: Entsprechend keine Integration
- **m**: Modul: Die Komponente wird als Modul gebaut, das bei Bedarf in den Kernel geladen werden kann. Details dazu umfasst Kapitel 3.
- **?**: Zeige einen Hilfe-Text. Im Beispiel wird dann der Text

  > CONFIG_IOSCHED_DEADLINE:
  >
  > The deadline I/O scheduler is simple and compact. It will provide CSCAN service with FIFO expiration of requests, switching to a new point in the service tree and doing a batch of IO from there in case of expiry.

  ausgegeben.

Neben solchen Fragen existieren noch weitere, beispielsweise kann der User zwischen Alternativen durch Eingabe einer Zahl wählen. Eine der Antwortmöglichkeiten wird hervorgehoben, im Beispiel durch einen Großbuchstaben (Y). Das ist jeweils die empfohlene Antwort, die ein Druck auf *Enter* übernimmt. Kernel-Version 2.6.18 umfasst „almost two thousand different configuration options“[3]. Jede Frage beantworten zu müssen, ist daher zeitintensiv. `make config` ist daher nicht die einzige Möglichkeit, wichtig sind vor allem:

- **randconfig**: Erzeugt eine Konfiguration mit zufälligen Antworten. Während sich ein Standard-User nicht darauf verlassen möchte, dass ein bestimmter Treiber *zufällig* integriert wird, sind zufällige Konfigurationen zum Entwickeln interessant, da sie Fehler aufdecken, die nur unter bestimmten Kombinationen auftreten.
- **defconfig**: Erzeugt eine Standard-Konfiguration. Diese liegen unter `linux/arch` und werden durch die Variable `ARCH` beeinflusst. `make ARCH=x86_64` baut beispielsweise die defconfig aus `linux/arch/x86/configs/x86_64_defconfig`.
- **allmodconfig**: Konfiguration, in der alle Module integriert werden, sofern möglich.
- **allnoconfig**: Antworte **n** auf alle Fragen.
- **allyesconfig**: Antworte **y** auf alle Fragen.
- **oldconfig**: Verhält sich wie `make config`, übernimmt allerdings Antworten aus einer `.config`, die bereits im Sourcetree liegt

[3] Kroah-Hartman, *Linux kernel in a nutshell*, p. 18.

Für die Entwicklung am Kernel ist die in Listing 2.4 dargestellte Methode am einfachsten, bei der die aktuelle Konfiguration aus dem `/boot`-Verzeichnis kopiert und mit `make oldconfig` aktualisiert wird. Dadurch wird sichergestellt, dass der kompilierte Kernel auf jeden Fall funktioniert.

```
$ cp /boot/config-`uname -r` .config
$ yes "" | make oldconfig
```

Listing 2.4: Kopieren der aktuellen Kernel-Konfiguration und Update mittels `make oldconfig`

Der Befehl `yes` drückt solange die angegebene Sequenz in den Standard Output, bis der User terminiert. Der erzielte Effekt ist, eine dauerhaft gedrückte Enter-Taste zu simulieren, sodass für jede Frage, die in der vorhandenen Konfiguration nicht beantwortet wurde, die Empfehlungen übernommen wird.

### 2.1.4 Build und Installation eines Kernels aus dem Sourcecode

Mit dem Kernel-Sourcecode und der Konfiguration kann der Kernel gebaut werden. Unter Debian ist es hinreichend, statt alle Komponenten (etwa das Kernel-Abbild und die Module) einzeln zu bauen, ein Debian-Paket erzeugen zu lassen, das sich über den Paketmanager installieren lässt. Dazu kann beispielsweise der in 2.5 dargestellte Befehl genutzt werden. Weitere Optionen für andere Systeme liefert die Ausgabe von `make help`.

```
# make -j8 deb-pkg LOCALVERSION=-lz4-test-v8-build-15
```

Listing 2.5: Beispiel-Befehl, um einen Kernel als Debian-Paket zu erzeugen

Der Parameter `-j` gibt an, wie viele Threads `make` zur parallelen Kompilation nutzen darf. Erzeugt werden am Ende eine Reihe von Paketen. Interessant für uns sind die Dateien `linux-headers-*.deb` und `linux-image-*.deb`, zum Beipiel `linux-headers-4.10.0-rc8-lz4-test-v8-build-15_4.10.0-rc8-lz4-`
Der String **-lz4-test-v8-build-15** stammt aus der Angabe von *LOCALVERSION*. Für meine Zwecke habe ich jeweils Versions-Informationen und den Build mit angegeben, um sie unterscheiden zu können. Der Kernel kann dann installiert werden, wie in Listing 2.6 dargestellt. Für die Entwicklung am Kernel empfehle ich eine virtuelle Maschine, da Fehler dort keinen Einfluss auf das laufende System haben.

```
# dpkg -i linux-headers-4.10.0-rc8-lz4-test-v8-build-15_4.10.0-rc8-lz4-
    test-v8-build-15-78_amd64.deb linux-image-4.10.0-rc8-lz4-test-v8-build
    -15_4.10.0-rc8-lz4-test-v8-build-15-78_amd64.deb
# reboot
$ uname -r
4.10.0-rc8-lz4-test-v8-build-15
```

Listing 2.6: Installation des selbst gebauten Kernels

# 3 Implementierung von LZ4fast als Kernel-Modul

Der LZ4-Code kann nicht in unveränderter Form in den Linux-Kernel übernommen werden, wie wir im Verlauf der folgenden Seiten sehen werden. Als Basis für die Anpassungen wurde LZ4 Version 1.7.3 genutzt. Die Version kann zum Beispiel wie in Listing 3.1 bezogen werden.

```
$ git clone https://github.com/lz4/lz4 lz4
$ cd lz4
$ git checkout tags/v1.7.3 -b lz4-kernel-module
Switched to a new branch 'lz4-kernel-module'
```

Listing 3.1: Beziehen des LZ4-Sourcecodes über `git`

Linux, wie auch die meisten anderen Unix-Systeme, unterstützt Module in Form von Objektdateien (unter Linux mit der Dateiendung `.ko` für **k**ernel **o**bject), die während der Laufzeit des Systems in den Kernel geladen und auch wieder entfernt werden können. Ein solches Modul heißt unter Linux *Loadable Kernel Module (LKM)*. Ziel ist es, beispielsweise Unterstützung für neue Hardware in Form von Treibern herzustellen oder sonstige Funktionalität (wie im Fall von LZ4 zur Kompression von Daten) bei Bedarf bereitzustellen, statt sie dauerhaft in den Kernel zu integrieren[1].

Unter Linux unterscheiden sich Module, die direkt in den Kernel verbaut wurden (Konfigurations-Option **y**), mit ihm ausgeliefert werden (**m**) oder eigenständig kompiliert werden, kaum voneinander.
Die Anatomie eines einfachen Kernel-Moduls wollen wir im Folgenden betrachten.

## 3.1 Anatomie eines LKM

```
1 static int __init hello_init(void)
2 {
3   printk(KERN_INFO "[hello-mod] Hello world from the kernel!");
4
```

[1] vgl. Wikipedia, *Loadable kernel module — Wikipedia, The Free Encyclopedia*

```
5     return 0;
6   }
7
8   static void __exit hello_exit(void) { ... }
9
10  int hello_exported(void) { ... }
11  EXPORT_SYMBOL(hello_exported);
12
13  MODULE_LICENSE("GPL");
14  MODULE_AUTHOR("Sven Schmidt");
15  ...
16
17  module_init(hello_init);
18  module_exit(hello_exit);
```

Listing 3.2: Grobe Anatomie eines Loadable Kernel-Moduls für Linux

Das Beispiel aus Listing 3.2 wurde gekürzt; der komplette Code findet sich im Anhang. Wichtig sind zunächst die Meta-Informationen, hier die Zeilen 13 und 14. Über verschiedene Makros werden Informationen zum Modul definiert. Diese sind sichtbar, wenn etwa `modinfo lz4_compress.ko` aufgerufen wird. Dabei ist es nicht trivial, welche Daten angegeben werden. Eine fehlende oder Nicht-Open-Source-Lizenz beispielsweise wird zu einer Warnmeldung führen, wenn versucht wird, das Modul in den Kernel zu laden: „[module name] will taint the kernel“[2].

Es existieren zwei spezielle Funktionen: `hello_init` und `hello_exit`, die in den Zeilen 17 und 18 an besondere Funktionen übergeben werden. Sie dienen als Ein- beziehungsweise Ausstiegspunkt. Wird ein Modul in den Kernel geladen, üblicherweise über `modprobe [modulename]` oder `insmod [modulename]`, wird der Objektcode des Moduls in den Kernel-Space kopiert und `hello_init` aufgerufen. Eine solche Funktion kann benutzt werden, um notwendige Resourcen zu reservieren oder Structures zu initialisieren. Die exit-Funktion wird entsprechend verwendet, um beispielsweise Speicher freizugeben, sobald das Modul über `modprobe -r` oder `rmmod` entladen wird. LZ4 selbst benötigt jedoch weder Ein- noch Ausstiegspunkt.
Da es in C die Sichtbarkeits-Modifikatoren wie *public*, *private* oder *internal* nicht gibt, existiert das Makro `EXPORT_SYMBOL` (Zeile elf). In C repräsentieren Funktionsnamen die Adresse der Funktion. Über das Makro wird dem Kernel mtigeteilt, dass diese Adresse jedem Modul verfügbar gemacht werden soll, das den Header `lz4.h` inkludiert.

## 3.2 Erste Anpassungen an LZ4

### 3.2.1 Architektur & Definieren der Schnittstelle

Zu den Anpassungen, die am LZ4-Code also vorgenommen werden mussten, um ihn als Kernel-Modul zu qualifizieren, gehörte daher als erstes das **Definieren der zu exportierenden Funktionen**. Das sind im Speziellen `LZ4_compress_fast`, `LZ4_compress_default`, `LZ4_decompress_fast`,

[2] Project, *The Linux Kernel Module Programming Guide*.

`LZ4_decompress_safe` sowie `LZ4_compress_HC`.

Zweitens hat das vorherige LZ4-Modul eine **Trennung zwischen Kompression, Dekompression und HC-Algorithmus** vorgenommen. Dabei ist zu erwähnen, dass der **Dekompressor nicht als Kernel-Modul** gebaut wird (Konfigurations-Option **m**), sondern direkt in den Kernel integriert (**y**). Hintergrund ist, dass es möglich ist, das Kernel-Abbild selbst beim Kompilieren zu komprimieren, der Dekompressor also eine Sonderstellung einnimmt.
Der Code in der `lz4.c` musste also zunächst in Kompression und Dekompression getrennt werden. Dabei ließ sich keine definitive Grenze ziehen: Es existieren auch gemeinsam genutzte Funktionen, Makros und Konstanten. Im Kernel existiert bereits die Datei `linux/lib/lz4/lz4defs.h`, die architekturabhängige Definitionen und Makros beinhaltet, die nicht öffentlich sind, zum Beispiel Funktionen, die Unterscheidungen nach Endianess oder Wortbreite vornehmen. Dorthin wurden die meisten gemeinsamen Definitionen verschoben. Öffentliche Definitionen und die Funktions-Prototypen der exportierten Funktionen finden sich dagegen in `linux/include/linux/lz4.h`.

### 3.2.2 Nutzen von Kernel-Makros statt Portabilität

Der Upstream-Code muss viele **verschiedene Architekturen und Compiler** und Kombinationen davon unterstützen und **portierbar** sein. Im Linux-Kernel gilt das nicht: Unterstützt wird nur gcc für Linux. Von daher können einige Ersetzungen vorgenommen werden

- Code-Zeilen wie in Listing 3.3, die einen anderen Compiler als gcc ansprechen, können einfach entfernt werden.

```
1 #if defined(_MSC_VER) && defined(_WIN32_WCE)   /* Visual Studio for
      Windows CE does not support Hardware bit count */
2 #  define LZ4_FORCE_SW_BITCOUNT
3 #endif
```

Listing 3.3: Defines, die von anderen Kompilern als GCC abhängen, sind unnötig

- Definitionen wie in Listing 3.4, in denen Unterscheidungen vorgenommen werden, können zusammengefasst werden, indem nur die Branches für gcc übrigbleiben (Zeile zehn).

```
1 #ifdef _MSC_VER    /* Visual Studio */
2 #  define FORCE_INLINE static __forceinline
3 #  include <intrin.h>
4 #  pragma warning(disable : 4127)        /* disable: C4127:
      conditional expression is constant */
5 #  pragma warning(disable : 4293)        /* disable: C4293: too
      large shift (32-bits) */
6 #else
7 #  if defined(__GNUC__) || defined(__clang__)
8 #    define FORCE_INLINE static inline __attribute__((always_inline)
      )
```

```
#  elif defined(__cplusplus) || (defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L) /* C99 */)
#    define FORCE_INLINE static inline
#  else
#    define FORCE_INLINE static
#  endif
#endif  /* _MSC_VER */
```

Listing 3.4: Die Definition von `FORCE_INLINE` ist in LZ4 hochgradig portierbar, was im Kernel nicht notwendig ist

- Für viele Unterscheidungen, die nur der Portierbarkeit dienen, existieren sehr einfache Kernel-Makros. `FORCE_INLINE` lässt sich daher beispielsweise wie in Listing 3.5 neu definieren.

```
#define FORCE_INLINE __always_inline
```

Listing 3.5: Umgeschriebenes `FORCE_INLINE`-Makro für den Kernel

- Im Upstream-LZ4 (Listing 3.6) wird die Wortbreite anhand der Größe eines Pointers zu void getestet. Wieder ist das ein gutes Beispiel für portablen Code. Da wir im Abschnitt *Konfiguration des Kernels* jedoch festgelegt haben, ob wir ein 64- oder 32-Bit-System wollen, nutzen wir hier `CONFIG_64BIT`. Für die Endianess existieren ebenfalls Makros: `__LITTLE_ENDIAN` und `__BIG_ENDIAN`. Entsprechende Ersetzungen sehen wir in Listing 3.7.

```
static unsigned LZ4_64bits(void) { return sizeof(void*)==8; }

static unsigned LZ4_isLittleEndian(void)
{
    const union { U32 u; BYTE c[4]; } one = { 1 };   /* don't use static : performance detrimental */
    return one.c[0];
}
```

Listing 3.6: LZ4-Funktionen für Endianess und Wortbreite

```
if (LZ4_64bits()) ...
#if defined(CONFIG_64BIT)

if (LZ4_isLittleEndian())
#if defined(__LITTLE_ENDIAN)
```

Listing 3.7: Eine Abfrage von Endianess oder Wortbreite ist im Kernel bereits vorhanden

### 3.2.3 Speicher-Reservierung

Die fünfte, große Änderung umfasst das **Reservieren von Speicher für die Kompression**. Der Kompressor nutzt eine Struktur, die beispielsweise die Hash-Tabellen hält, für die Speicher reserviert werden muss. Im Upstream-LZ4 gibt es dafür mehrere Varianten. Ist `HEAPMODE` gesetzt, so reserviert LZ4 den notwendigen Speicher vom Heap, indem die User-Space-Funktion `void *calloc(size_t num, size_t size)` genutzt wird (versteckt hinter einem Makro `ALLOCATOR`). `calloc` reserviert *num* Speicherbereiche der Größe jeweils *size* und überschreibt jedes Byte mit 0. Per default ist `HEAPMODE` allerdings 0, sodass der Speicher durch die Deklaration von `LZ4_stream_t ctx` automatisch vom Stack reserviert wird. In Listing 3.8 sehen wir den exakten Code.

```
1  #if (HEAPMODE)
2      void* ctxPtr = ALLOCATOR(1, sizeof(LZ4_stream_t));    /* malloc-calloc
       always properly aligned */
3  #else
4      LZ4_stream_t ctx;
5      void* const ctxPtr = &ctx;
6  #endif
```

Listing 3.8: LZ4 reserviert den Speicher für die Kompression vom Heap (`calloc`) oder Stack

Beide Varianten sind im User-Space problemlos möglich, im Kernel aber nur bedingt. `sizeof(LZ4_stream_t)` entspricht 16 416 Bytes. In der Checkliste für die Einsendung von Code in den Kernel heißt es „any one function that uses more than 512 bytes on the stack is a candidate for change“[3]. Das heißt, diese Variante ist für den Kernel ungeeignet.

Wie verhält es sich mit der Reservierung vom Heap? `calloc` steht im Kernel gar nicht zur Verfügung. Seine Alternativen sind vor allem `kmalloc` und `vmalloc`. Speicher-Reservierung im Kernel folgt allerdings nicht denselben Regeln wie im User-Space, das gilt vor allem im Hinblick auf die Größe der zu allokierenden Bereiche und die verfügbaren Funktionen. Speicherverwaltung im Kernel ist ein komplexes Thema, für das ich an dieser Stelle auf meine Ausarbeitung zum Thema „Speicherverwaltung im Kernel“ verweise[4]. Der wichtigste Punkt ist, dass die User-Space-Funktionen (`calloc`, `malloc`) annähernd identisch funktionieren. Zwischen den Kernel-Funktionen existieren jedoch gravierende Unterschiede: `kmalloc`, zum Beispiel, reserviert physisch zusammenhängenden Speicher, dessen Allokations-Verhalten sich durch einen Flags-Parameter steuern lässt. So ist es auch möglich, Notfall-Speicher zuzugreifen oder für den Fall, dass die Allokation schiefläuft, vorzusorgen. `vmalloc` reserviert, im Vergleich dazu, nur virtuell zusammenhängenden Speicher, dessen Verhalten sich nicht weiter steuern lässt. Zusammengefasst ist es schwierig, für den Kernel einen Standard-Allokator vorzugeben.

Die vorherige LZ4-Version im Kernel hat`lz4_compress` daher umgeschrieben, sodass *void *wrk-*

[3] linuxtv, *Development: Linux Kernel patch submittal checklist*.
[4] Siehe unter (Schmidt, *Speicherverwaltung im Kernel*)

*mem* als zusätzlicher Parameter eingeführt wurde. Das ist insofern elegant, als für jede Nutzung der Funktion der Speicher so reserviert werden kann, wie es für den Use-Case erforderlich ist. Beispielsweise nutzt `linux/crypto/lz4.c` `vmalloc`, um in der init-Funktion des Moduls Speicher für den LZ4-Kontext zu reservieren. Es ist in dieser Variante aber auch denkbar, die Speicher-Reservierung zum Beispiel als dringend zu definieren, sodass der Kernel sie priorisiert betrachtet.
Tatsächlich hat das aktuelle LZ4 auch eine Funktion als dritte Variante der Speicher-Reservierung, die von vornherein einen externen Kontext entgegennimmt: `LZ4_compress_fast_extState`. Der Aufruf von `LZ4_compress_fast` wird bereits in der Upstream-Version auf diese Funktion umgeleitet (und `LZ4_compress_default` vorher auf `LZ4_compress_fast`), nachdem der Speicher reserviert wurde (eben vom Stack oder vom Heap). In der neuen Variante umfasst `LZ4_compress_fast` daher nur noch eine Zeile, die genaue Definition ist in Listing 3.9 zu sehen.

```
1 int LZ4_compress_fast(const char *source, char *dest, int inputSize,
2   int maxOutputSize, int acceleration, void *wrkmem)
3 {
4   return LZ4_compress_fast_extState(wrkmem, source, dest, inputSize,
5     maxOutputSize, acceleration);
6 }
7 EXPORT_SYMBOL(LZ4_compress_fast);
```

Listing 3.9: LZ4_compress_fast mit externem Speicher als Parameter

Der wrkmem-Pointer muss auf einen Speicherbereich zeigen, dessen Größe `LZ4_MEM_COMPRESS` (`sizeof LZ4_stream_t`) entspricht. Die Kompressions-Funktionen übernehmen die Initialisierung des Speichers als LZ4-Stream.

## 3.3 Test der Funktionalität

Das LZ4-Kernel-Modul besteht nach den Anpassungen aus den Dateien `lz4_compress.c`, `lz4_decompress.c`, `lz4hc_compress.c` sowie `lz4defs.h` und `lz4.h`. Um aus diesen Dateien nutzbare Module zu generieren, wird eine Makefile benötigt, deren Inhalt im einfachsten Fall aussieht wie in Listing 3.10.

```
1 obj-m += lz4_compress.o
2 obj-m += lz4hc_compress.o
3 obj-m += lz4_decompress.o
4
5 all:
6     make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
```

Listing 3.10: Makefile für das LZ4-Modul außerhalb des Kernels

Wichtig sind die **obj-m** in den Zeilen eins bis drei. Der letzte Buchstabe entspricht dem aus dem Abschnitt Konfiguration des Kernels: In den Makefiles des Kernels werden die Konfigurations-

Optionen an dieser Stelle übernommen, zum Beispiel `obj-$(CONFIG_LZ4_COMPRESS) += lz4_compress.o`. Der Aufruf von `make` im entsprechenden Verzeichnis generiert dann den Output in Listing 3.11.

```
$ make
make -C /lib/modules/4.10.0-rc8-lz4-test-v8-build-15/build/ M=/home/lz4-
    kernel-module modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-rc8-lz4-test-
    v8-build-15'
  CC [M]  /home/lz4-kernel-module/lz4_compress.o
  CC [M]  /home/lz4-kernel-module/lz4hc_compress.o
  CC [M]  /home/lz4-kernel-module/lz4_decompress.o
  Building modules, stage 2.
  MODPOST 3 modules
  CC      /home/lz4-kernel-module/lz4_compress.mod.o
  LD [M]  /home/lz4-kernel-module/lz4_compress.ko
  CC      /home/lz4-kernel-module/lz4_decompress.mod.o
  LD [M]  /home/lz4-kernel-module/lz4_decompress.ko
  CC      /home/lz4-kernel-module/lz4hc_compress.mod.o
  LD [M]  /home/lz4-kernel-module/lz4hc_compress.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-rc8-lz4-test-v8
    -build-15'
$ ls *.ko
lz4_compress.ko  lz4_decompress.ko  lz4hc_compress.ko
```

Listing 3.11: Aufruf von `make` zum Bauen der LZ4-Module

Die Module können dann jeweils über `insmod` in den Kernel geladen werden: Zum Beispiel `insmod lz4_compress.ko`. Zum Testen der Funktionalität habe ich ein einfaches Test-Modul geschrieben. Der Sourcecode kann Anhang C.2 entnommen werden. Eine entsprechende Makefile zeigt Listing C.3, ebenfalls im Anhang. Wichtige Eckdaten des Moduls:

- Das Testmodul kann sowohl LZ4 im Kernel, als auch das externe Modul testen (mehr dazu im Kapitel 6).
- Für den Test der Module außerhalb des Kernels war es notwendig, die Datei `Module.symvers` aus dem Verzeichnis, in dem die LZ4-Module gebaut werden, in das Verzeichnis des Test-Moduls zu kopieren, zum Beispiel in der Makefile über `cp ../Module.symvers .`. Ansonsten kann das Testmodul die über `EXPORT_SYMBOL` exportierten Funktionen nicht finden.
- Neben den unter Anatomie eines LKM beschriebenen Merkmalen, wurden für das Test-Modul **Modul-Parameter** genutzt. Ein Beispiel zeigt Listing 3.12.

```
1 static int acceleration = LZ4_ACCELERATION_DEFAULT;
2 module_param(acceleration, int, S_IRUSR | S_IWUSR | S_IRGRP |
      S_IROTH);
```

Listing 3.12: Der LZ4-Parameter acceleration und weitere Settings werden im Test-Modul als Kommandozeilen-Parameter akzeptiert

Derartige Parameter erlauben es, das Verhalten von Kernel-Modulen zu steuern. Das Makro `module_param` akzeptiert als ersten Parameter diejenige Variable, in die der Wert gespeichert werden soll; der zweite Parameter ist der Typ, der dritte umfasst die von `chmod` bekannten Zugriffsrechte. Hier kann nur der Besitzer schreiben, alle anderen können lesen. Eingebaut wurden die folgenden Parameter:

- **useHC**: Nutzung von LZ4Fast (0) oder LZ4HC (1)
- **acceleration**: Zur Nutzung mit LZ4Fast, setzt den acceleration-Faktor
- **cLevel**: Grad der Kompression für den HC-Algorithmus

Durch die Parameter ist es nicht notwendig, das Modul bei jeder Änderung neu zu kompilieren, es muss lediglich unter Angabe der Parameter neu in den Kernel geladen werden.

- Das Modul startet seine Arbeit in der init-Funktion, wo unter anderem Speicher für die Kompression und ein Output-Buffer reserviert wird. die Funktionsweise beschränkt sich darauf, durch Aufruf von `LZ4_compress_fast` beziehungsweise `LZ4_compress_HC` die Testdaten zu in den Output-Buffer zu komprimieren und anschließend über `LZ4_decompress_safe` wieder zu dekomprimieren.

- Das Modul gibt über `printk` Debug-Meldungen aus. Diese werden in Debian in `/var/log/kern.log` gespeichert und können beispielsweise über `tail /var/log/kern.log` oder `dmesg` ausgelesen werden.

- Nach der Kompression wird die Ausgabe nach `/tmp/lz4_compressed` geschrieben, nach der Dekompression nach `/tmp/lz4_decompressed`.

Das bedeutet, durch das Test-Modul kann im Speziellen verifiziert werden, dass die Ausgabe von `LZ4_decompress_safe` der Eingabe in die Kompressor-Funktionen entspricht und die Größe in Bytes identisch ist. Das Test-Modul hat in diesem Schritt des Projekts gezeigt, dass die LZ4-Module grundsätzlich funktionieren. Durch dieses Modul konnte im späteren Verlauf ein Problem mit dem HC-Algorithmus aufgedeckt werden, da die Ausgabegröße nicht der Eingabegröße entsprach.

# 4 | Einbau in Lustre

Das Ziel, das mit einem im Kernel verfügbaren, aktuellen LZ4 verfolgt wird, ist die Verwendung in Lustre. Allerdings dauert es zum einen lange, bis ein neuer Kernel rausgegeben wird (üblicherweise etwas mehr als zehn Wochen), und zum anderen noch länger, bis diese oder eine aktuellere Kernel-Version in eine Distribution wie etwa Debian integriert wird. Der Arbeitsbereich nutzt für die Entwicklung mit Lustre CentOS 7. CentOS ist eine Enterprise-Distribution, die auf sehr lange Support-Zyklen zur Verwendung beispielsweise für wissenschaftliche oder staatliche Einrichtungen ausgelegt ist[1]. Das heißt, bis der LZ4 enthaltende Kernel in CentOS integriert ist, kann viel Zeit vergehen.
Um trotzdem LZ4 in Lustre verwenden zu können, ist die Idee folgende: Wird Kompression in Lustre aktiviert, und existiert weder `lz4_compress` (aus Kernel 3.11) noch das aktuellere `LZ4_compress_fast`, dann wird ein in Lustre integriertes LZ4 gebaut. Dieses entspricht exakt dem in diesem Projekt entwickelten LZ4, sodass zu dem Zeitpunkt, in dem jede wichtige Distribution das neue LZ4 beinhaltet, das integrierte LZ4 einfach entfernt werden kann.

## 4.1 Vorbereitungen in Autoconf

Das Dateisystem Lustre selbst besteht aus einer Vielzahl an Kernel-Modulen, die über GNU Autoconf geabaut werden. Fast jeder Linux-Nutzer kennt die drei Schritte, die üblicherweise benötigt werden, um ein Programm aus seinen Quelldateien zu kompilieren: `./configure`, `make` und `make install`. In Lustre beginnt der Prozess mit einem zusätzlichen Schritt `sh autogen.sh`, der bestimmte Dateien (u.a. Makefiles) generiert. Die mit Abstand interessanteste `configure`-Phase nimmt dem Entwickler Arbeit dabei ab, seine Software für verschiedene Architekturen, Kernel-Versionen usw. zu portieren, indem automatische Tests durchgeführt und Makros zur Verwendung in Makefiles und in C (`#define`) zur Verfügung gestellt werden. Den Test, ob der Kernel `LZ4_compress_fast` zur Verfügung stellt, zeigt Listing 4.1. Der Test für das vorherige LZ4 mit `lz4_compress` sieht ähnlich aus. Beide werden in `lustre/lustre/autoconf/lustre-core.m4` definiert.

```
1 AC_DEFUN([LC_HAVE_KERNEL_LZ4_COMPRESS_FAST], [
2 have_kernel_lz4_compress_fast="no"
3 LB_CHECK_COMPILE([if Linux kernel has 'LZ4_compress_fast'],
4 LZ4_compress_fast, [
```

[1] Wikipedia, *Lustre (file system) — Wikipedia, The Free Encyclopedia*.

```
5   #include <linux/lz4.h>
6  ],[
7    LZ4_compress_fast(NULL, NULL, NULL, NULL, NULL, NULL);
8  ], [
9    have_kernel_lz4_compress_fast="yes"
10   AC_DEFINE(HAVE_KERNEL_LZ4_COMPRESS_FAST, 1,
11     [kernel has lz4_compress_fast])
12 ])
13 ])
```

Listing 4.1: Autoconf-Tests für `LZ4_compress_fast`

Zeile eins definiert das Test-Makro. Dieses wird, wie aus C bekannt, durch den eigentlichen Code (den Test) ersetzt, sofern der Precompiler das Makro findet.
Diese Art von Test läuft im Prinzip so, dass der entsprechende Header inkludiert wird (`linux/lz4.h`) und die Funktion mit `NULL` für jeden Parameter aufgerufen wird. Wenn die Funktion existiert, werden die Zeilen neun bis elf ausgeführt, sonst nicht. Dort wird das Makro `HAVE_KERNEL_LZ4_COMPRESS_FAST` definiert. Dieses können wir in C über `#if defined(...)` oder `#ifdef ...` nutzen.
Über die Tests kann also herausgefunden werden, ob ein bestimmtes Feature zur Verfügung steht. Um nun aber dem User selbst die Entscheidung darüber zu überlassen, ob er Kompression überhaupt benötigt, bietet Autoconf Parameter. Ebenfalls in `lustre/lustre/autoconf/lustre-core.m4` wird der in Listing 4.2 definiert.

```
1  AC_DEFUN([LC_CONFIG_COMPRESSION], [
2  AC_MSG_CHECKING([whether to enable compression])
3  AC_ARG_ENABLE([compression],
4    AC_HELP_STRING([--enable-compression], [enable compression]),
5    [], [enable_compression="no"])
6
7  AC_MSG_RESULT([$enable_compression])
8  ])
```

Listing 4.2: Autoconf-Parameter, ob Kompression in Lustre aktiviert werden soll

Zur Funktionsweise: Auch hier wird das Test-Makro in Zeile eins definiert. Über `AC_MSG_CHECKING` (Zeile zwei) wird eine Nachricht in die Konsole geschrieben: „checking whether to...“. Das Ergebnis des Tests wird über Zeile sieben ausgegeben (z.B. „yes“). Die dritte und vierte Zeile definiert, dass der Parameter das Präfix *enable* nutzt, selbst den Namen *compression* trägt und per Default `$enable_compression="no"` (keine Kompression) gelten soll. `AC_HELP_STRING` ist die Ausgabe in der Hilfe, wenn `./configure -h` aufgerufen wird.
Zu diesem Zeitpunkt ist es möglich, den Befehl `./configure --enable-compresison` aufzurufen. Allerdings ist das noch nutzlos, da die Makros nirgends benutzt werden.

In `lustre/config/lustre-build.m4` wird definiert, was genau gebaut werden soll. In unserem Fall gilt zusätzlich, dass Kompression nur unterstützt werden soll, wenn als zugrundeliegendes Dateisystem ZFS genutzt wird (statt Ldiskfs). Das heißt, es ist für die Frage, ob Kom-

pression aktiviert wird und ob ein eigenes LZ4 gebaut wird, entscheidend, ob der Parameter `--enable-compression` in der `configure`-Phase übergeben wurde, ob es LZ4 im Kernel gibt und ob ZFS aktiviert wurde. Das alles leistet der Code in Listing 4.3.

```
1  AC_DEFUN([LB_CONFIG_COMPRESSION], [
2
3  AS_IF([test "x$enable_compression" = xyes -a "x$enable_zfs" = xno], [
4    enable_compression="no"
5    AC_MSG_WARN([Can not enable compression: Only ZFS backend is supported
       by now!])
6  ])
7
8  AS_IF([test "x$enable_compression" = xyes],
9    [AC_DEFINE(COMPRESSION_ENABLED, 1, [compression is enabled])])
10
11 AM_CONDITIONAL(COMPRESSION, [test x$enable_compression = xyes])
12
13 LC_HAVE_KERNEL_LZ4_COMPRESS
14 LC_HAVE_KERNEL_LZ4_COMPRESS_FAST
15
16 AC_MSG_CHECKING([whether to build integrated LZ4 compression module])
17
18 build_lz4_module="no"
19 AS_IF([test x$enable_compression = xyes -a
       x$have_kernel_lz4_compress_fast = xno -a x$have_kernel_lz4_compress =
       xno], [
20   build_lz4_module="yes"
21 ])
22 AC_MSG_RESULT([$build_lz4_module])
23 AM_CONDITIONAL(LZ4, [test x$build_lz4_module = xyes])
24 ])
```

Listing 4.3: Lustre-Build-Tests für die Aktivierung von Kompression und das Bauen eines eigenen LZ4

Zeile eins definiert wieder das Makro. In den Zeilen drei bis sechs wird getestet, ob Kompression aktiviert wurde, während gleichzeitig ZFS nicht benutzt wird. In diesem Fall wird eine Warnung in die Konsole geschrieben und die Kompression deaktiviert. Sollte der Parameter in den Zeilen acht und neun noch gesetzt sein (sowohl Kompression als auch ZFS aktiviert), dann wird wiederrum ein C-Makro `COMPRESSION_ENABLED` definiert, da wir erst zu diesem Zeitpunkt sicher sind, dass Kompression genutzt werden soll. Zeile elf definiert ein Makro mit derselben Belegung, allerdings zur Verwendung in Makefiles. In den Zeilen 13 und 14 werden nun die Tests auf die Verfügbarkeit von LZ4 ausgeführt. Die folgenden Zeilen klären dann, ob das eingebaute LZ4 genutzt werden soll. Das ist der Fall, sofern Kompression aktiviert wurde, aber weder `lz4_compress` noch `LZ4_compress_fast` im Kernel verfügbar sind.

## 4.2 Einbau von LZ4

Für den Fall, dass LZ4 im Kernel existiert, sind wir an dieser Stelle fertig. Da das in CentOS 7 jedoch nicht der Fall ist, muss dafür gesorgt werden, dass auch wirklich ein eigenes LZ4 gebaut wird. Das können wir innerhalb von Makefiles über das Makro `LZ4` (siehe Zeile 23 in Listing 4.3) getestet werden. Als Ziel für LZ4 wurde der Ordner `lustre/lustre/utils/compression/lz4` gewählt. Der Ordner `compression` existierte vorher noch nicht. Lustres Makefiles sind rekursiv definiert, das heißt der jeweils übergeordnete Ordner definiert, welche Unterverzeichnisse wie gebaut werden. Daher beinhalten beide Ordner ebenfalls Makefiles und autoMakefiles. Der Aufbau ist nicht sonderlich kompliziert. `lustre/lustre/utils/compression/Makefile.in` beinhaltet lediglich die Zeile `@LZ4_TRUE@subdir-m += lz4`, die autoMakefile eine Entsprechung für AutoMake. Ist `LZ4` nicht definiert worden, ersetzt Autoconf `@LZ4_TRUE@` durch das Zeichen `#` für einen Kommentar: `#subdir-m += lz4`. Das heißt, die Zeile wird einfach auskommentiert und von `make` entsprechend ignoriert. Die Makefiles von LZ4 entsprechen im Wesentlichen denen im Kernel, mit wenigen Änderungen für die Nutzung mit Autoconf.

Listing 4.4 zeigt die Ausgabe von `configure` zu diesem Punkt.

```
# sh autogen.sh
...
# ./configure --enable-compression --disable-ldiskfs 2>&1 | tee configure
    .log | egrep --color -w "compression|lz4|lz4_compress|
    lz4_compress_fast" -i
checking whether to enable compression... yes
checking if Linux kernel has 'lz4_compress'... no
checking if Linux kernel has 'LZ4_compress_fast'... no
checking whether to build integrated LZ4 compression module... yes
config.status: creating lustre/utils/compression/Makefile
config.status: creating lustre/utils/compression/autoMakefile
config.status: creating lustre/utils/compression/lz4/Makefile
config.status: creating lustre/utils/compression/lz4/autoMakefile
```

Listing 4.4: Aufruf von `./configure` mit Ausgabe für den Build von Lustre mit LZ4

Die Ausgabe ist selbsterklärend; sie sieht aus wie erwartet. `--disable-ldiskfs` deaktiviert ldiskfs, damit ZFS als einziges Backend gebaut wird. Zusätzlich zu dem gezeigten Aufruf wurde noch getestet, wie sich die Ausgabe ändert, wenn beispielsweise `--enable-compression` weggelassen wird oder ZFS deaktiviert wird. In allen Fällen arbeitet Lustre wie gehofft.
Nach der `configure`-Phase folgt `make`, im Fall von CentOS werden rpm-Pakete generiert (statt beispielsweise deb für Debian). Den Aufruf zeigt Listing 4.5.

```
# make rpms -j4 2>&1 | tee make.log | egrep --color -w "compression|lz4|
    lz4_compress|lz4_compress_fast" -i
# ls *.rpm
```

```
kmod-lustre-2.9.0-1.el7.centos.x86_64.rpm              lustre-2.9.0-1.el7.
    centos.x86_64.rpm
lustre-iokit-2.9.0-1.el7.centos.x86_64.rpm
kmod-lustre-osd-zfs-2.9.0-1.el7.centos.x86_64.rpm      lustre-2.9.0-1.src.rpm
lustre-osd-zfs-mount-2.9.0-1.el7.centos.x86_64.rpm
kmod-lustre-tests-2.9.0-1.el7.centos.x86_64.rpm
lustre-debuginfo-2.9.0-1.el7.centos.x86_64.rpm
lustre-tests-2.9.0-1.el7.centos.x86_64.rpm
```

Listing 4.5: Aufruf von `make rpms`, um Lustre zu kompilieren

Im Anschluss werden die Pakete installiert. Hier über den Package-Manager von CentOS, yum.

```
# yum -y install *.rpm
# reboot
# modinfo lz4_compress
filename:       /lib/modules/3.10.0-327.36.3.el7.x86_64/extra/lustre/fs/
    lz4_compress.ko
description:    LZ4 compressor
license:        Dual BSD/GPL
rhelversion:    7.2
srcversion:     45CE776B68759E4973E12B2
depends:
vermagic:       3.10.0-327.36.3.el7.x86_64 SMP mod_unload modversions
```

Listing 4.6: Installation der generierten Lustre-Pakete mit yum

Ein Beispiel, wie das LZ4 nun benutzt werden kann, zeigt Listing 4.7. Dabei hatte ich eine Funktion `LZ4_compress_test` direkt in LZ4 eingebaut, um zu testen, ob Funktionen aus dem Modul aufrufbar sind. Der Code wurde in `lustre/lustre/llite/super25.c` in die init-Methode des Lustre-Moduls eingefügt, d.h. beim Aufruf von `modprobe lustre` wurde dieser ausgeführt.

```
1 // Test purposes, remove later!
2 #ifdef COMPRESSION_ENABLED
3 LZ4_compress_test();
4 #endif
```

Listing 4.7: Test des Aufrufs von LZ4-Funktionen in `lustre/lustre/llite/super25.c`

# 5 Einbringen von LZ4Fast in den Linux-Kernel

Zu diesem Zeitpunkt existierte ein erstes LZ4-Kernel-Modul, das für den aktuellen Kernel funktioniert, und Lustre ist dazu in der Lage, mit entsprechender Konfiguration LZ4 selbst zu bauen und C-Makros zum Testen der Verfügbarkeit von LZ4 und für die Aktivierung der Kompression zur Verfügung zu stellen. Das Einbringen des LZ4 in den Linux-Kernel selbst ist der letzte Schritt, der, wie sich herausgestellt hat, durchaus der umfangreichste war.

## 5.1 Pushen einer Änderung

Für gewöhnlich werden Änderungen an einer Software über die drei bekannten Aufrufe `git add`, `git commit` und `git push` entweder direkt vollzogen, oder ein *Pull Request* eingereicht, den Entwickler mit entsprechenden Berechtigungen im Repository dann prüfen. Im Linux-Kernel funktioniert der Prozess allerdings anders. Das hat auch damit zu tun, dass jeder Maintainer eines Subsystems des Kernels ein eigenes Repository unterhält, wo zunächst nur Änderungen an den von ihm verwalteten Subsystemen gesammelt, getestet und bewertet werden. Die Änderungen propagieren dann irgendwann nach oben, über mehrere Repositories und Maintainer hinweg bis zu Linus Torvalds (mehr dazu in Abschnitt 5.2). Auch aufgrund der Masse der Änderungen ist ein System über Pushes nicht effizient umsetzbar.

### 5.1.1 Patches & Patchsets

Um Änderungen verfügbar zu machen, ohne diese zu pushen, wird im Kernel auf eine grundlegende Idee zurückgegriffen: Ein Entwickler *beschreibt*, welche Änderungen an einer Version der Software notwendig sind, um seine Änderungen einzupflegen. Um diese Vergleiche zwischen einer Basis-Version und den lokalen Änderungen effizient zu automatisieren, existiert beispielsweise das Werkzeug `git diff`; eine Beispiel-Ausgabe enthält Listing 5.1.

```
$ git diff lib/lz4/lz4_compress.c
diff --git a/lib/lz4/lz4_compress.c b/lib/lz4/lz4_compress.c
index cc7b6d4..0f683b1 100644
--- a/lib/lz4/lz4_compress.c
+++ b/lib/lz4/lz4_compress.c
```

```
@@ -157,7 +157,7 @@ static const BYTE *LZ4_getPositionOnHash(
        }
 }

-static FORCE_INLINE const BYTE *LZ4_getPosition(
+static const BYTE *LZ4_getPosition(
        const BYTE *p,
        void *tableBase,
        tableType_t tableType,
```

Listing 5.1: Benutzung von `git diff`, um lokale Änderungen nachzuvollziehen

Anhand dieser Ausgabe kann exakt der Stand hergestellt werden, den der Entwickler durch seine Änderungen erzeugt hat, indem im Beispiel die mit Minus markierte Zeile durch die ersetzt wird, der ein Plus vorangestellt ist. Die Nutzung dieser *Diffs* ist die Grundidee, wenn es darum geht, Änderungen für den Linux-Kernel einzureichen. Dabei werden Diffs per E-Mail verschickt und, am anderen Ende, von den Maintainern und Reviewern, in ihre Kopie des Sourcetrees eingebracht. Diese Diffs haben ein bestimmtes Format und werden ***Patch*** genannt. Eine Serie, die aus mehreren, aufeinander aufbauenden Patches besteht, wird ***Patchset*** genannt.
Offensichtlich ist es nicht effizient, längere Patches per Hand anzuwenden. Zur Vereinfachung dieses Tasks aus *Patch vorbereiten* und *Patch anwenden* existieren die Werkzeuge `git format-patch` zur Vorbereitung eines Patches oder einer -serie und `git am`, um einen Patch anzuwenden. Dabei entspricht jeder Patch genau einem Commit in das lokale Repository. Wird dann `git am` angewandt, so werden nicht nur die Änderungen übernommen, sondern auch der zugehörige Commit (nachvollziehbar über `git log`).

Um Änderungen als Patches zu versenden, werden diese zunächst commited. Dabei entspricht die erste Zeile der Commit-Beschreibung am Ende dem Dateinamen des Patches, wobei es eine Zeichenbegrenzung gibt, Leerstellen durch Bindestriche ersetzt werden und alle relevanten Informationen (etwa das Subsystem) enthalten sein sollten. Im Beispiel in Listing 5.2 wurden bereits fünf Commits getätigt.
Der Switch -s fügt folgende Zeile ans Ende der Commit-Nachricht hinzu: `Signed-off-by: Sven Schmidt <4sschmid@informatik.uni-hamburg.de>`. Mit dieser Zeile unterschreibt der Entwickler das so genannte *Developer's Certificate of Origin*[1], womit er unter anderem bestätigt, dass es sich bei seinem Patch um Open-Source-Software handelt und er berechtigt ist, diesen einzusenden.

```
$ git add lib/lz4/ include/linux/lz4.h
$ git commit -s -v
$ git format-patch -o patches -5 HEAD
$ cd patches
$ ls
0001-lib-Update-LZ4-compressor-module.patch
```

[1] siehe https://kernel.org/doc/html/latest/process/submitting-patches.html#developer-s-certificate-of-origin-1-1

```
0002-lib-decompress_unlz4-Change-module-to-work-with-n.patch
0003-crypto-Change-LZ4-modules-to-work-with-new-LZ4-mo.patch
0004-fs-pstore-fs-squashfs-Change-usage-of-LZ4-to-work.patch
0005-lib-lz4-Remove-back-compat-wrappers.patch
```

Listing 5.2: Vorbereiten eines Patches mit `git format-patch`

Mit `git format-patch` werden dann fünf Patches erzeugt. `-o` gibt dabei an, *in welches Verzeichnis* die Patches zu generieren sind (hier der Ordner `patches`). `-5` ist die Anzahl der Patches und `HEAD` ist der Commit, von dem ausgegangen werden soll; in diesem Fall also die letzten Änderungen.

Die Patchserie kann dann verschickt werden. Am anderen Ende wird als Gegenstück `git am` angewandt.

```
$ git am 0003-crypto-Change-LZ4-modules-to-work-with-new-LZ4-mo.patch
$ git log
commit 28d41ca1ac1d1472d8a108c6e742e12fe1141c73
Author: Sven Schmidt <4sschmid@informatik.uni-hamburg.de>
Date:   Tue Feb 14 15:56:16 2017 +0100

    crypto: Change LZ4 modules to work with new LZ4 module version

    Update the crypto modules using LZ4 compression as well as the test
    cases
    in testmgr.h to work with the new LZ4 module version.

    Signed-off-by: Sven Schmidt <4sschmid@informatik.uni-hamburg.de>
...
```

Listing 5.3: Anwenden eines Patches mit `git am`

Dabei ist darauf zu achten, dass Patches in der richtigen Reihenfolge angewandt werden. In Listing 5.3 sehen wir, dass nach der Ausführung von `git am` gefolgt von `git log` tatsächlich der zum Patch gehörige Commit vorhanden ist. Im Endeffekt haben dann (im Hinblick auf die geänderten Dateien) der Tree des Entwicklers und der des Testers denselben Stand.

### 5.1.2 Patches per E-Mail versenden

Um nun Patches per E-Mail zu versenden, kommt ein drittes Tool zum Einsatz: `git send-email`. Der Befehl akzeptiert beispielsweise Parameter wie `--to`, `--cc` oder `--subject` und erwartet die Angabe von Patch-Dateien. Zunächst ist jedoch zu klären, *wer* die Patches überhaupt bekommen soll und *wie* sie zu versenden sind, das heißt über welchen Server und mit welchen Zugangsdaten.

Das *Wie* ist zu lösen, indem Zugangsdaten zu einem Mail-Server in der Datei `linux/.git/config` hinterlegt werden. Das kann beispielsweise aussehen wie in Listing 5.4. Wichtig ist, dass die Informationen der `git`-Konfigurationen mit den E-Mail-Daten übereinstimmen. Ist das nicht der

Fall, wird der Patch durch eine zusätzliche Zeile *From:* ergänzt, der den Original-Autor angibt. Das ist der, der den Patch über `git format-patch` erzeugt hat.

```
$ nano .git/config
[user]
  email = 4sschmid@informatik.uni-hamburg.de
  name = Sven Schmidt
[sendemail]
  smtpuser = 4sschmid
  smtpserver = mailhost.informatik.uni-hamburg.de
  smtpencryption = tls
  smtpserverport = 587
  supresscc = self
  smtppass = password
```

Listing 5.4: `.git/config` enthält die Konfiguration für den SMTP-Server

Die Frage, *wer* die E-Mails bekommen soll, ist nicht offensichtlich. Heutzutage werden Patches nicht mehr direkt an Linus Torvalds gesendet, sondern an die Subsystem-Maintainer und Entwickler, die am betroffenen Code gearbeitet haben. Um herauszufinden, wer genau zuständig ist, liegt im Linux-Sourcetree das Script `linux/scripts/get_maintainer.pl`:

```
$ perl scripts/get_maintainer.pl < 0001-lib-Update-LZ4-compressor-module.
    patch
Bongkyu Kim <bongkyu.kim@lge.com> (commit_signer:1/1=100%,authored
    :1/1=100%,added_lines:2/2=100%,removed_lines:2/2=100%)
Andrew Morton <akpm@linux-foundation.org> (commit_signer:1/1=100%)
Rui Salvaterra <rsalvaterra@gmail.com> (commit_signer:2/2=100%,authored
    :2/2=100%,added_lines:13/13=100%,removed_lines:12/12=100%)
Greg Kroah-Hartman <gregkh@linuxfoundation.org> (commit_signer:2/2=100%)
Sergey Senozhatsky <sergey.senozhatsky@gmail.com> (commit_signer
    :2/2=100%)
linux-kernel@vger.kernel.org (open list)
```

Listing 5.5: Ausführung von get_maintainer.pl, um die zuständigen Maintainer zu erhalten

In der Ausgabe können wir einige Unterschiede zwischen den Personen erkennen.

- Bongkyu Kim und Rui Salvaterra haben aktiv an den Dateien gearbeitet (Zeilen eingefügt oder gelöscht).
- Greg Kroah-Hartman habe ich bereits als Maintainer des Stable-Linux vorgestellt, er hat in der Vergangenheit Patches abgesegnet (sign-off)
- Dasselbe gilt für Andrew Morton, bei dem es sich um den zuständigen Maintainer für `linux/lib` handelt.
- Wird in der aktuellen Mainline-Version der Befehl wiederholt, so taucht auch mein Name in der Liste auf, da ich im Rahmen dieses Projekts Änderungen vollzogen habe. Das heißt,

zukünftig werde auch ich E-Mails zu LZ4-Änderungen im Kernel bekommen.

- Neben den eigentlichen Maintainern sind immer auch Mailing-Listen enthalten. Hier *linux-kernel* und *linux-crypto*. Dadurch bekommen Entwickler die Möglichkeit, auf die Patchserie zu reagieren, die nicht direkt angesprochen sind, aber möglicherweise trotzdem ein Interesse daran haben. Die Liste *linux-kernel* ist die öffentliche Liste, die unter `https://lkml.org` archiviert wird. Die Diskussion zum LZ4-Modul findet sich beispielsweise unter `https://lkml.org/lkml/2016/12/20/511` nachvollzogen werden.

Wird der Vorgang aus Listing 5.5 für alle Patches wiederholt, so sieht der resultierende Aufruf von `git send-email` aus wie in Listing 5.6.

```
$ git send-email --compose \
--to akpm@linux-foundation.org \
--cc bongkyu.kim@lge.com \
--cc rsalvaterra@gmail.com \
--cc sergey.senozhatsky@gmail.com \
--cc gregkh@linuxfoundation.org \
--cc linux-kernel@vger.kernel.org \
--cc herbert@gondor.apana.org.au \
--cc davem@davemloft.net \
--cc linux-crypto@vger.kernel.org \
--cc anton@enomsg.org \
--cc ccross@android.com \
--cc keescook@chromium.org \
--cc tony.luck@intel.com \
./patches/*.patch \
--subject="[PATCH 0/5] Update LZ4 compressor module"
```

Listing 5.6: Aufruf von `git send-email` mit allen Empfängern

Durch Angabe von `--compose` wird eine nullte E-Mail als Einleitung geöffnet, in der bei Patchsets erläutert werden kann, was genau die Idee der Serie ist, welches Ziel sie verfolgt, was sie verbessert, welches Problem sie löst, kurz: warum die Maintainer sie in den Kernel aufnehmen sollten. Die durch `--subject` definierte Zeile ist der Betreff der Einleitungs-E-Mail.

## 5.2 Übersicht: Der Weg in den Kernel

Hat ein Entwickler den Stand erreicht, dass er seine Änderungen vollzogen hat, es keine (offensichtlichen) Build-Fehler gibt und er den Patch oder die -serie per E-Mail an die zuständigen Maintainer und Listen versendet hat, so werden nach einiger Zeit Reaktionen zum Patch als Antwort auf den gesamten Thread kommen.

Die ersten Reaktionen der Kernel-Entwickler, in meinem Fall von Greg Kroah-Hartman, hatten vor allem zum Ziel, die Frage zu klären, warum die Patchserie in den Kernel aufgenommen werden soll (welchen Vorteil sie bietet, welches Problem sie löst...) und um generelle Anforderungen zu klären, zum Beispiel, dass die Patchserie dem Kernel-Coding-Style genügt, korrekte

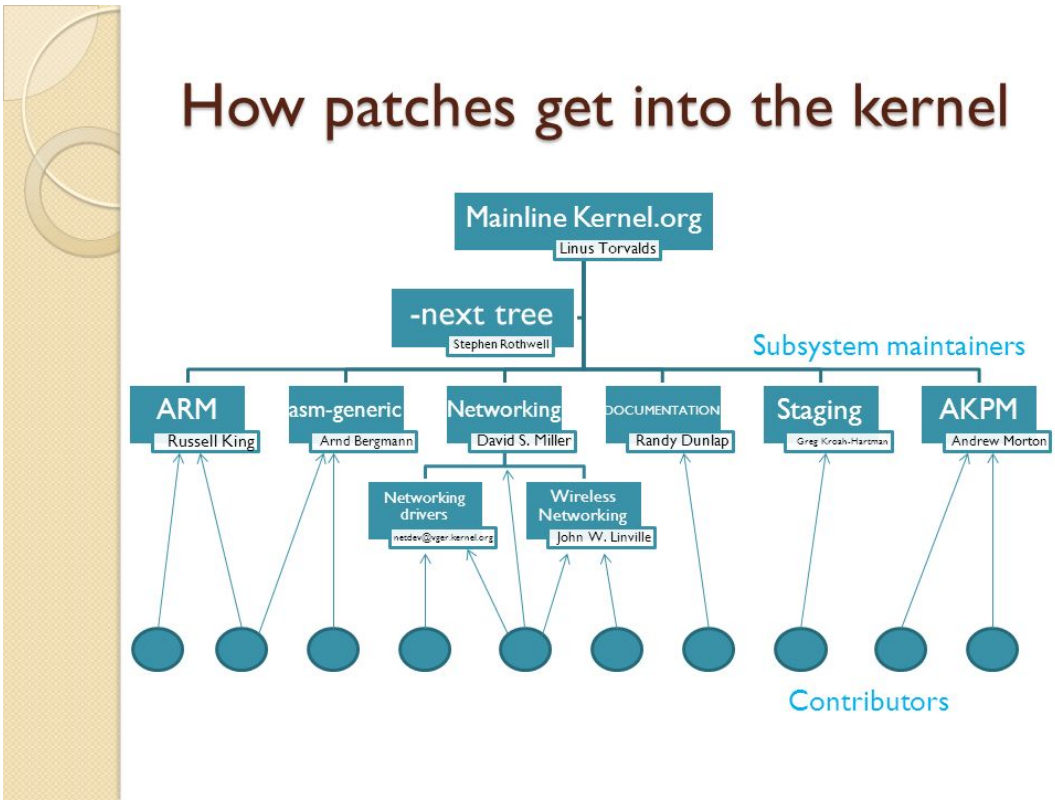


Abbildung 5.1: Der Weg eines Patches über die Subsysteme bis zu Linus Torvalds (Xuetao, *Linux Kernel Development: Getting started*)

Copyright-Notes aufweist, die Patches sinnvoll gesplittet sind usw. In Abbildung 5.1 ist dieser Teil des Prozesses die untere Ebene.
Je nachdem, zu welchem Teil des Kernels die Patchserie logisch gehört, nimmt der zuständige Subsystem-Maintainer sie im nächsten Schritt in seinen Tree auf, sofern der Patch nicht vorher abgelehnt wurde. Im Fall von LZ4 ist das der **-mm-Tree** von **Andrew Morton** (**AKPM** in Abbildung 5.1). Obwohl **-mm** historisch für **M**emory **M**anagement steht, umfasst sein Tree mittlerweile auch Patches, die sonst keine Zugehörigkeit haben[2]. Jeder Kernel-Verantwortliche, durch dessen Hände ein Patch geht, fügt ebenfalls ein Sign-Off hinzu: `Signed-off-by: Andrew Morton ...`. Am Ende des Prozesses trägt der Patch dann auch die *Unterschrift* von Linus Torvalds. Jeder Maintainer übernimmt dafür die Verantwortung dafür, dass der Patch *okay* ist.

Durch die Aufnahme in einen Subtree erreicht die Patchserie größere Aufmerksamkeit, sodass mehr, vor allem technisch anspruchsvolleres Feedback, folgt. Es geht nicht mehr darum, zu klären, **ob** der Patch aufgenommen wird, sondern **wie**, in welcher Form.

Im nächsten Schritt werden die Patches in den Tree **-next** von **Stephen Rockwell** aufgenommen. In der Kernel-Entwicklung öffnet **Linus Torvalds** nach jedem Kernel-Release ein zweiwöchiges Fenster, während dessen er Änderungen in seinen eigenen Tree aufnimmt. Der **-next**-Tree ist im Grunde ein Abbild davon, wie der Mainline-Kernel von Torvalds am Ende

[2]Wikipedia, *Mm tree — Wikipedia, The Free Encyclopedia*.

dieses Fensters aussehen soll und wird täglich aktualisiert. Obwohl die Graphik etwas anderes suggieriert, existieren keineswegs nur so wenig Subtrees. In der E-Mail zum **-next**-Tree vom 23. Februar schrieb Rockwell, er merge derzeit 253 Subtrees nach -next[3]. Dabei basiert das Mergen auf einem Vertrauens-System: Der Maintainer der jeweils darunterliegenden Ebene ist verantwortlich für seine Patches, sodass ein höhere Maintainer sie ohne viele Nachfragen übernimmt. Am Ende des Merge-Window, das Linus nach zwei Wochen schließt, startet die etwa achtwöchige **Stabilisierungs-Phase**, während der Release Candidates der nächsten Kernel-Version herausgegeben, getestet und angepasst werden. Nach Ablauf dieser Phase gibt Torvalds die nächste Version heraus und der Prozess beginnt von neuem mit dem Öffnen des nächsten Mege-Window. Die Entwicklung auf den unteren Ebenen schreitet währenddessen fort.

## 5.3 Anpassungen am LZ4-Code

Im Laufe des Review-Prozesses gab es einige Änderungen am Code. Der Großteil bezieht sich auf solche Punkte, die im Abschnitt 3.2 bereits erwähnt wurden, im Allgemeinen also Besonderheiten des Kernels im Vergleich zum User-Space. Im Speziellen gab es die im Folgenden diskutierten Änderungen.

### 5.3.1 Patchsets: Fallstricke

Oft passiert es, dass bei Versionsverwaltungs-Systemen wie `git` eine Reihe von Commits zusammen einen konsistenten Zustand hinterlassen, ein einzelner Commit der Serie allerdings Fehler einführt, die durch einen der folgenden behoben werden. Das sind beispielsweise Funktionen, die entfernt werden.
In der Kernel-Entwicklung ist es aufgrund der Frequenz und Anzahl an Änderungen besonders wichtig, dass jeder Patch einer Serie ohne die nachfolgenden Patches angewendet werden kann, ohne dass Build-Fehler auftreten. Dadurch kann tatsächlich fehlerhafter Code effizienter identifiziert werden. Dafür wird das Tool `git bisect` genutzt, bei dem die zugrunde liegende Idee darin besteht, einen *schlechten*, aktuellen Commit und einen *guten*, älteren Commit zu benennen und die dazwischen getätigten Commits zu testen.
Die Fehler, die bei mir durch automatische Build-Bots gemeldet wurden, entstanden also dadurch, dass Änderungen in einer ungünstigen, das heißt gegen diese ***Bisectibility*** verstoßenden, Reihenfolge getätigt wurden. Ein Einführen des neuen LZ4 in Patch eins führt nach alleiniger Anwendung des Patches zu einem Build-Fehler, da beispielsweise `crypto/lz4.c` noch die Funktion `lz4_compress` nutzt, die entfernt wurde.
Eine Lösung, die konform mit dem Bisectibility-Gedanken ist: In Patch eins wird LZ4 aktualisiert, aber es wird beispielsweise die Funktion `lz4_compress` wie in Listing 5.7 mit in den Code aufgenommen.

```
1 int lz4_compress(const unsigned char *src, size_t src_len, unsigned char
      *dst,
```

[3] https://lkml.org/lkml/2017/2/23/1

```
2     size_t *dst_len, void *wrkmem) {
3     *dst_len = LZ4_compress_default(src, dst, src_len,
4       *dst_len, wrkmem);
5
6     /*
7      * Prior lz4_compress will return -1 in case of error
8      * and 0 on success
9      * while new LZ4_compress_fast/default
10     * returns 0 in case of error
11     * and the output length on success
12     */
13    if (!*dst_len)
14      return -1;
15    else
16      return 0;
17  }
18  EXPORT_SYMBOL(lz4_compress);
```

Listing 5.7: Rückwärts-Kompabilitäts-Wrapper für `lz4_compress`

Das heißt, ein Wrapper für die alten Funktionen leitet den Aufruf an die neuen weiter, um Rückwärtskompabilität zu gewährleisten. In den Patches zwei bis vier werden dann die Stellen angepasst, die solche Funktionen noch nutzen. In Patch fünf werden die Wrapper dann entfernt. Insgesamt benötigt diese Methode also einen zusätzlichen Commit.

Um derartige Probleme in der Reihenfolge aufzudecken, ist es nicht hinreichend, den Sourcetree in dem Zustand zu bauen, in dem er bereits den vollständig angepassten Kernel beinhaltet. Eine mögliche Lösung ist es, zwei Sourcetrees zu unterhalten: Einen zum entwickeln, einen um die Patches wie in Listing 5.8 nacheinander anzuwenden und zu bauen.

```
$ git format-patch -n5 HEAD -o patches
$ cp patches/*.patch ../linux-test/patches
$ cd ../linux-test
$ git am patches/0001-lib-Update-LZ4-compressor-module.patch
$ make ...
$ git am patches/0002-lib-decompress_unlz4-Change-module-to-work-with-n.
    patch
$ make ...
...
```

Listing 5.8: Build-Prozess, um Bisectibility zwischen Patches zu gewährleisten

### 5.3.2 Code-Style

Im Linux-Kernel existieren klare Regeln für den Code-Style, die zum Beispiel in der Dokumentation beschrieben werden. Um die Patches auf Konformität mit dem Kernel-Style zu testen,

existiert das Skript `linux/scripts/checkpatch.pl`, dessen Nutzung und beispielhafte (gekürzte) Ausgabe Listing 5.9 zeigt.

```
$ perl scripts/checkpatch.pl patches/*
ERROR: "foo* bar" should be "foo *bar"
#3555: FILE: lib/lz4/lz4hc_compress.c:563:
+int LZ4_compress_HC_continue (LZ4_streamHC_t* LZ4_streamHCPtr, const
    char* source, char* dest, int inputSize, int maxOutputSize)

WARNING: space prohibited between function name and open parenthesis '('
#3555: FILE: lib/lz4/lz4hc_compress.c:563:
+int LZ4_compress_HC_continue (LZ4_streamHC_t* LZ4_streamHCPtr, const
    char* source, char* dest, int inputSize, int maxOutputSize)

WARNING: please, no spaces at the start of a line
#3557: FILE: lib/lz4/lz4hc_compress.c:565:
+    if (maxOutputSize < LZ4_compressBound(inputSize))$
...
```

Listing 5.9: Prüfen eines Patches über das Skript `checkpatch.pl`

In der ersten Patch-Version, die ohne größere Anpassungen den LZ4-Style beibehalten hat, existierten 1303 Fehler und 1357 Warnungen auf 3507 Zeilen. Häufig war vor allem die Position des Indirection-Operators * betroffen: `const char* ptr` (LZ4) versus `const char *ptr` (Kernel). Das zählt als Fehler, was am folgenden Beispiel nachvollziehbar ist:

- `uint8_t *a, *b, *c`: Die Variablen a, b und c haben den Typ *Pointer auf uint8_t*
- `uint8_t* a, b, c`: **Nur a** ist hat den Typ *Pointer auf uint8_t*, b und c haben nur den Typ `uint8_t`

Weitere, häufig gemeldete Fehler:

- Zeilen, deren Länge 80 Zeichen überschreitet (Warnung)
- Fehlende Leerstellen hinter Schlüsselwörtern wie `for` oder `if`: `for(...)` vs. `for (...)` (Warnung)
- Leerzeichen vor und hinter binären Operatoren: `a+2` vs. `a + 2` (Warnung)
- Der Kernel nutzt Tabs statt Spaces zur Einrückung (Fehler)
- `unsigned` als Abkürzung für `unsigned int` (Warnung)
- Conditionals mit nur einer Zeile: `if (foo) length++;` sollte einen Zeilenumbruch beinhalten (Fehler)
- Nach Deklarationen von Variablen wie `int i` muss eine Leer**zeile** folgen (Warnung)
- Die öffnenden, geschweiften Klammern bei Funktionen gehören in die nächste Zeile (selbe Zeile: Fehler), bei switch-Statements allerdings in dieselbe Zeile (nächste Zeile: Fehler)

- Die Nutzung von `typedef` (Warnung)
- `EXPORT_SYMBOL` soll der exportierten Funktion direkt anschließen (Warnung)

In den Listings 5.10 (LZ4-Style) und 5.11 (Kernel-Style) ist beispielhaft derselbe Code aus dem LZ4HC-Algorithmus zu sehen.

```
            const BYTE* const match = dictBase + matchIndex;
            if (LZ4_read32(match) == LZ4_read32(ip)) {
                size_t mlt;
                const BYTE* vLimit = ip + (dictLimit - matchIndex);
                if (vLimit > iLimit) vLimit = iLimit;
                mlt = LZ4_count(ip+MINMATCH, match+MINMATCH, vLimit) +
MINMATCH;
                if ((ip+mlt == vLimit) && (vLimit < iLimit))
                    mlt += LZ4_count(ip+mlt, base+dictLimit, iLimit);
                if (mlt > ml) { ml = mlt; *matchpos = base + matchIndex;
}   /* virtual matchpos */
            }
```

Listing 5.10: Ein Beispiel für den LZ4-Coding-Style

```
    const BYTE * const match = dictBase + matchIndex;

    if (LZ4_read32(match) == LZ4_read32(ip)) {
      size_t mlt;
      const BYTE *vLimit = ip
        + (dictLimit - matchIndex);

      if (vLimit > iLimit)
        vLimit = iLimit;
      mlt = LZ4_count(ip + MINMATCH,
        match + MINMATCH, vLimit) + MINMATCH;
      if ((ip + mlt == vLimit)
       && (vLimit < iLimit))
        mlt += LZ4_count(ip + mlt,
          base + dictLimit,
          iLimit);
      if (mlt > ml) {
        /* virtual matchpos */
        ml = mlt;
        *matchpos = base + matchIndex;
      }
    }
```

Listing 5.11: Ein Beispiel für den Kernel-Coding-Style

Der Kernel-Style ist „super simple“[4]. Anhand der Beispiele können wir insbesondere zwei Kategorien identifizieren:

1. Einige Regeln sind dazu gedacht, ungewollte Fehler im Kernel zu vermeiden. Ein Beispiel dafür ist die Position des Indirection-Operators. Aber auch `unsigned` als Abkürzung für `unsigned int` oder die Vermeidung von `typedef`: Jeder versteht, was mit `struct crypto_ctx *foo` gemeint ist, während `crypto_t *foo` nicht so eindeutig ist. Auch die Regel, dass `EXPORT_SYMBOL` der exportieren Funktion direkt folgen soll, ergibt Sinn, sobald ein Reviewer vor der Frage steht, *ob* die Funktion exportiert wird oder nicht.

2. Andere Regeln zielen auf die Lesbarkeit des Codes und einheitlichen Code ab: Tabs zur Einrückung, Leerzeilen hinter Deklarationen, Leerstellen bei binären Operatoren etc. Der Richtwert von 80 Zeichen für die Länge einer Zeile dient beispielsweise der Lesbarkeit „on a 80-character terminal screen“[5].

Insgesamt ist festzuhalten, dass der LZ4-Code sehr schlecht lesbar ist. Warum ein lesbarer Coding-Style wichtig ist und, in Folge dessen, die genannten Regeln sinnvoll sind, demonstriert das folgende Beispiel. Listing 5.12 zeigt eine Zeile aus dem LZ4HC, Listing 5.13 dieselbe Zeile nach dem Anpassen an den Kernel-Coding-Style. Durch den schlecht lesbaren LZ4-Code ist nicht sofort ersichtlich, dass die Zeile `*(*op)++ = (BYTE)len;` **nicht zum `for`-Loop** gehört. Das hat im Zuge der Styling-Anpassungen dazu geführt, dass LZ4HC nicht mehr korrekt funktionierte.

```
1 if (length>=(int)RUN_MASK) { int len; *token=(RUN_MASK<<ML_BITS); len =
    length-RUN_MASK; for(; len > 254 ; len-=255) *(*op)++ = 255;  *(*op)++
    = (BYTE)len; }
```

Listing 5.12: Beispiel für den schlecht lesbaren Coding-Style in LZ4, dessen Refactoring Fehler eingeführt hat

```
1  if (length >= (int)RUN_MASK) {
2          int len;
3
4          *token = (RUN_MASK << ML_BITS);
5          len = length - RUN_MASK;
6
7          for (; len > 254; len -= 255) {
8                  *(*op)++ = 255;
9                  *(*op)++ = (BYTE)len;
10         }
11 }
```

Listing 5.13: Ein Fehler, der durch das Refactoring des LZ4-Coding-Styles entstanden ist

[4] community, *Linux kernel coding style*.
[5] community, *Linux kernel coding style*.

### 5.3.3 Linux' crypto-API und `crypto/testmgr`

Neben den oberflächlichen Anpassungen, wozu der Code-Style, Copyrights, Lizenzen und exportierte Funktionen gehören, existieren im Linux-Kernel bereits Module, die auf LZ4 zurückgreifen, an denen Anpassungen vorgenommen werden müssen. Dazu zählt das bereits erwähnte ZRAM. Allerdings greift ZRAM nicht auf LZ4 selbst (`linux/lib/lz4`) zurück, sondern auf `linux/crypto/lz4`. `linux/crypto` beinhaltet die kryptographische Schnittstelle des Linux-Kernels und eine Sammlung verschiedener Algorithmen: RSA, MD4 und MD5, die SHA-Familie, Deflate, LZO und eben LZ4, um nur einige zu nennen.
Um Benchmarks für LZ4 vorzunehmen, eignet sich ZRAM hervorragend. Details dazu erläutere ich im Abschnitt über Benchmarks. Dafür müssen zunächst `crypto/lz4` und ZRAM in den Kernel geladen werden und LZ4 als Algorithmus für ZRAM registriert werden. Die einzelnen Schritte und die Ausgabe von `dmesg` sind in Listing 5.14 zu sehen.

```
# modprobe -a lz4 zram
# echo lz4 > /sys/block/zram0/comp_algorithm
# dmesg
alg: comp: Compression test 1 failed for lz4 ...
zcomp: Can't allocate a compression stream
```

Listing 5.14: Versuch, ZRAM mit LZ4 in den Kernel zu laden

Wichtig: lz4 ist der Name des Moduls `crypto/lz4.c`, das von mir gebaute Kompressor-Modul heißt `lz4_compress`. Durch Verwendung von `modprobe` wird dieses allerdings ebenfalls geladen, da es eine Abhängigkeit von `crypto/lz4` darstellt. Wie wir sehen, werden Fehlermeldungen geworfen, einmal von ZRAM selbst (zcomp) und eine weitere Meldung, die zuvor auftritt. Das sind ärgerliche Fehler, da sie erst dann nachvollzogen werden können, wenn manuelle Tests getätigt werden. Beim Bauen des Kernels tritt kein Fehler auf.
Sucht man die zweite Fehlermeldung ohne konkrete Werte (es handelt sich um Platzhalter), findet man die Datei `linux/drivers/block/zram/zcomp.c`. Ein Folgen des Aufruf-Pfades verrät, dass der zweite Fehler direkt durch den ersten verursacht wird: ZRAM versucht, eine Instanz von LZ4 von der crypto-API zu bekommen. Da dies fehlschlägt, kann ZRAM LZ4 nicht verwenden.

Der erste Fehler ist interessanter: Tatsächlich existiert innerhalb von crypto `linux/crypto/testmgr.c`, eine Test-Utility für die angebotenen Algorithmen, die ihre korrekte Funktion sicherstellt. Das funktioniert über Erwartungs-Tests: Wird der LZ4-Kompressor mit einer bestimmten Eingabe aufgerufen, so wird eine bestimmte Ausgabe mit bestimmter Länge erwartet. Der Test wird aufgerufen, sobald LZ4 angefragt wird. Für diese Vektoren aus Eingabe, Ausgabe und Länge muss ein Eintrag in `linux/crypto/testmgr.h` angelegt werden. Der Test-Vektor der vorherigen LZ4-Version ist nicht kompatibel mit der neuen; die Ausgabe-Länge hat sich um wenige Bytes (zugunsten des neuen LZ4) verändert. Das Problem konnte gelöst werden, indem ein neuer Test-Vektor für LZ4 (Listing 5.15) und LZ4HC sowie für den Dekompressor angelegt wird.
Es hatte sich jedoch gezeigt, dass der vorherige Text (zweimal der Satz „Join us now and share the software“) im Vergleich von LZ4 und LZ4HC keinen Unterschied gezeigt hat (selbe Ausgabe-

Länge), sodass ich mich entschieden habe, eine Zeile aus der LZ4-Readme zu nutzen. Die Ausgabe ist die Hexadezimal-Entsprechung des Binär-Outputs.

```
static struct comp_testvec lz4_comp_tv_template[] = {
  {
    .inlen  = 255,
    .outlen = 218,
    .input  = "LZ4 is lossless compression algorithm, providing"
      " compression speed at 400 MB/s per core, scalable "
      "with multi-cores CPU. It features an extremely fast "
      "decoder, with speed in multiple GB/s per core, "
      "typically reaching RAM speed limits on multi-core "
      "systems.",
    .output = "\xf9\x21\x4c\x5a\x34\x20\x69\x73\x20\x6c\x6f\x73\x73"
       "\x6c\x65\x73\x73\x20\x63\x6f\x6d\x70\x72\x65\x73\x73"
       "\x69\x6f\x6e\x20\x61\x6c\x67\x6f\x72\x69\x74\x68\x6d"
       ...,

  },
};
```

Listing 5.15: Der aktualisierte Test-Vektor für LZ4 in `Linux/crypto/testmgr.h` (gekürzt)

### 5.3.4 Performance-Regressions

Das LZ4-Modul ist nicht vollkommen neu. Wie bereits mehrfach erwähnt, existiert bereits seit Kernel 3.11 eine LZ4-Version im Linux-Kernel, die unter anderem von ZRAM genutzt wird. Im vorherigen Abschnitt habe ich erklärt, dass ZRAM und die crypto-API Fehler geworfen haben, da automatische Erwartungs-Tests fehlgeschlagen sind. Neben der offensichtlichen Erwartung der Linux-Community, dass neue oder aktualisierte Features überhaupt funktionieren, gibt es natürlich noch weitere Anforderungen. Im Fall von Kompressions-Algorithmen ist klar, dass eine aktualisierte Fassung vor allem hinsichtlich der Performance mindestens so gut abschneiden sollte, wie die vorhandene Lösung.

Ein großes Plus der Entwicklung von Open-Source-Software ist aus meiner Sicht, dass nicht ein Entwickler oder Entwickler einer einzelnen Firma an Software arbeiten, sondern verschiedene Programmierer mit den verschiedensten Hintergründen und Interessen. Unter Linux am Beispiel von LZ4 sind das zum Beispiel die Subsystem-Maintainer selbst, die große Erfahrung in der Kernel-Entwicklung haben, aber auch die Beteiligten von ZRAM, die Entwickler der crypto-API oder sonstige Nutzer von LZ4. E-Mails kamen beispielsweise von den Leuten der ARM-Architektur. Es werden also nicht nur Forderungen an das Produkt gestellt, sondern selber mit angepackt. Dadurch entwickelt sich eine **Schwarmintelligenz**.

Minchan Kim, mutmaßlich Entwickler von ZRAM, hatte beispielsweise in seiner E-Mail vom 9. Februar berichtet, dass er Performance-Tests durchgeführt hat und das neue LZ4 im Ver-

gleich zum alten wesentlich schlechter abschneidet. Seine Zahlen können Listing 5.16 entnommen werden.

```
                revert     lz4-update
    seq-write     1547        1339        86.55%
   rand-write    22775       19381        85.10%
    seq-read      7035        5589        79.45%
   rand-read     78556       68479        87.17%
 mixed-seq (R)    1305        1066        81.69%
 mixed-seq (W)    1205         984        81.66%
mixed-rand (R)   17421       14993        86.06%
mixed-rand (W)   17391       14968        86.07%
```

Listing 5.16: Benchmark-Ergebnisse des neuen LZ4 im Vergleich zum alten von Michan Kim

Für die Benchmarks hat er das Tool fio (flexible I/O-Tester) genutzt. „Fio simuliert ganz unterschiedliche Workloads und misst neben der Bandbreite auch die Anzahl der I/Os pro Sekunde, Latenzen sowie CPU-Auslastung“[6]. Seine Zahlen beziehen sich auf den zu dem Zeitpunkt aktuellen mmotm (-mm of the moment, ein Snapshot des -mm-Trees von Andrew Morton, den er regelmäßig aktualisiert). Dabei heißt *revert*, dass er die LZ4-Commits des aktualisierten Moduls rückgängig gemacht hat. Durch seine Konfiguration, die er zur Verfügung gestellt hat (siehe Anhang D.1) war es möglich, die Tests nachzuvollziehen.

Tatsächlich schnitt das neue LZ4 in meinem Test ebenfalls schlechter ab. Eine erste Idee hatte Eric Biggers in seiner E-Mail. Ihm fiel auf, dass einige Funktionen, die im Upstream-LZ4 als `FORCE_INLINE` definiert wurden, nicht mehr entsprechend markiert waren. Seinen Vorschlag, `FORCE_INLINE` in der (im Vergleich zum Upstream-LZ4 wesentlich kürzeren) Fassung `#define FORCE_INLINE __always_inline` neu zu definieren und die genannten Funktionen zu force-inlinen, hatte jedoch nicht den gewünschten Effekt.
Allerdings war die Richtung richtig: `FORCE_INLINE` ist für LZ4 ein besonders wichtiges Makro. Im Normalfall ist das Attribut `inline` wie in `inline int foo()` lediglich ein Vorschlag an den Compiler, den Aufruf der Funktion durch ihre Definition zu ersetzen. Vorschlag heißt, er kann das Attribut auch ignorieren. LZ4 nutzt eine `while(1)`-Schleife in der Kompression, die auf kurze Helfer-Funktionen (etwa das bereits bekannte `LZ4_read32`) zurückgreift. Es ist sinnvoll, diese Funktionen zu inlinene, um den Weg über den Call-Stack zu vermeiden: Für jeden Funktionsaufruf muss dieser auf den Stack gelegt werden, Parameter werden kopiert und gebunden, der Aufruf selbst findet statt und im Anschluss muss aufgeräumt werden. In einer solchen Schleife summieren sich diese Aufrufe. Die Idee von `FORCE_INLINE` ist es nun, den Kompiler zu *zwingen*, den jeweiligen Funktionsaufruf *immer* zu inlinen. Das bedeutet natürlich nicht, dass jede Funktion als `inline` zu definieren ist, im Gegenteil: Die Dokumentation über den Kernel-Coding-Style hat einen Absatz über die inline-Krankheit. Erwähnt wird dort vor allem, dass `inline` in der Konsequenz zu mehr Code führt (da ja die Funktionsaufrufe durch die jeweilige Definition ersetzt wird), was wiederrum zu Performance-Einbußen führen kann. Der Tenor ist also der, dass

[6] Steigerwald, *I/O-Benchmarks mit Fio*.

mit Möglichkeiten wie `inline` sorgsam umgegangen werden soll. `inline` sei allerdings Makros vorzuziehen.
***Makro*** ist hier das entscheidende Schlüsselwort: Das vorherige Kernel-LZ4 hat einige Funktionen als Makros in `lz4defs.h` definiert, die im Upstream-LZ4 normale Funktionen (ohne `inline`) sind. Das gilt im Wesentlichen für annähernd alle Funktionen, die der Kompressor und der Dekompressor gemeinsam nutzen: `LZ4_read*`, `LZ4_write*`, `LZ4_count`, `LZ4_NBCommonBytes` sowie `LZ4_copy8` (* ist dabei ein Platzhalter). Insgesamt sind das zehn Funktionen.
Interessant ist, dass es für die read*- und write*- Funktionen insgesamt drei Varianten gibt, die in gekürzter Fassung (am Beispiel von read32) in Listing 5.17 zu sehen sind.

```
1  /* 1 */
2  static U32 LZ4_read32(const void* memPtr) { return *(const U32*) memPtr;
       }
3
4  /* 2 */
5  typedef union { U16 u16; U32 u32; size_t uArch; } __attribute__((packed))
       unalign;
6
7  static U32 LZ4_read32(const void* ptr) { return ((const unalign*)ptr)->
       u32; }
8
9  /* 3 */
10 static U32 LZ4_read32(const void* memPtr)
11 {
12     U32 val; memcpy(&val, memPtr, sizeof(val)); return val;
13 }
```

Listing 5.17: Die verschiedenen Varianten von `LZ4_read32`

Benutzt wurde von mir zunächst Variante drei für gcc (Zeilen elf fortfolgend). Die Idee aller gezeigten Funktionen ist es, beispielsweise einen U32 (32 Bit `unsigned int`) aus einem Speicherbereich zu lesen oder darin zu schreiben, der nicht für U32 aligniert ist, sondern etwa für einen `unsigned char` (acht Bit). Diese und die anderen genannten Funktionen haben außerdem gemeinsam, dass sie sehr häufig aufgerufen werden. Eric Biggers hatte vorgeschlagen, die Funktionen für den Linux-Kernel umzuschreiben wie in Listing 5.18, indem die Kernel-Funktionen `read_unaligned` bzw. `write_unaligned` genutzt werden.

```
1  static FORCE_INLINE U32 LZ4_read32(const void *ptr)
2  {
3    return get_unaligned((const U32 *)ptr);
4  }
```

Listing 5.18: Reimplementation von `LZ4_read32` mit Kernel-Funktionen

Auch diese Funktionen sind ein erneutes Beispiel dafür, dass komplizierte und hochgradig portierbare Definitionen im Kernel stark zusammengefasst werden können, indem auf kernel-

eigene Funktionen zurückgegriffen wird. Außerdem wurde jeweils `FORCE_INLINE` ergänzt. Diese Änderungen haben die Performance tatsächlich merklich verbessert.
Ergänzt wurde dann noch `ccflags-y += -O3` in der Makefile, um das Optimierungs-Level des Compilers zu erhöhen. Der Kernel wird standardmäßig mit dem Level `-O2` kompiliert.

Mit allen genannten Änderungen hat sich dann gezeigt, dass die Regressionen nicht mehr beobachtet werden können. Tatsächlich ist das neue LZ4 meistens sogar schneller, als das alte LZ4.

## 5.4 Derzeitiger Stand

Vor kurzem[7] hat Linus Torvalds Linux 4.10 freigegeben. Gleichzeitig startete er das Merge-Window für Kernel 4.11, während dessen er Patches von den Maintainern in seinen Mainline-Tree integriert. Andrew Morton hat im Zuge dessen die LZ4-Änderungen als *[PATCH 114/124]* fortfolgend an Linus Torvalds gesendet[8]. In dem Mirror auf github unter `https://github.com/torvalds/linux/tree/master/lib/lz4` können die entsprechenden Commits bereits betrachtet werden. Alternativ genügt ein Aufruf von `git log --author=SSven Schmidt"` im Kernel-Repository.
Der nächste Schritt wird sein, dass zum Ende des Merge-Window ein erster Release Candidate veröffentlicht wird. Sofern keine gravierenden Probleme mehr auftreten, ist das Modul dann in Kernel 4.11 enthalten.

[7] Ende Februar 2017
[8] siehe `http://marc.info/?l=linux-mm-commits&m=148797728922812&w=2`

# 6 | Tests und Benchmarks

## 6.1 Testen des Kernel-LZ4

Um LZ4 im Kernel zu testen, sind zwei Grundpfeiler entscheidend: Umfangreiche **Build-Tests** und Tests im laufenden Betrieb (**Funktionstests**).

### 6.1.1 Build-Tests

Wir haben bereits im Abschnitt über die Konfiguration des Kernels gesehen, dass es viele verschiedene Konfigurationsmöglichkeiten zum Bauen des Kernels gibt. Allerdings umfassen meine Ausführungen nicht einmal annähernd die Menge aller Möglichkeiten. Im Linux-Kernel sind umfangreiche Build-Tests, das heißt wiederholte Aufrufe von `make`, daher besonders wichtig. LZ4 umfasst beispielsweise Zeilen wie in Listing 6.1. Der Precompiler wird unter einem 64-Bit-System die Zeilen im Else-Zweig entfernen, sodass der eigentliche Compiler nur diesen Pfad zu sehen bekommt; unter 32-Bit-Systemen verhält es sich umgekehrt. Das heißt, obwohl der Kernel unter einer bestimmten Konfiguration problemlos baut, bedeutet das nicht, dass das für alle Konfigurationen gilt.

```
1 #if LZ4_ARCH_64
2  ...
3 #else
4  ...
5 #endif
```

Listing 6.1: Architekturabhängige Pfade über #if oder #ifdef

Daraus folgt, der Entwickler sollte mit Blick auf seinen Code gezielt verschiedene Tests durchführen. Dabei kann der Entwickler sich die Arbeit erheblich vereinfachen: Werden beispielsweise keine Pakete erzeugt (`make deb-pkg ...`), sind Build-Tests weniger zeitaufwendig. Für umfangreiche Tests hat sich für mich der Befehl aus Listing 6.2 als Basis-Aufruf als wertvoll erwiesen.

```
# make -j8 2>&1 | tee log | egrep -w "lz4|sqashfs|pstore" -i --color
```

Listing 6.2: Häufig genutzter `make`-Aufruf für Build-Tests

Dabei wird die Ausgabe des Standard-Error-Streams (stderr) in den Standardoutput umgeleitet (2>&1), in eine Log-Datei (log) geschrieben, die bei Bedarf zum Beispiel über `grep` durchsucht werden kann, und die Ausgabe über egrep nach lz4, sqashfs und pstore gefiltert (`linux/fs/sqashfs` und `linux/fs/pstore` sind neben `linux/crypto` die weiteren LZ4-Nutzer). Einige wichtige Punkte beim Testen:

- Der `make`-Aufruf kann beispielsweise noch durch die Architektur ergänzt werden: `make ARCH=x86_64 ...` Die x86_64-Architektur definiert beispielsweise per Default die 64-Bit-Konfiguration.
- Es können Unterverzeichnisse eigenständig kompiliert werden: `make SUBDIRS=lib/lz4 ...` oder auch `make SUBDIRS=crypto ...`
- Es kann der Precompiler-Output betrachtet werden, das war beispielsweise interessant im Zuge der Performance-Regressionen: `make lib/lz4/lz4_compress.i` (die Endung `.i` steht für den Precompiler-Output)
- Testen verschiedener Konfigurationen: Maintainer wie beispielsweise Stephen Rothwell, der den next-Tree verwaltet, haben automatische Test-Werkzeuge mit Zufalls-Konfigurationen über `make randconfig`. Für LZ4 genügte allerdings die allmodconfig, zum Beispiel `make allmodconfig ARCH=x86_64 ...` Auch davon sollten mehrere Varianten getestet werden, beispielsweise ergänzend `make allmodconfig ARCH=arm64`
- Neben offensichtlichen Build-Fehlern, beispielsweise der Benutztung einer Funktion, die es gar nicht gibt (zum Beispiel weil sie über ein #if im Precompiler entfernt wurde), existieren noch generelle Fehler, die beim Programmieren auftreten können, zum Beispiel der Cast eines Pointers in einen Integer-Typ. Für diese Zwecke kann `make C=1` oder `make C=2` aufgerufen werden. Dadurch werden die Dateien durch Sparse getestet, einen Analyse-Werkzeug für Programmier-Fehler im Linux-Kernel[1]. Der Unterschied zwischen den Aufrufen besteht darin, dass `C=2` die Tests erzwingt.
- Ergänzend zu Spark gibt es noch weitere Werkzeuge, beispielsweise Smatch[2], das von einer Gruppe von Kernel-Entwicklern namens *kernel janitors* (Kernel Hausmeister) verwaltet wird. Tatsächlich bekam ich eine E-Mail mit einer Smatch-Ausgabe von den kernel janitors. Das Tool findet Fehler, die Sparse beispielsweise nicht findet. Ein solcher Test kann erfolgen durch `make CHECK="/opt/smatch/smatch -p=kernel"C=2`

Selbstverständlich ist es möglich und oft notwendig, die Befehle zu kombinieren. Beispiele:

- `make allmodconfig ARCH=x86_64 -j8`
- `make SUBDIRS=lib/lz4 -j8 CHECK="/opt/smatch/smatch -p=kernel"C=2 2>&1`

Jedoch sollte der Umfang dieser Tests niemanden abschrecken. Im Endeffekt hat jeder Linux-Nutzer ein Interesse daran, dass der Kernel so stabil, wie möglich funktioniert.

[1] Wikipedia, *Sparse — Wikipedia, The Free Encyclopedia*.
[2] siehe http://smatch.sourceforge.net/

### 6.1.2 Funktionstests

Neben den Build-Tests sind Tests im laufenden Betrieb notwendig. Für solche Funktionstests habe ich das Test-Modul für das LZ4-Modul entsprechend umgebaut. Der Code in Anhang C.2 ist bereits die geänderte Fassung. Zeilen vier bis acht nehmen eine Unterscheidung vor, welcher LZ4-Header genutzt werden soll: `../lz4.h` oder `linux/lz4.h`. Die Unterscheidung passiert über ein Makro `KERNELLZ4`, das über `make kernelLZ4` gesetzt wird. Mehr Unterschiede gibt es nicht, sodass ich an dieser Stelle auf die vorangegangenen Kapitel verweise, in denen bereits umfangreich beschrieben wurde, wie LZ4 getestet wurde. Vor allem sind das die Kapitel über notwendige Anpassungen für den Kernel und Funktionstests im LZ4-Modul.

## 6.2 Performance-Test mit ZRAM und fio

In Abschnitt 5.3.4 ging es darum, dass Minchan Kim Performance-Regressionen mit dem neuen LZ4 gefunden hat, die dann gemeinschaftlich behoben wurden. Im Zuge dessen stellte er seine Konfiguration für das Benchmarking-Tool *fio* zur Verfügung (Anhang D.1). Es liegt daher nahe, diese Konfiguration dankend anzunehmen und für eigene Tests zu verwenden.
fio selbst testet I/O-Operationen auf Geräte. Ein solches Gerät kann beispielsweise mit ZRAM erstellt werden. ZRAM hatte ich in dieser Arbeit bereits mehrfach erwähnt, denn es gehört zu den Nutzern von `crypto/lz4`. ZRAM bietet eine Möglichkeit, Daten aus dem Arbeitsspeicher in einem eigenen Gerät auszulagern, ähnlich wie es vom Swap-Speicher bekannt ist, der auf der Festplatte erstellt wird und daher ziemlich langsam ist. ZRAM erstellt ein komprimiertes Blockgerät direkt im Arbeitsspeicher[3]. Für die Kompression der Daten steht beispielsweise LZ4 zur Verfügung.

Zum Testen mit ZRAM und fio wird zunächst fio installiert, dazu genügt ein simples `sudo apt-get update && sudo apt-get install -y fio`. Die folgenden Schritte zeigt Listing 6.3.

```
# modprobe -a zram lz4
# echo lz4 > /sys/block/zram0/comp_algorithm
# echo 2G > /sys/block/zram0/disksize
# fio fio-job-desc > fio-log
```

Listing 6.3: Benchmarking mit fio und ZRAM

`modprobe` ist bereits bekannt. Mit dem Switch -a können mehrere Module spezifiziert werden. Die beiden echo-Aufrufe setzen Konfigurations-Parameter für das ZRAM-Device unter `/dev/zram0`: Als Algorithmus soll LZ4 verwendet werden und das Gerät erhält eine Größe von 2 Gigabyte. Der letzte Schritt ist der Aufruf von fio mit einer Datei, die auszuführende Jobs definiert (nämlich die von Minchan Kim). Den Inhalt von `fio-log` zeigt Anhang D.2.

[3] ubuntuusers, *zRam*.

# 7 | Fazit

Das Projekt hat viele Bereiche tangiert: Die Entwicklung für den Linux-Kernel, die Unterschiede zwischen User-Space- und Kernel-Software, den Prozess, Änderungen in den Linux-Kernel einzubringen und vor allem auch die Interna von Lustre. Zu jedem dieser Themen kann problemlos viel und umfangreich geforscht und gearbeitet werden. Und zu jedem dieser Themen habe ich viel lernen können.

Ziel dieses Projekts war es, ein aktuelles LZ4 in den Linux-Kernel und in Lustre zu integrieren. Die Integration in Lustre ist aus meiner Sicht gelungen.
Im Fall von LZ4 im Kernel werde ich noch einige Zeit beschäftigt sein, da der Prozess noch nicht abgeschlossen ist. Insgesamt muss man hier festhalten, dass Patchserien, die Neuerungen in den Kernel einbringen, schwieriger zu integrieren sind, als beispielsweise Bugfixes. Mit dieser Tatsache im Hinterkopf finde ich, dass die aufgetretenen Probleme gut gelöst werden konnten und ich freue mich, dass die Patchserie sich auf dem Weg in Kernel 4.11 befindet.

Mit diesem Bericht wollte ich vor allem auf die Arbeit mit dem Kernel genauer eingehen, um einen Einblick zu geben, wie Beiträge zum Linux-Kernel funktionieren und wie diese eigene Welt in ihrem Inneren funktioniert.

Zusammenfassend kann ich sagen, dass ich im Rahmen des Projekts und des zugehörigen Seminars viele wertvolle Einblicke bekommen und umfangreiches Wissen erwerben konnte.

# Literaturverzeichnis


Colett, Yann. *LZ4 github repository.* URL: https://github.com/lz4/lz4.

community, The kernel development. *Linux kernel coding style.* URL: https://01.org/linuxgraphics/gfx-docs/drm/process/coding-style.html (abgerufen am 10. 03. 2017).

Cyan. *Sampling, or a faster LZ4.* 7. Apr. 2015. URL: http://fastcompression.blogspot.de/2015/04/sampling-or-faster-lz4.html (abgerufen am 04. 03. 2017).

Kroah-Hartman, Greg. *3.10 Linux Kernel Development Rate.* Juli 2013. URL: http://kroah.com/log/blog/2013/07/01/3-dot-10-kernel-development-rate/ (abgerufen am 17. 02. 2017).

— *Linux kernel in a nutshell.* O'Reilly, 2007.

linuxtv. *Development: Linux Kernel patch submittal checklist.* URL: https://linuxtv.org/wiki/index.php/Development:_Linux_Kernel_patch_submittal_checklist (abgerufen am 04. 03. 2017).

Project, The Linux Documentation. *The Linux Kernel Module Programming Guide.* URL: http://www.tldp.org/LDP/lkmpg/2.6/html/x279.html (abgerufen am 18. 02. 2017).

Schmidt, Sven. *Speicherverwaltung im Kernel.* URL: https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2016_2017/ep-1617-schmidt-speicherverwaltung-im-kernel-praesentation.pdf.

Steigerwald, Martin. *I/O-Benchmarks mit Fio.* Mai 2011. URL: http://www.admin-magazin.de/Das-Heft/2011/05/I-O-Benchmarks-mit-Fio (abgerufen am 24. 02. 2017).

ubuntuusers. *zRam.* URL: https://wiki.ubuntuusers.de/zRam/ (abgerufen am 03. 03. 2017).

Wikipedia. *Category:Compression file systems — Wikipedia, The Free Encyclopedia.* 2017. URL: https://en.wikipedia.org/w/index.php?title=Category:Compression_file_systems&oldid=138962201 (abgerufen am 07. 03. 2017).

— *Loadable kernel module — Wikipedia, The Free Encyclopedia.* 2017. URL: https://en.wikipedia.org/w/index.php?title=Loadable_kernel_module&oldid=761957511 (abgerufen am 18. 02. 2017).

— *Lustre (file system) — Wikipedia, The Free Encyclopedia.* 2017. URL: https://en.wikipedia.org/w/index.php?title=Lustre_(file_system)&oldid=758996125 (abgerufen am 10. 01. 2017).

— *Mm tree — Wikipedia, The Free Encyclopedia.* 2016. URL: https://en.wikipedia.org/w/index.php?title=Mm_tree&oldid=731464717 (abgerufen am 24. 02. 2017).

— *Sparse — Wikipedia, The Free Encyclopedia.* 2016. URL: https://en.wikipedia.org/w/index.php?title=Sparse&oldid=733818935 (abgerufen am 04. 03. 2017).

Xuetao, Guan. *Linux Kernel Development: Getting started*. Mai 2012. URL: http://slideplayer.com/slide/6202889/ (abgerufen am 24. 02. 2017).

# Abbildungsverzeichnis

# Listing-Verzeichnis

# Anhang

# A | Benutzte Konventionen

**Bold**

Wird benutzt, um wichtige **Schlüsselbegriffe** zu kennzeichnen.

*Italic*

Wird benutzt, um Begriffe zu kennzeichnen, die vorher noch nicht verwendet wurden und die im jeweiligen Kontext definiert werden oder, um Begriffe hervorzuheben, denen im jeweiligen Kontext besondere Aufmerksamkeit zu widmen ist.

***Italic bold***

Wird benutzt, um Begriffe zu kennzeichnen, die vorher noch nicht verwendet wurden und die besonders wichtige Schlüsselbegriffe darstellen.

`Constant Width`

Wird für Funktionsnamen benutzt (`malloc`), aber auch für Konstanten (`PAGE_SIZE`), Shell-Befehle (`getconf`) oder Pfade (`/usr/bin/nano`).

$, #

Werden in den Beispiel-Eingaben in die Shell benutzt, um Eingaben als Root (#) bzw. User ($) zu verdeutlichen:

```
$ sudo su
# ...
```

Verzeichnisnamen

Eine Angabe eines Verzeichnisses beispielsweise als `linux/lib/lz4` impliziert, dass *linux* eine über `git clone` bezogene Kopie des Kernels. In Terminal-Eingaben überspringe ich ein eventuell notwendiges `cd linux`. Für Lustre gilt entsprechendes.

# B | Wichtige Referenzen

Einige wichtige Referenz-Werke für die Einsendung von Patches in den Linux-Kernel oder Linux-Kernel-Entwicklung an sich, die ich interessant, lesenswert oder nützlich finde:

- Kroah-Hartman, Greg (2007). Linux kernel in a nutshell. Farnham: O'Reilly, kostenlos verfügbar unter `http://www.kroah.com/lkn/`
- *Working with the kernel development community*, in der offiziellen Dokumentation des Kernels, ein sehr umfangreiches Werk über den kompletten Submitting-Prozess, das den Code-Style, die Arbeit mit den Maintainern und Reviewern, das Senden von E-Mails, die Bedeutung von Sign-Off und vieles mehr umfasst sowie eine Checkliste für das Testen von Patches bereithält
- Der Artikel *How to send patches with git-send-email*
- *Writing a Linux Kernel Module*, ein sehr gutes Schritt-für-Schritt-Werk zu diesem Thema
- Dasselbe Thema wie der vorherige Punkt, ebenfalls sehr verständlich und umfangreich: *Be a kernel hacker*
- Andrew Morton, The perfect patch
  Der Name ist selbsterklärend: Andrew Morton gibt eine Übersicht zu Punkten, die aus seiner Sicht wichtig für Patches sind
- Greg Kroah-Hartman: *How to piss off a Linux Kernel subsystem maintainer*, Part 1, 2, 3, 4, 5, 6
  In dieser Serie dreht Kroah-Hartman die Logik des vorherigen Punktes um: Er schreibt nicht, wie ein perfekter Patch aussehen sollte, sondern geht auf einige schlechte Patches ein, die ihm zugeschickt wurden.
- Das Smatch-Tool zum Testen von Kernel-Code
- Von Patchwork können Entwickler Patches beziehen, wenn sie nicht selbst Empfänger einer E-Mail sind.

# C | Sourcecodes

## C.1 „Hallo Welt“-Kernel-Modul

```
1  #include <linux/init.h>              // macros used to mark functions as
       __init or __exit
2  #include <linux/module.h>            // basic header for kernel modules
3  #include <linux/kernel.h>            // contains basic types, macros,
       functions for the kernel
4
5  static int __init hello_init(void)
6  {
7    printk(KERN_INFO "[hello-mod] Hello world from the kernel!");
8
9    return 0;
10 }
11
12 static void __exit hello_exit(void)
13 {
14   printk(KERN_INFO "[hello-mod] Goodbye from the kernel!");
15 }
16
17 int hello_exported(void)
18 {
19   // do meaningful stuff
20
21   return 1;
22 }
23 EXPORT_SYMBOL(hello_exported);
24
25
26 MODULE_LICENSE("GPL");
27 MODULE_AUTHOR("Sven Schmidt");
28 MODULE_DESCRIPTION("A simple hello world example module");
29 MODULE_VERSION("1.0.0");
30
31 module_init(hello_init);
32 module_exit(hello_exit);
```

Listing C.1: Ein simples „Hallo Welt“-Kernelmodul

## C.2 Testmodul

```
1  /*-************************************
2  * Includes
3  **************************************/
4  #ifdef KERNELLZ4
5  #include <linux/lz4.h>
6  #else
7  #include "../lz4.h"
8  #endif
9  #include <linux/init.h>
10 #include <linux/module.h>
11 #include <linux/moduleparam.h>
12 #include <linux/kernel.h>
13 #include <linux/slab.h>
14 #include <linux/fs.h>
15 #include <asm/uaccess.h>
16
17 /*-************************************
18 * Module info
19 **************************************/
20 MODULE_LICENSE("Dual BSD/GPL");
21 MODULE_AUTHOR("Sven Schmidt");
22 MODULE_DESCRIPTION("A simple test module for lz4_compress and
       lz4_decompress kernel modules.");
23 MODULE_VERSION("1.0.0");
24
25 /*-************************************
26 * Parameters
27 **************************************/
28
29 /*
30  * Get acceleration factor as a module param (LZ4fast only) or fallback
31  * to LZ4_ACCELERATION_DEFAULT (1) if not set
32 */
33 static int acceleration = LZ4_ACCELERATION_DEFAULT;
34 module_param(acceleration, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
35
36 /*
37  * Get compression ratio as as a module param (LZ4HC only) or fallback
38  * to LZ4HC_DEFAULT_CLEVEL (9) if not set
39  */
40 static int cLevel = LZ4HC_DEFAULT_CLEVEL;
41 module_param(cLevel, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
42
43 /*
44  * Should we test HC compression (1) or LZ4fast (0)?
45  */
46 static int useHC = 0;
```

```
47 module_param(useHC, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
48
49 /*-*********************************
50 * Test data
51 ***********************************/
52 struct comp_testvec {
53   int inlen;
54   char *input;
55   void *wrkmem;
56   void *dest;
57 };
58
59 static struct comp_testvec lz4_test_data = {
60   .inlen = 255,
61   .input = "LZ4 is lossless compression algorithm, providing compression
       "
62     "speed at 400 MB/s per core, scalable with multi-cores CPU. It"
63     " features an extremely fast decoder, with speed in multiple "
64     "GB/s per core, typically reaching RAM speed limits on "
65     "multi-core systems."
66 };
67
68 /* We will write the output to files in /tmp for debugging purposes */
69 #define lz4test_outFileName "/tmp/lz4_compressed"
70 #define lz4test_decompressFileName "/tmp/lz4_decompressed"
71
72 /*-*********************************
73 * Methods
74 ***********************************/
75
76 /* Prototype for later definition */
77 static void lz4_compress_test_clean(void);
78
79 /**
80  * write() - Writes data of size size into file path
81  * @path: File to write buffer into
82  * @data: Buffer containing the data to write
83  * @size: Size of data
84  */
85 static void write(char *path, char *data, size_t size)
86 {
87   struct file *fd;
88   mm_segment_t oldFS;
89
90   /*
91    * Open file write only (O_WRONLY), create if not existing (O_CREAT)
92    * truncate file contents before opening (O_TRUNC)
93    */
94   fd = filp_open(path, O_CREAT | O_TRUNC | O_WRONLY, S_IRWXU);
95
96   if (!IS_ERR(fd)) {
97     /*
```

```
98          * Backup current file system and set to KERNEL_DS
99          * ("Kernel data segment")
100         */
101        oldFS = get_fs();
102        set_fs(KERNEL_DS);
103
104        /* Write output to file */
105        vfs_write(fd, data, size, &fd->f_pos);
106
107        /* Restore old file system and close file */
108        set_fs(oldFS);
109        filp_close(fd, NULL);
110
111        printk(KERN_INFO "[LZ4Test]: Wrote to %s\n", path);
112    } else {
113        printk(KERN_ERR "[LZ4Test]: Error opening file %s", path);
114    }
115 }
116
117 static size_t testLZ4Compress(void)
118 {
119    size_t outSize;
120    const size_t maxOutSize = LZ4_compressBound(lz4_test_data.inlen);
121
122    if (useHC) {
123        printk(KERN_INFO "[LZ4Test]: Start compressing using LZ4HC with
          cLevel of %d, input is\n%s\n",
124            cLevel, lz4_test_data.input);
125
126        outSize = LZ4_compress_HC(lz4_test_data.input,
127            lz4_test_data.dest, lz4_test_data.inlen, maxOutSize,
128            cLevel, lz4_test_data.wrkmem);
129    } else {
130        printk(KERN_INFO "[LZ4Test]: Start compressing using LZ4Fast with
          acceleration of %d, input is\n%s\n",
131            acceleration, lz4_test_data.input);
132
133        outSize = LZ4_compress_fast(lz4_test_data.input,
134            lz4_test_data.dest, lz4_test_data.inlen, maxOutSize,
135            acceleration, lz4_test_data.wrkmem);
136    }
137
138    printk(KERN_INFO "[LZ4Test]: Compression %s result: %d bytes -> %d
          bytes\n",
139        (useHC ? "using LZ4HC" : "using LZ4fast"),
140        lz4_test_data.inlen, (unsigned int)outSize);
141
142    /* Save the result to a temp file for debugging purposes */
143    write(lz4test_outFileName, lz4_test_data.dest, outSize);
144
145    return outSize;
146 }
```

```
147
148 static size_t testLZ4Decompress(size_t compressedSize)
149 {
150   size_t outSize;
151
152   outSize = LZ4_decompress_safe(lz4_test_data.dest, lz4_test_data.input,
153     compressedSize, lz4_test_data.inlen);
154   printk(KERN_INFO "[LZ4Test]: Decompression result: %d bytes -> %d bytes
      \n",
155     (unsigned int)compressedSize, (unsigned int)outSize);
156
157   write(lz4test_decompressFileName, lz4_test_data.input, outSize);
158
159   return outSize;
160 }
161
162 static int __init lz4_compress_test_init(void)
163 {
164   size_t stateSize, compressedSize, decompressedSize;
165
166   printk(KERN_INFO "[LZ4Test]: Module loaded, starting tests\n");
167
168   /* Check how large the working memory must be */
169   if (useHC) {
170     stateSize = LZ4HC_MEM_COMPRESS;
171   } else {
172     stateSize = LZ4_MEM_COMPRESS;
173   }
174
175   /* Allocate working memory */
176   lz4_test_data.wrkmem = kmalloc(stateSize, GFP_KERNEL);
177
178   if (lz4_test_data.wrkmem == NULL) {
179     printk(KERN_ERR "[LZ4Test]: Could not allocate working memory!");
180
181     goto _no_mem;
182   }
183
184   /* Allocate destination buffer and fill with zeros (__GFP_ZERO) */
185   lz4_test_data.dest = kmalloc(LZ4_compressBound(lz4_test_data.inlen),
186     GFP_KERNEL | __GFP_ZERO);
187
188   if (lz4_test_data.dest == NULL) {
189     printk(KERN_ERR "[LZ4Test]: Could not allocate destination buffer!");
190
191     goto _no_mem;
192   }
193
194   /* Do tests */
195   compressedSize = testLZ4Compress();
196
197   if (compressedSize == 0) {
```

```
198     goto _err;
199   }
200
201   decompressedSize = testLZ4Decompress(compressedSize);
202
203   if (decompressedSize < 0) {
204     goto _err;
205   }
206
207   return 0;
208
209 _no_mem:
210   lz4_compress_test_clean();
211   return -ENOMEM;
212
213 _err:
214   lz4_compress_test_clean();
215   return -ENOEXEC;
216 }
217
218 static void lz4_compress_test_clean(void)
219 {
220   printk(KERN_INFO "[LZ4Test]: Cleaning up...");
221
222   /* Free allocated memory */
223   kfree(lz4_test_data.wrkmem);
224   kfree(lz4_test_data.dest);
225 }
226
227 static void __exit lz4_compress_test_exit(void)
228 {
229   lz4_compress_test_clean();
230   printk(KERN_INFO "[LZ4Test]: Module removed\n");
231 }
232
233 module_init(lz4_compress_test_init);
234 module_exit(lz4_compress_test_exit);
```

Listing C.2: Das Testmodul zum Testen des LZ4-Moduls und des Kernel-LZ4

## Makefile

```
1 obj-m += lz4_compress_test.o
2
3 all:
4     cp ../Module.symvers ./Module.symvers
5     make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
6
7 kernelLZ4:
```

```
8       make CPPFLAGS="-DKERNELLZ4" -C /lib/modules/$(shell uname -r)/build/
        M=$(PWD) modules

10  clean:
11      make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean

13  testhc:
14      insmod lz4_compress_test.ko useHC=1 cLevel=9
15      rmmod lz4_compress_test

17  testfast:
18      insmod lz4_compress_test.ko useHC=0 acceleration=1
19      rmmod lz4_compress_test
```

Listing C.3: Makefile zum Bauen des Testmoduls

# D | Benchmarks

## D.1 fio-Config von Minchan Kim

```
[global]
bs=4k
ioengine=sync
size=100m
numjobs=1
group_reporting
buffer_compress_percentage=30
scramble_buffers=0
filename=/dev/zram0
loops=10
fsync_on_close=1

[seq-write]
bs=64k
rw=write
stonewall

[rand-write]
rw=randwrite
stonewall

[seq-read]
bs=64k
rw=read
stonewall

[rand-read]
rw=randread
stonewall

[mixed-seq]
bs=64k
rw=rw
stonewall

[mixed-rand]
```

```
rw=randrw
stonewall
```

Listing D.1: fio-Konfiguration von Minchan Kim, die er für seine Tests genutzt hat

## D.2 Ausgabe von fio im Benchmark-Test mit ZRAM

```
seq-write: (g=0): rw=write, bs=64K-64K/64K-64K/64K-64K, ioengine=sync, iodepth=1
rand-write: (g=1): rw=randwrite, bs=4K-4K/4K-4K/4K-4K, ioengine=sync, iodepth=1
seq-read: (g=2): rw=read, bs=64K-64K/64K-64K/64K-64K, ioengine=sync, iodepth=1
rand-read: (g=3): rw=randread, bs=4K-4K/4K-4K/4K-4K, ioengine=sync, iodepth=1
mixed-seq: (g=4): rw=rw, bs=64K-64K/64K-64K/64K-64K, ioengine=sync, iodepth=1
mixed-rand: (g=5): rw=randrw, bs=4K-4K/4K-4K/4K-4K, ioengine=sync, iodepth=1
fio-2.1.11
Starting 6 processes

seq-write: (groupid=0, jobs=1): err= 0: pid=2436: Fri Mar  3 11:20:56 2017
  write: io=1000.0MB, bw=904594KB/s, iops=14134, runt=  1132msec
    clat (usec): min=13, max=649, avg=18.39, stdev=25.01
     lat (usec): min=13, max=649, avg=18.44, stdev=25.02
    clat percentiles (usec):
     |  1.00th=[   14],  5.00th=[   15], 10.00th=[   15], 20.00th=[   16],
     | 30.00th=[   16], 40.00th=[   17], 50.00th=[   17], 60.00th=[   17],
     | 70.00th=[   17], 80.00th=[   18], 90.00th=[   18], 95.00th=[   19],
     | 99.00th=[   28], 99.50th=[   41], 99.90th=[  516], 99.95th=[  540],
     | 99.99th=[  588]
    lat (usec) : 20=96.71%, 50=2.83%, 100=0.12%, 250=0.09%, 500=0.14%
    lat (usec) : 750=0.11%
  cpu          : usr=1.06%, sys=98.32%, ctx=46, majf=0, minf=7
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=0/w=16000/d=0, short=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=1
```

```
rand-write: (groupid=1, jobs=1): err= 0: pid=2437: Fri Mar  3 11:20:56 2017
  write: io=1000.0MB, bw=674572KB/s, iops=168642, runt=  1518msec
    clat (usec): min=1, max=655, avg= 1.42, stdev= 2.05
     lat (usec): min=1, max=655, avg= 1.45, stdev= 2.06
    clat percentiles (usec):
     |  1.00th=[    1],  5.00th=[    1], 10.00th=[    1], 20.00th=[    1],
     | 30.00th=[    1], 40.00th=[    1], 50.00th=[    1], 60.00th=[    1],
     | 70.00th=[    2], 80.00th=[    2], 90.00th=[    2], 95.00th=[    2],
     | 99.00th=[    3], 99.50th=[    3], 99.90th=[    5], 99.95th=[   12],
     | 99.99th=[   62]
    lat (usec) : 2=62.92%, 4=36.83%, 10=0.20%, 20=0.03%, 50=0.01%
    lat (usec) : 100=0.01%, 250=0.01%, 500=0.01%, 750=0.01%
  cpu          : usr=8.17%, sys=90.97%, ctx=60, majf=0, minf=7
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=0/w=256000/d=0, short=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=1
seq-read: (groupid=2, jobs=1): err= 0: pid=2440: Fri Mar  3 11:20:56 2017
  read : io=1000.0MB, bw=1388.1MB/s, iops=22222, runt=   720msec
    clat (usec): min=5, max=875, avg=40.91, stdev=60.32
     lat (usec): min=5, max=875, avg=40.97, stdev=60.32
    clat percentiles (usec):
     |  1.00th=[    6],  5.00th=[    6], 10.00th=[    6], 20.00th=[    6],
     | 30.00th=[    6], 40.00th=[    7], 50.00th=[    7], 60.00th=[    7],
     | 70.00th=[    8], 80.00th=[  137], 90.00th=[  141], 95.00th=[  143],
     | 99.00th=[  169], 99.50th=[  191], 99.90th=[  274], 99.95th=[  310],
     | 99.99th=[  756]
    lat (usec) : 10=73.53%, 20=1.07%, 50=0.28%, 100=0.17%, 250=24.82%
    lat (usec) : 500=0.11%, 750=0.01%, 1000=0.01%
  cpu          : usr=2.78%, sys=96.24%, ctx=45, majf=0, minf=20
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=16000/w=0/d=0, short=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=1
```

```
rand-read: (groupid=3, jobs=1): err= 0: pid=2441: Fri Mar  3 11:20:56 2017
  read : io=1000.0MB, bw=988417KB/s, iops=247104, runt=  1036msec
    clat (usec): min=2, max=705, avg= 3.30, stdev= 2.40
     lat (usec): min=2, max=705, avg= 3.33, stdev= 2.40
    clat percentiles (usec):
     |  1.00th=[    2],  5.00th=[    3], 10.00th=[    3], 20.00th=[    3],
     | 30.00th=[    3], 40.00th=[    3], 50.00th=[    3], 60.00th=[    3],
     | 70.00th=[    3], 80.00th=[    4], 90.00th=[    4], 95.00th=[    4],
     | 99.00th=[    5], 99.50th=[    6], 99.90th=[   15], 99.95th=[   21],
     | 99.99th=[   76]
    lat (usec) : 4=73.27%, 10=26.59%, 20=0.08%, 50=0.03%, 100=0.03%
    lat (usec) : 250=0.01%, 500=0.01%, 750=0.01%
  cpu          : usr=19.32%, sys=79.61%, ctx=49, majf=0, minf=5
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=256000/w=0/d=0, short=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=1
mixed-seq: (groupid=4, jobs=1): err= 0: pid=2442: Fri Mar  3 11:20:56 2017
  read : io=531840KB, bw=474434KB/s, iops=7413, runt=  1121msec
    clat (usec): min=5, max=452, avg=66.53, stdev=87.62
     lat (usec): min=5, max=452, avg=66.57, stdev=87.63
    clat percentiles (usec):
     |  1.00th=[    5],  5.00th=[    5], 10.00th=[    5], 20.00th=[    6],
     | 30.00th=[    6], 40.00th=[    6], 50.00th=[    7], 60.00th=[    7],
     | 70.00th=[  137], 80.00th=[  139], 90.00th=[  157], 95.00th=[  270],
     | 99.00th=[  282], 99.50th=[  290], 99.90th=[  350], 99.95th=[  398],
     | 99.99th=[  454]
  write: io=492160KB, bw=439037KB/s, iops=6859, runt=  1121msec
    clat (usec): min=9, max=648, avg=11.85, stdev= 8.12
     lat (usec): min=9, max=648, avg=11.90, stdev= 8.12
    clat percentiles (usec):
     |  1.00th=[    9],  5.00th=[    9], 10.00th=[   10], 20.00th=[   10],
     | 30.00th=[   10], 40.00th=[   10], 50.00th=[   11], 60.00th=[   11],
     | 70.00th=[   12], 80.00th=[   12], 90.00th=[   17], 95.00th=[   17],
```

```
99      | 99.00th=[   19], 99.50th=[   23], 99.90th=[   32], 99.95th=[
    99],
100     | 99.99th=[  652]
101     lat (usec) : 10=37.12%, 20=43.64%, 50=0.41%, 100=0.11%, 250=14.09%
102     lat (usec) : 500=4.61%, 750=0.01%
103   cpu          : usr=0.00%, sys=99.29%, ctx=53, majf=0, minf=7
104   IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%,
      >=64=0.0%
105     submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
      >=64=0.0%
106     complete   : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
      >=64=0.0%
107     issued     : total=r=8310/w=7690/d=0, short=r=0/w=0/d=0
108     latency    : target=0, window=0, percentile=100.00%, depth=1
109 mixed-rand: (groupid=5, jobs=1): err= 0: pid=2443: Fri Mar  3 11:20:56
    2017
110   read : io=512440KB, bw=393277KB/s, iops=98319, runt=  1303msec
111     clat (usec): min=2, max=636, avg= 3.37, stdev= 2.62
112     lat (usec): min=2, max=636, avg= 3.40, stdev= 2.64
113     clat percentiles (usec):
114     |  1.00th=[    2],  5.00th=[    3], 10.00th=[    3], 20.00th=[
    3],
115     | 30.00th=[    3], 40.00th=[    3], 50.00th=[    3], 60.00th=[
    3],
116     | 70.00th=[    4], 80.00th=[    4], 90.00th=[    4], 95.00th=[
    4],
117     | 99.00th=[    5], 99.50th=[    6], 99.90th=[   16], 99.95th=[
    20],
118     | 99.99th=[   95]
119   write: io=511560KB, bw=392602KB/s, iops=98150, runt=  1303msec
120     clat (usec): min=1, max=129, avg= 1.49, stdev= 1.15
121     lat (usec): min=1, max=129, avg= 1.53, stdev= 1.17
122     clat percentiles (usec):
123     |  1.00th=[    1],  5.00th=[    1], 10.00th=[    1], 20.00th=[
    1],
124     | 30.00th=[    1], 40.00th=[    1], 50.00th=[    1], 60.00th=[
    2],
125     | 70.00th=[    2], 80.00th=[    2], 90.00th=[    2], 95.00th=[
    2],
126     | 99.00th=[    3], 99.50th=[    3], 99.90th=[    5], 99.95th=[
    13],
127     | 99.99th=[   70]
128     lat (usec) : 2=28.12%, 4=56.08%, 10=15.71%, 20=0.06%, 50=0.01%
129     lat (usec) : 100=0.02%, 250=0.01%, 750=0.01%
130   cpu          : usr=9.22%, sys=90.02%, ctx=88, majf=0, minf=7
131   IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%,
      >=64=0.0%
132     submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
      >=64=0.0%
133     complete   : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
      >=64=0.0%
134     issued     : total=r=128110/w=127890/d=0, short=r=0/w=0/d=0
```

```
135       latency   : target=0, window=0, percentile=100.00%, depth=1
136
137 Run status group 0 (all jobs):
138   WRITE: io=1000.0MB, aggrb=904593KB/s, minb=904593KB/s, maxb=904593KB/s,
        mint=1132msec, maxt=1132msec
139
140 Run status group 1 (all jobs):
141   WRITE: io=1000.0MB, aggrb=674571KB/s, minb=674571KB/s, maxb=674571KB/s,
        mint=1518msec, maxt=1518msec
142
143 Run status group 2 (all jobs):
144    READ: io=1000.0MB, aggrb=1388.1MB/s, minb=1388.1MB/s, maxb=1388.1MB/s,
        mint=720msec, maxt=720msec
145
146 Run status group 3 (all jobs):
147    READ: io=1000.0MB, aggrb=988416KB/s, minb=988416KB/s, maxb=988416KB/s,
        mint=1036msec, maxt=1036msec
148
149 Run status group 4 (all jobs):
150    READ: io=531840KB, aggrb=474433KB/s, minb=474433KB/s, maxb=474433KB/s,
        mint=1121msec, maxt=1121msec
151   WRITE: io=492160KB, aggrb=439036KB/s, minb=439036KB/s, maxb=439036KB/s,
        mint=1121msec, maxt=1121msec
152
153 Run status group 5 (all jobs):
154    READ: io=512440KB, aggrb=393277KB/s, minb=393277KB/s, maxb=393277KB/s,
        mint=1303msec, maxt=1303msec
155   WRITE: io=511560KB, aggrb=392601KB/s, minb=392601KB/s, maxb=392601KB/s,
        mint=1303msec, maxt=1303msec
156
157 Disk stats (read/write):
158   zram0: ios=765233/647829, merge=0/0, ticks=1444/1712, in_queue=3156,
      util=43.88%
```

Listing D.2: Ausgabe von fio im Test mit ZRAM