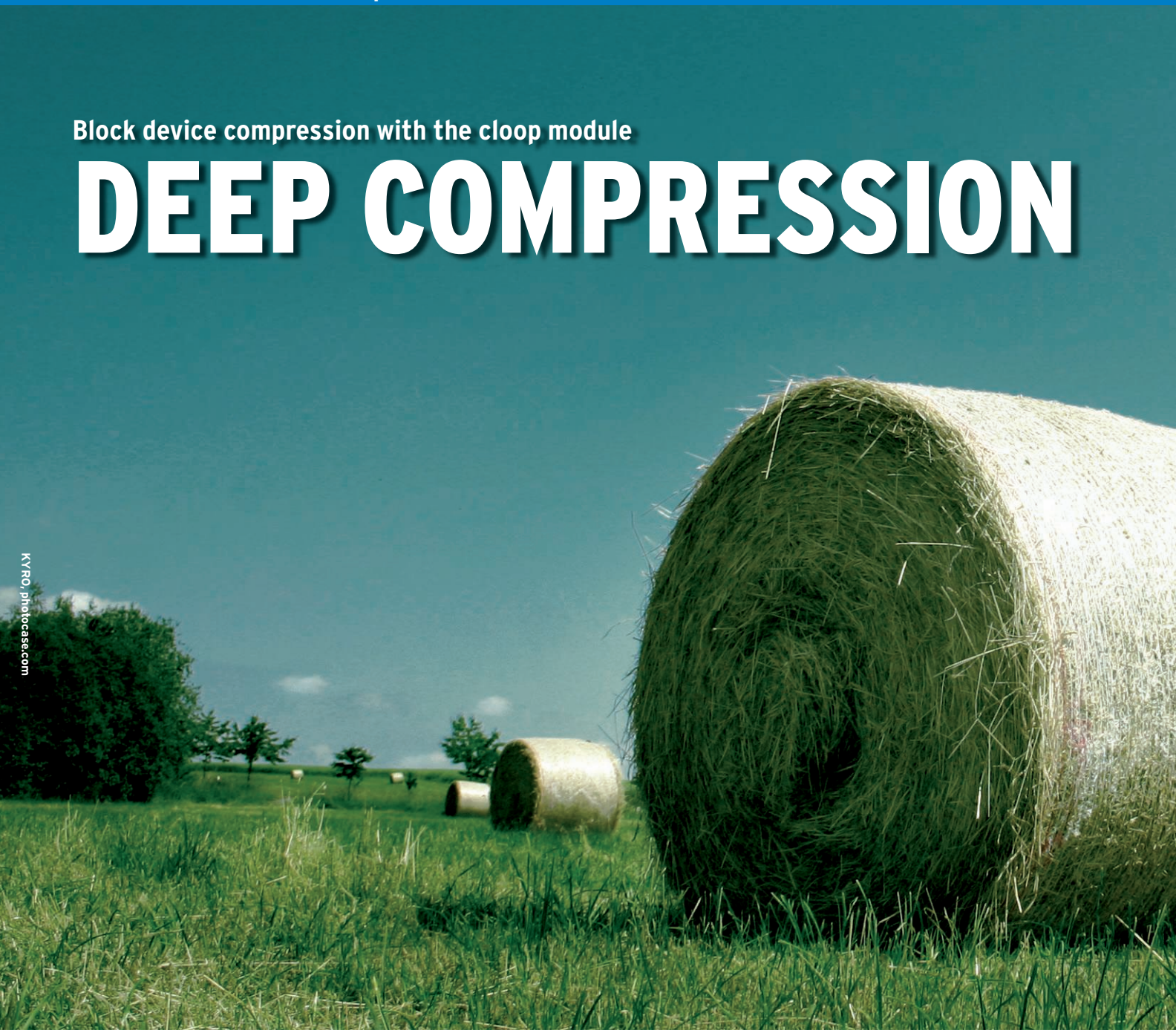
Block device compression with the cloop module

DEEP COMPRESSION



KYRO, photocase.com

The cloop module lets you manage compression at the block device level. Read on to learn how Knoppix and other Live CDs fit all that software on a single disc. **BY KLAUS KNOPPER**

Cloop is a kernel block device module used in Live CDs such as Knoppix. The cloop module allows the system to read compressed data, usually from a file, thus creating compressed virtual disks. Using cloop, a Linux installation of about 2GB fits on a single 700MB CD-R disc. In this article, I look at how cloop works and provide some insight into general kernel structures of a block device.

A Unix system traditionally distinguishes between *character-based* and *block-based* devices. If you look into the output of *ls -l /dev*, you will easily recognize these devices by the prefix – c for character-based and *b* for block-based devices – at the beginning of the output line (see Listing 1).

Character-based devices, such as tape drives, mice, and gamepads, provide sequential, character-by-character access to data.

Block devices instead allow direct access to arbitrary blocks of data, indexed by block number or sector (segments of 512 bytes), and they are usually used for random access storage like ramdisks, CD-ROMs, floppy disks, hard disks, and hard disk partitions.

Filesystems are a logical representation of ordered data that is often present on a block device. A filesystem turns raw data into the familiar directory/file view. The *mount* command is the bridge between a block device partition and its projection into a mount point directory.

Cloop: A Compressed Loopback Block Device

One block device included in any Linux kernel is loop, which maps a plain file to a block device node in */dev/loop [number]*. The loop-attached file can then be mounted like an ordinary hard

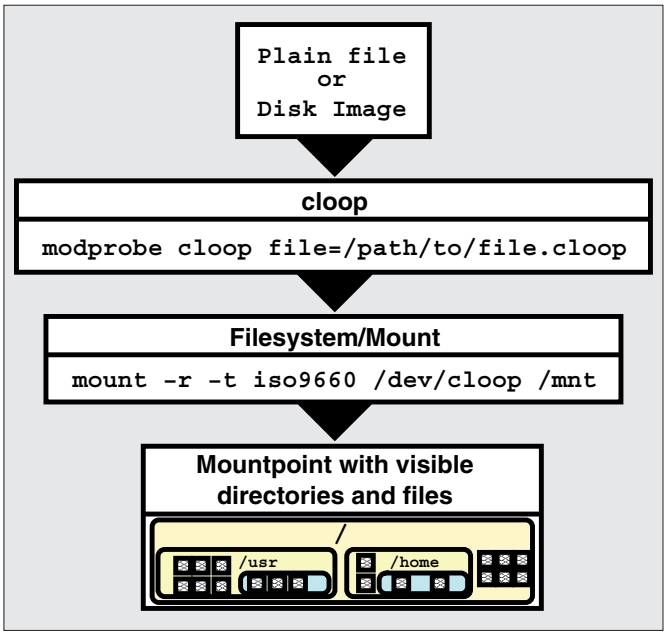


Figure 1: (C)loop maps a hard disk image file to a block device and lets you mount the image like a real hard disk partition.

disk, optionally with encryption. However, this driver still only provides a translation between different data representations of the same size.

Cloop, which was first written in 1999 for kernel 2.2 by iptables author Paul 'Rusty' Russell, uses a blockwise compressed input file that reduces the necessary space to around 1/3 of the original size for executable files. I ported, and later rewrote, cloop for newer kernel versions, extending the module to support 64-bit file access (thus allowing file sizes beyond 4GB), multiple input files, and *ioctl* extensions to permit the exchange of input files on-the-fly.

Since cloop is a block device and not a filesystem like squashfs, the compressed image can be anything, from raw data to arbitrary filesystem or database structures. On a Live CD, the cloop image usually contains an iso9660 file system, which is the most popular read-only and read-optimized filesystem used for data CDs. Cloop supports all filesystem features necessary for a Unix system through the standard RockRidge extension (long file names, permissions, symlinks, etc.).

Cloop also comes with several validity checks that return non-fatal error codes to the calling application in case invalid compressed data is read or read errors occur (scratches on the CD surface sometimes make data partly unreadable). Cloop only uses data that has been successfully read and decompressed, and it never terminates or interrupts access to the underlying device. That way, even partly damaged data can still be restored, and crashes because of read errors are unlikely (though, of course, an executable program that has been damaged will most likely refuse to run and terminate with a segmentation fault). This feature may give cloop a slight advantage against compressing filesystems, in which an entire file becomes unreadable if there is a read error right at the beginning of that file. In cloop, at maximum, one block is missing when read errors occur within this block.

One disadvantage is that cloop is read-only. Attempting to rewrite cloop to support write operations would probably be very inefficient because, at each write operation, the compressed block structure would have to be reordered, and blocks on a block device can never be "deleted," since the block device has no information about whether or not data is actually needed.

Also, when compressing a block, the resulting block size is not really predictable. In fact, a compressed block can be slightly larger than an uncompressed block, in case that the data is already stored compressed. When exchanging compressed blocks, the block index table would have to be recalculated (all offsets after the changed block will also change). Therefore, cloop is very optimized for reading data, but writing is not currently supported and probably never will be.

If you need write support on read-only media, you are better off with a filesystem such as AuFS or Unionfs that merges the read-only directory with a directory offering write support.

Cloop can handle parallel accesses to multiple cloop image files. If you have to split files because of size limits in the underlying filesystem, this is important (i.e., the 2GB and 4GB maximum file size in FAT32 and iso9660, respectively).

Listing 1: Character and Block Device Nodes

```
01 crw-rw---- 1 root root  10, 1 2007-10-18 10:41 /dev/psaux
02 brw-rw---- 1 root disk   3, 0 2007-10-18 10:41 /dev/hda
03 brw-rw---- 1 root disk 240, 0 2007-11-01 21:38 /dev/cloop
```

Listing 2: Attaching a New Cloop Image

```
01 $ sudo losetup /dev/cloop1 /media/cdrom/backup.cloop
02 $ sudo mount -r -t ext2 /dev/cloop1 /mnt
03 $ dmesg | tail -2
04  cloop: Initializing cloop v2.622
05  cloop: loaded (max 8 devices)
06  cloop: losetup_file: 3571 blocks, 65536 bytes/block, largest block
    is 65562 bytes.
```

Listing 3: Starting the Cloop Block Device

```
01 register_blkdev(major=240, cloop_name="cloop");
02 for(i=0; i<cloop_max; i++) cloop_dev[i] = cloop_alloc(i);
03 if(file) {
04  initial_file=filp_open(file,O_RDONLY|O_LARGEFILE,0x00);
05  cloop_set_file(0,initial_file,file);
06 }
```

Cloop images can be attached and detached at run time using the standard *losetup* command (see Listing 2).

Cloop can work with input files that are mounted over NFS. This feature is important if you plan to run completely diskless clients with filesystems mounted over cloop. Another interesting feature of cloop is that it can read and decompress data asynchronously using a kernel thread, so it's not blocking a process group and not staying in kernel space for a noticeable time.

Access to cloop image files can be suspended while they are in use (a feature introduced by Fabian Franz). The idea behind this feature is that you can temporarily remove a storage device that is in use by cloop without getting read errors. After receiving a special *ioctl* call, cloop will patiently wait until the underlying file is activated (and re-analyzed) again, so you could eject a Live CD while an OS is running from it.

One drawback is that all processes that access the cloop device will be blocked until the underlying file is reinserted. In the worst case, the desktop will freeze, which could make it hard to send the command for unfreezing cloop.

Kernel Block Device Components

To start a kernel 2.6.x block device in the module initialization, you need to call a few kernel procedures. Listing 3 should give you the basic idea, although the code in Listing 3 does not include extra features such as error checks and bailout procedures.

Listing 4: Disk Operations

```
01 static struct block_device_
   operations clo_fops =
02 {
03    owner:   THIS_MODULE,
04     open:   cloop_open,
05  release:   cloop_close,
06    ioctl:   cloop_ioctl
07 };
08
09 ...
10
11 ((struct gendisk *) clo_
   disk)->fops = &clo_fops;
```

If a file is given as module parameter *file* = "*/path/to/first/image*", this file is opened and associated with the first cloop device */dev/cloop0*.

How does */dev/cloop0* know what to do when a process opens the block device and reads from it? Unlike filesystems, block devices have a more complex way of performing input and output.

Depending on which kind of block device you are using, there are some operations that have to be supported and others that won't. In case of a disk or partition, you can use a structure like the one in Listing 4, which tells the kernel what to do if someone opens or closes a */dev/cloop** file.

cloop_open() needs to increase the use counter of the device so that the module knows it is in use. *cloop_close()* does the opposite. *cloop_ioctl()*, which is a special case that handles *losetup* for exchanging the underlying cloop image file, also suspends device operations in case the *CLOOP_SUSPEND ioctl* is sent.

Block devices have two methods for reading from the device. The direct method is setting up a request function that will be called each and every time a *read()* on the device is performed. This request function is then associated with the default block queue that has been created for the major device ID of cloop when the block device was set up (see Listing 5a).

cloop_request_fn() has to find out which device was accessed and then transfer data from the cloop file into the memory space that was given in *buffer_head*. This direct method was used until cloop version 2.06.

A disadvantage of this direct method is that the entire device is blocked until *cloop_request_fn()* returns. Actually, probably because of this, the direct method ceased to work for cloop starting from kernel 2.6.22.

Per-Device Wait Queue

The new approach to block device I/O is to use a per-device wait queue. With this

Listing 5a: Setting Up a Direct Callback

```
01 static int cloop_request_fn(request_queue_t *q, int rw, struct
   buffer_head *bh)
02 { /* do something with the read request ...  */ }
03
04 blk_queue_make_request(BLK_DEFAULT_QUEUE(cloop_major=240),
                          cloop_request_fn);
```

Listing 5b: Adding a Wait Queue

```
01 /* Called while queue_lock is held by kernel. */
02 static void cloop_do_request(struct request_queue *q) {
03  struct request *req;
04  while((req = elv_next_request(q)) != NULL) {
05   struct cloop_device *clo = req->rq_disk->private_data;;
06   blkdev_dequeue_request(req); /* Dequeue request first. */
07   list_add(&req->queuelist, &clo->clo_list); /* Add to working list
   for thread */
08   wake_up(&clo->clo_event);    /* Wake up cloop_thread */
09  }
10 }
11
12 ...
13
14 cloop_dev[i].clo_queue = blk_init_queue(cloop_do_request,
   &clo->queue_lock);
15 cloop_dev[i].clo_disk->queue = cloop_dev[i]->clo_queue;
```

method, the kernel collects requests from various sources in a linked list and occasionally delivers them to a procedure in cloop.

For slow physical reads, letting the kernel collect read requests and deliver them all at once makes the rest of the system act more efficiently, since less time is spent in kernel space.

Cloop version 2.622 takes each request out of the block device queue and puts it into an internal queue, transferring it to a per-device kernel thread, which does the real work with lower priority and no spinlock. This means that the block device I/O scheduler never blocks, because it does not have to wait for a slow physical I/O to complete (Listing 5b).

The queue has to be added to the disk associated with the cloop device, and this requires a lock that can be held by the kernel when requests are processed in order to avoid parallel queue manipulation. A kernel thread is created for handling the real work of processing the internal queue of read requests (see Listing 6).

cloop_handle_request() will now read blocks from the cloop image file, decompress them into memory, and transfer the parts of decompressed data that were requested by the calling process to that process' buffer.

The instructions can be arbitrarily complex and take as long a necessary because *cloop_handle_request()* is running in the kernel thread, which is a separate process and does not block the entire system.

Because the request had been assembled from block I/O segments, it must be divided into data-to-be-read units, which is what *rq_for_each_bio()* and *bio_for_each_segment()* do (Listing 7).

cloop_load_buffer() consists of a physical read procedure and a decompressor based on the kernel's internal *uncompress()*, which is also used for decompressing the initial ramdisk and some kinds of compressed data packets in network protocols.

Explaining kernel-library methods like *do_generic_read()* would be an article in itself, but it is fair to say that they basically do the same thing as *read()* or *fread()*, which you may know from the C library, just on the (much more complex) kernel layer, which has a very low-level view of files.

cloop_load_buffer() expects the block number of the block-to-read, which can be calculated from the byte offset of the data section requested by a process, and the block size used in the cloop image file associated with the device.

Listing 6: A Kernel Thread Processes Requests

```
01 static int cloop_thread(void *data) {
02  struct cloop_device *clo = data;
03  current->flags |= PF_NOFREEZE;
04  set_user_nice(current, -20);
05  while (!kthread_should_stop()||!list_empty(&clo->clo_list)) {
06   struct list_head *n, *p;
07   int err = wait_event_interruptible(clo->clo_event,
08             !list_empty(&clo->clo_list) || kthread_should_stop());
09   if(unlikely(err)) continue;
10   list_for_each_safe(p, n, &clo->clo_list) {
11    int uptodate;
12    struct request *req = list_entry(p, struct request, queuelist);
13    spin_lock_irq(&clo->queue_lock);
14    list_del_init(&req->queuelist);
15    spin_unlock_irq(&clo->queue_lock);
16    uptodate = cloop_handle_request(clo, req); /* do the read/
   decompression */
17    spin_lock_irq(&clo->queue_lock);
18    if(!end_that_request_first(req, uptodate, req->nr_sectors))
19     end_that_request_last(req, uptodate);
20    spin_unlock_irq(&clo->queue_lock);
21   }
22  }
23  return 0;
24 }
25
26 ...
27
28 clo->clo_thread = kthread_create(cloop_thread, clo, "cloop%d",
                                    cloop_num);
```

Creating a Cloop Image File

After this short walk through the kernel side of cloop, the next step is to consider the image file. The cloop file format (see Figure 2) is much less complicated than the module source would suggest.

The 128 bytes that appear at the beginning of the cloop image file are for future extensions and the possibility of making an "executable" cloop image file. Usually, these 128 bytes contain a shell script that will call modprobe with the right parameters in order to activate this image, so that you can attach the image to a cloop device by typing its pathname. The version number within the header is used to distinguish between older versions that still use 32-bit numbers.

In general, all numbers contained in the header of a cloop file have an architecture-independent network byte order, so that the same image will work on big-endian as well as little-endian systems.

Cloop assumes uncompressed blocks of constant size (at least within the same image file) because block devices always use a constant block size, and it's easier to calculate block numbers by the total size of a partition that way. The

block size is present in the header of a cloop file.

Compressing blocks of constant size, however, leads to different sizes in the compressed output, depending on how compressible the contained data is. Because of this, the image file needs an index of compressed block locations at the beginning, before the compressed data starts.

The compressed data, block after block, follows until the end of file is reached. With the block index in the header part, cloop knows where the compressed blocks can be found within the data part.

Although decompressing a cloop file is most conveniently done by just copying from */dev/cloop* to a partition or file, compression is done via a userspace program called *create_compressed_fs*.

This program is misnamed because *create_compressed_fs* does not compress a filesystem, but arbitrary data in cloop format. The most basic call to *create_compressed_fs* is:

```
create_compressed_fs ➥
inputfile  blocksize > ➥
outputfile
```

The use of - as the input file name will cause a read from stdin. *create_compressed_fs* can be used as a pipe, without the need of a temporary file, so you can make compressed backups on CD:

```
mkisofs -l -R /
home/username |➥
 create_
compressed_fs -
 65536 |➥
 cdrecord -v -
```

Because *create_compressed_fs* has to write the block index header before the data, the entire compressed image is stored in virtual memory (ram + swap) until all data is compressed and indexed, so *cdrecord* will start writing at the end of the compression process.

Make sure you have enough swap space available when using this method.

Adding the *-b* ("best") option will try *gzip-0* (uncompressed) to *gzip-9* (best gzip compression) and *7zip* compression (gzip-compatible mode) one after the other and use the smallest result. This compresses about 7% better than *gzip-9* (the default) alone; the decompressor automatically finds the right decompression method of each compressed block.

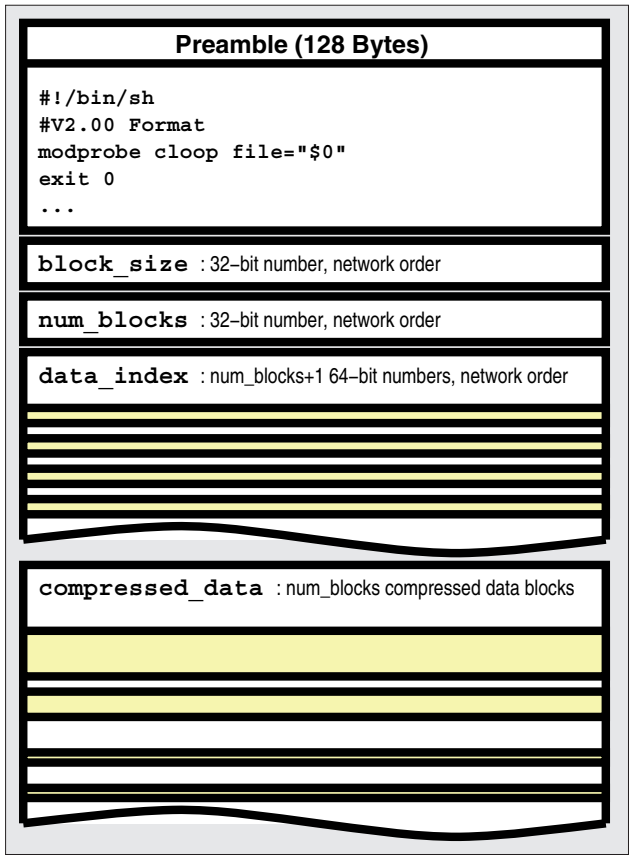


Figure 2: Structure of a cloop input file.

Debian maintainer Eduard Bloch has rewritten *create_compressed_fs* and added an alternative syntax that allows using temporary files instead of memory, threaded/parallel compression for multiprocessor machines, and daemon mode, which allows setting up a cluster of computers for fast compression of huge amounts of data. When installing the cloop-utils from Debian, you will get this newer version of *create_compressed_fs*, which is also present in the current cloop source.

Sources and Compiling

The most current source code for cloop is always at *http://debian-knoppix.alioth.debian.org/*. Compilation should succeed with

```
make KERNEL_DIR=➥
/path/to/your/kernel/sources
```

which will build the cloop module *cloop.ko* as well as the cloop compression utility *create_compressed_fs*. ■

Listing 7: Splitting a Request

```
01 /* This function does all the real work. */
02 /* returns "uptodate" */
03 static int cloop_handle_request(struct cloop_device *clo,
04                                  struct request *req) {
05  struct bio *bio;
06  rq_for_each_bio(bio, req) {
07   struct bio_vec *bvec;
08   int vecnr;
09   bio_for_each_segment(bvec, bio, vecnr) {
10    /* read compressed data from file, decompress,
11       transfer data to process */
12    ...
13    buffered_blocknum = cloop_load_buffer(clo,block_offset);
14    ...
15   }
16  }
17 }
```