Porting the HAMMER File System to Linux

Daniel Lorch

June 10, 2009

Abstract

Hammer [10] is a file system developed by Matt Dillon for DragonFly BSD. Hammer features large file systems up to 4096 TB, fine grained history retention (without requiring explicit checkpoints) and instant crash recovery.

Hammer is only available for DragonFly BSD. My contribution consists in providing a port of this filesystem for Linux. This port is restricted to read operations only.

1 Introduction

File systems in DragonFly BSD and Linux essentially interact with two interfaces in the kernel. Firstly, there is the VFS (Virtual Filesystem Switch [3] [20]) which provides an abstract interface for system calls like 'open' or 'read' to concrete file systems like Hammer or UFS.

Secondly, there is the block device (typically a hard disk). The block device is tightly interacting with the buffer cache. In Linux, the buffer cache is unified with the page cache [1].

This assignment consisted in learning about the necessary interfaces in both operating systems and providing a mapping between the two, while trying to leave most of Hammer's original code unmodified.

Along with VFS and the buffer cache, the kernel's thread primitives and locking primitives had to be considered. Since only read support was required, most of the locking code has been turned into no-ops. Background threads for flushing operations and other tasks have been deactivated due to the same reason.

The remainder of this document is organized as follows: section 2 illustrates the motivation of porting Hammer to Linux, followed by a brief presentation of its main features in section 3. A lot of the work has gone into reading source code and documents, as well as evaluating the required tool set, which is detailled in section 4. The actual porting effort is then described in section 5. Finally, a conclusion is presented in section 6 and acknowledgements are given in section 7.

2 Motivation

The number of people using Hammer is limited by the number of users running DragonFly BSD. But how many users are there?

Some methods have recently been suggested to measure the active user base of an open source project [2], but to date, they have neither been applied to DragonFly BSD nor to Hammer.

While exact numbers are not known, DragonFly BSD has an active and very dedicated user base. This can be observed by looking at DragonFly's mailing list archive, as well as the commit history of DragonFly's main git repository, showing constant activity. Furthermore, DragonFly had been accepted into Google's Summer of Code programme in 2008 and 2009 [13], showing its importance as an open source project.

For Linux, different estimation methods have been applied to measure its installed base. The Linux Counter project [7] conjectures that there must be 29 million Linux users (desktop and servers) by march 2005, basing their approximation on studies by Gartner and Netcraft. IDC [12] thinks that there must be 2.8 million Linux servers (considering paid as well as free distributions).

Without comparing numbers, it is safe to say that the number of Linux installations is by orders of magnitude larger than the number of DragonFly BSD installations.

This provided my main motivation to engage in the porting effort. Since Hammer is only available in DragonFly BSD, very few people have the opportunity to actually try it out. Exposing the file system to a wider audience will also increase the number of peer reviewers, providing valuable input to the development of Hammer on DragonFly.

3 A Hammer File System Walkthrough

3.1 Fine Grained History Retention

One of Hammer's most intriguing features is its ability to keep a history of all modifications made to its files and directories, in a similar fashion to version control software. Each modification creates a new 'transaction id', which is a 64-bit long identifier and looks like `0x00000001061a8ba6`.

A given file at a given time can then be retrieved by appending '@@' and its transaction id to it. This also works for directories and is the main mechanism to provide access to consistent 'snapshots' of entire directory trees.

The following example shows how a file is created and subsequently modified. The 'hammer history' command lists all available versions of the file along with the date and time when the transaction occurred.

```
# echo Hello > test
# echo World >> test
# hammer history test
test 000000010061aac0 clean {
00000001007a1520 23-Mar-2009 20:04:11
00000001007a1580 23-Mar-2009 20:04:43
}
# cat test@@0x00000001007a1520
Hello
# cat test@@0x00000001007a1580
Hello
World
```

```
# cat test
Hello
World
```

Snapshot creation is done using the ‘hammer snapshot’ command. Note that this will simply create a softlink to the given directory along with its current snapshot id, as the following example illustrates:

```
# hammer snapshot /mnt /mnt/snap
/mnt/snap
# ls -l snap
lrwxr-xr-x 1 root wheel 25 Mar 23 20:07 snap -> /mnt/
   @@0x00000001007a15c0
# ls snap/
test
```

3.2 Pruning

Since all of the file system’s history is kept permanently (unless the the ‘nohistory’ option was provided on mount), deleting files will not actually free any space and file system usage will continue to grow until the disk is full.

Pruning allows old generations of files to be deleted permanently. The ‘hammer prune’ command is clever in that it will not delete a file if it is pointed to by any snapshot softlink.

3.3 Pseudo File Systems

‘Pseudo File Systems’ are a feature of Hammer to create the equivalent of a disk partition, but inside the file system.

Typically, pseudo file systems are used to devise different policies for backup, pruning and mirroring (see subsection 3.4 for the latter) to a given subdirectory. While the user can specify that a given directory should not generate any history records by applying ‘chflags nohistory’ to it, pseudo file systems have the advantage of having their independent i-node number space, making it suitable as a target for replication.

3.4 Master-Slave Replication

Hammer supports a single-master, multiple-slave replication scheme. For this to work, one has to create a pseudo file system (subsection 3.3), designate it as master and may then define multiple slaves. Masters and slaves are paired using an unique ‘uuid’.

Mirroring is initiated manually with the ‘hammer mirror-copy’ command. Since transaction id’s are strictly incremental, the master and slave must only negotiate on the range of transactions that need to be transferred.

Mirroring can span multiple computers and be performed remotely, e.g. the ‘hammer mirror-copy’ command conveniently opens an SSH connection to a remote host and performs the mirroring through a secure connection. This is different from a simple ‘rsync’ in that the entire file system history is replicated along with it.

4 Tool Evaluation and Interfaces Study

4.1 Virtualization Software

Developing in the kernel on a live machine is very unpleasant, especially when programming errors occur. For Linux kernel development, several sources on the Internet suggest to use two physical machines interconnected with a serial cable and a specialized debugging software. Since the work I am doing is neither performance-critical, nor depending on specific hardware features, a virtualization software can be used for development.

VMWare provides a solution to simulate [18] the aforementioned setup on a single machine. While I managed to initate a remote connection with gdb, I was unable to load the debug symbols, probably since my host machine was running a different operating system (OS X) than the guest.

I have then decided to try User-Mode-Linux [21]. UML runs the Linux kernel as a userspace process, analogous to DragonFly BSD's virtual kernels [5]. These are the two virtualization solutions I finally ended up using, as they allow for a quick recompile-restart cycle and can be debugged easily with standard gdb.

4.2 Standalone Hammer File System

After installing DragonFly BSD, I created a partition for Hammer, put some test files and directories on it and finally dumped the result to a file using 'dd'. Having such a dump allowed me to experiment with `hammerread.c` [4], which is a read-only implementation of the Hammer filesystem for DragonFly's bootloader. When compiled with `-DTESTING`, the result will be a standalone binary.

The output of this program is as follows:

```
# cd /usr/src/lib/libstand
# cc -DTESTING -std=c99 hammerread.c -o hammerread
# ./hammerread
usage: hammerread
# ./hammerread /dev/ad1s0e
signature: valid
name: HAMMER
# ./hammerread /dev/ad1s0e /test
signature: valid
name: HAMMER
/test 0/0 100644 12
Hello
World
```

I subsequently ported this file to Linux [17], giving me a first working Hammer implementation on the destination platform.

4.3 Porting to FUSE

It seemed natural to wrap `hammerread.c` into a File Systems in Userspace module. Looking at the hello world example for FUSE [15], the functions that are required corresponded more or less directly to the functions defined in `hammerread.c`, e.g. `hstat()`, `hread()` and `hreaddir()`.

I added some glue code and released the result [16] to the Hammer newsgroup. The work inspired others and shortly after, Jeremy C. Reed released a port to NetBSD [19].

4.4 B+Trees

Trying to educate myself on the inner workings of the file system, I read Matt Dillon's design documents [8] [9] and about B+Trees [6] [11].

Hammer's B+Tree is a modified B-Tree, where nodes have additional boundary information. This boundary information allow searches to terminate early and speeds up lookups. Furthermore, searches must not begin at the root, but at any intermediary node (starting at a given location, e.g. the current directory).

The B+Tree nodes have a very high fanout degree (63) to reduce the number of disk accesses involved locating an element.

4.5 Linux Kernel Modules

The next logical step was to develop a Linux kernel module. This is described in detail in a tutorial [14]. The interface for modules that are built into the kernel and those that are loaded on demand is the same, so for a developer there is very little left to do.

Several simple file systems provided examples on how Linux VFS works, among them Ravi Kiran's [22] proved to be a very illustrative example.

5 Porting Work

5.1 VFS Interface

Linux and DragonFly BSD provide an analogous API for the VFS. The actual code behind these APIs is unfortunately very different (different function names, different function arguments).

Rather than providing a generic adapter for VFS, I decided to rewrite the VFS part from scratch and make calls to the Hammer backend where appropriate. Luckily, the Hammer source code is very well abstracted and had been written with porting in mind.

5.1.1 Vocabulary

There are some differences when it comes to the vocabulary used between the two operating systems.

In DragonFly BSD, the on-disk data structure representing a file is called an i-node. The data structure in-memory representing a file's contents is called a v-node. When interfacing with the VFS, one therefore has to deal with v-nodes.

In Linux, the term 'i-node' is used interchangeably for data structures on disk and those in memory. When interfacing with the VFS, one therefore has to deal with i-nodes.

5.1.2 Superblock Operations

For super block operations (mounting, unmounting, etc..), DragonFly defines a `struct vfsops`, having a `vfs_mount` function.

In Linux, the anologous function in `struct file_system_type` is called `get_sb`. This function fills the `struct super_block` data structure with information about the file system.

The first difference, which will become more apparent in subsubsection 5.1.5, is that access to the underlying block device is done via the `struct super_block` (which contains a pointer to the related block device) in Linux, but via the root v-node (the v-node representing the block device) in DragonFly BSD.

There is also a subtle difference when it comes to parsing optional arguments to the ‘mount’ command (those specified with ‘-o’). In DragonFly BSD, a call to ‘mount -t hammer’ will actually execute the program ‘mount_hammer’ in place. This programm will do the parsing of optional parameters, allocate a ‘struct’ and pass a memory pointer to VFS.

In Linux, the ‘-o’ string is copied verbatim to the mount command and parsing is done in the kernel.

5.1.3 Address Space Operations

In Linux, the mapping of a file to page cache can be manipulated with the `struct address_space_operations` [20], providing functions for reading pages from disk and writing them back. Implementing these functions is necessary to support ’mmap’, as well as making use of the page cache.

In DragonFly BSD, the corresponding functions `getpages` and `putpages` are available in the `struct vop_ops`.

5.1.4 I-node, File and Directory Operations

In DragonFly BSD, the convention is to put i-node, file and directory operations in a file called ‘foo_vnops.c’. The `struct vop_ops` specifies all the **v**node **op**erations on a given file system. Typically, only a single such structure is defined and the corresponding function, e.g. the read function, checks whether the target i-node is a file or directory and raises an error if a wrong entry type is encountered.

In Linux, the convention is to split this functionality into seperate files ’dir.c’, ’file.c’ and ’inode.c’. The `struct file_operations` applies to open files, whereas the `struct inode_operations` to i-nodes. Typically, multiple `struct inode_operations` are defined, one for each type (e.g. one for directories, one for all types of files).

5.1.5 Buffer Cache Interface

In DragonFly BSD, access to an underlying block device is done via the `bread()` (short for ‘block read’) call. `bread()` takes as argument the v-node of the given device (the root v-node), a byte-granular offset and the number of bytes that should be read.

In Linux, the analogous call is `sb_bread()`. This function takes the file system’s super block as an argument, where a block size (typically 1024 Bytes and must be smaller than the page size) had been defined beforehand using sb_set_blocksize. Reads can only occur in granularity and size of this block size.

For the given port, the bread() function was emulated in Linux by allocating a sufficiently large block of memory using kmalloc(), iteratively calling sb_bread() and copying the data with memcpy to that large block.

5.2 Wash, Lather, Rinse, Repeat

When porting the file system to Linux, a rather crude approach was taken. After defining a basic skeleton module for Linux [14], the two wrapper files dfly_wrap.h and dfly_wrap.c were created. Then, the source files were copied verbatim from DragonFly BSD to dfly/.

For each of the given .c files in dfly/, I created a new file including the wrapper, then the original source file. Following is an example for hammer_prune.c:

```
#include "dfly_wrap.h"
#include "dfly/vfs/hammer/hammer_prune.c"
```

Then, I compiled the project and observed the errors:

```
daniel@daniel:~/linux-2.6.29.1$ make ARCH=um 2>&1 |
    grep 'error: ' | sed -e 's/.*error: //g' | sort |
    uniq
'EFTYPE' undeclared (first use in this function)
'FREAD' undeclared (first use in this function)
'FSCRED' undeclared (first use in this function)
'FWRITE' undeclared (first use in this function)
'LK_EXCLUSIVE' undeclared (first use in this function)
'LK_RETRY' undeclared (first use in this function)
...
```

This is the screen I have been looking at for several weeks. For each of the given compile time errors (like missing structs), I searched for the appropriate definition in DragonFly's source code and added it to dfly_wrap.h.

Analogously, for each of the link time errors, I added a stub method to dfly_wrap.c raising a kernel panic when called, for example:

```
int nlookup(struct nlookupdata *nd) {
    panic("nlookup");
}
```

Certain files, like hammer_vnops.c, could be omitted, as a corresponding implementation in Linux VFS was created. Other files, which required to be rewritten entirely (e.g. hammer_io.c), were copied and the modifications were made in-place.

Interestingly, out of the 18 source files ported, 14 files could be included directly using the wrapper trick mentioned above, and only 4 files had to be modified in-place.

After the file system compiled and linked, the final step consisted in actually running the code and following each of the kernel panics, providing a compatible wrapper implementation where necessary or turning it into a no-op function.

6 Conclusion

Porting a file system was a study of interfaces of two different operating systems, and not the study of the file system itself, as one might expect. In most parts of the porting effort, the file system could be treated as a black box. Not surprisingly, it was not long into development cycle as I began to explore many of Hammer's features. All of the functionality is well abstracted – from access to I/O to locking, almost all functions are wrapped around functions that made the porting effort less painful.

In the current code, the functions 'getattr', 'read', 'readdir' are implemented. This is enough to mount a Hammer device, list its contents and read from a file. More effort is required to support writing and to port the userland utilities. Since the source code is released to the community I hope to find more contributors to help with the porting effort.

7 Acknowledgements

I would like to thank Simon Schubert, my assistant, and Matt Dillon, the Hammer author, for having taken their time to help me with the porting effort. I have always immediately received a response to all my questions – either on IRC, on the newsgroup, on the phone or from Simon in his office.

References

[1] Tigran Aivazian. Linux kernel 2.4 internals: Linux page cache. http://tldp.org/LDP/lki/lki-4.html, 2002. [Online; accessed 7-June-2009].

[2] Kevin Crowston Andrea Wiggins, James Howison. Heartbeat: Measuring active user base and potential user interest. http://www.slideshare.net/AniKarenina/heartbeat-measuring-active-user-base-and-potential-user-interest, 2009. [Online; accessed 9-June-2009].

[3] DragonFly BSD. The new vfs model. http://www.dragonflybsd.org/goals/index.html#vfsmodel, 2009. [Online; accessed 7-June-2009].

[4] DragonFly BSD. Standalone hammer filesystem implementation. http://gitweb.dragonflybsd.org/dragonfly.git/blob_plain?f=lib/libstand/hammerread.c, 2009. [Online; accessed 7-June-2009].

[5] DragonFly BSD. Vkernel. http://leaf.dragonflybsd.org/cgi/web-man?command=vkernel, 2009. [Online; accessed 7-June-2009].

[6] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[7] Linux Counter. Linux counter: Estimates of the number of linux users. http://counter.li.org/estimates.php, 2009. [Online; accessed 9-June-2009].

[8] Matt Dillon. The hammer filesystem. http://www.dragonflybsd.org/hammer/hammer.pdf, 2008. [Online; accessed 7-June-2009].

[9] Matt Dillon. The hammer filesystem at nycbsdcon 2008. http://www.dragonflybsd.org/presentations/nycbsdcon08/, 2008. [Online; accessed 7-June-2009].

[10] Matt Dillon. The hammer filesystem website. http://www.dragonflybsd.org/hammer/, 2009.

[11] Matt Dillon. Hammer's b+tree implementation. http://gitweb.dragonflybsd.org/dragonfly.git/blob_plain/HEAD:/sys/vfs/hammer/hammer_btree.h, 2009. [Online; accessed 7-June-2009].

[12] Al Gillen. The opportunity for linux in a new economy. http://www.linuxfoundation.org/sites/main/files/publications/Linux_in_New_Economy.pdf, 2009. [Online; accessed 9-June-2009].

[13] Google. Dragonfly soc 2009. http://socghop.appspot.com/org/home/google/gsoc2009/dragonflybsd, 2009. [Online; accessed 9-June-2009].

[14] The Linux Kernel Module Programming Guide. Hammer's b+tree implementation. http://www.tldp.org/LDP/lkmpg/2.6/html/, 2007. [Online; accessed 7-June-2009].

[15] FUSE: Filesystem in Userspace. Hello world example. http://fuse.sourceforge.net/helloworld.html, 2005. [Online; accessed 7-June-2009].

[16] Daniel Lorch. hammerread.c + fuse = read hammer filesystem on linux. http://hammerfs-ftw.blogspot.com/2009/04/hammerreadc-fuse-read-hammer-filesystem.html, 2009. [Online; accessed 7-June-2009].

[17] Daniel Lorch. hammerread.c ported to linux. http://hammerfs-ftw.blogspot.com/2009/04/hammerreadc-ported-to-linux.html, 2009. [Online; accessed 7-June-2009].

[18] Vyacheslav Malyugin. Debugging linux kernels with workstation 6.0. http://stackframe.blogspot.com/2007/04/debugging-linux-kernels-with.html, 2007. [Online; accessed 7-June-2009].

[19] Jeremy C. Reed. Netbsd hammer with fuse and hammerread. http://leaf.dragonflybsd.org/mailarchive/hammer/2009-04/msg00002.html, 2009. [Online; accessed 7-June-2009].

[20] Pekka Enberg Richard Gooch. Overview of the linux virtual file system. http://lxr.linux.no/linux+v2.6.29/Documentation/filesystems/vfs.txt, 2005. [Online; accessed 7-June-2009].

[21] User-Mode-Linux. Kernel hacking with uml. http://user-mode-linux.sourceforge.net/hacking.html, 2008. [Online; accessed 7-June-2009].

[22] Ravi Kiran UVS. Writing a simple file system. http://www.geocities.com/ravikiran_uvs/articles/rkfs.html, 2007. [Online; accessed 7-June-2009].

Appendix

Source Code

The source code can be retrieved with ‘git clone git@plan.nine.ch:hammerfs.git’ via anonymous git and is released under the DragonFly BSD license. No restrictions are imposed on the code and contributors are more than welcome.