Kernel File Systems: A Reusable Resource

Author
Affiliation
email

Author
Affiliation
email

Abstract

Writing code is easy although time-consuming. Making code work in the real world is usually substantially harder. Major open source operating systems provide a wealth of kernel code which not only already exists and is available for use, but has also been tested and proven in the real world. The ability to reuse this code in applications provides a library of functionality for free.

This paper shows that using existing kernel file system implementations as part of userspace applications is possible without modifying the kernel file system code base. Two different operating modes are explored: 1) a transparent mode, where the file system is mounted in the typical fashion by using the kernel code as a userspace server, and 2) a standalone mode, where applications can make file system calls without going through the kernel system call interface. The first mode provides isolation from the trusted computing base and a secure way for mounting untrusted file system images on a monolithic kernel. Additional uses include debugging and development. The second mode is useful for file system utilities, and applications, such as mtools. File system access is possible without host kernel support.

The design and implementation of a framework already integrated into NetBSD is discussed. Under this framework ten disk-based file systems and two other kernel file systems have been tested to be functional. The usefulness of both operating modes of the framework is examined. Although not optimizing for performance, it is measured to be acceptable: worst case 19% slower than the kernel. The prototype of a similar framework for Linux was also implemented and portability was verified: Linux file systems work on NetBSD and NetBSD file systems work on Linux.

1 Introduction

Motivation. *"Userspace or kernel?"* A typical case of driver development starts with that exact question. The tradeoffs are classically well-understood: speed, efficiency and stability for the kernel or ease of programming and a more casual development style for userspace.

The question stems from the different programming environments offered by the two choices. Even if code written for the kernel is designed to be run in userspace for testing, it is often crippled and does not support all the features. Additionally, such code is frequently cluttered with `#ifdef`'s.

Typical operating system kernels already offer a plethora of tested and working code just waiting to be used. An excellent example of this is file system code, which in the case of most operating systems even comes with a practical virtual file system [21] interface making code use independent of the file system type.

By making kernel file system code usable in userspace, readily available code can be used for free in applications. We have accomplished this by creating a shim layer to emulate enough of the kernel to make it possible to compile, link and run the kernel file system code. Additionally, we have created supplementary components necessary to fully integrate the file system code with a running system, i.e. mount it as a userspace file server. Our scheme requires no modification to pre-existing kernel file system code and therefore no added maintenance costs for supported file systems.

We define a *rump*, or Runnable Userspace Meta Program, to be kernel code used as part of a userspace program. Specifically, a rump file system is kernel file system code running in a userspace application or as a userspace file server.

Userspace file systems. While this paper touches file systems in userspace, it is by no means a paper about userspace file system frameworks. Userspace file system frameworks provide two things: a programming interface for the file server to attach to and a method for transporting file system requests in and out of the kernel. This paper explores running kernel file system code as an application in userspace. Our approach requires a userspace file system framework only in case mounting the resulting rump file system is desired. The choice of the framework is mostly orthogonal. We chose puffs [19] because it is the native solution on NetBSD. Similarly, would the focus of implementation have been for example Linux or Windows NT, the choice could have been FUSE [2] or FIFS [8], respectively.

Results. NetBSD [5] is a free, 4.4BSD derived OS running on over 50 platforms and used in the industry especially in embedded systems. A real world usable implementation for NetBSD, already integrated into the main source tree, has been done.

The following NetBSD kernel file systems have been *tested to be usable in userspace without source modifications*: cd9660, EFS, Ext2fs, FFS, HFS+, LFS, MSDOSFS, NTFS, puffs, SysVBFS, tmpfs, and UDF. After creating the initial support for FFS, support for additional file systems required at minimum nothing and at most small tweaks. The only major real file system not currently supported is NFS, as it requires a lot of additional support from the kernel networking subsystem.

Additionally, a quick prototype of a similar system for the Linux kernel has been implemented. Under it, the jffs2 [31] journalling file system from the Linux kernel is mountable as a userspace server on NetBSD. There is no reason other Linux file systems could not work using the same scheme, but as jffs2 is a reasonably simple file system, some extra work would be required to support them.

Contributions. This paper shows that it is possible and desirable to use pre-existing kernel file system code in userspace. The new contribution is the *use of kernel file system code in userspace applications*. In the quest for microkernel operating systems, the useful possibility of treating kernel code as a programming library resource instead of a kernel resource has been completely ignored.

While not strikingly novel, the paper also describes a way to make a monolithic style kernel operate like a multiserver microkernel. However, in contrast to previous work, it *gives the user the choice* of micro- or monolithic kernel operation, thereby avoiding the need to get into the whole microkernel performance discussion.

The paper also shows it is possible to use kernel code in userspace on top of a POSIX environment irrespective of the kernel platform the code was originally written for. This paves way to thinking about *kernel modules as reusable operating system independent components*.

Paper organization. The rest of this paper is organized as follows: Chapter 2 deals with issues related to the architecture. Some major details in the implementation are discussion in Chapter 3. The work is measured and evaluated in Chapter 4. Chapter 5 surveys related and prior work and finally Chapter 6 concludes and envisions future work.

2 Architecture

Before going into details about the architecture of the implementation, let us recall how file systems are implemented in a monolithic kernel such as NetBSD or Linux.

- The well-known interface through which the file system is accessed is known as the virtual file system interface [21]. It provides virtual nodes, *vnodes*, as abstract objects for the kernel to access files independent of the file system type.

- To access the file system backend, the file system implementation uses the necessary routines from the kernel. These are for example the disk driver for a disk-based file system such as ffs [23], the network for nfs or the virtual memory subsystem for tmpfs [29]. Access is usually done through the buffer cache.

- To maximize file content caching and provide for memory mapped I/O, a modern operating system is very heavily tied to the virtual memory subsystem [28]. In addition to the pager's get and put routines, various supporting routines are required. This integration also provides the page cache.

- Finally, a file system uses various kernel services. Examples range from a hashing algorithm to timer routines and memory allocation.

If the reuse of file system code in userspace is desired, all of these interfaces must be provided in userspace. As most parts of the kernel do not have anything to do with hardware but rather just implement algorithms, they can be simply compiled and linked to a userspace program. We call such code *environment independent* (EI). On the other hand, for example device drivers, scheduling routines and CPU routines are *environment dependent* (ED) and must be reimplemented.

2.1 Kernel and Userspace Namespace

To be able to understand the general architecture, it is important to note the difference between the namespaces defined by the C headers for kernel and for user code. Selection of the namespace is usually done with the preprocessor, e.g. `-D_KERNEL`. Any given module must be *compiled* in either the kernel or user namespace. However, after compilation the modules from different namespaces can be *linked* together, assuming that the application binary interface (ABI) is the same.

Code cannot use both namespaces simultaneously due to collisions. For example, on a BSD-based system libc `malloc`() takes one parameter while the kernel interface takes three. Trying to declare both variants will cause C compilation to fail. To solve the problem, we identify components which require the kernel namespace and

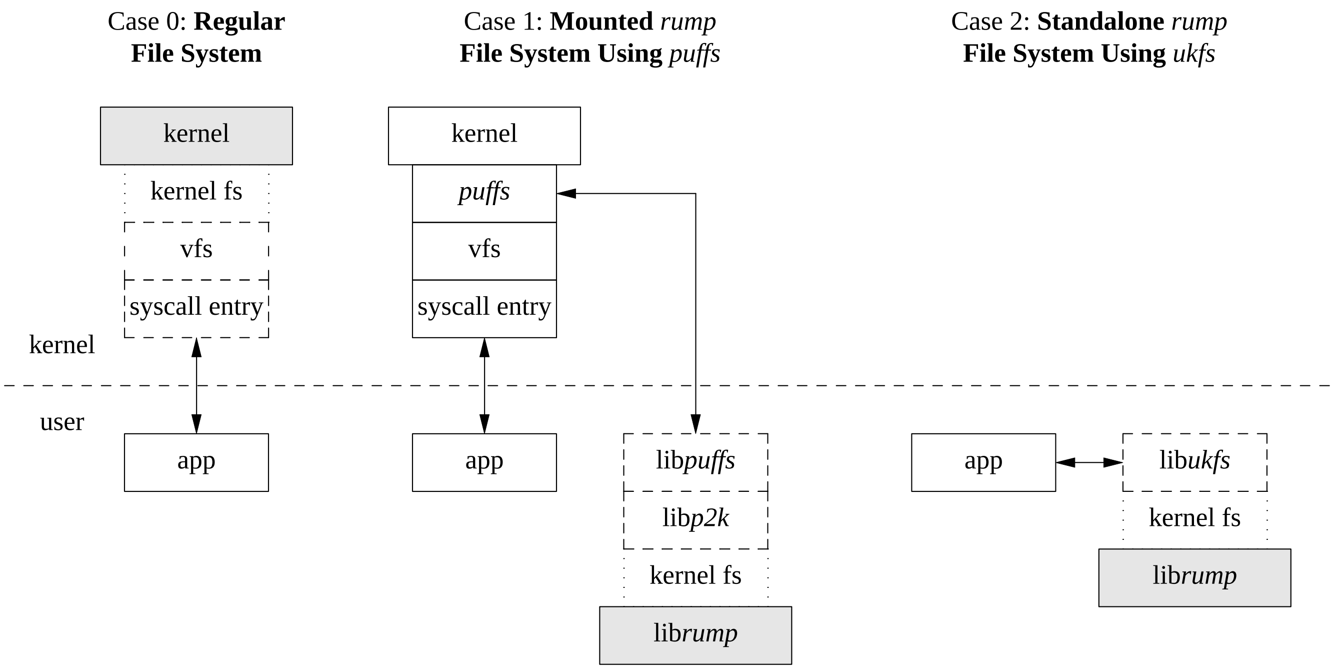

Case 0: **Regular File System**
Case 1: **Mounted** *rump* **File System Using** *puffs*
Case 2: **Standalone** *rump* **File System Using** *ukfs*
kernel
kernel fs
vfs
syscall entry
kernel
user
app
kernel
puffs
vfs
syscall entry
app
libpuffs
libp2k
kernel fs
librump
app
libukfs
kernel fs
librump

Figure 1: rump File System Architecture

components which require the user namespace and compile them as separate compilation units. We let the linker handle unifying them.

The issue is even more severe if we wish to use rump file systems foreign platforms. We cannot depend on anything in the NetBSD kernel namespace to be available on other systems. Worse, we cannot even include anything from the NetBSD kernel namespace in applications on other platforms, since it will create conflicts. For example, think what will happen if an application includes both the native and NetBSD `<sys/stat.h>`. Therefore, rump itself provides a namespace which applications can use to make calls to the kernel namespace. For example, the kernel vnode operation `VOP_READ`() is available under the name `RUMP_VOP_READ`().

2.2 Component Overview

The architecture of the framework is presented in Figure 1 using three different cases to illuminate the situation. Analogous parts between the three are signaled. The differences are briefly discussed below before moving to further dissect the components and architecture.

Regular File System (case 0). For comparison, ”Regular File System” shows the architecture relevant to running a file system in the kernel.

Mounted rump File System Using puffs (case 1). From the application perspective, a mounted rump file system looks and behaves like the same file system code running in the kernel. The NetBSD userspace file systems framework, puffs [19], is used to attach the file system to the kernel virtual file system.

Standalone rump File System Using ukfs (case 2). A standalone rump file system is not mounted into the normal file system namespace. Rather, applications use a special programming interface to mount and access the file system. While this requires specially crafting applications to access the file systems, it allows complete freedom for the applications, including the ability to specially target some certain file system and make calls past the virtual file system abstraction. Such a scheme is useful for example in very fine-grained testing.

2.3 The File System Code Itself

Kernel file systems are obviously kernel code. The compilation process itself, though, is like building a normal userspace library. The end result is a library of kernel file system code ready to be linked into a binary. The ABI of the library is regrettably currently not precisely equal to a kernel module, as we provide a userspace version of a few architecture dependent data structures. If the ABI were the same with the kernel, a file system kernel module could be used directly instead of requiring a separately compiled library.

2.4 librump

The interfaces required by a file system were classified in the beginning of Chapter 2. The component to provide these interfaces in the absence of the real kernel is librump. It emulates enough of the kernel environment for the file system code to be able to run.

Figure 2: Examples of *rumpuser* interfaces

```
int rumpuser_gettimeofday(struct timeval *tv,
          int *error);

ssize_t rumpuser_pread(int fd, void *buf,
          size_t bufsize, off_t offset, int *error);

int rumpuser_thread_create(void *(*f)(void *), void *);

void rumpuser_mutex_enter(struct rumpuser_mtx *);
```

Table 1: rump library size analysis

Component	# of lines
rumpuser	432
rumpkern (ED)	3099
std kern (EI)	19017
puffs (kernel)	3411
FFS	14912

2.4.1 Internal Division

librump is conceptually kernel code: it emulates the kernel proper. However, to successfully emulate the kernel in userspace, it must be able to make calls to certain interfaces in the user namespace, such as the read and write system calls used for accessing the file system image. The parts which make these calls cannot be compiled in the kernel namespace for reasons discussed in Chapter 2.1.

We split librump into two portions: *rumpkern* and *rumpuser*. rumpkern contains routines implementing the kernel interfaces, while rumpuser provides bridge interfaces to call interfaces in application namespace. rumpuser is used only by rumpkern and a means to defeat the namespace problem.

Figure 2 lists some example routines provided by the rumpuser. There are currently two main classes of calls provided by rumpuser: system calls and threading library calls. Additionally, some support calls such as memory allocation are provided.

A convenient observation is to note that the file systems only call routines within themselves and interfaces in our case provided by rumpkern. rumpkern only calls routines within itself, the file system (via callbacks) and rumpuser. Therefore, by closure, this makes rumpuser the component defining the portability of a rump file system. Since rumpuser is only a handful of calls and apart from diagnostic routines uses POSIX functionality, the portability of the system is very high.

2.4.2 Development Approach

As maintainability is an issue, we want to write as little code for librump as possible - kernel interface implementations which are duplicated in librump need to be modified if the kernel interface changes. The strategy is to identify environment independent modules from the kernel sources and compile them into librump directly.

The decision between environment dependent and independent modules is left to the implementor. However, the trend during development has been to increase the amount of code compiled directly from the kernel sources. It is likely that specifically-written code will diminish even more over time once solutions to the remaining problems become clearer and environment independent code suitable both for rumps and the kernel is created.

See Table 1 for a rough idea on how much code had to be written (rumpuser and rumpkern) and how much could be compiled directly from the kernel sources. The count is measured without comments or empty lines. The Fast File System and puffs are included for comparison to put us on the map about code size. While the number of code lines implemented for rump seems modest, it should be remembered that the real challenge is to keep the amount of reimplemented lines down and therefore maintainability up. The subject of code size will also be revisited in Chapters 4.3 and 4.5.

2.5 libp2k

Mounted rump file systems build upon the functionality provided by the NetBSD native userspace file systems framework, puffs [19]. We rely on two key features. The first one is transporting file system requests to userspace, calling the file server, and sending the results back to the kernel. The second one is the cleanup of an abruptly unmounted file system, such as in the case of a file server crash. The latter prevents any kernel damage in the case of a misbehaving file server.

puffs provides an interface for implementing a userspace file systems. While this interface is clearly heavily influenced by the virtual file system interface, there are multiple differences. For example, the kernel virtual file system identifies files by a `struct vnode` pointer, whereas puffs identifies files using a `puffs_cookie_t` value. Another example of a parameter difference is the `struct uio` parameter. In the kernel this is used to inform the file system how much data to copy and in which address space. puffs passes this information to the read interface as a pointer where to copy to along with the byte count - the address space would make no difference since a normal userspace process can only copy to addresses mapped in its vmspace. In both cases the main idea is the same, although the details differ.

The *p2k*, or puffs-to-kernel, library is our request translator between the puffs userspace file system interface and the kernel virtual file system interface. It also interprets the results from the kernel file systems and

Figure 3: p2k_node_read() Implementation

```
int
p2k_node_read(struct puffs_usermount *pu,
        puffs_cookie_t opc, uint8_t *buf,
        off_t offset, size_t *resid,
        const struct puffs_cred *pcr, int ioflag)
{
        kauth_cred_t cred;
        struct uio *uio;
        int rv;

        cred = cred_create(pcr);
        uio = rump_uio_setup(buf, *resid,
            offset, RUMPUIO_READ);
        VLS(opc);
        rv = RUMP_VOP_READ(opc, uio, ioflag, cred);
        VUL(opc);
        *resid = rump_uio_free(uio);
        cred_destroy(cred);

        return rv;
}
```

converts them back to a format that puffs understands. To receive requests from puffs for vfs translation, p2k pretends to be a file system and registers itself to puffs.

To give an example of p2k operation, we discuss reading a file. This is illustrated by Figure 3, which includes the full p2k read routine. We see the uio structure created by `rump_uio_setup()` before calling the vnode operation and freed after the call while saving the results. We also notice the puffs credit type being converted to the *kauth_cred_t* type used in the kernel. This is done by the p2k library's `cred_create()` routine, which in turn uses `rump_cred_create()`. The `VLS()` and `VUL()` macros in p2k to deal with NetBSD kernel virtual file system locking protocol. They take a shared (read) lock on the vnode and unlock it, respectively.

Mount utilities

Standard kernel file systems are mounted with utilities such as `mount_efs`, `mount_ffs`, `mount_tmpfs`, etc. These utilities parse the command line arguments and call the `mount()` system call to attach the file system as a part of the operating system namespace.

Our equivalent mountable rump file system counterparts are called `rump_efs`, `rump_ffs`, `rump_tmpfs`, etc. To maximize system integration and minimize differences with the counterparts, these utilities share the same command line argument parsing code with the mount utilities and therefore have the same syntax - this makes usage interchangeable. The rump utilities attach the file system via p2k and puffs.

2.6 libukfs

The ukfs, or user-kernel file system, library provides a standalone approach to using kernel file systems in userspace (Case 2 from Figure 1). It does not mount the file system as a part of the running operating system's directory namespace, but provides file system access via ukfs library calls instead of system calls. While this means that applications need special support for ukfs, there is no need for file system kernel support on the application host platform.

Figure 4: Examples of ukfs interfaces

```
struct ukfs *ukfs_mount(const char *vfstype,
               const char *devpath,
               const char *mntpath, int mntflags,
               void *arg, size_t arglen);

int ukfs_modload(const char *libpath);

ssize_t ukfs_read(struct ukfs *u, const char *file,
        off_t off, uint8_t *buf, size_t bufsize);

int ukfs_rmdir(struct ukfs *u, const char *dir);
```

Two classes of interfaces are provided by libukfs, and two interfaces from each class are shown in Figure 4. Both classes are discussed below:

Initialization. For a file system to be usable, it must be mounted. The mount routine returns a pointer handle of type `struct ukfs` which is passed to all other calls. This handle is analogous to the mountpoint path in a mounted file system.

Additionally, routines for dynamically loading file system libraries are provided. This is similar to loading kernel modules, but since we are talking about userspace, `dlopen()` is used and the file system module attach method is called. This facilitates creating and shipping applications independent of what file systems they support, provided of course that they only use interfaces above the virtual file system layer.

File system access. Accessing file system images is done with calls in this class. Most calls have an interface similar to system calls, but as they are all self-contained calls, all interfaces take a filename instead of for example requiring a separate open before passing a file descriptor to a call. The rootpath is the root of the mounted file system, but the library provides tracking of the current working directory, so passing non-absolute paths is also possible.

If an application wishes to make lower level calls for performance or granularity reasons, it is free to do so even if it additionally uses ukfs routines.

3 Implementation

Most of the details in implementation can be resolved with creative hacking and knowledge about the kernel file system and virtual memory internals. This section deals with major interests in the implementation. While the discussion is written with NetBSD terminology, it attempts to take into account the general case with all operating systems.

3.1 System Calls

As mentioned when discussing ukfs in Chapter 2.6, the ukfs interface is very similar to the system call interface. While the backend of each call is handled by the file system, in the kernel the system call entry points perform various tasks before calling the vfs interfaces. For example, the `mkdir`() system call does a lookup for node to be created and calls `VOP_MKDIR`() only if the desired node does not already exist.

A reimplementation of the kernel syscall entrypoints for file system calls was initially attempted. While it was possible to do so, getting all the error handling done exactly correctly proved to be a challenge. Therefore a scheme to use the kernel system call entry points directly was crafted.

In NetBSD, parts of the system call code are autogenerated from a master table. This is done for the benefit of e.g. automatic argument marshalling and demarshalling. We used this autogenerator to our advantage by marking system calls relevant to rump with ”RUMP” in the master table and then edited the code generator script to create entry points for rump. As a result, the interface `rump_sys_syscall`() performs the same tasks as `syscall`(), but instead of trapping to the kernel, makes only a function call to librump. For example, calling `rump_sys_pread`() does the same as `pread`() from a normal application (and of course `rump_sys_open`() must be called first get a valid file descriptor for use in the call).

3.2 Locking and Multithreading

File systems make heavy use of locking to avoid data corruption. Most file systems do not create separate threads, but use the context of the requesting thread to do the operations. In case of multiple requests there may be multiple threads in the file system and they must synchronize access. Still, some file systems create explicit threads, e.g. for garbage collection.

To support multithreaded file systems in userspace, we have four different issues to solve: locks, threads, interrupt priority level and legacy interfaces. The standard userspace library for supporting these routines is libpthread, so we map the kernel counterparts to libpthread.

Locks and condition variables. The primitives in NetBSD are modeled closely after the ones in Solaris [22]. There are three different classes: mutexes, rwlocks and cv’s. These map almost directly to pthread interfaces. The only differences are that the kernel routines are typically of type `void` while the pthread routines return a success value. However, as getting an error from e.g. `pthread_mutex_lock`() means a programming error such as trying to lock a mutex already locked by the calling thread, failures can be handled by asserts.

Threads. The kernel provides interfaces to create and destroy threads. Apart from some esoteric arguments such as binding the thread to a specific CPU, which we ignore, the kernel thread interfaces can be emulated directly by mapping to a pthread library call.

Interrupt priority level. Interrupt priority level is a monitor-type synchronization mechanism used especially in non-preemptive uniprocessor kernels to mask CPU interrupts during critical sections. NetBSD allows fine-grained control over which levels to disable and still uses the facility to some degree, although with MP kernels other locking is also required to protect critical sections from processors not running in interrupt context.

We map interrupt priority levels to pthread rwlocks. Raising the interrupt priority level is a per-CPU operation and is mapped to a read lock. This way multiple threads can ”mask” the same interrupts simultaneously, and a single thread can raise the ipl multiple times. An ”interrupt” is issued when a read or write operation on the storage device completes. This takes a write lock and therefore makes sure that no threads are currently executing in the critical section.

Legacy interfaces A historic BSD interface still in use in some parts of the kernel is `tsleep`(). It is a facility for waiting for events and maps to pthread condition variables.

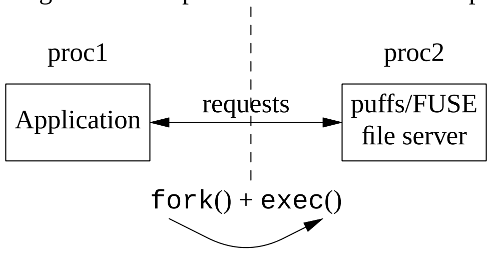
Observations. It is clear that the NetBSD kernel and pthread locking and threading interfaces are very close to each other. However, there are minimal differences such as the naming and of course under the hood the implementations are different. Providing a common interface for both [12] would be a worthwhile exercise in engineering for a platform where this was not considered initially.

3.3 puffs as a rump file system.

Using rump, puffs can be run in userspace on top of rump and a regular userspace file system on top of it. At first thought, running the kernel portion of a userspace file system framework in userspace sounds like a weird curiosity. However, upon closer examination, it gives the benefit of being able to access any file system via ukfs, regardless of whether it is a kernel file system or a userspace file system. Since puffs provides emulation for the FUSE interface as well, any FUSE file system is usable through the same interface too. For instance, a utility which lists the directory tree of a file system works regardless of if the file system is the NetBSD kernel FFS or FUSE ntfs-3g.

Naturally, it would be possible to call userspace file system interfaces from applications without a system as complex as rump. However, since we already do have rump, we can provide total *integration* for all file systems with this scheme. It would be entirely possible to make ukfs use different callpaths based on the type of

Figure 5: Userspace File Servers with rump



file system used. However, that would require protocol conversion in ukfs to e.g. FUSE. Since the puffs stack already speaks all the necessary protocols, it is much more elegant to run everything through it.

We assume userspace file servers are precompiled binaries and do not require any modification to them. Rather, we use a model with two processes. The userspace file server is executed in a separate process and operates completely normally. However, instead of receiving requests from the kernel it actually receives them from the application process. This is illustrated in Figure 5. The result is the ability run binary userspace file servers as rump file systems.

3.4 Installation

rump is installed for userspace consumers as libraries. There are a number of separate libraries: librump (rumpkern), librumpuser (rumpuser), libukfs, libp2k, and finally all the individual file system libraries, e.g. librumpfs_efs, librumpfs_ntfs and so forth.

Since the kernel environment is constantly in a flux, the standard choice of bumping the major library version each time the ABI changes did not seem reasonable - it would require a library ABI bump for every kernel ABI change. Instead, currently the compatibility between librump and the file system libraries is handled exactly like for kernel modules: both librump and the file system libraries are embedded with the ABI version they were built against. When a file system library is attached to librump, the versions are compared, and if incompatible the attach routine returns `EPROGMISMATCH`.

3.5 Foreign Platforms

Different kernel version. An interesting implication of rump file systems is the ability to mount file systems for a different kernel version. While it is possible to load and unload kernel modules on the fly, they are closely tied by the kernel ABI. Since a rump file system is a self-contained userspace entity, it is possible to use a file system from a newer or older kernel or even mix-and-match. Reasons include taking advantage of a new file system feature without having to reboot or avoiding a bug present in newer code.

NetBSD rumps on Linux. Running NetBSD rumps on Linux means using NetBSD kernel file systems on a Linux platform. As Linux does not support puffs, libp2k cannot be used. A port to FUSE would be required. Despite this, the file system code can be used via ukfs and accessing a file system image using NetBSD kernel code on Linux has been verified to work. A notable fact is that structures are returned from ukfs using the ABI from the file system, e.g. `struct dirent` is in NetBSD format and must be interpreted by callers as such. A platform independent format for ukfs may be devised later.

Linux kernel file systems on NetBSD. Running Linux kernel file systems on NetBSD is interesting because there are several file systems written against the Linux kernel which are not available natively in NetBSD or in more portable environments such as userspace via FUSE. Currently, our prototype Linux implementation supports only jffs2 [31]. This file system was chosen as the initial target because of its relative simplicity and because it has a potential real-world use case in NetBSD, as NetBSD lacks a wear leveling flash file system.

An emulation library targeted for Linux kernel interfaces, *lump*, was written from scratch. In addition, a driver emulating the MTD flash interface used by jffs2 for the backend storage was implemented.

Finally, analogous to libp2k, we had to match the interface of puffs to the Linux kernel virtual file system interface. The main difference was that the Linux kernel has the *dcache* name cache layer in front of the virtual file system nodes instead of being controlled from within each file system individually. Other tasks were mostly straightforward, such as converting the `struct kstat` type received from Linux file systems to the `struct vattr` type expected by puffs and the NetBSD kernel.

ABI Considerations. Linking code compiled with the NetBSD kernel headers to code compiled with Linux headers is technically not legal. There are no guarantees that that the application binary interfaces for both are identical and will therefore work when linked together.

The only problem that was discovered when testing on i386 hardware was related to the `off_t` type. On Linux, `off_t` is 32bit by default, while it is always 64bit on NetBSD. Making the type 64bit on Linux made everything work.

4 Evaluation

To evaluate the usefulness of rump and rump file systems, we discuss them from the perspectives of security, development uses, the maintenance cost of rump, performance, and application use. We also estimate the differences between a rump environment and a real kernel environment and the impact of the differences.

4.1 Security

General purpose OS file systems are commonly written assuming that the file system image to be mounted contains trusted input. While this was true a long time ago, in the age of USB sticks and DVDs it no longer holds. Still, almost all users mount untrusted file systems using kernel code. Even the Linux manual page for mount warns: ”*It is possible for a corrupted file system to cause a crash*”. This is trivial to observe by mounting a suitably corrupted file system.

However, we suspect that the problem is more severe and that by carefully crafting a file system image for the machine to be attacked, a security attack beyond a simple denial-of-service could be mounted.

Using the approach described in this paper, the file system code dealing with the untrusted image is isolated in its own domain, thus mitigating the possibility for attack. As was seen in Table 1, the size difference between a real kernel file system and the kernel portion of a userspace file system is considerable, about five-fold. Since an OS usually supports more than one kernel file system, the code size difference in reality is much higher.

To give an example of a useful scenario, a recent NetBSD mailing list posting described a problem with mounting a FAT file system from a USB stick causing a kernel crash. By using a mountable rump file system, this problem with untrusted input would have been reduced to an application core dump. The problematic image was received from the reporter and problem in the kernel file system code was debugged and dealt with - using rump.

```
golem> rump_msdos ~/img/msdosfs.img /mnt
panic: buf mem pool index 23
Abort (core dumped)
golem>
```

4.2 Development and Debugging

Anyone who has ever done kernel development knows that the kernel is not the most pleasant environment for debugging and iteration. A common approach is to first develop the algorithms in userspace and later integrate them with the kernel environment. However, this adds an extra phase to development.

The following items attempt to capture ways in which the method described in this paper is superior to any single preexisting method.

- **no separate development cycle**: There is no need for a separate userspace development cycle before writing kernel code.
- **same environment**: userspace operating systems and emulators boot the operating system to a separate environment. Migrating for example applications (e.g. OpenOffice or Firefox) and network connections there is challenging. Since rump integrates as a mountable file system on the development host, this problem does not exist.
- **no bit-rot**: There is no maintenance cost for case-specific userspace code because it does not exist.
- **short test cycle**: The code-recompile-test cycle time is very short and a crash results in a core dump and inaccessible files, not a kernel panic and total application failures.
- **userspace tools**: Userspace dynamic analysis tools such as Valgrind [25] can be used to instrument the code. A normal userspace debugger can be used.
- **complete isolation**: Changing interface behavior for e.g. fault and crash injection [18, 26] purposes can be done without worrying about bringing the whole system down.

As an example, support for allocation of an in-fs journal was added to the NetBSD ffs journalling solution recently. The author, Simon Burge, used rump and ukfs for development. He described the process thusly: ”Instead of rebooting with a new kernel to test new code, I was just able to run a simple program, and debug any issues with gdb. It was also a lot safer working on a simple file system image in a file.” [7].

Another benefit is rapid prototyping. One of the reasons for implementing the log-structured file system (LFS) cleaner in userspace on 4.4BSD was the ability to easily try different cleaning algorithms [27]. Using rump file systems it is easy to prototype and test different algorithms without having to increase implementation complexity by requiring an additional component in a different domain. Notably, if the code is written so that it assumes the cleaner in a different domain, the complexity remains in place even when not performing development. With rump file systems the algorithms can be run purely in a single domain once they have been developed and tested to work.

Although it is impossible to measure the ease of development by any formal method, we would like to draw the following analogy about the convenience of development: kernel development on real hardware is to using emulators as using emulators is to developing as a userspace program.

Differences between environments

rump file systems do not duplicate all corner cases accurately with respect to the kernel. For example, Zhang & Ghose [33] list problems related to flushing resources as the implementation issues with using BSD VFS. Theoretically, the flushing behavior can be different if the file system code is running in userspace, and therefore some bugs might be left unnoticed. On the flip-side, the potentially different behavior might expose bugs otherwise very hard to detect when running in the kernel. The

Table 2: Commit analysis for rump source tree

Total commits (HEAD)	336
Unique committers	30
Build fixes	14 (4.1%)
Functionality fixes	5 (1.5%)

framework does not obviously possess exactly the same timing properties and intricacies as the real kernel environment. Our position is that this is not a huge issue.

Differences can also be a benefit. Varying usage patterns can expose bugs where they were hidden before. For example, the recent NetBSD problem report kern/38057 described a FFS bug which occurs when the file system device node is not on FFS itself, e.g. /dev on tmpfs. Commonly, /dev is on FFS, so regular use did not trigger the problem. However, since this does not hold when using FFS through rump, the problem is triggered more easily.

4.3 Maintaining rump in NetBSD

As rump implements environment dependent code in parallel with the rest of the kernel, the implementation needs to keep up with the changing kernel interfaces. There are two kinds of possible breakage: the kind resulting in compilation failure and the kind resulting in non-functional compiled code. The statistics in Table 2 have been collected from version control logs from the period August 2007 - September 2008, during which rump has been part of the official NetBSD source tree. The number of commits represents the number of changes to the main branch. Counting all branches, the number is over double.

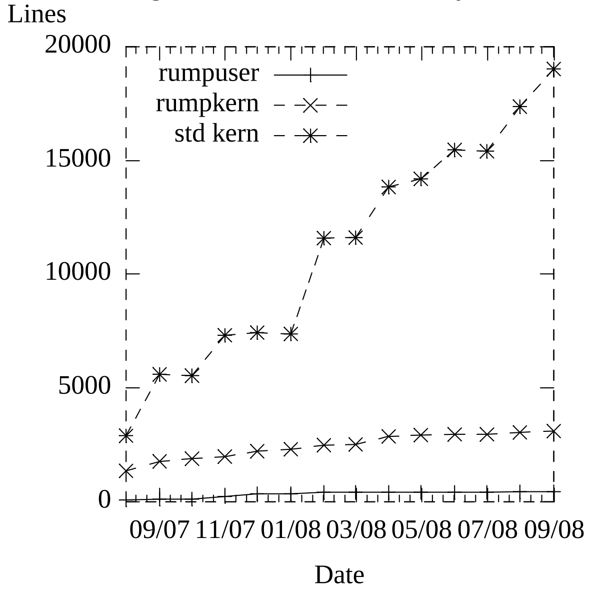
The number of build fixes is calculated from the amount of commits that were done after the kernel was changed and rump not build anymore as a result. For example, a file system being changed to require a kernel interface not yet supported by rump is this kind of failure. Commits, where the rump tree was patched along with the kernel proper were not counted in with this figure.

Similarly, functionality fixes include changes to kernel interfaces which prevented rump from working, in other words the build worked but running the code failed. Regular bugs were not counted in with this, only changes related to other kernel changes.

Unique committers represents the number of people from the NetBSD community who committed changes to the rump tree. The most common case was to keep up with changes in other parts of the kernel.

Based on our observations, the most important factor in keeping a system such as rump functional in a changing kernel is educating developers about its existence and how to test it. Initially there was a lot of confusion about

Figure 6: Lines of Code History



how to test build rump, but things have since gotten better and breakage has become less frequent.

While examining Table 2, it should be kept in mind that over the same period of time the NetBSD kernel underwent very heavy restructuring to better support multiprocessor architectures. As it was the heaviest set of changes over the past 15 years, the data should be considered ”worst case” instead of ”typical case”.

For an idea of how much code there is to maintain, Figure 6 analyses the number of lines of code monthly for the period the rump framework has been integrated into NetBSD. The count is again without empty lines or comments.

Over a year, the number of lines of environment dependent code (rumpkern + rumpuser) has gone up from 1443 to 3531 (245%) while the number of code lines used directly from the kernel has gone up from 2894 to 19017 (657%). Features have been added, but much of this has been done with environment independent code. Not only does this reduce code duplication, but it makes rump file systems behave closer to kernel file systems on a detailed level.

4.4 Performance

The goal of this work is not to advance state-of-the-art in microkernel performance - we can still use the file system as part of a monolithic kernel if performance is the paramount criterion. The purpose of these measurements is to show that performance is at a level which can be considered usable.

To measure the performance of rump, we chose to modify the Postmark [20] benchmark to use the ukfs interface. This approach was chosen instead of bench-

Figure 7: Postmark, all operations

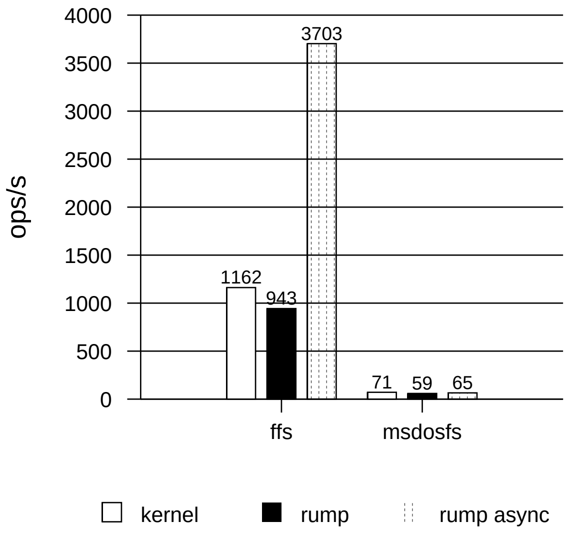


Figure 8: Kernel File System

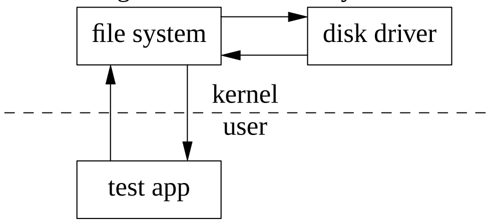
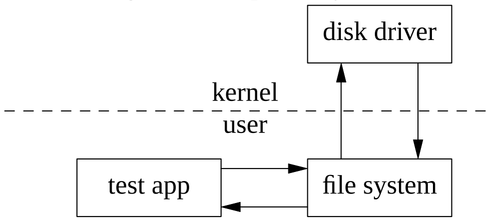


Figure 9: rump File System



marking the transparently mounted case to avoid measuring the performance of puffs instead of the performance of rump.

The conversion of Postmark was done in a straightforward fashion simply by replacing calls to system calls with the relevant ukfs call, e.g. `rmdir`() was replaced with `ukfs_rmdir`(). However, since the ukfs interface is more about convenience than performance, it was optimized slightly. For instance, since `ukfs_write`() internally opens a file, writes to it and closes it, calling it multiple times in a row is inefficient. Rather, the rump system calls to open a file, write to it multiple times and finally close it were used.

To establish a reference, the regular unmodified Postmark was run with the same parameters on the same file system using a mounted file system running in the kernel. This run was performed with the stdio buffering option of Postmark turned off.

Two file systems were tested. The FFS tests were done on a hard drive partition, while msdosfs (FAT) was run on a USB stick flash backend. Async rump means that forced flushes were disabled and is explained further below. The results are presented in Figure 7.

To understand the results, the conceptual difference between the normal case and the rump case is illustrated in Figures 8 and 9. An in-kernel file system has much more control over how it flushes its buffers. In userspace on NetBSD we only have three device operations available: read, write and full cache flush (fsync). Basic FFS [23] depends heavily on synchronous metadata writes for correct operation. The kernel can flush only the buffers it desires, while if data writes are intermingled with metadata writes, every metadata flush from userspace demands the flush of all outstanding buffers. The slowdown of flushes is evident from the speed difference to async mode, where no forced flushes happen during operation. The FAT implementation does not use synchronous writes as heavily, and therefore the speed difference is much more modest.

Also, a single file system operation requires numerous disk operations. For example, extending a file must read the file inode, search for free space, allocate the free space, update the inode and write the data.

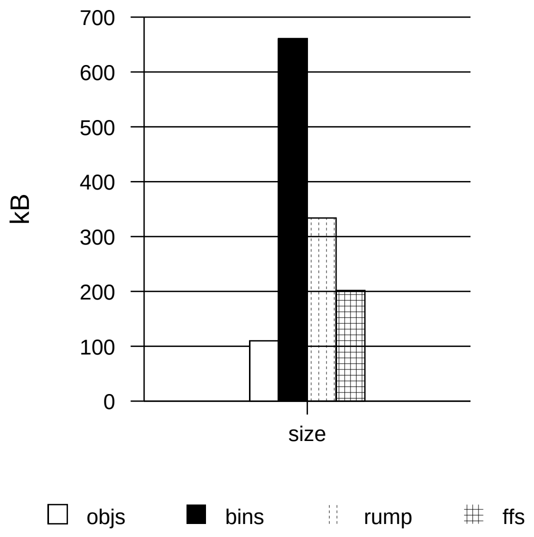
Part of the performance gap could be addressed by adding an additional interface to userspace to better be able to control the syncing of writes. Also, currently the rump disk driver flushes all writes one-by-one. Pooling them is more efficient. However, as we see performance acceptable for our purposes, this is not a high priority task.

4.5 Use in Applications

System utilities. Many userspace applications dealing with file systems reimplement functionality necessary for understanding the file system. For instance, the *fsck* program needs to understand the on-disk layout in order to be able to repair a damaged file system. The kernel code naturally understands the on-disk layout and also provides routines for accessing it. But since kernel code was historically not available in userspace, this functionality was in part reimplemented in a form suitable for userspace. The result is of course functionality duplication and added maintenance load.

The list of userspace programs using code from the ffs codebase and therefore requiring an application-specific

Figure 10: Binary code size comparison



reimplementation of some functionality is: *badsect*, *clri*, *dump*, *dumpfs*, *fsck_ffs*, *fsdb*, *fsirand*, *newfs*, *quotacheck* and *tunefs*. Additionally, the *makefs* [24] utility, which creates a file system image from a given directory tree, uses hand modified kernel file system code. Reimplementation using ukfs would make the utility work using standard, unmodified kernel code.

The only downside for using rump instead of code hand adjusted for userspace is the size difference. A comparison for the ffs file system is presented in Figure 10.

The figure ”objs” represents the combined size of ffs binary objects from the compilation directories above mentioned utilities, while ”bins” is the combined resulting binary sizes. The true amount of binary code size for userspace reimplementation is somewhere between these two. The relevant rump libraries (rumpkern, rumpuser, ukfs) are represented by ”rump” and the ffs file system size is represented by ”ffs”.

While the size increase of a few hundred kilobytes seems unnoteworthy these days, for some embedded system manufacturers it is unacceptable. One idea is to try to use binary code from the file system kernel module instead of requiring a separate binary library. Unfortunately, this will not benefit scenarios where the boot kernel is stored in a location inaccessible at runtime.

File system applications. The best-known application suite for dealing with file system images is probably mtools [4]. It enables to access and modify FAT file systems purely from a userspace application without any kernel support. Additionally, since FAT was commonly used on floppy disks, which usually were less than reliable, a kernel mount could easily hang or crash the system.

However, mtools implements support only for FAT. As the rump suite is file system independent, a set of utilities which work on all supported kernel file systems can be done. The fs-utils [32] project provides a plethora of utilities which behave like standard Unix utilities, but access file system images by means of ukfs. Examples of utilities provided by fs-utils are `fsu_ls`, which lists the contents of an image and `fsu_touch`, which modifies a file's timestamp while creating it if it does not already exist.

5 Related Work

The Alpine [13] network protocol development infrastructure provides an environment for running kernel code in userspace. It is implemented before the system call layer by overriding certain libc symbols and is run in the same process context as an application. This approach both makes it unsuitable for statically linked programs and creates difficulties with shared global resources such as the `read`()/`write`() calls used for I/O beyond networking. Furthermore, from a file system perspective, this kind of approach shuts out kernel-initiated file system access, e.g. NFS servers.

Rialto [12] is an operating system with a unified interface both for userspace and the kernel making it possible to run most code in either environment. However, this system was designed from ground-up that way. Interesting ideas include the definition of both internal and external linkage for an interface. While the ideas are inspiring, we do not have the luxury to redo everything.

Mach is capable of running Unix as a user process [15]. Lites [16] is a Mach server which can run a wide variety of binaries from different Unix flavors at the same time. It is is based on the 4.4BSD Lite code base. Debugging and developing 4.4BSD file systems under Mach/Lites is possible by using two Lites servers: one for running the debugger and one running the file system being developed, including applications using the file system. If the Lites server being debugged crashes, applications inside it will be terminated. Being a single server solution, it does not provide isolation from the trusted computing base, either. A multiserver microkernel such as SawMill [14] addresses the drawbacks of a single server, but does not permit a monolithic mode or use of the file system code as an application library.

Operating systems running in userspace, such as User Mode Linux [11], make it possible to run the entire operating system as a userspace process. The main aims in this are providing better debugging & development support and isolation between instances. However, for development purposes, this approach does not provide isolation between the component under development and the core of the operating system - rather, they both run in the same process. This results in complexity in, for example, using fault injection and dynamic analysis tools. Neither does a userspace operating system integrate into the host, i.e. it is not possible to mount the userspace operating system as a file server. Even if that could be addressed, booting an entire kernel every time a ukfs application is run is a very heavyweight solution.

Sun's ZFS [6] file system ships with a userspace testing library, libzpool. In addition to some kernel interface emulation routines, it consists of the Data Management Unit and Storage Pool Allocator components of ZFS compiled from the kernel sources. The ztest userspace program plugs directly into these interfaces for running tests. This approach has several shortcomings when compared to rumps. First, it does not test the entire file system code architecture, for example the VFS interface layer. The effort for getting the VFS interface (in ZFS terms known as the *ZFS POSIX Layer* or *ZPL*) right was specifically listed as the hardest part in porting ZFS to FreeBSD [10]. Second, it does not facilitate userspace testing with real applications because it cannot be mounted. And third, the test program is specific to ZFS.

Many projects reimplement file system code for userspace purposes. Examples include the already earlier mentioned mtools [4] and e2fsprogs [1]. Their functionality provided overlaps that which is readily already provided by the kernel. Especially e2fsprogs must track Linux kernel features and perform an independent reimplementation.

fuse-ext2 [3] is a userspace file server built on top of e2fsprogs. It implements an a translator from FUSE [2] to e2fsprogs. The functionality provided by fuse-ext2 is the same as that if rump_ext2fs, but requires specifically written code and maintenance. Similarly, the ChunkFS [17] prototype is fully mountable, but it is implemented on top of FUSE [2] and userspace interfaces instead of the kernel file system interface.

Simulators [9, 30] can be used to run traces against file systems. Thekkath [30] et al go as far as to run the HPUX FFS implementation in userspace. However, these tools execute against a recorded trace and do not permit mounting.

6 Conclusions and Future Work

In this paper we defined and described a Runnable Userspace Meta Program (rump) framework for using preexisting kernel file system code in userspace. There are two different use modes for the framework: the puffs-to-vfs mode in which file systems are mounted so that they can be accessed like any other mounted file system, and a standalone mode in which applications can use file system routines through the ukfs library interface. The first mode brings a multiserver microkernel touch to a monolithic kernel Unix OS, but gives the user the option of monolithic operation for scenarios where absolutely maximal performance counts. The second mode enables reuse of the available kernel code in applications such as those involved in file system repair and image access.

The NetBSD implementation was evaluated. We discovered that the system has security benefits especially for file systems on untrusted removable media. It made debugging and developing kernel file system code easier and more convenient, but did not require additional case-specific ”glue code” for making kernel code runnable in userspace. The issues regarding the maintenance of rump were examined by looking at a year's worth of version control system commits. The build had broken 14 times and functionality 5 times. These were attributed to the lack of a proper regression testing facility and developer awareness. The uses for kernel file system code in applications were discussed.

The performance of rump file systems was measured against a standard kernel mount and observed to be dominated by I/O speed, e.g. ffs was 19% slower when using synchronous metadata writes and 319% faster when using asynchronous metadata. We found this acceptable and did not concentrate on optimizing performance, since, as was already mentioned, the same code can be run inside the kernel in case performance is key.

Future work will concentrate on making rump file system libraries and file system kernel modules binary-equivalent. This will completely defer the choice of kernel or userspace usage until runtime without requiring two sets of almost equivalent binaries. Further integration with system utilities such as fsck and makefs will also be attempted.

As a concluding remark, the technology has shown real world use and having kernel file systems from major open source operating systems available as portable userspace components would vastly increase system cross-pollination and reduce the need for reimplementations. We encourage kernel programmers to not only to think about code from the classical machine dependent/machine independent viewpoint, but also from the environment dependent/environment independent perspective to promote code reuse.

Acknowledgments

This work was supported by green tea, dark chocolate and black coffee.

References

[1] E2fsprogs: Ext2/3/4 Filesystem Utilities. http://e2fsprogs.sourceforge.net/.

[2] FUSE - Filesystem in Userspace. http://fuse.sourceforge.net/.

[3] fuse-ext2. http://sourceforge.net/projects/fuse-ext2/.

[4] Mtools. http://mtools.linux.lu/.

[5] The NetBSD Project. http://www.NetBSD.org/.

[6] ZFS source tour. http://www.opensolaris.org/os/community/zfs/source/.

[7] An interview about NetBSD WAPBL. *BSD Magazine*, 1(3), December 2008. unpublished draft.

[8] D. Almeida. FIFS: a framework for implementing user-mode file systems in windows NT. In *WINSYM'99: Proc. of USENIX Windows NT Symposium*, 1999.

[9] P. Bosch and S. J. Mullender. Cut-and-paste file-systems: Integrating simulators and file-systems. In *USENIX Annual Technical Conference*, pages 307–318, 1996.

[10] P. J. Dawidek. Porting the ZFS file system to the FreeBSD operating system. In *Proc. of AsiaBSDCon 2007*, pages 97–103, 2007.

[11] J. Dike. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th conference on 4th Annual Linux Showcase & Conference*, 2000.

[12] R. Draves and S. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft Research, 1997.

[13] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USENIX Symp. on Internet Technologies and Systems*, pages 171–184, 2001.

[14] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 109–114, 2000.

[15] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *Proc. of USENIX Summer*, pages 87–95, 1990.

[16] J. Helander. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994.

[17] V. Henson, A. van de Ven, A. Gud, and Z. Brown. ChunkFS: using divide-and-conquer to improve file system reliability and repair. In *HOTDEP'06: Proc. of Hot Topics in System Dependability*, 2006.

[18] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[19] A. Kantee. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon 2007*, pages 29–42, 2007.

[20] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, 1997.

[21] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun UNIX. In *Proc. of USENIX Summer*, pages 238–247, 1986.

[22] J. Mauro and R. McDougall. *Solaris internals: Core Kernel Architecture*. 2001.

[23] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[24] L. Mewburn and M. Green. build.sh: Cross-building NetBSD. In *Proc. of BSDCon*, pages 47–56, 2003.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI*, pages 89–100, 2007.

[26] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, 2005.

[27] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX Winter*, pages 307–326, 1993.

[28] C. Silvers. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *Proc. of USENIX, FREENIX Track*, pages 285–290, 2000.

[29] P. Snyder. tmpfs: A virtual memory file system. In *Proc. EUUG Conference*, pages 241–248, 1990.

[30] C. A. Thekkath, J. Wilkes, and E. D. Lazowska. Techniques for file system simulation. *Software - Practice and Experience*, 24(11):981–999, 1994.

[31] D. Woodhouse. Jffs2 the journalling flash file system. In *Ottawa Linux Symposium*, 2001.

[32] A. Ysmal. FS Utils. http://netbsd-soc.sourceforge.net/fs-utils/.

[33] Z. Zhang and K. Ghose. hFS: a hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys*, pages 175–187, 2007.