Performance and Extension of User Space File

Aditya Raigarhia and Ashish Gehani
Stanford and SRI

ACM Symposium on Applied Computing (SAC)
Sierre, Switzerland, March 22-26, 2010

Introduction (1 of 2)

- Developing in-kernel file systems challenging
  - Understand and deal with kernel code and data structures
  - Steep learning curve for kernel development
    - No memory protection
    - No use of debuggers
    - Must be in C
    - No standard C library
- In-kernel implementations not so great
  - Porting to other flavors of Unix can be difficult
  - Needs root to mount – tough to use/test on servers

Introduction (2 of 2)

- Modern file system research adds functionality over basic systems, rather than designing low-level systems
  - Ceph [37] – distributed file system for performance and reliability – uses client in users space
- Programming in user space advantages
  - Wide range of languages
  - Use of 3rd party tools/libraries
  - Fewer kernel quirks (although still need to couple user code to kernel system calls)

Introduction - FUSE

- File system in USEr space (FUSE) – framework for Unix-like OSes
- Allows non-root users to develop file systems in user space
- API for interface with kernel, using fs-type operations
- Many different programming language bindings
- FUSE file systems can be mounted by non-root users
- Can compile without re-compiling kernel
- Examples
  - WikipediaFS [2] lets users view/edit Wikipedia articles as if local files
  - SSHFS access via SFTP protocol

Problem Statement

- Prevailing view – user space file systems suffer significantly lower performance compared to kernel
  - Overhead from context switch, memory copies
- Perhaps changed due to processor, memory and bus speeds?
- Regular enhancements also contribute to performance?
- Either way – measurement of "prevailing view"

Outline

- Introduction (done)
- Background (next)
- FUSE overview
- Programming for FS
- Benchmarking
- Results
- Conclusion

Background – Operating Systems

- Microkernel (Mach [10], Spring [11]) have only basic services in kernel
  - File systems (and other services) in user space
  - But performance is an issue, not widely deployed
- Extensible OSes (Spin[1], Vino[4]) export OS interfaces
  - User level code can modify run-time behavior
  - Still in research phase

Background – Stackable FS

- Stackable file systems [28] allow new features to be added incrementally
  - FiST [40] allows file systems to be described using high-level language
  - Code generation makes kernel modules – no recompilation required
- But
  - cannot do low-level operations (e.g., block layout on disk, metatdata for i-nodes)
  - Still require root to load

Background – NFS Loopback

- NFS loopback servers [24] puts server in user-space with client
  - Provides portability
  - Good performance
- But
  - Limited to NFS weak cache consistency
  - Uses OS network stack, which can limit performance

Background - Misc

- Coda [29] is distributes file system
  - Venus cache manager in user space
  - Arla [38] has AFS user-space daemon
  - But not widespread
- `ptrace()` – process trace
  - Working infrastructure for user-level FS
  - Can interacept anything
  - But significant overhead
- puffs [15] similar to FUSE but NetBSD
  - FUSE built on puffs for some systems
  - But puffs not as widespreadh

Background – FUSE contrast

- FUSE similar since loadable kernel module
- Unlike others is mainstream – part of Linux since 2.6.14, ports to Mac OSX, OpenSolaris, FreeBSD and NetBSD
  - Reduces risk of obsolete once developed
- Licensing flexible – free and commercial
- Widely used (examples next)

Background – FUSE in Use

- TierStore [6] distributed file system to simply deployment of apps in unreliable networks
  - Uses FUSE
- Increasing trend for dual OS (Win/Linux)
  - NTFS-3G [25] open source NTFS uses FUSE
  - ZFS-FUSE [41] is port of Zeta FS to Linux
  - VMWare disk mount [36] uses FUSE on Linux

FUSE Example – SSHFS on Linux

https://help.ubuntu.com/community/SSHFS

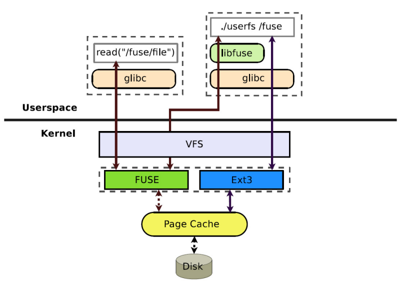
```
% mkdir ccc
% sshfs -o idmap=user claypool@ccc.wpi.edu:/home/claypool ccc
% fusermount -u ccc
```

Outline

- Introduction (done)
- Background (done)
- FUSE overview (next)
- Programming for FS
- Benchmarking
- Results
- Conclusion

FUSE Overview

- On userfs mount, FUSE kernel module registers with VFS
  - e.g., call to "sshfs"
- userfs provides callback functions
- All file system calls (e.g., `read()`) proceed normally from other process
- When targeted at FUSE dir, go through FUSE module
- If in page cache, return
- Otherwise, to userfs via `/dev/fuse` and libfuse
- userfs can do anything (e.g., request data from ext3 and add stuff) before returning data



- fusermount allows non-root users to mount

FUSE APIs for User FS

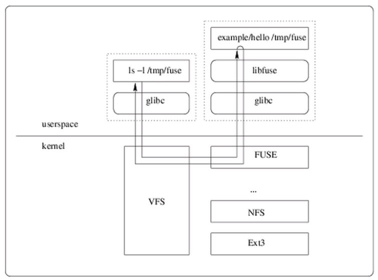
- Low-level
  - Resembles VFS – user fs handles i-nodes, pathname translations, fill buffer, etc.
  - Useful for "from scratch" file systems (e.g., ZFS-FUSE)
- High-level
  - Resembles system calls
  - User fs only deals with pathnames, not i-nodes
  - libfuse does i-node to path translation, fill buffer
  - Useful when adding additional functionality

FUSE – Hello World Example

Run

```
~/fuse/example$ mkdir /tmp/fuse
~/fuse/example$ ./hello /tmp/fuse
~/fuse/example$ ls -l /tmp/fuse
total 0
-r--r--r-- 1 root root 13 Jan 1 1970 hello
~/fuse/example$ cat /tmp/fuse/hello
Hello World!
~/fuse/example$ fusermount -u /tmp/fuse
~/fuse/example$
```

Flow



http://fuse.sourceforge.net/helloworld.html

FUSE – Hello World (1 of 4)

```c
#define FUSE_USE_VERSION 26

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

static const char *hello_str = "Hello World!\n";
static const char *hello_path = "/hello";
-----------------------------------------------------------

static struct fuse_operations hello_oper = {
        .getattr        = hello_getattr,
        .readdir        = hello_readdir,
        .open           = hello_open,
        .read           = hello_read,
};

int main(int argc, char *argv[])
{
        return fuse_main(argc, argv, &hello_oper, NULL);
}
```

Callback operations

Invoking does 'mount'

http://fuse.sourceforge.net/helloworld.html

FUSE – Hello World (2 of 4)

```c
static int hello_getattr(const char *path, struct stat *stbuf)
{
        int res = 0;

        memset(stbuf, 0, sizeof(struct stat));
        if (strcmp(path, "/") == 0) {
                stbuf->st_mode = S_IFDIR | 0755;
                stbuf->st_nlink = 2;
        } else if (strcmp(path, hello_path) == 0) {
                stbuf->st_mode = S_IFREG | 0444;
                stbuf->st_nlink = 1;
                stbuf->st_size = strlen(hello_str);
        } else
                res = -ENOENT;

        return res;
}
```

Fill in file status structure (type, permissions)

http://fuse.sourceforge.net/helloworld.html

```c
static int hello_open(const char *path, struct fuse_file_info *fi)
{
        if (strcmp(path, hello_path) != 0)
                return -ENOENT;

        if ((fi->flags & 3) != O_RDONLY)
                return -EACCES;

        return 0;
}

static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                      struct fuse_file_info *fi)
{
        size_t len;
        (void) fi;
        if(strcmp(path, hello_path) != 0)
                return -ENOENT;

        len = strlen(hello_str);
        if (offset < len) {
                if (offset + size > len)
                        size = len - offset;
                memcpy(buf, hello_str + offset, size);
        } else
                size = 0;

        return size;
}
```

Check that path is right
Check permissions right (read only)

Check that path is right
Copy data to buffer

http://fuse.sourceforge.net/helloworld.html

FUSE – Hello World (4 of 4)

```
static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                         off_t offset, struct fuse_file_info *fi)
{
        (void) offset;
        (void) fi;

        if (strcmp(path, "/") != 0)
                return -ENOENT;

        filler(buf, ".", NULL, 0);
        filler(buf, "..", NULL, 0);
        filler(buf, hello_path + 1, NULL, 0);

        return 0;
}
```

Copy in directory listings

http://fuse.sourceforge.net/helloworld.html

Performance Overhead of FUSE : Switching

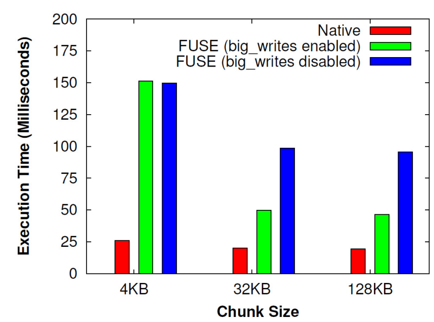
- When using native (e.g., ext3)
  - Two user-kernel mode switches (to and from)
    - Relatively fast since only privilege/unpriviledge
  - No context switches between processes/address space
- When using FUSE
  - Four user-kernel mode switches (adds up to userfs and back)
  - Two context switches (user process and userfs)
    - Cost depends upon cores, registers, page table, pipeline

Performance Overhead of FUSE : Reading

- FUSE used to have 4 KB read size
  - If memory constrained, large reads would do many context switch each read
    - swap out userfs, bring in page, swap in userfs, continue request, swap out userfs, bring in next page …
- FUSE now reads in 128 KB chunks with `big_writes` mount option
  - Most Unix utilities (`cp`, `cat`, `tar`) use 32 KB file buffers

Performance Overhead of FUSE : Time for Writing

(Write 16 MB file)



Note, benefit from 4KB to 32KB, but not 32KB to 128KB

Performance Overhead of FUSE : Memory Copying

- For native (e.g., ext3), write copies from application to kernel page cache (1x)
- For user fs, write copies from application to page cache, then from page cache to libfuse, then libfuse to userfs (3x)
- `direct_io` mount option – bypass page cache, user copy directly to userfs (1x)
  - But reads can never come from kernel page cache!

Performance Overhead of FUSE : Memory Cache

- For native (e.g., ext3), read/written data in page cache
- For user fs, libfuse and userfs both have data in page cache, too (extra copies) – useful since make overall more efficient, but reduce size of usable cache

Outline

- Introduction (done)
- Background (done)
- FUSE overview (done)
- Programming for FS (next)
- Benchmarking
- Results
- Conclusion

Language Bindings

- 20 language bindings – can build userfs in many languages
  - C++ or C# for high-perf, OO
  - Haskell and OCaml for higher order functions (functional languages)
  - Erlang for fault tolerant, real-time, distributed (parallel programming)
  - Python for rapid development (many libraries)
- *JavaFuse* [27] built by authors

Java Fuse

- Provides Java interface using Java Native Interface (JNI) to communicate from Java to C
- Developer writes file system as Java class
- Register with JavaFuse using command line parameter
- JavaFuse gets callback, sends to Java class
- Note, C to Java may mean more copies
  - Could have "file" meta-data only option
  - Could use JNI non-blocking I/O package to avoid
  - →But both limit portability and are not thread safe

Outline

- Introduction (done)
- Background (done)
- FUSE overview (done)
- Programming for FS (done)
- Benchmarking (next)
- Results
- Conclusion

Benchmarking Methodology (1 of 2)

- Microbenchmarks – raw throughput of low-level operations (e.g., `read()`)
- Use Bonnie [3], basic OS benchmark tool
- 6 phases:
  1. write file with `putc()`, one char at a time
  2. write from scratch same file, with 16 KB blocks
  3. read file with `getc()`, one char at a time
  4. read file, with 16 KB blocks
  5. clear cache, repeat `getc()`, one char at a time
  6. clear cache, repeat read with 16 KB blocks

Benchmarking Methodology (2 of 2)

- Macrobenchmarks – application performance for common apps
- Use Postmark – small file workloads by email, news, Web-based commerce under heavy load
  - Heavy stress of file and meta-data
  - Generate initial pool of random text files (used 5000, from 1 KB to 64 KB)
  - Do transactions on files (used 50k transactions, typical Web server workload [39])
- Large file copy – copy 1.1 GB movie using `cp`
  - Single, large file as for typical desktop computer or server

Testbed Configurations

- Native – ext4 file system
- FUSE – null FUSE file system in C (passes each call to native file system)
- JavaFuse1 (metadata-only) – null file system in JavaFuse, does not copy `read()` and `write()` over JNI
- JavaFuse2 (copy all data) – copies all over JNI

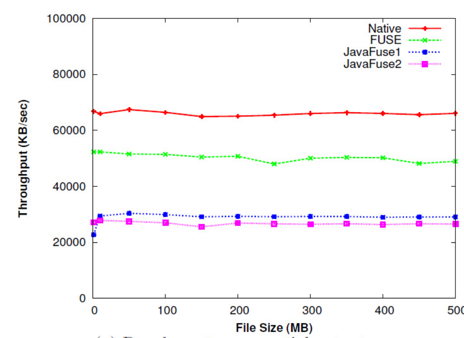
Experimental Setup

- P4, 3.4 GHz, 512 MB RAM (increased to 2 GB for macrobenchmarks), 320 GB Seagate HD
- Maximum sustained throughput on disk is 115 MB/s
  - So, any reported throughputs above 115 MB/s must benefit from cache
- Linux 2.6.30.5, FUSE 2.8.0-pre1

Outline

- Introduction (done)
- Background (done)
- FUSE overview (done)
- Programming for FS (done)
- Benchmarking (done)
- Results (next)
- Conclusion

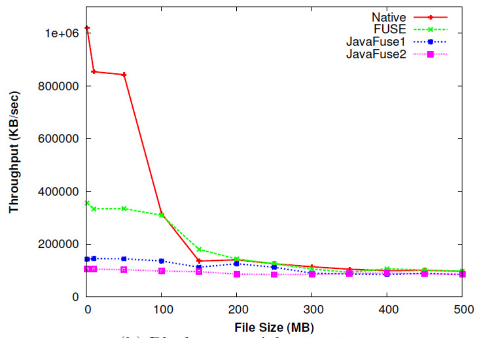
Microbenchmark Results



(a) Per-character sequential output.

- FUSE ~25% overhead
  - Context switches
- JavaFuse more overhead (context switches between JNI and C), and doing copies not much worse than metadata

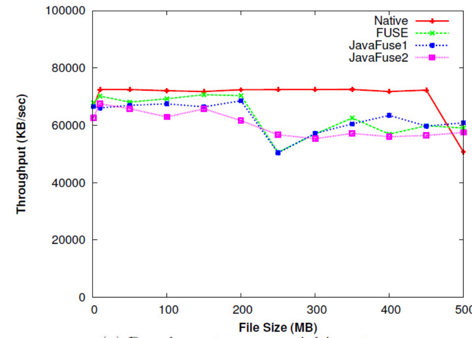
Microbenchmark Results



(b) Block sequential output.

- Native much faster than FUSE for small files (written to memory)
- For large cache, written to disk which dominates

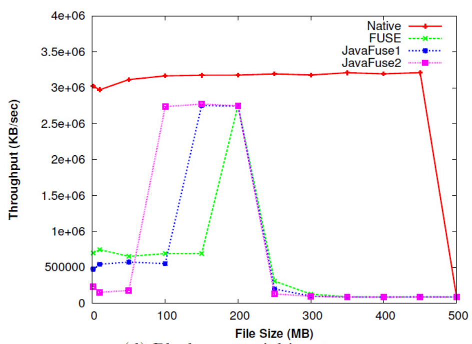
Microbenchmark Results



(c) Per-character sequential input.

- Data already in page cache, so fast for all
- Spikes are when 512 MB RAM exhausted (FUSE has two copies, so earlier around 225)

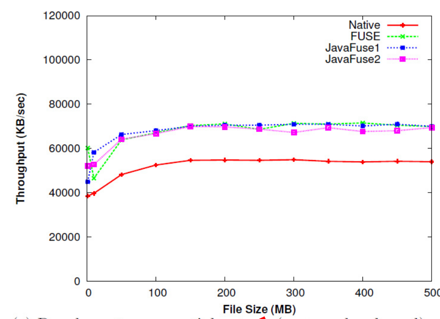
Microbenchmark Results



(d) Block sequential input.

- For native, can cached up to 500 MB
- For Java, spike is caused by artificial nature of benchmark (previous version still in cache)

Microbenchmark Results

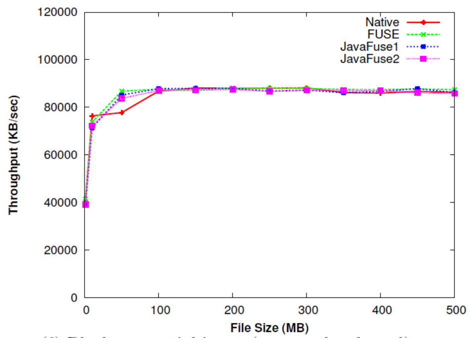


(e) Per-character sequential ~~output~~ (page cache cleared).

input

- When cache cleared, starts lower gets higher for larger
  - Kernel optimizes for sequential read
- FUSE does better – think it's because fusefs and ext4 both read-ahead

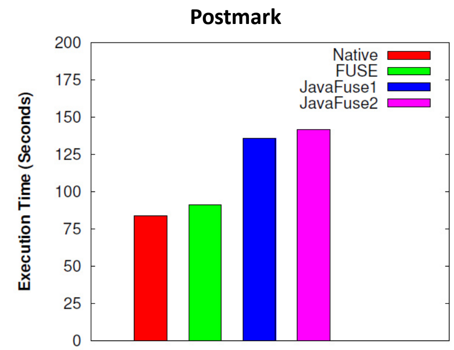
Microbenchmark Results



(f) Block sequential input (page cache cleared).

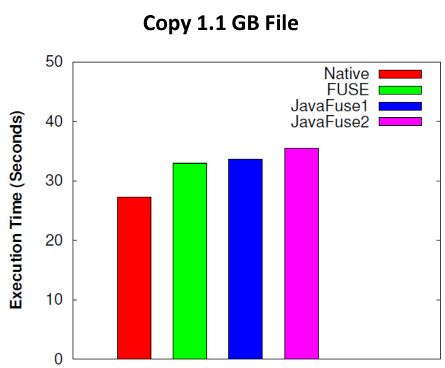
- Native gets same benefit as FUSE, so no apparent difference

Macrobenchmark Results



- FUSE overhead less than 10%
- Java overhead about 60%
  - Greater CPU and memory use

Macrobenchmark Results



- FUSE comparable to native (~30%)
- Java overhead minimal over FUSE

Conclusion

- FUSE may be feasible depending upon workload
- Performance comparable to in-kernel for large, sustained I/O
- Overhead noticeable for large number of meta data (e.g., Web and many clients)
- Adequate for PCs and small servers for I/O transfer
- Additional language (Java) incurred additional overhead
  - But overhead can be reduced with optimizations (e.g., shared buffers)