Project 2

Storage Management Part 2

in a

Hybrid SSD/HDD File system

Part 1

Project due on April 11th (11.59 EST)

- Start early ☺
  - Milestone1: finish part 1 by March 25th
  - Milestone 2: finish part 2 by April 6th
  - Last you a few days to work on performance analysis and report writing

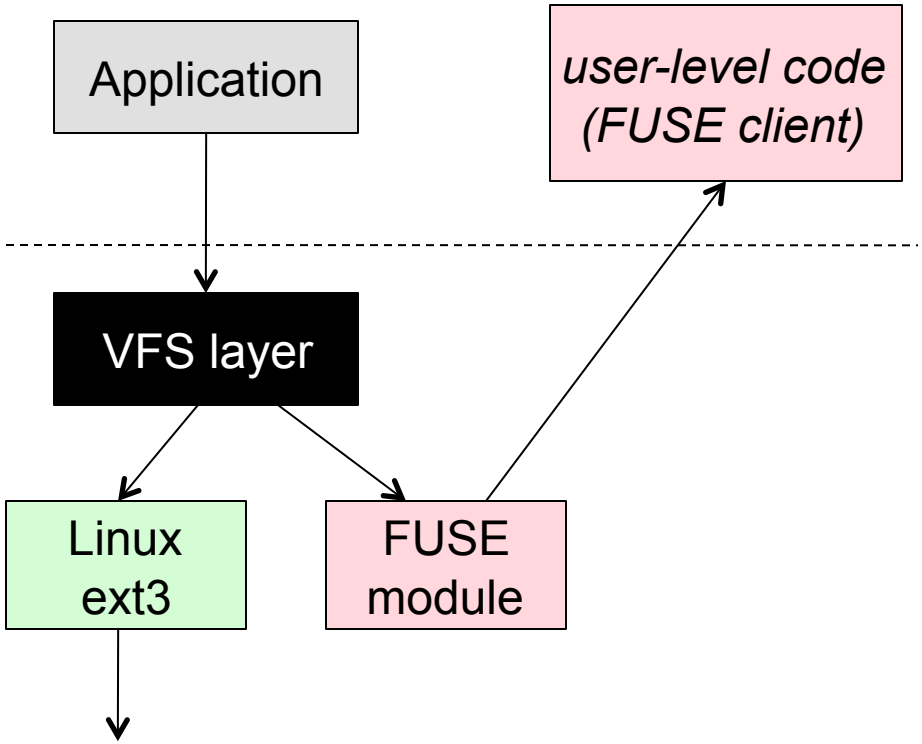
- "Oh well, I will write the report on April 11th"
  - Bad idea ☺
  - 30% of the project grade is divided among project report and source code correctness/quality
  - Project report consists of design overview and performance analysis of your results

Summary of the project

- Write a user-level file system called MelangeFS and two simple storage management applications
  - File system in user-space (FUSE) toolkit
  - Code to be written in C
- Testing and evaluation setup
  - All development and testing done in Linux using the VirtualBox virtual machine setup
  - Your code must compile in the Linux images on the virtual machine
  - Demo after the slides!

Part 0: Primer on the FUSE toolkit

- Interposition layer to redirect VFS calls to your user-level code
- FUSE client code is programmable
  - Talk to remote nodes
  - Interact with local FSs
- FUSE clients need to implement a minimal set of protocols



FUSE API

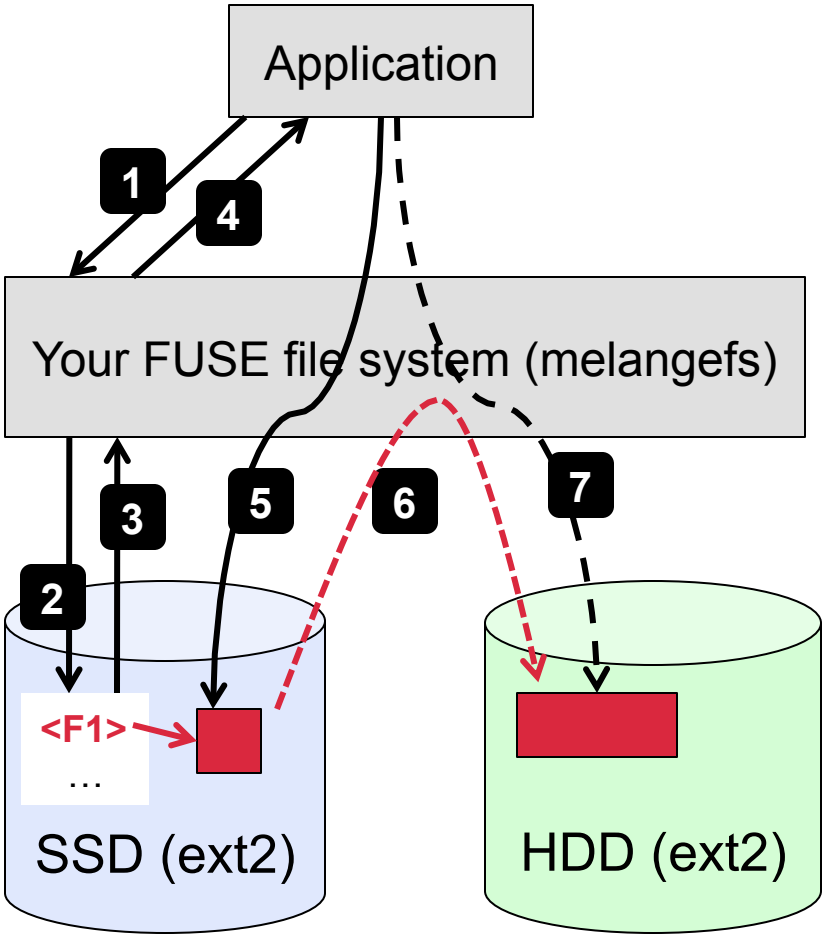
- Supports most VFS calls
  - This API is the “high-level” interface using path names
- You don’t need to implement all the calls
  - Simple systems, such as MelangeFS, can getaway with basic calls
  - More in the demo!

```
int(*  getattr )(const char *, struct stat *)
int(*  readlink )(const char *, char *, size_t)
int(*  mknod )(const char *, mode_t, dev_t)
int(*  mkdir )(const char *, mode_t)
int(*  unlink )(const char *)
int(*  rmdir )(const char *)
int(*  symlink )(const char *, const char *)
int(*  rename )(const char *, const char *)
int(*  link )(const char *, const char *)
int(*  chmod )(const char *, mode_t)
int(*  chown )(const char *, uid_t, gid_t)
int(*  truncate )(const char *, off_t)
int(*  utime )(const char *, struct utimbuf *)
int(*  open )(const char *, struct fuse_file_info *)
int(*  read )(const char *, char *, size_t, off_t, struct fuse_file_info *)
int(*  write )(const char *, const char *, size_t, off_t, struct fuse_file_info *)
int(*  statfs )(const char *, struct statvfs *)
int(*  flush )(const char *, struct fuse_file_info *)
int(*  release )(const char *, struct fuse_file_info *)
int(*  fsync )(const char *, int, struct fuse_file_info *)
int(*  setxattr )(const char *, const char *, const char *, size_t, int)
int(*  getxattr )(const char *, const char *, char *, size_t)
int(*  listxattr )(const char *, char *, size_t)
int(*  removexattr )(const char *, const char *)
int(*  opendir )(const char *, struct fuse_file_info *)
int(*  readdir )(const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *)
int(*  releasedir )(const char *, struct fuse_file_info *)
int(*  fsyncdir )(const char *, int, struct fuse_file_info *)
void *(*  init )(struct fuse_conn_info *conn)
void(*  destroy )(void *)
int(*  access )(const char *, int)
int(*  create )(const char *, mode_t, struct fuse_file_info *)
int(*  ftruncate )(const char *, off_t, struct fuse_file_info *)
int(*  fgetattr )(const char *, struct stat *, struct fuse_file_info *)
int(*  lock )(const char *, struct fuse_file_info *, int cmd, struct flock *)
int(*  utimens )(const char *, const struct timespec tv[2])
int(*  bmap )(const char *, size_t blocksize, uint64_t *idx)
unsigned int  flag_nullpath_ok: 1
unsigned int  flag_reserved: 31
int(*  ioctl )(const char *, int cmd, void *arg, struct fuse_file_info *, unsigned int flags, void
*data)
int(*  poll )(const char *, struct fuse_file_info *, struct fuse_pollhandle *ph, unsigned
*reventsp)
```

Part 1: Hybrid Flash-Disk Systems

- Hybrid storage systems: Best of both worlds
  - Capacity cost closer to magnetic disk
  - Random access speed closer to flash
- MelangeFS is a layered file system
  - Higher-level file system (MelangeFS) splits data between the two devices
  - Each device is running it's own local file system (Ext2) which you will *not* modify
- Key idea
  - All small objects in flash, all large objects in magnetic disk
  - All small IO (metadata access) should go to the flash

Size-based data-placement

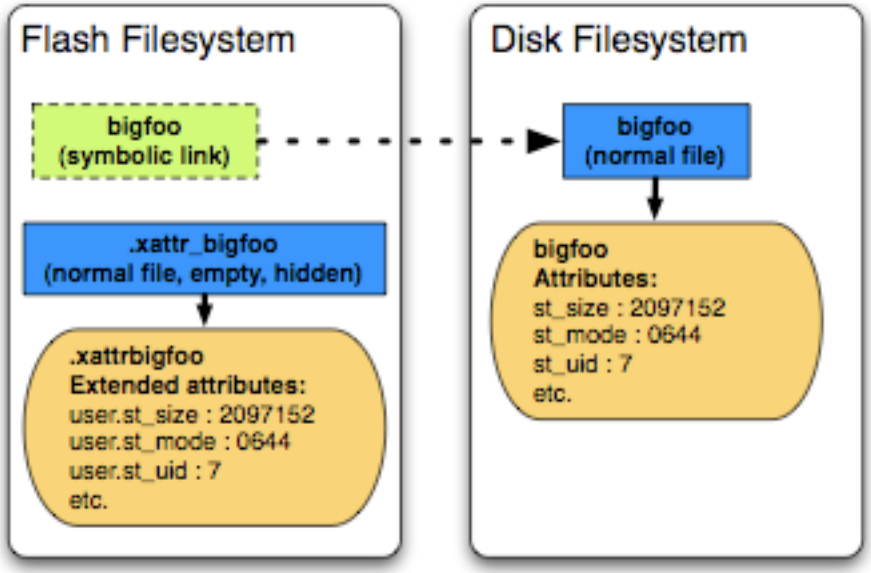


Skeleton Code for melangefs is provided to you in the distribution

1) App does create(f1.txt)
2) MelangeFS creates “f1.txt” in SSD
3) Ext2 on SSD returns a handle for “f1.txt” to FUSE
4) FUSE “translates” that handle into another handle which is is returned to the app
5) App uses the returned handle to write to “f1.txt” on the SSD
6) When “f1.txt” grows big, MelangeFS moves it to HDD, and “f1.txt” on the SSD becomes a symlink to the file on HDD
7) Because this migration has to be transparent, app continues to write as before (all writes go to the HDD).

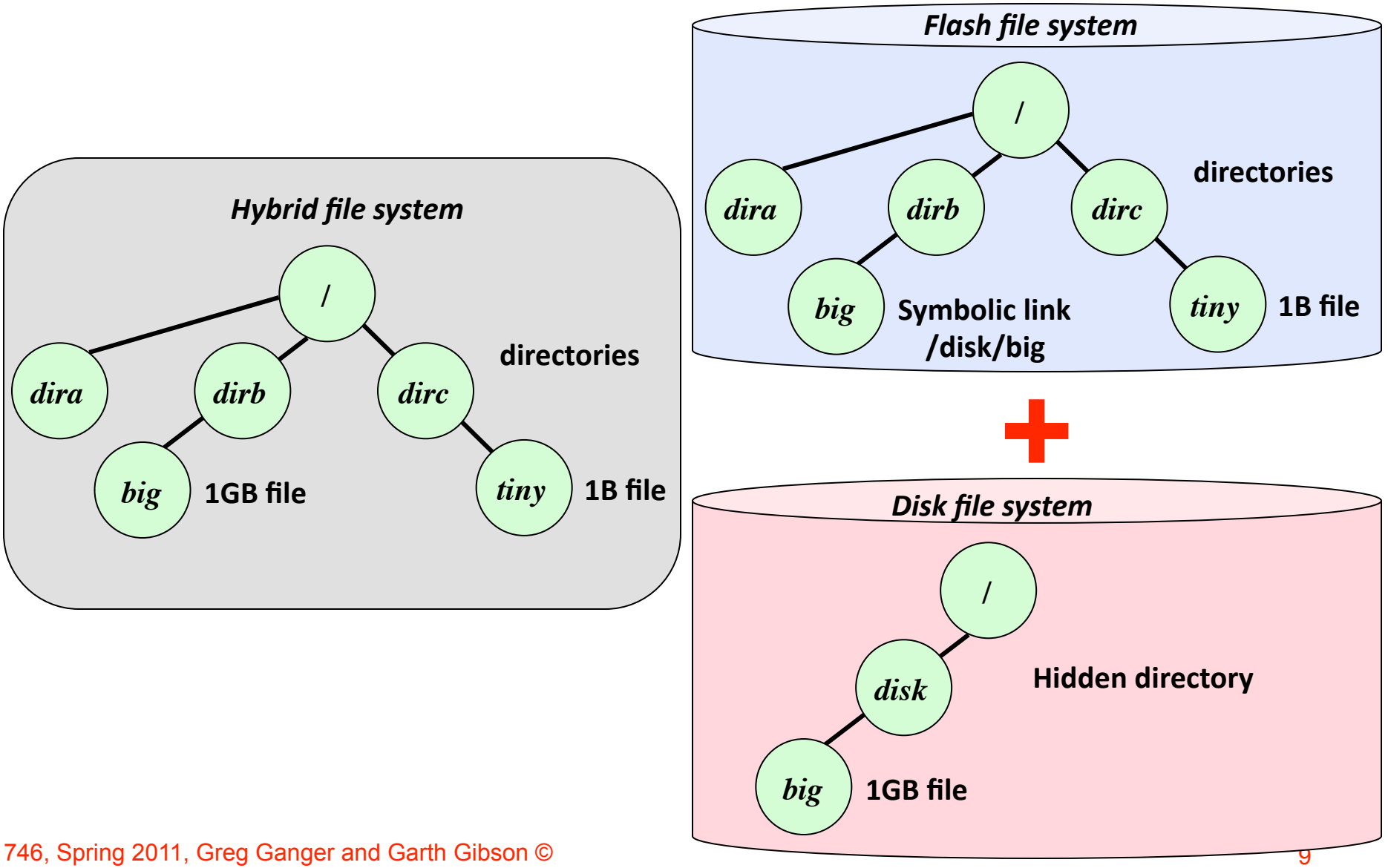
Attribute replication: avoid small IO on Disks

- After migration from SDD to HDD, “ls –laR” will follow the symlink and get the attributes from HDD
- To avoid doing small IOs on HDD, replicate “real” attributes from HDD to SSD
- Stored as empty, hidden “dot” file whose xattrs replicate the “real” attributes from HDD



(directory structure not shown)

Logical and Physical View of MelangeFS



Testing and Evaluation

- Test scripts (test_part1.sh) that perform three steps
  - Extract an .tar.gz file in the MelangeFS mount point
  - Compute checksum on all the files in the tar-ball
  - Perform a directory scan of all files in the tar-ball
- Script allows you to test with different dataset sizes
- To facilitate measurements, each test …
  - … will empty caches before runs by remounting melangeFS
  - … will measure number of block IOs using “vmstat”
  - *(virtualized setup makes time-based measurement hard)*
- More details in the README file in the “src/scripts/”

Expected output for correctness

- Test scripts returns the number of blocks read/written during each of the step (by parsing “vmstat –d”)

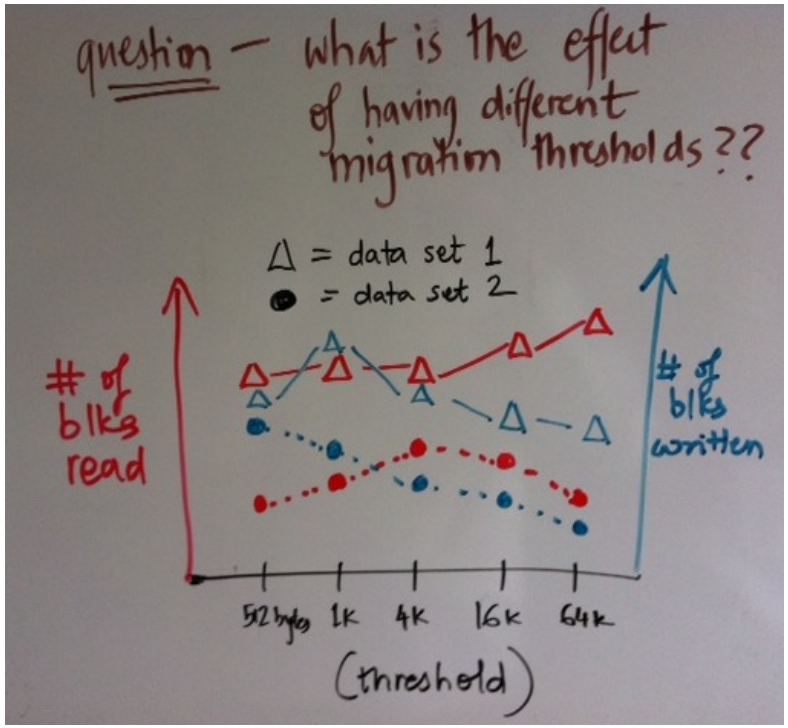
Testing step	Expected Correct Output
Extracting the TAR file in MelangeFS	For a small file, HDD should have zero blocks written
Performing a checksum on the all the files in the TAR file	For large files, HDDs should have a large number of blocks being read (compared to SSDs)
Scanning the whole directory of the TAR file using “ls –laR”	With attribute replication, only the SSD should have block reads (HDD should have zero reads)

- Other useful tool is btrace/blktrace

How to write the report?

- Key design ideas and data-structures
  - Pictures are useful; but good one need thought and time (start early ☺)
- Reason about the performance
  - Don't just copy-paste the output in the report
  - Show us that you know why it is happening

Data set	# of blks read SSD	# of blks read HDD	# of blks written SSD	# of blks written HDD
d1	123	23	756	345
d2	144	0	101	1623



Part 2: Faster Storage Management

- Storage management
  - UNIX examples: “find [filter]” and “du –s”
  - Other examples: context search, backups
- Key idea – scan the file system looking at attributes of all objects (and apply the user-defined “filter”)
  - `readdir() + lstat() on each object` (lots of random IO)
  - On current FSs, with 10^6 files, queries may run for hours
  - Future FSs, with 10^9 to 10^{12} files, may run for days/weeks
- Goal – design and implement optimizations to speed up storage management queries

Motivating example (on a real SSD)

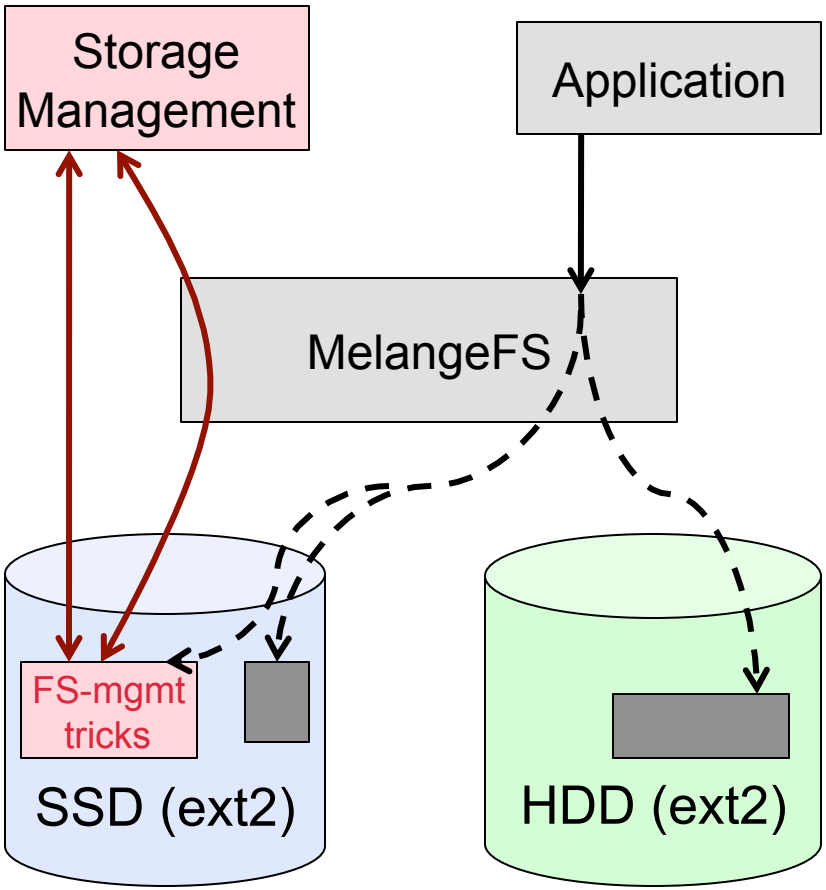
- Prototype implementation: Mac OS X (HFS) on SSD
  - Simple approach: “du” to list sizes of all files on one volume
  - My approach: built a way that avoids random IO (hint ☺)
- Performance results
  - Two cases: 1.15 million and 2 million files (different SSDs)
  - Average of three runs (zero variance) with empty caches
  - Of course, this is simple case with no real “computation”

	DataSet 1: 1.15M files	DataSet 2: 2M files
Completion time of *simple approach*	149 seconds	163 seconds
Completion time of *my approach*	1.3 seconds (>100X speed up)	1.4 seconds (>100X speed up)

Design decisions (and hints)

- Tradeoff – who does more work: the file system or the storage mgmt application?
  - No FS support for queries → may need to perform full scans
  - With FS support → fast queries, but high memory footprint
- Goal – what optimizations can FSs provide to speed up storage management queries?
  - Two kinds of queries: top-k and aggregator (read handout)
- Designs used by real file systems
  - IBM GPFS: ioctl() to dump i-nodes of all files and directories
  - Microsoft NTFS: replicate attributes in the directories
  - Use an external database to store all attributes

Storage Management Strategies



Skeleton Code for melangefs is provided to you in the distribution

- Applications continue to do their operations normally
- Your code in MelangeFS does the “cool things” to speed up storage management
- Storage management apps perform their queries using different ways
  - GPFS-like ioctl()
  - Leverage “cool things” directly
  - Or your trick

Problem Specification: top-k, aggregator

- Two kinds of storage management applications
  - `top-k`
  - `aggregator`
- You will implement these two user-level storage management programs
- DO's and DON'Ts
  - No in-memory solutions
  - Out-of-core structures can use space on SSDs
  - Storage management apps can ONLY read/lookup attributes; on the FUSE file system can write/update attributes and related data-structures

Testing and Evaluation

- Test scripts (test_part2.sh) that perform three steps
  - Starts with a file system with some data and run sample queries that emulate both “top-K” and “aggregator”
  - You should extend these scripts to use your of “top-k” and “aggregator” programs
- You will be graded on correctness and performance
- Examples of extending tests:
  - Performance – Run sample queries in the script, measure number of block IOs, perform queries using your program, measure again
  - Correctness – Run queries, check output, do some FS operations to files, run queries, check that output is correct

How to write the report?

- Key design ideas and data-structures
- Questions
  - What is the speed up in storage management?
  - How much more work did MelangeFS do to get this benefit (i.e., more block IO?)

Data set	Simple approach (# of blks read	Your approach # of blks read)
X1	**245**	**23**
X2	**378**	**32**

Data set	# of blks written without StrMgmt	# of blks written with StrMgmt
Y1	**37**	**50 (33% more)**
Y2	**124**	**250 (100% more)**

Once again – start early ☺

- Project due on April 11th 2011
  - Milestone1: finish part 1 by March 25th
  - Milestone 2: finish part 2 by April 6th
  - Leaves you a few days to work on performance

- Demo
  - VirtualBox
  - FUSE
  - Running tests

More analysis and performance debugging

Analyzing statistics tools

- First, what is in the flash and disk ext2 file systems?
  - ls –ailR /tmp/flash /tmp/disk
  - du –ak /tmp/flash /tmp/disk
    - Sorry for the lost+found side-effect of ext2 ☺
    - Look for the appropriate symbolic links and xattribute holding files
    - Look for the appropriate sizes of files (ext2 allocates 4KB blocks)
  - getfattr /tmp/flash/small/a/1 reports all attributes of file 1
- Next, what operations get done on the file systems?
  - In two separate terminal windows in Vbox, before your tests
  - [window1] btrace /dev/sdb1 | tee flashtrace
    - trace IOs to stdout (to watch) and file flashtrace (for later analysis)
  - [window2] btrace /dev/sdc1 | tee disktrace

Btrace output

- dev, cpu, seq, time, pid, action, iotype, address+length, info

```
8,16   0        1     0.000000000    23  A   W 8279 + 8 <- (8,17) 8216
8,17   0        2     0.000000844    23  Q   W 8279 + 8 [pdflush]
8,17   0        3     0.000012158    23  G   W 8279 + 8 [pdflush]
8,17   0        4     0.000016528    23  P   N [pdflush]
8,17   0        5     0.000018858    23  I   W 8279 + 8 [pdflush]
8,17   0        6     0.004114016     0 UT   N [swapper] 1
8,17   0        7     0.004132003    10  U   N [kblockd/0] 1
8,17   0        8     0.004152279    10  D   W 8279 + 8 [kblockd/0]
8,17   0        9     0.005084990     0  C   W 8279 + 8 [0]
8,16   0       10    22.818093118  3234  A   R 12375 + 8 <- (8,17) 12312
8,17   0       11    22.818093658  3234  Q   R 12375 + 8 [flashndisk]
8,17   0       12    22.818105498  3234  G   R 12375 + 8 [flashndisk]
8,17   0       13    22.818106768  3234  P   N [flashndisk]
8,17   0       14    22.818108018  3234  I   R 12375 + 8 [flashndisk]
8,17   0       15    22.818112428  3234  U   N [flashndisk] 1
8,17   0       16    22.818120222  3234  D   R 12375 + 8 [flashndisk]
8,17   0       17    22.818375643  3234  C   R 12375 + 8 [0]
8,16   0       18    22.818625434  3234  A   R 20709447 + 8 <- (8,17) 20709384
8,17   0       19    22.818625941  3234  Q   R 20709447 + 8 [flashndisk]
8,17   0       20    22.818627933  3234  G   R 20709447 + 8 [flashndisk]
8,17   0       21    22.818629149  3234  P   N [flashndisk]
8,17   0       22    22.818629811  3234  I   R 20709447 + 8 [flashndisk]
8,17   0       23    22.818633398  3234  U   N [flashndisk] 1
8,17   0       24    22.818640422  3234  D   R 20709447 + 8 [flashndisk]
8,17   0       25    22.825372038     0  C   R 20709447 + 8 [0]
```

Understanding btrace output

- btrace is really blktrace | blkparse
- man blkparse tells you how to read the output
- As our devices are virtual, time is not very interesting
- We care about numbers of sectors read and written
- “Action C” is completion of an IO (address + length)
- Types are R read, W write, RA readahead
- Next you want to understand what is being read or written – need to tie “address” to disk structure
  - debugfs /dev/sdb1 # to debug ext2 file system on flash

Using debugfs to map files to disk sectors

- imap /small/a/1 # where is inode for file 1
  - Inode XXXXXX is part of block group YY
    located at block AAAA, offset 0x0400
- bmap /small/a/1 0 # block number of block 0 in file 1
  - AAAA
- Blocks are not sectors, and the disk image is offset
  - Blktrace sector address = AAAA*8+63
  - Because blocks are 8 sectors, and the flash and disk images are in partition 1, which is 63 sectors into the disk
    - ```
      %> fdisk -lu /dev/sdb
      Disk /dev/sdb: 17.1 GB, 17179869184 bytes, 255 heads, 63 sectors/track, 2088
      cylinders, total 33554432 sectors; Units = sectors of 1 * 512 = 512 bytes;
      Disk identifier: 0x3bac36e8
      Device Boot      Start         End        Blocks   Id  System
      /dev/sdb1              63    33543719    16771828+  83   Linux
      ```
      ```

So the Analysis ……

- Can you attribute every sector read and written during your runs of md5 and ls on the flash, the disk?
  - Remember free list bitmaps for inodes and data blocks
  - Remember directory entries
  - Remember indirect blocks
  - Remember extended attributes (linked like indirect blocks)
  - Remember that inodes are smaller than blocks
  - Remember that “allocate, free, allocate” may be a new block
  - Accounting for everything may be hard, just try your best
- How well does this correspond to “small random on flash, large sequential on disk”?