science + computing

A Bull Group Company

Lustre –
Finding the Filesystem Bottleneck

Daniel Kobras

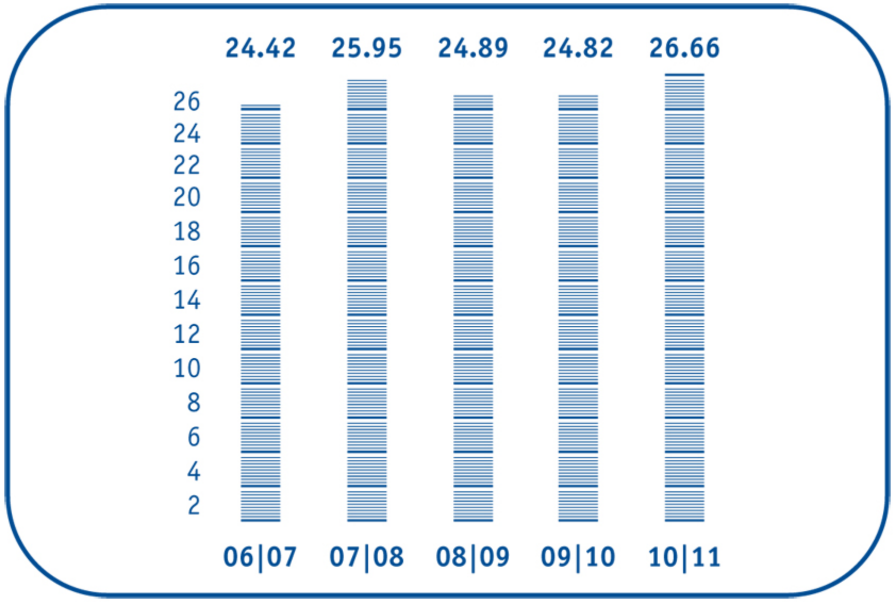
science + computing ag

IT-Dienstleistungen und Software für anspruchsvolle Rechnernetze

Tübingen | München | Berlin | Düsseldorf

science+computing

Founded in	**1989**
Offices	**Tuebingen**
	Munich
	Duesseldorf
	Berlin
Employees	**268**
Shareholder	**Bull S.A. (100%)**
Turnover 10/11	**26.7 Mio. EUR**



Portfolio

IT Service for complex computing environments
Complete solutions for Linux- and Windows-based **HPC**

scVENUS System management software for efficient administration of homogeneous and heterogeneous environments

Motivation

science + computing | A Bull Group Company

`'Dear admin, filesystem is slow, please fix'`

- Performance problems are among the hardest problems to debug
- Often no error messages available
- Finding root cause is hard, especially in distributed systems comprising of many components

- May not be an actual problem at all, but
  - Overload from legitimate use
  - Overload because of (deliberate) imbalanced sizing
  - Unrealistic expectations

Performance debugging roadmap

- **Check servers**
  - Are the servers (over)loaded?
  - Which servers are (over)loaded?
  - Which operations are (over)loading the filesystem?
  - Which clients are (over)loading the filesystem?

- **Check clients**
  - Which processes/users are (over)loading the filesystem?

- **Check applications**
  - Why are the processes (over)loading the filesystem?

Assumptions

- Examples assume Lustre filesystem version 1.8
- No apparent errors on
    - Interconnects
    - Clients
    - Servers
    - Storage backends

Checking Lustre servers

- Find out what keeps Lustre servers busy
- Necessary information readily available in stats files
- No problem, case closed

Checking Lustre servers

- Find out what keeps Lustre servers busy
- Necessary information readily available in stats files
- ~~No problem, case closed~~
- Except for **information overload**
- Example from a production environment
  MDS
  ```
  # find /proc/fs/lustre -name "*stats*" | wc -l
  2499
  ```
  (9 files/MDS + 4 files/MDT + 3 files/OST + 4 files/client)

  OSS
  ```
  # find /proc/fs/lustre -name "*stats*" | wc -l
  11034
  ```
  (6 files/OSS + 5 files/OST + 3 files/[client*OST])

The Needle in the Haystack

science + computing
A Bull Group Company

Too many sources to monitor, check, and understand

- Tools like llstat, lstats.sh, ltrack_stats, lustre_req_history, collectl etc. help collecting, but not reducing and interpreting information
- Instead, create artificial load that mimicks **typical usage**
- Watch statistics to identify relevant files and lines
- Provides small subset of information sources to look at **first**
- Not comprehensive, but useful for **fast initial debugging** of common scenarios

- Common troublemakers:
  `ls -lR`, creating many small files, **`rm -rf`**, small random i/o, heavy bulk i/o

Metadata loads: `readdir()+stat()`

- This and similar types of load are created from recursive filesystem scans like **`ls -lR`, `find -newer`, `du -s`**, etc.
- Shows up in statistics on MDS

Host	File	Counter	Comment
MDS	/proc/fs/lustre/mdt/MDS/mds_readpage/stats	mds_readpage mds_close	readdir()
MDS	/proc/fs/lustre/mdt/MDS/mds/stats	mds_getattr	min/avg/max
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	getattr	

- min/avg/max make it easier to tell apart unusual load spikes
- Example:

```
mds# llstat -i 1 /proc/fs/lustre/mdt/MDS/mds/stats
Name            Cur.Count  Cur.Rate    #Events    Unit     last    min
    avg     max     stddev
mds_getattr     0          0           88727997   [usec]   0       5
    22.86   287438 295.06
```

Metadata loads: `readdir()+unlink()`

- Typical loads: job cleanup, transfer scripts, `rm -rf`
- Shows up primarily in statistics on MDS

Host	File	Counter	Comment
MDS	/proc/fs/lustre/mdt/MDS/mds/stats	mds_readpage mds_close	readdir()
MDS	/proc/fs/lustre/mdt/MDS/mds/stats	mds_reint_unlink	min/avg/max
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	unlink	

Metadata loads: file creates

- Typical loads: job output into many small files
- Shows up primarily in statistics on MDS

Host	File	Counter	Comment
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	open setattr	

- There's also a counter called **`create`**, but creates are (usually) accounted in **`open`**

Metadata loads: summary

Host	File	Counter	Example
MDS	/proc/fs/lustre/mdt/MDS/mds_readpage/stats	mds_readpage mds_close	`ls -R`
MDS	/proc/fs/lustre/mdt/MDS/mds/stats	mds_getattr	`ls -lR`
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	getattr	`ls -lR`
MDS	/proc/fs/lustre/mdt/MDS/mds/stats	mds_reint_unlink	`rm -rf`
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	unlink	`rm -rf`
MDS	/proc/fs/lustre/mds/<fsname>-MDT0000/stats	open setattr	`touch`

Data I/O: generic statistics

- Read/write I/O statistics accounted by I/O request (per server), and by throughput (per OST)

Host	File	Counter	Comment
OSS	/proc/fs/lustre/ost/OSS/ost_io/stats	ost_read ost_write	min/avg/max
OSS	/proc/fs/lustre/obdfilter/<fsname>-OST*NNNN*/stats	read_bytes write_bytes	use `llobdstat`

- Provides general overview
- Cannot distinguish between types of I/O (sequential vs. random, small vs. large I/O request size)
- Example:

```
# llobdstat /proc/fs/lustre/obdfilter/aerohpc1-OST0016/stats 1
Timestamp   Read-delta  ReadRate  Write-delta  WriteRate
1348329721    0.00MB    0.00MB/s   139.00MB  138.85MB/s
1348329722    0.07MB    0.07MB/s    83.00MB   82.92MB/s
```

Data I/O: detailed statistics

- Detailed I/O statistics collected per OST allow to identify well-behaved (large, sequential) and ill-behaved (small, random) I/O patterns

Host	File	Counter	Ideal
OSS	/proc/fs/lustre/obdfilter/<fsname>-OST*NNNN*/brw_stats	pages per bulk r/w	most RPCs at max
		discontiguous pages/blocks	most RPCs at 0
		disk fragmented I/O	most ios at 1
		disk I/O size	most ios at 1M

- Read statistics are usually close to ideal (read-ahead), write statistics can reveal ill-behaved I/O

Identifying source of I/O

- Type of problematic I/O should now be known
- Need to find source of I/O
    - Check per-client (per NID) statistics
    - Derive information from RPC request history
- Using RPC request history usually easier
- All per-server stats files accompanied by req_history files providing history of last RPC requests
- RPC history deactivated by default

Using RPC request history

- Activate RPC history by configuring non-zero buffer size, eg.

```
# lctl set_param \
  ost.OSS.ost_io.req_buffer_history_max=10240
```

  (saves last 10k I/O RPCs on an OSS)
- After a while, read out RPC history, eg.

```
# lctl get_param ost.OSS.ost_io.req_history
```

- Output format
  identifier:target_nid:source_nid:rpc_xid:rpc_size:rpc_status:arrival_time:service_time(deadline) opcode, eg.

```
4134542441:10.1.2.3@o2ib:12345-10.1.2.4@o2ib
:x1406392481581555:448:Complete:1348243976:
0s(-8s) opc 3
```

- Filtering by **opcode** and accounting by **source_nid** reveals client(s) producing the most of the problem RPCs

Matching opcodes to stats

- RPC requests identified by names in stats files, but by numbers in req_history
- Mapping in header `lustre/include/lustre/lustre_idl.h`

Opc	Makro	Opc	Makro	Opc	Makro	Opc	Makro
0	OST_REPLY	11	OST_OPEN	37	MDS_READPAGE	48	MDS_QUOTACTL
1	OST_GETATTR	12	OST_CLOSE	38	MDS_CONNECT	49	MDS_GETXATTR
2	OST_SETATTR	13	OST_STATFS	39	MDS_DISCONNECT	50	MDS_SETXATTR
3	OST_READ	16	OST_SYNC	40	MDS_GETSTATUS	101	LDLM_ENQUEUE
4	OST_WRITE	17	OST_SET_INFO	41	MDS_STATFS	102	LDLM_CONVERT
5	OST_CREATE	18	OST_QUOTACHECK	42	MDS_PIN	103	LDLM_CANCEL
6	OST_DESTROY	19	OST_QUOTACTL	43	MDS_UNPIN	400	OBD_PING
7	OST_GET_INFO	33	MDS_GETATTR	44	MDS_SYNC	401	OBD_LOG_CANCEL
8	OST_CONNECT	34	MDS_GETATTR_NAME	45	MDS_DONE_WRITING	402	OBD_QC_CALLBACK
9	OST_DISCONNECT	35	MDS_CLOSE	46	MDS_SET_INFO	(...)	(...)
10	OST_PUNCH	36	MDS_REINT	47	MDS_QUOTACHECK		

Client-side statistics

- With client NID known, need to find process that is I/O source
- Usually easy task on cluster nodes running single/few jobs
- Much harder on large multi-user systems
- Determine PIDs with active Lustre I/O, (ab)using extents stats
- Limit statistics to subset of processes

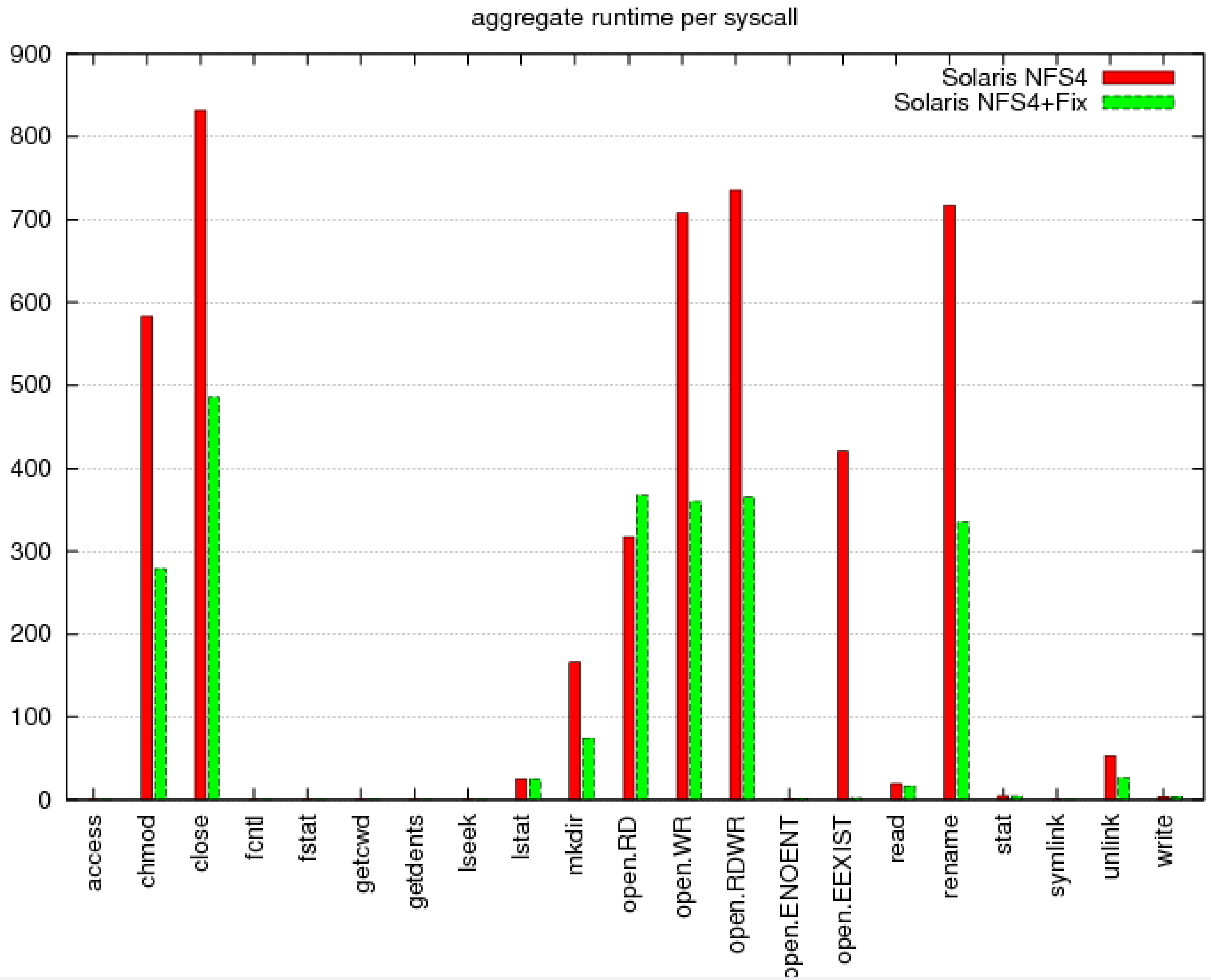
Host	File	Comment
Client	`/proc/fs/lustre/llite/<fsname>-<uuid>/extents_stats_per_process`	PIDs with active I/O
Client	`/proc/fs/lustre/llite/<fsname>-<uuid>/stats`	Generic client-side stats
Client	`/proc/fs/lustre/llite/<fsname>-<uuid>/stats_track_pid`	Limit stats to PID
Client	`/proc/fs/lustre/llite/<fsname>-<uuid>/stats_track_ppid`	Limit stats to PPID
Client	`/proc/fs/lustre/llite/<fsname>-<uuid>/stats_track_gid`	Limit stats to GID

Simple I/O profiles

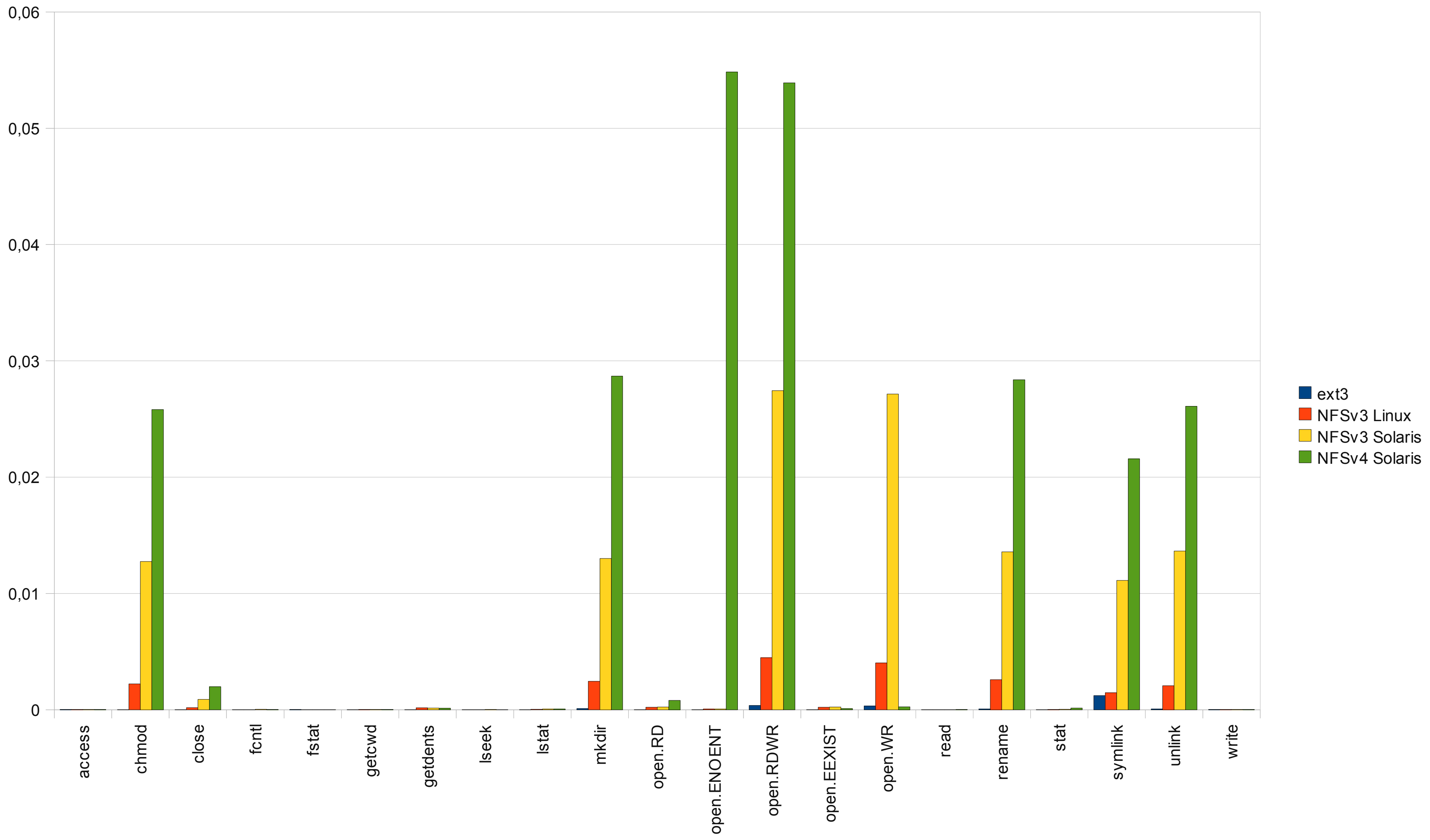
Never attribute to the filesystem that which is adequately explained by application stupidity.

- Trace slow or misbehaving applications
- Try to determine where most of I/O time is spent
- **`strace -T`** is a great tool to easily obtain simple I/O profiles
- Practical example:
  Slow checkout of large SVN repository

I/O profile


aggregate runtime per syscall
Solaris NFS4
Solaris NFS4+Fix
900
800
700
600
500
400
300
200
100
0
access
chmod
close
fcntl
fstat
getcwd
getdents
lseek
lstat
mkdir
open.RD
open.WR
open.RDWR
open.ENOENT
open.EEXIST
read
rename
stat
symlink
unlink
write

Filesystem profiles



Conclusion

- Lustre provides huge amount of profiling information
- Pinpointing the right information can be challenging to the uninitiated
- Pre-defined subset of information useful for monitoring, or initial checks to identify most common usage scenarios
- Simple application profiling can reveal sub-optimal I/O patterns
- Optimising applications can be more effective than filesystem tuning

science + computing
A Bull Group Company

Thank you!

Daniel Kobras

science + computing ag

www.science-computing.de

www.hpc-wissen.de

Telefon 07071 9457-0

info@science-computing.de