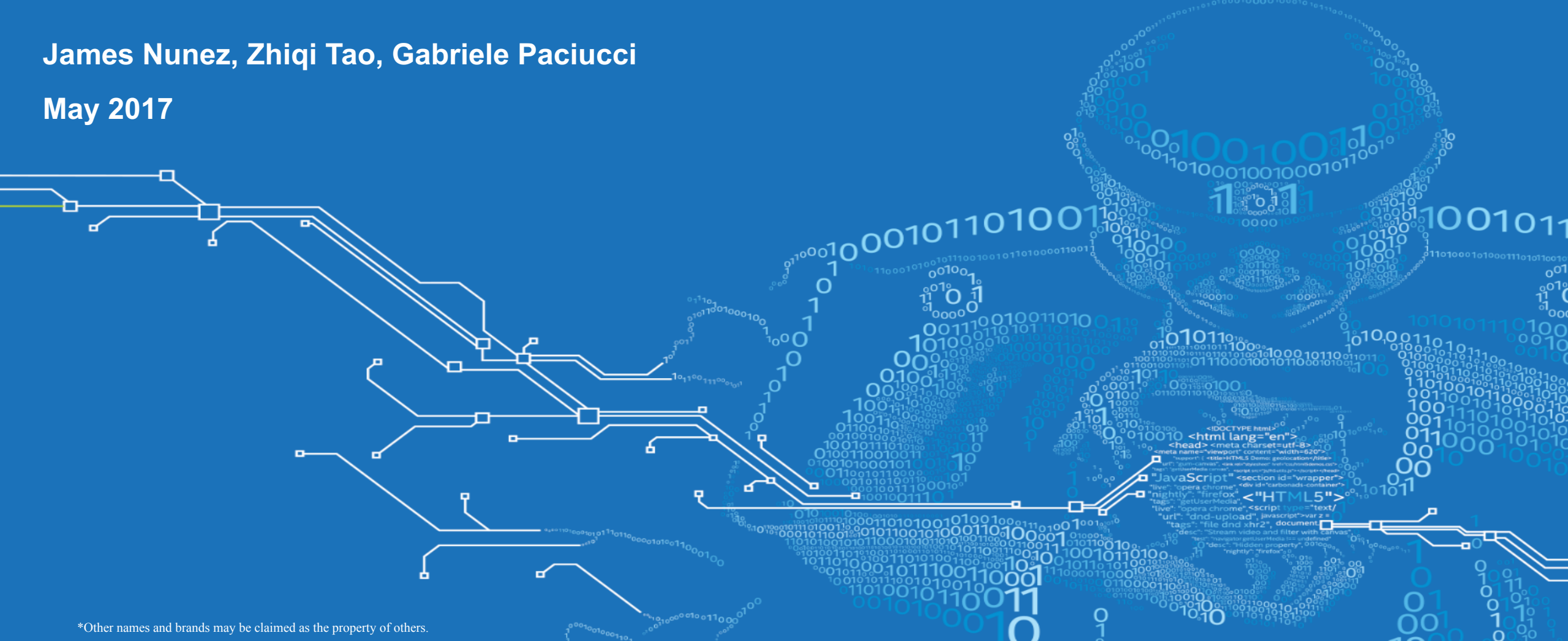
Profiling Application IO Pattern with Lustre*

James Nunez, Zhiqi Tao, Gabriele Paciucci

May 2017



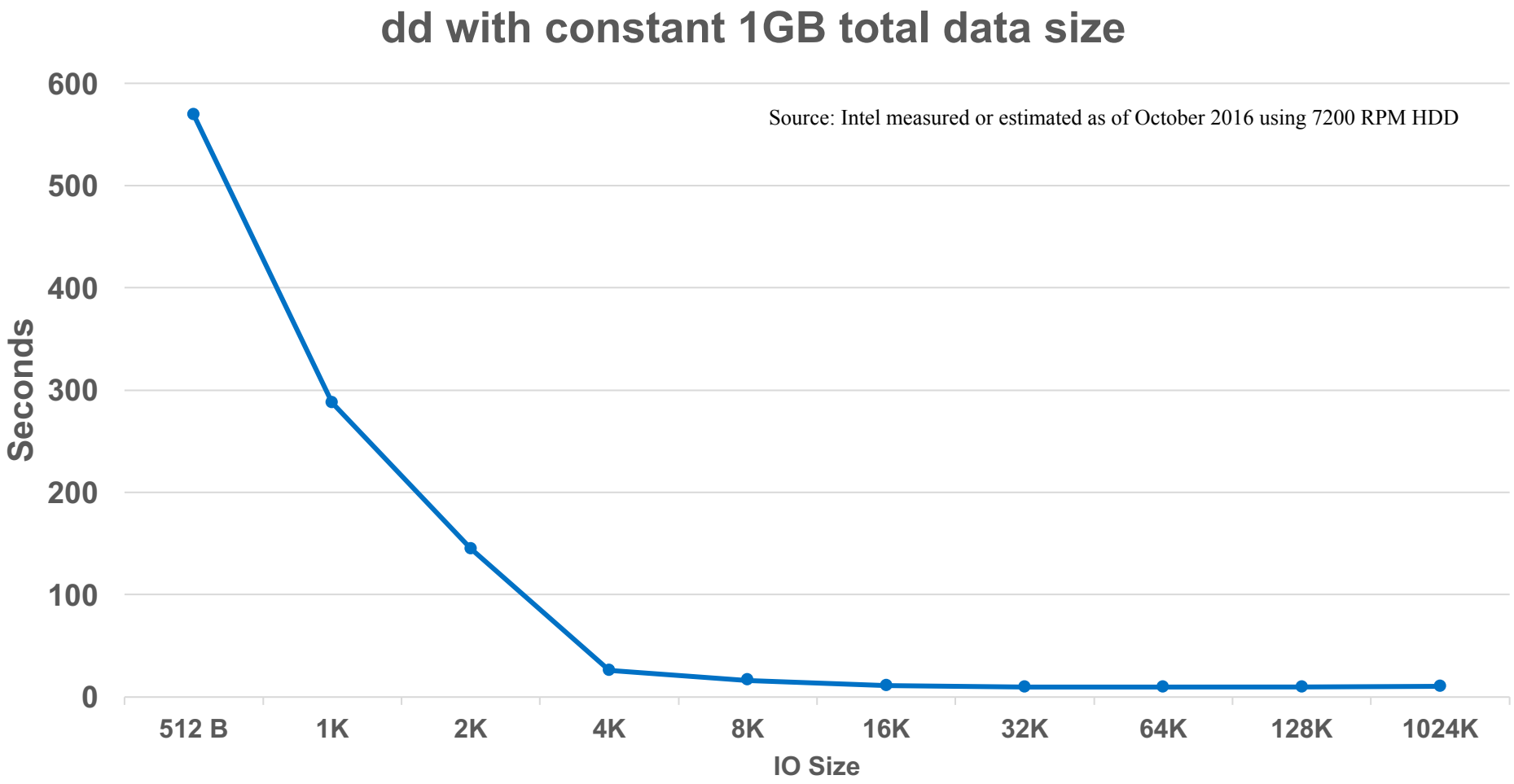
*Other names and brands may be claimed as the property of others.

Outline

- Why profile application IO patterns?
- What do we need in order to understand the Application IO?
- A brief intro about how Lustre works
- A few synthetic IO examples
- A real example – Nemo Ocean
- What we could learn from the Application IO pattern?
- Tune to make Applications run better

Why Profile Application IO patterns?

Different IO patterns would result in significantly different IO performance



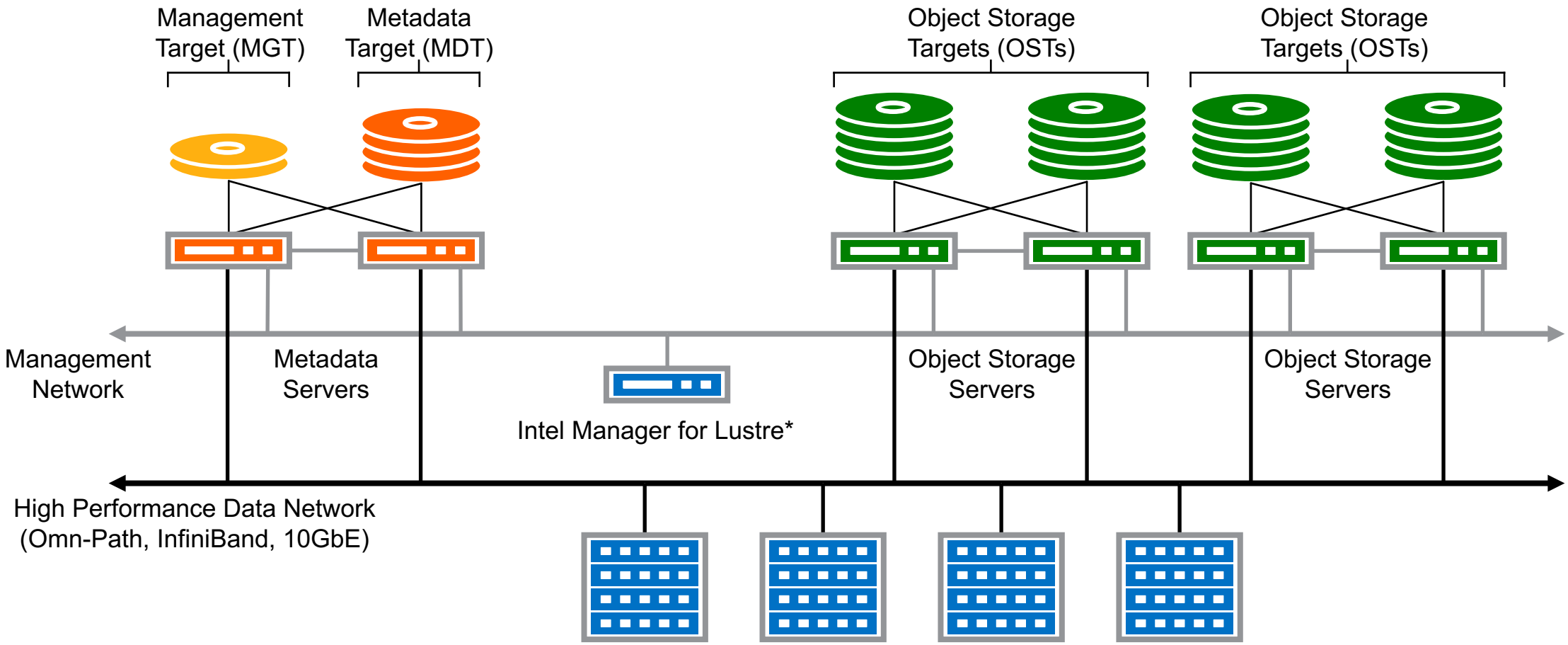
intel Software

What do we need in order to understand the Application IO?

1. The size of data an application would generate
2. The number of files an application would generate
3. The distribution of File sizes
4. The distribution of File IOs
5. The number of IO processes going on simultaneously

intel Software

What Does a Lustre* Solution Look Like?

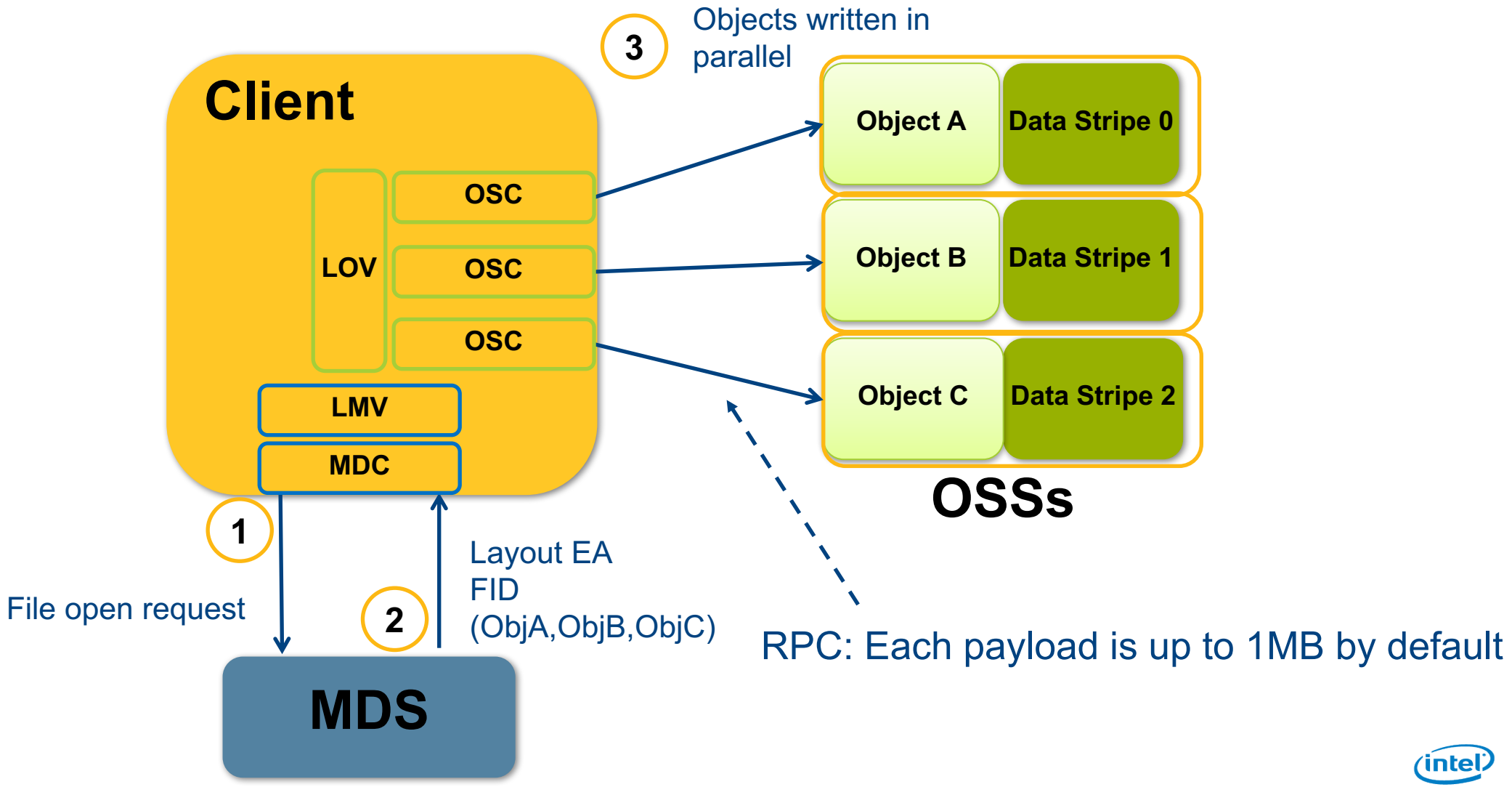


Where Applications Run

intel Software

*Other names and brands may be claimed as the property of others.

Basic Lustre I/O Operation



intel Software

Write a file

1) Run a simple DD command

```
# dd if=/dev/zero of=/mnt/edafs/ost01/file_1 count=100 bs=10M
...
1048576000 bytes (1.0 GB) copied, 0.887849 s, 1.2 GB/s
```

2) Verify the number of IOs and the IO size

```
# lctl get_param llite.edafs-*.extents_stats
                read                    |     write
extents         calls    % cum%         |     calls    % cum%
0K - 4K   :         0     0     0       |         0     0     0
    :
4M - 8M   :         0     0     0       |         0     0     0
8M - 16M  :         0     0     0       |       100   100   100
```

intel Software

Write a big file with really large IO size

1) Reset the counters and Run a simple DD command

```
# lctl set_param llite.edafs-*.extents_stats=1
# dd if=/dev/zero of=/mnt/edafs/ost01/file_1GB count=1 bs=1024M
```

2) Verify the number of IOs and the IO size

```
# lctl get_param llite.edafs-*.extents_stats
snapshot_time:         1477427524.133356 (secs.usecs)
                            read            |           write
    extents                calls    %  cum%  |           calls    %  cum%
  0K -    4K :                 0    0     0  |               0    0     0
  4K -    8K :                 0    0     0  |               0    0     0
      :
256M -  512M :                 0    0     0  |               0    0     0
512M - 1024M :                 0    0     0  |               0    0     0
  1G -   2G :                  0    0     0  |               1  100   100
```

intel Software

Write several files and each with different IO sizes

1) Reset the counters and Run three simple DD command

```
# lctl set_param llite.edafs-*.extents_stats=1
# dd if=/dev/zero of=/mnt/edafs/ost01/file_1GB count=1024 bs=1M &
# dd if=/dev/zero of=/mnt/edafs/ost01/file_2 count=1024 bs=128k &
# dd if=/dev/zero of=/mnt/edafs/ost01/file_3 count=1024 bs=31k &
```

2). Check the counters

```
# lctl get_param llite.edafs-*.extents_stats
                            read        |              write
     extents                calls    %  cum%  |          calls    %  cum%
       :
   16K -   32K :                0    0     0  |           1024   33   33
       :
  128K - 256K :                 0    0     0  |           1024   33   66
       :
    1M -    2M :                0    0     0  |           1024   33  100
```

How do we match which process was generating which IO?

intel Software

Check the IO pattern of each process

```
# dd if=/dev/zero of=/mnt/edafs/ost01/file_1GB count=1024 bs=1M &   [1] 225695
# dd if=/dev/zero of=/mnt/edafs/ost01/file_2 count=1024 bs=128k &   [1] 225709
# dd if=/dev/zero of=/mnt/edafs/ost01/file_3 count=1024 bs=31k &    [1] 225716

# lctl get_param llite.edafs-*.extents_stats_per_process
                            read            |                write
     extents                calls    % cum%  |           calls    % cum%

PID: 225695
           :
   1M -    2M :                0    0    0  |            1024  100  100

PID: 225709
           :
 128K -  256K :                0    0    0  |            1024  100  100

PID: 225716
           :
  16K -   32K :                0    0    0  |            1024  100  100
```

intel Software

How does Lustre actually handle the different IO size?

```
# lctl set_param osc.edafs-OST0001*.rpc_stats=0
# dd if=/dev/zero of=/mnt/edafs/ost01/file_2 count=8192 bs=128K
1073741824 bytes (1.1 GB) copied, 1.08719 s, 988 MB/s

# lctl get_param osc.edafs-OST0001*.rpc_stats

                        read                    write
pages per rpc         rpcs   % cum % |       rpcs   % cum %
  :
256:                     0   0   0   |       1024 100 100
```

Lustre can aggregate small IOs together

intel Software

But no aggregation when “Sync” IO is enforced

1) Use the “sync” IO

```
# dd if=/dev/zero of=File_2 count=1024 bs=100K oflag=sync
104857600 bytes (105 MB) copied, 2.58155 s, 40.6 MB/s
```

2). Verify the number of RPCs and size

```
# lctl get_param osc.krakenfs-OST0001*.rpc_stats
snapshot_time:          1477343197.889076 (secs.usecs)
read RPCs in flight:  0
write RPCs in flight: 0
pending write pages:  0
pending read pages:   0
                        read                    write
pages per rpc           rpcs   % cum % |       rpcs   % cum %
  :
32:                        0   0   0   |       1024 100 100
```

intel Software

But no aggregation when “Sync” IO is enforced

The rpc_stats counter is associated with each OST

```
# lctl get_param osc.krakenfs-OST0001*.rpc_stats
```

To limit IO to only one OST, we could use OST pool

```
[root@kcli01 ost01]# lfs pool_list edafs.ost01
Pool: edafs.ost01
edafs-OST0001_UUID
[root@kcli01 ost01]# lfs setstripe -p edafs.ost01 ost01
[root@kcli01 ost01]# lfs getstripe ost01
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pool:           ost01
        obdidx           objid           objid           group
             1          652559        0x9f50f               0
```

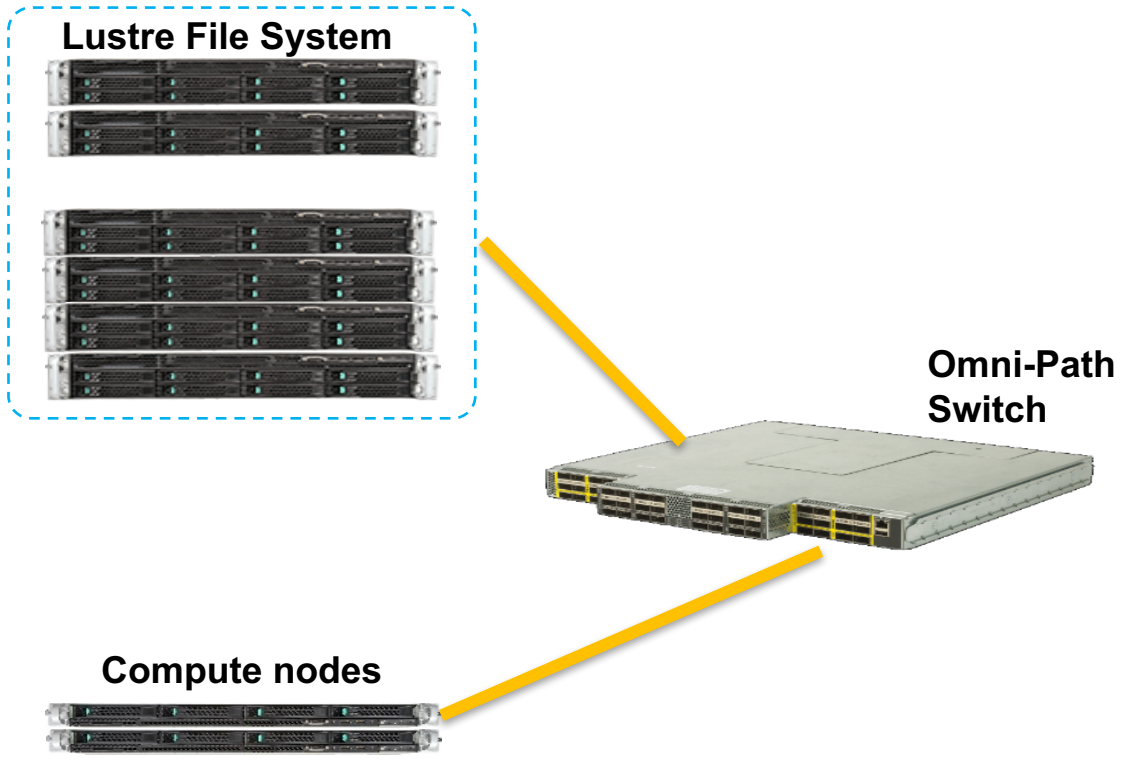
intel Software

A real application case – IO Profiling with Nemo Ocean

http://www.nemo-ocean.eu/ **A popular Ocean Modeling framework**

Goals:

1. Understand the Nemo IO Pattern
   - Hard to find IO pattern with many processes running
2. Tune to make Nemo run better



Test Environment:

- 2x compute nodes
- OmniPath as the Lustre network
- Lustre file system based on SATA SSD

The Profiling Methodology

1). Let Nemo write/read all from one directory that is striped to one OST pool

2). Reset the counters on the KNL node side

3). Check the IO patterns and also CPU utilizations

4). Evenly distribute workloads to two KNL nodes

```
nemo_cores=64
xios_cores=2
ppn=33

mpiexec.hydra -perhost $ppn -f hostfile -n $[nemo_cores / 2] ./nemo.$ntype.exe \
                                : -n $[xios_cores / 2] ./xios_server.$xtype.exe \
                                : -n $[nemo_cores / 2] ./nemo.$ntype.exe \
                                : -n $[xios_cores / 2] ./xios_server.$xtype.exe
```

When it started, NEMO read some files which are all kinds of different sizes

```
[RHEL7.2 dknl03 20161025_1653 ~]# lctl get_param llite.zfs-*.extents_stats
                              read        |                 write
     extents             calls     %  cum%  |         calls     %  cum%
   0K -     4K :          1864    11    11  |          3409    98    98
   4K -     8K :            16     0    11  |             0     0    98
   8K -    16K :             6     0    11  |             0     0    98
  16K -    32K :            14     0    11  |             0     0    98
  32K -    64K :            32     0    11  |             0     0    98
  64K -   128K :            51     0    11  |             1     0    98
 128K -   256K :           116     0    12  |             0     0    98
 256K -   512K :           236     1    13  |             0     0    98
 512K -  1024K :           647     3    17  |             0     0    98
   1M -     2M :          1820    10    28  |             0     0    98
   2M -     4M :         11390    68    96  |             1     0    98
   4M -     8M :           551     3   100  |            47     1   100
```

IO Pattern Emerges - 4KB read, 4MB write

```
# lctl get_param llite.zfs-*.extents_stats
                                read        |          write
       extents            calls    % cum%   |     calls    % cum%
   0K -    4K :            3273   80   80   |       658    5    5
   4K -    8K :               0    0   80   |         0    0    5
   8K -   16K :               0    0   80   |         2    0    5
  16K -   32K :               0    0   80   |         4    0    5
  32K -   64K :               1    0   80   |         2    0    5
  64K -  128K :               2    0   80   |         2    0    5
 128K -  256K :               4    0   80   |        34    0    5
 256K -  512K :               6    0   81   |        85    0    6
 512K - 1024K :              39    0   81   |       162    1    7
   1M -    2M :             156    3   85   |       276    2    9
   2M -    4M :             565   13   99   |       372    2   12
   4M -    8M :               9    0  100   |      9954   79   92
   8M -   16M :               0    0  100   |       939    7  100
```

intel Software

Determine process generating IOs via extent_stats_per_process

```
# lctl get_param llite.zfs-*.extents_stats_per_process
                                read        |                write
      extents              calls     % cum%  |           calls     % cum%

PID: 272699
   0K -     4K :                1   100   100  |                0     0     0
      :
   8M -    16M :                0     0   100  |                1   100   100

PID: 272718
...
```

It's nemo.par who does small read and large write!

```
# ps fx|grep 272699
272803 pts/3    S+      0:00          \_ grep 272699
272699 pts/2    Rl      3:43              |   \_ ./nemo.par.exe
```

intel Software

Confirm the process generating IOs

Both `Nemo.par` and `xios_server.par` saturated the CPU cores

```
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
     :
272723 root      20   0 6107576 2.496g  22516 R  99.1  2.3   6:19.82 nemo.par.exe
272724 root      20   0 6107528 2.493g  22428 R  99.1  2.3   6:19.47 nemo.par.exe
272725 root      20   0 6107524 2.488g  22540 R  99.1  2.3   6:19.27 nemo.par.exe
272727 root      20   0 6108472 2.493g  22524 R  99.1  2.3   6:19.34 nemo.par.exe
272728 root      20   0  0.583t 1.525g  13276 R  99.1  1.4   7:00.23 xios_server.par
```

But `xios.server.par` hardly does any IO as its PID did not appear in the output of

```
# lctl get_param llite.zfs-*.extents_stats_per_process|grep 272728
```

But nemo.par.exe does so

```
# lctl get_param llite.zfs-*.extents_stats_per_process|grep 272723
PID: 272723
```

Randomly checked several `nemo.par` processes. All did the same IO pattern as previous page

intel Software

So what actual IOs went to Lustre?

```
# lctl get_param osc.zfs-OST000e*.rpc_stats

                        read                    write
pages per rpc         rpcs   %  cum %  |       rpcs   %  cum %
1:                     675  86    86    |        147   0    0
2:                      14   1    88    |        135   0    0
4:                       0   0    88    |        233   0    0
8:                       3   0    89    |        588   0    0
16:                      5   0    89    |       1137   0    1
32:                      2   0    89    |       2782   1    3
64:                      2   0    90    |       7354   4    7
128:                     4   0    90    |      18430  11   18
256:                    72   9   100    |     134787  81  100
```

There are some 4K IOs

intel Software

After reset the counter

```
# lctl get_param osc.zfs-OST000e*.rpc_stats

                           read                     write
pages per rpc         rpcs   %  cum %  |       rpcs   %  cum %
1:                       0   0      0   |         16   0    0
2:                       0   0      0   |          8   0    0
4:                       0   0      0   |         24   0    0
8:                       0   0      0   |         54   0    0
16:                      0   0      0   |        112   0    1
32:                      0   0      0   |        243   1    2
64:                      0   0      0   |        695   4    7
128:                     0   0      0   |       1643  10   18
256:                     0   0      0   |      12611  81  100
```

So these 4K IOs are repetitive data. After they have been read, the Lustre client would cache them and therefore no need to read from Lustre server again

intel Software

Multiple Stages of the Nemo run show the similar IO Patterns

```
# lctl get_param llite.zfs-*.extents_stats
```

extents	read calls	read %	read cum%	write calls	write %	write cum%
0K - 4K :	17098	100	100	3055	3	3
4K - 8K :	0	0	100	8	0	3
8K - 16K :	0	0	100	4	0	3
16K - 32K :	0	0	100	32	0	3
32K - 64K :	0	0	100	108	0	3
64K - 128K :	0	0	100	190	0	3
128K - 256K :	0	0	100	572	0	4
256K - 512K :	0	0	100	904	0	5
512K - 1024K :	0	0	100	1702	1	7
1M - 2M :	0	0	100	3290	3	10
2M - 4M :	0	0	100	6461	7	18
4M - 8M :	0	0	100	61863	68	86
8M - 16M :	0	0	100	12452	13	100

Software

What we learnt from the profiling exercise?

- Since these small 4K reads are mostly cached, all we need to do is to tune for large sequential writes, which is Lustre's sweet spot
- This explained why we had much better performance after migrating to Lustre

Does the IO profiling method require the root access?

Lustre runs in the kernel space. All of counter resets would require root. But if you don't mind doing some math, you can get these stats as a regular user.

```
$ lctl get_param osc.zfs-OST000e*.rpc_stats
snapshot_time:         1477440651.11612 (secs.usecs)
                        read                    write
pages per rpc          rpcs   % cum % |       rpcs   % cum %
  :
256:                     90  96 100   |       2171  99 100
```

```
$ lctl get_param osc.zfs-OST000e*.rpc_stats
snapshot_time:         1477440659.903599 (secs.usecs)
read                    write
pages per rpc          rpcs   % cum % |       rpcs   % cum %
  :
256:                     90  96 100   |       2465  99 100
```

256 pages per RPC means 1MB IO.
In 8.7875 seconds, 294 MB data was sent to the OST000e on the Lustre file system

intel Software

Summary

- Lustre provides a comprehensive list of statistics (rpc stats, brw stat, extents stat, etc.) that can help demystify what is really happening on the IO side
- We can treat the applications as black boxes and use these built-in statistics to observe their behavior and also how the file system reacts to these IO requests
- We can use this methodology to profile both proprietary applications and open source software
- Our methodology uses all of Lustre built-in statistics and generic Linux tool - nothing related to proprietary software

Notices and Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit **www.intel.com/benchmarks**

© Intel Corporation. Intel, Intel Inside, the Intel logo, Xeon, Intel Xeon Phi, Intel Xeon Phi logos and Xeon logos are trademarks of Intel Corporation or its subsidiaries in the United States and/or other countries.


intel
Software