File I/O on Intel Xeon Phi coprocessors: RAM Disks, VirtIO, NFS and Lustre

Andrey Vladimirov, Vadim Karpusenko and Tony Yoo
Colfax International

July 28, 2014

Abstract

The key innovation brought about by Intel Xeon Phi coprocessors is the possibility to port most HPC applications to manycore computing accelerators without code modification. One of the reasons why this is possible is support for file input/output (I/O) directly from applications running on coprocessors. These facilities allow seamless usage of manycore accelerators in common HPC tasks such as application initialization from file data, saving running output, checkpointing and restarting, data post-processing and visualization, and other.

This paper provides information and benchmarks necessary to make the choice of the best file system for a given application from a number of the available options:

- RAM disks,
- virtualized local hard drives, and
- distributed storage shared with NFS or Lustre.

We report benchmarks of I/O performance and parallel scalability on Intel Xeon Phi coprocessors, strengths and limitations of each option. In addition, the paper presents system administration procedures necessary for using each file system on coprocessors, including bridged networking and InfiniBand configuration, software installation and MPSS image modifications. We also discuss the applicability of each storage option to common HPC tasks.

Table of Contents

1. File I/O on Intel Xeon Phi coprocessors

Intel Xeon Phi coprocessors are manycore computing accelerators for highly parallel applications. Unlike GPGPUs, coprocessors run a Linux operating system, which allows them to execute not only offload workloads, but also native applications. Native applications use coprocessors as additional compute nodes in a cluster with their own cores, memory, storage media and interconnects. This approach simplifies application porting from general-purpose CPUs to coprocessors, because the programmer does not need to instrument offload traffic in the code. This applies to shared-memory codes, as well as distributed and even heterogeneous applications (see [2]). Native applications running on Intel Xeon Phi coprocessors can use the same parallel frameworks as general-purpose applications, such as OpenMP and MPI (see, e.g., [3] or [4]) and communicate with other network nodes using Gigabit Ethernet or InfiniBand fabrics (see [5]).

Continuity of programming facilities between CPUs and coprocessors also applies to file I/O. Coprocessor applications can read and write files using Linux system calls. This gives native applications on coprocessors the same latitude for working with data as CPU applications.

There are three distinct ways to work with files from native Intel Xeon Phi coprocessor applications:

1) Direct access to a RAM disk,
2) Virtualized access to a physical drive,
3) Network access to a distributed file system.

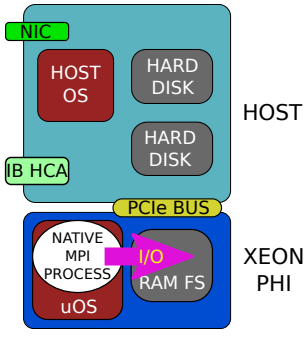

Figure 1: RAM disk file systems: `tmpfs` or `ramfs`.

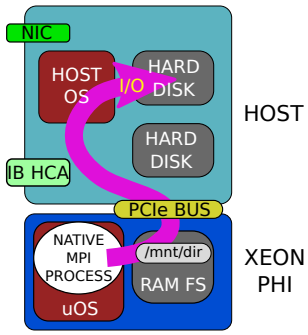

Figure 2: Virtualized access to a local physical drive: VirtIO.

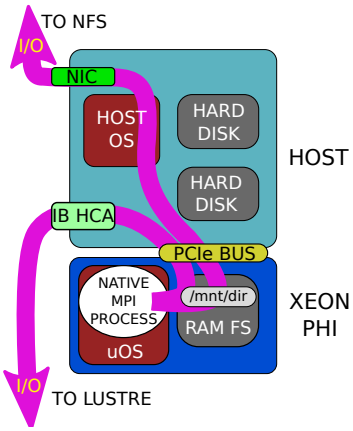

Figure 3: Distributed file system: NFS or Lustre.

In the first case (see Figure 1), the RAM disk is stored in the memory of the coprocessor. This storage device is not persistent across coprocessor reboots and is limited in size by the amount of available onboard RAM. However, this option is readily available when the coprocessor boots. The user can boost performance of the RAM disk by enabling MPSS optimizations or mounting a `ramfs` file system instead of the default `tmpfs`. Section 3 discusses this case in more detail.

In the second case (see Figure 2), the coprocessor reads and writes files physically stored on a data storage device such as a hard drive in the host system. The Linux device representing the drive is virtualized on the coprocessor using the VirtIO protocol. This is the same technology that is used to give control over a hard drive to a virtual machine. This option requires a bit more work to set up, and it does not allow sharing files between multiple coprocessors or systems. However, it offers persistent high-capacity storage.

Finally, the third option, distributed file system, comes in the form of two protocols: Network File Sharing (NFS) protocol [6] and Lustre distributed file system [7] (see Figure 3). To work with files in an NFS or Lustre file system from an Intel Xeon Phi coprocessor, some system configuration steps must be performed, most importantly, virtual network configuration in the coprocessor OS (Ethernet or InfiniBand). The result is persistent, high-capacity, distributed storage, in which files can be shared across a cluster.

In this paper we demonstrate the procedures for setting up each of the available file systems on an Intel Xeon Phi coprocessor. We also measure the performance of streaming read and write operations and study the parallel scalability of file I/O performance.

2. METHODOLOGY

2.1. SYSTEM CONFIGURATION

For RAM disk, VirtIO and NFS benchmarks, we used CentOS 6.5 Linux with kernel 2.6.32-431.el6.x86_64 on the host and MPSS 3.3 for the coprocessor [8]. For Lustre benchmarks, we used the Linux kernel 2.6.32-431.17.1.el6.x86_64 and MPSS 3.1.2 for compatibility with the Intel Enterprise Edition Lustre (IEEL) version 2.0.0 [9].

The computing system used for benchmarks is a Colfax SXP8600p Workstation with four Intel Xeon Phi 7120P coprocessors. Each of these coprocessors has 61 active cores and 16 GB of onboard RAM. The workstation is connected to the Lustre server with Mellanox InfiniBand ConnectX-3 Single-Port VPI adapters (model version MHQH19B-XTR at 4X QDR (40 Gb/s) via a 36-port Mellanox Infiniscale IV switch (model IS5025). Connection to the NFS server is via a NETGEAR JGS524 Gigabit Ethernet switch.

The hard drives used for testing are:

- For VirtIO and NFS: a software RAID0 array of two 2 GB Toshiba MG03ACA200 SATA hard disk drives (HDDs), and
- For Lustre: four 4 TB Toshiba MG03ACA400 HDDs in a single object storage server.

2.2. IOZONE CROSS-COMPILATION

All tests discussed below use the open source IOzone benchmark [10]. In order to cross-compile the benchmark for the Intel manycore architecture native mode, we modified the Makefile to use the Intel C compiler and included the argument `-mmic` into compiler and linker flags. The resulting executable was copied to the coprocessor and run from the terminal.

In all console listings, the host name of the server is shown as `lyra`, and the Linux username is shown as `vega`. Hostname `lyra-mic0` corresponds to the first coprocessor in the system. For convenience, many of the tests are run under the `root` account, because superuser access is required for some of the necessary operations, including dropping disk caches and unmounting/remounting file systems. These procedures are described in Section 2.4.

2.3. "WRITE" AND "WRITE+SYNC"

For testing the performance of the various file systems, we used operations representative of typical HPC tasks such as initializing an application, storing running output, checkpoint-restart, and post-processing or visualization jobs following a simulation. In most applications, data participating in these operations can be stored in the form of large contiguous arrays. For that reason, of most interest to us is the performance of sequential reads and writes.

The first test we perform for all file systems is the standard "write" test, which in IOzone is represented by the `write()` system call. This system call blocks until it is safe to use the write buffer in the user application. In our Figures 5 – 7 and in the summary table (Table 1), we call it the "write" test.

The "write" is an accurate measure of write performance in applications with low intensity of file I/O. That is because when the `write()` system call returns, the written data is usually placed into the OS disk cache, but not flushed to the storage medium or pushed across the network (see Figure 4). In low-intensity I/O, the OS may flush the data later, in the background, without slowing down the user application.

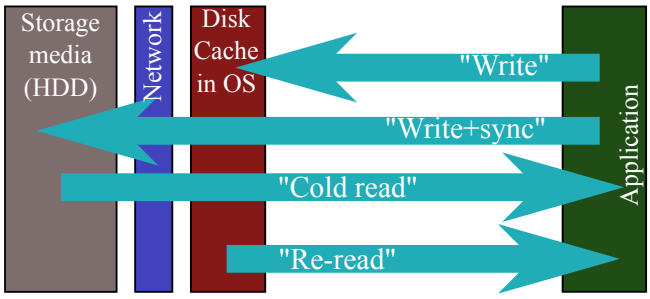

Figure 4: "Write", "write+sync", "cold read" and "re-read" tests performed on Intel Xeon Phi coprocessors in this work.

In order to simulate applications with high intensity and long duration of file I/O, we perform a different benchmark, which we call "write+sync". With intense I/O, the OS disk cache may eventually fill up. When that happens, the `write()` operation will block for the amount of time required to flush the data to the physical medium or send it across the network, which is much slower than working with the cache. For the "write+sync" test, after the `write()` call, IOzone makes the `sync()` call to ensure that the data are written to the medium or pushed across the network. The duration of `sync()` is included in timing. This is done by calling IOzone with the argument `-e`.

Listing 1 demonstrates the syntax with which we call IOzone for the "write" and "write+sync" tests. The result of the "write" test is taken for the column labelled "write" produced by the first call to IOzone. The performance of "write+sync" is also taken from the column "write", but in the second IOzone call with the argument `-e`. Column "rewrite" produced by IOzone is ignored in this paper.

```
vega@lyra-mic0% # Write test:
vega@lyra-mic0% ./iozone.MIC -i 0 -r 4M -s 1G
... (use "write" column):
              kB  reclen   write rewrite ...
           32768    4096  294795  334773
vega@lyra-mic0% # Write + sync test:
vega@lyra-mic0% ./iozone.MIC -i 0 -e -r 4M -s 1G
... (use "write" column)
              kB  reclen   write rewrite ...
           32768    4096   27972   27982
```

Listing 1: I/O tests "write", "write+sync".

2.4. "COLD READ" AND "RE-READ"

For benchmarking file reading performance, we also view this task from the HPC application point of view.

One situation is when the application reads input data which has never been transferred from the storage medium to the coprocessor RAM. To represent this case, we perform a test that we call "cold read". For the "cold read" test, prior to benchmarking the reading of an existing file, we use Linux functionality to clear disk caches and also unmount and re-mount the file system (unless we are using `tmpfs` or `ramfs`). This trick works for NFS and Lustre, however, for VirtIO these measures do not help to eliminate cache effects. For that reason, only for VirtIO benchmarks, we create files for reading on the host side, which provides a clean "cold read" measurement. The goal of the "cold read" test is to represent the situation where the file is read by the coprocessor OS for the first time since reboot, or when the amount of read data exceeds the amount of coprocessor memory available for disk cache.

```
root@lyra-mic0% # 1) Create the test file:
root@lyra-mic0% for i in {1..5}; do
> cat ~/pattern >> /nfs/file32M ; done
root@lyra-mic0% # 2) Unmount the file system:
root@lyra-mic0% umount /nfs
root@lyra-mic0% # 3) Drop caches on coprocessor:
root@lyra-mic0% echo 3 >/proc/sys/vm/drop_caches
root@lyra-mic0% # 4) and on storage server:
root@lyra-mic0% ssh nfs-server \
>            "echo 3 > /proc/sys/vm/drop_caches"
root@lyra-mic0% # 5) Mount the file system:
root@lyra-mic0% mount /nfs
root@lyra-mic0% # 6) Run the benchmarks:
root@lyra-mic0% ./iozone.MIC -i 1 -+E \
>                -r 4M -s 32M -f /nfs/file32M
... (use "read" column for "cold read" test
...  and "reread" column for "re-read" test)
     kB  reclen  write  rewrite   read    reread
  32768    4096                 24443   986690
```

Listing 2: Tests "cold read" and "re-read".

Another situation is, for example, restarting a computing application from checkpoint data, or reading the output of a simulation by a subsequent post-processing or visualization job. In these cases, the data has been in the coprocessor memory before, and there is a chance (though not a guarantee) that it may still be in the disk

cache. To represent this situation, we allow IOzone to conduct the second test, in which the file that had just been read is re-read. We call this test "re-read". The goal of this test is to represent the situation where the data read from a file system is present in the disk cache of the coprocessor OS.

Listing 2 shows the procedure that we use to obtain the "cold read" and the "re-read" performance. The file called `pattern` is a specially prepared file 1 MB in size filled with characters of hexadecimal value 0xF4. This is the pattern that IOzone expects to find in the read file. Note that the "cold read" result is taken from the "read" column in IOzone output, and the "re-read" result – from the "reread" column in the same run.

2.5. Parallel I/O Scalability

In HPC applications, it is common for multiple concurrent read or write operations to be performed on a compute node or a coprocessor. These multiple operations may come from different MPI processes each reading or writing their own chunk of data, or from multiple threads within one process parallelizing I/O for added performance. In this paper, we assume that the IOzone parallel I/O tests represent the situations of parallel I/O in real HPC applications. It is important to note that for RAM disks, different parallel frameworks (e.g., Pthreads, OpenMP, MPI) can lead to different file I/O performance because of the differences in multithreading overhead, process pinning, memory page sharing, and other aspects.

Listings 3 and 4 demonstrate our methodology for obtaining benchmarks of parallel I/O performance on Intel Xeon Phi coprocessors. Note that each reader or writer accesses a different file:

```
root@lyra-mic0% # write test with 2 threads:
root@lyra-mic0% ./iozone.MIC -i 0 -r 4M -s 32M \
>                            -e -T -t 2
... (use the "write" result for children)
root@lyra-mic0% # write+sync, 2 threads:
root@lyra-mic0% ./iozone.MIC -i 0 -r 4M -s 32M \
>                            -T -t 2
... (use the "write" result for children)
```

Listing 3: Parallel write tests (this example uses `t`=2 threads).

```
root@lyra-mic0% # Drop cold read  and re-read:
root@lyra-mic0% # 1) Create files for reading:
root@lyra-mic0% for i in {1..5}; do
> cat ~/pattern >> /nfs/file32M-1
> cat ~/pattern >> /nfs/file32M-2
> done
root@lyra-mic0% # 2) Unmount the file system:
root@lyra-mic0% umount /nfs
root@lyra-mic0% # 3) Drop caches on coprocessor:
root@lyra-mic0% echo 3 >/proc/sys/vm/drop_caches
root@lyra-mic0% # 4) and on storage server:
root@lyra-mic0% ssh nfs-server \
>            "echo 3 > /proc/sys/vm/drop_caches"
root@lyra-mic0% # 5) Mount the file system:
root@lyra-mic0% mount /nfs
root@lyra-mic0% # 6) run the benchmarks, 2 thr:
root@lyra-mic0% ./iozone.MIC -i 1 -+E -r 4M \
> -s 32M -T -t 2 -F file32M-1 file32M-2
... (use "read" and "re-read" for children)
```

Listing 4: Parallel read tests (this example uses `t`=2 threads).

With benchmark methodology established, in subsequent sections we configure and benchmark the above mentioned file systems available to Intel Xeon Phi coprocessors.

3. SYSTEM ADMINISTRATION PROCEDURES

This section describes the steps necessary to set up access to the `tmpfs`, `ramfs`, VirtIO, NFS and Lustre file systems from Intel Xeon Phi coprocessors.

3.1. USING RAM DISKS

3.1.1. CONFIGURATION OF TMPFS

Intel Xeon Phi coprocessor operation is controlled by a stack of drivers and utilities called Intel MPSS [8]. MPSS tools create a file system image containing the root directory of the coprocessor OS and use this image to boot the coprocessor. During the boot process, this image is unpacked into a `tmpfs` file system residing in the onboard memory of the coprocessor.

This means that nothing needs to be done to use a `tmpfs` RAM disk in a native coprocessor application. When a user application operates in the native mode, it can perform I/O in the user's home directory on the coprocessor, or other directories, as long as directory permissions allow the requested I/O operations. The size of this RAM disk is limited by the available amount of RAM. Listing 5 demonstrates querying and using `tmpfs` in the user's home directory.

```
vega@lyra% ssh mic0 # Log in to the coprocessor
vega@lyra-mic0% mount # View mounted filesystems
rootfs on / type rootfs (rw)
none on /proc type proc (rw,relatime)
none on / type tmpfs (rw,relatime,size=4996868k,
                                    mode=755)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
none on /dev type tmpfs (rw,relatime,mode=755)
devpts on /dev/pts type devpts (rw,relatime,
                                gid=5,mode=620)
none on /var/volatile/tmp/COI2MB type hugetlbfs
                                  (rw,relatime)

vegar@lyra-mic0% df -h # Query available space
Filesystem  Size   Used  Availbl Use% Mounted on
none       12.9G 162.0M    12.7G   1% /
none        7.6G  44.0K     7.6G   0% /dev

vega@lyra-mic0% # Create a file and check on it:
vega@lyra-mic0% touch /home/vega/foo
vega@lyra-mic0% ls -l /home/vega
-rw-r--r--     1 vega  vega  0 Jul 21 16:13 foo
```

Listing 5: Using `tmpfs` on an Intel Xeon Phi coprocessor.

3.1.2. CONFIGURATION OF RAMFS

Administrator of the coprocessor OS can also mount a RAM disk with the `ramfs` file system. This file system does not perform checks for exceeding the RAM disk storage capacity. The `ramfs` disk grows as more data is written into it. Because of this difference, `ramfs` may perform faster than `tmpfs`. However, the cost of the added performance is that with `ramfs` the coprocessor may crash if it runs out of memory. Listing 6 shows how to mount and use `ramfs` in an Intel Xeon Phi coprocessor.

```
root@lyra% ssh mic0 # Log in to the coprocessor
root@lyra-mic0% mkdir /scratch # Mount point
root@lyra-mic0% mount -t ramfs ramfs /scratch
root@lyra-mic0% mount | grep scratch
ramfs on /scratch type ramfs (rw,relatime)
root@lyra-mic0% chmod 777 /scratch # Open access
root@lyra-mic0% su vega # Try it
vega@lyra-mic0% touch /scratch/bar
```

Listing 6: Mounting `ramfs` on a coprocessor.

3.1.3. MPSS OPTIMIZATIONS

MPSS supports experimental optimizations of RAM disk filesystems described in [11]. As of MPSS 3.3, these optimizations must be manually enabled by including additional parameters to `ExtraCommandLine` in `/etc/mpss/default.conf` as shown in Listing 7. After that, the administrator must stop MPSS, run `micctrl --resetconfig` and start MPSS again.

```
root@lyra% cat /etc/mpss/default.conf
...
# Additional command line parameters.
ExtraCommandLine "highres=off \
   vfs_write_optimization=on \
    vfs_read_optimization=on"
...
```

Listing 7: Enabling additional MPSS optimizations (necessary for MPSS 3.3; put all arguments in one line)

3.2. Access to host drives with VirtIO

The VirtIO block device support in MPSS allows the virtualization of physical storage media such as hard drives in the Intel Xeon Phi coprocessor OS.

3.2.1. Preparation of a logical volume

A reliable way to designate how much space on the physical drive will be given to the coprocessor is to create an LVM logical volume on the drive. Listing 8 demonstrates the process of configuring a physical volume on the RAID device `/dev/md0`, creating a volume group `vg_phi` containing that physical volume, and partitioning it with a logical volume `lvol0` that occupies 30 GB.

```
root@lyra% pvcreate /dev/md0
Physical volume "/dev/md0" successfully created
root@lyra% vgcreate vg_phi /dev/md0
Volume group "vg_phi" successfully created
root@lyra% lvcreate vg_phi -L 30GB
  Logical volume "lvol0" created
```

Listing 8: Creating a logical volume for VirtIO.

3.2.2. Using a logical volume in VirtIO

The next step is instructing MPSS to enable VirtIO on that logical volume. This is done by echoing the path to the logical volume into the file `/sys/class/mic/mic0/virtblk_file`, as seen in Listing 9. The path contains `mic0`, which in this procedure indicates that VirtIO is configured for the first coprocessor. After that, MPSS must be restarted.

```
root@lyra% echo "/dev/mapper/vg_phi-lvol0" >\
>               /sys/class/mic/mic0/virtblk_file
root@lyra% service mpss restart
root@lyra% ssh mic0
root@lyra-mic0% mkfs.ext2 /dev/vda # Format
root@lyra-mic0% mkdir /virtio # Create mnt point
root@lyra-mic0% mount /dev/vda /virtio # Mount
root@lyra-mic0% df -h # Check result
Filesystem  Size   Used  Availbl Use% Mounted on
none       12.9G  56.1M    12.8G   0% /
none        7.6G  48.0K     7.6G   0% /dev
/dev/vda   46.7G  59.4M   139.2G   0% /virtio
...
```

Listing 9: Giving control of the logical volume to `mic0`.

When MPSS boots, the user will find a new device on `mic0` at `/dev/vda`. If the logical volume is already formatted, then the coprocessor OS will automatically mount this drive at `/media/vda`. Otherwise, `/dev/vda` may be formatted from the coprocessor using `mkfs.ext2` and thereafter mounted at any mount point. We choose `/virtio` for the mount point location. When that is done, user applications can write into `/virtio` as long as Linux permissions allow that. The files written into that directory will be physically stored on the RAID array `/dev/md0`, which is installed in the host system.

Some peculiarities of VirtIO that we observed during testing are:

a) Once shared `/dev/mapper/vg_phi-lvol0` is virtualized as `/dev/vda` on the coprocessor, the coprocessor OS owns this device. That means that the files written to it cannot be viewed on the host or shared with other coprocessors until the drive is unmounted on the coprocessor and re-mounted on the host.

b) The host OS refuses to change the contents of `/sys/class/mic/mic0/virtblk_file` if MPSS is loaded. In order to change it, the administrator must run on the host the command "`service mpss unload`" followed by "`modprobe mic`".

c) The VirtIO block device has an upper size limit of 2 TB, as seen by the tool `fdisk` on the coprocessor. That is, for logical volumes smaller than 2 TB, the VirtIO device on the coprocessor has the same size as the logical volume. However, for logical volumes greater than 2 TB, the coprocessor sees only the fraction of the logical volume size above 2 TB or above a multiple of 2 TB. For example, a 2.5 TB logical volume on host appears as a 0.5 TB device on the coprocessor.

d) Instead of a logical volume, it is also possible to export a file in the host file system. We have not tested the performance or limitations of this method.

3.3. NFS OVER GIGABIT ETHERNET

NFS distributed storage is a standard tool in Linux systems [6]. It allows to share a mount point hosted by the NFS server with a number of clients communicating with the server over TCP/IP.

3.3.1. NETWORK CONFIGURATION

In order to NFS-share a directory with Intel Xeon Phi coprocessors, networking on coprocessors must be configured so that TCP packets are able to travel between the NFS server and the coprocessors. If the NFS server is the OS hosting the coprocessors, then default network configuration of MPSS is sufficient. This configuration has the static pair topology, in which coprocessors form a private network within the host.

However, if the NFS server is a remote machine, then network configuration on coprocessors must have the external bridge topology. This allows coprocessors to direct TCP/IP packets to the private network of the host via the host's Ethernet adapter. Thus, the coprocessor can communicate with other servers and coprocessors in other compute nodes on the network. Listing 10 shows an example of bridged networking configuration on a system with two Intel Xeon Phi coprocessors. More details on bridged networking are given in our publication [5].

```
root@lyra% service mpss stop
root@lyra% micctrl --addbridge=br0 \
>                  --type=external --ip=10.33.1.2
root@lyra% micctrl --network=static \
>         --bridge=br0 --ip=10.33.1.22:10.33.1.42
root@lyra% service mpss start
root@lyra% cat /etc/hosts | grep "nfs\|mic"
10.33.1.1  nfs-server
10.33.1.22 lyra-mic0 mic0 #Generated-by-micctrl
10.33.1.42 lyra-mic1 mic1 #Generated-by-micctrl
root@lyra% ssh mic0 # Log in to coprocessor
root@lyra-mic0% ping 10.33.1.1 # Check
... (ping must succeed) ...
```

Listing 10: Configuration of bridged networking on coprocessors.

3.3.2. CLIENT CONFIGURATION

Listing 11 demonstrates how NFS client software can be configured on an Intel Xeon Phi coprocessor. For manual mounting, it is sufficient to run the command "`mount -t nfs`" with the arguments pointing to the storage server and to the local mount point. This mount will be gone if the coprocessor is rebooted. In order to make the mount persistent, `micctrl` may be used on host, as shown in the continuation of Listing 11. Note that by default, `micctrl` will add the NFS mount to all coprocessors installed in the host.

```
root@lyra-mic0% # First, mount manually:
root@lyra-mic0% mkdir /nfs # Create mount point
root@lyra-mic0% mount -t nfs \
>                     10.33.1.1:/nfs-exp /nfs
root@lyra-mic0% df -h | grep -v none
Filesystem     Size  Used Availb Use% Mounted on
10.33.1.2:/nfs 1.8T 11.3G   1.7T   1% /nfs
root@lyra-mic0% exit
root@lyra% # Now configure automatic mounting:
root@lyra% service mpss stop
root@lyra% micctrl --addnfs=10.33.1.1:/nfs-exp \
>                  --dir=/nfs
root@lyra% service mpss start
root@lyra% ssh mic0 # Log in to coprocessor
root@lyra-mic0% df -h | grep -v none
Filesystem     Size  Used Availb Use% Mounted on
10.33.1.2:/nfs 1.8T 11.3G   1.7T   1% /nfs
```

Listing 11: Mounting an NFS share on a coprocessor.

For additional options of NFS mounting on coprocessors, run `micctrl --addnfs --help`.

Note: with IPoIB configured (see Section 3.4), one can mount an NFS share over the InfiniBand network. However, we have found in our testing that, as of MPSS 3.3, single-threaded performance on coprocessors of NFS over InfiniBand is not different from performance over Ethernet. Because of this, we do not report results of NFS over InfiniBand here.

3.4. LUSTRE OVER INFINIBAND

Lustre is a scalable distributed file system used for large-scale cluster computing and data storage. Intel Xeon Phi coprocessors support mounting Lustre shares over the InfiniBand fabric.

All machines in the Lustre solution and the benchmark compute node are interconnected with Mellanox InfiniBand links (see Section 2.1). The setup of Lustre server is fairly complicated, requiring multiple machines with different functions. Information on Intel Enterprise Edition Lustre server configuration can be found in [9]; also, Colfax offers turn-key Lustre solutions [12]. In Section 3.4, only client-side configuration steps for mounting a Lustre share on an Intel Xeon Phi coprocessor are discussed.

3.4.1. IPOIB CONFIGURATION

Prior to mounting Lustre, the administrator must configure IPoIB on the host as well as on the coprocessor. Listing 12 demonstrates that procedure.

```
root@lyra% # Configure IPoIB on host
root@lyra% cat \
>       /etc/sysconfig/network-scripts/ifcfg-ib0
DEVICE=ib0
HWADDR=80:00:00:48:FE:80:00:00:00:00:00:00:00:00...
TYPE=InfiniBand
UUID=c8ab4dab-4672-4044-93f2-94d8423dd6de
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=static
IPADDR=10.34.1.2
NETMASK=255.255.0.0
root@lyra% # Configure IPoIB on coprocessors
root@lyra% cat /etc/mpss/ipoib.conf
... (comments omitted) ...
ipoib_enabled=yes
mic0_ib0="10.34.1.22 netmask 255.255.0.0"
mic1_ib0="10.34.1.42 netmask 255.255.0.0"
root@lyra% service mpss stop
root@lyra% # Propagate configuration to MPSS
root@lyra% micctrl --resetconfig
root@lyra% service mpss start
root@lyra% service ofed-mic restart
root@lyra% ssh mic0 # Go to the coprocessor and
root@lyra-mic0% # ping Lustre metadata server:
root@lyra-mic0% ping 10.34.0.2
... (ping must succeed) ...
```

Listing 12: Configuring IPoIB.

3.4.2. CLIENT CONFIGURATION

To install Lustre client for Intel Xeon Phi coprocessors, we chose to use RPMs provided in IEEL 2.0.0. These RPMs are compiled for MPSS 3.1.2 in CentOS Linux with kernel 2.6.32-431.17.1.el6.x86_64 and OFED 1.5.4.1. So we had to install these specific kernel and MPSS versions. Importantly, MPSS and OFED modules for this kernel version are not included in the MPSS 3.1.2 distribution, so we had to recompile these kernel modules. Instructions for these procedures are given in MPSS User's Guide [8].

After installing MPSS and OFED, the administrator may install the Lustre client for Xeon Phi RPMs as shown in Listing 13.

```
root@lyra% tar -xf ieel-2.0.0.tar.gz
root@lyra% cd ieel-2.0.0/
root@lyra% tar -xf \
>         xeon-phi-client-2.5.19-bundle.tar.gz
root@lyra% name="lustre-client-mic"
root@lyra% vers="2.5.19"
root@lyra% mpss="2.6.38.8+mpss3.1.2"
root@lyra% rpm -ivh --nodeps \
> ${name}-${vers}-${mpss}.x86_64.rpm \
> ${name}-modules-${vers}-${mpss}.x86_64.rpm
```

Listing 13: Installing Lustre client for Intel Xeon Phi coprocessors from IEEL 2.0.0 RPM packages.

Finally, to mount a Lustre file system, steps shown in Listing 14 must be performed on the coprocessor.

```
root@lyra-mic0% echo \
> 'options lnet networks=o2ib0(ib0)' > \
> /etc/modprobe.d/lustre.conf
root@lyra-mic0% mkdir /lustre # mount point
root@lyra-mic0% mount.lustre \
>        10.34.0.2@o2ib0:/lustre /lustre
root@lyra-mic0% df -h | grep -v none
Filesystem             Size  Used ...Mounted on
10.34.0.2@o2ib:/lustre 14.5T 42.8G.../lustre
```

Listing 14: Mounting a Lustre share on a coprocessor.

The administrator may make the file `lustre.conf` and the mount point persistent across coprocessor reboots. This is done by adding these files and directories to the MIC root directory image located at `/var/mpss/mic*` on the host. Instructions for that are given in MPSS User's Guide [8].

4. BENCHMARK RESULTS

4.1. PHYSICAL MEDIA BENCHMARKS

Before proceeding with benchmarks of file systems on Intel Xeon Phi coprocessors, it is informative to study the storage media that we use for backing these file systems. All disk-based storage options: VirtIO, NFS and Lustre use the Toshiba MG03ACAxxx family HDDs. In this section, we measure the performance of these drives on the host with our "write+sync" and "cold read" tests.

The methodology for this testing is described in Sections 2.3 and 2.4. We benchmark three configurations: a stand-alone MG03ACA200 drive, a RAID0 array of two drives, and a RAID0 array of four drives. The drives and RAID arrays are formatted with the ext4 file system. For benchmarks, we use a file size of 1 GB (large enough so that it exceeds the combined 64 MB data buffers of the drives) and a record size of 4 MB.

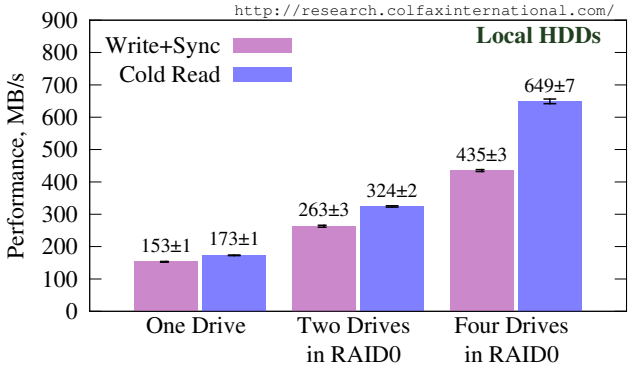

Figure 5: Benchmarks of the storage media on the host.

Results are shown in Figure 5. They set the upper bound on the performance of file systems mounted on Intel Xeon Phi coprocessors.

4.2. SINGLE-THREADED I/O BENCHMARKS

We conducted single-threaded "write+sync", "cold read", "write" and "re-read" benchmarks on the filesystems discussed above. The results are shown in Figure 6 with bars labelled "1 thread".

Horizontal dashed lines in Figure 6 are the results of the storage media tests also shown in Figure 5 and the theoretical maximum throughput of Gigabit Ethernet. These lines are added for convenience so that the reader may judge whether the measured I/O speed is limited by the disk cache, storage media or network bandwidth.

All measurements in Figure 6 are for a file size of 32 MB and a record size of 4 MB. We do not report results with other file sizes in these plots, because we have found the benchmarks to be only weakly dependent on the file size. Interested reader may refer to Appendix 5 for a study of file sizes in the range from 4 MB to 1 GB.

4.3. PARALLEL I/O BENCHMARKS

We also performed multi-threaded "write+sync", "cold read", "write" and "re-read" benchmarks on the filesystems discussed above. The parallel speedup of performance of these tests, as a function of the number of threads, is shown in Figure 7. The performance with the best speedup is shown in Figure 6 with bars labelled "N threads", where N is the optimal number of threads.

All measurements of parallel I/O performance are for file size of 32 MB, except for `tmpfs` and `ramfs`, where we used 128 MB files. That is because the overhead of multi-threading in IOzone reduces reported performance if the amount of I/O per thread is too low.

Table 1 contains a summary of the measurements. It also contains miscellaneous information on the usage models and limitations of the tested file systems.

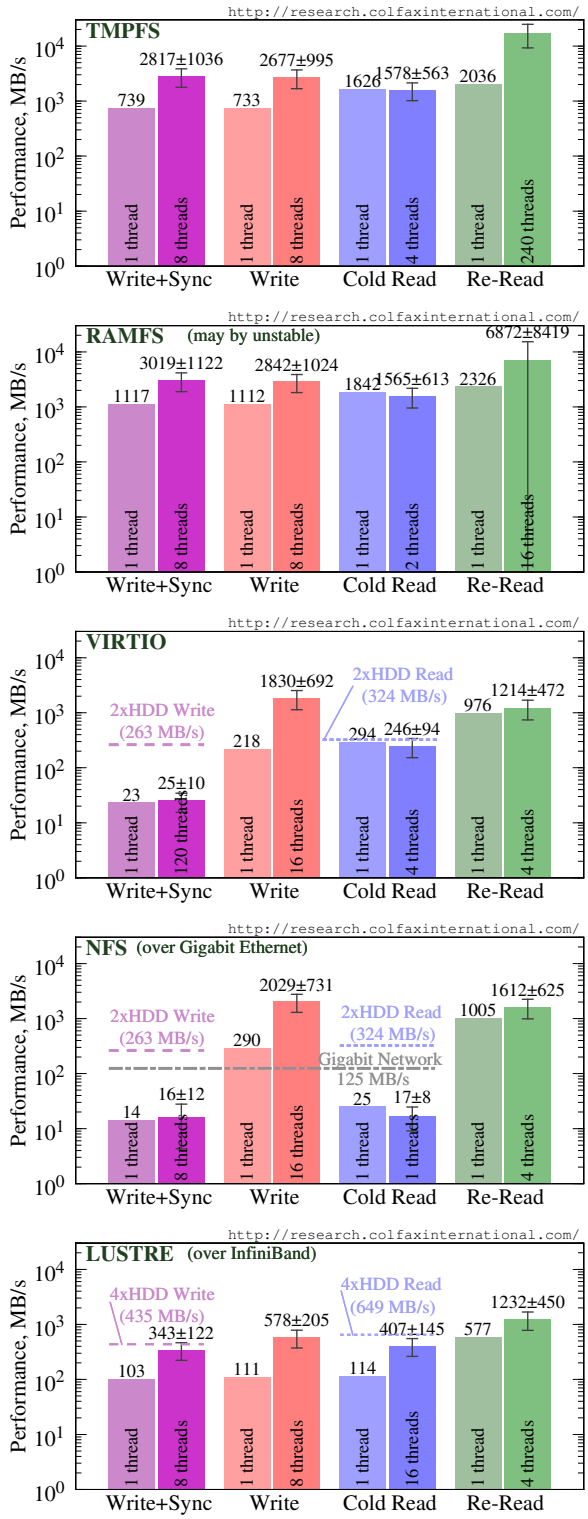

Figure 6: I/O performance in `tmpfs`, `ramfs`, VirtIO, NFS and Lustre filesystems on an Intel Xeon Phi coprocessor. For each test, single-threaded and best multi-threaded performance are shown. Horizontal dashed lines indicate the limits imposed by the media or network throughput.

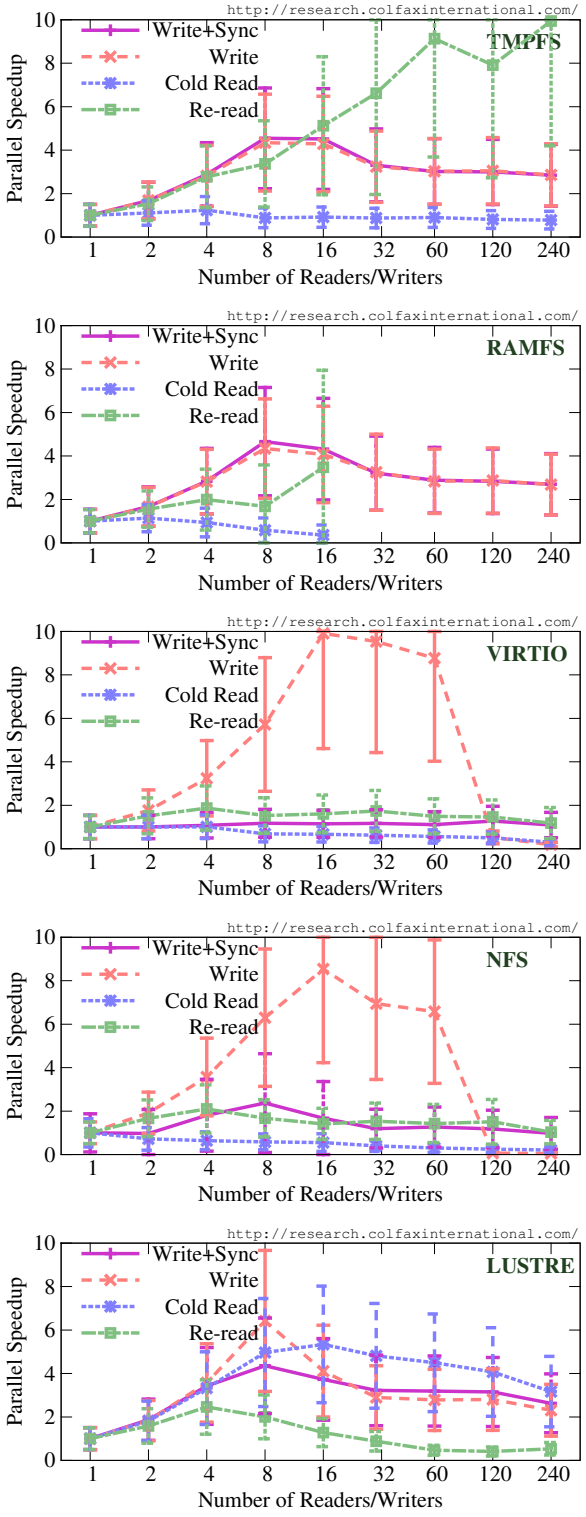

Figure 7: Parallel scalability of I/O performance in `tmpfs`, `ramfs`, VirtIO, NFS and Lustre filesystems on an Intel Xeon Phi coprocessor. Speedup is the ratio of multithreaded performance to single-threaded performance for the same test on the same file system.

Property	`tmpfs`	`ramfs`	VirtIO	NFS	Lustre
Configuration Difficulty	none	one line	minutes	minutes	1 hour[×]
Requires `root` access	no	yes	yes	yes	yes
Data persistence across reboots	no	no	yes[⋆]	yes	yes
File sharing across cluster	no	no	no	yes	yes
Can be used as swap space	no	no	yes	no	no
"Write+sync", MB/s	700	1100	23	14	100
Parallel "write+sync", MB/s	3000±1000	3000	25	16±12	350±100
"Write", MB/s	700	1100	220	300	
Parallel "write", MB/s	3000±1000	3000	2000	2000±700	600±200
"Cold read", MB/s	1600	1800	300	25	
Parallel "cold read", MB/s	1600±600	1600±600	250±100	17±8	400±150
"Re-read", MB/s	2000	2300	900	1000	
Parallel "Re-read", MB/s	≈20000	unstable	1200±500	1600±600	1200±450
Scales across multiple threads	yes	unstable	no/yes[#]	no/yes[#]	yes
Max file system size	13.8 GB[+]	13.8 GB[+]	2 TB[†]	Unknown[*]	Unknown[‡]

Table 1: Summary of file systems available to native applications on Intel Xeon Phi coprocessors.

[×] Lustre client setup takes up to 1 hour, which includes MPSS and OFED installation, networking configuration and Lustre client RPM installation. Lustre server-side configuration depends on the scale of the system and may take considerably longer.

[⋆] We noticed that the write performance of the partition shared with VirtIO degrades after MPSS restart, and is restored after re-formatting the partition.

[#] In VirtIO and NFS, only cache-backed write operation can be accelerated by multi-threading. All other operations (cold and warm read and write operation that has to be synced with the media or network) gain little or no additional performance from parallel I/O.

[+] Measured on a coprocessor with 16 GB onboard memory, without any running user applications. This limit is determined by the amount of installed onboard GDDR5 memory, which is dependent upon the coprocessor SKU.

[†] Empirically determined value (see Section 3.2).

[*] We did not detect a limit on Intel Xeon Phi coprocessors for NFS share sizes in our tests. The practical limit on the server side is determined by NFS limitations.

[‡] We did not detect a limit on Intel Xeon Phi coprocessors for Lustre share sizes in our tests. Reportedly, the maximum theoretical Lustre file system size is 64 PB, and production deployments come close to that value.

5. DISCUSSION

We measured the performance of RAM disks (`tmpfs` and `ramfs` filesystems), VirtIO block device, NFS and Lustre shares on an Intel Xeon Phi coprocessors. We provide results of single-threaded and multi-threaded (parallel) sequential file writing and reading for two situations:

a) Disk cache in the coprocessor OS serves the I/O operations. This is the situation measured by the "write" and "re-read" tests. These results pertain for low-intensity I/O ("write") or accesses to previously read files ("re-read").

b) The data are served by the physical storage media or the network. This is the situations measured by the "write+sync" and "cold read" tests. These results simulate to high-intensity, long-duration I/O ("write+sync") or access to previously not read data or large data sets ("cold read").

Below, we discuss the results and their implications for the different file systems in detail.

RAM DISKS

As expected, RAM disks with the `tmpfs` and `ramfs` file systems are the fastest storage option for Intel Xeon Phi applications. Single-threaded "write" test in `tmpfs` achieves 700 MB/s, and there is no difference between "write" and "write+sync", because this file system is not cached. Single-threaded "read" and "re-read" achieve 1600 and 2000 MB/s, respectively.

Multi-threading can further improve performance: parallel "write" can be accelerated by a factor of 4x and "re-read" by 10x; however, "cold read" does not scale with the number of threads.

For all but the least arithmetically intensive workloads, RAM disks provide performance so high that file I/O is likely to take a negligible amount of time. The disadvantage of these file systems is that they are limited in size by the amount of onboard memory in the coprocessor, and files stored in them reduce the amount of RAM available to computing applications.

Regarding the use of `ramfs`, which provides marginal speedup over `tmpfs`, we need to report that multi-threaded tests in `ramfs` consistently crashed for large numbers of threads. For that reason, we mention `ramfs` as "unstable" in Table 1.

VIRTIO

VirtIO allows the coprocessor to read and write files from/to a partition on a hard drive in the host system.

For low-intensity I/O backed by the OS disk cache, performance of VirtIO is high and scalable, with "write", "re-read" achieving 220 and 355 MB/s, respectively, and scaling up to ≈2000 and 1200 MB/s, respectively, with multi-threading.

However, in situations operations calling for actual data transfer to/from the storage media ("write+sync" and "cold read"), there is a striking difference between writing and reading. "Cold read" performance at ≈300 MB/s comes close to the read speed of the physical media, at least for the two-drive RAID0 that we tested. At the same time, "write+sync" is very slow, reaching just over 20 MB/s and not scaling with the number of threads.

Based on our testing, VirtIO is a great way to give the coprocessor access to read-only file data, such as large (up to 2 TB) static data sets. It can also be used for storing large libraries, which would otherwise take up valuable memory if they are stored on a RAM disk. For output, VirtIO provides reasonable performance only if write speed is absorbed by disk caches (i.e., in low-intensity output situations). However, dumping large amounts of output data of a coprocessor application into a VirtIO device with intensity over 20 MB/s is likely to slow the application down. Large in this context means greater than the amount of coprocessor RAM available for disk cache.

VirtIO is the only file system that can be used as a Linux swap partition. While swap space is rarely needed in HPC applications, it may be helpful in some debugging tasks.

NFS

The performance of NFS in the coprocessor OS is very similar to the performance of VirtIO, with the exception that "cold read" in NFS is also very slow at

≈20 MB/s. Therefore, for the purpose of distributing large read-only data sets, NFS is inferior to VirtIO.

Parallelism in I/O with NFS can be beneficial, but only in cases handled by the disk cache (read-only small data sets or low-intensity write operations).

However, NFS has a very significant functional difference from VirtIO: it allows to share a directory between multiple coprocessors and/or hosts. This makes NFS a convenient solution when it comes to access to shared data, or when file I/O performance is not significant. That includes producing a parallel application log file, sharing configuration files and small libraries across the cluster, etc.

LUSTRE

According to our test results, a Lustre file system shared over InfiniBand can provide single-threaded read and write performance of order 100 MB/s, and multi-threaded I/O can come close to saturating the speed of the 4-drive parallel array (around 350 MB/s for "write+sync" and 400 MB/s for "cold read"). These benchmarks demonstrate the versatility of Lustre:

- Write speeds that Lustre yields are orders of magnitude better than in VirtIO and NFS.
- Lustre offers the file system sharing functionality of NFS, but a much better "cold read" speed.
- Unlike VirtIO, Lustre does not have the 2 TB size limitation. Lustre is highly scalable in terms of size.
- Parallelism in I/O is beneficial for all operations in the Lustre file system.

It is worth mentioning that parallel I/O in Lustre on Xeon Phi has a "sweet spot" between 8 and 16 threads. Greater numbers of threads are counter-productive to I/O performance.

The practical disadvantage of Lustre as a storage solution for clusters with Intel Xeon Phi coprocessors is relative difficulty in Lustre server system configuration compared to VirtIO and NFS. This can difficulty be alleviated with OEM-provided solutions such as [12].

This work did not pursue the goal of determining the maximum performance of Lustre on an Intel Xeon Phi coprocessor. Lustre performance may vary depending on the configuration of the storage server, fabrics and software (see, e.g., [13]). However, what we demonstrated here is that in today's state of the art, Lustre scales far beyond the 20 MB/s limit experienced by VirtIO and NFS.

REFERENCES

[1] Landing page for this paper "File I/O on Intel Xeon Phi Coprocessors...". http://research.colfaxinternational.com/post/2014/07/28/MIC-IO.aspx.

[2] Hetetogeneous Clustering with Homogeneous Code. http://research.colfaxinternational.com/post/2013/10/17/Heterogeneous-Clustering.aspx.

[3] Primer on Computing with Intel Xeon Phi Coprocessors. Slides from a presentation, with links to additional resources. http://research.colfaxinternational.com/post/2014/03/06/Geant4-Tutorial.aspx.

[4] Colfax International. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. ISBN: 978-0-9885234-1-8. Colfax International, 2013. http://www.colfax-intl.com/xeonphi/book.html.

[5] Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors. http://research.colfaxinternational.com/post/2014/03/11/InfiniBand-for-MIC.aspx.

[6] Linux NFS Overview, FAQ and HOWTO Documents. http://nfs.sourceforge.net/.

[7] Lustre (file system). http://en.wikipedia.org/wiki/Lustre_(file_system).

[8] Intel Manycore Platform Software Stack (MPSS). http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[9] Intel Solutions for Lustre software. http://www.intel.com/content/www/us/en/software/intel-enterprise-edition-for-lustre-software.html.

[10] IOzone Filesystem Benchmark. http://www.iozone.org/.

[11] Ravi Murty and Rajesh Sudarsan. Improving File IO performance on Intel Xeon Phi Coprocessors. https://software.intel.com/en-us/blogs/2014/01/07/improving-file-io-performance-on-intel-xeon-phi.

[12] Intel Enterprise Edition for Lustre: information and solutions on the Colfax International Web site. http://www.colfax-intl.com/nd/solutions/intel-enterprise-edition-for-lustre.aspx.

[13] Dmitry Eremin, Zhiqi Tao, and Gabriele Paciucci. Running Native Lustre Client inside Xeon Phi. http://cdn.opensfs.org/wp-content/uploads/2014/04/D2_S17_RunningNativeLustreClientInsideXeonPhi.pdf.

APPENDIX: ADDITIONAL DATA

NOTE ON STORAGE MEDIA DIFFERENCES

Even though in our tests, VirtIO and NFS RAID0 arrays of two drives, and Lustre used four drives, it does not bias our results. This is because, according to our benchmarks, I/O performance in VirtIO and NFS is bottlenecked by the throughput of the software implementing them, rather than by the read and write speeds of the storage media. Namely, NFS "cold read" and "write+sync" and VirtIO "write+sync" are all limited at ≈20 MB/s regardless of the physical media, and VirtIO "cold read" saturates at ≈350 MB/s when the read speed of the media exceeds 500 MB/s. We confirmed that by running benchmarks on a single HDD, and a two-drive, three-drive and a four-drive RAID0 array. Results are shown in Figure 8.

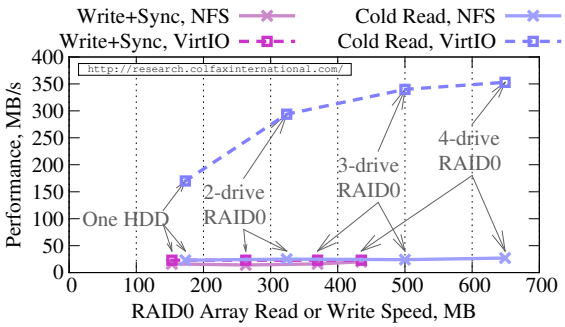

Figure 8: Performance of NFS and VirtIO with different media.

Furthermore, using a two-drive RAID0 array for NFS and VirtIO and four drives in parallel on Lustre simulates a realistic situation. Indeed, in practice, one would rarely configure a RAID0 in a compute node or an NFS server on as many as four drives. Similarly, a complex, scalable file system such as Lustre is unlikely to use as few as two drives in a practical solution.

ADDITIONAL PERFORMANCE PLOTS

In Figure 9 we display the results of single-threaded tests on all five file systems that report the dependence of the I/O speed on the file size. Record size is 4 MB in all tests.

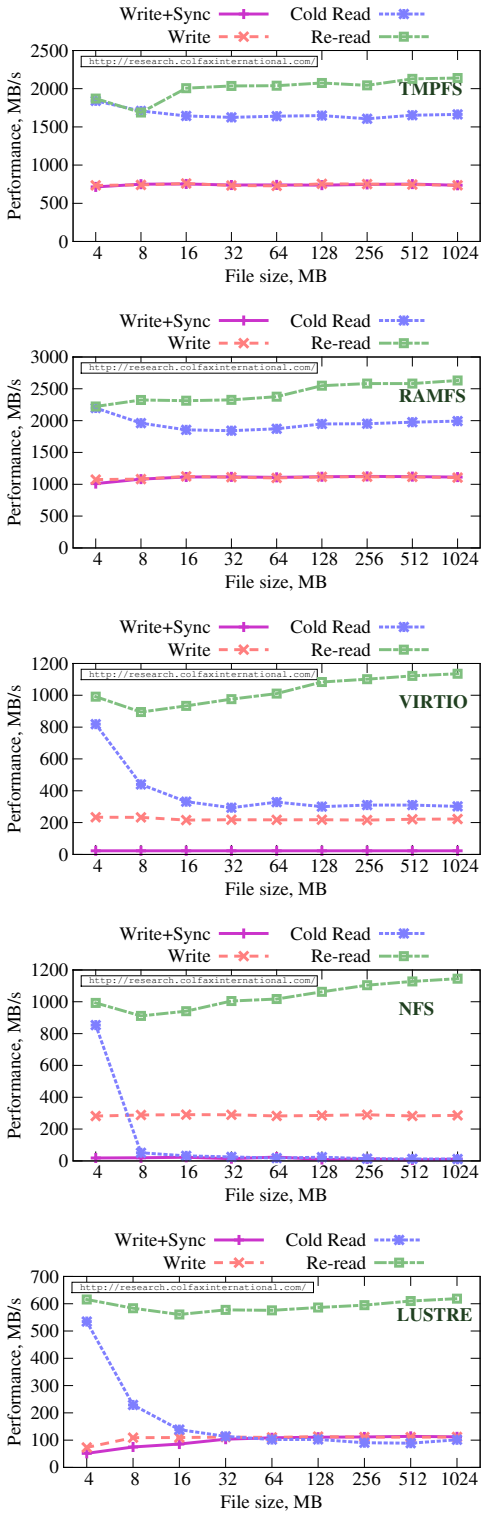

Figure 9: Single-threaded performance of file systems on coprocessor as a function of file size.