VSL™

ioMemory VSL

PEAK PERFORMANCE GUIDE

AUGUST 20, 2013

FUSION-IO®

Table of Contents

Table of Contents **2**

ioMemory VSL Peak Performance Guide **4**

Legal Notices 4

Introduction **5**

Performance Considerations 5

General Performance Optimization Settings **6**

Disable CPU Frequency Scaling 6

Set CPU Frequency to Highest Level 6

Disable CPU Power Throttling 6

Set Fan Speeds to High 7

Disable Hyper Threading 7

Use Asynchronous I/O 7

Use Direct I/O 7

Setting NUMA Affinity 8

Set the Interrupt Handler Affinity 8

System Configuration **10**

Hardware 10

BIOS 10

Software 10

Testing **12**

Appendix A - Sample Linux fio Job Files **14**

fio-job-01 - Random 512B Write Test for Latency 15

fio-job-02 - Random 512B Write Test for Peak IOPS 16

fio-job-03 - Sequential 512B Write Test for Peak IOPS 17

fio-job-04 - Random 1M Write Test for Peak Bandwidth 18

fio-job-05 - (PRECONDITION) Sequential 1M Complete Write Test 19

fio-job-06 - Random 512B Read Test for Latency 20

fio-job-07 - Random 512B Read Test for Peak IOPS 21

fio-job-08 - Sequential 512B Read Test for Peak IOPS 22

fio-job-09 - Random 1M Read Test for Peak Bandwidth 23

Appendix B - Programming for Direct I/O 24

Direct I/O on Linux 24

Direct I/O on Windows 25

Appendix C - Testing and Tuning Strategies 30

Precondition the Device 30

Writing Tests on a Full Device 30

Disk Aggregation (RAID) Considerations 31

Partition Alignment 32

Linux-specific Tuning 32

Appendix D - Debugging Performance Issues 36

Over-Subscribed Bus 36

Handling PCI Express Errors 37

PCIe Link Width Improperly Negotiated 38

Using cp and Other System Utilities 39

Additional Information and Support 40

Legal Notices

Fusion-io
2855 E. Cottonwood Parkway, Box 100
Salt Lake City, UT 84121
USA

(801) 424-5500

Part Number: D0005087-000_2
Published: August 20, 2013

Introduction

Data sheets are available at http://www.fusionio.com for each ioMemory product (including ioDrive2 and ioDrive2 Duo devices). The data includes performance attributes for each device, such as read and write bandwidth, IOPS (Input/output Operations Per Second), and latency.

This document describes the server settings and configurations used to produce those performance data.

Performance Considerations

These settings and configurations produced optimal results for this particular system and specific tests. Your system and applications may require different or additional tuning to produce the best results for your particular use case.

Performance tuning is often a trade off; over-optimizing for one performance attribute may have negative affects on another. **You should not try to achieve peak performance for all raw block I/O patterns**. You should tune performance for your particular use case and application.

Each attribute (for example read latency, write IOPS, or read bandwidth) has a specific test that is tuned to determine the peak performance for just that attribute. For example, the maximum write IOPS is achieved by testing 512B writes. However, to achieve the highest bandwidth possible, 1M blocks of data are used.

For example, to achieve peak IOPS performance, an ioMemory device must be used by an application that is accessing small chunks (512B) of data. While that specific configuration may achieve peak IOPS on a device, that I/O pattern may not achieve peak bandwidth on the device. Meanwhile, the same device may achieve peak bandwidth when it is used by a different application that is reading large chunks of data (such as video editing or streaming).

You should test and tune your system for the performance attribute or attributes that *provide the greatest impact for your use case*. Otherwise you may limit application performance by focusing on improving attributes that don't impact your application.

Data Destructive Tests
Write tests (including pre-conditioning) will write data to the device. These tests will write over all the data on the device. Only perform write tests on new or unused devices.

General Performance Optimization Settings

Obtaining peak performance requires some system-level tuning. The following are performance tuning actions that we recommend.

Disable CPU Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) are power management techniques that adjust the CPU voltage and/or frequency to reduce power consumption by the CPU. These techniques help conserve power and reduce the heat generated by the CPU, but they adversely affect performance while the CPU transitions between low-power and high-performance states.

These power-savings techniques are known to have a negative impact on I/O latency and IOPS. When tuning for performance, you may benefit from reducing or disabling DVSF completely, even though this may increase power consumption.

DVFS, if available, is often configurable as part of your operating systems power management features as well as within your system's BIOS interface. Within the operating system and BIOS, DVFS features are often found under the Advanced Configuration and Power Interface (ACPI) sections; consult your computer documentation for details.

Set CPU Frequency to Highest Level

If your BIOS allows you to set a fixed CPU frequency, make sure the frequency always stays at the highest level.

Again, this will ensure that the CPU always stays at the highest frequency, and no performance is lost due to the transitioning between frequencies.

Disable CPU Power Throttling

Newer processors have the ability to go into lower power modes when they are not fully utilized. These idle states are known as ACPI C-states. The `C0` state is the normal, full power, operating state. Higher C-states (`C1`, `C2`, `C3`, etc.) are lower power states.

While ACPI C-states save on power, they can have a negative impact on I/O latency and maximum IOPS. With each higher C-state, typically more processor functions are limited to save power, and it takes time to restore the processor to the `C0` state.

When tuning for maximum performance you may benefit from limiting the C-states or turning them off completely, even though this may increase power consumption.

Setting ACPI C-State Options

If your processor has ACPI C-states available, you can typically limit or disable them in the BIOS interface (sometimes referred to as a Setup Utility). APCI C-states may be part of of the Advanced Configuration and Power Interface (ACPI) menu. Consult your computer documentation for details.

C-States Under Linux

Newer Linux kernels have drivers that may attempt to enable APCI C-states even if they are disabled in the BIOS. You can limit the C-state in Linux (with or without the BIOS setting) by adding the following to the kernel boot options:

```
intel_idle.max_cstate=0 processor.max_cstate=0 idle=poll
```

In this example, C-states are completely disabled and the processor is run at full throttle even when idle.

Set Fan Speeds to High

ioMemory devices monitor operating temperatures. In an attempt to maintain temperatures within an optimal range, and prevent thermal damage, the ioMemory VSL will start throttling write performance at a set temperature. If temperatures continue to rise, the VSL will shut down the device once it reaches the maximum operating temperature. See the *ioMemory Hardware Installation Guide* for information on the temperature thresholds for your device and how to monitor them.

To prevent thermal throttling, set your server fan speeds to high. If your BIOS has a High Performance/Power Mode, enable it when using ioMemory devices.

Power-saving Modes
You should also disable any power-saving modes. This will prevent the system from lowering the fan speeds. It will also prevent operating systems and the BIOS from suspending PCIe devices (using ASPM), including ioMemory devices. ioMemory devices do not support ASPM.

Disable Hyper Threading

Hyper-Threading Technology (HT), also known as Simultaneous Multi-Threading (SMT), creates a logical core that is separate from the physical core. The logical core can then be used to complete I/O processes. This may improve performance if your applications are I/O bound with non-CPU intensive processes.

With ioMemory devices installed, CPUs are freed from slower I/O processes, and HT may degrade performance.

If HT is enabled on your CPU, you should investigate whether your application performance would benefit from disabling HT technology. You can disable HT within the BIOS interface.

Use Asynchronous I/O

To improve throughput, latency, and responsiveness, use asynchronous input/output (sometimes called **AIO**, **async I/O**, or **non-blocking I/O**) with your application. This advanced approach to input and output operations will allow your operating system to complete other operations while it waits for an I/O operation to complete.

Without this enabled, the system resources will remain idle until each requested I/O operation is completed. This traditional approach is called synchronous I/O or blocking I/O.

Use Direct I/O

Direct I/O bypasses the OS page cache and allows applications to read and write directly from ioMemory devices.

About the Page Cache

Traditional I/O paths include the page cache. The page cache is a DRAM cache of buffered data from storage. By buffering data (both reads and writes) from storage devices in memory, the system can often avoid the time-consuming disk seeks and slower speeds of those legacy disk drives. Typically, all memory that is not allocated to applications is used for the page cache.

Page caching is why increasing RAM on a system usually increases the speed and responsiveness of I/O on that system. With a larger caching space, more I/Os can be buffered by the RAM. ioMemory devices are fast enough that this buffering in DRAM is often detrimental to performance.

About Direct I/O

Using direct I/O to bypass the page cache provides the following benefits:

- Less complex I/O path
- Lower overall CPU utilization
- Less host memory capacity and bandwidth usage

In most cases, direct I/O increases performance for ioMemory device-based systems, but you should benchmark your application to verify this. Some workloads that don't use AIO, threads, or multiple processes to create multiple outstanding requests may benefit from the page cache instead.

Many I/O-intensive applications have tunable parameters that control how they interact with the low-level I/O subsystem, including turning on direct I/O. Application tuning is outside of the scope of this document and is covered in application-specific notes published by Fusion-io.

Programming for Direct I/O on page 24 provides a more in-depth look at how to write C code utilizing Direct I/O.

Setting NUMA Affinity

Servers with a NUMA (Non-Uniform Memory Access) architecture may require special installation instructions in order to maximize ioMemory device performance. This includes most multi-socket servers.

On some servers with NUMA architecture, during system boot, the BIOS will not associate PCIe slots with the correct NUMA node. Incorrect mappings result in inefficient I/O handling that can significantly degrade performance. To prevent this, you must manually assign ioMemory devices optimally among the available NUMA nodes.

See the *ioMemory VSL User Guide* for Windows or Linux for more information on setting this affinity.

Set the Interrupt Handler Affinity

Device latency can be affected by placement of interrupts on NUMA systems. We recommend placing interrupts for a given device on the same NUMA node that the application is issuing I/O from. If the CPUs on this node are overwhelmed with user application tasks, in some cases it may benefit performance to move the the interrupts to a remote node to help load-balance the system.

Many operating systems will attempt to dynamically place interrupts across the nodes, and generally make good decisions.

Linux IRQ Balancing

In Linux this dynamic placement is called IRQ Balancing. You can check to see if the IRQ balancer is effective by checking `/proc/interrupts`. If the interrupts are unbalanced (too many device interrupts on one node) or on an overwhelmed node, you may need to stop the IRQ balancer and manually distribute the interrupts in order to balance the load and improve performance.

> Restarting the IRQ Balancer after the ioMemory VSL loads (and the ioMemory devices are attached) may resolve interrupt affinity issues. For example, run one of the following commands (depending on your distribution):
>
> ```
> /etc/init.d/irqbalance start
> ```
>
> ```
> /etc/init.d/irq_balance start
> ```
>
> If that does not resolve the affinity issues, then we recommend manual pinning the device interrupts to specific nodes.

Hand-tuning interrupt placement in Linux is an advanced option that requires profiling of application performance on any given hardware. Please see your operating system documentation for information on how to pin specific device interrupts to specific nodes.

Windows IRQ Policy

By default, Windows uses a policy of `IrqPolicyAllCloseProcessors` and a priority of `IrqPriorityNormal`, which should work best for most applications.

If manual tuning is needed, Windows provides the Interrupt Affinity Policy Tool. Information on this tool can be found at: http://msdn.microsoft.com/en-us/windows/hardware/gg463378. The settings that the application changes are listed at: http://msdn.microsoft.com/en-us/library/ff547969(v=vs.85).aspx.

With Windows Sever 2008 or newer on a machine with more than 64 processors, there's an additional `GroupPolicy` parameter that can be set through the registry in order to set the affinity to a different processor group. This is documented at: http://msdn.microsoft.com/en-us/windows/hardware/gg463349.

System Configuration

This section describes an example system configuration that may be used to produce ioMemory performance data.

Hardware

Component	Quantity	Description
Server	1	SuperMicro Server 1026GT-TF 1U
CPU	2	Intel Xeon 6 Core X5690 3.46GHz with 12MB Cache
RAM	12	4GB 1333MHz DDR3 ECC Registered CL9 x4 Dual Rank PC3-10600 DIMM
Chipset	1	Intel 5520 Chipset + ICH10R
PCIE Slots	3	2- x16 PCIe 2.0, 1- x4 PCIe 2.0
Power Supply	1	1200 Watts
Networking	1	Dual Intel 82576EB Gigabit
HDD	2	Seagate Constellation 7200 500GB SATA Hard Drive

BIOS

As an example, here are the BIOS options, referenced in the previous sections, as they appear on this system:

```
BIOS version: x8dtg1.a28

Exit -> Load Optimal Defaults

Processor and Clock Options -> Simultaneous Multi-Threading -> Disabled
                            -> Intel Turbomode Boost -> Disabled
                            -> C1E Support -> Disabled
                            -> Intel C-State Tech -> Disabled

Advanced Chipset Control -> CPU Bridge Configuration -> Throttling Closed
Loop -> Disabled

Hardware Health Configuration -> FAN Speed Control Modes -> Full Speed / FS

Save Changes and Exit
```

Software

Operating System: Ubuntu 11.04

I/O Tool: fio version 2.0.8 or higher

ioMemory VSL: Latest Version

As an example, here is how you can disable CPU performance scaling and IRQ balancing on an Ubuntu 11.04 system

```
apt-get -y remove irqbalance
```

```
cp /etc/init.d/ondemand /etc/init.d/ondemand.bak
sed -i 's|echo -n ondemand|echo -n performance|' /etc/init.d/ondemand
/etc/init.d/ondemand stop
/etc/init.d/ondemand start
```

Follow the user guide instructions to install and load the ioMemory VSL.

Installing fio

We recommend testing the performance of your ioMemory device(s) using the `fio` utility. The `fio` utility is an open source application; it is not created or maintained by Fusion-io. It stands for "Flexible I/O" Tester and is available at http://freecode.com/projects/fio

It works on most Unix-based operating systems and on Windows. A Windows installer package is available at http://bluestop.org/fio/

If you are testing performance using an ioMemory device installed in a virtualized environment, you should install `fio` on the guest OS that will either use the ioMemory device directly (through PCI Passthrough/VMDirectPathIO) or use the ioMemory device as a virtualized volume.

Sample Installation

Here are sample commands for installing the dependencies for building and running the `fio` test tool on an Ubuntu 11.04 system:

These commands require root permissions.

```
apt-get -y install      build-essential \
                       debhelper \
                       libaio-dev \
                       git
```

The `fio` utility is in active development, we recommend installing the latest version. These commands will make and install `fio`:

```
git clone git://git.kernel.dk/fio.git
cd fio
make
make install
cd
```

The fio I/O tool uses .ini configuration files to define specific I/O benchmark patterns. Command-line parameters can also be used; however .ini configuration files reduce the potential for user input error.

See Sample Linux fio Job Files on page 14 for the test scripts to produce the performance metrics for the ioMemory data sheets.

Please observe the following guidelines when performing raw I/O benchmarks:

- If run all together, these tests should be run in numerical order.
- Prior to each write test, detach the device, format it, and attach it.
- Prior to all read tests, the device needs to be pre-conditioned (i.e. filled with data). This is the fio-job-05 command/file. Once the device has been pre-conditioned, you may perform any number of read tests on the device.

Data Destructive Tests
Write tests (including pre-conditioning) will write data to the device. These tests will write over all the data on the device. Only perform write tests on new or unused devices.

- For the peak IOPS tests that are smaller than 4K, the device should have a formatted sector size of 512B.
- For tests using block sizes greater than or equal to 4KB, format the device with a sector size of 4KB.

If you are running tests on Windows, launch the Command Prompt as Administrator. Click **Start** and enter Command Prompt. Then right-click on **Command Prompt** and select **Run as administrator**.

- Run each test script with the following command:

```
fio --output=fio.output.<number> fio-job-<number>.ini
```

Example output:

```
/dev/fioa: (g=0): rw=rw, bs=512-512/512-512, ioengine=libaio, iodepth=32
...
/dev/fioa: (g=0): rw=rw, bs=512-512/512-512, ioengine=libaio, iodepth=32
fio-2.0.8
Starting 4 processes

/dev/fioa: (groupid=0, jobs=4): err= 0: pid=30918
  Description : [fio sequential 512 write peak IOPS]
  write: io=7018.6MB, bw=239540KB/s, iops=479075, runt= 30001msec
    slat (usec): min=15 , max=523 , avg=54.63, stdev=41.91
    clat (usec): min=1 , max=2059 , avg=206.50, stdev=78.85
      lat (usec): min=35 , max=2142 , avg=261.18, stdev=84.16
    clat percentiles (usec):
      | 1.00th=[ 70], 5.00th=[ 102], 10.00th=[ 120], 20.00th=[ 143],
      | 30.00th=[ 161], 40.00th=[ 179], 50.00th=[ 195], 60.00th=[ 215],
      | 70.00th=[ 237], 80.00th=[ 266], 90.00th=[ 306], 95.00th=[ 346],
      | 99.00th=[ 434], 99.50th=[ 474], 99.90th=[ 668], 99.95th=[ 820],
      | 99.99th=[ 1144]
    bw (KB/s) : min= 1, max=63280, per=24.58%, avg=58889.02, stdev=8769.50
    lat (usec) : 2=0.01%, 4=0.01%, 10=0.01%, 20=0.01%, 50=0.16%
    lat (usec) : 100=4.32%, 250=70.62%, 500=24.53%, 750=0.29%, 1000=0.05%
    lat (msec) : 2=0.02%, 4=0.01%
  cpu : usr=7.20%, sys=45.61%, ctx=5305818, majf=0, minf=105
  IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.1%, 32=115.3%, >=64=0.0%
    submit : 0=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    complete : 0=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, 64=0.0%,
>=64=0.0%
    issued : total=r=0/w=14372752/d=0, short=r=0/w=0/d=0

Run status group 0 (all jobs):
  WRITE: io=7018.6MB, aggrb=239539KB/s, minb=239539KB/s, maxb=239539KB/s,
mint=30001msec, maxt=30001msec

Disk stats (read/write):
  fioa: ios=0/8637747, merge=0/0, ticks=0/1407990, in_queue=1412150,
util=100.00%
```

You can convert the bandwidth in this example to MB/s by dividing by 1024.

Appendix A - Sample Linux fio Job Files

These Linux test files are optimized for the sample test system (as described in System Configuration on page 10). Optimal settings to test your system may vary depending on your system and target applications. For example, you may need to use a different `ioengine`; see the `fio` utility documentation for more information.

Data Destructive Tests
Write tests (including pre-conditioning) will write data to the device. These tests will write over all the data on the device. Only perform write tests on new or unused devices.

Windows Substitutions
If you are using these jobs on Windows, use the following substitutions:

Example Linux Job	Windows Substitutions
`ioengine=libaio/sync`	`ioengine=windowsaio`
`thread=0`	`thread=1`
`[/dev/fioa]` `filename=/dev/fioa`	`[\\.\PhysicalDrive`*`X`*`]` `filename=\\.\PhysicalDrive`*`X`*
`cpus_allowed=1-4`	***Remove this line for Windows*** `(nothing)`

Where *X* is the disk number (visible in **Disk Management**). For example, the path to Disk 2 would be `\\.\PhysicalDrive2`

Solaris Substitutions
If you are using these jobs on Solaris, use the following substitutions:

Example Linux Job	Solaris Substitutions
`ioengine=libaio`	`ioengine=solarisaio`
`direct=1`	`direct=0`
`[/dev/fioa]` `filename=/dev/fioa`	`[/dev/rdsk/`***`c*d0p0`***`]` `filename=/dev/rdsk/`***`c*d0p0`***
`cpus_allowed=1-4`	***Remove this line for Solaris*** `(nothing)`

Where ***`c*d0p0`*** is the master partition covering the entire raw block device.

fio-job-01 - Random 512B Write Test for Latency

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-01.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=0
blocksize=512
ioengine=sync
numjobs=1
thread=0
direct=1
iodepth=1
iodepth_batch=1
iodepth_batch_complete=1
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 512 write LATENCY
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-02 - Random 512B Write Test for Peak IOPS

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-02.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=0
blocksize=512
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 512 write peak IOPS
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-03 - Sequential 512B Write Test for Peak IOPS

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-03.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=rw
rwmixread=0
blocksize=512
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio sequential 512 write peak IOPS
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-04 - Random 1M Write Test for Peak Bandwidth

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-04.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=0
blocksize=1M
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 1M write peak BW
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-05 - (PRECONDITION) Sequential 1M Complete Write Test

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-05.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=write
rwmixread=0
blocksize=1M
ioengine=libaio
thread=0
size=100%
iodepth=16
group_reporting=1
description=fio PRECONDITION sequential 1M complete write

[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-06 - Random 512B Read Test for Latency

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-06.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=100
blocksize=512
ioengine=sync
numjobs=1
thread=0
direct=1
iodepth=1
iodepth_batch=1
iodepth_batch_complete=1
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 512 read LATENCY
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-07 - Random 512B Read Test for Peak IOPS

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-07.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=100
blocksize=512
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 512 read peak IOPS
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-08 - Sequential 512B Read Test for Peak IOPS

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-08.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=rw
rwmixread=100
blocksize=512
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio sequential 512 read peak IOPS
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

fio-job-09 - Random 1M Read Test for Peak Bandwidth

To use this script:

1. Create a file with the script contents
2. Name it `fio-job-09.ini`
3. Replace `/dev/fioa` with the particular device you wish to test.
4. Follow the instructions in Testing on page 12

```
[global]
readwrite=randrw
rwmixread=100
blocksize=1M
ioengine=libaio
numjobs=4
thread=0
direct=1
iodepth=32
iodepth_batch=16
iodepth_batch_complete=16
group_reporting=1
ramp_time=5
norandommap=1
description=fio random 1M read peak BW
time_based=1
runtime=30
randrepeat=0


[/dev/fioa]
filename=/dev/fioa
cpus_allowed=1-4
```

Appendix B - Programming for Direct I/O

Direct I/O on Linux

Under Linux, the best way to enable direct I/O is on a per-file basis. This is done by using the `O_DIRECT` flag to the open() system call. For example, in an application written in C, you may see a line similar to this:

```
fd = open(filename, O_WRONLY);
```

To make this file be accessed through unbuffered or direct I/O, you would change the line to this:

```
fd = open(filename, O_WRONLY | O_DIRECT );
```

Note that the ioMemory device requires that all I/O performed on a device using O_DIRECT must be 512-byte aligned and a multiple of 512 bytes in size. Buffers used to read data to and write data from must be sector-size aligned. The I/O performed must also be equal or greater to the sector size on the device. For example, if you have formatted the ioMemory device to 4KB sectors, all I/O performed must be at least 4KB. See the sample code below for an example on how to align buffers for O_DIRECT.

The following is a simple application that writes a pattern to the entire ioMemory device in 1MB chunks, using `O_DIRECT`:

```
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#define __USE_GNU
#include <fcntl.h>
#include <string.h>

#define FILENAME "/dev/fioa"

int main( int argc, char **argv) {
    void *buf;
    int ret = 0;
    int ps = getpagesize();
    unsigned long long int bytes_written = 0;
    int fd;

    if( (ret = posix_memalign(&buf, ps, ps*256)) ) {
        perror("Memalign failed");
        exit(ret);
    }

    memset(buf, 0xaa, ps*256);
    if( (fd = open(FILENAME, O_WRONLY | O_DIRECT) ) < 0 ) {
        perror("Open failed");
        exit(ret);
    }

    bytes_written = 0;
    while( (ret = pwrite(fd, buf, ps*256, bytes_written)) == ps*256) {
        bytes_written += ret;
    }

    printf("Wrote %lld GB\n", bytes_written/1000/1000/1000);

    close(fd);
    free(buf);
}
```

In most cases, performance can be slightly enhanced by using `pwrite` instead of `write`.

Direct I/O on Windows

The following is sample code for implementing Direct I/O on Windows. The sample includes two methods for aligning the buffer on sector boundaries.

Code Sample

```
#include <Windows.h>
#include <iostream>
#include <malloc.h>
using namespace std;

// defining the following means we will use _aligned_malloc to get a memory
aligned buffer.
// Comment out or delete if not wanting to use _aligned_malloc
#define USE_ALLOC_MEMALIGN

// change these two values for your system #define drivePath
L"\\\\.\\PhysicalDrive1"
#define filePath L"T:\\test.txt"

#ifndef USE_ALLOC_MEMALIGN
    #define ROUND_UP_SIZE(Value,Pow2) ((SIZE_T) ((((ULONG)(Value)) + (Pow2)
- 1) & (~(((LONG)(Pow2)) - 1))))
    #define ROUND_UP_PTR(Ptr,Pow2)  ((void *) ((((ULONG_PTR)(Ptr)) + (Pow2)
- 1) & (~(((LONG_PTR)(Pow2)) - 1))))
#endif

class DirectFile
{
    public:
        DirectFile() {};
        ~DirectFile() {
            if(hFile != INVALID_HANDLE_VALUE)
                CloseHandle(hFile); };
        int open(wchar_t* filename);
        void write(void* alignedXferAddr, DWORD xferSize);

    private:
        HANDLE hFile;
};
int DirectFile::open(wchar_t* filename)
{
    /*
    * In the following call to CreateFile, we use both the FILE_FLAG_WRITE_
THROUGH
    * and FILE_FLAG_NO_BUFFERING flags because the latter prevents system
caching
    * but does not affect hard disk caching, and the former prevents hard
disk caching
    */
    hFile = CreateFileW(filename,
                    GENERIC_READ | GENERIC_WRITE,
                    FILE_SHARE_DELETE | FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL,
                    CREATE_ALWAYS,
                    FILE_FLAG_WRITE_THROUGH | FILE_FLAG_NO_BUFFERING,
                    NULL
                );
    if(hFile == INVALID_HANDLE_VALUE)
    {
        cout << "Windows Error: " << GetLastError() << " prevented opening
the file " << filename << endl;
        return -1;
    }

    return ERROR_SUCCESS;
}

void DirectFile::write(void* alignedXferAddr, DWORD xferSize) {
    DWORD bytesWritten = 0;
    DWORD bytesToWrite = xferSize;
    // To write to a device asynchronously, use WriteFileEx
    BOOL ret = WriteFile(
                hFile,
                alignedXferAddr,
                bytesToWrite,
                &bytesWritten,
                NULL
            );
    if(FALSE == ret)
    {
        cout << "Windows Error: " << GetLastError() << " prevented writing
to the file." << endl;
    }
    else
    {
        if(bytesWritten != bytesToWrite)
        {
            cout << "ERROR: Not all data was written to the file." << endl;
        }
        else
        {
            cout << "Successfully wrote " << bytesWritten << " bytes." <<
endl;
        }
    }
}

int main( int argc, char **argv)
{
    HANDLE hDevice = INVALID_HANDLE_VALUE;
    wchar_t* wszPath = drivePath;
    BOOL result = FALSE;
    STORAGE_PROPERTY_QUERY qp;
    STORAGE_ACCESS_ALIGNMENT_DESCRIPTOR sa;
    DWORD bytesRet = 0;
    DirectFile myFile;
    DWORD sectorSize = 0;
    DWORD numSectors = 1024;
    void* unalignedBufAddr = NULL;
    DWORD unalignedBufSize = 0;
    void* alignedXferAddr = NULL;
    DWORD alignedXferSize = 0;

    ZeroMemory(&qp, sizeof(qp));
    ZeroMemory(&sa, sizeof(sa));

    qp.QueryType = PropertyStandardQuery;
    qp.PropertyId = StorageAccessAlignmentProperty;

    // Get the device's sector information
    hDevice = CreateFileW(wszPath,
                          0,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          NULL,
                          OPEN_EXISTING,
                          0,
                          NULL);
    if(hDevice == INVALID_HANDLE_VALUE)
    {
        cout << "Failed to open the physical device" << endl;
        goto cleanup;
    }
    result = DeviceIoControl(hDevice,
                            IOCTL_STORAGE_QUERY_PROPERTY,
                            &qp,
                            sizeof(STORAGE_PROPERTY_QUERY),
                            &sa,
                            sizeof(STORAGE_ACCESS_ALIGNMENT_DESCRIPTOR),
                            &bytesRet,
                            NULL);
    CloseHandle(hDevice);

    sectorSize = sa.BytesPerPhysicalSector;

#ifdef USE_ALLOC_MEMALIGN
    unalignedBufSize = numSectors * sectorSize;
    alignedXferSize = unalignedBufSize;
    unalignedBufAddr = _aligned_malloc(unalignedBufSize, sectorSize);
    if(unalignedBufAddr != NULL)
        alignedXferAddr = unalignedBufAddr;
    else
        return -1;
#else
    /*
    * In order to get a buffer big enough for the write, we need to be sure
it can handle the
    * number of sectors to write as well as alignment on a sector boundary.
Therefore, we use
    * ROUND_UP_SIZE to make sure it is big enough for the number of sectors
to write, and then
    * we add sectorSize to this value in order to allow for aligning on a
sector boundary.
    *
    * ROUND_UP_PTR will move a pointer to a sector boundary within the newly
allocated buffer,
    * and it is this pointer that we need use when writing out to disk.
    */
    unalignedBufSize = sectorSize + ROUND_UP_SIZE((numSectors * sectorSize),
sectorSize);
    alignedXferSize = unalignedBufSize - sectorSize; // we don't want to
memset beyond the end of the buffer
    unalignedBufAddr = (LPBYTE) malloc(unalignedBufSize);
    if(!unalignedBufAddr)
    {
        goto cleanup;
    }

    alignedXferAddr = ROUND_UP_PTR(unalignedBufAddr, sectorSize); #endif

    wchar_t* filename = filePath;
    if(myFile.open(filename) < 0)
    {
        goto cleanup;
    }

    memset(alignedXferAddr, 0xaa, alignedXferSize);
    myFile.write(alignedXferAddr, alignedXferSize);

cleanup:
#ifdef USE_ALLOC_MEMALIGN
    if(unalignedBufAddr)
        _aligned_free(unalignedBufAddr); #else
    if(unalignedBufAddr)
        free(unalignedBufAddr);
#endif

    return 0;
}
```

Appendix C - Testing and Tuning Strategies

Previous sections in this guide describe the system configurations and tests that may be used to produce peak ioMemory performance data.

You may wish to apply the principles of this guide to test an ioMemory device on your particular system. While no synthetic workload can mimic the behavior of real applications, the considerations in this appendix will help you design tests that more accurately represent real-world use of an ioMemory device.

Benchmarking ioMemory Devices

ioMemory devices have distinct design differences from traditional "spinning" media. These differences result in reduced latency and increased bandwidth. Because ioMemory devices perform differently from traditional devices, some of the traditional benchmarking tests and methodology may no longer be appropriate.

For example, some traditional tests are designed to incorporate disk seek time, which no longer applies to ioMemory devices. Many of the strategies in this appendix are created for the unique characteristics of ioMemory devices.

Precondition the Device

When a read command is issued to an ioMemory device for a Logical Block Address (LBA) region that has never been written to, the device will automatically return all zeros. It will do this without accessing the NAND flash because it contains no data. This will result in speeds that are greater than the real-world capabilities of the ioMemory device.

To prevent this, be sure to "precondition" the device by writing to all of the LBAs that you intend to read from. For example, run `fio-job-05`. This will fill the device with random data and it will result in read tests that more closely resemble real-world use.

Writing Tests on a Full Device

Some benchmarking tests perform constant writes randomly across all regions of a device. This legacy strategy was designed to show how the seek time of the device impacts the performance of writes to random areas of the device. Because Solid State Storage (SSS) devices, including ioMemory devices, have no variable seek latency, this test method is often unnecessary. It may not represent realistic use of the device unless your application uses a high percentage of "hot" data.

In real-world applications, the data on a storage device is often characterized by a mix of "hot" (changing) data and "cold" (relatively static) data. Completely filling the device and then running random write tests across the device simulates a scenario in which 90% or more of the data on the device is hot (continually being rewritten). This less-common scenario will result in reduced performance unless the device is tuned for the heavy/constant write workload by increasing the reserve space on the ioMemory device.

As writes occur and the ioMemory device becomes full, the device relies on a reserved space (in addition to the advertised capacity) to perform maintenance on the data. This reserve space is large enough for most applications and uses but, as the percentage of hot data on a full device increases, the device must rely on the reserve space to wear level the device.

If a heavy write workload is important to your application, we recommend the following testing strategy:

1. Fill the device only until it reaches the total capacity that the application will use.
2. Determine your application's mix of hot and cold data.
3. Limit the benchmarking application to continually rewrite only the hot data portion.

How to Tune for Write-heavy Workloads

You can use either the ioSphere management software or the `fio-format` command-line utility to down-format the device. This will increase the reserve space at the cost of overall device capacity, but it will improve performance for heavy write workloads.

- **ioSphere**: The ioSphere software gives you two down-formatting preset options: **Improved Performance** and **Maximum Performance**. With ioMemory VSL 3.x and later, these presets under-format the device to these percentages of factory capacity:
    - **Improved Performance**: 90% of factory capacity
    - **Maximum Performance**: 80% of factory capacity
- **`fio-format`**: Use the `-s` option to set the device capacity to a specific size:

```
fio-format -s <size M|G|T|%> <device>
```

Where `T`, `G`, or `M` are specific capacities in terabytes, gigabytes or megabytes, respectively. Or you can use the percent symbol (`%`) and set the size as a percentage of the advertised capacity, for example:

```
fio-format -s 90% /dev/fct1
```

Decreasing Latency for Writes

ioMemory devices are designed to ensure that all data that has been acknowledged as received by the drive will not be lost — even in the event of a power loss. Some systems don't properly detect and handle power loss or other brown out events, resulting in the system sending corrupted data to the ioMemory device to be stored.

In order to prevent corrupt data from being received during power loss, a `5 µsec` delay is added to writes. This allows the ioMemory device to drop possibly corrupted data when unexpected power loss is detected. For systems where this is not a concern or that are known to properly handle power loss, this delay can be tuned using the `iodrive_dma_delay` driver parameter.

Default is `16` (`5 µsec`), valid values are `0-31` with `0` being no delay and `31` adding a `10 µsec` delay. To disable the delay set:

```
iodrive_dma_delay=0
```

Disk Aggregation (RAID) Considerations

Depending on your application, you may see a small, but measurable, I/O overhead that impacts IOPS when using Microsoft Windows Dynamic Disk for software RAID, Linux MD RAID, or other volume management techniques.

You can test for this overhead by benchmarking a single, aggregated RAID 0 (consisting of two or more ioMemory devices) and comparing that output to the same benchmark settings run on just the ioMemory devices (without the RAID configuration).

If application-layer striping or aggregation is available, you should benchmark both the application-layer aggregation and the Operating System layer aggregation, and then determine which works better for that application.

Partition Alignment

There is no need to tune the ioMemory device for partition alignment.

Linux-specific Tuning

Linux Filesystem Tuning

We currently recommend using the XFS or EXT4 filesystems.

Mounting Linux filesystems with noatime

On Linux systems, one of the standard file statistics that is tracked by filesystems is `atime` (access time), along with `mtime` (modification time) and `ctime` (creation time). For most use cases, tracking `atime` is not needed and just results in additional work for the filesystem that can slow performance.

Most filesystems support a mount time option that disables tracking of `atime` using the `noatime` option. For example, under linux adding the `noatime` to the options field of `/etc/fstab` will cause it to be set on next boot, for example:

```
/dev/fioa   /srv/nfs   ext4   defaults,noatime   0 0
```

The `noatime` option can also be specified when manually mounting a filesystem under Linux, like so:

```
$ mount -t ext4 -o noatime /dev/fioa /srv/nfs
```

XFS Filesystem

We recommend using the `noatime`, `nobarrier`, and `discard` (TRIM) options when using the XFS filesystem. Discard for XFS is only available with newer Linux Kernels

ext2-3-4 Tuning

We recommend using ext4 over ext2/3. The ext4 filesystem can achieve up to 3x the performance of a tuned ext2 or ext3 solution. We recommend using the `noatime` and `discard` (TRIM) options when using the ext4 filesystem.

Setting Stride Size and Stripe Width for ext* when Using a RAID

The ext* filesystem family has create-time options of stride and stripe width. Stride helps the filesystem ensure that critical metadata structures are spread across the disks in the RAID evenly to keep any given disk from becoming a hotspot. Stripe width allows for filesystem optimization.

To calculate the correct values for stride size, take the chunk size of the RAID array and divide it by the block size of the filesystem. For example:

stride = (chunk size / filesystem block size)

Stripe width calculation requires the stride size and the number of data-bearing disks. The following table shows the calculation for number of data bearing disks per raid type.

dbd = # of data bearing disks

total_disks = # of active disks

mirrored_sets = The number of raid 1 mirrored sets that are used to form the higher\- level group

RAID Level	Data bearing disks (dbd)
0 (Striping)	total_disks
1 (Mirroring)	1
5	total_disks - 1
6	total_disks - 2
10	mirrored_sets
50	mirrored_sets - 1

To calculate the stripe width, take the number of data bearing disks and multiply it by the stride.

stripe_width = dbd * stride

Sample Configuration

Create a RAID10 with 10 ioMemory devices combined into five mirrored sets, (two mirrored ioMemory devices per set) and 256KB chunk size.

Create an ext3 filesystem with a 4K block size:

(stride) 64K = 256K / 4K

(dbd) 5: 5 mirrored sets and 1 dbd per mirror (5), then 1 stripe across all 5 sets.

(stripe_width) 320 = 5 * 64K

This results in the following command:

```
$ mkfs.ext3 -b 4096 -E stride=64 -E stripe_width=320 /dev/md0
```

For more information on setting the stripe size, see the following:

http://wiki.centos.org/HowTos/Disk_Optimization

⚠ Not all versions of mkfs.ext3 support the stripe_width option, for best practices for your distribution, please see its documentation.

Tuning Virtual Memory (Paging) in Linux

Paging Background

With paging, the operating system divides the data stored in memory into same-size blocks called *pages*. With pages, a single process can be addressed as multiple chunks (pages) over noncontiguous sectors of RAM. This increases performance and reduces various storage and fragmentation problems. See http://en.wikipedia.org/wiki/Paging for more information.

Virtual Memory Subsystem Background

Paging also allows the kernel to manage data through the Virtual Memory subsystem. The system manages data through the page cache and swapping space.

- **Page Cache**: Applications running under the Linux Kernel do not write directly to block devices. Instead, they write to a portion of the RAM (or pages) that buffers the writes. This buffer is called the page cache, and it is backed by a block device. Writes are then flushed from the page cache to the block device. This process is transparent to applications. See http://en.wikipedia.org/wiki/Page_cache for more information.

  ⚠ Depending on your use case, you may consider bypassing the page cache entirely by implementing direct I/O on your file system. This will allow the application to write directly to the ioMemory device. For more information, see Programming for Direct I/O on page 24.

- **Swap Space**: The kernel can off load and temporarily store pages of data on a block device that has space allocated as a swap space. This frees up RAM for other processes, and the kernel can load (swap) the page back into the RAM when needed. The kernel swaps out pages when the it needs more RAM (memory pressure) or if the data in a page has not been used for some time. How aggressively the kernel pushes inactive pages to swap (page cache) can be controlled. See the *Swappiness* section below for more information.

Tuning the Virtual Memory Subsystem

The VM subsystem is located at `/proc/sys/vm`. If you navigate to that directory and run the `ls -l` command, it will output the various files of this subsystem.

Tuning Swappiness

Swapiness is how aggressively the kernel pushes inactive pages to the swap space and it is controlled with the `swappiness` file. This file has a valid range of `0-100` and a default value of `60`. The higher the value, the more agressive the swapping. A value of 100 will result in no swapping due to inactivity. You can check the current value by running this command:

```
# sysctl vm.swappiness
```

Sample output:

```
vm.swappiness = 60
```

You can set this value using the following command (in this example a less aggressive value is used):

```
# sysctl -w vm.swappiness=20
```

Note the following:

- Under Linux, ioMemory devices are often better utilized as accelerated block devices than as a swap space. While ioMemory devices are faster than hard disk swap spaces, frequent page swaps is a symptom that more RAM is required or the system needs to be tuned for maximum performance.
- On a large memory system, the kernel may page out large amounts of data that it considers inactive. This can cause I/O saturation on the swap device both when the data is going out and coming back in.

Tuning Dirty Pages

Dirty pages are pages that have been modified (written to) in the page cache, but haven't been committed to the block device. Once the data is committed to the block device, it has been "flushed," and the page is freed. You can improve performance by tuning how often the `pdflush` daemon flushes the dirty pages.

There are several parameters in the VM subsystem that you can modify:

- `dirty_writeback_centisecs`: This value (in 100ths of a second) defines how often the `pdflush` writeback daemon wakes up and flushes dirty pages that are marked for flushing. The default value is `500` (or 5 seconds).
- `dirty_expire_centisecs`: This value (in 100ths of a second) defines how old a dirty page should be before it should be marked for flushing. The default value is `3000` (or 30 seconds).
- `dirty_background_ratio`: This is a ratio (percentage) of total system memory that can contain dirty pages before the `pdflush` writeback daemon will start writing out dirty data. The default value is 10 (10%).

If the system has a large amount of memory and heavy I/O, you may want to tune these parameters so flushes occur more often or at lower ratios. Otherwise, the amount of dirty pages may grow substantially until they are marked for flushing (after the default 30 seconds) or they hit the default ratios.

You can change these values just like the `swapiness` parameter:

```
# sysctl -w vm.<parameter>=<new-value>
```

`dirty_ratio`:

Some online guides may suggest that you lower the percentage value of the `dirty_ratio` parameter. Doing so may actually hurt performance.

This is a ratio (percentage) of total system memory that can contain dirty pages before a flush must occur. Once this level of dirty pages exist in the cache, further buffered writes will be stalled until the page cache is flushed; the system may seem sluggish and I/O latency can experience significant spikes. This default value is 40 (40%).

Appendix D - Debugging Performance Issues

The `fio-pci-check` utility, which is recommended in debugging several of the following performance issues, is not fully functional on all operating systems. Consult the *ioMemory VSL User Guide* for your operating system for more information.

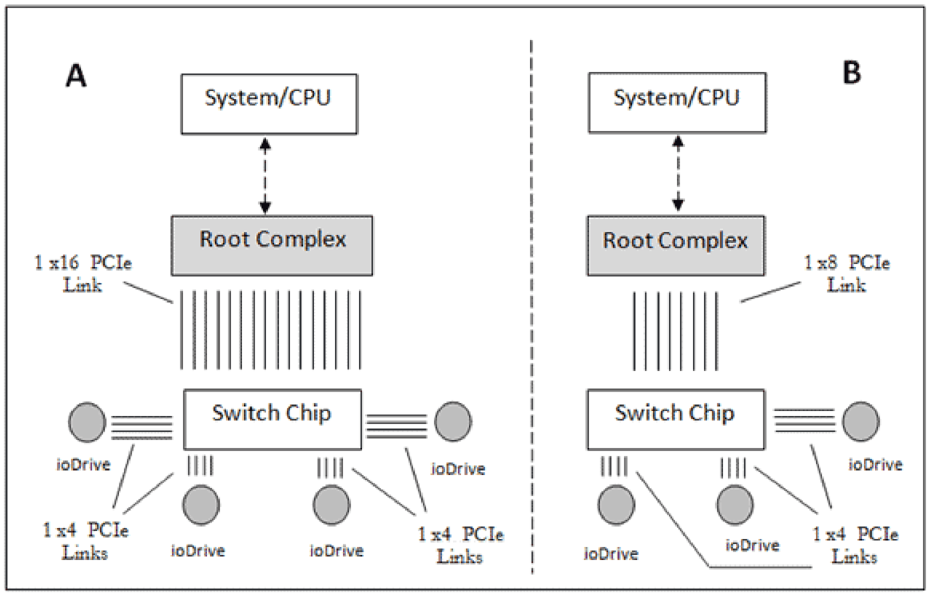
Over-Subscribed Bus

Problem 1

It is possible to install multiple ioMemory devices and other PCIe peripherals in a way that causes unequal performance from each of the drives. In networking, this is frequently called over-subscription, and it is common in all but the latest generation of systems. For PCIe bus based high-performance peripherals, having an over-subscribed topology can drastically affect performance, especially if the devices are part of a RAID.

The following figure illustrates a balanced bus vs. an over-subscribed bus. System A has 16 PCIe lanes connected from ioMemory devices to a switch chip, and 16 lanes from the switch chip to the root complex. This results in a balanced topology. System B has 12 PCI lanes connected from ioMemory devices to a switch chip, but only 8 lanes from the switch chip to the root complex. This causes an over-subscription condition, decreasing performance.

Balanced bandwidth (A) vs. over-subscription (B):



Keep in mind that other PCIe devices such as high performance network cards and graphic cards can also create an over-subscription condition.

Solution 1

To verify that there are no bandwidth bottlenecks in the PCIe bus, it is important to run the `fio-pci-check utility` and look for errors.

Problem 2

A related common performance problem that is harder to diagnose is when the PCIe lanes are run with the southbridge. Using the southbridge chipset is inherently slower than using the northbridge. This can create an oversubscribed configuration that `fio-pci-check` may not be able to diagnose.

Solution 2

Doing this diagnosis is beyond the scope of this document. For assistance in determining if your system suffers from this problem, collect a `fio-bugreport` and the results of the system-vetting benchmarks. Send the bundle to support@fusionio.com, requesting assistance in debugging a performance issue.

Below is an example of a bandwidth error reported by `fio-pci-check`, caused by an over-subscription condition.

Any line printed with an asterisk indicates a possible problem detected by fio-pci-check.

Bandwidth error, over-subscription:

```
fio-pci-check

Root Bridge PCIe 3000 MB/sec
    Bridge 00:02.00 (01-05)
      * Needed 3000 MB/sec Avail 2000 MB/sec
          Bridge 01:00.00 (02-04)
              * Needed 3000 MB/sec Avail 2000 MB/sec
                  Bridge 02:00.00 (03-03)
                    * Needed 1000 MB/sec Avail 1000 MB/sec
                        ioDrive 03:00.0 Firmware 14071
```

Handling PCI Express Errors

Problem

Data errors may be introduced when data is passed across the bus.

Solution

PCIe detects data errors, and in many situations it can also correct data errors. These errors can also be detected by running `fio-pci-check`. If such errors have been detected on your system:

- Reseat all risers and the ioMemory devices in the system.
- Run some significant data to and from the ioMemory device in question and check for errors again.

- If the errors continue, the most common problem is a riser card, so try swapping that out.
- If errors persist, try swapping out the ioMemory device, and finally the motherboard.

Because of their high performance, ioMemory devices are very demanding on the PCIe bus and they show errors in the system that other devices might not reveal.

If multiple cards are installed in the system, the `fio-status` and `fio-beacon` utilities can be used to determine which of the drives is failing. `fio-status` returns the serial number for the device, and `fio-beacon` turns on the beacon LED for identification.

⚠ Some PCIe chips do not properly report PCIe errors, and they may report errors when none exist. In most cases this has been found to happen on a bridge chip. This failure typically shows under the following conditions:

- Multiple rapid executions of `fio-pci-check` have been issued.
- No data has been passing over the bus reporting errors.
- All drivers for attached peripherals are unloaded.

Below is an example of PCIe errors captured on a system with an ioMemory device.

⚠ Windows does not allow clearing of all errors, so only errors registered on the ioMemory devices should be considered.

PCI Express errors:

```
Root Bridge PCIe 3000 MB/sec
       Bridge 00:02.00 (09-12)
               Needed 1000 MB/sec Avail 1000 MB/sec
                 Bridge 09:00.00 (0a-0f)
                       Needed 1000 MB/sec Avail 1000 MB/sec
                     * Fatal Error(s): Detected
                     * Unsupported Type(s): Detected
                       Clearing Errors
                         Bridge 0a:00.00 (0b-0d)
                               Needed 1000 MB/sec Avail 1000 MB/sec
                                 ioDrive 0b:00.0 Firmware 14071
       Bridge 00L04.00 (13-15)
               Needed 1000 MB/sec Avail 1000 MB/sec
                 ioDrive 0b:00.0 Firmware 14071
```

PCIe Link Width Improperly Negotiated

Problem

PCIe devices negotiate link speeds. If the system is having difficulties communicating with the ioMemory device, the system might talk to the ioMemory device using only an x1 link, which has 1/4th the performance of an x4 link.

Solution

Use the `fio-pci-check` or `fio-status` utilities to check for this problem and to report any issues with link width.

Below is an example of PCI link width errors captured on a system with an ioMemory device.

`fio-pci-check` Output:

```
Root Bridge PCIe 250 MB/sec

Bridge 00:01.00 (01-01)
 Needed 250 MB/sec Avail 250 MB/sec
 Current control settings: 0x0000
 ...
 Maximum link speed: 2.5 Gb/s
 Maximum link width: 16 lanes
 Current link_capabilities: 0x00001011
 Link speed: 2.5 Gb/s
 Link width is 1 lanes
```

`fio-status` Output:

```
Adapter: Dual Controller Adapter
 Fusion-io ioDrive2 DUO 2.41TB, Product Number:F01-001-2T41-CS-0001, FIO
SN:1149D0969
 ...
 PCIe negotiated link: 1 lanes at 5.0 Gt/sec each, 500.00 MBytes/sec total
```

Using cp and Other System Utilities

Problem

Most traditional system utilities, such as `cp` and `rsync`, were built with slow legacy storage in mind and do not achieve optimal performance with ioMemory devices.

This is not to say that ioMemory devices do not work well with standard utilities. Performance is better than traditional storage using the same utilities, and additional performance benefits will be available in the future as these utilities are optimized for high-performance storage.

Solution

Avoid using traditional system utilities for general benchmarking purposes, as they do not provide a good representation of peak performance.

Additional Information and Support

Please visit http://support.fusionio.com for more information about the ioMemory VSL software (including release notes and user guides), Knowledge Base articles, or to contact Customer Support.