|  |  |
| --- | --- |
| M:\My Pictures\Drawing Scans\pheonixcoloured.JPG | ***XDD***  ***User Guide*** |

Version 7.0.0.rc28

February 2015

Oak Ridge National Laboratory & I/O Performance, Inc.

Copyright © 1992-2015, I/O Performance, Inc.

Copyright © 2009-2015, Oak Ridge National Laboratory / UT-Battelle

|  |  |  |
| --- | --- | --- |
| Principle Author:  Contributing Authors: | | Thomas M. Ruwart  tmruwart@ioperformance.com  Steve Hodson, DoE/ORNL  hodsonsw@ornl.gov  Steve Poole, DoE/ORNL  spoole@ornl.gov  Bradley Settle Myer, DoE/ORNL  settlemyerbw@ornl.gov  Russell Cattelan, Digital Elves  russell@thebarn.com  Alex Elder  Kelly M. T. Huffer, DoE/ORNL  testakm@ornl.gov |
|  | |  |
| Phone: | | (865) 574-0706 |
| Email: | | testakm@ornl.gov |
|  |  | |

**Change History:**

The revision numbers have the following meaning. The “number” such as 5.7, 5.8, 5.9, …etc. represent major release changes such as the addition of new options or sections to XDD.

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Author | Description of Changes |
| 7.0 | 18 JAN 2010 | Thomas M. Ruwart | Release of 7.0 |
| 7.0 | 01FEB2011 | Thomas M. Ruwart | Pre-release 19 of 7.1 |
| 7.0 | 07APR2011 | Thomas M. Ruwart | After release of Version 7.0 rc19 |
| 7.0 | 18APR2011 | Thomas M. Ruwart | New heartbeat suboptions |
| 7.0 rc21-1 | 02APR2012 | Thomas M. Ruwart | Maintenance Release |
| 7.0 rc27 | 23NOV2014 | Thomas M. Ruwart & Kelly M. T. Huffer | Addition of xddmcp |
| 7.0 rc28 | 28FEB2015 | Kelly M. T. Huffer &  Thomas M. Ruwart | Release with rc 28 |

Table of Contents

1 Introduction 5

1.1 About this document 5

1.2 Acknowledgements 5

1.3 About XDD 5

1.4 License Agreement 5

1.5 Distribution of XDD 5

1.6 Versioning 6

1.7 System Requirements 6

2 What’s new 7

2.1 In Release 7.0.0 7

3 Compiling and Installing XDD 7

3.1 XDD Source Code Overview 7

3.2 Build Notes 8

3.2.1 Configure and Make 8

3.2.2 Supported Operating Systems 9

4 Theory of Operation 10

4.1 XDD Program Structure Overview 10

4.2 Basic Operation 11

4.3 Command Line Options and the Setup File 11

4.4 Operation Specification and Request Sizes 12

4.5 Target Specification and Multiple Target Synchronization 12

4.6 XDD Run Time or Run Length 12

4.7 I/O Range 13

4.8 Access Patterns 13

4.9 Reading and Writing Devices and Files 14

4.10 Processor Allocation and Priority Assignment 15

4.11 Preallocation 15

4.12 I/O Time Stamping 17

5 Running XDD Programs 18

5.1 XDD Command Line Arguments Synopsis 18

5.2 Detailed Option Specifications 22

5.3 Target-specific options 32

5.4 Lockstep Operations 32

5.5 I/O Operation Ordering 32

5.5.1 Serial Ordering 34

5.5.2 Loose Ordering 34

5.5.3 No Ordering 35

5.6 Timeserver and Gettime 35

5.6.1 Timeserver and Gettime Command Line Arguments 37

6 XDDCP 38

6.1 XDDCP Usage 39

6.1.1 How to Restart 39

7 XDDMCP 40

7.1 XDDMCP Usage 40

7.1.1 Restart Usage 40

7.2 End-to-End Theory of Operation 41

8 Output and Reports 44

8.1 Reporting Options 44

8.2 Output Format 44

8.3 Example Timestamp Output 49

9 Performance Tuning Hints 51

9.1 Caches and Write Performance 51

9.2 Fibre Channel Frame Sizes 51

10 Examples 52

10.1 Example 1 – Basic XDD command line 52

10.2 Example 2 – Specifying multiple targets and time limit 52

10.3 Example 3 – Time Stamping and Setup File 52

10.4 Example 4 – Random Seeks 53

10.5 Example 5 – End to End operation 53

11 Under the Hood 55

11.1 XDD General Operation 55

11.2 The XDD Thread Structure 55

11.3 XDD Barriers 56

12 Acknowledgements 57

# Introduction

## About this document

This document is a guide to compiling, installing, and running the XDD program. It also has a variety of examples and hints for understanding the use and results of XDD.

## Acknowledgements

The continuing work on XDD is supported by funding and resources of the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL), the Extreme Scale Systems Center (ESSC) at ORNL, and the Department of Defense.

## About XDD

XDD was originally developed in 1992 as a command line tool for measuring performance and characterizing disk subsystem I/O behavior on anything from a single system to a large cluster of systems – something a little bit better than the traditional dd utility in UNIX. The goal was to provide consistent and reproducible performance measurements of disk I/O traffic. After its success in the SGI IRIX[[1]](#footnote-1) environment, where it was originally developed, it has been ported to run under Linux, AIX, Solaris, Mac OS X, and Windows.

Over the years XDD has assumed many capabilities. As of XDD Release 7.0, it is possible to use XDD to reliably copy exceedingly large files from one computer to another over a network at speeds approaching the capability of the underlying hardware. See the end-to-end option and examples for more information on this feature.

## License Agreement

It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License Version 2 for more details. You should have received a copy of the GNU General Public License along with the program in a file named “Copying”; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, or visit their web site at <http://www.gnu.org/licenses/gpl.html>.

## Distribution of XDD

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

## Versioning

The version numbering format is as follows:

Major.Minor.Revision.ReleaseCandidate

Where the Major and Minor are numbers (i.e. 7.0).

A final release distribution file would look like so:

**xdd-7.0.0.rc28.tar.gz**

## System Requirements

XDD can run on a relatively minimally configured system, but it is recommended that adequate main memory capacity and processor speed be available when using some of the more advanced features of XDD. The XDD program itself is not CPU intensive, but the faster the processor and the less loaded a system is the more accurate and consistent the results will be. As a rule, a minimum of 128MB of main memory is recommended just for the XDD program itself. All other system parameters are left to the discretion of the user.

# What’s new

## In Release 7.0.0

There are several major changes in release 7.0 that include:

* Major restructuring of the source code for readability, extensibility, and supportability
* Addition of an end-to-end option that allows XDD to quickly and reliably copy data from one computer system to another while measuring the end-to-end performance of the operation (**-endtoend** option)
* A restart monitor used in conjunction with the end-to-end operations to resume a failed copy operation
* Enhanced Heartbeat options
* A Results Manager that is used to collect and display results from all I/O threads
* The ability to selectively reformat the output fields (**-outputformat** option)
* Ability to specify the exact number of bytes to transfer (**-bytes** option)
* I/O Operation Ordering is more tightly controlled and can be specified as serial, loose, or none. See section on I/O Operation Ordering for more details.
* Addition of XDDMCP, a command line interface that makes use of the end-to-end option for transferring data over a network

# Compiling and Installing XDD

## XDD Source Code Overview

The XDD distribution comes with all the source code necessary to install XDD and the companion program, XDDMCP. For UNIX systems, a Makefile is used to build any of these programs. The XDD distribution directory hierarchy has the following structure:

## Build Notes

### Configure and Make

The process has been simplified for most XDD-supported operating systems.

From the directory that contains the script called “configure” run the following command:

./configure

This will build the file called “Makefile” that contains all the OS-dependencies adjusted for the build.

At that point run any of the following:

make

or

make install

or

make clean ; make

A simple ‘make’ will simply make the XDD and XDDMCP executables in the local XDD “bin” subdirectory.

The ‘make install’ will attempt to copy the XDD and XDDMCP executables into “/bin” on the local system. The user must have root privileges in order to perform a ‘make install’.

The ‘make clean’ is used to remove all object files and previously compiled executables from the local XDD “bin” directory. It must be followed by an explicit ‘make’ or ‘make install’ to recompile the executables.

### Supported Operating Systems

XDD is currently supported on a number of mainstream operating systems with limited support for some legacy operating systems. These operating systems include:

* Linux (kernel versions 2.6 and above only)
* AIX™ from IBM
* Mac OS X

#### Linux

Make uses either the cc or gcc compiler, and associated libraries. Insure that the pthread libraries are installed for correct compilation and linking. This is normally the case for most modern Linux installations.

At the successful conclusion of the build process, the “bin” directory will contain the following files:

* “xdd”
* “xddmcp”
* “xddmcp-server”
* “xddprof”
* “xddcp”
* “xddft”
* “xdd-getfilesize”
* “xdd-gethostip”
* “xdd-read-tsdumps”
* “xdd-truncate”

#### AIX

As in Linux, ensure that the pthread libraries are installed.

#### Mac OS X

Building on the Darwin platform is very similar to Linux, except make uses the cc, gcc, or clang compiler, and associated libraries.

# Theory of Operation

## XDD Program Structure Overview

XDD consists of a main XDD thread and one or more Target Threads that are created to perform the actual I/O operations on a specific target device or file. The following terms are important to understand in order to make sense of the program and source code structure:

* The time from the invocation of the XDD command until it returns control to the shell is referred to as a run
* Each run consists of one or more passes
* Each pass consists of one or more I/O operations

For example, the following simple XDD command will perform an XDD run that consists of 5 passes of 128 write I/O operations (4KB each) to be performed on the specified target called “testfile”. The **-verbose** option will cause XDD to display intermediate results for each pass as the pass completes.

xdd –op write –target testfile -reqsize 4 –numreqs 128 –passes 5 –verbose

The overall program structure has been simplified with several basic parts as shown:

*Worker*

*Thread*

target\_pass subroutine

Target Thread

## Basic Operation

XDDis a program that performs data transfer operations between memory and a disk device or file (or multiple disk devices / files), and collects and displays performance information about these I/O operations. XDD creates one Target Thread for every device or file under test. Each XDD Target Thread creates one or more Worker Threads and issues I/O tasks to available Worker Threads. A Worker Thread given an I/O task will issue an I/O operation to the target, which is either a disk storage device or a file. Each I/O operation is either a read or a write operation of a fixed size known as the request size (**-reqsize** option). Optionally, the Worker Thread will also perform a send or receive operation to a network socket if the **-endtoend** option has been specified.

An XDD run consists of several passes, the number of which is specified by the **–passes** option. Each pass will execute some number of I/O requests on the specified target(s) at the given request size. In general, each pass is identical to the previous passes in a run with respect to the request size, number of requests to issue, and the access pattern unless certain options are used to alter these parameters between passes. Passes are run one after another with no delays between passes.

Multiple passes within an XDD run are used to determine the reproducibility of the results. In theory, the results from each pass in an XDD run should be the same or at least very close to the same given the same set of run-time parameters.

Upon invocation, XDD will parse the command line arguments and spawn one Target Thread for each target specified for the run. The Target Thread will spawn additional Worker Threads according to the **-queuedepth** option, or the **–t** option in XDDMCP. After the Target Threads have been initialized they will wait at an initialization barrier until XDD’s main thread has spawned and initialized the Results Manager thread, and optionally the Heartbeat and/or the Restart Monitor threads. The Heartbeat and Restart Monitor threads will be initialized if the **–heartbeat** and **–restart** options, respectively, are used.

## Command Line Options and the Setup File

XDD has a command line interface that requires all the run-time parameters to be specified either on the XDD invocation command line or in a setup file. The format of the setup file is similar to the XDD command line in that the options can be simply entered into the setup file the same as they would be seen on the command line. The following example shows an XDD invocation using just the command line and the same invocation using the setup file along with the contents of the setup file.

Command line:

xdd –op read –targets 1 /dev/scsi/disk1 –reqsize 8 –numreqs 128 –verbose

Using a Setup file:

xdd –setup XDDrun.txt

Where the setup file “XDDrun.txt” is an ASCII text file that contains the following:

–op read –targets 1 /dev/scsi/disk1

–reqsize 8

–numreqs 128

–verbose

## Operation Specification and Request Sizes

The operation to perform is specified by the –**op** option with the parameter of either “read” or “write”. This version of XDD will mix read and write operations within an XDD run according to the read/write ratio set using the **–rwratio** option. Each read/write operation will transfer a given amount of data known as the request size. The request size is specified by the –**reqsize** option in units of blocks. A block is, by default, 1024 bytes but can be specified to be any positive integer value by using the **–blocksize** option. The block is used as the basic unit for all other options requiring a data size, unless otherwise noted. It is recommended that the block size be specified as numbers that are integer multiples of 512 bytes (i.e. 1024, 2048, 5120, …etc.), since this tends to be the predominant sector size for most storage devices at the current time.

## Target Specification and Multiple Target Synchronization

All requests are sent to a target, which can be either a disk device or a file. A single XDD run can operate on a single target or multiple targets simultaneously. Target names are specified using the **–targets** option. It is always necessary to specify the number of targets followed by the individual target names. In order to simplify the list of target names, the **-targetdir** option can be used to specify the directory where the target devices or files reside.

The execution of the XDD threads on multiple targets is synchronized through the use of barriers. Each XDD Target Thread initializes itself and waits for all the other XDD Target Threads to reach the same starting point. Once all the XDD Target Threads reach this point, they are all released simultaneously. Each XDD Target Thread will run independently until it has either completed all of its requested I/O operations or reached its time limit (as specified by the **–timelimit** input parameter). At this point the XDD Target Threads synchronize with the Results Manager thread that will collect and display the intermediate results for the pass that just completed. After displaying the results the Target Threads will be released to begin another pass or to perform their end-of-run processing.

It is also possible to resynchronize the Target Threads at specific points within a pass by using the **-syncio** and/or **-syncwrite** options. The -**syncio** option instructs each of the Target Threads to synchronize after some number of I/O operations specified as an argument to this option. The default is to synchronize only at the beginning of each pass. The -**syncwrite** option is used to synchronize buffered write operations at the end of each pass by flushing all the file system data buffers to the physical media. This option is not able to flush the cache buffers on the disk controllers or disk drives themselves.

## XDD Run Time or Run Length

It is necessary to specify a limit on how long XDD will run. There are several ways to do this. First, it is possible to explicitly specify the number of transfers to perform using the **-numreqs** option. This specifies the number of read or write calls to make for a single pass. It is also possible to simply specify the number of megabytes to transfer using the **-bytes**, **-kbytes**, or **-mbytes** option. It is important to note that a kilobyte, megabyte, gigabyte in this context is a power of two: 1024, 1048576, or 1073741824 bytes. Finally, it is possible to specify a time limit for each XDD pass using the **-timelimit** option. This will cause each pass to terminate after the specified number of seconds has elapsed, after executing the specified number of requests, or after transferring the specified number of megabytes, whichever occurs first.

## I/O Range

For random I/O operations, each Target Thread performs its I/O operations within a certain consecutive range of blocks on the target (see Figure 1). The range is either implicitly specified as the number of megabytes to transfer, or explicitly specified using the **-range** option. For example, if the user specifies 2048 purely sequential data transfers at a request size of 128 blocks each, then the range will be implied as 262144 blocks (2048 \* 128). However, if the user wants to transfer the same 2048 requests randomly over a 2 GB area (or range) on the target then the range needs to be explicitly specified as 268435456 blocks or 2 GB.

The beginning of the I/O range defaults to the beginning of the target whether it is a device or a file. It is possible to specify a starting offset such that the I/O range begins at some distance into the target (see Figure 1). This can be done several ways. First, the **-startoffset** option can be used to start I/O operations at some distance into the target for an XDD run. The **-passoffset** can be used to incrementally move the starting offset further into the target device on subsequent passes within an XDD run by some specified number of blocks. Finally, the **-targetoffset** can be used to move the starting offset into the target device by a number of blocks that is determined by the target offset value times the target’s ordinal number. For example, given an XDD run on 4 targets with a target offset of 1024 and a starting offset of 0, I/O will start at block 0 on target 0, block 1024 on target 1, block 2048 on target 2, and block 3072 on target 3. It is worth noting that target ordinal numbers start at 0.

## Access Patterns

The range can be specified to start anywhere within the target so long as care is taken to insure that the end of the range is still within the confines of the target. This is particularly true when randomly accessing blocks within a target that is a regular file. Within the I/O range, data access patterns can be either

1) purely sequential

2) staggered sequential

3) purely random

Starting offset

I/O Range

Entire disk or file

**Figure 1. Example of the I/O range and the offset.**

Purely sequential I/O is the default access pattern. This pattern accesses consecutive data blocks starting at the beginning of the range to the end of the range.

A staggered sequential access pattern also starts at the beginning of the range and ends at the end of the range but only transfers every Nth data block. This access pattern is specified by the **-seek staggered** parameter. For example, if reading 256 MB within a 2 GB range, the 256 MB is spread evenly over the entire 2GB range. Therefore, for a given request size, every 8th data block is read with gaps of unread data in-between since 256 MB is 1/8th of 2 GB.

A purely random access pattern is one that accesses data blocks at the specified request size randomly throughout the specified range. It is important to note that for a given range and given request size, the same random pattern is generated for each successive pass/run in order to yield reproducible results. The random access pattern used for each pass, within a run, can be changed by using the **-randomize** parameter. And, the random access pattern used for each run can be changed by using the **-seek seed** parameter.

It should be noted that when running XDDon a regular file especially in random access mode, the direct I/O option (**-dio**) should be used to avoid using the file system buffer cache. The file system data caching mechanisms will produce misleading (and non-reproducible) results. See the section on Reading and Writing Devices and Files for more information.

Finally, it is possible to specify a “null” access pattern whereby the first block in the range is continually accessed. This is accomplished by specifying the **-seek none** option. This is useful for testing the effectiveness of caching algorithms and/or the speed of the cache on the target device or file.

## Reading and Writing Devices and Files

XDD reads and writes devices and files. When running XDD on regular files, the target files tend to be large and can be left behind after the testing. The **-deletefile** option will remove a target file at the conclusion of the XDD run. This option is not recommended when running XDD on device files for obvious reasons.

As previously implied, sequentially reading data is relatively straight forward: start at the beginning of the device or file plus the starting offset and read data until the end of the data range is reached or the time limit expires. When reading a file it is assumed that the file is at least as large as the desired data range so that the read does not go past the end of the file. If the file is smaller than the desired or specified data range then a warning message is displayed regarding this condition.

Reading a file on most systems uses a file system buffer cache. In this mode, data is read into a file system buffer and then copied into the XDD I/O buffer by the CPU. This can cause two different problems with respect to the performance results. First, the maximum bandwidth for reading data for the first time is limited to the memory copy speed of the processor that can be much lower than the true bandwidth of the disk I/O subsystem under evaluation. Secondly, if all the data from the file fits into the file system buffer then subsequent reads to the same file will be satisfied by copying the data from the file system buffer cache and not from the actual disk subsystem. In this case, XDD is reporting the memory copy performance rather than the speed of the disk subsystem. To avoid this problem the direct I/O, **-dio**, option can be used to bypass the use of the file system buffer cache and forcing all read requests to access the disks. There are certain I/O request size and alignment restrictions that must be observed in order to use direct I/O and these restrictions are operating system dependent. These restrictions essentially state that the I/O requests must start on a page-size boundary and must be in units of the native block size of the underlying file system.

Writing files is very similar to reading files except for the data transfer direction. However, writing files has additional complexities that can affect the performance results. This is due primarily to the allocation mechanisms used by the file system manager to allocate disk space when writing a file for the first time. Generally speaking, each write operation will cause the file system manager to allocate space on the disk to accommodate the data being written. These allocation operations can require additional accesses to the disk that are invisible to XDD, but do show up in the performance results as essentially slower write operations. In order to minimize the effects of this allocation process, the **-preallocate** option can be used to do the entire allocation operation before the file write operations take place. The argument to this option specifies the number of blocks to pre-allocate before the write operations start. This number should be greater than or equal to the number of blocks that will be in the file. (For more information, see section 4.11 Preallocation)

Write performance results can also be affected by the file system buffer cache. When writing a file the data is first copied into the file system buffer and later actually written to the physical disk subsystem. As in the case of the read operation, it is possible to use the **-dio** option to avoid this data copy operation. The same request size and alignment restrictions apply to direct I/O in this case as in the read case.

One last note about writing to a target device or file. The data pattern to be written to the target can be specified by the **-datapattern** option. This will fill the data buffer with a single character data pattern, a random data pattern, or one of several predefined data patterns depending on the argument to this option. No data verification is done with the data that is written out using this pattern. It is simply an option to use in case it is necessary to find out where XDD has written.

Finally, there is a **-memalign** option to re-align the internal data buffer within memory if necessary. This is used more for testing computer system memory performance rather than I/O performance of a target and, therefore, may have limited usefulness. It is only mentioned here for completeness.

## Processor Allocation and Priority Assignment

On multi-processor systems it is possible to assign XDD threads to specific processors. This is accomplished with the **-processor**, **-singleproc**, **-roundrobin** option. The –processor option allows the explicit assignment of a processor to a specific XDD Target Thread and/or Worker Threads. The **-singleproc** option will assign all XDD Target Threads to a single processor specified as an argument to this option. The **-roundrobin** option will distribute the XDD threads across M processors, where M is the number specified as an argument. M should be less than or equal to the number of processors in the system. The processor-numbering scheme used is 0 to N-1 where N is the number of processors in the system. For example, if there are five XDD threads running on a computer with eight processors, then the round robin processor assignment will assign threads 0 thru 4 on processors 0 thru 4. However, if there were only two processors on the computer, then XDD threads 0, 2, and 4 will be assigned to processor 0 and threads 1 and 3 will be assigned to processor 1.

The priority of each thread defaults to the “normal” priority on the system. It is possible to increase the priority to a maximum level by using the **-maxpri** option. Maximizing the runtime priority of the XDD threads decreases the effects on the I/O performance of other processes running on the system. It is also possible to lock the XDD process and I/O buffers using the **-processlock** option. This is done to prevent the XDD process or any of its I/O buffers from being paged or swapped out of the system. The **-maxall** option is a shortcut for specifying both **-maxpri** and **-processlock**.

## Preallocation

Preallocation is used to tell the file system manager to allocate a specified amount of space in the file system before the actual write operations occur. The advantage to this is limiting the metadata overhead involved with allocations during write operations, but also to limit the number of file system extents used to hold file data. Normally, when using direct I/O, each write operation can have the effect of allocating a new file extent resulting in potential file fragmentation. The use of the **-preallocate** option circumvents this problem by allocating all the extents for the file ahead of time.

However, even though the space is allocated, it does not mean that the file size is that large. In fact, it is possible to preallocate, say 1GB of space and still have a zero length file.

Here is a synopsis of the various pre-allocation idiosyncrasies:

* Create a 0-length file
* Call xfsctl to reserve 16MB
* One 16MB extent is created for the file
* Write 8KB into the file
* Close the file

The file is now 8KB long but takes up 16MB of physical space (one extent).

* Open the same file
* Call xfsctl to preallocate 64MB
* Close file

The file now takes up two extents:

* The first 16MB extent
* A second 48MB extent

The file is still 8KB in length and the 8KB are still valid but the file now occupies 64MB of space spanning two extents

* Open the file
* LSeek +32KB into the file
* Write 4KB
* Close file

The file is now 36KB in length

* The first 8KB are valid
* Bytes 8KB to 32KB are NULL (as expected)
* Bytes 32KB to 36KB are valid

The file still retains its two extents (16MB+48MB)

* Open file
* LSeek +8KB into the file
* Write 3\*8KB (three separate calls to write)
* Close file

The file is still 36KB

* The first 8KB is still valid
* Bytes 8KB-32KB are new and correct
* Bytes 32KB-36KB are still valid

Note: All I/O was done with direct I/O enabled. I also ran similar tests without direct I/O and the results were the same. In conclusion, direct I/O does not seem to affect the outcome because the space allocation has already been done. With respect to allocation, the only difference between direct I/O and buffered I/O is when the allocation is done. For direct I/O allocations are performed for each and every I/O in the order in which they are done. For buffered I/O the allocations are done at some later time based on the assumption that many adjacent blocks can be grouped into a single largish allocation rather than a bunch of smallish ones as in the direct I/O case.

## I/O Time Stamping

While running, each XDD thread has the option to enter time stamp information into a table that is later written to a file for further analysis. Each I/O operation is time stamped before the operation starts and just after the operation ends. The time stamp table contains all the information necessary to understand when I/O operations started, ended, the block location being accessed, and the amount of data transferred. The time stamps themselves are taken from the system's high-resolution timer and re-normalized in units of picoseconds so that this data can easily be correlated with time stamp data from the other associated XDD output files.

The format of the time stamp output file contains a header followed by the time stamp data.

In addition to a binary output file, the time stamp table information can be dumped in ASCII readable text. There are several options for ASCII output including a summarized and a detailed output specified by the **-output summary** and **-output detailed** options. Section 9 includes examples of the summarized and detailed outputs of the time stamp table. The time stamp ASCII output file names have a “.csv” file extension so that it can more easily be read by a spreadsheet program such as Excel®.

# Running XDD Programs

XDD programs include XDD itself, the XDDCP, and the XDDMCP programs. Example command lines are given in Section 11, and are also contained in a file called tests.txt in the distribution bin directory.

## XDD Command Line Arguments Synopsis

xdd

-blocksize *[target <target#>] number\_of\_bytes\_per\_block*

-bytes *[target <target#>] number\_of\_bytes\_to\_xfer\_per\_pass*

*-*combinedout *filename*

-createnewfiles *[target <target#>]*

*-*csvout *filename*

-datapattern *[target <target#>]*

*character\_pattern*

**random**

**sequenced**

**prefix** *<hex digits>*

**inverse**

**ascii** <*string*>

**hex** <*hex digits 0-9, a-f or A-F*>

**replicate**

**lfpat**

**ltpat**

**cjtpat**

**crpat**

**cspat**

-deletefile *[target <target#>]*

-dio *[target <target#>]*

-dryrun

-endtoend *[target <target#>]*

**issource**

**isdest**

**port** *#*

**portcount** *#*

**dest <***hostname:base\_port,number\_of\_ports>*

-errout *filename*

-flushwrite *#ops*

-fullhelp

-heartbeat *[target <target#>]*

*<#seconds>*

**hostname**

**target**

**tod**

**elapsed**

**lf**

**output** *<filename>*

**bw**

**ops**

**bytes**

**kbytes**

**mbytes**

**gbytes**

**iops**

**pct**

**etc**

-help *option\_name*

-id

**commandline**

**<***id\_string>*

-kbytes *[target <target#>] number\_of\_kilobytes\_to\_transfer*

*-*lockstep *<master\_target> <slave\_target> <when> <howlong> <what> <howmuch>*

*<startup> <completion>*

-looseordering *[target <target#>]*

-maxall

-maxerrors *number\_of\_errors*

-maxpri

-mbytes *[target <target#>] number\_of\_megabytes\_to\_transfer*

-memalign *[target <target#>] alignment\_value\_in\_bytes*

-minall

-nomemlock

-noordering *[target <target#>]*

-noproclock

-numreqs *[target <target#>] number\_of\_requests\_to\_perform*

-op *[target <target#>] rea*d|*write|noop*

-ordering *[target <target#>]*

**storage** *<serial\_or\_loose\_or\_none>*

**network** *<serial\_or\_loose\_or\_none>*

**both** *<serial\_or\_loose\_or\_none>*

-output *filename*

-outputformat

**add** *<format\_id\_string>*

**new** *<format\_id\_string>*

-passdelay #.#*seconds*

-passes *number\_of\_passes*

-passoffset *[target <target#>] offset\_in\_blocks*

-preallocate *[target <target#>] number\_of\_bytes*

-pretruncate *[target <target#>] number\_of\_bytes*

-processlock

-processor *[target <target#>] processor\_number*

-queuedepth *[target <target#>] number\_of\_commands\_per\_target*

-randomize *[target <target#>]*

-recreatefiles *[target <target#>]*

-reopen *[target <target#>]*

-reportthreshold *[target <target#>] #.#*

-reqsize *[target <target#>] number\_of\_blocks*

-restart *[target <target#>]*

**enable**

**frequency** *<seconds>*

**file** *<name\_of\_restart\_file>*

**offset** *<offset\_in\_bytes>*

-roundrobin

*number\_of\_processors*

**all**

-runtime #.#*seconds*

-rwratio *[target <target#>] readwriteratio*

-seek *[target <target#>]*

**save** *<filename>*

**load** *<filename>*

**disthist** <*number\_of\_categories>*

**seekhist** *<number\_of\_categories>*

**random**

**range** *<number\_of\_blocks>*

**stagger**

**interleave #**

**seed** *<seed\_value>*

**none**

-serialordering *[target <target#>]*

-setup *setup\_filename*

-sgio

-sharedmemory *[target <target#>]*

-singleproc *processor\_number*

-startdelay *[target <target#>] #.#seconds*

-startoffset *[target <target#>] starting\_block\_number*

-starttime *#seconds*

-stoponerror

-syncio *number\_of\_ops*

-syncwrite *[target <target#>] number*

-target  *filename*

-targetdir *[target <target#>] directoryname*

-targetoffset *[target <target#>] offset\_in\_blocks*

-targets *N name1 name2 … nameN*

-targetstartdelay *[target <target#>] #.#seconds*

-throttle *[target <target#>]*

**ops** *operations/sec*

**bw** *megabytes/second*

-timelimit *[target <target#>] #.#seconds\_per\_pass*

-timeserver

**host** *hostname*

**port** *port#*

**bounce** *bounce\_count*

-timestamp (or –ts) *[target <target#>]*

**output** *output\_filename\_prefix*

**summary**

**detailed**

**normalize**

**summary**

**wrap**

**oneshot**

**size #**

**triggertime** *#seconds*

**triggerop** *operation#*

**append**

**dump** *dump\_filename\_prefix*

-verbose

-version

## Detailed Option Specifications

**-blocksize** specifies the number of bytes per block. This defaults to 1024 bytes per block. Block sizes must be powers of 2 or the results are unpredictable.

**-bytes** specifies the number of bytes to transfer per pass. This can be any positive number up to 264-1. See also: -**kbytes** and **-mbytes**.

**-combinedout** *filename*will append just the combined results output to the file specified by filename. This allows for collecting the combined performance data from multiple runs in a single file. This does not include error messages. See the **–errout** option for more information on redirecting error output.

Scope: Global

-**createnewfiles** will cause new target files to be created for each pass in an XDD run. Each new file has the file name that was given as the target file name but it is appended with a number that represents the pass in which it was created. It is not a good idea to run this on a target file because the new target will get recreated as a regular file and not a special device file.

**-csvout** *filename*will send all the results output to the file specified by filename similar to the **–output** option. The difference between csv output and the normal output is that this is a comma separated values file that is directly importable into MS Excel. This does not include error messages. See the **–errout** option for more information on redirecting error output.

Scope: Global

**-datapattern** specifies either a single byte data pattern character or a special operator as described below.

* If the operator “**random**” is specified then a random data pattern is generated in the entire I/O buffer.
* If the operator “**sequenced**” is specified, then the data pattern will be sequential 64-bit integers starting with the current block address times the size of a 64-bit integer. It should be noted that writing this sequenced pattern to the device will result in additional CPU overhead that may affect overall performance. Similarly, for read operations, if the “**sequenced**” data pattern is specified, the data is checked to see if the data read is in fact what was expected.
* If the word “**prefix**” is specified for the pattern then the specified hex digits will be placed in the upper N bits of the 64-bit pattern
* If the operator “**inverse**” is specified for the pattern then the actual pattern will be the 1's compliment of the specified pattern
* If the word “**ascii**” is used as the data pattern, then the data pattern will be the string specified after the “**ascii**” operator. The string is only written once at the beginning of the I/O buffer and is not repeatedly copied throughout the I/O buffer.
* If the word **“hex”** is specified then the following hex characters <0-9,a-f or A-F> are used as the pattern
* If the word “**replicate**” is specified then whatever pattern was specified is replicated throughout the buffer
* If the word “**lfpat**” is specified then the low-frequency 8B/10B pattern is used
* If the word “**ltpat**” is specified then the low-transition 8B/10B pattern is used
* If the word “**cjtpat**” is specified then the compliant jitter 8B/10B pattern is used
* If the word “**crpat**” is specified then the compliant random 8B/10B pattern is used
* If the word “**cspat**” is specified then the compliant sequential 8B/10B pattern is used
* The default data pattern is all binary 0's

**-deletefile** will cause the target file to be deleted at the end of a run. To have files deleted and recreated between passes, see the **–recreatefiles** option.

**-dio** will turn on the direct I/O option when accessing a regular file. This option cannot be used when accessing special device files. Certain blocksize, request size, and alignment restrictions apply and will cause problems if the wrong combination of block size, request size, and offsets are chosen.

**-dryrun** will exit after parsing the arguments and displaying initialization information.

**-endtoend** specifies that the target device will participate in an end-to-end operation. There are two sides to an end-to-end (aka e2e) operation: the source and the destination. It takes two concurrent instances of XDD to perform an e2e operation: one running on the source side and one running on the destination side. The instance of XDD on the source side of the e2e operation will read the specified target file/device and transfer the data over a TCP/IP socket to the instance of XDD on the destination side. The destination instance of XDD will receive the incoming data and write it to its specified target file/device. See section on Examples for more information.

The e2e option has the following operators:

* If the operator “**issource**” is specified then this particular instance of XDD assumes that it is on the source side of an e2e operation.
* If the operator “**isdest**” is specified then this particular instance of XDD assumes that it is on the destination side of an e2e operation.
* The operator “**dest** *hostname[:baseport[,#ports]]*” specifies the name of the host on the destination side of an e2e operation. The hostname can be specified as a name or an IP Address. Optionally, the base port to use and the number of ports to use starting from the base port number can also be specified as indicated. It is necessary to specify this address on both instances of XDD, source side and destination side, in order to explicitly define the correct network interface to use for the data transfer. Furthermore, if the base port and number of ports is specified, the queue depth will be implied thereby eliminating the need to specify the **-queuedepth** option.
* The “**port** #” defines the specific port to use for the data transfer. In the event that multiple threads are used on each side of the e2e operation (i.e. **–queuedepth** 48) then the port number becomes the base port number and is incremented by 1 for each subsequent Worker Thread. For example, if **–queuedepth** 4 is specified along with **–e2e** **port** 3010 then the port assignments will consume port numbers 3010, 3011, 3012, and 3013 for Worker Threads 0, 1, 2, and 3, respectively. For this type of operation it is important to choose a base port number that has free ports within the range of ports that will be consumed.

**-errout** filenamewill send all the error message output to the file specified by filename. This does not include normal results output. See the **–output** option for more information on redirecting results output.

Scope: Global

**-flushwrite** will force a sync() operation to occur every so many write operations as specified by the associated argument to this option.

**-fullhelp** will display a list of these options anda one-line explanation of each.

Scope: Global

**-heartbeat** *[target <target#>]* **<***suboption***>**

Where “*suboption*” can be one of the following:

* *#seconds* will display pass number and requested information for each target I/O thread every N seconds where N is specified as *#seconds*.
* **hostname** *-* Will include the name of the host machine that the instance of XDD is running on. The “hostname” appears at the beginning of the output line and is prefixed by “HOST”.
* **target** *-* Will include the target number before each output grouping. The target number is followed by “tgt” to indicate that it is the Target number.
* **tod** - Will include the current time of day on each heartbeat update. The time of day appears at the beginning of the output line and is prefixed by “TOD”.
* **elapsed**- Will include the elapsed number of seconds for the run on each heartbeat update. The elapsed seconds appears at the beginning of the output line and is prefixed by “ELAPSED”.
* **lf**- Will include a line feed character at the end of each heartbeat output line.
* **output** *<filename\_prefix>* - will send all heartbeat updates to a file with a name of the form “*filename\_prefix*.T####.csv” where “*filename\_prefix*” is the argument to the **–hb output** option and T#### is the letter “T” followed by a four-digit target number. The heartbeat output is presented as an ASCII string of comma-separated values hence the filename will end with a file extension of “.csv” so that it can be more easily imported into a spreadsheet. Only the heartbeat values for the specific Target is written to the heartbeat output file.
* **bw**- Will display the aggregate bandwidth of each Target Thread on each heartbeat update. The bandwidth is expressed in units of 106 or MB per second.
* **ops**- Will display the total number of completed I/O operations on each heartbeat update.
* **bytes**- Will display the current total number of bytes transferred on each heartbeat update.
* **kbytes**- Will display the current total number of KB (103 bytes) transferred on each heartbeat update.
* **mbytes**- Will display the current total number of MB (106 bytes) transferred on each heartbeat update.
* **gbytes**- Will display the current total number of GB (109 bytes) transferred on each heartbeat update.
* **iops**- Will display the current aggregate I/O operations per second on each heartbeat update.
* **pct**- Will display the current percent of I/O operations completed on each heartbeat update.
* **etc**- Will display the current estimated time to completion on each heartbeat update.

Scope: Target Specific

**-id** allows the user to specify an ASCII string to be displayed in the output in order to identify the run. If the word “**commandline”** is used as the character string then the input command line is used as the id. Multiple instances of the **-id** option will concatenate each specified id to the previous one.

Scope: Global

**-kbytes** specifies the integer number of kilobytes to transfer on each pass. In this case, one kilobyte is equal to 1024 bytes. If the –**numreqs** option is also specified, –**numreqs** takes precedence.

See also: -**mbytes** and **-bytes**.

**-lockstep** *<master\_target> <slave\_target> <when> <howlong> <what> <howmuch> <startup> <completion>*Where

*master\_target* is the target that tells the slave when to do something. *slave\_target* is the target that responds to requests from the master. *when* specifies when the master should tell the slave to do something.

The word "*when*" should be replaced with the word:   
            "**time**"   
            "**op**"  
            "**percent**"  
            "**mbytes**"  
            "**kbytes**"

*howlong* is either the number of seconds, number of operations, ...etc.

* The interval time in seconds (a floating point number) between task requests from the master to the slave. i.e. if this number were 2.3 then at every 2.3-second interval the master would request the slave to perform its task.
* The operation number that defines the interval on which the master will request the slave to perform its task. i.e. if the operation number is set to 8 then upon completion of every 8 (master) operations, the master will request the slave to perform its task.
* The percentage of operations that must be completed by the master before requesting the slave to perform a task
* The number of megabytes (1024\*1024 bytes) or the number of kilobytes (1024 bytes

*what* is the type of task the slave should perform each time it is requested to perform  
 a task by the master. The word "what" should be replaced by:  
               "**time**"   
               "**op**"  
               "**mbytes**"  
               "**kbytes**"   
*howmuch* is either the number of seconds, number of operations, ...etc.

* The amount of time in seconds (a floating point number) the slave should run before pausing and waiting for further requests from the master.
* The number of operations the slave should perform before pausing and waiting for further requests from the master.
* The number of megabytes (1024\*1024 bytes) or the number of kilobytes (1024 bytes) the slave should transfer before pausing and waiting for further requests from the master

*startup* is either "**wait**" or "**run**" depending on whether the slave should start running  
               upon invocation and perform a single task or if it should simply wait for the  
               master to request it to perform its first task.

*completion* - in the event that the master finishes before the slave, then the slave will  
               have the option to complete all of its remaining operations or to just stop at  
               this point. This should be specified as either "**complete**" or "**stop**".

**-looseordering** will cause the ordering of I/O operations issued to a target to be loosely ordered. This is equivalent to specifying “**-ordering** **storage** **loose**”. See the section on I/O operation ordering for more information.

Scope: Target Specific

**-maxall** will set the **-maxpri** and **-plock** options.

Scope: Global

**-maxerrors** *number\_of\_errors* will limit the number of errors to *number\_of\_errors* so as not to clutter up the output with endless lines of errors. Once this limit has been reached the XDD pass will end.

Scope: Global

**-maxpri** will set the priority of all XDD threads to maximum. NOTE: Use of this option can result in system hangs due to deadlocks with other system functions.

Scope: Global

**-mbytes** specifies the integer number of megabytes to transfer on each pass. In this case, one megabyte is equal to 1024\*1024 or 1048576 bytes.

If the –**numreqs** option is also specified, –**numreqs** takes precedence. See also: -**kbytes** and **-bytes**.

**-memalign** *memory\_alignment\_value\_in\_bytes* will cause the internal memory address alignment of the I/O buffer to be offset by the number of bytes specified as *memory\_alignment\_value\_in\_bytes.* The I/O buffer is normally page aligned.

**-minall** will set the **–noproclock** and **-nomemlock** options.

Scope: Global

**-nomemlock** will prevent the XDD memory buffers from being locked so that they can be paged or swapped out.

Scope: Global

**-noordering** will cause the ordering of I/O operations issued to a target to be ignored. This is equivalent to specifying “**-order** storage none”. See the section on I/O operation ordering for more information.

**-noproclock** will prevent XDD process from being locked in memory so that it can be paged or swapped out.

Scope: Global

**-numreqs** specifies the integer number of request size requests to perform on each pass.

If the –**mbytes** or –**kbytes** option is also specified, –**numreqs** takes precedence.

**-op** specifies the operation to perform: either “**read**”, “**write**”,or“**noop**”may be specified.

**-ordering** will cause the ordering of I/O operations issued to a target to be either “**serialized**”, “**loose**”, or “**none**”. Ordering for the “**storage**” or ”**network**” side of a Worker Thread I/O can be specified. See the section on I/O operation ordering for more information

**-output** *filename*will send all the results output to the file specified by filename. This does not include error messages. See the **–errout** option for more information on redirecting error output.

Scope: Global

**-outputformat** will either “**add**” items to the XDD output lines or create a “**new**” output line. See section on Output Format for more information.

Scope: Global

**-passdelay #.#** where #.# is the number seconds to delay between passes. The pass delay occurs after each pass except for the last pass of a run.

Scope: Pass delay is a global variable and affects all targets.

Notes: The pass delay time is not included in the COMBINED rate calculations. For example, given a 5-pass run with a pass delay specified as 7.3 seconds and a run time of 3 seconds per pass, each pass will take 3+7.3=10.3 seconds except for the last pass which would only take 3 seconds. Hence the total run would take 4\*10.3 + 3 = 44.2 seconds. However, the time used to calculate the bandwidth and IOP rates is 5\*3=15 seconds, which does not include the pass delay time.

See also: -**startdelay** and **-targetstartdelay**.

**-passes** #, where # is the number of passes to perform.

Scope: Global

**-passoffset** *#*,where # is the number of blocks to offset for each pass.

**-preallocate** # will preallocate # bytes. This is used when writing to a target that is a regular file. This option has no effect when reading or when the target is a real device. See section on “Preallocation” for more details.

-**pretruncate** # will truncate the target file to the specified number of bytes before beginning transfer.

**-processlock** will lock the XDD process in memory so that it cannot be paged or swapped out. This is useful on a crowded system.

Scope: Global

**-processor** *target**target\_number processor\_number* This option allows an XDD thread for a specific target to run on a specific processor. The *XDD* Target Thread for target *target\_number* is assigned to processor *processor\_number.*

**-queuedepth** # specifies the number of commands to send to each target at one time. This exercises the command queuing capabilities of a storage device or, if the target is a file, it will mimic parallel I/O – multiple readers/writers to a single file.

**-randomize** will cause the seek locations to be randomized between passes.

-**recreatefiles** will cause the target files to be closed, deleted, and re-created for each pass in an XDD run. It is not a good idea to run this on a target device file because the target will get recreated as a regular file and not a special device file.

-**reopen** will cause the target file to be closed and re-opened for each pass in an XDD run.

-**reportthreshold** will report the byte location of the operation that exceeded the specified threshold time.

**-reqsize** specifies the number of blocksto transfer where the size of the block is specified by the **-blocksize** parameter.

**-restart** indicates that a previously failed end-to-end operation failed and must be resumed at some point into the file being transferred from a source to a destination (see also –**endtoend**) . The option parameters are:

* **enable** – requires no parameters but tells the end-to-end operation to start the Restart Monitor that will keep track of the most recent successful write operation on the destination side of an end-to-end operation. The restart information is written to a restart file.
* **frequency** *seconds* – specifies the number of seconds between updates to the restart file.
* **file** *filename –*specifies the name of the restart file to generate during an end-to-end operation.
* **offset** *#bytes –*specifies the number of bytes into the source/destination file to use as the point at which to resume the data copy*.*

**-roundrobin** *#*will assign successive XDD threads to processors in a roundrobin fashion across # processors.

**-runtime** *#.#seconds*will cause XDD to terminate completely after it has run for the specified number of seconds.

Scope: Global

Notes: If the **-timestamp** option is also specified the timestamp buffer wrap option is automatically enabled so that the internal timestamp buffer is not overrun.

See also: **-timelimit**.

**-rwratio***#.#*specifies the percentage #.# of operations that should be read operations. The remaining operations will be write operations. For example, specifying a value of 30.2 (i.e. ***–*rwatio** *30.2*) will cause 30.2% of the total number of operations being performed on the target to be read operations and 69.8% of the operations will be write operations. Values less than 0 or greater than 100 are not allowed.

**-seek** specifies a number of parameters that are specific to the access pattern used on each target. The default access pattern is purely sequential. These parameters are:

* **save** *filename -*will save the list of seek locations in an ASCII text file specified by filename*.* This file can later be used by the –**seek load** option.
* **load** *filename -*will load the list of seek locations from an ASCII text file specified by filename*.*
* **range** #blocks - will specify the range in blocks over which to perform random seek operations.
* **random** will generate a random list of locations to access over the range
* **seed** *seed\_value* specifies a seed value to use when generating random locations
* **stagger** will stagger the requests sequentially and evenly over the range
* **interleave** *factor,* where *factor* is the interleave factor to used (see section on parallel I/O and seek interleave).
* **none** will cause XDD to continuously read the startingblock on a target until for a total of **–mbytes** or **–numreqs** of data transfers completes.
* **disthist** *#categories –*will display an ASCII readable histogram of the seek distances.
* **seekhist** *#categories –*will display an ASCII readable histogram of the seek locations.

**-serialordering** will cause the ordering of I/O operations issued to a target to be serialized. This is equivalent to specifying “**-order** storage serial”. See the section on I/O operation ordering for more information.

**-setup** specifies a file that contains commonly used XDD options. This file is read in and the options contained within the file will be inserted into the command line.

Scope: Global

**-sgio** will perform I/O operations to the specified target using the SCSI generic protocol rather than the normal read/write system calls. This is only valid on Linux systems and is used to provide raw-like access to a device.

**-sharedmemory** tells XDD to use a shared memory segment (via shmget/shmat) for the I/O buffer rather than using the normal valloc()/malloc() system calls.

**-singleproc** *processor\_number* will assign all XDD threads to the specified processor.

**-startdelay** *#.#seconds*will cause the Target Threads to start after a specific startup delay specified in seconds.

Scope: Target specific.

Notes: The start delay time does not affect the performance calculations.

See also: **-targetstartdelay**

**-startoffset** specifies the starting block number. This defaults to block 0. The value must be a positive integer.

**-starttime** *global\_time*will cause the target I/O threads to all start at the specified time. The global time is the time value returned by the timeserver and is consistent for all systems using the timeserver. See the **–timeserver** for more information.

Scope: Global

**-stoponerror** will cause all Target Threads to stop processing and exit on the first error encountered.

**-syncio** *number* will cause each of the XDD I/O threads to synchronize every nth I/O operation, where n is specified as *number.*

Scope: Global

**-syncwrite** will cause each of the XDD I/O threads to synchronize write operations at the end of each pass flushing all data to the physical media.

**-target** specifies the device or file to perform the operation on. This is a required parameter, unless the **-targets** option is used instead.

**-targetdir** specifies the name of the directory to be pre-pended to the target(s). For example, specifying a parent directory of “/dev/rdsk/” (i.e. **–targetdir** /dev/rdsk/ ) and target names of “dks1d2s0” and “dks7d3s0” will cause I/O to be directed to “/dev/rdsk/dks1d2s0” and “/dev/rdsk/dks7d3s0”, respectively. It is important to remember to put the trailing slash (“/”) at the end of the parent directory name.

**-targetoffset** specifies the offset in blocks that is used by each XDD process to determine their respective starting locations. The purpose of this is to be able to run multiple XDD threads on a single device but to have each thread start at a different location.

**-targets** must first specify the number of targets (*N*) followed by the target device names or file names of each of the *N* targets. For example, “**-targets** 2 dks1d2s0 dks7d3s” will perform I/O to the target devices “dks1d2s0” and “dks7d3s0”, respectively. In the output reports, these two targets will also be identified as targets 0 and 1 respectively.

**-targetstartdelay** *#.#seconds*will cause each Target Thread to start after some number of seconds has elapsed. The number of seconds that any given target is delayed is the target’s number times the specified value. For example, a value of

“**-targetstartdelay** 1.2”

would allow target 0 to start immediately (0×1.2=0), target 1 to start 1.2 seconds (1×1.2=1.2) from invocation, target 2 to start 2.4 seconds (2×1.2=2.4) from invocation and so on.

Scope: Target specific.

Notes: The target start delay time does not affect the performance calculations. Also, the target start delay is applied on each pass of a run.

See also: **-startdelay**

**-throttle** specifies the I/O operations per second (**ops**) or bandwidth (**bw**) limit for the target(s) depending on which of the two parameters are specified. Valid values are positive real numbers greater than 0.000. The parameters **ops** and **bw** are mutually exclusive and the last one specified for a targets takes precedence. Example usages:

* “**-throttle** **ops** 7.8” will limit all targets to running at 7.8 I/O operations per second.
* “**-throttle** **bw** 87.2” will limit all targets to running at 87.2 megabytes per second.
* “**-throttle** target 2 ops 7.8*”*  will limit only target 2 to running at 7.8 I/O operations per second and all other targets will have no throttle limit unless specified with another **–throttle** option.

**-timelimit** *#.#*will impose a time limit of # seconds on each pass. This value must be a positive integer.

**-timeserver** is used to specify the hostname of the timeserver that acts as the master clock for all timing information when running XDD across multiple machines. This option takes one of three operators as described below. See the **–starttime** option for additional information.

* If the operator “**host**” is specified, then the specified *hostname* is used as the timeserver.
* If the operator “**port**” is specified, then the specified *port\_number* is used to connect to the timeserver.
* If the operator “**bounce**” is specified then the specified *bounce\_count* specifies the number of times to access the timeserver in order to resolve the time delta between the timeserver and the client.

Scope: Global

**-timestamp** or **–ts** <suboption>

Sub options are:

* **summary** will generate a summary of all the I/O operations in the time stamp trace (see figure 4 for details).
* **detailed** will generate a detailed report of each I/O operation in the time stamp trace and a summary report. It is recommended that the **-output** *filename* be specified when using detailedreporting since the trace data can be exceedingly verbose (see figure 4 for details).
* **normalize** will cause all the time stamp values to be normalized to the global clock. This is useful when running XDD on multiple machines so that the events in the timestamp file can be correlated in time.
* **output** *output\_filename* will cause the detailed and/or summary reports to be written to a file of *output\_filename.* The output defaults to standard out.
* **append** will cause the detailed and/or summary reports to be appended to the specified output file*.*
* **dump** *dump\_filename\_prefix* will dump a binary file that contains all the time stamp data. The following structure contains the format of that file.
* **wrap** will cause the internal timestamp buffer to wrap around to the beginning of the buffer if/when the end is reached. This is used in conjunction with the size option described below. The reason for wrapping the timestamp buffer is to essentially capture the most recent I/O operations in a timestamp buffer that is smaller than required for the number of operations being processed by XDD.
* **oneshot** specifies that time stamping will stop once the internal timestamp buffer is full.
* **size #** specifies the size of the internal timestamp buffer in terms of the number of operations that will fit into the buffer. If the size specified is smaller than the number of operations to be performed, the timestamp buffer will be wrapped after the last timestamp buffer entry is used.
* **triggertime #***seconds*will cause timestamping to start at the specified time as measured in global-time seconds.
* **triggerop** *operation#*will cause timestamping to start when the specified operation number is reached.

**-verbose** will display performance information for each pass.

Scope: Global

**-version** will display the version number for this XDD program.

Scope: Global

## Target-specific options

Many of the options can be target-specific. These options are listed with the optional [target <target#>] arguments that immediately follow the option name. The word “target” indicates that the associated option is to be set for the target with a target number of <target#>. Target numbers are from 0 to N-1, where N is the total number of targets being accessed in this run. For example, specifying “**–op** target 3 read” will cause target 3 to issue read operations regardless of what the other targets are doing. This capability is useful for tailoring the behavior of each target in a run to meet specific I/O requirements. For example, it is possible to have a single XDD run accessing several targets using different throttle values so that one target does not overwhelm the others.

It is important to note that the options are evaluated from left to right on the command line, and from top to bottom in a setup file. Also, latter options (to the right) take precedence over prior options (to the left). Take the following command line for example:

xdd –op read –op target 1 write –targets 3 s1 s2 s3 –reqsize 1 –numreqs 7

The “**–op** read” option will cause all three targets (s1, s2, and s3) to perform read operations. However, the **“–op** target 1 write” option will override the read operation for target number 1 (target s2) causing it to perform write operations.

## Lockstep Operations

Lockstep operations are used to simulate the I/O interaction between multiple applications running on a system. For example, one application may be creating files that a second application will use just after their creation – aka the “read-after-write” scenario, where a file is being ingested from a source, written to a file and another application is reading blocks just after they are written in order to process the data as quickly as possible.

An example of this is as follows:

xdd -targets 2 /dev/disk1 /dev/disk1 \

-op target 0 write \

-op target 1 read \  
   -reqsize 1024 -mbytes 2048 \  
   -lockstep 0 1 op 1 op 1 wait complete

This will cause target 0 to write a block and then signal target 1 to start reading. Since they are the same target starting at the same locations, target 1 will be exactly 1 operation behind target 0 all the time. Essentially this tells target 0 to do 1 operation, signal target 1 which will do 1 operation and then wait for target 0 to signal it again and so on.

The current version of XDD only supports lockstep operations on a single computer system.

## I/O Operation Ordering

I/O operation ordering is used to control when I/O operations are issued to the device. Currently only I/O operations directed at the storage system are affected by ordering. The I/O operation ordering options can be used for standard storage system testing or with an end-to-end operation.

The basic idea with I/O operation ordering is to control when I/O operations are issued to a storage system across multiple Worker Threads. There are three modes of ordering defined:

* Serial Ordering – The Worker Thread responsible for I/O operation N+1 is released after the completion of I/O operation N.
* Loose Ordering – The Worker Thread responsible for I/O operation N+1 is released just before I/O operation N is issued.
* No Ordering – There is no coordination or dependencies between Worker Threads as to when the issue their respective I/O operations.

When a Worker Thread gets an I/O task from the Target Thread it will go through a number of housekeeping things before the I/O operation is issued to the storage system. After these initial housekeeping tasks are done the Worker Thread will check to see if it needs to wait for the previous I/O operation before issuing its own I/O operation. If so, then the Worker Thread will enter a semaphore and essentially be put to sleep by the operating system until the Worker Thread responsible for the previous I/O operation releases the sleeping Worker Thread.

When a sleeping Worker Thread is released, it means that the releasing Worker Thread issued a post to the semaphore that the sleeping Worker Thread was waiting for. The semaphore post operation causes the operating system to mark the sleeping Worker Thread “available to run” and places it on a queue of other process threads that are ready to run. At some point in the future the Worker Thread responsible for I/O operation N+1 will continue from where it left off and issue its I/O operation. More importantly though, just before or just after issuing its I/O operation it will issue a semaphore post for the Worker Thread that is responsible for I/O operation N+2. And the ordering scheme continues like this throughout the run.

The semaphores used to implement the ordering schemes are kept in the Target Offset Table, otherwise known as the TOT. There is one TOT per Target Thread. A TOT is an array of TOT entries, each of which contains a semaphore and variables that keep track of I/O operation information for reference purposes. The size of the TOT is always an integer multiple of the number of Worker Threads, as specified by the **–queuedepth** option, or the **-t** option in XDDCP and XDDMCP. The default size of the TOT is 20 times the Queue Depth for reasons that are explained in the section that discusses the TOT in more detail. For purposes of this discussion, when a Worker Thread needs to post the semaphore for the next I/O operation (N+1) it calculates the offset into the TOT, which is a simple function of the I/O operation number mod the number of entries in the TOT. It is left as an exercise for the reader to do the math to see how this works.

### Serial Ordering

Serial ordering causes I/O operation N+1 to be issued only after I/O operation N has completed. This is done to guarantee that I/O operations are performed in a specific order.

Slide1.EMF

### Loose Ordering

Loose ordering is similar to serial ordering but is not as strict. In loose ordering the Worker Thread responsible for I/O operation N+1 is supposed to issue its I/O operation after I/O operation N has been issued, but before I/O operation N has completed. The idea is that the storage controller will receive several I/O operation requests and that after each request finishes, there is another request ready to be executed. There is also the distinct possibility that the subsequent I/O requests have some inherent locality of reference and the storage controller can optimize access to the storage devices based on this locality information.

In the case of purely sequential read operations, if the storage controller sees a consistent, well-ordered set of locations to access, it will be able to perform read-ahead operations thereby maximizing the read bandwidth of the underlying storage subsystem. Similarly, for write operations the storage controller could use its write cache more effectively by taking several write operations and coalescing them into a single large write to the storage subsystem.

In practice, loose ordering seems to perform better than serial ordering for purely sequential read operations because the storage controller always has an I/O operation to work on.

Slide2.EMF

### No Ordering

When no ordering is used the Worker Threads will issue their respective I/O operations without waiting for any prior I/O operations to complete. The resulting order in which the storage controller sees I/O operations should be reasonably sequential (in terms of order not necessarily in terms of access locations). However, the order in which the I/O operations actually get issued depends largely on when the various Worker Threads are scheduled to run on the system. It has been observed that some Worker Threads can be “starved” for CPU cycles resulting in some I/O operations experiencing long delays before they are issued and/or completed. This is more an artifact of the operating system process scheduling algorithms than it is a property of XDD. If I/O operation ordering is an issue, the serial and loose ordering schemes can be used.

## Timeserver and Gettime

Time synchronization can be done across multiple machines each running XDD. This is accomplished by using the timeserver and gettime programs. The timeserver program provides a single global reference clock that is used by the timestamping in order to be able to more accurately correlate events in time from multiple computer systems. A single global time is useful when testing performance over multiple machines, but it is not necessary for end-to-end tasks. The timeserver program should be run on a single machine as a background task. This machine must be accessible via a LAN to all the machines that will be running XDD. This LAN should preferably be a lightly used LAN for optimum results.

Timeserver

Fibre Channel Switch

For the SAN

XDD 2

XDD 1

XDD n

Ethernet LAN

The preferred way to make this work is to use ssh to start the XDD programs on each of the nodes using the **-starttime** and **-timeserver** options. The **-timeserver** option tells the XDD program the hostname or IP address of the timeserver machine. Each XDD program will contact this machine to determine the global time they will use as a frame of reference. The **-starttime** option specifies the time to start in global time.

Before running the XDD command line in a script, it is necessary to determine a global time sometime in the future at which all the XDD programs will start. The way to do this is to use the gettime program to contact the timeserver, determine the global time, add a specified number of seconds, and display the result on standard out. This global start time can then be used as the argument to the **-starttime** option for each of the XDD invocations. An example script might look like this:

set g=`gettime –timeserver 192.10.11.12 –add 20`

# At this point ${g} will be the current global time plus

# 20 seconds.

foreach i ( 1 2 3 4 )

ssh host${i} xdd –starttime ${g} –op read –targets 1 /dev/dsk/c1d2s0 –mbytes 5 –timeserver host 192.10.11.12 &

end

wait

This example script will set the local variable “${g}” to the global time plus 20 seconds. This value is then passed to each of the XDD programs that are started on host1, host2, host3, and host4. This will result in each XDD program starting at exactly the same time.

### Timeserver and Gettime Command Line Arguments

Command synopsis of the timeserver program is

timeserver [-**port** *#*]

Where

**-port #** specifies the port number to use for the timeserving function.

The gettime program is used to obtain the global clock value from the timeserver machine. The command synopsis is:

gettime

-timeserver *hostname*

-port *#*

-add *seconds*

-bounce *times*

-verbose

-waitfortime *milliseconds*

Where:

**-timeserver** *hostname*specifies the name or IP address of the machine running the timeserver.

**-port** *#*specifies the port number to use when contacting the timeserver.

**-add** *seconds*specifies number of seconds to add to the global time that is displayed as the output of this program.

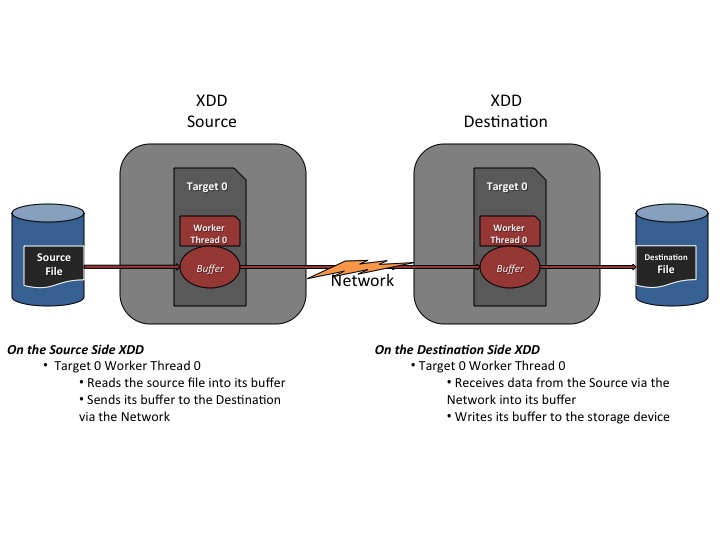
**-bounce** *times*specifies the number of times to ping the timeserver in order to get a minimum round trip time. The higher the bounce count, the more accurate the global time will be.

**-verbose** will cause gettime to display more information than simply the global clock value.

**-waitfortime** will cause gettime to wait until the specified global time in milliseconds is reached at which time gettime will complete. This is useful when running it in a script to block the execution until a particular global time.

# XDDCP

XDDCP is an executable located in the “src/tools” subdirectory of the standard XDD distribution. This script is used to copy one or more files from one computer to another over a network. It is intended to be used for very large files, typically in excess of 50 GB each, over high-speed networks starting at 10Gigabits/sec. The figure below illustrates the basic operation of XDDCP.



XDDCP uses XDD and the end-to-end option to accomplish the data copy operation. It does this by starting an instance of XDD on the destination computer using ssh, and then it starts an instance of XDD on the source computer. The source side is defined to be where the original file exists. The destination side is defined to be where the copy of the file will be created.

The destination side is started first and listens on the specified socket numbers. This version of XDD requires a unique socket number for each Worker Thread pair. A Worker Thread pair consists of a source side Worker Thread that sends data to a corresponding destination side Worker Thread through a specific port number.

## XDDCP Usage

**xddcp** *[OPTIONS]* source*\_file destination\_host:destination\_file [size]*

*source\_file* - complete /filepath/name for source file on source host

*destination\_host* - destination host IP or Name over which data is transferred

*destination\_file* - complete /filepath/name for destination file on destination host

*size* - number of bytes to transfer [Default: size of source file]

**OPTIONS**

–**a** – Specifies that this invocation of xddcp should keep a restart file so that the copy operation can be resumed at a later time if it is interrupted. If this is the resumption of a prior invocation of xddcp then it is used to resume a transfer from the point where the previous invocation was interrupted.

–**d** – Use direct I/O on the destination side of the copy.

–**s** – Use direct I/O on the source side of the copy.

–**f** – If ports are unavailable on destination, attempt to kill any running XDD’s that are running on the destination side and retry starting the destination-side XDD.

–**F** – Transfer a list of files named in *source\_file*, which is a simple ASCII text file with one file name per line. File names can be either relative or fully qualified path names.

–**h** – Print out usage information.

–**p** *portnum* – First port to use for transfer [Default: 40010]. This is also known as the base port.

–**o** – Ordered-mode, write data in serial order on the destination side. Normally, no ordering is used on the destination side.

–**t** *threads* – Used to specify the number of Worker Threads to use on the source and destination sides of a copy operation. [Default: 8]

–**v** – Additional information added to the output log files. This will also cause timestamp files to be generated on both the source and destination sides of a copy.

–**w** – Same as the –v option but will also generate a binary timestamp

**NOTE:** 'xdd' must be in your PATH env on both source and destination hosts

### How to Restart

In the event an XDDCP operation terminates before all the data has been transferred, it is possible to have the subsequent XDDCP restart the copy operation where the previous left off. The **-a** option of XDDCP will cause the destination side of the copy operation to maintain a restart file that contains the offset from the start of the file where the last data block committed to stable storage. It is guaranteed that all the data from the beginning of the file to the restart offset has been written to stable storage.

If an XDDCP operation terminates before it completes the copy, just reissue the same command, which will cause the destination side to restart the copy operation from the point specified in the most recent restart file for the specific destination file.

# XDDMCP

XDDMCP is very similar in function to XDDCP, but XDDMCP can be used with multiple source hosts.

## XDDMCP Usage

**xddmcp** *[OPTIONS] [source\_host[,source\_host]:]*source*\_file [destination\_host:]destination\_file*

*source\_host* - source host IP(s) or name(s) from which data is transferred

*source\_file* - complete /filepath/name for source file on source host(s)

*destination\_host* - destination host IP or name over which data is transferred

*destination\_file* - complete /filepath/name for destination file on destination host

**OPTIONS**

–**a**, **--resume** – Specifies that this invocation of xddmcp should keep a restart file so that the copy operation can be resumed at a later time if it is interrupted. If this is the resumption of a prior invocation of xddmcp then it is used to resume a transfer from the point where the previous invocation was interrupted.

–**d** *s|d|b* – Use direct I/O for the specified side (s for source, d for destination, b for both sides)

–**s** *size*, **--size=***size*– The number of bytes to transfer from the source(s).

–**h, --help** – Print out usage information.

–**n** *N*, **--retry**=*N* – If a failure occurs, retry the transfer up to N times [Default: 0]

–**p** *port,* ***--port****=port* – First port to use for listening [Default: 40010].

–**o** *s|d|b* – Use serial ordering for the specified side (s for source, d for destination, b for both).

–**r** – Copy the files recursively [Default: On]

–**t** *threads* – Used to specify the number of Worker Threads to use on the destination side of a copy operation. This is similar to **–queuedepth** for XDD. The source side(s) of the copy operation will have equally divided number of Worker Threads, so the total Worker Threads across all sources is equal to *threads*. [Default: 8]

–**v** – Create a log file.

–**V** – Add a timestamp output to the log.

**NOTE:** 'xdd' must be in your PATH env on both source and destination hosts

### Restart Usage

In the event an XDDMCP operation terminates before all the data has been transferred, it is possible to have the subsequent XDDMCP restart the copy operation where the previous left off. The **-a** option of XDDMCP will cause the destination side of the copy operation to maintain a restart file that contains the offset from the start of the file where the last data block committed to stable storage. It is guaranteed that all the data from the beginning of the file to the restart offset has been written to stable storage.

If an XDDMCP operation terminates before it completes the copy, just reissue the same command, which will cause the destination side to restart the copy operation from the point specified in the most recent restart file for the specific destination file.

## End-to-End Theory of Operation

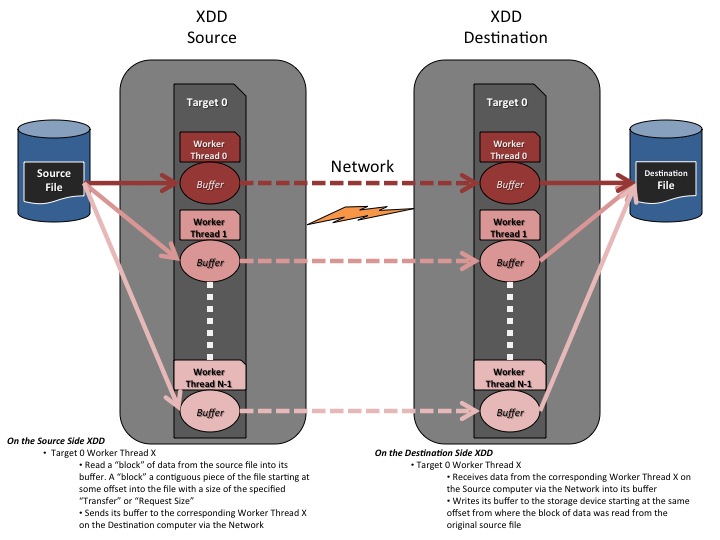
XDDMCP uses the end-to-end options to accomplish the task of copying data from one machine to another over a network. The end-to-end operation is accomplished by pairing together Worker Threads from the source machines to Worker Threads on the destination machine. The source machines are defined to be the system that reads in the original file(s) and transmits its contents over a network to the destination machine, which is defined to be the system that writes the copy of the file to stable storage. Data movement is always from the source(s) to the destination.

XDDMCP is a program that makes use of XDD’s end-to-end capabilities. When an XDDMCP instance is invoked, it actually invokes separate instances of XDD on the source and destination machines. The XDD invocations are done behind-the-scenes to the user, but it is important to understand the mechanics.

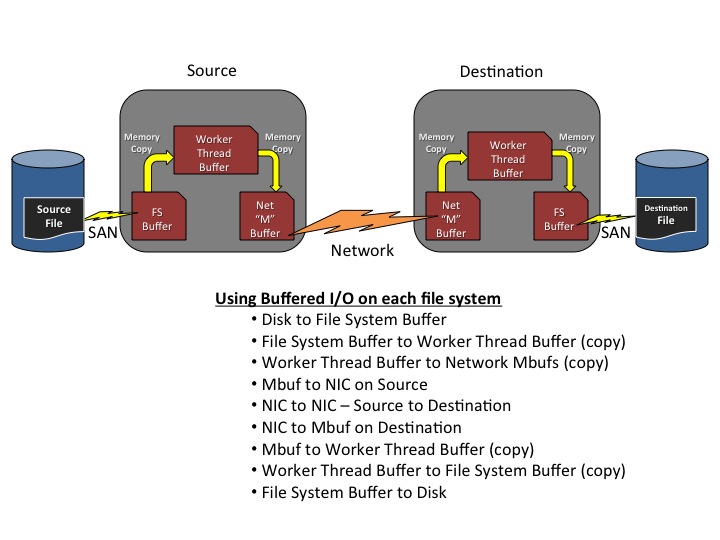
In order for the source and destination instances of XDD to communicate, they must be matched in the sense that they each have the same queue depth (i.e. number of Worker Threads) and they must agree on which network addresses and ports to use. For XDD end-to-end operations the queue depth can either be specified explicitly using the **-queuedepth** option, or it can be specified implicitly using the “**-e2e dest** *hostname:base\_port,number\_of\_ports*” option. When the queue depth is specified implicitly, it will take precedence over the **–queuedepth** option if it is also specified. Furthermore, the queue depth implied on the **“-e2e dest…”** option is the sum of all number\_of\_ports specified for a given XDD run.

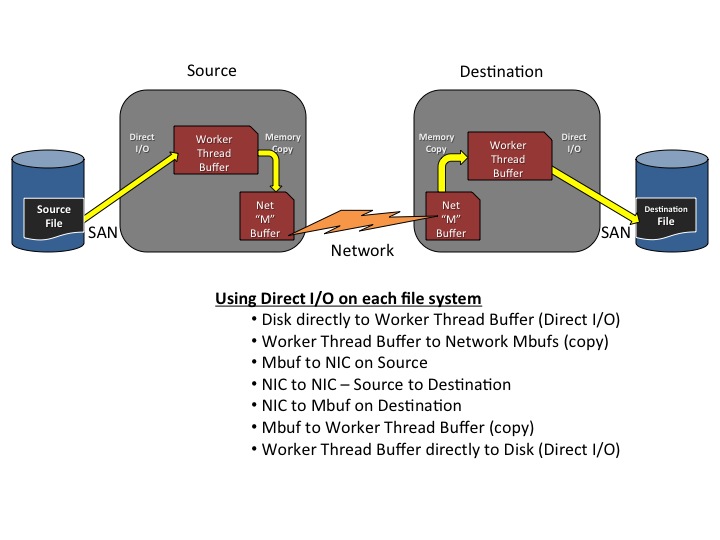
For XDDMCP, the **–t** *number\_of\_threads* option will explicitly specify the number of Worker Threads on the destination side. The number of threads is divided evenly (or as close as possible) across the source machines. So, if the user specified “**-t** 8” with two sources, then the destination side would have 8 Worker Threads receiving data from the network, and each source would use 4 Worker Threads to send the data.

The following diagram describes how end-to-end operations work with multiple threads:



XDD can use either buffered I/O or direct I/O for reading the source file or writing the destination file. Buffered I/O will increase the CPU overhead and possibly result in lower overall bandwidth performance of an end-to-end operation. Therefore, it is recommended that direct I/O be used whenever possible. The following diagrams illustrate the basic difference between buffered I/O and direct I/O as they relate to an end-to-end operation.





# Output and Reports

## Reporting Options

The **–verbose** option will display performance information for each pass within an XDD run along with per-target totals and combined averages. There is also an **-id** option that will display a specified string of identification information along with the normal output of the run. If the id string is specified to be “commandline” then the entire XDD command line plus all the options will be display. This is useful when running XDD from a shell script to get more qualitative information about the run into the output file.

Since XDD I/O operations can fail due to device failures or option specifications it is possible to generate a tremendous number of error messages. To avoid filling up output files with too many error messages, the **–maxerrors** option can be used to limit the number of error messages to display to some small, finite number. Once this number of errors has been reached, I/O operations to that target are halted for that pass.

## Output Format

The **–outputformat** option allows the user to specify which variables should be displayed in the output line. For example, given a normal XDD command line with the usual options, it is possible to use the option as follows:

xdd … -outputformat “+PASS+TARGET+BYTESREAD+OPS+READBANDWIDTH”

This will produce output that would only contain the pass number, target number, bytes read, operations performed, and achieved read bandwidth.

The following is a list of defined output format identifier strings that can be used. Each format identifier string must begin with a “+” as indicated.

**+WHAT**

**+PASS**

**+TARGET**

**+QUEUE**

**+BYTESTRANSFERED**

**+BYTESXFERED**

**+BYTESREAD**

**+BYTESWRITTEN**

**+OPS**

**+READOPS**

**+WRITEOPS**

**+BANDWIDTH**

**+READBANDWIDTH**

**+WRITEBANDWIDTH**

**+IOPS**

**+READIOPS**

**+WRITEIOPS**

**+LATENCY**

**+ELAPSEDTIME1STOP**

**+ELAPSEDTIMEPASS**

**+OVERHEADTIME**

**+PATTERNFILLTIME**

**+BUFFERFLUSHTIME**

**+CPUTIME**

**+PERCENTCPUTIME**

**+PERCENTCPU**

**+USERTIME**

**+USER**

**+USRTIME**

**+SYSTEMTIME**

**+SYSTEM**

**+SYSTIME**

**+PERCENTUSER**

**+PERCENTUSERTIME**

**+PERCENTUSR**

**+PERCENTSYSTEM**

**+PERCENTSYSTEMTIME**

**+PERCENTSYS**

**+OPTYPE**

**+XFERSIZEBYTES**

**+XFERSIZEBLOCKS**

**+XFERSIZEKBYTES**

**+XFERSIZEMBYTES**

**+E2EIOTIME**

**+E2ESRTIME**

**+E2EPERCENTSRTIME**

**+E2ELAGTIME**

**+E2EPERCENTLAGTIME**

**+E2EFIRSTREADTIME**

**+E2ELASTWRITETIME**

**+DELIM**

**+NODENAME**

\*\* Aside from the obvious ones, the “+DELIM” can be used to print a specific delimiter between variables that get displayed. The “+WHAT” is used to indicate which output line is being displayed: PASS, Worker Thread Average, Target Average, Combined, …etc. All the others are reasonably self-explanatory

The output of XDD comes in several sections. The first section describes the state of options selected for the run that apply to all the targets being tested. The second section of output describes the options that apply to each of the individual targets. The third section contains information about the system that XDD is being run on. **The fourth section describes the rate information.**

**IOIOIOIOIOIOIOIOIOIOI XDD version Linux.7.0.0.rc9.012810.Build.0317 based on Linux.7.0.0.rc9.011810.Build.1412 IOIOIOIOIOIOIOIOIOIOIOI**

**xdd - I/O Performance Inc., US DoE/DoD Extreme Scale Systems Center <ESSC> at Oak Ridge National Labs <ORNL> - Copyright 1992-2010**

**XDD DISCLAIMER:**

**\*\*\* >>>> WARNING <<<<**

**\*\*\* THIS PROGRAM CAN DESTROY DATA**

**\*\*\* USE AT YOUR OWN RISK**

**\*\*\* IOPERFORMANCE and/or THE AUTHORS ARE NOT LIABLE FOR**

**\*\*\* >>>> ANYTHING BAD <<<<**

**\*\*\*\* THAT HAPPENS WHEN YOU RUN THIS PROGRAM**

**...although we will take credit for anything good that happens**

**but we are not \*liable\* for that either.**

**Starting time for this run, Thu Jan 28 03:22:27 2010**

**ID for this run, 'No ID Specified'**

**Maximum Process Priority, disabled**

**Passes, 1**

**Pass Delay in seconds, 0**

**Maximum Error Threshold, 0**

**Target Offset, 0**

**I/O Synchronization, 0**

**Total run-time limit in seconds, 0**

**Output file name, test-run.txt**

**CSV output file name, test-run.csv**

**Error output file name, stderr**

**Pass synchronization barriers, enabled**

**Number of Targets, 1**

**Number of I/O Threads, 1**

**Computer Name, pod7.ccs.ornl.gov, User Name, tmruwart**

**OS release and version, Linux 2.6.18-128.1.6.el5 #1 SMP Wed Apr 1 06:58:14 EDT 2009**

**Machine hardware type, x86\_64**

**Number of processors on this system, 8**

**Page size in bytes, 4096**

**Number of physical pages, 8241895**

**Megabytes of physical memory, 32194**

**Clock Ticks per second, 100**

**Seconds before starting, 0**

**Target[0] Q[0], /data/xfs/testfile**

**Target directory, "./"**

**Process ID, 2523**

**Thread ID, 2524**

**Processor, all/any**

**Read/write ratio, 0.00 READ, 100.00 WRITE**

**Throttle in MB/sec, 0.00**

**Per-pass time limit in seconds, 0**

**Pass seek randomization, disabled**

**File write synchronization, disabled**

**Blocksize in bytes, 1024**

**Request size, 4096, blocks, 4194304, bytes**

**Number of Operations, 16, of, 16, target ops for all Worker Threads**

**Start offset, 0, blocks, 0, bytes, 0.000 KBytes, 0.000 MBytes, 0.000 GBytes, 0.000000 TBytes**

**Total data transfer for this Target, 65536, blocks, 67108864, bytes, 65536.000 KBytes, 64.000 MBytes, 0.062 GBytes, 0.000061 TBytes**

**Total data transfer for this Worker Thread, 65536, blocks, 67108864, bytes, 65536.000 KBytes, 64.000 MBytes, 0.062 GBytes, 0.000061 TBytes**

**Pass Offset, 0, blocks, 0, bytes, 0.000 KBytes, 0.000 MBytes, 0.000 GBytes, 0.000000 TBytes**

**Seek range, 1048576, blocks, 1073741824, bytes,1048576.000 KBytes,1024.000 MBytes, 1.000 GBytes, 0.000977 TBytes**

**Seek pattern, sequential**

**Flushwrite interval, 0**

**I/O memory buffer is a normal memory buffer**

**I/O memory buffer alignment in bytes, 4096**

**Data pattern in buffer,0x00**

**Data buffer verification is disabled.**

**Direct I/O, enabled**

**Preallocation, 0**

**Queue Depth, 1**

**Timestamping, enabled for DETAILED SUMMARY**

**Timestamp ASCII output file name, test-run.target.0000.Worker Thread.0000.csv**

**Delete file, disabled**

What Pass Target Queue Bytes\_Xfered Ops Elapsed Bandwidth IOPS Latency Pct\_CPU Op\_Type Xfer\_Size

UNITS>> Number Number Number Bytes #ops seconds MBytes/s Ops/s millisec percent text bytes

TARGET\_PASS 1 0 1 67108864 16 1.375 48.820 11.640 85.914 0.727 write 4194304

TARGET\_AVERAGE 1 0 1 67108864 16 1.375 48.820 11.640 85.914 0.727 write 4194304

COMBINED 1 1 1 67108864 16 1.375 48.820 11.640 85.914 0.727 write 4194304

Ending time for this run, Thu Jan 28 03:22:28 2010

The default output of XDD is a line of text with the following format

**What Pass Target Queue Bytes\_Xfered Ops Elapsed Bandwidth IOPS Latency Pct\_CPU Op\_Type Xfer\_Size**

***UNITS>> Number Number Number Bytes #ops seconds MBytes/s Ops/s millisec percent text* *bytes***

The first line indicates the variable being reported, the second line provides the units for each variable, and the lines beyond that are the values themselves.

The fields have the following meanings:

**What** - This is an identifier that explains how the results should be interpreted for the given line. The possible values of “What” are:

* TARGET PASS – displayed only when the **–verbose** option is specified. Provides the results for a specific target for a particular pass. The target results are a combined average of all the associated Worker Threads for the target. The result in this case reflects the total number of Worker Threads operating on behalf of the target.
* TARGET AVERAGE – displayed only when the **–verbose** option is specified. Provides the results for a specific target averaged over all passes in the run.
* COMBINED – Provides the combined results over all targets and Worker Threads for all passes in a run.

**Pass** is the target pass number of the specific result. Pass numbers start at 1.

**Target** is the target number, relative to 0 (zero). A list of the targets and the associated numbers precedes this line of output.

**Queue** is the total number of Worker Threads being used for that target.

**Bytes\_Xfered** is the total number of bytes that were transferred during the XDD run for this target.

**Ops** is the total number of read or write operations performed on this target.

**Elapsed** is the number of seconds that elapsed for the current pass in a multi-pass run for the target or Worker Thread being reported.

**Bandwidth** is the average I/O rate in MB per Second where 1 MB/sec equals 106 bytes per second.

**IOPS** is the average number of I/O operations per second for this pass in a multi-pas run.

**Latency** is the average time it takes to perform each operation.

**Pct\_CPU** is the percentage of the CPU that was used by all the Worker Threads on behalf of this target. This includes system and user time.

**Op\_type** is either read, write, or mixed.

**Xfer\_Size** is the average request size in block-size blocks that was used for that target or Worker Thread. Generally this is a constant value for any particular test run.

If the **–verbose** option is specified then each line has a pass number associated with it and the final output lines report overall averaged values for the average bandwidth and elapsed time. For a multi-target run, the sum total of all the targets is presented as the combined average. For example, if two targets are being tested and each target performs at an average of 75 MB/sec, the combined average is 150 MB/sec.

## Example Timestamp Output

Target and Worker Thread number for this report, 0, 0

IOIOIOIOIOIOIOIOIOIOI XDD version Linux.7.0.0.rc9.012810.Build.0317 based on Linux.7.0.0.rc9.011810.Build.1412 IOIOIOIOIOIOIOIOIOIOIOI

xdd - I/O Performance Inc., US DoE/DoD Extreme Scale Systems Center <ESSC> at Oak Ridge National Labs <ORNL> - Copyright 1992-2010

XDD DISCLAIMER:

\*\*\* >>>> WARNING <<<<

\*\*\* THIS PROGRAM CAN DESTROY DATA

\*\*\* USE AT YOUR OWN RISK

\*\*\* IOPERFORMANCE and/or THE AUTHORS ARE NOT LIABLE FOR

\*\*\* >>>> ANYTHING BAD <<<<

\*\*\*\* THAT HAPPENS WHEN YOU RUN THIS PROGRAM

...although we will take credit for anything good that happens

but we are not \*liable\* for that either.

Starting time for this run, Thu Jan 28 03:22:28 2010

ID for this run, 'No ID Specified'

Maximum Process Priority, disabled

Passes, 1

Pass Delay in seconds, 0

Maximum Error Threshold, 16

Target Offset, 0

I/O Synchronization, 0

Total run-time limit in seconds, 0

Output file name, test-run.txt

CSV output file name, test-run.csv

Error output file name, stderr

Pass synchronization barriers, enabled

Number of Targets, 1

Number of I/O Threads, 1

Computer Name, pod7.ccs.ornl.gov, User Name, tmruwart

OS release and version, Linux 2.6.18-128.1.6.el5 #1 SMP Wed Apr 1 06:58:14 EDT 2009

Machine hardware type, x86\_64

Number of processors on this system, 8

Page size in bytes, 4096

Number of physical pages, 8241895

Megabytes of physical memory, 32194

Clock Ticks per second, 100

Seconds before starting, 0

Target[0] Q[0], /data/xfs/testfile

Target directory, "./"

Process ID, 2523

Thread ID, 2524

Processor, all/any

Read/write ratio, 0.00 READ, 100.00 WRITE

Throttle in MB/sec, 0.00

Per-pass time limit in seconds, 0

Pass seek randomization, disabled

File write synchronization, disabled

Blocksize in bytes, 1024

Request size, 4096, blocks, 4194304, bytes

Number of Operations, 16, of, 16, target ops for all Worker Threads

Start offset, 0, blocks, 0, bytes, 0.000 KBytes, 0.000 MBytes, 0.000 GBytes, 0.000000 TBytes

Total data transfer for this target, 65536, blocks, 67108864, bytes, 65536.000 KBytes, 64.000 MBytes, 0.062 GBytes, 0.000061 TBytes

Total data transfer for this WORKER THREAD, 65536,blocks, 67108864, bytes, 65536.000 KBytes, 64.000 MBytes, 0.062 GBytes, 0.000061 TBytes

Pass Offset, 0, blocks, 0, bytes, 0.000 KBytes, 0.000 MBytes, 0.000 GBytes, 0.000000 TBytes

Seek range, 1048576, blocks, 1073741824, bytes, 1048576.000 KBytes, 1024.000 MBytes, 1.000 GBytes, 0.000977 TBytes

Seek pattern, sequential

Flushwrite interval, 0

I/O memory buffer is a normal memory buffer

I/O memory buffer alignment in bytes, 4096

Data pattern in buffer,0x00

Data buffer verification is disabled.

Direct I/O, enabled

Preallocation, 0

Queue Depth, 1

Timestamping, enabled for DETAILED SUMMARY

Timestamp ASCII output file name, test-run.target.0000.Worker Thread.0000.csv

Delete file, disabled

Start of DETAILED Time Stamp Report, Number of entries, 16, Picoseconds per clock tick, 1000000, delta, 0

Target, RWV Op, Pass, OP Number, Block Location, Distance, StartTS, EndTS, IO\_Time\_ms, Relative\_ms, Delta\_us, Loop\_ms, Inst MB/sec

0,w,1,0,0,0,10288350303715490112,10288350437959490112, 134.24400, 0.00000, 0.00000, 0.00000, 31.244

0,w,1,1,4096,0,10288350437966490112,10288350478994490112, 41.02800, 134.25100, 7.00000, 41.03500, 102.230

0,w,1,2,8192,0,10288350478997490112,10288350533440490112, 54.44300, 175.28200, 3.00000, 54.44600, 77.040

0,w,1,3,12288,0,10288350533446490112,10288350736743490112, 203.29700, 229.73100, 6.00000, 203.30300, 20.631

0,w,1,4,16384,0,10288350736750490112,10288350789934490112, 53.18400, 433.03500, 7.00000, 53.19100, 78.864

0,w,1,5,20480,0,10288350789937490112,10288350839706490112, 49.76900, 486.22200, 3.00000, 49.77200, 84.275

0,w,1,6,24576,0,10288350839708490112,10288350973728490112, 134.02000, 535.99300, 2.00000, 134.02200, 31.296

0,w,1,7,28672,0,10288350973733490112,10288351026693490112, 52.96000, 670.01800, 5.00000, 52.96500, 79.198

0,w,1,8,32768,0,10288351026697490112,10288351109349490112, 82.65200, 722.98200, 4.00000, 82.65600, 50.747

0,w,1,9,36864,0,10288351109354490112,10288351173507490112, 64.15300, 805.63900, 5.00000, 64.15800, 65.380

0,w,1,10,40960,0,10288351173509490112,10288351250697490112, 77.18800, 869.79400, 2.00000, 77.19000, 54.339

0,w,1,11,45056,0,10288351250703490112,10288351290625490112, 39.92200, 946.98800, 6.00000, 39.92800, 105.062

0,w,1,12,49152,0,10288351290627490112,10288351353374490112, 62.74700, 986.91200, 2.00000, 62.74900, 66.845

0,w,1,13,53248,0,10288351353380490112,10288351431280490112, 77.90000, 1049.66500, 6.00000, 77.90600, 53.842

0,w,1,14,57344,0,10288351431284490112,10288351621563490112, 190.27900, 1127.56900, 4.00000, 190.28300, 22.043

0,w,1,15,61440,0,10288351621569490112,10288351678337490112, 56.76800, 1317.85400, 6.00000, 56.77400, 73.885

End of DETAILED Time Stamp Report

Start of SUMMARY Time Stamp Report

Average seek distance in 1024 byte blocks, 0, request size in blocks, 4096

Range: Longest seek distance in blocks, 0, shortest seek distance in blocks, 0

Average I/O time in milliseconds, 77.51937, average dead time in microseconds, 4.53333

I/O Time Range: Longest I/O time in milliseconds, 203.29700, shortest I/O time in milliseconds, 39.92200

Dead Time Range: Longest dead time in microseconds, 7.00000, shortest dead time in microseconds, 2.00000

End of SUMMARY Time Stamp Report

# Performance Tuning Hints

This section describes various hints about performance tuning.

## Caches and Write Performance

When writing to a storage device whether it is a single disk drive or a disk array it is important to know the status of the caches on each of the devices that have cache. For maximum performance of write operations, it is necessary to enable the write caches on a disk array controller, as well as the write caches on the disk drives themselves.

## Fibre Channel Frame Sizes

Fibre Channel host bus adapters (HBA), switches, and target devices all have frame sizes defined and negotiated when any two FC devices are connected together. For maximum bandwidth performance, it is important to make certain that the FC frame size is set to 2048 bytes. For maximum transaction performance for small transaction sizes (i.e. around 512 bytes per transaction), a smaller frame size of 512 or 1024 bytes can be used.

# Examples

The following is a list of examples on how to run XDD.

## Example 1 – Basic XDD command line

xdd –op read –targets 1 /dev/rdsk/dsk10d2s0 –reqsize 128

-mbytes 64 –passes 3 –verbose

This is a very basic test that will read sequentially from target device “/dev/rdsk/dks10d2s0” starting at block 0 using a fixed request size of 128 blocks until it has read 64 MB (64 \* 1024\*1024 bytes). It will do this 3 times and display performance information for each pass. The default block size is 1024 bytes per block so the request size in bytes is 128 KB (128 \* 1024 bytes). Please note that all these options need to be on a single command line unless they are in the setup file where they can be on separate lines.

## Example 2 – Specifying multiple targets and time limit

xdd –op write –targets 2 /raid/BIGFILE1 /raid/BIGFILE2

-blocksize 512 –reqsize 128 -mbytes 64 –verbose

–passes 3 -timelimit 10

This test will write sequentially from 2 target files “/raid/BIGFILE1” and “/raid/BIGFILE2” starting at the beginning of each file using a fixed request size of 128 blocks of 512 bytes per block until it has read 64 MB (64\*1024\*1024 bytes), oruntil it has reached a time limit of 10 seconds, at which time it will end the current pass and proceed to the next pass. It will do this 3 times and display performance information for each pass. The combinedperformance of both devices is calculated and displayed at the end of the run.

## Example 3 – Time Stamping and Setup File

xdd –op write –targets 2 /dev/rdsk/dks10d2s0 /dev/rdsk/dks10d2s0

-setup XDD.setup –ts detailed –ts output ts.write

This test will read sequentially from 2 targets that are actually a single device, “/dev/rdsk/dks10d2s0”. The request size of 128 blocks at 2048 bytes per block, read limit of 4096 MB (4096\*1024\*1024 bytes), the time limit of 10 seconds for each pass, verbose output, and pass count of 3 are all specified in the “XDD.setup” file which looks like so:

-blocksize 2048

–reqsize 128

-mbytes 4096

–verbose

–passes 3

-timelimit 10

The timestamp option (**-ts)** is also used in this example to dump an ASCII output file called “ts.write”. It should be noted that these timestamp file names are appended with a t# where # is the number of the target associated with the particular timestamp file. In this example, since there are two targets, the timestamp files will be “ts.write.t0” and “ts.write.t1”.

## Example 4 – Random Seeks

xdd –op read –targets 1 /dev/rdsk/dsk10d2s0 –reqsize 8

-mbytes 16 –passes 3 –verbose

–seek random –seek range 4000000

This is a very basic random I/O test that will read from target device “/dev/rdsk/dks10d2s0” starting at some random location using a fixed request size of 8 blocks until it has read 16 MB (16\*1024\*1024 bytes). It will do this 3 times and display performance information for each pass. The default block size is 1024 bytes per block so the request size in bytes is 8 KB (8\*1024 bytes). The number of requests that need to be generated to read 16 MB in 8192 byte chunks is 2048. Since this is a purely random I/O pattern, these 2048 requests are distributed over a range of 4,000,000 blocks (again 1024 bytes per block). This is useful in constraining the area over which the random locations are chosen from. The same seek locations are used for each pass in order to generate reproducible results. In fact, upon each invocation of XDD using the same parameters, the same random locations are generated each time. This allows the user to change the disk, or starting offset, and observe the effects. The random locations may be changed from pass to pass within an XDD run by using the **–randomize** option in which case a new set of locations is generated for each pass. Furthermore, the random locations may be changed from run to run using the **-seek seed** option to specify a different random number generation seed value for each invocation of XDD.

## Example 5 – End to End operation

Perform an end-to-end operation between two hosts, hostA and hostB, where hostA is the source and hostB is the destination.

Start the instance of XDD on hostB, the destination side, first. This is required because if the destination side is not running when the source side starts, the source side will terminate early because it will not be able to connect to the instance of XDD on the destination side.

Hence, on the destination side:

xdd –op write –targets 1 /tmp/foo2 –reqsize 4096 -mbytes 3000

–verbose –e2e isdest –e2e dest 192.168.17.10

–e2e port 2010

Once this is running it will wait for a connection from the source side before writing data to the destination target file “/tmp/foo2”.

On the source side:

xdd –op read –targets 1 /tmp/foo1 –reqsize 4096 -mbytes 3000

–verbose –e2e issource –e2e dest 192.168.17.10

–e2e port 2010

When the source side starts, it will open a socket to the destination host then start reading the source file “/tmp/foo1” and sending it over the socket to the destination side.

Once the target file on the source side has been read and transferred, assuming no additional passes are requested, then the source side will terminate followed by the destination side. Each will display the usual results with an additional value at the end of each output line. This value will be a number between 0 and 100 and represents the percentage of total time that was spent by XDD transferring data over the network.

It is important to follow these basic rules when running an end-to-end operation:

* The source and destination XDD commands must contain either “**–e2e** **issource**” or “**–e2e** **isdest**”, respectively in order to properly identify their respective roles
* The operation or **–op** for the source must be “read”
* The operation or **–op** for the destination must be “write”
* The number of megabytes to read from the source target should be equal to the number of megabytes written to the destination target
* The queue depth (i.e. **–queuedepth** option) must be the same on the destination side as the total over all sources, for a given target
* The destination hostname/address must be specified in both source and destination XDD commands and must be the same

# Under the Hood

This section is a look at XDD program organization and data structures.

## XDD General Operation

XDD is a command line driven program. The first thing it does upon invoking it is to parse the command line. If the command line contains the **-setup** option, it will parse the command line options from left to right up to the **–setup** option. It will then parse the setup file options followed by parsing the remaining command line options.

After the all the requested options have been set, the main XDD program will start all the appropriate threads one at a time. Each thread will go through its initialization phase, which includes the following:

* Open the target and prepare it for access
* Generate the list of locations to access and the associated access pattern and operations
* Allocate I/O buffers
* Perform timestamp setup if necessary
* Display target-specific information as requested

After the thread has completed its setup process, it will enter a serialization barrier that will release the main XDD parent thread, which will then start the initialization of the next thread. The Target Thread that has just completed initialization will enter the main barrier waiting for all the other Target Threads to complete initialization. Once all the threads have been initialized, the last thread to be initialized will enter the starting barrier and cause all the threads to be released and the fun begins.

All the threads will do their respective I/O operations for a single pass and then enter the pass barrier. The pass barrier causes threads to wait until all threads have completed a pass before starting the next pass. If this is the last pass, then upon being released from this barrier, all the threads will perform any cleanup operations and exit.

Another function of the pass barrier is to hold all the threads dormant while the Results Manager gathers all the results information from each of the threads, generates the appropriate intermediate results, and displays them, if requested. This is also the case when all threads have completed all passes and the Results Manager needs to process summary information as well.

## The XDD Thread Structure

XDD uses POSIX threads for all processes that it controls. There can be one or more threads per target. Each target will have a primary thread, or Target Thread, and some number of Worker Threads. The Worker Threads are used to perform parallel, asynchronous I/O on a particular target. The concept of Worker Threads is essentially AIO but since AIO is different from system to system, it was easier to simply implement one within XDD.

Each Target Thread and Worker Thread has a data structure associated with it that contains all the information related to a thread running at any given time. These structures are passed to the various routines within XDD and are allocated on an as-needed basis. The layout of the Target and Worker Data Structures are shown in Figure 1.

**TDS**

**Target 0**

**TDS**

**Target 1**

**TDS**

**Target N**

**WDS**

**Worker Thread 0**

**WDS**

**Worker Thread 1**

**WDS**

**Worker Thread X**

**WDS**

**Worker Thread 0**

**WDS**

**Worker Thread 1**

**WDS**

**Worker Thread Y**

**WDS**

**Worker Thread 0**

**WDS**

**Worker Thread 1**

**WDS**

**Worker Thread Z**

Figure 1. Target Data Structure (TDS) and Worker Data Structure (WDS) layout.

In the above diagram there are N targets each with some number of Worker Threads. The number of Worker Threads is equal to the queue depth specified for each target (using the **–queuedepth** option). The number of Worker Threads can be different for each target as is shown in this diagram but in practice the number of Worker Threads is the same for all targets during a run. The Target Data Structure for each target has a pointer to its Worker Thread 0. If there is more than one Worker Thread for a target, the subsequent Worker Threads are chained together as shown in the diagram. The last Worker Thread in the chain is identified by the fact that its next-Worker-Thread pointer is null.

## XDD Barriers

XDD uses a number of synchronization barriers in order to provide precise control and certain functionality such as lockstepping. The primary barriers cause all the threads to begin execution at precisely the same time for each pass of a run. The barriers are implemented using mutexes and semaphores. The amount of time required to process a semaphore request is significantly less than the time scales of an I/O operation and has no measurable effect on the results.

# Acknowledgements

HPUX is a registered trademark of Hewlett-Packard, Inc.

AIX is a registered trademark of IBM Corporation.

IRIX is a registered trademark of SGI.

Solaris and SPARC are registered trademarks of Sun Microsystems, Inc.

Windows, Windows NT, and Windows 2000 are registered trademarks of Microsoft, Inc.

The author would like to acknowledge and thank all those people who have helped with adding various functions and bug fixes to XDD over the years.

**The recent work on XDD version 6.7 and beyond is supported by the Oak Ridge National Laboratory Extreme Scale System Center and the United States Department of Defense.**

1. As of release 7.0.0 the IRIX Operating System is no longer supported. [↑](#footnote-ref-1)