

Contents

- 1 Use Cases
 - ◆ 1.1 Testing Large Terabyte LUNs
 - ◆ 1.2 Testing Large Aggregates
 - ◆ 1.3 Testing Deduplication
 - ◆ 1.4 File/LUN Copy and Verify
 - ◆ 1.5 Sparse File Testing
 - ◆ 1.6 Data Generation Tool
 - ◆ 1.7 Butterfly Effect
 - ◆ 1.8 Emulating crud Tool

Use Cases

Most of the dt program documentation describes how to use this tool, and/or recommendations, but this document describes use cases to hopefully peak your interest (many of these features are not available in other I/O generation tools).

Testing Large Terabyte LUNs

This test example, is documented in dt's Overview and shows how large terabyte LUNs can be tested using thinly provisioned (-o noreserve) storage.

This test shows a method to verify large TB sized disks without doing I/O to the full capacity. A number of options are used, here are the key ones:

- slices=16 to divide the capacity into 16 sections, each with their own process
- aios=4 to queue 16 requests per process for improved performance.
- step=4g to seek after every record, thereby avoid writing every blocks.
- iodir=reverse to start at the end of sections then work forward.
- prefix='%d@%h' defines a prefix written to beginning of each block.
- alarm=5s and noprogt=10s reports I/Os taking longer than 10 seconds.
- other options are left as an exercise to the reader!

```
shaix11# dt \
of=/dev/rhdisk10 \
slices=16 \
step=4g \
disable=pstats \
aios=4 \
oncerr=abort \
min=b \
```

```

max=256k          \
incr=var          \
align=rotate      \
pattern=iot       \
iodir=reverse     \
prefix='%d@%h'    \
alarm=5s          \
noprogt=10s

```

- Notes

- ◆ The *runtime=value* option can be used to run for a specific amount of time, just make sure the time specified is long enough to perform both the write and read passes.
- ◆ The no-progress options are useful when doing cluster failover (CFO) testing.

Please Note: The IOT pattern used above, encodes a 32-bit LBA which wraps at 2TB. This will be extended to a 64-bit LBA in a future release.

Testing Large Aggregates

This example will be similar to the above example for large terabyte LUNs. The difference in this example is we'll specify the path and size of the file to create on a file system (or CIFS/NFS volume) created on a large aggregate new in Boilermaker. The [Large Aggregate FAQ](#) is a good starting point, esp. for how to create a [fake Large Aggregate](#).

```

roanoke% dt          \
  of=largefile.data  \
  limit=15t          \
  slices=15          \
  step=1g            \
  disable=pstats     \
  aios=4             \
  oncerr=abort       \
  min=b             \
  max=256k           \
  incr=var           \
  align=rotate       \
  pattern=iot        \
  iodir=reverse      \
  prefix='%d@%h'     \
  alarm=5s           \
  noprogt=10s        \
  dispose=keep

```

...

dt (17982): End of Read pass 1/1, 34773 blocks, 16.979 Mbytes, 1024 records, errors 0/1, elapsed 00m

Total Statistics (17982):

```

Output device/file name: largefile.data (device type=regular)
Type of I/O's performed: sequential (reverse, rseed=0x3304261d)
Slice Range Parameters: position=13194139533312 (lba 25769803776), limit=1099511627776
Buffer alignment options: rotating through 1st 4 bytes
Current Slice Reported: 13/15
Data pattern prefix used: 'largefile.data@roanoke'
Data pattern string used: 'IOT Pattern' (blocking is 512 bytes)
Total records processed: 2048 with min=512, max=262144, incr=variable
Total bytes transferred: 35607552 (34773.000 Kbytes, 33.958 Mbytes)
Average transfer rates: 355791 bytes/sec, 347.452 Kbytes/sec
Asynchronous I/O's used: 4

```

User:Rtmiller/datatest_(dt)_Use_Cases

```
Number I/O's per second: 20.464
Total passes completed: 1/1
Total errors detected: 0/1
Total elapsed time: 01m40.08s
Total system time: 00m01.88s
Total user time: 00m08.00s
Starting time: Sun Nov 9 19:34:33 2008
Ending time: Sun Nov 9 19:36:17 2008

roanoke% ls -ls largefile.data
991680 -rw-r--r-- 1 rtmiller engr 16492674416640 Nov 9 19:35 largefile.data
roanoke% scu eval 991680k
Expression Values:

Decimal: 1015480320
Hexadecimal: 0x3c870000
512 byte Blocks: 1983360.000000
Kilobytes: 991680.000000
Megabytes: 968.437500
Gigabytes: 0.945740
Terabytes: 0.000924
roanoke% rm largefile.data
roanoke%
```

The above test will create 15 processes, each operating on its' own 1TB slice, each process queuing 4 async I/O's, all the the same file. For example, this might be a good test to generate concurrent I/O's across a striped Coral file systems.

Depending on the size of the aggregate, and the amount of backing storage you have available, you can alter the above options to limit the amount of data written. As you can see, for the options used above, ~968MB of storage was required. Large aggregates will test up to 100TB.

The resulting file will be a sparse file (holes that read as zeroes), due to the step option.

Testing Deduplication

To test deduplication, a test tool must write contiguous dupable blocks of data, so deduplication will actually happen. Many tools generate unique data in each block, and rightfully so, since this makes it easier to troubleshoot data corruptions, and to that end, *dt* has several options to create unique block data. But for testing deduplication, we need some set of contiguous duplicated data blocks. Described below, is one method to accomplish this using *dt*.

Please Note: The *hammer* I/O generation tool is not good for dedupe testing, since each blocks' data is unique.

There are several predefined data pattern files included in the source kit, containing the following data patterns:

- pattern_0 - psuedo-random pattern.
- pattern_1 - psuedo-random pattern alternating with 0's.
- pattern_2 - all 0s.
- pattern_3 - all 1s (FFH).
- pattern_4 - 16 bit shifting 1 in a field of 0s.
- pattern_5 - 16 bit shifting 0 in a field of 1s.
- pattern_6 - alternating 01 pattern (AAH).
- pattern_7 - byte incrementing (OOH - FFH).

- pattern_8 - encrypted data pattern.
- pattern_9 - psuedo-random pattern1 alternating with 0's.
- pattern_all ? concatenation of all of the above patterns.
- pattern_dedup - each pattern file written (bs=4k records=5).

These files can be specified with the pf=file option, or you can create your own.

The pattern_all file has been used effectively for testing tape testing, which does byte compression, but since ONTAP does block deduplication and its' blocks are 4k, the pattern_dedup file was created to ensure blocks have duplicate data via this simple script:

```
roanoke% cat makededup
#!/bin/ksh
alias dt=/u/rtmiller/Tools/dt.d-WIP/solaris-sparc/dt
rm -f pattern_dedup
dt pf=pattern_0 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_1 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_2 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_3 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_4 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_5 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_6 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_7 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_8 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
dt pf=pattern_9 of=pattern_dedup oflags=append bs=4k records=2 disable=stats,compare dispose=keep
ls -ls pattern_dedup
roanoke%
```

You can customize the above script to suit your needs (of course).

The resulting data pattern file can be used thusly:

```
sf49ers02% dt of=/var/tmp/dt.data \
bs=32k \
min=4k \
max=256k \
incr=4k \
flags=direct \
limit=100m \
pf=~rtmiller/Tools/dt.d-WIP/pattern_dedup \
disable=pstats \
dispose=keep
dt: End of Write pass 0/1, 204800 blocks, 100.000 Mbytes, 804 records, errors 0/1, elapsed 00m02.80s
dt: End of Read pass 1/1, 204800 blocks, 100.000 Mbytes, 804 records, errors 0/1, elapsed 00m02.43s

Total Statistics:
  Output device/file name: /var/tmp/dt.data (device type=regular)
  Type of I/O's performed: sequential (forward)
  Data pattern file used: /u/rtmiller/Tools/dt.d-WIP/pattern_dedup (81920 bytes)
Data pattern read/written: 0x00000000 (first 4 bytes)
  Total records processed: 1608 with min=4096, max=262144, incr=4096
  Total bytes transferred: 209715200 (204800.000 Kbytes, 200.000 Mbytes)
  Average transfer rates: 40098509 bytes/sec, 39158.700 Kbytes/sec
  Number I/O's per second: 307.457
  Total passes completed: 1/1
  Total errors detected: 0/1
    Total elapsed time: 00m05.23s
    Total system time: 00m00.17s
    Total user time: 00m00.75s
```

Starting time: Tue May 20 12:59:56 2008
 Ending time: Tue May 20 13:00:01 2008

sf49ers02%

The iozone tool also provides options to generate datasets to test dedupe.

File/LUN Copy and Verify

This example shows how to use *dt* to copy and verify a file. The file can already exist, as in this example, or you can use create the file with *dt* and use "dispose=keep" so it isn't deleted at the end of a test. Although this example shows copying a file, you can copy entire LUNs and/or copy/verify to tapes. You can also use "iomode=verify" to just read and verify the data matches.

```
roanoke% dt if=/kernel/genunix      \
          min=b                      \
          max=256k                   \
          incr=var                   \
          of=/var/tmp/genunix        \
          iomode=copy
```

Copy Statistics:

```
Data operation performed: Copied '/kernel/genunix' to '/var/tmp/genunix'.
Total records processed: 6 with min=512, max=262144, incr=variable
Total bytes transferred: 1555372 (1518.918 Kbytes, 1.483 Mbytes)
Average transfer rates: 25922867 bytes/sec, 25315.299 Kbytes/sec
Number I/O's per second: 100.000
Total passes completed: 0/1
Total errors detected: 0/1
Total elapsed time: 00m00.06s
Total system time: 00m00.01s
Total user time: 00m00.00s
Starting time: Tue Oct 21 16:44:27 2008
Ending time: Tue Oct 21 16:44:28 2008
```

Verify Statistics:

```
Data operation performed: Verified '/kernel/genunix' with '/var/tmp/genunix'.
Total records processed: 6 with min=512, max=262144, incr=variable
Total bytes transferred: 1555372 (1518.918 Kbytes, 1.483 Mbytes)
Average transfer rates: 51845733 bytes/sec, 50630.599 Kbytes/sec
Number I/O's per second: 200.000
Total passes completed: 1/1
Total errors detected: 0/1
Total elapsed time: 00m00.03s
Total system time: 00m00.01s
Total user time: 00m00.02s
Starting time: Tue Oct 21 16:44:27 2008
Ending time: Tue Oct 21 16:44:28 2008
```

Total Statistics:

```
Input device/file name: /kernel/genunix (device type=regular)
Type of I/O's performed: sequential (forward, rseed=0)
Total records processed: 12 with min=512, max=262144, incr=variable
Total bytes transferred: 7776860 (7594.590 Kbytes, 7.417 Mbytes)
Average transfer rates: 86409556 bytes/sec, 84384.332 Kbytes/sec
Number I/O's per second: 133.333
Total passes completed: 1/1
Total errors detected: 0/1
```

```

Total elapsed time: 00m00.09s
Total system time: 00m00.02s
Total user time: 00m00.02s
Starting time: Tue Oct 21 16:44:27 2008
Ending time: Tue Oct 21 16:44:28 2008

```

```

roanoke% cmp /kernel/genunix /var/tmp/genunix
roanoke% rm /var/tmp/genunix
roanoke%

```

Sparse File Testing

To create a sparse file, use the step=value option, which seeks over specific value to create holes in the file. These holes should read as zero, since nothing was written to them.

In the example, records of 16k are written, with holes of 32k, up to 50m bytes of data. The file disposition is set to "keep" so the file isn't deleted, then the holes in the file are read to verify zeroes are read back.

```

roanoke% dt of=/var/tmp/dt.data      \
      bs=16k                        \
      step=32k                      \
      disable=pstats                \
      dispose=keep                  \
      limit=50m
dt: End of Write pass 0/1, 102400 blocks, 50.000 Mbytes, 3200 records, errors 0/1, elapsed 00m02.87s
dt: End of Read pass 1/1, 102400 blocks, 50.000 Mbytes, 3200 records, errors 0/1, elapsed 00m01.06s

Total Statistics:
  Output device/file name: /var/tmp/dt.data (device type=regular)
  Type of I/O's performed: sequential (forward)
  Data pattern read/written: 0x39c39c39
  Total records processed: 6400 @ 16384 bytes/record (16.000 Kbytes)
  Total bytes transferred: 104857600 (102400.000 Kbytes, 100.000 Mbytes)
  Average transfer rates: 26681323 bytes/sec, 26055.980 Kbytes/sec
  Number I/O's per second: 1628.499
  Total passes completed: 1/1
  Total errors detected: 0/1
  Total elapsed time: 00m03.93s
  Total system time: 00m00.46s
  Total user time: 00m01.27s
  Starting time: Fri Oct 24 18:49:43 2008
  Ending time: Fri Oct 24 18:49:47 2008

```

```

roanoke% dt if=/var/tmp/dt.data      \
      position=16k                  \
      bs=32k step=16k              \
      disable=pstats                \
      limit=50m                    \
      pattern=0

```

```

Total Statistics:
  Input device/file name: /var/tmp/dt.data (device type=regular)
  Type of I/O's performed: sequential (forward)
  Data pattern read: 0x00000000
  Total records processed: 1600 @ 32768 bytes/record (32.000 Kbytes)
  Total bytes transferred: 52428800 (51200.000 Kbytes, 50.000 Mbytes)
  Average transfer rates: 42625041 bytes/sec, 41626.016 Kbytes/sec
  Number I/O's per second: 1300.813
  Total passes completed: 1/1

```

```
Total errors detected: 0/1
Total elapsed time: 00m01.23s
Total system time: 00m00.14s
Total user time: 00m01.09s
Starting time: Fri Oct 24 18:50:51 2008
Ending time: Fri Oct 24 18:50:52 2008
```

```
roanoke% ls -ls /var/tmp/dt.data
102576 -rw-r--r--    1 rtmiller engr      157253632 Oct 24 18:49 /var/tmp/dt.data
roanoke% rm /var/tmp/dt.data
roanoke%
```

Data Generation Tool

dt can be used as a data generation tool. That is, *dt* can generate the data to be used by other programs.

Here's an example of using *dt* to pipe its' data to the SCSI pass through (spt) program:

```
[root@sf49ers02 tmp]# dt of=- bs=512 count=16 disable=verify,stats | \
                        spt dsf=/dev/sdc din=- cdb="0a 1f ff f0 10 0 " dir=write length=8192
[root@sf49ers02 tmp]#
```

In the above example, *dt* writes 16 blocks of its' default data pattern to *spt* which then uses this data to write 16 blocks to the underlying SCSI disk.

Likewise, *dt* can be used to verify previously written data now being read via:

```
[root@sf49ers02 tmp]# spt dsf=/dev/sdc dout=- cdb="08 1f ff f0 10 0 " dir=read length=8192 | \
                        dt if=- bs=512 count=16 disable=stats
[root@sf49ers02 tmp]#
```

Tools such as *tar*, *pax*, and others that allow reading standard in (stdin) or writing standard out (stdout), can be used with *dt* in similar ways.

Butterfly Effect

The butterfly effect does I/O between the outer and inner parts of the disk, continuing until the two meet in the middle of the disk. Although *dt* does not have an I/O direction option to mimic the butterfly effect exactly, this script illustrates how two processes can simulate this.

Note: The test below is performed on a SAN LUN, but could be adapted to a large disk file.

```
#!/bin/ksh
#
# Script: butterfly.ksh
# Author: Robin T. Miller
# Date:   May 1st, 2008
#
# Description:
#           This script uses the data test (dt) program, to force I/O to be
#           like a butterfly, that is from front to back and visa versa.
#
# Please Note: The I/O pattern won't alternate every other I/O, due to
# process scheduling, and possibly reordering by the I/O subsystem.
```

```
#
# Note: On Linux, block device special files (DSF's) should either be
# bound to /dev/rawNN DSF's via raw(8) interface, or Direct I/O (DIO)
# should be enabled to bypass the buffer cache and its' affects.
#
alias dt=/u/rtmiller/Tools/dt.d-WIP/linux2.6-x86/dt
#
# Create two processes writing from each end of the disk to the middle.
#
dt of=/dev/sdc bs=256k flags=direct slices=2 slice=1 iodir=forward enable=debug pattern=0xaaaaaaaa 1
dt of=/dev/sdc bs=256k flags=direct slices=2 slice=2 iodir=reverse enable=debug pattern=0xbbbbbbbb 1
#
# Keep it simple (for now), just wait for all jobs.
# Note: Exit status should be gathered for each process.
#
set -m
wait
exit $?
sf49ers02# scu -f /dev/sdc show cap
```

Disk Capacity Information:

```
Maximum Capacity: 2097152 (1024.000 megabytes)
Block Length: 512
```

```
sf49ers02# ./butterfly.ksh
[2] + Done (254)          /u/rtmiller/Tools/dt.d-WIP/linux2.6-x86/dt of=/
[1] - Done                /u/rtmiller/Tools/dt.d-WIP/linux2.6-x86/dt of=/
sf49ers02#
```

Note: The exit status of 254 indicates end of file was reached.

Emulating crud Tool

The PTC folks have a tool named *crud*, which does the following:

- create 32 threads
- each thread does:
 - ◆ opens a unique file
 - ◆ seeks to (i * STRIPE_WIDTH - WRITE_OFFSET) for 2 iterations (i = 1,2)
 - ◆ writes WRITE_SIZE bytes
 - ◆ flush the data to disk
 - ◆ truncates the file
 - ◆ seeks to (i * STRIPE_WIDTH - WRITE_OFFSET) for 2 iterations (i = 3,4)
 - ◆ writes WRITE_SIZE bytes
 - ◆ flush the data to disk
 - ◆ close file
 - ◆ delete the file
- execute the above sequence for specified runtime (300 seconds by default)

The sample script below emulates this as closely as possible using *dt*, and is meant to show how creative test scripts can be developed without writing C programs.

Note: This example is **not** meant to replace the C *crud*, which consumes less system resources using threads, but is rather to show *dt*'s flexibility.


```

#!/bin/ksh
#
# File:          crud.ksh
# Author:        Robin T. Miller
# Date:          December 22nd, 2008
#
# Description:
#       A Korn shell script using the datatest (dt) program to
#       simulate the I/O behaviour implemented within the crud program.
#
# Note: The original crud in cluster/fs/crud/linux/crud, is a
# multi-threaded tool. Since dt is not multi-threaded at present,
# multiple processes are used to closely simulate the crud tool.
# Also, the original crud writes two stripes, truncates the file,
# then writes the last two stripes, and does not verify the data.
# The C crud tool reopens and continues after errors occur.
#
# RFE's? Allow nodes, threads, and stripe width to be arguments.
#
typeset prog=${0##*/}
typeset DT_PATH=${DT_PATH:-$(whence dt)}
typeset DT_PATH=${DT_PATH:-/usr/software/test/bin/dt.latest}

#
# Local Definitions:
#
typeset NODES=${NODES:-4}
typeset -i THREAD_COUNT=${THREAD_COUNT:-32}
typeset STRIPE_WIDTH=${STRIPE_WIDTH:-2*1024*1024}
(( STRIPE_WIDTH=$STRIPE_WIDTH ))
typeset -i WRITE_OFFSET=${WRITE_OFFSET:-4096}
typeset -i WRITE_SIZE=${WRITE_SIZE:-8192}

function usage
{
    print "Usage: $prog <filename> [runtime]"
    exit 1
}

#
# Expect Two Arguments:
#
[[ $# -lt 1 ]] && usage
[[ $# -gt 2 ]] && usage

#
# Parse The Arguments: (one required, one optional)
#
typeset FILE=$1
if [ $# -eq 2 ] ; then
    typeset RUNTIME=$2
else
    typeset RUNTIME=300
fi

#
# Show Tool Version:
#
${DT_PATH} version

#
# Execute the Command Line:

```

```
#
# Note: Add enable=Debug to see seek/write/read operations!
#
${DT_PATH} of=$FILE \
           bs=${WRITE_SIZE} \
           position="${STRIPE_WIDTH}-${WRITE_OFFSET}" \
           count=${NODES} \
           step=${STRIPE_WIDTH} \
           oflags=trunc \
           procs=${THREAD_COUNT} \
           runtime=${RUNTIME} \
           disable=pstats,verbose

exit $?
```