DT TOOL OVERVIEW

1 Overview

The datatest (dt) tool is open source and has been kicking around for over 30 years! Yea, likely older than many reading this! 😊

This tool was originally developed for testing tape drives and a streaming tape driver I wrote for SunOS in the late 1980's. Later, support for testing direct disks, file systems, memory mapped I/O, asynchronous I/O, serial lines, parallel lines, pipes, and pretty such any device that support the standard open/close/read/write API's can be tested with dt. But that said, most of these other devices are no longer compiled into the current executable.

Beyond the goal to support any device, another goal was to provide a flexible set of options so users can essentially roll-their-own tests. The tool grew from a single threaded single device tool, to providing multiple devices with multiple threads in a single invocation, reducing process overhead, sharing execution code, and allowing multiple jobs/threads to be easily controlled.

With SCSI disks, you can even switch to direct SCSI CDB's via the OS SCSI Pass-Through (SPT) API. For Linux, direct NVME I/O is also supported.

Since dt was developed for doing device qualification and device driver verification, data validation is always enabled by default, and over the years higher levels of data validation were achieved by an pattern known as IOT, prefixes, and most recently block tagging. These features generate unique data per block and provide sufficient information to help with troubleshooting data corruptions. Support for compression and de-duplication is currently achieved via a pattern file.

Folks familiar with the Unix *dd* utility, will be familiar with many of its' options: **if**= (input file) **of**= (output file), **bs**= (block size), etc

In addition to high data validation, dt supports:

- multiple I/O behaviors
- dynamic job control
- I/O monitoring
- keep alive messages
- internal/external triggers
- extended error reporting
- map file offsets to physical LBA's
- I/O via direct SCSI or NVME API's
- corruption analysis (for IOT pattern)
- SCSI UNMAP (via *spt* tool)
- I/O latency reporting
- predefined workloads
- copying from disk to disk or file system to file system
- copy/verify with mirror disks is also supported
- communicate via CLI, pipes, or script files

After so many years, dt is mature and feature rich, suitable for most testing needs!

Please review dt Release Notes for other features added since this overview was written.

FYI: Each I/O tool has its' own pros and cons, but **dt** is one of the best at what it does!

Disclaimer: Please know, the addition of this dt tool is **not** intended to replace your favorite tool(s), but rather to augment tools in your toolbox! My belief is that multiple I/O tools provide better coverage and quality, since each has its' own strengths, I/O footprint, and timing, so I urge folks to use the best tool suited to their requirements.

1.1 Other Key Tool Considerations

Beyond the tool features, these are a few other key things to consider:

- do you *trust* the tool?
- does the tool find problems?
- does it report useful information?
- can you easily reproduce failures?
- is it reducing time to resolve issues?
- does it support troubleshooting features?
- does it provide the ability to trace operations?
- is it maintained and supported, and can you get help?

FWIW: Doing simple write/read/verify operations is one of the least interesting parts of an I/O tool, at least for me. All these other aspects are equally important (IMHO).

2 Documentation

- dt-UsersGuide.doc - dt User's Guide
- DTv14tov21Overview.pptx - Differences between versions 14 and 21.
- Other documentation links

But to be honest, "**dt help**" provides extensive help text and is kept up to date!

2.1 GitHub Repositories

Please find the Open Source dt source repository here:

- The Open-Source version is here: https://github.com/RobinTMiller

2.2 Training Sessions

I have several training sessions available, including:

1. Overview.
2. Workloads / Use Cases – Built-in workloads and Use Cases.
3. Advanced Usage - Dynamic job control, dt scripts, example scripts.
4. Troubleshooting - dt options to help with triage and troubleshooting.
5. Data Corruptions - Usually included with Troubleshooting, but it's a big topic.
6. For teams doing SCSI testing, the SCSI Pass-Through (spt) tool is available.

I have given presentations to larger groups, or to smaller groups focused on your test requirements.

Please contact Robin for group specific training, describing the key tool requirements for your team.

3 Tool Location

From GitHub, download the OS specific dt executable or simply clone the repository.

3.1 Tool Invocation

There are multiple ways to invoke dt, including:

- Command Line Interface (CLI) - all tools usually support this
- Script Files - multiple dt commands, can be nested 5 levels deep
- Pipe Interface - send commands and receive output via pipes
- Interactive - enter interactive mode when no command is provided

4 I/O Control

There are numerous options to control the I/O direction, type, direction, limits, etc:

- **bs=value** - The block size to read/write. (Default: 512)
    - **bs=random** - Random size between 512 and 256k.
- or control input/output block sizes independently:
    - **ibs=value** - The read block size. (overrides bs=)
    - **obs=value** - The write block size. (overrides bs=)
- **min=value** - Set the minimum record size to transfer.
- **max=value** - Set the maximum record size to transfer.
- **incr=value** - Set number of record bytes to increment.
    - or **incr=variable** - Enables variable I/O request sizes.
- **capacity=value** - Set the device capacity in bytes.
- **capacityp=value** - Set capacity by percentage (range: 0-100).
- **iodir=direction** - Set I/O direction to: {**forward**, reverse, or vary}.
- **iomode=mode** - Set I/O mode to: {copy, mirror, **test**, or verify}.
- **iops=value** - Set I/O per second (this is per thread).
- **iotype=type** - Set I/O type to: {random, sequential, or vary}.
- **seed=value** - To repeat a previous I/O sequence. (64 bit value)
- **readp=value** - Percentage of accesses that are reads. Range [0,100].
    - 'random' keyword makes the read/write percentage random.
- **randp=value** - Percentage of accesses that are random. Range [0,100].
    - Sequential accesses = 0%, else random percentage
- **rrandp=value** - Percentage of read accesses that are random. Range [0,100].
- **wrandp=value** - Percentage of write accesses that are random. Range [0,100].

There's also a wide range of options to control directories, files, file sizes, etc.

Please know, for direct disk testing the disk capacity and block size is acquired automatically.

Also be aware that random I/O does *not* write every block (no random I/O map maintained). Instead, the same random seed is used during each write/read pass, ensuring the same set of blocks are accessed. Overwrites may occur, and usually do, during random I/O, thus safe patterns such as IOT and/or IOT with block tags *must* be used to avoid false data corruptions.

POSIX AIO is supported along with multiple threads to generate high load and stress.

Note: The native Linux libaio interface has **not** been implemented, recommend using threads.

5 Data Patterns

There are numerous ways to specify built-in data patterns, a pattern file, a pattern prefix string, and a block tag for each.

The options related to pattern generation are: (the pattern= option is overloaded, but kept for backwards compatibility)

- **pattern=value** The 32 bit hex data pattern.
    - there are 13 internal patterns, rotated across each pass
    - the default pattern, if not specified, is pattern=0x39c39c39
- **pattern=iot** - Special IOT test pattern.
- **pattern=incr** - Use an incrementing data pattern.
- **pattern=string** - The string to use for the data pattern. (don't start with hex characters)
- **prefix=string** - The data pattern prefix string. (roll you own prefix string for each block)
- **pf=filename** - The data pattern file to use. (the entire pattern file read into memory)
- **enable=btags** - Adds a block tag prior to the prefix string and/or data pattern bytes.

A combination of these options allows for very flexible pattern generation.

A set of pattern files are available in repository directory: **dt/data/**

Compression and deduplication is currently accomplished via **pattern_all** and **pattern_dedup** files.

5.1 IOT Pattern

This special IOT pattern starts with the LBA in the 1st 4 bytes, then a seed constant is added to each 32-bit word throughout the block to create unique data per block. The IOT seed starts with 0x01010101, then for each pass this seed is multiplied by the pass count to ensure the data changes on each subsequent pass.

```
000000 00 00 00 00 01 01 01 01 02 02 02 02 03 03 03 03

000010 04 04 04 04 05 05 05 05 06 06 06 06 07 07 07 07
```

```
  . . .
0001e0 78 78 78 78 79 79 79 79 7a 7a 7a 7a 7b 7b 7b 7b

0001f0 7c 7c 7c 7c 7d 7d 7d 7d 7e 7e 7e 7e 7f 7f 7f 7f
```

When used with a prefix string, the block is traceable to the host and device the data belongs to.

For example, a **prefix='%d@%h'** expands to the file/device and host name: **iot.data@rtpqa-rhel001**

Here are options for very high validation: **enable=btags pattern=iot prefix='%d@%h'**

FYI: Instead specifying the above options, this workload template is predefined: **workload=high_validation**

The block tag layout (btag) is discussed in a later section and includes a lot of useful information.

FYI: I've toyed with expanding the 4-byte IOT LBA to 64-bits, since this wraps at 2TB, but for now add **dsize=4k** to artificially set the block size to 4k versus 512 bytes. In fact, many disk drives and storage arrays support a 4k block size to avoid misaligned block access

5.2 Job Control

For advanced test cases, job control is available for all methods except CLI.

In a nutshell, what is job control?

- commands to control background (async) jobs
- ability to query job statistics or modify debug options on the fly
- ability to pause, resume, or stop/cancel jobs
- group jobs by tags and control via same tag name
- ability to wait for job(s) to complete

6 Log Files

Because the varying needs of many testers, a flexible set of log files is available:

- thread logs (**log=**) - individual thread logs
- job logs (**job_log=**) - job log, one or more threads
- error log (**error_log=**) - error log, all errors appended
- master log (**master_log=**) - combination of all job/thread logs

The job and master logs are synchronized with locks, so nothing gets lost or overwritten!

The error log is useful to monitor errors without searching log files. Generally, the error logs are deleted prior to starting dt workloads, then when errors occur the error log is created and a summary of the error information is appended to the log file. Checking for the existence of the error log quickly tells you if an error has occurred on any of the device threads running.

Another common option is "**enable=syslog**" to log messages to the host system log. When enabled, the start time with command line is logged, errors are logged, and test finish time is logged. This can be useful in conjunction with other kernel messages when errors occur.

dt uses a default logging prefix, and like most things, this can be redefined to match your preference!

- default log prefix: "**%prog (j:%job t:%thread):** ", which looks like this: "**dt (j:1 t:2):** "
- log w/timestamps: "logprefix='%seq %date %et %prog (j:%job t:%thread): '"
    - results in: "   **0 Fri May 31 11:29:42 2019 00m00.01s dt (j:1 t:1):** message..."

Note: Timestamps are not enabled by default, since many automation frameworks add their own.

The above timestamp is available via this template: **workload=log_timestamps**

The "%et" format control string reports the elapsed time since the thread started. (I hate doing math, that's what computers are for! 🙂)

7 Load and Stress

Within a single invocation, you can start jobs across multiple devices/file systems, with each job using multiple threads or slices and/or asynchronous I/O (AIO) to generate higher load and stress. On Unix POSIX AIO and threads are used, on Windows native AIO and threads are used.

The options include:

- **aios=value** - Set number of AIO's to queue.
- **slices=value** - The number of disk slices to test.
- **threads=value** - The number of threads to execute.
- **enable=iolock** - I/O lock control. (modifies threads behavior)

Note: Using POSIX AIO on Linux is *not* recommended since this is implemented via a user level thread library. With other Unix OS's, aio_read() and aio_write() are implemented via light weight threads in the kernel, so provide much higher performance. Linux decided to create its' own native AIO library and kernel interface, but this is **not** portable so has not been implemented in dt yet. This latter Linux libaio method is supported by the *fio* tool, and does provide the highest Linux I/O performance.

7.1 I/O Lock Behavior

Normally, when threads are created, they operate independently on the same range of blocks.

But, when adding **enable=iolock**, a set of threads operate nicely in concert synchronized by a lock (mutex). While this provides the best performance across multiple threads in sequential mode, too many threads generate high lock contention and hurts performance (YMMV).

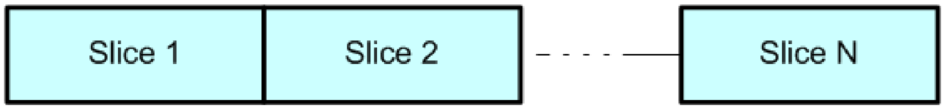
7.2 Slice and Dice

So, what's this slice option all about? A slice is a thread with a range of blocks assigned to it!

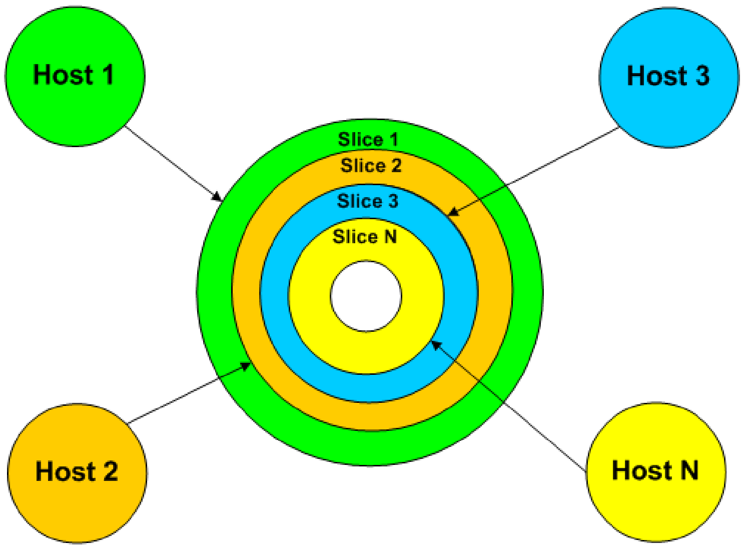
When specifying **slices=*value***, dt divides up the data limit into *value* slice ranges, doing all the calculations internally so you don't have to. 🙂

All slices will be equal in size, except residual blocks are added to the final slice, as required.

Therefore, each slice will operate on a range of blocks:



This option was originally added for multi-initiator host testing, so each host has its' own slice:



dt slices=N slice=1-N, so each host does I/O to its' own range of blocks.

But this option has proven useful for both disks and file systems, since multiple slices generate higher load and stress. Due to the nature of the slices, one can envision this is a form of random

I/O, so for rotating hard drives there's a lot of head movement. But, for virtual storage on arrays (LUNs) and SSD, slices still generate high performance.

7.2.1 Example Slice Debug Information

```
[root@rtpqa-rhel001 romiller]# dt of=/dev/mapper/mpathg count=1 slices=10 slice=5 enable=debug disable=stats,scsi_info
00m00.00s dt (j:0 t:0): Cloning device 0xba7010
00m00.00s dt (j:0 t:0): BLKSSZGET Sector Size: 512 bytes
00m00.00s dt (j:0 t:0): BLKGETSIZE Capacity: 61440000 blocks (512 byte blocks).
00m00.00s dt (j:0 t:0): Data limit set to 31457280000 bytes (30000.000 Mbytes), 61440000 blocks.
00m00.00s dt (j:0 t:0): Cloning device 0xbb0470 - /dev/mapper/mpathg...
00m00.00s dt (j:0 t:0): The Monitor Thread ID is 0x7fd87b137700
00m00.01s dt (j:1 t:1): Starting I/O, Job 1, Thread 1, Thread ID 0x7fd87a76e700
00m00.01s dt (j:1 t:1):
00m00.01s dt (j:1 t:1): Slice 5 Information:
00m00.01s dt (j:1 t:1):                  Starting offset: 12582912000 (lba 24576000)
00m00.01s dt (j:1 t:1):                    Ending offset: 15728640000 (lba 30720000)
00m00.01s dt (j:1 t:1):                     Slice length: 3145728000 bytes (6144000 blocks)
00m00.01s dt (j:1 t:1):                       Data limit: 3145728000 bytes (6144000 blocks)
00m00.01s dt (j:1 t:1):                     Random range: 12582912000 (lba 24576000) - 15728640000 (lba 30720000)
00m00.01s dt (j:1 t:1): Opening output file /dev/mapper/mpathg, open flags = 040001 (0x4001)...
00m00.01s dt (j:1 t:1):
00m00.01s dt (j:1 t:1):                        File Name: /dev/mapper/mpathg
00m00.01s dt (j:1 t:1):                        Operation: open
00m00.01s dt (j:1 t:1):                         I/O Mode: write
00m00.01s dt (j:1 t:1):                         I/O Type: sequential
00m00.01s dt (j:1 t:1):                        File Type: output
00m00.01s dt (j:1 t:1):             Device Serial Number: 81e726e1cfac11996c9ce9002d492233
00m00.01s dt (j:1 t:1):                Device Identifier: 81e7-26e1-cfac-1199-6c9c-e900-2d49-2233
00m00.01s dt (j:1 t:1):                   Desired Access: 0x4001 = O_WRONLY|O_DIRECT
```

8 I/O Monitoring

The purpose of dts' I/O monitoring is to detect long outstanding or hung I/O. Oddly enough, this monitoring feature has been one of the most popular features and is key to ensure component System Level Agreement (SLA) is met, from I/O with disruptions (path fail overs, etc), to volume moves, or anything else timing critical.

The monitoring is done via a thread, thereby detecting long I/O resumptions times while operations are still outstanding. This includes open, close, read, write, and flush operations.

The dt I/O monitoring is ***not*** enabled by default, but key standard built-in workloads have it enabled by default.

The options provided give the user complete control and include:

1. Low level threshold of when to start reporting long I/O. (**noprogt=value**)
2. An upper threshold of when to take action. (**noprogtt=value**)
3. The upper threshold can invoke a trigger: (**trigger=action,...**)

    - an external script/program (**trigger=cmd:program**)
    - send a SCSI command to trigger an analyzer (thus the name)
    - triage option to collect SCSI device state (Inquiry, TUR, etc) (**trigger=triage**)
    - trigger scripts exit status controls action thereafter (continue or exit)
    - FWIW: triggers are also invoked for I/O errors and data corruptions:
        - **trigger_on={all, errors, miscompare, or noprogs}**
    - Longevity I/O Manager workloads include: **trigger_on=miscompare**
        - therefore, triggers are *only* executed for data corruptions (miscompares)

- Monitoring is done on all device operations: open, close, flush, read, write, etc
  - an option exists to control which API's to monitor (**notime=optype**)

4. If triggers are **not** specified, and the trigger time is exceeded, the job will be stopped (by default).

   - please know, a stop state is set, but dt does **not** attempt to abort outstanding I/O
   - what this means is hung I/O will remain outstanding, and multiple noprog messages will persist
     that said, there's logic expecting jobs to terminate within a certain amount of time (**180 seconds default**)
     - if this terminate wait time (**term_wait=time**) is exceeded, dt will attempt to cancel the job threads
     - but please know, I/O hung hard in the kernel, will usually prevent threads from exiting! 😞

Note: Multiple triggers can be specified and often are.

8.1 Verifying I/O Monitor Behavior

Please know, the *forced_delay=value* option is available to help verify long I/O detection and trigger actions:

- **noprogt=1 noprogtt=3 trigger_on=miscompare forced_delay=7**

When the upper trigger time is exceeded or when noprog triggers are disabled, then dt will stop the current job.

The above will result in messages such as this:

```
root@rtp-smc-qa18-4 ~# dt of=/var/tmp/dt.data limit=1m noprogt=1 noprogtt=3 enable=raw
trigger_on=miscompare forced_delay=7 disable=stats

dt (j:1 t:1): No progress made for record 1 (lba 0, offset 0) during write() on /var/tmp/dt.data for
2 seconds! Since: Tue Oct 5 13:20:24 2021
dt (j:1 t:1): No progress made for record 1 (lba 0, offset 0) during write() on /var/tmp/dt.data for
3 seconds! Since: Tue Oct 5 13:20:24 2021
dt (j:1 t:1): No progress made for record 1 (lba 0, offset 0) during write() on /var/tmp/dt.data for
4 seconds! Since: Tue Oct 5 13:20:24 2021
dt (j:1 t:1): This requests' elapsed time of 4, has exceeded the noprogtt of 3 seconds!
dt (j:1 t:1): The current time is: 1633454428 seconds => Tue Oct 5 13:20:28 2021
dt (j:1 t:1): The initiated time was: 1633454424 seconds => Tue Oct 5 13:20:24 2021
dt (j:1 t:1): Note: For more information regarding noprog's, please visit this link:
dt (j:1 t:1): <vendor link can be added here, update dt.h then recompile with updated URL>
dt (j:1 t:1): ERROR: Error number 1 occurred on Tue Oct 5 13:20:28 2021
dt (j:1 t:1): ERROR: No triggers or noprog triggers are not enabled, so stopping this job
```

and its' threads...
```
dt (j:0 t:0): Job 1 is being stopped (1 thread)
```

Please Note: Earlier versions of dt did **not** stop jobs when **trigger_on=miscompare** was specified. This was not expected behavior so has been corrected!

8.2 Example I/O Monitoring Messages

The Longevity workloads use these options: **noprogt=30s noprogtt=5m**

This means dt will start warning of outstanding I/O after 30 seconds, then take trigger action after 5 minutes.

```
28784 2021-04-18,13:25:16 1d19h57m57.00s dt.exe (j:10 t:4): No progress made for record
2223 (lba 1173898, offset 601035776) during write() on G:\dt-Administrato
r-0b17d996-58bb-4797-ba74-1f7eb2625ddf.data-j10t4 for 31 seconds! Since: Sun Apr 18
13:24:45 2021
28785 2021-04-18,13:25:17 1d19h57m58.00s dt.exe (j:10 t:4): No progress made for record
2223 (lba 1173898, offset 601035776) during write() on G:\dt-Administrato
r-0b17d996-58bb-4797-ba74-1f7eb2625ddf.data-j10t4 for 32 seconds! Since: Sun Apr 18
13:24:45 2021
28786 2021-04-18,13:25:18 1d19h57m59.00s dt.exe (j:10 t:4): No progress made for record
2223 (lba 1173898, offset 601035776) during write() on G:\dt-Administrato
r-0b17d996-58bb-4797-ba74-1f7eb2625ddf.data-j10t4 for 33 seconds! Since: Sun Apr 18
13:24:45 2021
```

The above messages continue up to the upper limit, shown below:

```
29052 2021-04-18,13:29:46 1d20h02m27.00s dt.exe (j:10 t:4): No progress made for record
2223 (lba 1173898, offset 601035776) during write() on G:\dt-Administrator-0b17d996-58bb-
4797-ba74-1f7eb2625ddf.data-j10t4 for 301 seconds! (05m01.00s) Since: Sun Apr 18 13:24:45
2021
29053 2021-04-18,13:29:46 1d20h02m27.00s dt.exe (j:10 t:4): This requests' elapsed time of
301, has exceeded the noprogtt of 300 seconds!
29054 2021-04-18,13:29:46 1d20h02m27.00s dt.exe (j:10 t:4): The current time is: 1618777786
seconds => Sun Apr 18 13:29:46 2021
29055 2021-04-18,13:29:46 1d20h02m27.00s dt.exe (j:10 t:4): The initiated time was:
1618777485 seconds => Sun Apr 18 13:24:45 2021
```

9 I/O Throttling

When many hosts are driving I/O to the same storage, you may wish to throttle the I/O to overdriving the storage.

- **iops=value** - Set I/O per second (this is per thread).

- **iotune=filename -** Set I/O tune delay parameters via file.

Today dt only throttles I/O down by injecting necessary delays. Throttling up is not currently implemented.

The I/O tune file can be used by external monitoring software to pass I/O delays to dt via a file via normal *delay options. But that said, intelligent automation can utilize job control features.

10 Workloads

So, you may be asking yourself, what the heck can I do with this data test tool?

Before you start defining your own workloads, please use "**dt workloads**" to see a list of internal workloads, and their definitions. You may find an existing workload already exists, such as **san_disk** or **san_file_system**, or you can augment those workloads with additional options, which is what I often do. Recently, several workloads were added for the Longevity team, whose definitions can be seen via "**dt workloads longevity**".

For example:

```
robin@DESKTOP-SBC6MG3 ~/GitHub/dt/windows/x64/Release
$ ./dt.exe workloads longevity
Valid Workloads:

    longevity_common: Longevity Common Options (template)
        min=8k max=1m incr=vary enable=raw,reread,log_trailer,syslog
history=5 history_data=152 enable=history_timing logprefix='%seq %nos %et
%prog (j:%job t:%thread): ' keepalivet=5m runtime=-1 onerr=abort noprogt=30s
noprogtt=5m stopon=C:\temp\stopit

    longevity_file_dedup: Longevity File System w/Dedup Workload
        workload=longevity_common min_limit=1m max_limit=2g incr_limit=vary
dispose=keep flags=direct notime=close,fsync oflags=trunc maxdatap=75
threads=4 pf=x:\noarch\dtdata\pattern_dedup

    longevity_disk_dedup: Longevity Direct Disk w/Dedup Workload
        workload=longevity_common capacityp=75 slices=4
pf=x:\noarch\dtdata\pattern_dedup

    longevity_file_system: Longevity File System Workload
        workload=longevity_common workload=high_validation min_limit=1m
max_limit=2g incr_limit=vary dispose=keep flags=direct notime=close,fsync
oflags=trunc maxdatap=75 threads=4

    longevity_disk_unmap: Longevity Direct Disk w/SCSI UNMAP Workload
        workload=longevity_common workload=high_validation capacityp=75
slices=4 unmap=unmap

    longevity_disk: Longevity Direct Disk Workload
        workload=longevity_common workload=high_validation capacityp=75
slices=4

    longevity_disk_write_only: Longevity Direct Disk Write Only
        workload=longevity_disk disable=raw,reread,verify
```

```
longevity_file_write_only: Longevity File System Write Only
             workload=longevity_file_system disable=raw,reread,verify

  robin@DESKTOP-SBC6MG3 ~/GitHub/dt/windows/x64/Release
  $
```

Note: The pattern file (pf=FILE) will need to be overridden with your data file location.

Linux Example:

# dt of=/dev/mapper/mpathf,/dev/mapper/mpathg workload=longevity_disk runtime=1h slices=10

Override options *must* come *after* the workload definitions to replace those in the workload.

Please know you can also define your own workloads via a *~/.datatestrc* file using the *define* option! 🙂

10.1 Defining Workloads

Here's a simple example of defining your own workload:

```
# cat ~/.datatestrc
define my_workload:"Simple workload example" workload=longevity_disk limit=25g
runtime=0 workload=job_stats_only
# dt workloads my_workload
Valid Workloads:

   my_workload: Simple workload example
      workload=longevity_disk limit=25g runtime=0 workload=job_stats_only

# export DISKS=/dev/mapper/mpathcp,/dev/mapper/mpathcq,/dev/mapper/mpathcr

      OR use SCSI pass-through tool:

# export DISKS=$(spt show devices spaths=/dev/mapper/mpath sfmt='%paths'
vid=<your vendor ID> | tr '\n' ',')

# dt of=${DISKS} workload=my_workload
```

Pretty simple, right? No recompiling the tool to add new workloads, extend or add your own! 🙂

Please know, workload definitions do **not** usually include the devices to test, since these will change.

10.2 Workload Templates

There are several workloads defined which are templates to be used with other workloads or your command line. These have been defined for ease of use, since the set of options can be lengthy and/or difficult to remember (even for the author). It also makes it easier for automation to help generate complex sets of options:

Here are a couple templates recently added for log file and log prefix:

all_logs: Define options for creating all logs (template)
  job_log='dt_job%job-%dsf.log' log='dt_thread-j%jobt%thread-%dsf.log' error_log=dt-ERROR.log

job_logs: Define options for creating job logs (template)
  job_log='dt_job%job-%dsf.log' error_log=dt-ERROR.log

thread_logs: Define options for creating thread logs (template)
  log='dt_thread-j%jobt%thread-%dsf.log' error_log=dt-ERROR.log

log_timestamps: Define options for adding log file timestamps (template)
  logprefix='%seq %date %et %prog (j:%job t:%thread): '

Please Note: The log directory is purposely omitted since this varies. Use the **logdir=path** option to specify the directory.

You will also notice the error log option is specified. Any errors from this invocation of dt will be reported to the error log.

The error log is appended to, so add **enable=deleteerrorlog** or delete this file first (as required).

Update: The error log file is now automatically deleted before starting workloads (jobs).

11 Simple Examples

Well, nothing is as simple as using the predefined workloads, but you can use very simple dt commands to get started while learning the tool.

A couple things to remember:

- a job is for a single device/file with one to many threads
- specifying an input file, **if=**, will enable read with compare (assumes data previously written)
- specifying an output file, **of=**, will write/read/verify which internally is considered a single pass
- specifying **if=** *and* **of=**, like the Unix *dd* utility, will copy input file to output file, with a read/verify pass
    - **src=** and **dst=** options can be used instead of **if=** and **of=** options:

    - # **dt src=/dev/mapper/mpathg dst=/dev/mapper/mpathh bs=256k slices=10**

- when you wish to write only, add **disable=verify** to skip the read/verify pass
- when you don't care about data written, add **disable=compare,verify** flags
- when you wish to read without comparing data, add **disable=compare**
- use **aios=**, **slices=**, or **threads=** w/**iolock** for higher performance
- use **disable=pstats** or **disable=stats** when statistics are not important (for triage, please include stats!)
- use **workload=job_stats_only** to report just job statistics (all thread stats accumulated)

11.1 Direct Disk Tests

```
# dt of=/dev/mapper/mpathh
```

A single thread doing sequential I/O is the default.

The default block size will be the disk block size, usually 512 bytes.

dt detects the disk capacity and block size automatically, via OS specific API's.

There are 13 internal fixed 32-bit data patterns dt will cycle through, by default.

A single write pass followed by a read/verify pass will be performed with statistics.

That's as simple as it gets, but using defaults is very slow and not very interesting (for most).

Note: I usually recommend selecting a predefined workload, then add options (as desired) to customize.

11.2 SCSI I/O with UNMAPs (thin provisioned array volumes)

This workload requires the SCSI-Pass-Through (spt) tool to perform the SCSI UNMAP requests.

```
# dt of=/dev/mapper/mpathh workload=disk_unmaps enable=scsi_io capacity=1g passes=2 runtime=0
```

At the end of a pass, a SCSI UNMAP will be performed on the range of blocks for that slice.

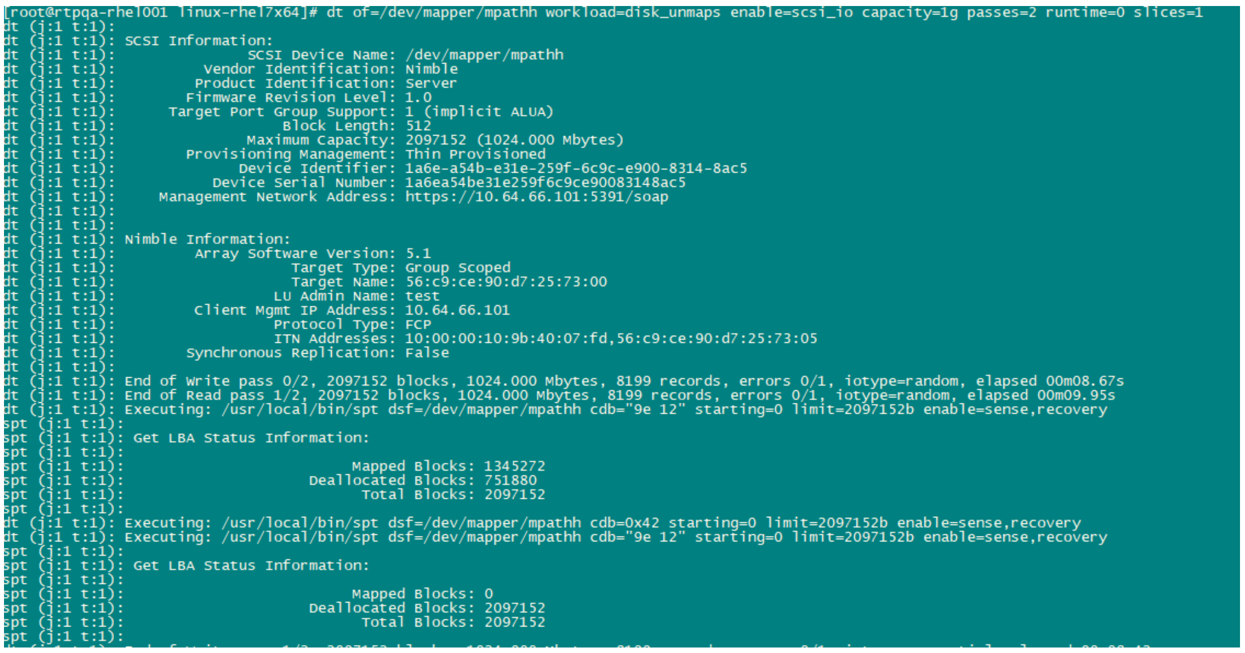
For Unix systems, spt is expected in **/usr/local/bin** (right now), but **spt_path=** can override this path.

BTW: dt supports unmapping blocks using one of these SCSI methods:

- **unmap=type** The SCSI unmap type.

Valid types are: **random, unmap, write_same, zerorod**.

You will see the SCSI Get LBA Status both before and after the UNMAP to show mapped vs. deallocated blocks:



```
[root@rtpqa-rhel001 linux-rhel7x64]# dt of=/dev/mapper/mpathh workload=disk_unmaps enable=scsi_io capacity=1g passes=2 runtime=0 slices=1
dt (j:1 t:1):
dt (j:1 t:1): SCSI Information:
dt (j:1 t:1):                SCSI Device Name: /dev/mapper/mpathh
dt (j:1 t:1):           Vendor Identification: Nimble
dt (j:1 t:1):          Product Identification: Server
dt (j:1 t:1):         Firmware Revision Level: 1.0
dt (j:1 t:1):       Target Port Group Support: 1 (implicit ALUA)
dt (j:1 t:1):                    Block Length: 512
dt (j:1 t:1):                Maximum Capacity: 2097152 (1024.000 Mbytes)
dt (j:1 t:1):         Provisioning Management: Thin Provisioned
dt (j:1 t:1):               Device Identifier: 1a6e-a54b-e31e-259f-6c9c-e900-8314-8ac5
dt (j:1 t:1):            Device Serial Number: 1a6ea54be31e259f6c9ce90083148ac5
dt (j:1 t:1):      Management Network Address: https://10.64.66.101:5391/soap
dt (j:1 t:1):
dt (j:1 t:1):
dt (j:1 t:1): Nimble Information:
dt (j:1 t:1):          Array Software Version: 5.1
dt (j:1 t:1):                     Target Type: Group Scoped
dt (j:1 t:1):                     Target Name: 56:c9:ce:90:d7:25:73:00
dt (j:1 t:1):                   LU Admin Name: test
dt (j:1 t:1):          Client Mgmt IP Address: 10.64.66.101
dt (j:1 t:1):                   Protocol Type: FCP
dt (j:1 t:1):                   ITN Addresses: 10:00:00:10:9b:40:07:fd,56:c9:ce:90:d7:25:73:05
dt (j:1 t:1):         Synchronous Replication: False
dt (j:1 t:1):
dt (j:1 t:1): End of Write pass 0/2, 2097152 blocks, 1024.000 Mbytes, 8199 records, errors 0/1, iotype=random, elapsed 00m08.67s
dt (j:1 t:1): End of Read pass 1/2, 2097152 blocks, 1024.000 Mbytes, 8199 records, errors 0/1, iotype=random, elapsed 00m09.95s
dt (j:1 t:1): Executing: /usr/local/bin/spt dsf=/dev/mapper/mpathh cdb="9e 12" starting=0 limit=2097152b enable=sense,recovery
spt (j:1 t:1):
spt (j:1 t:1): Get LBA Status Information:
spt (j:1 t:1):
spt (j:1 t:1):                           Mapped Blocks: 1345272
spt (j:1 t:1):                      Deallocated Blocks: 751880
spt (j:1 t:1):                            Total Blocks: 2097152
spt (j:1 t:1):
dt (j:1 t:1): Executing: /usr/local/bin/spt dsf=/dev/mapper/mpathh cdb=0x42 starting=0 limit=2097152b enable=sense,recovery
dt (j:1 t:1): Executing: /usr/local/bin/spt dsf=/dev/mapper/mpathh cdb="9e 12" starting=0 limit=2097152b enable=sense,recovery
spt (j:1 t:1):
spt (j:1 t:1): Get LBA Status Information:
spt (j:1 t:1):
spt (j:1 t:1):                           Mapped Blocks: 0
spt (j:1 t:1):                      Deallocated Blocks: 2097152
spt (j:1 t:1):                            Total Blocks: 2097152
spt (j:1 t:1):
```

11.2.1 UNMAP via Writing Zeros

Some storage, when blocks of zeros are written, will perform an implicit UNMAP of those blocks.

Therefore, in addition to the SCSI UNMAP operations described above, you can also write zero data to UNMAP blocks.

In the case of file system testing, prefilling files with a fill pattern of zero will also UNMAP blocks. Some modern file systems perform unmap, aka file trim operations, when files are deleted or truncated.

The dt prefill options were described earlier, but these will do the trick:

- **fill_pattern=0 enable=fill_always**

But that said, please know file systems return zero data for blocks never written (aka sparse files), so prefilling with zero may mask where the zero blocks originate from without I/O tracing at some level.

New in dt v22.0: For Linux and Windows, file offsets are now mapped to physical LBA's during corruptions, and if the offset is **not** in the extent map, dt reports "**<not mapped>**", which indicates the file system metadata does not believe the data at the requested offset has been written. Note, on Windows you must have elevated privileges such as Administrator.

11.3 File System Tests

dt assumes a file system has been created, it won't do this part for you!

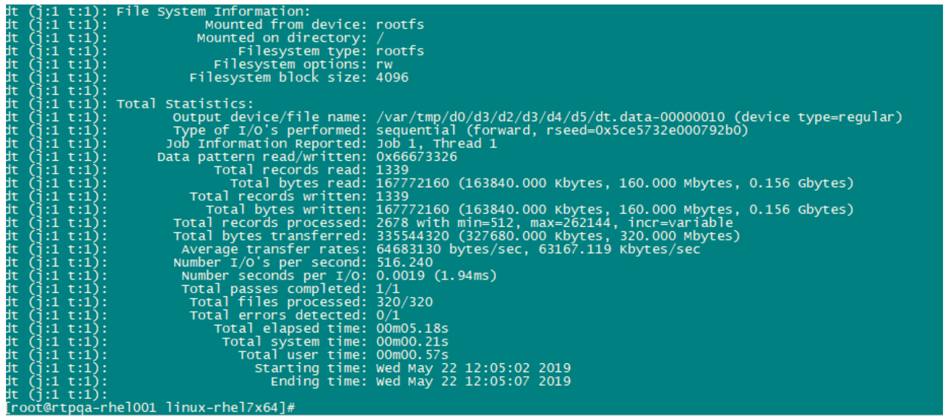
That said, dt will create multiple files, multiple directories and sub-directories.

By default, dt will delete anything it creates, unless **dispose=keep** is specified.

BTW: **dispose=keeponerror** is recommended so files persist after an error occurs.

```
# dt of=/var/tmp/dt.data bs=random limit=1m files=10 sdirs=3 depth=5
```

The above will create 320 files across 3 sub-directories each 5 levels deep!



```
dt (j:1 t:1): File System Information:
dt (j:1 t:1):            Mounted from device: rootfs
dt (j:1 t:1):           Mounted on directory: /
dt (j:1 t:1):                Filesystem type: rootfs
dt (j:1 t:1):             Filesystem options: rw
dt (j:1 t:1):          Filesystem block size: 4096
dt (j:1 t:1):
dt (j:1 t:1): Total Statistics:
dt (j:1 t:1):        Output device/file name: /var/tmp/d0/d3/d2/d3/d4/d5/dt.data-00000010 (device type=regular)
dt (j:1 t:1):        Type of I/O's performed: sequential (forward, rseed=0x5ce5732e000792b0)
dt (j:1 t:1):       Job Information Reported: Job 1, Thread 1
dt (j:1 t:1):      Data pattern read/written: 0x66673326
dt (j:1 t:1):             Total records read: 1339
dt (j:1 t:1):               Total bytes read: 167772160 (163840.000 Kbytes, 160.000 Mbytes, 0.156 Gbytes)
dt (j:1 t:1):          Total records written: 1339
dt (j:1 t:1):            Total bytes written: 167772160 (163840.000 Kbytes, 160.000 Mbytes, 0.156 Gbytes)
dt (j:1 t:1):        Total records processed: 2678 with min=512, max=262144, incr=variable
dt (j:1 t:1):        Total bytes transferred: 335544320 (327680.000 Kbytes, 320.000 Mbytes)
dt (j:1 t:1):         Average transfer rates: 64683130 bytes/sec, 63167.119 Kbytes/sec
dt (j:1 t:1):        Number I/O's per second: 516.240
dt (j:1 t:1):         Number seconds per I/O: 0.0019 (1.94ms)
dt (j:1 t:1):         Total passes completed: 1/1
dt (j:1 t:1):          Total files processed: 320/320
dt (j:1 t:1):          Total errors detected: 0/1
dt (j:1 t:1):             Total elapsed time: 00m05.18s
dt (j:1 t:1):              Total system time: 00m00.21s
dt (j:1 t:1):                Total user time: 00m00.57s
dt (j:1 t:1):                  Starting time: Wed May 22 12:05:02 2019
dt (j:1 t:1):                    Ending time: Wed May 22 12:05:07 2019
dt (j:1 t:1):
[root@rtpqa-rhel001 linux-rhel7x64]#
```

And if a single thread doesn't generate sufficient load and stress, simple add threads= option:

```
$ dt of=/var/tmp/dtdir/dt.data bs=random limit=1m files=10 sdirs=3 depth=5 threads=3
dt (j:0 t:0): Warning: Top level directory /var/tmp/dtdir, will *not* be deleted!
```

Now, each thread will generate its' own set of directories and files.

The workload=dt_hammer will beat the tar out of a single file system well:

```
$ dt workloads hammer
Valid Workloads:

    dt_hammer: dt Hammer File System Workload (requires ~6.20g space)
        bs=random min_limit=b max_limit=5m incr_limit=vary files=250 maxdatap=75
iodir=vary iotype=vary onerr=abort enable=btags,deleteperpass prefix='%d@%h'
pattern=iot bufmodes=buffered,cachereads,cachewrites,unbuffered history=5 hdsize=128
enable=htiming alarm=3 noprogt=15 noprogtt=3m disable=pstats keepalivet=5m threads=10
```

```
$
```

Please Note: The *hammer* I/O behavior is now available thanks to Chris from NetApp getting permission to open source both *hammer* and *sio* tools, integrated into *dt*. Please use "**dt iobehavior=hammer**" to select this tool. Each tool has its' own help and options.

11.3.1 File Prefilling

Sometimes it's useful to prefill files with a pattern, other than the one being used for testing. This is usually done as part of troubleshooting.

The following options are available to control file prefilling:

- **fill_pattern=value** - The 32 bit hex fill pattern to use
- enable/disable flags:
    - **fill_always** - Always fill files. (Default: disabled)
    - **fill_once** - Fill the file once. (Default: disabled)

When the fill pattern is not specified, the first 4 bytes of the pattern is inverted and used.

Please know that if you wish to do 100% reads of a file, the file is prefilled once automatically:

```
$ dt of=/var/tmp bs=random readp=100 limit=100m files=10
dt (j:1 t:1): Warning: Files will be filled once to populate with data for reading.
```

11.3.2 File System UNMAPs

Some OS file systems have the ability to trim blocks, and in the case of SAN storage this turns into SCSI UNMAP requests.

For **Windows**, two file system trim options exist:

- **fstrim_freq=value** - The file system trim frequency (in files).
- **enable/disable=fstrim** - File system trim. (Default: disabled)

Several years have passed since adding this support, so perhaps other OS file systems now support trimming?

Additional file trim support will be added, esp. for Linux, but enabling this on the file system is best for now. For example, on Windows with NTFS when either deleting or truncating files, a file trim operation is performed.

Also know, the Linux *fio* tool, has an extensive set of options for controlling file trim operations.

Note: Custom file system trim operations can be executed via the per pass script (**pass_cmd=string**) option.

11.3.2.1 Windows NTFS and File Trim Operations

On Windows, NTFS can be configured to enable automatic file trimming when files are deleted or truncated.

The dt options to enable NTFS file trim are:

- **oflags=trunc** - Adds TRUNCATE_EXISTING, when the file is open'ed for output (writing).
- **enable=deleteperpass** - After each pass, the file(s) will be deleted.

With a simple set of options, you can generate huge amounts of SCSI UNMAPs:

For example: **files=10 sdirs=3 depth=3 limit=25m enable=deleteperpass threads=3 flags=direct**

The above will create 3 threads each working on their own set of sub-directories and files, 200 files total.

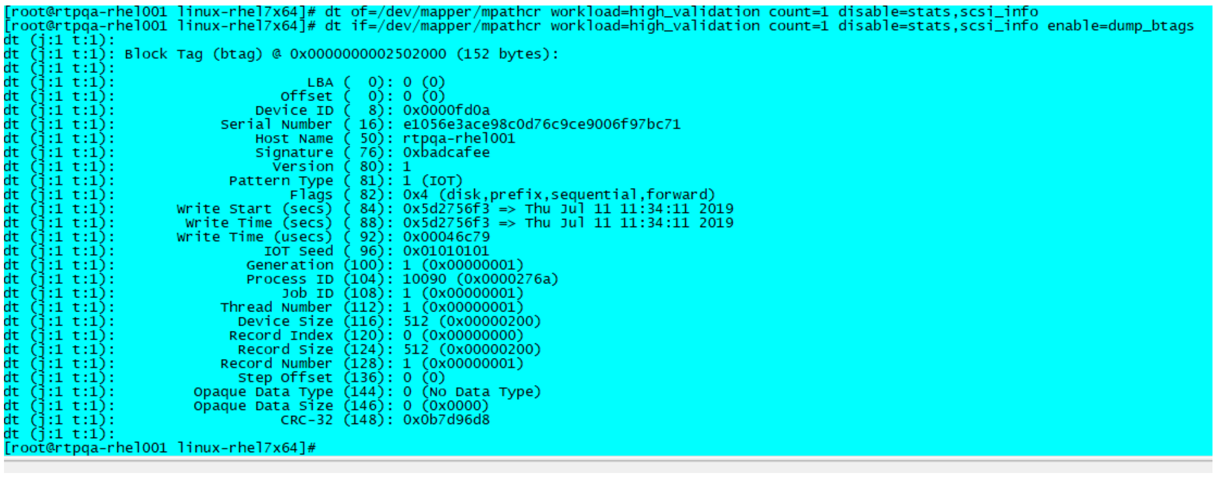
Extended runs usually use **runtime=value** or **passes=value**.

You can add **enable=fdebug** to see file operations performed.

12 Block Tags

A block tag is unique information added to every disk block. Please know, since this is unique per block, therefore it is not suitable for verifying compression and/or de-duplication. More to come on this in a separate document.

Here's an example of what the block tag looks like:



```
[root@rtpqa-rhel001 linux-rhel7x64]# dt of=/dev/mapper/mpathcr workload=high_validation count=1 disable=stats,scsi_info
[root@rtpqa-rhel001 linux-rhel7x64]# dt if=/dev/mapper/mpathcr workload=high_validation count=1 disable=stats,scsi_info enable=dump_btags
dt (j:1 t:1):
dt (j:1 t:1): Block Tag (btag) @ 0x0000000002502000 (152 bytes):
dt (j:1 t:1):
dt (j:1 t:1):                        LBA (  0): 0 (0)
dt (j:1 t:1):                     Offset (  0): 0 (0)
dt (j:1 t:1):                  Device ID (  8): 0x0000fd0a
dt (j:1 t:1):              Serial Number ( 16): e1056e3ace98c0d76c9ce9006f97bc71
dt (j:1 t:1):                  Host Name ( 50): rtpqa-rhel001
dt (j:1 t:1):                  Signature ( 76): 0xbadcafee
dt (j:1 t:1):                    Version ( 80): 1
dt (j:1 t:1):               Pattern Type ( 81): 1 (IOT)
dt (j:1 t:1):                      Flags ( 82): 0x4 (disk,prefix,sequential,forward)
dt (j:1 t:1):         Write Start (secs) ( 84): 0x5d2756f3 => Thu Jul 11 11:34:11 2019
dt (j:1 t:1):          Write Time (secs) ( 88): 0x5d2756f3 => Thu Jul 11 11:34:11 2019
dt (j:1 t:1):         Write Time (usecs) ( 92): 0x00046c79
dt (j:1 t:1):                   IOT Seed ( 96): 0x01010101
dt (j:1 t:1):                 Generation (100): 1 (0x00000001)
dt (j:1 t:1):                 Process ID (104): 10090 (0x0000276a)
dt (j:1 t:1):                     Job ID (108): 1 (0x00000001)
dt (j:1 t:1):              Thread Number (112): 1 (0x00000001)
dt (j:1 t:1):                Device Size (116): 512 (0x00000200)
dt (j:1 t:1):               Record Index (120): 0 (0x00000000)
dt (j:1 t:1):                Record Size (124): 512 (0x00000200)
dt (j:1 t:1):              Record Number (128): 1 (0x00000001)
dt (j:1 t:1):                Step Offset (136): 0 (0)
dt (j:1 t:1):           Opaque Data Type (144): 0 (No Data Type)
dt (j:1 t:1):           Opaque Data Size (146): 0 (0x0000)
dt (j:1 t:1):                     CRC-32 (148): 0x0b7d96d8
dt (j:1 t:1):
[root@rtpqa-rhel001 linux-rhel7x64]#
```

Note: A SCSI device *must* be available to report the serial number shown above. For file systems, dt will try to find the disk from the mount table, but SCSI CDB's do **not** work for LVM devices.

13 Debugging

The tool has various debug flags than can be enabled:

- debug - Debug output. (Default: disabled)
- Debug - Verbose debug output. (Default: disabled)
- btag_debug - Block tag (btag) debug. (Default: disabled)
- edebug - End of file debug. (Default: disabled)
- fdebug - File operations debug. (Default: disabled)
- jdebug - Job control debug. (Default: disabled)
- ldebug - File locking debug. (Default: disabled)
- mdebug - Memory related debug. (Default: disabled)
- mntdebug - Mount device lookup debug. (Default: disabled)
- pdebug - Process related debug. (Default: disabled)
- rdebug - Random debug output. (Default: disabled)
- tdebug - Thread debug output. (Default: disabled)
- timerdebug - Timer debug output. (Default: disabled)
- sdebug - SCSI debug output. (Default: disabled)

Example: **enable=debug,Debug** (very verbose)

Many of these flags are for developer debug, so you won't normally need them.

13.1 Data Corruptions

Please refer to dt data corruption documents.

14 Triggers

What is the purpose of a trigger?

For those who are new to this tool and this concept, the idea of a trigger comes from "triggering analyzers". The idea here is to send a SCSI operation code (opcode), that the analyzer has been configured to detect and stop its' trace, thereby capturing all the SCSI requests leading to the failure. Similar triggers can be implemented in the array I/O path with custom firmware.

What are the types of failures we may need triggers for?

Within dt, I have created three different categories, with an option to control each, as to when a trigger type should be executed:

- **errors** - when any OS API returns a failure (open, close, read, write, flush, etc)
- **miscompare** - whenever data read does **not** compare against expected data (aka data corruption)
- **noprogs** - when the I/O monitoring detects OS API's are **not** completing quickly or are hung

Which triggers are best for your testing varies, of course, but please know if any action is to be taken, this is driven by the trigger options specified by you, the test executioner! In other words, triggers are not automatically executed regardless of the error condition.

Beyond the internal triggers currently implemented, a trigger command option was added to allow external commands/scripts to perform a set of user required steps. These steps may or may not include triggering an analyzer but hosts often are used to perform a set of host and/or array commands to help troubleshoot issues. This can stem from collecting key data, crashing the host, or accessing the storage array to execute vendor specific CLI commands.

Please Know: Additional triggers can always be implemented, and any SCSI CDB can be sent via the **trigger=cdb:CDB_BYTES** option**.**

14.1 Trigger Recommendations

When failures occur, usually engineers wish to preserve the state of the storage so debugging occurs as close to the original failure as possible. This will generally provide the best chance that "*evidence*" has been preserved, which will hopefully help root cause issues more quickly. To that end, the general recommendation is to execute triggers that can be acted upon by storage rapidly, followed by trigger methods that take longer, usually due to their access methods. dt supports up to four (4) triggers, each with their own way to stop I/O and/or trigger analyzers.

Recommended trigger order is:

1. In-band SCSI command, often a vendor unique CDB. (fastest method)
2. Execute an external trigger command/script. (time varies based on access method)

The in-band SCSI method requires a valid path to be available. This will normally be the direct disk device, or a device path gleaned from the mount table (Unix) or drive letter (Windows). There is also a SCSI device option (**sdsf=PATH**) where users can specify their own SCSI path, which does not need to be the device under test (consider an iSCSI device to allow VU commands to be sent). Note: dt only accepts on SCSI device today and does **not** support disks part of logical volumes (LVM), where multiple SCSI devices are configured. Restriction today could be enhanced, after all it's only software! 🙂

Also know, the vendor unique SCSI opcode, can be implemented in the SCSI Target to distribute requests across the cluster to other controllers, and/or crash the kernel to collect crash dumps!

The external command/script method requires an additional process to be created, which takes time based on load, and then the access method used in the script requires execution time. For example, using *curl* to send a REST API, or using *ssh* to send Secure Shell commands can take several seconds. In this short time, the storage array can and does execute many more operations, so we risk losing the valuable evidence required. Please know internally dt uses the POSIX API popen() to start external commands, so on Unix with will start your default shell, and on Windows it starts the DOS command shell. Nonetheless, the external scripts can be any scripting language, so there is a lot of flexibility (Perl, Python, executable tool, etc).

FYI: My personal preference, and what I recommend, is to send an in-band SCSI CDB (if applicable), followed by a trigger script. Usually, one of the two methods will be effective.

14.2 Trigger Options

A number of trigger options are supported, specifically for triggering analyzers or otherwise taking action on errors:

- The triggers to execute on errors.
  - **trigger=**{br, bdr, lr, seek, **cdb:bytes**, cmd:str, and/or triage}
- The trigger control (when to execute):
  - **trigger_action=**value The trigger action (for noprogs).
  - **trigger_on=**{all, errors, miscompare, or noprogs} (Default: all)

All triggers are executed via in-band SCSI CDB's or TMF's, except for **trigger=cmd:str,** where 'str' can be either a command or an external script. On Unix systems, the default SHELL is executed by the POSIX popen() API, and on Windows the DOS command shell is invoked. Since triggering has become important as of late, see additional sections below for more details.

Please know, multiple triggers can be specified, up to 4 triggers!

Why use multiples? Well, in-band SCSI commands are usually preferred so the SCSI target can act immediately, but this requires a valid path from the host. Therefore, an external script is usually also preferred as an alternative. The external script usually uses Secure Shell (ssh) to access the controller/group to issue CLI commands.

The **cdb:bytes** will allow any arbitrary SCSI CDB to be sent. The CDB bytes can be either space or comma separated, with commas often being easier than spaces in automation due to quoting.

Examples:

- **trigger=cdb:'xx xx xx …'**
- **trigger_on=miscompare - only send on corruptions.**

14.3 Triage Trigger

The **trigger=triage** feature will do the following commands, useful to determine if SCSI device paths exist and if they are accessible:

1. Sends an Inquiry command. (equivalent to network ping, to see if the disk is alive)
2. Sends a Test Unit Ready command.
3. Some arrays support a vendor unique health check operation.

These SCSI commands are sent via the SCSI pass-through API, assuming dt found a valid SCSI device before starting the test.

The SCSI status and Request Sense data returned on failures, is often useful for troubleshooting issues. Real SCSI errors, not generic EIO error.

15 dt Scripts

You can add dt commands to a dt script file to perform a set of advanced workloads:

- scripts have a default extension of .dt
- environment variables can be expanded in the form of ${VAR}
- environment variables can be set using this syntax:
    - $VAR="text" - Always set variable.
    - $VAR=${VAR:-"text"} - Only set variable, if **not** already set.
    - $VAR=${VAR:?message} - Report a message and exit w/failure if **not** set.
- add "**enable=scriptverify**" to trace script execution, otherwise it's silent.
- example scripts are located here: **/auto/share/IOGenerationTools/Dt/Scripts**
- use the "**script=FILE[.dt]**" command to execute the script file.

Example copying disks and files:

```
# export DISK_SRC=/dev/mapper/mpathq
# export DISK_DST=/dev/mapper/mpathw
# export FILE_SRC=/mnt/mpathv/src_test_file.data
# export FILE_DST=/mnt/mpathab/dst_test_file.data
# dt script=docopy
```

16 Use Cases

This section describes a few use cases to whet your appetite!

Since dt has a flexible set of options, generating many workload types is possible.

In addition to these use cases, please see "**dt workloads**" for others built in.

Note: Scripts referenced below are in this repository directory: **dt/Scripts/**

16.1 Testing Large Terabyte LUNs

Testing large amounts of data can take a very long time, so utilizing multiple threads and using the **step=** option, allow the full capacity to be verified without doing I/O to all the associated blocks. With file systems, these holes are normally referred to as a sparse file and will read as zero since nothing has been written to that area.

See "**workload=terabyte_lun**" or "**workload=terabyte_file**"

16.2 Testing Deduplication/Compression

To test de-duplication, a test tool must write duplicate 4kb blocks of data, so de-duplication will happen. For compression, there must be duplicate sequences of bytes.

Many tools generate unique data in each block, and rightfully so, since this makes it easier to detect data corruption, and to that end, *dt* has several options to generate unique data in each block. But for testing de-duplication and compression, we need some set of duplicated bytes and blocks. Described below, is one method to accomplish this using *dt*.

There are several predefined data pattern files included in the *dt* 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 pattern file **(pf=file)** option, or you can create your own.

See "**workload=dt_dedup**"

As of this writing, the data files reside in this repository directory: **dt/data/**

16.3 File/LUN Copy and Verify

dt has options to allow file or LUN copy and verify, or verifying data written to synchronous replication (mirror) LUN.

Please see these scripts for examples: (view script defaults prior to executing)

- docopy.dt - Copy disk to disk and file to file with data verification.
- dodircopy.dt - Populate files in source tree then copy them to destination directory.
- dofilecopy.dt - Populate single file and copy the same to a destination file.
- doluncopy.dt - Populate source and destination LUNs, then copy source to destination.
- domirror.dt - Verify destination LUN and mirror LUN operations.

Please Note: For more efficient LUN copying, use *spt* and extended copy operations.

16.4 Data Generation Tool

dt can be used as a data generation tool to feed data to other tools that read from pipes. dt accepts '-', like *dd* utility, for read/writing via pipes (shell '|'). So for example, dt can generate its' unique data patterns for other tools such as *dd or spt* or creating files for snapshots, cloning, replication, etc that can be verified later.

Please review Scripts/{s3t.sh,test-s3t.sh) for examples of using dt to test cloud storage.

16.5 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.

See script **butterfly*.ksh** for examples.

16.6 Improved File System Testing

dt has file system options to create multiple directories and multiple files per directory all with one invocation. Generally anything created is automatically deleted when finishing tests, but the **dispose=keep** or **keeponerror** options can avoid deletions. If your goal is testing NFS/CIFS or file systems, these options are useful.

16.7 Write Files Then Read Files Continually

See script **doworm.dt**

16.8 File System Full Conditions

dt supports testing file system full conditions, as well as quota exceeded, which is handled like a file system full. But, there is logic which will report an error, if these conditions occur on the **first file** and on the **first write**. The reason this error is reported is there will be no testing since nothing was written, thus nothing can be read.

That said, many people do not wish to test file system full conditions, and this is just a side effect of the dt parameters provided on a file system too small for all the files and/or data being written. One way to avoid file system full conditions, is to determine the amount of free space available, then calculate and specify a value less than the free space using these options:

- **maxdata=value** - The maximum data limit (all files).
- **maxdatap=value** - The maximum data percentage (range: 0-100).

Writing will stop when this maximum data limit is reached, thus avoiding file system full.

16.9 Over Provisioned LUNs

Like the max data options, there are options for LUNs to avoid using the full capacity:

- **capacity=value** - Set the device capacity in bytes. (use a smaller capacity)
- **capacityp=value** - Set capacity by percentage (range: 0-100).

These options are useful for storage arrays with thinly provisioned LUNs that lack sufficient back end storage.

Alternatively, sequential I/O with the **step=** option or specifying ranges of blocks can also limit storage consumed.

16.10 Dynamic I/O Throttling

The ability to automatically throttle I/O based on array or host CPU utilization. Requires external script. (**TBA**)

16.11 Avoiding Misaligned I/O

Many storage arrays have an underlying back-end block size that differs from the host block size advertised. For example, the storage discovered by hosts may report a block size of 512 bytes per block, while the underlying array block size may be 4k, 8k, or larger. dt normally discovers the host LUN block size automatically but does support a device size (**dsize=value**) option to override the disk block size detected. When specified, offsets and transfer sizes will be modulo the specified device block size.

See "**workload=disk_aligned_io**" and "**workload=disk_unaligned_io**"

16.12 I/O Percentages

Several options exist to control the percentage of reads and writes as well as sequential versus random I/O:

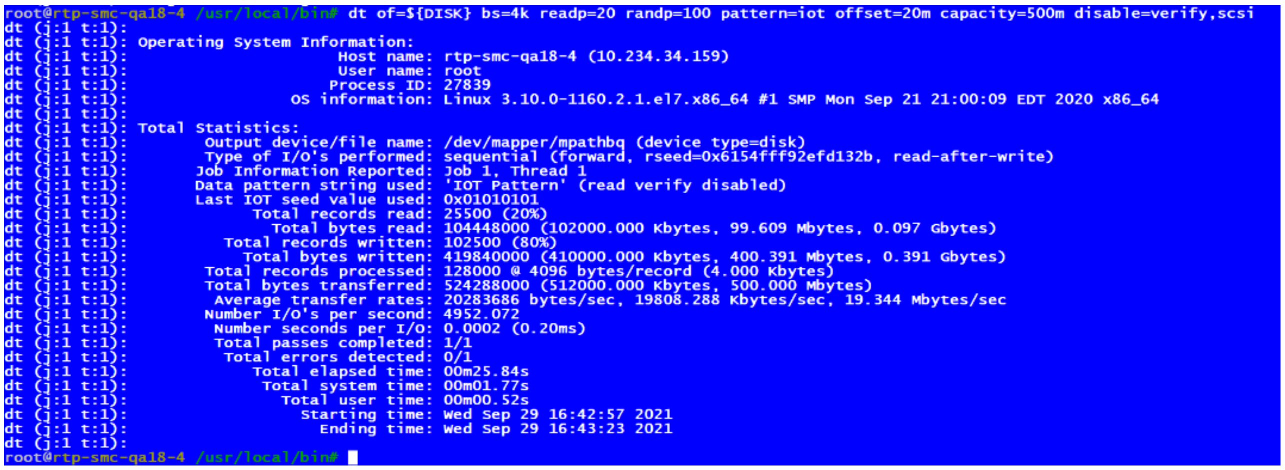
randp=value - Percentage of accesses that are random. Range [0,100].
rrandp=value - Percentage of read accesses that are random. Range [0,100].
wrandp=value - Percentage of write accesses that are random. Range [0,100].
readp=value - Percentage of accesses that are reads. Range [0,100].

For example, to generate 4k I/O's with 100% random and 20% reads/80% writes, use this command:

```
# dt of=${DISK} bs=4k readp=20 randp=100 pattern=iot offset=20m capacity=500m disable=verify
```

Please Note: While custom workloads like this create desired mixes, data verification is disabled since dt does not track read/writes internally.

The total statistics will show the read/write percentages achieved:



```
root@rtp-smc-qa18-4 /usr/local/bin# dt of=${DISK} bs=4k readp=20 randp=100 pattern=iot offset=20m capacity=500m disable=verify,scsi
dt (j:1 t:1):
dt (j:1 t:1): Operating System Information:
dt (j:1 t:1):                      Host name: rtp-smc-qa18-4 (10.234.34.159)
dt (j:1 t:1):                      User name: root
dt (j:1 t:1):                     Process ID: 27839
dt (j:1 t:1):                 OS information: Linux 3.10.0-1160.2.1.el7.x86_64 #1 SMP Mon Sep 21 21:00:09 EDT 2020 x86_64
dt (j:1 t:1):
dt (j:1 t:1): Total Statistics:
dt (j:1 t:1):        Output device/file name: /dev/mapper/mpathbq (device type=disk)
dt (j:1 t:1):        Type of I/O's performed: sequential (forward, rseed=0x6154fff92efd132b, read-after-write)
dt (j:1 t:1):       Job Information Reported: Job 1, Thread 1
dt (j:1 t:1):       Data pattern string used: 'IOT Pattern' (read verify disabled)
dt (j:1 t:1):       Last IOT seed value used: 0x01010101
dt (j:1 t:1):             Total records read: 25500 (20%)
dt (j:1 t:1):               Total bytes read: 104448000 (102000.000 Kbytes, 99.609 Mbytes, 0.097 Gbytes)
dt (j:1 t:1):          Total records written: 102500 (80%)
dt (j:1 t:1):            Total bytes written: 419840000 (410000.000 Kbytes, 400.391 Mbytes, 0.391 Gbytes)
dt (j:1 t:1):        Total records processed: 128000 @ 4096 bytes/record (4.000 Kbytes)
dt (j:1 t:1):        Total bytes transferred: 524288000 (512000.000 Kbytes, 500.000 Mbytes)
dt (j:1 t:1):         Average transfer rates: 20283686 bytes/sec, 19808.288 Kbytes/sec, 19.344 Mbytes/sec
dt (j:1 t:1):        Number I/O's per second: 4952.072
dt (j:1 t:1):         Number seconds per I/O: 0.0002 (0.20ms)
dt (j:1 t:1):         Total passes completed: 1/1
dt (j:1 t:1):          Total errors detected: 0/1
dt (j:1 t:1):             Total elapsed time: 00m25.84s
dt (j:1 t:1):              Total system time: 00m01.77s
dt (j:1 t:1):                Total user time: 00m00.52s
dt (j:1 t:1):                  Starting time: Wed Sep 29 16:42:57 2021
dt (j:1 t:1):                    Ending time: Wed Sep 29 16:43:23 2021
dt (j:1 t:1):
root@rtp-smc-qa18-4 /usr/local/bin#
```

17 Futures?

From my experience, there is no perfect tool, and frankly there's always more things to consider:

- automatically gather protocol traces, on host and/or array
- monitor host and array system resources, record this in logs
- automatically tune I/O based on system or array resources
- interact with tools dynamically rather than statically
- lots of other ideas, perhaps you have your own?

FYI: Since retiring in January 2022, future development is limited due to lack of hardware resources. Nonetheless, I am available to answer questions, fix issues, and/or implement minor enhancements.