BTEST - Block Level Exercise Tool v1.4

Table of Contents

1 Main features and abilities

1.1 General description

Btest is a Linux command line block level IO exerciser, loader, tester and benchmarking tool for block oriented devices and files. It is a low level tool, oriented for direct command line usage and scripts.

1.2 IO modes

Btest is Linux specific and as such is able to operate in all major Linux IO modes: synchronous IO, asynchronous IO (write behind), Direct IO, (native) aio, and sg (SCSI Generic).

1.3 IO Patterns

Btest is able to generate a large range of IO patterns, starting from plain sequential and random IO, and ending with complex mixes of random, sequential, block sizes, alignments, block ranges, etc.

1.4 High performance, low overhead

Btest is a multi-threaded program and is able to generate very high loads. In most cases Btest overhead is negligible unless data verification modes are used, and even then the impact on the IO load is low.

1.5 Data patterns

Btest can generate several types of data pattens meant to exercise content aware systems, and to test data validity. In particular it is able to produce dedup-aware pattens and is able to control the expected data dedup factor.

1.6 Verification

Btest is able to check the data integrity by checking the data patterns after each read. Special features for tracking dynamic data patterns (such as the dedup aware stamps) and for saving/loading the verification meta data to/from meta data files are provided. The load/save feature enables the user to run a sequence of different tests on the same data sets without losing the verification state. A special running mode is provided to verify the test data using a meta-data file(s). Btest can even run this mode concurrently with other running tests.

1.7 Data set initialization

Btest can initialize the data section that is to be used during the test either by writing zero blocks on it, and/or by using ATA trim commands (that are supported by most modern SSD devices).

1.8 Reports

Btest produces detailed summary reports and is able to produce various real-time reports. These reports can be used to generate detailed graphs (in real time or after the test) using standard external tools such as gnuplot or excel.

1.9 Debugging

Btest has several debug levels and a special lock free verification meta-data binary trace.

2 Limitations and missing functionalities

2.1 Low level tool

Btest is not a benchmarking application that can produce a single number that can be used to compare the performance of different devices and/or systems. It is a low level tool that requires good low level understanding. Note that btest may be used to benchmark and compare devices, but it is the users responsibility to define and commit the benchmark.

2.2 Linux only

Btest is not portable and runs only on Linux 2.6 systems. However, it should be relatively easy to port it to other OSes.

2.3 Command line only version

Btest has no graphical user interface. It is possible to build a GUI wrapper that will use the command line btest as its low level engine.

2.4 Synthetic tool

Btest can produce various types of IO patterns and data patterns that may or may not resemble real application IO workloads. Btest is not modeled after any real application and cannot be used to predict the IO performance of any given application, unless the application IO load is known and is simple enough to simulate using btest patterns.

2.5 Random testing tool

Btest generates random patterns and in the general case it can not exactly reproduce an already generated pattern. Note that in some cases (single thread, known random seed) it is possible to reproduce a pattern, but in general btest can not be used as a “scenario” playing tool (i.e. a tool that can deterministically repeat a predefined pattern).

2.6 Limited data patterns

Btest supports only limited set of data pattern. This may limit its usage as low level media data integrity checking tool.

3 Btest installation and invocation

3.1 Deployment

Btest is implemented as a single standalone binary. The only dynamic libraries it uses are common libraries such as libc, libpthreads and libaio. In most cases you can just copy the executable to an arbitrary location and run it from there. Btest by itself doesn't require privileged permissions to run, but the target test devices may require such permission to be able to read/write from/to them.

3.2 Invocation

Btest has many options and parameters. The exact and complete list of these can be found in the help screen that can be displayed using “btest -h” or “btest –help”.

The following sections provide a more “use case” oriented usage description.

4 Basic usage

The simplest method to use btest is as follows:

```
btest <random IO ratio> <read ratio> <device1> <device2> …
```

The device parameters are anything that looks like a file, i.e. either a real file, raw device, LVM/device mapper device, etc.

By default btest uses fixed block size of 4k, simple synchronous IO, and a single thread for each given device.

By default, the test is time limited and lasts for 60 seconds.

The ratio parameters are numbers between 0 and 100 such that

```
btest 40 70 /dev/sda
```

means 40% random IO and 60% sequential IO with 70% reads and 30% writes.

In addition a single character notation can be used as follows

Pattern name	Number notation	Character notation
Sequential	0 (random ratio)	**S**
Random	100	**R**
Write	0 (read ratio)	**W**
Read	100	**R**

Examples:

Pure random write, fixed 4k block size, synchronous IO, single threaded 60 second test:

```
btest R W /dev/sda
```

Same, but with mix of 50% percent random writes and 50% percent random reads:

```
btest R 50 /dev/sda
```

4.1 Changing the test exit condition

Btest test duration is limited to 60 seconds by default. Use the *-t* option to set any other time limit. Use *-t 0* to set unlimited test duration.

Btest can be limited by IO operations. Use the *-n* option to set the requested total number of IO operations. Note that the specified number is the total number of operations of any type, done by all threads, on any device.

In addition, btest can be set to exit when end of file (device) is reached using the -e flag. This flag makes sense only for sequential IO patterns.

Note that the default 60 seconds time limit is not applied when any limit condition is explicitly specified. On the other hand, multiple limit conditions can be specified.

Examples:

Pure random write, fixed 4k block size, synchronous IO, single threaded 120 seconds test:

```
btest -t 120 R W /dev/sda
```

Same but limited to 10000 operations (no time limit):

```
btest -n 10000 R W /dev/sda
```

Same but limited to 10000 or 120 seconds, whichever comes first:

```
btest -n 10000 -t 120 R W /dev/sda
```

Sequentially write the entire device exactly once (no time limit):

```
btest -e S W /dev/sda
```

Sequentially write the entire device exactly once (no time limit):

```
btest -e S W /dev/sda
```

Same, but with additional 600 seconds time limit and 10000 operations limit (which ever happens first):

```
btest -e -n 10000 -t 600 S W /dev/sda
```

Unlimited 4k random write:

```
btest -t 0 R W /dev/sda
```

4.2 Setting the block size and alignment

By default btest uses 4k blocks aligned on 4k. You can change the block size using the *-b <blocksize>* option and the alignment can be changed using the *-a <alignsize>* option. If the *-a* option is not used, the IO is aligned to the same size as the block size. Both options accept size parameter that can be specified in bytes and supports the following size postfixes

postfix	units
(none)	Bytes
b	Blocks (512 bytes)

K or k	**Kilobytes (1024 bytes)**
M or m	**Megabytes (2^{20})**
G or g	**Gigabytes (2^{30})**
T or t	**Terabytes (2^{40})**
P or p	**Petabytes (2^{50})**

Examples:

100% random write, fixed 32k block size, aligned to 32k, synchronous IO, single thread per device, 60 second test:

```
btest -b 32k R W /dev/sda /dev/sdb
```

Same but aligned to 512 bytes

```
btest -b 32k -a 512 R W /dev/sda /dev/sdb
```

4.3 Setting the test data range

By default btest is generating IOs that span on the entire device/file size. You can use the *-l <length>* option to set the data range length, and the *-o <startoffset>* option to change the start offset. Both options support the same size postfixes as the block size options.

Examples:

100% random write, fixed 4k block size, limit IO range to first gigabyte of each device, synchronous IO, single thread per device, 60 second test:

```
btest -l 1G R W /dev/sda /dev/sdb
```

Same but data range is from 100 megabyte to 300 megabyte:

```
btest -o 100m -l 200m R W /dev/sda /dev/sdb
```

5 IO modes

By default btest uses synchronous mode (O_SYNC). In this mode each write is immediately flushed to the backend device, but the generic block cache layer is used such that reads may be cached (i.e. replied from the cache and not from the device).

Use the *-W* to request write-behind mode (classic UNIX “asynchronous” mode). In this mode all writes are written to the generic block cache and flushed according to the OS policy. Note that at the end of the test, a flush is forced and the test summary statistics are computed twice, once before the flush, and once after it.

To avoid using the generic OS block cache, direct IO may be used by using the *-D* flag. In this mode, the block cache is entirely skipped, such that all writes are flushed immediately, and also the reads are replied by the device and not from the cache.

Devices that are formatted as /dev/sgX are automatically opened and treated as SCSI generic devices and uses the SCSI generic read and write calls. In this IO mode the generic block cache is also skipped as it is in the direct IO mode.

Note that all the above IO modes use the same threading model – the requested number of threads (using the *-T <nthreads>* option) is created for each device and each thread is issuing a single IO at a time.

The native Linux aio mode is activated using the *-w <io_window>* option. In this mode the threading model is completely different: instead of creating the requested n_threads threads per device, an absolute number of n_threads are created for the all devices, but each of these threads maintains a constant IO window of the requested io_window **per device.** In other words, each thread issues an io_window number of IO requests per device and for each completed request a new request is issued resulting in each device having io_window IOs pending in any point in time.

IO modes table

IO mode	Controlling Option	Threading model	Number of concurrent IOs per device	Caching	Comments
Synchronous	<none>	N threads per device	N_threads	reads	**The default (O_SYNC)**
Write-behind	-W	N threads per device	N_threads	Reads and writes	
Direct	-D	N threads per device	N_threads	None	**O_DIRECT**
Scsi generic (SG)	<none>	N threads per device	N_threads	None	**For /dev/sgX devices only**
Native AIO	-w <window>	N threads Total	N_threads X window	None	

Notes:

1. Not all devices support all IO modes. For example some devices do not support Direct IO, and not all block sizes are supported for this mode. IO modes cannot be mixed.
2. All devices must use the same IO mode.

6 Controlling the number of threads

The *-T <nthreads>* option controls the number of thread btest uses. The actual number of threads used is determined by the IO mode. In non aio (asynchronous IO) modes the number of threads is per device, where in aio mode it is the absolute number of threads, regardless of the number of devices.

Note that the actual number of pending IO requests is determined by the IO mode: in non aio (asynchronous IO) modes, each btest thread is issuing a single IO request, such that the pending IOs number (per device) is the threads number. In aio mode each btest thread

is maintaining the requested window size requests (see “IO modes“ above) such that the pending IOs per device is the window size multiplied by the thread number.

Examples:

100% random write, fixed 4k block size, synchronous IO, 2 threads per device (total 4 threads and 4 requests), 60 second test:

```
btest -T 2 R W /dev/sda /dev/sdb
```

100% random write, fixed 4k block size, asynchronous IO, 2 threads (total 2 threads), 2 request per thread, two devices (total 8 requests), 60 second test:

```
btest -T 2 -w 2 R W /dev/sda /dev/sdb
```

7 Sequential patterns and multithreading/mutiple asynchronous requests

It is possible to do sequential IO using several threads or requests. In this case the file/device (or the data range within it) is treated as a long sequence of blocks, and on its turn, each thread/request is doing IO to the next available (unused) block. Note that the actual order among the threads/requests cannot be guaranteed due the system scheduler and/or the parallel execution on several physical CPUs (or cores). On the other hand, assuming that the system is fair, the threads/requests should follow each other within a giving “window”, meaning all threads will perform IOs on very near areas. On most systems this should be good enough because the host and/or the storage reorder IOs to minimize disk arm movements (“elevator algorithm”). Furthermore, modern Linux storage sub-systems may even join several IOs to one big IO, so two threads/requests generating sequential IO using 4k blocks on the same file can be regarded almost as one thread doing sequential IO using 8k blocks.

Examples:

100% sequential write, fixed 4k block size, synchronous IO, 2 threads per device (total 4 threads), 60 second test:

```
btest -T 2 S W /dev/sda /dev/sdb
```

8 Data patterns generation

Btest supports two types of data patterns – static data patterns, i.e. patterns that do not change from one block to another, and dynamic data patterns that are generated in real time per block (write). Some patterns can be mixed.

Btest supports the following static data patterns by default:

1. Compression aware patterns (option -Z)
2. Zero block: (*option -P 0, -Z -1*)

Btest supports the following dynamic data patterns

1. Dedup aware patterns: symbol stamp per (stamp) block (*options -P, -p*), also used for data verification

2. Offset based pattern: data is filled with offset specific pattern (*option -O*)

8.1 Default pattern

By default, btest uses a random filled buffer with changing dedup stamp such that its data can't be easily deduped and/or compressed. Note that systems that do compression/dedup not on block boundaries but rather on a stream, can still compress/dedup such data.

Option wise btest is configured as follows:

- -p 0 => zero dedup pattern, each data block has its own changing data stamp
- -P "blocksize" => the patterns are filled on IO blocks basis
- -Z 0 => the base data buffer is randomized such that it can not be compressed

8.2 Static patterns

8.2.1 Compression aware patterns

By default, the base (non changing) buffer pattern is filled with random 32 bit words to avoid compression. The pattern can be set such that it can be compressed using the "*-Z <compression rate>*" or "--compression <rate>" option where the compression rate is the best expected compression factor you should get from **a block level compression**. Stream level compression (gzip for example) utilities can compress the data much better because they may inspect much larger "window" of data. The actual compression rate depends on the compression algorithm and method used. Compression rate -1 will disable compression pattern at all. Compression aware pattern can be used together with dedup stamps (see the seconds below). They may contradict offset patterns (see below).

Examples:

100% random write, fixed 4k block size, no-dedup pattern, uncompressable data, synchronous IO, single thread, 60 second test:

```
btest R W /dev/sda
```

100% random write, fixed 4k block size, no-dedup pattern, data is compressible by factor 2 (50% reduction) , synchronous IO, single thread, 60 second test:

```
btest -Z 2 R W /dev/sda
```

100% random write, fixed 4k block size, no-dedup pattern, disable compression aware pattern, synchronous IO, single thread, 60 second test:

```
btest -Z -1 R W /dev/sda
```

8.2.2 Zero block data

Btest can be forced to use zero block data by specifying "-P 0" and "-Z -1" (i.e by disabling both dynamic and static patterns). This makes the stamp block to be zero length and disables all data patterns. This can be used to compare data patterns effects, test hole punching systems (i.e. systems that treat zero block as "hole") or to simulate other older tools. Note that in the general case this pattern is highly discouraged, as many

modern systems treat zero block (i.e. a block filled with binary zeros) as special case, and in most case will avoid writing it.

Example:

100% random write, fixed 4k block size, zero data block, synchronous IO, single thread, 60 second test:

```
btest -P 0 R W /dev/sda
```

8.3 Dynamic patterns

8.3.1 Dedup aware patterns

8.3.1.1 Stamp block

Btest is able to change the data pattern for each write using data stamps. Data stamps are 8 bytes of “dedup stamp” and another 8 bytes for device ID. Data patterns are formed in stamp block sizes chunks. By default stamp block size is the smallest size out of the following:

- the IO block size or see “Setting the block size and alignment“ above),
- The smallest IO block size used by any workload,
- 4k (4096 bytes)

The stamp block size can be explicitly set using the *-P* options (or *–stampblock*). Note that the stamp block cannot be bigger than the IO block size, and that the stamp block alignment must match the IO block alignment and vice versa.

There are several possible use cases for setting the stamp block to be different from the IO block size:

- Avoid thin-provisioning and/or dedup optimizations: if for example the tested system is able to ignore zero 4k blocks and you use 1MB IO block size, then by default the stamp will appear only on the first 4k sub block (see the next section) and the other 255 block will be ignored. To overcome such optimization just set the stamp block size to 4k (of below) such that each 4k sub block will be stamped on its own. This is also the default settings.
- When verification is used (see “Verification“ below), then the stamp block is the unit used to verify the data. Therefore, the stamp block unit must be coordinated among the different workloads.

Examples:

100% random write, fixed 4k block size, stamp block 512 bytes, no-dedup pattern, synchronous IO, single thread, 60 second test:

```
btest -P 512 R W /dev/sda
```

Same but without any data pattern:

```
btest -P 0 R W /dev/sda
```

8.3.1.2 Symbol stamping and dedup control

Btest selects a symbol (a 64 bit number) per stamp block. By default, this number is picked at random. This behavior is used to overcome de-duplicating mechanisms.

You can control the expected dedup factor by by using the *-p* (or *–dedup*) option where the parameter of this option is the expected dedup factor. Btest has two methods for "dedup" fill. The default one is random fill where btest selects the symbol from a range of space_size*/dedup_factor* symbols. In this mode, the expected dedup factor is reached after the space is completely filled, meaning (almost) each block is written at least once.

The other fill method is "progressive fill" and it is set by using the *-g (or –progressive) option.* In this mode, the same stamp is written "dedup_factor" times before it is being replaced. This maintains the requested dedup rate right from the start, even if most blocks are never written. This method is very useful for partial fills and for handling very large objects, but on the other hand, it is less desired for verification use cases.

For example, if the size of /dev/sda is 1GB, and the following btest parameters are used

```
btest -b 4k -n 300000000 -p 2 R W /dev/sda
```

then after at least 256M blocks are written, the expected dedup factor is 2 or in other words on the average each block content appears twice within the dataset (see also the "Symbol stamping implementation" below).

Progressive fills are very similar but do not require a complete fill. For example, the following run will results the same dedup factor (even after 60 seconds):

```
btest -b 4k -t 60 -p 2 -g R W /dev/sda
```

Specifying 0 as the dedup parameter forces btest to use the full 64 bit symbol range leading to a very low probability of block data repetitions. This is also the default behavior. Specifying "*-1*" as the dedup value will disable the dedup stamps at all.

Note that "*-p -1*" is different from "*-P 0*" (setting stamp block to zero length): the former disables only dedup stamps while other (offset stamps or extensions' patterns) still apply. The "*-P 0*" disables any dynamic data patterns including the dedup stamps.

The "*-p -2*" option will make btest to use a single stamp for all blocks. In fact this is another mode of static (non changing) pattern.

Examples:

100% random write, fixed 4k block size, stamp block 512 bytes, expected dedup factor 7 pattern, synchronous IO, single thread, 60 second test:

```
btest -P 512 -p 7 R W /dev/sda
```

Same but with dedup stamps disabled any data pattern:

```
btest -P 512 -p -1 R W /dev/sda
```

8.3.2 Offset specific pattern

The *-O* (or *–offset_stamp*) may be used in addition to the symbol stamping patten (see the section above) or instead of it. This patten fills the stamp block (short of the symbol

stamp , if any) by writing the blocks offset in ASCII using 16 bytes blocks. This can be used to test data integrity and used for data verification (see the “Verification” section below). For dedup control see the above section.

Examples:

100% random write, fixed 4k block size, stamp block 1024 bytes, no dedup pattern (but with stamp), synchronous IO, single thread, 60 second test:

```
btest -P 1024 -p 0 -O R W /dev/sda
```

Same but without dedup stamps and 4k block stamps:

```
btest -p -1 -O R W /dev/sda
```

8.4 Other patterns

Btest extensions may implement additional data pattens in addition or instead the default patterns. Please refer to the specific btest extension documentation (if any).

8.5 Mixing patterns

Compression aware pattern and dedup aware patterns can be mixed, as the first type is static full buffer range pattern and the second type affect only 16 first bytes. This is also the default settings. Offset based patterns are not compatible with compression aware patterns.

9 Verification

By default, btest does not check that the data is actually written, does not verify read data, and relies on the completion error codes for errors detection. For the storage exercising/load generation use cases this is good enough. For data verification tests the following verification mechanisms are provided:

- Stamp based verification (*options -c/-v*) – provides basic read/write data verification
- Persistent stamp based verification meta-data (*option -m*) – provides the ability to stop and restart btest sessions without losing the verification state, the ability to coordinate several concurrent btest processes and as a simple snapshot verification mechanism
- Offset pattern based verification (*options -O with -c/-v*) – to be used in addition to the stamp based verification or instead, as a low resource verification option
- Verification mode (*option -C*) – special read only verification mode to check stamp based verification with persistent meta-data

Note that all verification mechanisms use the stamp block unit as their base unit (see “Dynamic patterns” above).

9.1 Stamp based verification

In this mode, activated by option *-v* (*--verify*) or -c (*--check*), the dedup stamps (see “Symbol stamping and dedup control ” above) are recorded in memory during each write

and are compared to the device data stamp after each read. The difference between *-v* and *-c* is that *-v* makes btest record and report verification errors, and *-c* makes it panic on the first verification error. Note that the reuse of the dedup stamps is intentional, such that dedup aware patterns can be checked too. The downside of this is that high dedup factor rate may cause btest to use small range of stamp symbols and by that limit the verification effectiveness. Note that you can use the format/trim options (see "Data set initialization" below) to force the verification meta-data into a known state. Without that, each (stamp) block state is considered unknown (i.e. cannot be checked) until the first write to this block is completed.

Examples:

50% random write, 50% random read, fixed 4k block size, stamp block 1024 bytes, no dedup pattern (but with stamp), stamps verification, synchronous IO, single thread, 60 second test:

```
btest -P 1024 -p 0 -v R 50 /dev/sda
```

Same using check mode (panic on errors):

```
btest -P 1024 -p 0 -c R 50 /dev/sda
```

9.2 Verification mode

Btest provides a special running mode for data verification. In this mode, for each target device, btest uses the corresponding verification metadata file to sequentially scan the target file, and any data blocks that have valid verification data are read and checked. When reaching the end of the device, btest exits immediately. Blocks without verification metadata are skipped.

Notes:

- In this mode the command line parameters are different from the regular mode, such that there are no random ratio and read ratio parameters (see the example below).
- It is possible to run verification scan in parallel to another btest processes that share the same verification metadata file ("Persistent verification meta-data uses", "**Concurrent data check**" item).
- The data generation parameters must match the scan(check) parameters. In particular the stamp blocks must match (or at least the check stamp block must be less or equal the data generation stamp block), and obviously, the data range parameters much match.

Examples:

Perform 100% random write on the first 10GB of the device, stamps verification with md file using 1k stamp blocks, synchronous IO, single thread, 60 second test:

```
btest -P 1024 -l 10G -m /tmp/verf R W /dev/sda
```

Scan the same 10GB using 3 threads and check the written blocks:

```
btest -P 1024 -T 3 -C /tmp/verf /dev/sda
```

9.3 Persistent stamp based verification meta-data

In some cases, it is required to keep the verification meta-data generated by the stamp based verification across btest sessions. This is done by specifying a meta-data (base) name such that each a meta-data file for each device is created or loaded upon test start, and synced back to the file upon test end. For example the following line

```
btest -m md R W /dev/sda /dev/sdb
```

will cause btest to use “md-_dev_sda” as the meta-data file for /dev/sda device and “md-_dev_sdb” for /dev/sdb device. Meta data base names can (and maybe should) include a prefix directory.

If only a single device/file is used, then the meta-data base name is used as an explicit path for the meta-data file.

Note that also format/trim operations are done only by the first process to open the meta-data file (if requested).

Of course, all btest processes sharing the same data set must use this option to maintain the verification ability.

9.3.1 Persistent verification meta-data uses

1. **Multiple phase test:** for example, you may wish to perform a long test, force complete data set verification and issue another test (or another cycle of test) example – write random 4k blocks using dedup factor 2 for 60 seconds, then force entire data set check, then continue with mixed random read and writes using dedup factor 4:
   ```
   btest -p 200 -m /tmp/md R W /dev/sda
   btest -C /tmp/md /dev/sda
   btest -p 400 -m /tmp/md R 50 /dev/sda
   ```
2. **Concurrent btest processes:** make multiple concurrent btest processes to shared and verify the same data set. For example – the following first btest instance uses 4k random writes and read, the second uses 32k random writes and reads:
   ```
   btest -p 200 -m /tmp/md R 50 /dev/sda &
   btest -p 400 -m /tmp/md -b 32k -P 4k R 50 /dev/sda &
   ```
3. **Concurrent data check:** check data validity without stopping the main test. For example – the first following main btest is kept running while the second one is used to verify the data set:
   ```
   btest -t 3600 -m /tmp/md R W /dev/sda &
   #(after a while)
   btest -C /tmp/md /dev/sda
   ```
4. **Snapshot validation**: a simple snapshot validation testing can be implemented as follows: start the first btest on the source device, stop it, take a snapshot copy the validation meta-data file to match the snapshot device name and then restart the first btest and/or start another btest on the snapshot. For example – the following first creates a LV (logical volume) and test if for 60 seconds using random writes, then takes a snapshot, copies the md file and finally continues testing both LVs:

```
lvcreate -L 10G -n parent system
btest -p 200 -m /tmp/md R W /dev/system/parent
lvcreate -L 10G -s -n child system/parent
cp md-_dev_system_parent md-_dev_system_child
btest -p 400 -m /tmp/md R 50 /dev/system/child&
btest -p 400 -m /tmp/md R 50 /dev/system/parent&
```

10 Real time checks

Btest is able to check that the test flow is sane. Currently too checks are implemented: An activity check and timeout check.

10.1 Activity check

To be able to discover long system hiccups, btest is able to check that at least one IO is completed during the last differential report interval (see "Real-time reports" below). This option is activated using the *-A option (--activity_check).* Once activated, an report interval that passed without that at least a single IO succeeded will make btest to stop.

Examples:

100% random write, generate differential reports each 5 seconds, and stop if there was no activity during the last 5 seconds, 60 second test:

```
btest -r 10 -A R W /dev/sda
```

10.2 Timeout check

To be able to discover single IO hiccups (latency spikes) or even stale IOs, btest is able to validate that all IOs completes within a predefined amount of time. This is controlled by the *-B <timeout_ms>* (or *–timeout <ms>*) option. If this check is used, every completed IO is checked to validate that its latency is under the predefined threshold (in milliseconds), and every fixed time interval, all IOs are checked to discover stale IOs. The stale IOs check interval is either one second or the timeout threshold (rounded to seconds), which ever is bigger. In any case, if an IO fails the timeout check, btest will stop.

Examples:

100% random write, stop if an IO takes more then 400 msec to complete, 60 second test:

```
btest -B 400 R W /dev/sda
```

11 Data set initialization

Data set initializations are used to reset the data set to a known state. This can be important to resized files and for devices that are content aware and/or for thin-provisioning devices.

11.1 Pre-Formating

The *-F* options can be used to request a pre-format operation for each target device/file. During the format process, btest sequentially writes big blocks filled with zeros to the

entire data set range. Note that the actual effect of such writes is different from target device and/or files system writes. For example, on most file systems (e.g. ext4, ntfs), such write operations force the file system to resize and/or pre-allocate the dataset space and for some (ntfs, xfs) this allows for a more efficient allocation (extent allocation) compared to on-demand data allocation. Other file systems such as ZFS may use the zero block writes as de-allocation commands (hole punching). Block level devices may have the equivalent behavior, such that some of them may be forced to per-allocate the actual space, while other may use such zero writes to reclaim the space.

To optimize the pre-format operation, all devices/files are formatted in parallel, each by a single thread.

Examples:

100% random write, fixed 4k block size, pre format each file, one thread per file, 60 second test:

```
btest -F R W /tmp/file1 /tmp/file2
```

Note that the per-format operation may be mandatory if the specified files need to be created or resized.

11.2 Pre-Trimming

Trim is a SSD specific operation. See “Trim“ below.

12 Complex workload patterns

By default, btest uses a single workload pattern. In most cases this is the desired behavior to ensure that the results can be easily interpreted to enable a clear understanding of the measured performance. On the other hand, such single pattern workload is very synthetic and may not represent a real life application workload. In order to simulate more complex patterns btest supports an option to provide a workload pattern definitions file such that up to 10 concurrent workloads may be defined.

The Workload definitions file is specified using the *-f <filename>* option. This mode replace most of the standard workload definition flags so the following flags should not be used from the command line:

- -b/--block_size
- -a/--alignment_size
- -o/--offest
- -l/--length
- -p/--dedup
- -P/--stampblock
- the sequential/random ratio parameter <S|R|rand-ratio>
- the read/write ratio parameter <R|W|read-ratio>

The other flags, affecting the IO modes, threads number, reporting, etc., must still be specified in the command line or not at all. This changes the btest invocation to the following:

```
btest -f <workloads_file> [other options] <device1>
<device2> …
```

12.1 Workload definitions file format

In the workload definitions file, each line represents a workload. Each workload definition line begins with a weight of this workload, that must be a number between 1 and 100. This weight specifies the ratio that this workload is used relative to the other workloads. For example, if the first line has weight of 10 and the second line has a weight of 20, then the second workload is to be used 20 times for each 10 time of the first workload usage.

The weight is followed by the workload options that are any subset of the following options:

- *-b/--block_size* – see *"Setting the block size and alignment" above*
- *-a/--alignment_size* – see*"Setting the block size and alignment" above*
- *-o/--offest* – see *"Setting the test data range" above*
- *-l/--length* – see *"Setting the test data range" above*
- *-p/--dedup* – see *"Symbol stamping and dedup control " above*
- *-P/--stampblock – see "Dynamic patterns" above*

Finally, the sequential/random ratio parameter *<S|R|rand-ratio>* and the read/write ratio parameter *<R|W|read-ratio>* are specified in the same way they are used in the command line (see "Basic usage" above).

Example of a workload definition file content:

```
10  -b 4k  -p -1 -l 200m R W
50  -b 32k -P 4k -p 10 -l 100m -o 100m R 30
20  -b 8k  -a 4k -p 3 -P 4k -l 100m R 50
```

Notes:

- Currently the data initialization options will operate on the range that start from the lowest offset used by any workload to the highest offset used by any workload. In other words, unused areas in this range are initialized too.
- The stamp block size (if used) must be coordinated among all workloads such that all offsets and sizes are aligned to it. By default btest will try to use the smallest stamp block size specific by any of the workloads. This may be a suitable size, but if not, btest will panic and refuse to run the test.

13 SSD specific options

SSD devices can come in many configurations, protocols and form factors. The following section is probably relevant mainly to SSD devices that mimic magnetic (spinning) disks and use standard SCSI (SAS) or IDE (SATA) protocols.

13.1 Trim

The *-X* option is used to request a pre-trim operation. Trim is an IDE specific verb meant to help (SATA) SSD devices to reclaim the trimmed space such that it can be used to optimize the device's operation. Most modern SATA SSD devices support such trim operations (e.g. Intel's X-25M device). Note also that trim is not the same as secure erase. Trim operations can be done instead or in addition to the format operation (see “Pre-Formating“ above).

Note that for most SSD devices, the TRIM operation is much faster then formatting the data, and it should improve the initial device performance at least until this space is used again. On the other hand, due to these performance effects, it may be wiser to fill the SSD with random writes to avoid such behavior changes. To sum this up, the advised method to test SSD in the general case is:

1. Trim the entire SSD to remove existing patterns effects
2. Fill the entire SSD with random 4k random writes
3. Run the test itself

Examples:

100% random write, fixed 4k block size, pre-trimming, single thread, 60 second test:

```
btest -X R W /dev/sda
```

13.2 Trim after write

Modern file systems (e.g. Win7's NTFS and Linux's ext4) attempt to help SSDs avoid performance degradation by notifying the underlying SSD device about unused blocks using the IDE TRIM command (see also the section above). Such a mechanism allows the SSD to utilize free blocks for its own internal IO mechanisms/flows and by that optimize its operation. To be able to mimic such behavior, btest supports a “trim after write” option that causes btest to issue TRIM commands after each write, to mimic a “write-replace” scenario where the file system writes new data blocks that replace some old data blocks. The option activating this feature is *-x <trim block size>*. For example:

Examples:

100% random write, fixed 16k block size, each write follows by a trim of 8k in size, single thread, 60 second test:

```
btest -x 8k -b 16k R W /dev/sda
```

14 Reports

14.1 Real-time reports

Real time reports are provided to allow real time monitoring of the test status, use as sample points for later offline graph generation (or even realtime graph generation), and to allow detection and analysis of temporal phenomena. There are two kinds of real-time reports – subtotals, and differential subtotals. The subtotal reports provide statistics collected starting the beginning of the test until the report printing time, and the differential subtotals reports are reports that sum up the period of time from the last time such report was printed until the printing time of the current report. The frequency of each report is controlled using a dedicated option: option *-r <frequency in sec>* for the differential reports and *-R <frequency in sec>* for the subtotals. Value of 0 disables the corresponding report.

For most use cases, the differential reports are much more important for monitoring, graph generation and temporal phenomena analysis so by default only differential reports are enabled to be printed every 10 seconds, and subtotals are off.

Examples:

100% random write, generate differential reports each 10 seconds, and subtotal reports each 30 seconds, 60 second test:

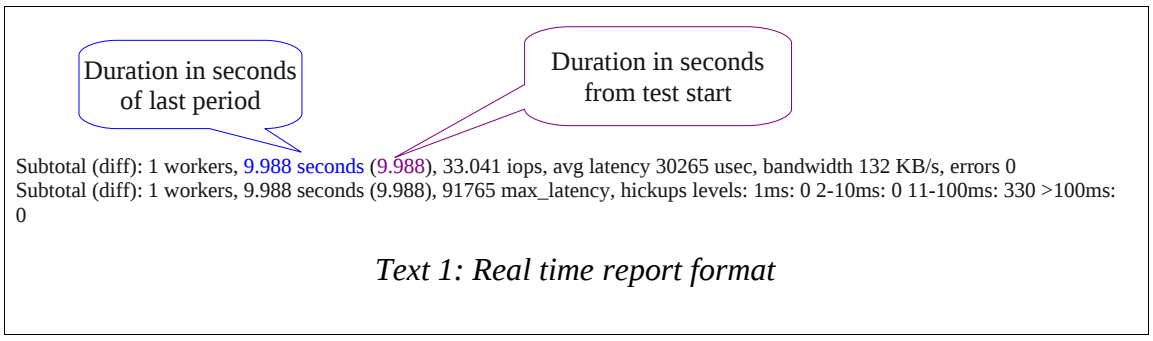
```
btest -r 10 -R 30 R W /dev/sda
```

100% random write, disable differential reports, print subtotal reports each 20 seconds, 60 second test:

```
btest -r 0 -R 20 R W /dev/sda
```

14.1.1 Real time reports format

Each report is composed of at least two lines (depicted in ”Text 1: Real time report format“), the first summarizes the IOPS, average latency, bandwidth and errors during the relevant period of time, and the second line provides latency histogram.



Text 1: Real time report format

Both lines starts with a fixed header to provide information on the type of the report (“Subtotal (diff):”), the amount of threads used (“X threads,”) the duration in seconds of the relevant differential period and the absolute duration from the test start (“x.y seconds (x.y),”.

Absolute subtotal reports (non differential reports) have the same format but the header is different. For example (note that the two lines are broken):

```
subtotal: 1 workers, 9.972 seconds (9.972), 33.392 iops, avg latency 29946 usec,
bandwidth 133 KB/s, errors 0

Subtotal: 1 workers, 9.972 seconds (9.972), 58425 max_latency, hiccups levels: 1ms: 0 2-
10ms: 0 11-100ms: 333 >100ms: 0
```

14.1.2 Per worker reports

By default, btest prints only aggregated reports, i.e. reports that sum up the results for all threads and/or aio requests. The option -z makes btest print per worker reports. Note that Worker is an IO context, in non aio modes there is a single worker per thread and in aio mode there are window_size workers per thread per file. Each line in these reports is a general statistics line for a different thread, a line that resembles the subtotal reports (see the above section) but its header is different – it specifies the threads number, the name of the file that the thread is operating on, the start offset, and the end offset. The rest is identical to the second part of the subtotal reports. For example:

```
#(differential per thread reports)

Worker 0: /tmp/kkk 0 104857600: last 10.018 seconds (39.976), 22.360 iops, avg latency
44721 usec, bandwidth 89 KB/s, errors 0

Worker 1: /tmp/kkk 0 104857600: last 9.967 seconds (39.951), 22.072 iops, avg latency
45306 usec, bandwidth 88 KB/s, errors 0

#(subtotal per thread reports)

Worker 0: /tmp/kkk 0 104857600: 39.976 seconds, 25.565 iops, avg latency 39115 usec,
bandwidth 102 KB/s, errors 0

Worker 1: /tmp/kkk 0 104857600: 39.951 seconds, 23.328 iops, avg latency 42866 usec,
bandwidth 93 KB/s, errors 0
```

Warning: using this option with large number of workers/threads may affect the test, and may make the results less accurate. Use with caution.

14.1.3 Real time reports triggered by signals

Regardless if the reports are enabled or disabled, real time report generation can be triggered by sending signals to the btest process: SIGUSR1 will generate a subtotal report, and SIGUSR2 will generate a differential report. Note that the differential reports generated by a signal will also affect the repeated internal reports generation (by shortening the last report period) so it must be used with care.

14.2 Summary reports

Summary reports are printed when the btest is finished. These reports are identical to the absolute (non differential reports) print out short of the header that specifys the report type (“Total:”), and the absolute test duration (“60.030 seconds,”).

For example:

```
Total: 60.030 seconds, 47.309 iops, avg latency 42331 usec, bandwidth 189 KB/s, errors 0

Total: 60.030 seconds, 145310 max_latency, hickups levels: 1ms: 0 2-10ms: 0 11-100ms:
2819 >100ms: 21
```

Note that some options and/or features add other lines to the summary report, and the per-worker report option (*-z*) applies here too.

15 Debugging

Btest is equipped with two main debugging mechanisms: the debugging traces and the real time binary traces used to debug verification tests.

15.1 Debugging levels

The *-d* options (*--debug*) controls the btest's debug level. By default debug is off. A single *-d* option is setting the debug level to 1. Any additional *-d* flag increases the debugging level. Only 5 debugging level are significant:

- No debug (level 0): no debug traces are printed
- Configuration debug (level 1): prints out traces of various setup/wrap up time operations to allow better insight of internal operations
- Enhanced configuration and verification debug (level 2): prints out more traces during setup/wrap up time, and also print traces per stamping/verification operations
- IO level debug (level 3): prints out detailed traces per IO. Note that this level already significantly affects the test performance
- Data level debug (level 4): in addition to the level 3 debug, hex dumps any data buffer read or written to the test devices. This level produce enormous sized logs. And is mainly used to debug the data stamping generation and data verification.

Examples:

100% random write, generate differential reports each 10 seconds, and subtotal reports each 30 seconds, 60 second test, show configuration debug trace:

```
btest -d R W /dev/sda
```

Show sample of 256k buffer hex dump with dedup stamps and offset stamping (exit after one operation):

```
btest -dddd -p 200 -O -n 1 R W /dev/sda
```

15.2 Verification traces

In addition to the simple debug text traces, a binary verification trace mechanism is provided to be able to debug verification runtime operations. This mechanism is activated using the *-L <debuglines>* (or *–debuglines <lines>*) where “lines” are the number of debug lines kept in memory. Each line consumes about 50 bytes. The required number of lines depends on your storage performance and required backlog size. For the purposes of debug line sizing, you can assume that a single IO will use up to 10 debug lines. This means that if you have a storage that can do 1000 IOPS, and you need 5 seconds backlog, you will want to use 1000 * 5 * 10 => 50000 debug lines. The debug line buffer is used

as cyclic buffer, meaning that each new debug line is always overwriting the oldest debug line. At the end of the test (either due the test end or due to panic), the binary buffer is translated to human readable format and is written to a temporary file named "*/tmp/debuglines-<tid>.log*" .

The traces format and its meaning require low level understanding of the verification system. If you need such understanding – read the code.

16 Implementation

Btest is implemented in (plain good old) C. Many efforts are invested in btest to make it simple (as it can be) fast, and reliable. Still significant system programming skills are required to be able to understand its low level implementation. In particular, the various Linux IO modes must be understood, as well as the threading model and shared memory synchronization techniques, in addition to a good understanding of "classical" IO performance issues. Btest coding and low level documentation assumes you have good understanding of the mentioned topics.

16.1 Low level documentation

Btest uses doxygen to produce low level HTML based documentation using javadoc style comments. In addition, regular comments are added for clarify important issues. Short of that, the code is the documentation.

16.2 Symbol stamping implementation

The *-p* (or *–dedup*) option sets the expected dedup factor, denoted in percents. This is implemented by making btest use a symbol range of 1/dedup_factor.
For example, if /dev/sda is 1GB large, and the following btest parameters are used

```
btest -p 200 R W /dev/sda
```

In the above case the IO block size and the stamp block are both 4k (the default) so we have 256 M blocks of 4k, meaning that the symbols used are from 1 to 128M (2 power of 27) range. In other words, we use a symbol space that is half of the possible block space so after writing all block at least once we can expect to find each symbol written twice (in the average), reaching our expected dedup factor of 2.

16.3 Stamp verification implementation

The verification meta-data files are created on first access, or used if they already exists. The meta-data files are memory mapped to each btest process and updated in real-time. The last process to exit without panic syncs the memory mapped files back to the persistent file storage. Note that a panicing process may leave the meta-data in an inconsistent state such that the entire meta-data file must be considered unusable after such event.

Each read and each write operation generates a new stamp before they start the operation. Then the operation tries to suggests its stamp as the next data stamp for this block. The first operation to access the block sets its stamp as the new stamp. As long as at least one operation is referencing the specific block, its data stamps cannot be changed. During that period each read checks that the data stamp is either the new or the old one, but not any

other. Any write operation operating on this block must use the suggested new stamp to build their data. The last operation to decrement its reference to the block, checks whether at least a single write operation operated on this block. If such operation or operations exists, the old stamp is replaced in core by the new suggested one. If only read operation access this block, the new suggested stamp is dropped and the old stamp is kept.

The above algorithm is implemented using lock-free synchronization verbs and is designed to affect the normal test flow as little as possible.

17 Alternatives

Here is a very partial list of open source and commercial tools that can be used to exercise/benchmark block devices:

- IOMeter – a very common benchmarking tool. Window GUI and worker, Linux worker only support.
- Data Test (DT) – a very powerful tool to verify block level devices (http://www.scsifaq.org/RMiller_Tools/dt.html)
- XDD: general IO generation tool (http://www.ioperformance.com/default.html)
- Bonnie++ (http://www.coker.com.au/bonnie++/)