`blkreplay` and Sonar Diagrams

A manual for system administrators,
kernel developers,
hardware technicians,
and experts in IO systems

Thomas Schöbel-Theuer (`tst@1und1.de`)

Version 1.0.0

Abstract

`blkreplay` is a GPL'ed toolkit, driving the block layer of Linux (or other Unix-like OSes) while measuring latency and throughput of IO operations for later visualization (so-called "sonar diagrams" and others).

`blkreplay` comes with a modular and extensible **test suite**, automating large projects for testing and/or benchmarking.

`blkreplay` can be used to **test physical hardware**, e.g. compare different brands of hard disks, or RAID controllers / their settings / RAID rebuild performance degradation, or to evaluate the effect of SSD caching, or to compare different block level transports like iSCSI vs Fibrechannel (over different kinds of storage networks).

It can compare **virtual hardware** (like `vmware` or `XenServer` block devices, or any type of block-level **storage virtualization**) to each other or to physical hardware, provided the test setup is handled very carefully[1].

`blkreplay` can compare **commercial storage** systems from vendors like EMC, NetApp, IBM, Hitachi, etc to each other or to cheap off-the-shelf hardware (in order to determine the price/performance ratio), provided the same care[2] is taken.

Furthermore, it can be used for tests of the **Linux kernel**, e.g. for testing device drivers, comparing IO schedulers at different load patterns, determining the overhead of Linux `dm` targets or the impact of network problems to DRBD, and much more.

In addition to artificial loads like random read-write sweeps and various kinds of *overload tests*, it can also replay **natural loads** which have been recorded by `blktrace` at heavily-loaded production servers at big data centers. `blkreplay` comes with a **large collection of natural loads** from a wide spectrum of applications (such as web servers, databases, dedicated servers, etc) which have been released to the public by 1&1 under GPL. Some of these natural loads have recorded the real-life disk access behaviour from servers serving thousands of customers in parallel. Static analysis of the workingset behaviour of such natural loads is implemented.

At 1&1, `blkreplay` has even been used as a tool for root cause analysis of incidents: for example, high load peaks presumably stemming from traffic jam (or other sources of overload) were recorded at production sites in real time by `blktrace`, and later replayed in the laboratory (without causing customer impact) seeking for the cause of trouble, or improving the safety margins by choice of better hardware.

For experts in IO subsystems, visualization techniques like "sonar diagrams" can reveal (parts of) the internal structure of complex IO systems, such as cache hierarchies or other hierarchical storage systems.

As a community project under GPL, `blkreplay` is open to contributions from hardware vendors, other data centers, the kernel hacker community, etc.

[1] Otherwise you may get *useless fake results* measuring the *cache* performance or even *sparse accesses to holes* instead of the real hardware performance. Such fake results may differ from real results by *factors*, er even by *orders of magnitude*. `blkreplay` comes with thorough descriptions teaching you how to avoid the most common pitfalls.

[2] Notice that most commercial storage systems ***are*** in fact nothing but virtualized storage, so the above warnings about possible *fake results* apply.

Contents

1. Why Synthetic Benchmarks suck

There are a lot of benchmark tools around, like `iozone`, `iometer`, `iperf`, `bonnie`(`++`) and many others.

What do they have in *common*[1]?

> Simply, most of them generate an **artificial load** onto your system.

Artificial loads, as opposed to **natural loads**, have a main disadvantage: they cannot answer questions like "will my application Z run on system A reliably?"

question	artificial	natural
Is system A better than B for application Z?	*partly*	√
Does application Z run on system A?	-	√

It seems that some people *believe* that synthetic benchmarks can be used even at the position of the dash in the above table. These people are wrong.

Experiences at big data centers at 1&1 show that sometimes the differences between results from artificial benchmarks and real-world application behaviour are very large. We found cases where artificial benchmarks (adjusting parameters like IOPS) suggested that a particular application *should* run on a particular hardware system, but the real application *didn't*: after deployment, a systematic series of incidents *disproved*[2] the validity of the former benchmark results for the *originally intended statement*. The failed prediction from the artificial benchmarks led to a **failed invest**.

What can we do about that?

Obviously, parameters like IOPS (even when enriched with attributes like "average IOPS" or "peak IOPS") are *not* representative for description of the real-world behaviour of applications. Attempts to describe real-world behaviour in mathematical terms of analytic functions have been already made in the 1970's; they failed. All such models can *try* to describe an *approximation* of real-world behaviour, if enough knowledge about the application *would* be available.

So why trying to deal with tools that never can fully describe real-world application behaviour, when there *exist* tools which definitely *can* do?

One of them is `blkreplay`.

[1] Of course, the mentioned performance measuring tools are targeted at inspection of different components of an OS, such as network, filesystem layer, and block IO layer. Here the question is a about *commonality*, not differences.

[2] Sometimes we got results in the other direction: artificial benchmarks suggested that a particular application would *not* run, but in reality it *did* run.

2. How `blkreplay` works

2.1. Principle

In some sense, `blkreplay` is just the opposite of the well-known Linux kernel tool `blktrace`: recordings made by `blktrace` are simply replayed on *another* block device.

A `blktrace` record of an IO request contains the following information:

1. timestamp of the IO request (nanosecond resolution)
2. position of the IO request (sector#)
3. length of the IO request (#sectors)
4. direction: R[ead] or W[rite]

Notice that `blktrace` records do *not* contain any data. Therefore, `blkreplay` must later generate some *fake data* in order to repeat the timely and positionly behaviour of the original recording. By default, NULL blocks enriched with some internal header information are generated. The internal headers may be used for *verification* of the correctness of IO semantics, either by immediate re-read after each write, or in a separate verify pass after the end of an ordinary `blkreplay` run.

Notice that these NULL blocks will (together with the internal header information) **destroy** any previous content (such as filesystem data) on your block device!

Therefore, *never* use `blkreplay` on production systems.

Always use `blkreplay` in the laboratory, always on devices which don't contain any valuable data!

Running `blkreplay` *in parallel* to mounted filesystems on the same device[1] will certainly destroy your data and almost certainly crash your kernel. Always run at most a single `blkreplay` instance on a single device!

In general, `blkreplay` is a tool only for *experienced* technicians who know what they do. They should be at least at a senior level.

2.2. Architecture of `blkreplay`

The main challenge for `blkreplay` is to generate **sufficient IO parallelism**.

Ordinary production systems like web servers are serving many thousands of HTTP requests per second, which may lead to an IO parallelism at the block level of several hundred outstanding IO requests at the same time.

In order to simulate such a behaviour in the lab, there are principally two alternatives:

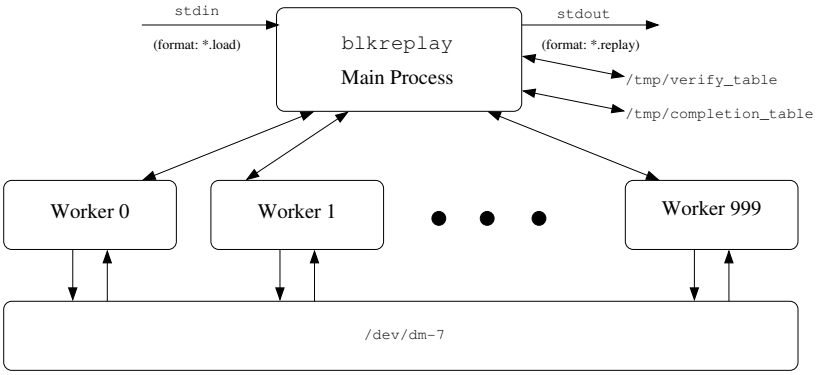
1. use the `aio` interface of the kernel to fire off a large number of IO requests in parallel.
2. use a sufficiently large number of kernel threads or processes in parallel, where each of them fires up only at most one IO request at the same time (blocking IO).

[1] Although a single Linux kernel instance tries to probihit such a disaster, there are cases where you can "achieve" that effect. Examples are iSCSI connections to the same iSCSI target in parallel.

The current version of `blkreplay` supports only method 2; method 1 is planned for a future release.

Method 2 is motivated by a typical Apache behaviour: it is almost a "fork bomb", in particular under high connection rates and slow block devices. A high number of ordinary Unix processes is doing conventional `read()` / `write()` operations in parallel.

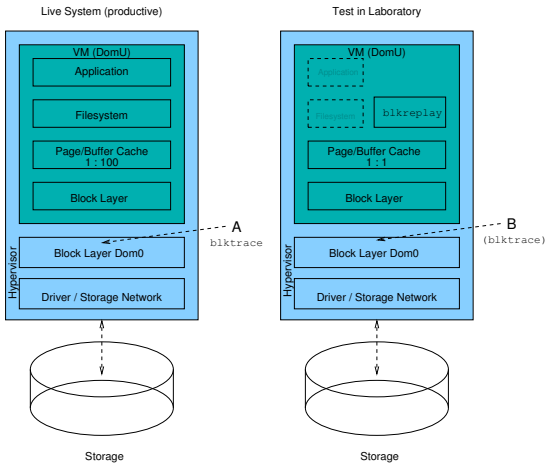
For easy portability even to historic Unix flavours, `blkreplay` uses ordinary Unix processes generated by simple `fork()`s and communicating via anonymous pipes, in favour to a shared-memory `pthreads` model. However, future versions may also support `pthreads`.



2.3. Mode of Operation

It is **essential** that you understand the concepts described in this chapter. Otherwise you may produce **useless fake results**, deviating from valid measurements by *factors*, or even *orders of magnitude*.

It is crucial to understand the *operating environment* where `blkreplay` is running. Please take a look at the following picture:



In the live system, `blktrace` will "listen" to the events occurring at point A, and will record them. In this example, we have a complex system, running virtual machines inside a hypervisor.

Notice that all IO requests of the application will not only go through the filesystem, but also through the buffer / page cache. The in-memory cache will usually serve most IO requests from the application, without causing physical IO at the block layer (so-called "cache hits").

On some well-tuned production servers, it is no problem to achieve cache hit rates of 99% or more, leading to a kind of "gear ratio" of 1:100, or even 1:1000 (in long-term runs). Of course, there also exist heavy workloads running on thin servers, where sometimes less than 1:10 can be achieved. Even in that case, there will be *always* some cache hits, for example caused by metadata requests from the filesystem. In practice, the cache hit rate will never go down to 0%. Notice that these inevitable cache effects are *already included* in any `blktrace`!

Now look at the situation in the laboratory. The application and the filesystem is no longer present, but its *effects* are simulated by `blkreplay`. Due to the architecture of the Linux kernel, all IO requests will continue to run through the buffer cache[2].

It should be clear by the very nature of our experiment, that at measuring point B *exactly* the same events *should* happen as had been formerly observed / recorded at point A.

Thus, the page / buffer cache of the laboratory system *must* be switched off. Otherwise, a "gear ratio" of 1:10 (or let it be only 1:1.1) would lead to distortions of the measurement results. In order to switch off the page / buffer cache, `blkreplay` uses `O_DIRECT` mode as offered by the Linux kernel.

Notice that even by these measures, there may remain some subtle differences between the operations occurring at point A and point B. The block layer of the Linux kernel does some optimizations, for example it tries to *coalesce* adjacent requests, or to *split* some requests, depending on the capabilities of the hardware. Usually, these are minor modifications, occurring at less than 1% of all requests. However, keep in mind (and check) these effects.

The **elevator strategy** (aka IO scheduler) of the Linux block layer is a bigger influence factor. It can *reorder* requests, and it can even add *artificial latencies*. For example, `CFQ` adds some speculative latencies in some cases to increase the chances for request mergers. Selecting the "wrong[3]" scheduler may lead to larger distortions, even to seemingly "bad" behaviour (which is *not* the fault of bad hardware). In order to really get (almost) the same behaviour at points A and B, you *must* select the `NOOP` or at least the `DEADLINE` scheduler. See `/sys/dev/block/*/queue/scheduler`.

2.4. Overlapping of IO Requests

In general, there are two kinds of overlapping between IO requests in real production systems (at the time when a `blktrace` record is made):

1. timely
2. positionly

Both kinds form a two-dimensional space:

overlapping	timely yes	timely no
positionly yes	-	√
positionly no	√	√

Pure timely overlapping (without positionly overlapping) is a frequent case in almost any IO system (usually called "IO parallelism" in folklore). In opposite, purely positionly overlapping (without timely overlapping) is also a frequent case, for example when the same sector is re-read after a while, or re-written when the contents of a file changes frequently. Completely unrelated requests (neither timely nor positionly overlapping) are probably the most frequent case in most practical load scenarios at production sites. There is no problem with any of them, indicated by the checkmarks.

However, what about both kinds of overlapping at the same time?

The case of *both* timely and positionly overlapping (simultanously) of IO operations is called *damaged IO*.

In ordinary OS kernels, damaged IO usually *never* occurs. Here are the reasons:

IO requests are usually generated by in-kernel memory caches like buffer caches or page caches. Even in case of databases using Direct IO when submitting requests to the block layer,

[2] Several commercial Unices used a concept called "raw device" which circumvented their buffer cache. In contrast, the internal structures of Linux device driver are internally interwoven with the page cache in a rather sticky way. Instead of "raw devices", Linux uses the concept of "direct IO", which *tries* to minimize any caching effects.

[3] Many sources from the internet claim that `CFQ` is the "best" IO scheduler. While this is often true for typical workstation load pattern (and interactive user expectations), our experience at 1&1 is different with regards to *server* loads. Check yourself! Run some `blkreplay` comparisons between different IO schedulers. Hint: there may be even non-linear dependencies from the lower level, whether you have some RAID with BBU cache, or not.

their internal database buffer cache works similarly to in-kernel caches. It simply makes *no sense* to write to the same sector twice at the same time, because the result will be undefined. A similar argument holds for reads in parallel to writes to/from the same sector.

In theory, concurrent reads from the same sector would be possible without causing harm to data integrity on the block device. In operating system caches, this would introduce *copies* of the same data into the buffer or page cache, violating its internal *uniqueness* properties stating that any sector is cached at most once. Consequently, this case also never appears *in practice* at *real-life systems(!)*.

However, in terms of **correctness of storage semantics** the case of damaged IO involving read/read to the same sector is *allowed*. Probably you already know the following table from database textbooks and others:

conflict?	read	write
read	no	yes
write	yes	yes

Some block IO systems like DRBD show some misbehaviour in case of concurrent writes to the same sector, or in some cases they even fail. Some DRBD versions[4] will at least delay further IO requests for several milliseconds, lowering IO bandwidth or even leading to temporary hangs.

So, damaged IO *should* be avoided under all circumstances. Failing to do so may result in a disaster; in general, some IO devices like elder tape drives may even be corrupted as a whole.

While avoidance of damaged IO is automatically guaranteed in real production systems by the buffer / cache page of the Linux kernel (or other components like database memory buffers), our tool `blkreplay` must be designed very carefully not to step into that pitfall.

Why is there a risk that `blkreplay` could (accidentally) start some damaged IO?

Well, replay of the original timing of requests is not always possible. Even if `blkreplay` *tries* to start IO requests in the same timely pattern as at the original site, a very slow device (or a heavily overloaded device) may delay an IO operation for a very long time. In overload scenarios, or in case of iSCSI network hangs, it is possible that some IO requests may take 5 minutes to complete (or even more, or even *never* complete in case of fatal errors). In such cases, it is not unlikely that a new write to the same sector is started before the old one has completed.

In order to avoid damaged IO, `blkreplay` uses some in-memory hash tables to detect any (potential) conflicts between IO requests.

In order to deal with (potential) problems caused by damaged IO, we use the following options in `blkreplay` to control its behaviour at *replay* time:

`--strong=0` For the sake of conflict detection, only write/write conflicts will count. This mode ensures that the "storage semantics" is obeyed with respect to the *written data*[5], but it allows reads to permute with writes (which is "wrong" in *strong* sense, hence the name). In essence, the following conflict table is used internally for conflict detection:

conflict?	read	write
read	no	no
write	no	yes

`--strong=1` (default) Use the standard conflict table as known from the literature. Only read/read is treated as non-conflicting:

conflict?	read	write
read	no	yes
write	yes	yes

`--strong=2` *All* damaged IO is treated as conflicting, even read/read. This may be useful for simulating the behaviour of *real* OS caches. This is equivalent to the following table:

[4] At present, this seems to be an undocumented behaviour observed by the author. Even if DRBD's behaviour may change in the future: damaged IO is a bad idea by itself. It would be unfair to blame DRBD for "psychologically unexpected" behaviour under illegal load patterns, which should never occur. In general, making code rubust against damaged IO could decrease performance during ordinary operation. Thus damaged IO should be avoided at its *source*.

[5] This allows to check the end result via `verify` modes for correctness.

conflict?	read	write
read	yes	yes
write	yes	yes

Indepently from the correctness criterion, the following operation modes may be selected. They determine the kind of reaction in case of detected conflicts:

`--with-conflicts` No countermeasures against damaged IO are taken. Consequently, is does not matter which `--strong=` mode you have selected before.

`--with-drop` Whenever a new request is conflicting with an old (already issued) request, it is simply dropped. This has no side effects, other than that some reads and/or writes may be missing (depending on `--strong=` mode). This has the lowest overhead of all conflict-avoiding methods. Depending on properties of the load, the number of dropped requests may be rather high. Please check the tail of the result file. In the statistics section, you will find the number of dropped requests. If that number is higher than, say, 5%, you should consider one of the following options.

`--with-partial` (default) Any conflicting requests (as determined by `--strong=`) are pushed back to an internal pushback list and kept there until the conflict is gone. Pushed back requests are immediately submitted as soon as the conflict has gone away. This results in a reordering of the *affected* requests, while *trying* to replay unaffected ones at original timestamps. This leads to a partial ordering of requests, which may be *very different* from the original ordering, and thus may "violate" the "original storage semantics" if it would make a difference on the replay system.
This mode leads to a 2-class society, where ordinary requests are processed faster than conflicting ones. Further details may be found in section 5.2.1.
Attention! as a side effect, this mode may *increase* the actual IO parallelism to a larger number than configured via the `--threads=` parameter, because some request slots (and in turn, also some threads) must[6] be reserved *in advance* for pushed-back requests. When conflicts are gone and the system tries to "catch up", additional IO requests may be submitted.
In practice, this can happen in particular with some Windows loads, where a lot of writes seem to be repeated. Frequently, pushed back requests (colored differently for better distinction from ordinary requests) are the main contributors to delays. Check them!

`--with-ordering` The main process spreads its IO requests to the worker processes in a round-robin fashion over their anonymous pipes. Whenever conflicting requests (as determined by `--strong=`) are detected, this spreading process will stop until the conflict has gone away. In consequence, the original ordering of requests will be preserved as much as possible. As a side effect, *all following* IO requests are also delayed, even if they don't conflict with anything else. This can lead to *artificial delays*.
In general, this mode is the "most robust" one.

In many cases, you will prefer `--with-partial`, which is the default. It delivers almost the same throughput as `--with-conflicts` without the downsides of `--with-drop`, while minimizing artificial delays[7].

However, `--with-ordering` is also useful in many scenarios. Practical experience from many hardware tests shows that the artificial delays caused by `--with-ordering` seem to be an *advantage*. Whenever such stalls occur more than seldomly, they act as an **indicator** for massive IO problems of the test candidate. `--with-ordering` is often a kind of "detector" for hardware problems, since it visualizes any problems in an eye-catching way.

[6] Otherwise a reactivated request from the pushback list could have to wait for an ordinary request slot to become free, which would result in an artificial delay. Experiments have show that such kinds of distortions can be serious.

[7] In extreme cases, pushed-back requests (visualized in stronger colors) can form some kind of "transitive queues", e.g. when pushback requests depend on other pushback requests transitively. In such a case, their delays can sum up over a longer time, independently from "normal" requests (clearly visible in the delay diagram). Although this is often a property of the load, you can *try(!)* to minimize such an effect by setting the expert option `ahead_limit` to values lower than 1s (however too low values will destroy throughput at all). Dangerous!

When you start a lot of threads (typically ≥ 256), `--with-ordering` may yield *better(!)* throughput than `--with-partial`. In such cases, the reason is *counter-intuitive*, as explained in section 5.2.2: artificial delays caused by `--with-ordering` will *decrease* the *average* request queue length actually occuring at runtime, which will in turn *increase* the average throughput, depending on counter-intuitive properties of your test candidate. A way to find out is just to run the same benchmarks with `--threads=16` or `--threads=32`. In case `--with-partial` is now better than `--with-ordering`, you likely detected that problem. Don't draw wrong conclusions from such counter-intuitive effects!

2.5. Verification of Storage Semantics

In order to allow verification of the sector headers and their timestamps / version stamps, `blkreplay` needs some temporary storage where information can be kept for a longer time than just during IO. Two temporary files are used: `/tmp/verify_table` and `/tmp/completion_table`. Both are sparse files, containing a simple (sparse) array of sequence numbers, indexed by the position (sector#). Whenever a write is started, a new sequence number is recorded in `/tmp/verify_table`. Whenever that write is completed, the same number is recorded in `/tmp/completion_table`. At any time, both tables keep track of the current progress of write operations.

Whenever the sequence numbers at the same position (sector#) are different between both files, we know that a write operation has not yet completed. If they are the same, we know which sequence number should appear in the header of the corresponding sector.

There are following variants of verification modes:

`--with-verify` Whenever a sector is read (by a regular read request) which has been written before (by an ordinary write request), the read data is checked against the sequence numbers from the tables. Any mismatches are reported by the string `VERIFY ERROR` in the result file. Thus, `zgrep "VERIFY ERROR" *.replay.gz` will show them to you. In addition, the statistics section at the end of the output file will contain some valuable information.
Warning! this mode can only reveal errors in the storage semantics if written blocks are re-read somewhen. When sectors are just written, but never read, this mode will not detect anything.

`--with-final-verify` In addition to `--with-verify`, a separate pass will be started at the end of a `blkreplay` run. All sectors which have been touched before, will be checked.

`--with-paranoia` In addition, any written sector will be immediately re-read during operation. This doubles the IO rate and leads to extremely high distortions of measurement results.

All verify modes will create temporary tables in `/tmp/`. Although the temporary files are sparse, they can use up a significant amount of storage (depending on the load). The additional IOs form a *sequential bottleneck*, and therefore can slowdown `blkreplay` considerably. Please use the verify modes only for *validation*, but not for measurements / determining performance!

3. How to use `blkreplay`

Running `blkreplay` naïvely without reading this chapter may easily lead to **completely worthless fake results** which would be only useful for production of bullshit!

In science, it would be unethical to produce such bullshit willingly or even deliberately.

In industry, usage of such bullshit (even inadvertendly) may easily lead to failed invests up to millions of Euros / Dollars (depending on the application and the size of your datacenter).

Most of the following advice will also apply to other benchmark tools like `iometer`.

3.1. How to Avoid Common Pitfalls

Don't skip this section! Read it *completely*, even if you are impatient or under time pressure!

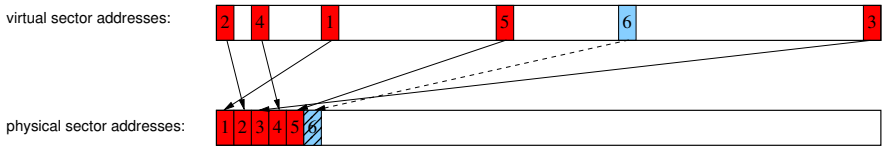
Modern IO subsystems often use some kind of **storage virtualization** interally. More often than you can dream of, concepts from storage virtualization are used (interally) in places where you don't expect them.

Example: seemingly "simple" SSDs or even some USB sticks(!) show some of the behaviours described next.

3.1.1. Pitfalls from Storage Virtualization

The basic idea of storage virtualization is some kind of "translation" (or "mapping") between *virtual storage addresses* and *"physical" storage addresses*.

In many[1] cases, the address translation / mapping is created on the fly, dynamically at runtime. In the following simplified[2] example, the timely order of accesses is marked by increasing numbers, while the type of IO request is indicated by colors (red = write, blue = read):



We start with an empty logical address space (often called "logical volume", shortly LV), having a logical size of several terabytes. Here, "empty" means that initially *no* assignment between logical and physical sector numbers exists. Now we start a short benchmark (whether `blkreplay` or others like `iometer`). When request #1 (a write) arrives, no physical location exists yet. Therefore, a new location must be assigned. In this example, the locations are always taken from the beginning of the physical address space.

Thus, all the physical sectors occurring in this example will be allocated in a very small and compact area at the start of the physical address space. Imagine the drawing not being true to scale: imagine a total size of several terabytes ($> 10^9$ sectors), and a `blkreplay` benchmark touching only a few thousands of sectors. What will be the effect?

[1] A promient exception is classical LVM as implemented by the Linux kernel: unless you use LVM snapshots or other advanced features, it uses almost static mappings, and it carries almost no observable overhead in many practical scenarios.

[2] For simplicity, this example assumes that the address translation uses the same granularity as the benchmark, e.g. single sectors. We also don't discuss the internals of the mapping which can also have a drastic impact on measurement results.

1. Only a *tiny* fraction of the physical space will be actually used, usually less than one per mille or even less than a millionth.
   In contrast, real world applications as well as real customers tend to use up significant space with real data, usually more than 50% (and up to 100%).

2. Even if accesses to the LV are (pseudo-)random, accesses to the "physical hot area" will *not* remain random: as you can see, they are translated to (purely / almost) *sequential* access patterns. If the physical addresses are residing on a mechanical hard disk, (almost) no seek operations will occur. Additionally, the physical operations are in *ascending* order, which is a classical use case for BBU-cached writes and/or readahead strategies. Notice that ascending sequential IO on hard disks is usually faster than random IO by a factor of 100 or even more (depending on hardware and further factors like RAID, between one and three orders of magnitude).
   In contrast, real-world writes will be spreaded much more randomly over the physical partition, and there will be a significant amount of *in-place updates* in many practical use cases.

3. Even worse, read requests need not be mapped at all (indicated by shading in the drawing). When reading from an address where never anything had been written before, NULL blocks may be returned on-the-fly, without causing any physical access at all[3]. Notice that such "fake reads" can be faster than true read accesses by *several* orders of magnitude.

4. Even in case read requests are also mapped upon first reference[4], perhaps leading to a physical IO (or perhaps not), the same arguments as for writes apply.

5. If you *repeat* the same `blkreplay` benchmark once again, immediately after the first run, you will get another surprise: this time the mapping between logical and physical addresses already exists, thus you will likely get different results, usually drastically better, but seldomly slightly worse, depending on the vendor of the storage virtualization (and on many other factors such as the size of the logical volume). In scientific terminology: your experiment is not truly *repeatable*.

What can you do about that?

There is no general solution for all cases. It depends on the *statement* you want to prove or disprove by usage of `blkreplay`.

The following is just an *approximation* if you want to reveal the real-life[5] behaviour of virtualized storage systems:

1. Whenever you start a new run of a benchmark, you ***must*** **delete** your old LVs, and create new ones. Otherwise, your old run will influence the new one in some way you cannot predict easily. Remember that the mapping table in the above example is a kind of "memory" which records not only the sector numbers occurring in your benchmark, but even their timely order. Make sure that this kind of "memory" is erased completely[6] between any runs!

2. After each fresh creation of a logical volume, **fill it with data**. This is the only reliable[7]

[3] A similar effect is known from holes in traditional Unix *sparse files*.

[4] Several commercial storage boxes are known to do so. However, notice that this behaviour is not documented, and thus not guaranteed by the vendors. They sell you a blackbox. All you can do is to analyze such behaviour if you are curious about their internals. In their next firmware release, the behaviour may be already different without notice.

[5] In real life, customer data or enterprise data is stored on LVs. Thus benchmarks of empty LVs are *completely wrong* if you try to reveal real life behaviour.

[6] Even delete your LVs if you believe that's unnecessary, because you have obeyed point 2 and have filled it with data to initialize the mapping: some storage systems make *re-assignments* of the mapping during your benchmarks. Because many commercial storage systems are blackboxes, you cannot immediately see that. Always keep in mind that usually ordinary benchmarks will only touch a *tiny fraction* of all physical sectors, compared to real life!

[7] Some storage vendors have internal functions which *preallocate* the space for a LV. Don't use them! Don't trust any claims that this would be equivalent to filling with random data - we found cases where we could *disprove* such claims, where results differed even by *factors*. Just fill your LVs with random data to be sure, even if this delays your measurements for some hours or even days. Keep in mind that later production systems will take weeks or even months to be filled with data before potential problems could show up, so don't hesitate to resemble such kind of effort in the laboratory.

way. Best practice is to use tools like `wipe`, filling the whole[8] LV with random data. Filling with NULL blocks is not recommended, because some blackbox storage systems might detect this easily and circumvent it by not creating a mapping at all (or even erasing an old mapping similar to `punch` operations).

3. *Immediately* after filling with random data, start your benchmark **exactly once**. *Never*, really *never* kill a run of `blkreplay` (or any other benchmark tool) and restart it. In case of any error or misbehaviour, you *must* start over with step 1!

4. There is a single exception if you really know what you are doing: immediately after the first run, you may restart the *same* benchmark once again, in order to reveal some hidden properties of the mapping. You *must* name your output files differently from the first run, and you *must not* confuse the meaning of the second run with the meaning of the first run.

3.1.2. Pitfalls from Caches

It is easy to be caught by these pitfalls (even if you try to avoid them very hard), since caches occur very frequently in almost any type of storage system, and even in places where you don't expect them. In addition, some real-life loads have hidden properties you cannot see at first glance.

3.1.2.1. Pitfalls from Cache Operation States

Many people believe that the most important case is *cold caches* versus *hot caches*. Although this is not completely wrong, it is not always fully true. There is another more important property of cache states: **steady state**.

Steady state is not the same as hot state. When you start your system freshly, many caches will be of course empty[9]. An empty cache is always "cold". During operation, it will be filled with data. A cache is called "hot", if the **cache hit rate** is "significantly high enough". What's that? All of these terms are rather vague and depend on the application. However, some caches may *never* reach "hot" state, for example when the cache design / architecture is "inefficient" (cf section 3.1.3). What then? The term "hot" is not the right one for describing that problem: when your cache never gets hots, your testing candidate will just fail your performance test, but your test as such will be valid: the result is just telling you that the cache is not tuned well enough for your application workload.

What is "steady state"?

Intuitively, it just means that "nothing changes (fundamentally) any more". In practice, there is a simple rule of thumb: your benchmark should just run **long enough** to get into steady state. In section 3.1.3, some theoretical methods are described how to compute the time δ until steady state is reached. In practice, just make a few experiments in order to determine steady state intuitively. Many people will get a good feeling for "steady state" after some experience.

Once you know when you have reached the "golden" steady state, you can do one of two things:

1. ignore all benchmark results from the time span before steady state has been reached.

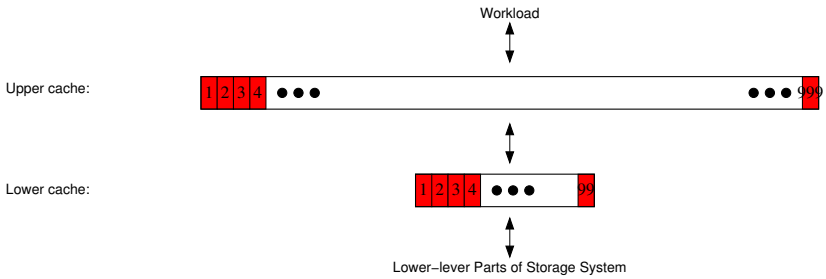
2. let the whole benchmark run *at least* 10x as long as the time to reach steady state. The longer, the better.

3.1.2.2. Pitfalls from Cache Size

The following illustration explains a general problem of cache hierarchies. Our example demonstrates an ill-designed behaviour: the lower cache is *smaller* than the upper one.

[8] If you are consciously concerned that filling the *whole* LV might not catch your usecase where (say) only 50% of logical space is actually used, you *could* try to fill only 50% of the LV. However, be *sure* to fill any blocks which occur in the benchmarks. Otherwise, the effects described above will almost certainly lead to a *higher* distortion of your results than just filling with 100%. Notice that some of the above effects deal with orders of magnitude, not just a few percent.

[9] There are some exceptions: SSD caches may start in a hot state (caused by your previous benchmark run), and BBU caches at RAID controllers may also survive even power failure.



Now assume that the LV has a size of several terabytes, and the workload is operating on a large part of that. On one hand, this is several orders of magnitude larger than the upper cache, but on the other hand typical application workloads will not access all sectors with the same uniform probability. In general, caches are only useful in two cases (non-exclusively):

1. the cache is strictly larger than the workload.

2. the workload contains unevenly distributed access frequencies.

Case 1 occurs only in special cases, such as in-memory databases (or workstation loads on large RAM machines). Case 2 appears more often.

Back to the above example: assume that only case 2 applies to the upper cache. Consequently, case 2 will also apply to the lower cache, because the lower cache is *strictly smaller* than the upper cache in this ill-designed example. Now assume we have an *inclusive*[10] cache hierarchy, and are using a standard cache eviction strategy like LRU. As a consequence from LRU (or any other strategy avoiding anomalies[11]), any sector present in the lower cache will be also be present in the upper cache[12]. As a further consequence, we see that the lower cache is superfluous: by removing it, the system could even become faster[13] due to less overhead.

What can we do about that?

Simply, just design your system in such a way that lower caches are always strictly larger than higher ones, by a factor of k. In order to be useful, $k \geq 2$ should be used, but for really good performance $k \geq 10$ should be selected.

Probably you already know this, and you think you don't violate it. However, it is possible you might violate it unwillingly. The standard case is a server from a data center, equipped with several gigabytes of RAM. Almost all of the main memory can be used by the buffer/page caches of the Linux kernel. Therefore, you already may have a rather large upper cache you didn't think about. As a consequence, caches at the block storage level (e.g. SSD caches etc) should be larger by an order of magnitude (in this case $\sim$ 1TB or more). At the time of writing this paper, some commercial storage systems don't match this seemingly simple requirement.

There is another variant of this pitfall: records made by `blktrace` are measuring the IO traffic *below* the buffer/page cache in most cases. Therefore, most (if not all) natural loads obtained by `blktrace` contain effects[14] of the caches of the original system. In some cases (e.g. published loads from the `blktrace` project), you don't know the original RAM size. Even if you know, you often cannot tell how large the page cache *really* was at the time of recording: how much RAM was spent for other purposes like processes, how much for other filesystems / partitions?

Even if you knew all that: do you know the *workingset size* of your application workload? Read on...

[10] In case of *exclusive* cache hierarchies, the whole picture can be *approximately(!)* replaced by a simplified one having only one cache level. The size of the new simplified cache is just the *sum* of the sizes of both original caches.

[11] Probably the best known anomaly is the famous FIFO anomaly, as explained in most text books about operating systems.

[12] This is just the *definition* of non-anomaly behaviour.

[13] Notice that there are exceptions. For example, internal memory caches present at hard disk drives are way too small to be able to contribute to classical hierarchical caching, but they can act as cylinder buffers for *local aggregation strategies* like readahead.

[14] One of the more well-known effects of caches is called *cache inversion*. At the time of writing this paper, Wikipedia didn't carry much about it. Consult a really good textbook or some research papers to learn more about it.

3.1.3. Pitfalls from Workingset Sizes

The workingset theory has been developped by Denning in the late 1960s, and has been originally used for the description of the behaviour of hardware-MMU-based paging / swapping systems and their strategies like LRU. It is also useful in other areas, such as block storage. Here is an adaptation of Denning's theory to our needs:

$$WS(t,\delta) = \{\text{set of all sectors touched in the time interval [t-}\delta\text{,t]}\}$$

Usually, δ is treated as a constant, called *window size*. Then the workingset $WS(t,\delta)$ at some point in time t is simply the set of sectors occurring in a `blktrace` during a time window of δ seconds *before*[15] the interesting point in time t. Notice that we have a *set* here: it makes no difference how often a particular sector occurs during the time window δ, it just matters *whether* that sector appears or not. Also, it makes no difference whether a particular sector is read or written (or both). Thus the workingset model is a *reduction* of the reality to a handy theoretical model, but a model which is known to *preserve* some relevant and very interesting properties of the reality.

Example: for any two window sizes $\delta_1 < \delta_2$, the condition $WS(t,\delta_1) \leq WS(t,\delta_2)$ holds at any point in time t. In other words: increasing the window size δ will never make the workingset smaller; the workingset can only *grow* if the window becomes larger.

The efficiency of block storage caches can be predicted by the workingset theory in a rather easy and intuitive way. We assume that accesses to the cache are *much faster* than accesses to the background storage, such that we can *neglect* the access times to the cache when compared to accesses to the background storage. Then we can model the following interesting property:

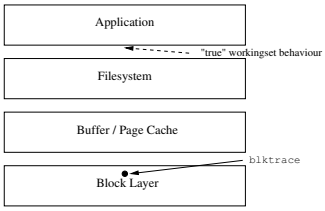
> A cache is called to be designed **efficiently**, iff at any point in time t the following condition holds:
> $|WS(t,\delta)| \leq |\text{cachesize}|$ for some $\delta \geq$ (time to fill the cache once).

The potential painpoint is easy to see: just take δ as the time to fill the cache from the background storage, which is (accesstime to storage) $\star$ (cachesize). If there would exist a point in time t where the workingset $WS(t,\delta)$ would be *larger* than could be transferred to/from the storage during that *same* time window δ, the data transfers could become a *bottleneck* of the system. Vice versa: if the workingset during window δ would always *fit* into the cache, the cache will usually[16] speed up things.

Consequences:

- The **workingset behaviour of the application** is *crucial* for any storage system.
- The above argumentation contains an *oversimplification*: of course, accesses to the cache are not "indefinitly fast" as our above neglect assumed. Therefore, don't take the above inequalities as verbatim inequalities. Multiply some *factor* onto them. In practice, if you really want blastingly fast caches, make sure your cache is at least **one order of magnitude larger** than the workingset size of your application.

In order to do that, you would need the know the workingset size of your *application*. Please keep in mind that this is not the same as the workingset size measured by `blktrace`:

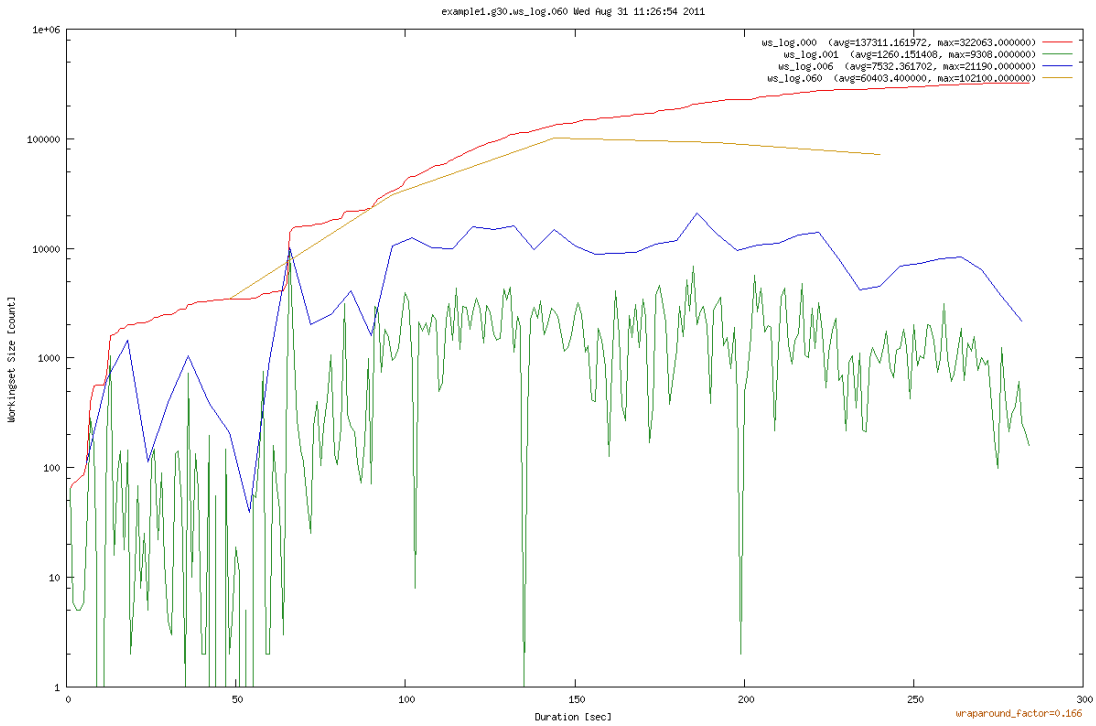


[15] Notice: Dennings original theory used the time interval $[t, t+\delta]$.We find our definition more handy for practical purposes, because we need no "lookahead" into the "future".

[16] Of course, this holds only if the workload contains some *repetitions* during the window δ, and if the cache employs some "good" replacement strategy like LRU. The latter assures that "hot" pages from the workingset will be kept in the cache. In addition, keep in mind that neglecting the access times to the cache could be an *oversimplification* in many cases.

As you can see, the measuring point of **blktrace** sits *below* the buffer / page cache, therefore it does not *directly* measure the application behaviour (in addition to influences from the filesystem, like metadata updates etc). In practice, **blktrace** measurements may differ from the "real" application workload of stateless webserver designs by *several* orders of magnitude. However, *all* modern OSes employ caches. Even if we were able to measure the true application workingset in some way, it would not be relevant for block storage systems, because it would be *impractical* to operate those systems without caches. We need the above picture for understanding the fundamental properties of **blktrace** measurements, and for determining the window size δ. If that is not possible, try to estimate it. As a very rough estimation, take δ as several minutes.

The **blkreplay** suite comes with a small tool to visualize the workingset behaviour as measured by **blktrace**, which is currently the best approximation of the workingset behaviour of the application we can easily get access to. Following is an example, showing different window sizes δ in the same picture (where 000 means cumulation to ∞):



Spend some time on it! Your replay needs to last *vastly longer* than the δ_{old} needed to descibe the steady state of the original buffer/page cache, as well as the δ_{new} to describe the steady state of your replay system. Otherwise, you can get fake results which differ from real practical performance by factors, or even orders of magnitude. As a rough rule of thumb, any replay of natural loads should take at least one hour. Better, make a few measurements lasting 8 hours, or even 24 hours, and check whether the results differ more than expected (besides natural variations).

3.1.4. Pitfalls from Replay Device Sizes and Others

There is a simple intuitive rule: your replay device for **blkreplay** should have (almost) the *same* size as the original device where **blktrace** has taken its measurement from[17].

For easy checking, our output graphics will display the so-called **wraparound_factor** in the lower right corner: it shows the ratio between the max block# occuring in the load, and the size of the replay device. Ideally, it should be near 1.0. When it deviates from this by more than a factor of two (or less than one halve), it is printed in purple color to warn you.

[17] If you don't know this, just make a test run with **blkreplay** and check the output file. At the end, you will find a human-readable statistics showing the highest original block number occurring in the replay, as well as some other interesting numbers.

Some people don't take this seriously, and some don't even believe that this can have a *tremendous* effect.

Practical experience at 1&1 tells that the above rule is *valid*, and that results *will* almost certainly vary. The bias can be *considerable*.

Example: the original `blktrace` recording was taken from a production server equipped with 20 TB RAID. Since in the lab we had only a smaller system with 4 TB, `blktrace` measurements were run despite the smaller size. Whenever `blkreplay` tries to start an IO request outside the LV size, it just *remaps* the sector number modulo the (new) LV size. Therefore, results *appear* to be valid, since you cannot see any "big" holes or anomalies. You will find out the difference only if you compare to the *correct* setup. When repeating the *same* measurements with correct LV sizes of ~20 TB, there is a *significant* difference.

Some people think that such differences can be attributed solely to the natural differences in *spindle count*[18], and therefore it would not hurt if different models of hard disks were used. Although there *are* some effects by spindle count, that opinion is wrong. In order to disprove such a "theory", just build two different RAIDs with same spindle count, but fundamentally[19] different disk drive models (resulting in different total capacity), and compare (under otherwise equal conditions).

In short, any of the following factors can influence the performance, independently from each other (and in no particular order):

- Total capacity, just by itself. If you don't believe it, just create LVs with 1/10 size of the physical storage and compare (on the *same* hardware) with results from the *full* physical storage size. Of course, the original recording must stem from sufficiently large devices, otherwise your "disproof" will be false.
- Model/class of disk drives.
- Number of spindles.
- RAID level.
- Vendor / firmware version of the controller.
- Interconnect technology, such as SATA vs SAS.
- any caches in the hierarchy, such as different BBU cache sizes or different parameters.

... and probably many others.

General rule: *never* expect any of these effects to be linear. Almost always, they are **non-linear**[20]. Anyone claiming something else must **prove** it!

Consequence: anyone violating the above rule produces **invalid** results, unless proven the opposite!

3.1.5. Pitfalls from Parallelism in IO Systems

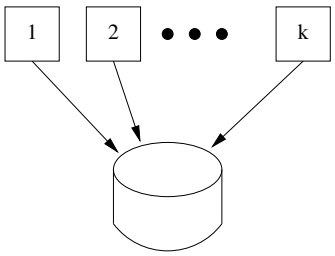
3.1.5.1. Pitfalls from *missing* Parallelism

This pitfall is usually trapping less people, because some intuitive sense for the effects of IO parallelism is more widespread. Even if you are already aware of this pitfall, read on. There are some subtleties.

[18] When using the same hard disk model, a 20 TB RAID must contain 5 times as much spindles as an equivalent 4 TB system. It is clear that additional spindles can benefit random IO. But those effects are **non-linear**!

[19] Of course, there are a lot of disk drives (in the same "class") showing only minor differences. A "fundamental" difference is for example between a cheap SATA disk versus a smaller but faster 15k SAS disk.

[20] Non-linear effects cannot be combined with each other in a predictable way, at least in general.



A frequent use case is *storage consolidation*. Up to k CPU nodes are connected to some "central" storage via some kind of storage network.

You may want to evaluate such an architecture in advance with help of `blkreplay`, in order to avoid failed invests. The `blkreplay` suite will help you, because its supervisor scripts like `tree-replay.sh` are ready to run masses of `blkreplay` instances in parallel to each other, and on different nodes.

For obvious reasons, you should determine the optimum consolidation factor k for each given hardware candidate. If you vary the factor k, you will get at least two effects:

1. With higher k, you need to build up more CPU nodes and more network connections. Otherwise you will neglect effects from IO parallelism (which don't scale linearly[21]), at multiple places of the picture: at potential bottlenecks at the CPU nodes, at potential bottlenecks inside your storage network, and at lots of potential bottlenecks in your central storage system.
   For example, it is completely wrong to just double the load on $k/2$ nodes. Experiences with some test candidates at 1&1 show that results can differ enormously from results from single loads, each on k nodes.

2. The total capacity of the central storage will also vary with k. As known from section 3.1.4, this will not only influence results by itself, but in addition by further effects like spindle count, non-scaling of caches with factor k, and many others.

3.1.5.2. Pitfalls from *too high* IO Parallelism

You can tune the number of `blkreplay` threads via the parameter `threads=...` (see appendix A). As a side effect, this will also influence the maximum number of outstanding IO requests which can be "on the fly" in parallel, together with the replay parallelism (number of `blkreplay` instances running in parallel on the same physical system).

Many people believe that an increasing number of outstanding IO requests will improve overall throughput.

However, some devices / drivers / IO schedulers may respond in some *counter-productive* way when hammered with too many IO requests in parallel. Sometimes, your throughput can even *collapse* to less than 1/10 of the maximum. There are many possible reasons for this unexpected behaviour.

In most cases, you will notice collapsing effects only during **overload situations**. During normal operation, you will not notice anything, because the device can *catch up* with the IO demands. There are almost no queues, because the *average service time* is shorter than the average request rate.

As soon as the device gets overloaded, the situation will change rapidly. Masses of `blkreplay` threads are trying to fire off their requests in time, starting to overlap now because the service time increases. Suddenly, some queuing will take place, somewhere. In mathematical theory, the queue lengths could even grow indefinitely whenever the average request rate is higher than the average service time. In practice, there will be some limits somewhere.

Imagine you are going shopping in a supermarket. When the cashier girl cannot catch up with the demand from the customers, their shopping trolleys will start to form a queue in front of her. Now assume the following behaviour: the longer the queue, the *slower* she will work. Imagine that! After a while, customer satisfaction will go down to zero, because the queue will get longer and longer. And the longer, the slower she will work. And so on. There is no escape, other than stopping to buy anything from there.

[21] There is no general way to predict the behaviour of an unknown system! Many non-linear effects show some kind of "binary" behaviour, suddenly collapsing at some point where you didn't expect it.

A similar behaviour can be observed in some IO systems. Once a queue has formed, there is almost no escape from a behaviour similar to **traffic jam**. If you cannot change the system, your only chance is to reduce load, or even to remove the load at all.

Sometimes it is even hard to trigger such behaviour. A systems appears to run smoothly for months, but suddenly it collapses.

If you have such a system (or cannot be sure to have one[22]) and want to simulate its behaviour in the laboratory, you should be carefully tuning the `threads` parameter. In reality, the number of outstanding requests is often limited in some way. For example, a typical workstation load caused by a single user has often some intrinsically limited IO parallelism, similar to a limited number of shopping trolleys in the supermarket. On the other hand, some server loads (such as those caused by Apache) can `fork()` off a high number of threads.

So it is quite possible to observe some traffic jam behaviour in practice, depending on your application. It is quite possible that such behaviour is **relevant** for IT operations, at least for *some* applications (but not for all).

Taking any of these **non-linear** (and sometimes even *binary collapsing*) effects not seriously and/or using a wrong number of threads / replay parallelism can easily lead to **invalid results**, and in turn to failed invests!

Hint: a frequent case are distributed systems by *themselves*. They tend to produce **queues** at times and in places where you don't expect them, and they tend to produce avalanche-like negative effects (self-amplifying), similar to suddenly appearing **traffic jams** where you cannot determine one single reason in isolation.

3.2. Recommended Setup and Usage

3.2.1. Planning Phase

Never try to plan a project without deep knowledge of the pitfalls described in section 3.1. In addition, some experience with `blkreplay` is helpful. In order to gain such experience, consider a *test project* just for playing around, and for getting familiar with the pitfalls.

3.2.1.1. Describe the Scope of Project

Before starting, you should get conscious with yourself. *What exactly* is the question you want to answer with help of `blkreplay`?

Write down the question both as shortly, as well as precisely as possible. Here are some examples:

- Compare hardware vendor `A` with `B` and `C` for my production workload X.
- Compare hardware vendor `A` with `B` and `C` in general.
- Debug kernel module `xxx`.
- Compare iSCSI with Fibrechannel for my production workload X.
- ...

Next, write down a precise description of your intended **test environment**. Best practice is to name hardware vendors, models, all components (including intermediate gear like network switches), and so on.

Last step: describe all the parameters which know of, which *could* have an influence onto your test results. Example:

[22] To find out, we recommend the artifical loads `*bursts*.load.gz` for creating anal kinds of overload.

Parameter	Varying?
16 GB RAM in storage node	no
10GBit vs 1 GBit network speed	yes
Cheap 2TB SATA vs expensive 600GB raptor disk (model "tyrannosaurus rex")	yes
RAID-Level $\in \{1, 5, 6\}$	yes
RAID Stripesize $\in \{16, 32, 64, 128\}$kB	yes
...	...

3.2.1.2. Describe the Setup of your Experiment

In many cases, the parameters described in your table will make up a multi-dimensional problem space (cartesian product) which is too large to be explored exhaustively. As explained in section 3.1.4, many of them will influence your results *non-linearly*.

Thus, you will have to consider the following general strategies:

- You may fix some of the parameters to particular values. Although this saves time, you may miss an opportunity to find an optimal solution.
- You may select some/enough random samples from your multi-dimensional problem space and try them randomly (Monte Carlo methods).
- Stepwise refinement: explore the multidimensional space by varying exactly *one* paramenter at once. This is slow, but you can be sure of the effects caused by this.

3.2.1.3. Select `blkreplay` Load

Depending on your project, you should consider both artificial and natural loads, but not too many of them. Usually, more than three loads are impractical for an ambitious project (unless you want to compare masses of loads on the same reference hardware).

If your project tries to answer a question for some specific workload X, you should just record that workload if you can do so (see chapter 4).

If you cannot obtain your real workload in advance, you have to select one from the `blkreplay` project (or other sources) which comes as close as possible to your (intended) natural workload.

For the sake of static comparison of workloads, `cd` to a directory containing your `*.load.gz` files and issue the following command:

```
/path/to/graph.sh --static myname.load.gz
```

This will produce some `.png` graphics, describing the throughput and workingset behaviour of your load (cf section 3.1.3).

You need enough space in `/tmp/` (or in another `$TMPDIR`) for temporary intermediate files. If your `*.load.gz` file is very large (several hours or even days), you may need several gigabytes. Please don't interrupt `graph.sh` as it spawns lots of subprocesses and creates lots of temporary files. Currently, there are no checks for free space in `/tmp/`, so running out of space may produce wrong results *silently*. As a countermeasure, run `watch df /tmp/` in a separate window during your run of `graph.sh`.

Hint: some `blktrace` recordings (but not all) contain some timing information about the original IO latencies as measured at the original site. Use `/path/to/graph.sh --dynamic myname.load.gz` to create some additional graphics about them.

3.2.1.4. Selection of Replay Duration

This is a hairy problem, as already described in section 3.1.3. Often, you cannot run 24h replays for many hundred times.

If you want to be sure that a particular load will run even under *worst-case* conditions, you should definitly select some appropriate time window around *load peaks*, measured both in throughput as well as in workingset size.

In addition, you can try the following strategies:

- For explorative phases in your project, such as determining the optimum in your parameter space, you can try to minimize running times as much as possible. But not too much. Otherwise you will be caught by the pitfalls described in section 3.1.

- For final verification of your results, you should repeat benchmarks with a longer window (at least 8h or 24h).

3.2.1.5. Total Project Time

Working with huge parameter spaces is not all you have to consider. Setup of different RAID levels, re-initialization after changes of stripe sizes, filling LVs with random data, etc, may take a very long time, in addition to the benchmark themselves. Don't forget that! Your only chance are nights and weekends, if you manage to run something unattendedly. But predictions are sometimes wrong. In addition, something may fail and then needs to be restarted. Calculate some spare time for that!

If there is a high time pressure in the project timeline, you probably will have to *rework* some parts of your project plan.

Planning is crucial! When you find any discrepancies, try to re-think your plan as a whole, not just some parts of it.

3.2.2. Setup Phase

3.2.2.1. Lab setup

Ensure that all your hard- and software components are ready in the lab and operational.

In addition, you need some workstation (or server) where the `blkreplay` suite is checked out. Do the following steps:

- Ensure that `gcc`, `make`, `gnuplot`, and some standard tools like `grep` / `gawk` are installed. If you need multiple machine architectures (such as `x86_64` and `i386`) in parallel, ensure that `gcc` can cross-compile via flags `-m32` / `-m64` and that the appropriate libs are installed.

- `git clone https://github.com/schoebel/blkreplay`

- `cd blkreplay`

- `./configure`

- `make`

- Attention! `make install` is not yet supported. Just leave everything in place. You can either put `/path/to/blkreplay/scripts/` into your `$PATH`, or call the scripts via hard path.

Ensure that all your test machines are reachable as `root` via `ssh` from your central workplace, without need for any password prompt. In order to achieve that, you should consider `ssh-agent`, in addition so some tweaking of `/etc/ssh/ssh_config` (and probably `/etc/ssh/sshd_config` on each of the target hosts).

On your workstation, you should have enough disk space to store your results. Create a subdirectory there for your project. Copy `/path/to/blkreplay/example-run/default-main.conf` (and possibly other `*.conf` files) to that new subdirectory, and finally `cd` to it. For the rest of your life, you will be working there ;)

You can now either call `/path/to/blkreplay/scripts/`*`something`*`.sh` as hard paths as indicated in the following examples, or you may put `/path/to/blkreplay/scripts/` into your `$PATH`.

Customization of `default-main.conf` is described in the following.

3.2.2.2. Configuration Files

You should edit `default-main.conf` to reflect the default setup for your project. If you want to run multiple variants of your default setup, you can do so by creating additional files like *something*`.conf` as well as a subdirectory *something*`/` (having the same name without suffix `.conf`). When you later start your benchmark, the values from *something*`.conf` will override those from `default-main.conf`. It is highly recommended to override only *one* parameter inside *something*`.conf`, otherwise it may become difficult to reveal the real impact of changed parameters onto your test candidate.

In general, you may override *any* parameter from `default-main.conf`, even hostnames, or input files `*.load.gz`, or whatever.

3.2.2.3. Meaning of the Config File Parameters

The meaning of the parameters is documented in the following places:

1. Comments inside `default-main.conf` should provide enough information for experienced administrators, at least for a quick start, and should guide you through the most basic steps.

2. The same information is available in appendix A.

3. Last but not least: read the sources, if you are in doubt about anything.

3.2.3. Benchmark Phase

The basic idea is simple: after customization of `default-main.conf` (and probably other `default-*.conf` files when using additional modules), you create a new subdirectory for each benchmark run.

Whenever you call `/path/to/blkreplay/scripts/tree-replay.sh` (without parameters), a whole bundle of benchmarks will be started, one for each *leaf*[23] subdirectory (starting from `cwd`), provided that for each (intermedidate) subdirectory name *xxx* there exists some *xxx*`.conf` in the current working directory or in one of its parents. In the whole subdirectory structure, any directory ending with `.old`, or including the substring `ignore`, or containg a file `skip` will be ignored.

Example: you have created a subdirectory `./short/` as well as two nested subdirectories `./short/model1/` and `./short/model2/`. You further have prepared the config files `short.conf`, `model1.conf` and `model2.conf`, existing in the current working directory or in some parent directory down the `../` chain. Then exactly *two* benchmark runs will be started, namely in `./short/model1/` and in `./short/model2/`. The benchmark running in `./short/model1/` will include the following `*.conf` files, in the following order: `default-main.conf`, `short.conf`, and finally `model1.conf`. Each of the specialized config files may override any previous setting, but it is highly recommended to change only one parameter at a time and to use short but expressive names. Notice: the intermediate directory `./short/` is no leaf (since it contains some subdirectories), therefore no benchmark will be started inside it. Later, you just need to create `./long/` as well as `./long/model1/` and `./long/model2/` and some `long.conf` in order to repeat the same benchmarks with a longer `replay_duration` setting.

Hint: using "intuitive" names like `short` and `long` bears some danger. A few years later, you will not remember what they exactly have meant. Looking into `*.conf` will not help other people, for example if you publish your benchmark results somewhere. Therefore it may be wise to use "speaking" names like `duration_600`, at least if you have more than two variants. On the other hand, "intuitive" names are better for presentation to some less-deeply involved audience. Take some time for creating well-designed names for `*.conf` and your directory hierarchy! Changing that names later is cumbersome. Better to design your names in advance in a systematic (but simple) way.

[23] A leaf has no further subtree inside it.

On large investigation projects, deeply nested structures may be necessary, involving different loads, different hardware, different hardware setup, etc. Not all of them are currently automated. You can use the generic module mechanism to extend the default scripts with further functionality, to push automation further.

However, not all setup tasks *can* be automated at all. Some of them like forcing physical RAID degradation must be started by physically removing a disk, which cannot be automated (other than buying extremely expensive robots). Therefore, you may include some human-readable dialogs inside your `*.conf` files (in shell script syntax), or in some new modules you have written. In any case, it is advisable to write some script code to *check some preconditions* (such as RAID status) in order to prevent wrong measurements.

In general, `tree-replay.sh` will never repeat any benchmark which has already completed (i.e. there exists an output file `*.replay.gz` in that leaf directory). This allows an incremental style of working.

Continued example: normally, you can call `tree-replay.sh` again, but nothing happens, because all leaf directories already contain some results. However, then you find a problem with the results in `./short/model1/`, so you want to repeat that benchmark without deleting your first (questionable) results. Now you create a subdirectory `./short/model1/try1.old/` and move your results there. As said above, any directory names ending with `.old` or containing the substring `ignore` are ignored, so the directory `./short/model1/` will continue to count as a leaf (despite its newer subdirectory, which is just ignored). Since the `*.replay.gz` files are now missing in `./short/model1/` due to the `mv`, `tree-replay.sh` will repeat that single benchmark there.

Here are some useful hints:

- You may skip any directory by creating a file `skip` inside it. `touch ./short/model1/skip` will disable that directory. Later, you can remove that file in order to fire off that benchmark.

- `skip` files are also working in intermediate directories like `./short/`, disabling the whole subtree in one step.

- Design your `*.conf` files such that arbitrary combinations are *possible* (cartesian product). In contrast, your directory hierarchy need not (and, in many cases, *will not*) exploit the *full* cartesian product.

- You may create a new leaf directory (somewhere in the subtree) even in parallel to an already running benchmark. Whenever the currently running benchmark has completed, `tree-replay.sh` will re-scan the subdirectory structure, find any freshly created leaf directories, and determine which benchmark to start next. All leaf directory names which have not yet completed are sorted alphabetically, and the first name according to ASCII sort order is taken first.

- When 10 or more variants could appear somewhere (even after a while), use leading zeros in any names like `v001`, `v002` etc to ensure that ASCII sort order is the same as numerical order.

- Sometimes the ASCII sort order of names like `short` vs `long` is boring, because you want to run the `short` benchmarks first. As a workaround in larger projects, add some numerical prefixes like in `01_short` vs `09_long` (leaving some numerical space such that you can later add `05_medium`). As a side effect, this also improves the ASCII sort order of your later `*.png` graphics.

- You can rearrange the order in another way: just create an empty file `prio-nnn` in a leaf directory, where `nnn` is a number denoting a priority class. Priority classes are overriding the global ASCII sort order. Directories having the lowest priority class are run first, while directories without any class are run last. Inside of each class, the ASCII sort order is obeyed.

- When you design your *.conf files systematically as a cartesian product, in theory it makes no difference whether you permute some directory components (e.g ./model1/short/ instead of ./short/model1/). However, in practice it influences the ASCII sort order (taking the *full* path) and therefore the order in which your benchmarks are run.

- When some benchmark fails, just delete the corresponding output files. On the next cycle, tree-replay.sh will detect the missing files and just restart that benchmark (possibly among others).

- ***xxx*.conf** files can even reside in some parent directory of the current working directory. This way, you need not copy your .conf files inside a complex directory hierarchy (even spanning multiple projects). However, only the ***xxx*.conf** files corresponding to directories reachable from the current working directory will be included. Example: if you go to a leaf of your subtree and start tree-replay.sh there, no *.conf file other than default-*.conf will be included. This may produce different results than expected. Make sure you start your replays always in the same base directory!

It is easy to misconfigure almost anything by accident. Check each step you make. In particular, run tools like top, xosview, iostat, watch df /tmp/ etc on *all(!)* your involved machines in order to get a chance for noticing when anything goes wrong! *Never*, really *never* run tree-replay.sh **blindly**!

3.2.4. Visualization of Results

You need enough space in /tmp/ (or in another $TMPDIR) for temporary intermediate files. If your blkreplay run was very long (several hours or even days), or if your replay had a high degree of parallelism, you may need several gigabytes, in extreme cases even several hundreds of gigabytes (as well as rather long running times – please don't interrupt graph.sh as it spawns lots of subprocesses and creates lots of temporary files). Currently, there are no checks for free space in /tmp/, so running out of space may produce wrong results *silently*. As a countermeasure, run watch df /tmp/ in a separate window during your run of graph.sh.

If you have enabled the module graph in the config file default-graph.conf, the following steps will be carried out automatically for you. Alternatively or in addition, the script tree-graph.sh can be used to (re-)create all *.png graphics in a whole directory hierarchy, analogously to tree-replay.sh (and even sourcing the same *.conf files in the same way). In case you need some individual graphics, read on for details.

After a run of tree-replay.sh, cd to one of the subdirectories where your result files *.replay.gz have been produced. There should be as many *.replay.gz files as there was replay parallelism (on multiple devices in parallel). Check that. In addition, check that no errors are inside them, for example by typing:

```
zgrep ERROR *.replay.gz
```

If all is right, issue the following command:

```
/path/to/graph.sh *.replay.gz
```

This will produce *.png files, which you can inspect with any graphical viewer like eog / konqueror etc, print via lpr, or even work on with graphical editors like gimp.

All files *.replay.gz will be taken together, to form a single result from contemporary replays on several devices in parallel. This means: where possible, results from multiple replay devices will be merged together into a single graphics.

Hint: If you want to view only a single particular device (or zoom into it), just call /path/to/graph.sh with a single argument.

As output, multiple kinds of graphics are produced. Each one starts with the same prefix, but has another suffix. For example, `yourname.g01.latency.realtime.png` is a graphics file showing the measured latencies in realtime. The numbering part `.g01.` etc is for sorting in the shell, such that the "most interesting[24]" graphics will come first.

By default, parameters matching `*.load.gz` will produce graphics containing only *static* analyses. When matching `*.replay.gz`, only *dynamic* ones are produced. In order additionally switch on any (or both), just add one of the options `--static` or `--dynamic`.

There is one exception: the throughput graphics are produced always, independently from `--static` or `--dynamic`. In the following, the meaning of different suffixes occurring in the output filenames is described:

`*.thrp.*.png`
: The throughput is displayed on the y axis. There is an overview variant, where the overall requested throughput is depicted green, while the actual throughput (when running in dynamic mode) is orange, for comparison. When both are exactly identical (which occurs for example when you have an extremely fast disk or a very low-demanding load), don't be confused when the orange line completely covers the green one (a property of `gnuplot`). There are also more detailed variants where the read vs write throughput is distinct.

3.2.4.1. Static Analysis (`--static`)

`*.ws_log.*.png`
: On the y axis, the workingset size (cf section 3.1.3) is displayed in logarithmic scale. Multiple window sizes δ are displayed together in one graphics: `001` means $\delta = 1s$, `006` means 6 seconds, `060` means 60 seconds, and `000` means $\delta = \infty$. The latter is nothing but cumulation of *all* occurring sector numbers into one set[25].

`*.ws_lin.*.png`
: Like `ws_log`, but the y axis is in linear scale. May be useful for detecting more fine-grained behaviour in peaks.

`*.sum_dist.*.png`
: On the y axis, the *distance* between the lowest and the highest sector number occurring in each workingset window is displayed. As a result, we can see something like a "total seek distance", as if a disk elevator strategy *would* sort all requests inside a workingset window according to sector numbers, in order to optimize throughput in favour of latency. For experts, this can reveal some interesting internal property of the workload.

`*.avg_dist.*.png`
: Same as `sum_dist`, but the average distance is displayed (normalization against the workingset size). This results in something like an idealized "average seek distance" during each time window.

`*.rqsize.*.png`
: Displays a histogram of the request sizes (#sectors) occurring in the load, in units of sectors.

`*.rqpos.*.png`
: Displays a histogram of the request position (sector#) occurring in the load, in coarse units of whole GiB. The coarse units are necessary to avoid too much jitter in the y axis.

`*.turns.*.png` (both static and dynamic variants)
: A turn is a request where the sector number is *lower* than the sector number of its predecessor. For a mechanical hard disk, a turn means that it has to "seek backwards", which is often more time-consuming than seeking forward. Imagine an old-fashioned magnetic tape (e.g. DAT), then you will get some feeling for the meaning of a turn. Turns are an interesting measure for the locality behaviour of the load (or the replay result). There is no freak that some IO schedulers try to minimize the number of turns (for example the famous sawtooth strategy). In this graphics, the y axis shows the *relative*

[24] For many people; of course, there may be different needs. Feel free to rename your result files as you like.
[25] The slope of the `000` line is an indicator for the "repetitiveness" of the workload.

number of turns occurring during a time window of 1 seconds (in percent, related to the total number of requests occurring during that time window).

3.2.4.2. Dynamic Analysis (--dynamic)

`*.latency.*.png`
: On the y axis, latencies are displayed.

`*.delay.*.png`
: On the y axis, the delays between the intended starting time and the real starting time are displayed.

`*.thrp.*.png`
: On the y axis, the troughput (IOPS) is displayed.

`*.flying.png`
: The y axis shows the total number of requests which are currently "on the fly".
Examples: `*.latency.flying.png` is derived from the latencies, and thus showing the number of requests currently submitted to the device, while `*.delay.flying.png` is derived from the delay, showing the number of requests currently waiting in the logical "delay queue".

`*.smooth.*`
: This internal infix indicates that the y axis has been smoothed, in order to be able to see anything in wildly jiggling data. Example: `*.smooth.latency.flying.png`.

`*.realtime.png`
: The x axis is ordered according to the real starting timestamps which have actually occurred during replay, possibly containing any delays. Notice: when multiple concurrent `*.replay.gz` have been supplied as an argument, they are *merged* (since their timestamps are usually from the same range), similar to the effect of overlay slides.

`*.setpoint.png`
: The x axis is ordered according to the *intended* request submission time (starting point), i.e. when the request *should have* started (ignoring any delays). As before, multiple `*.replay.gz` are merged.

`*.completed.png`
: The x axis is ordered by the completion time of the requests. Note that this may differ from the submission time.

`*.points.png`
: The x axis is ordered by *requests*, not timestamps. As a result, requests on the x axis are *equidistant*, even in case of heavy throughput differences. For `*.replay.gz`, the order is the *completion* order. This graphics is useful as a kind of "looking glass", to reveal more details from performance hot spots. Notice: multiple concurrent replays are *not* merged on the x axis (as is the case with `*.realtime.png`, `*.setpoint.png` and `*.completed.png`). Instead, the runs are just concatenated (pasted together sequentially) in the same order as in the corresponding files.

`*.bins.png`
: The new y axis is now a *histogram* showing absolute frequency occurring at the former y axis, while the new x axis now carries the role of the former y axis.
Examples: `*.latency.bins.png` shows the latencies on the x axis (while the former `*.latency.realtime.png` had it on the y axis), while the new y axis now shows the absolute frequency of that latency (how often that latency occurs, independently from the former replay timestamps).

`*.xy.png`
: The x axis as well as the y axis are displaying non-timely data. This is useful for visualization of *corrolations*. The following variants exist:

`*.latency.xy.png`
: While latencies are displayed at the y axis, the x axis inidicates the number of flying requests (request queue length). Thus you can see the corrolation between request queue length and service time.

`*.delay.xy.png`
: Similarly, but shows the corrolation between request queue length and delays (the time needed to *enter* the request queue at all).

`*.latency.delay.xy.png`
: Immediate corrolation between latency and delay.

Hint: the internal data format of `*.replay.gz` is the same as `*.load.gz`. Thus you can use `/path/to/graph.sh --static *.replay.gz` to additionally create the same static analysis graphics as described in section 3.2.1.3. In difference to analysis of `*.load.gz`, this time you may have selected a different time window, and you may have merged multiple replays together.

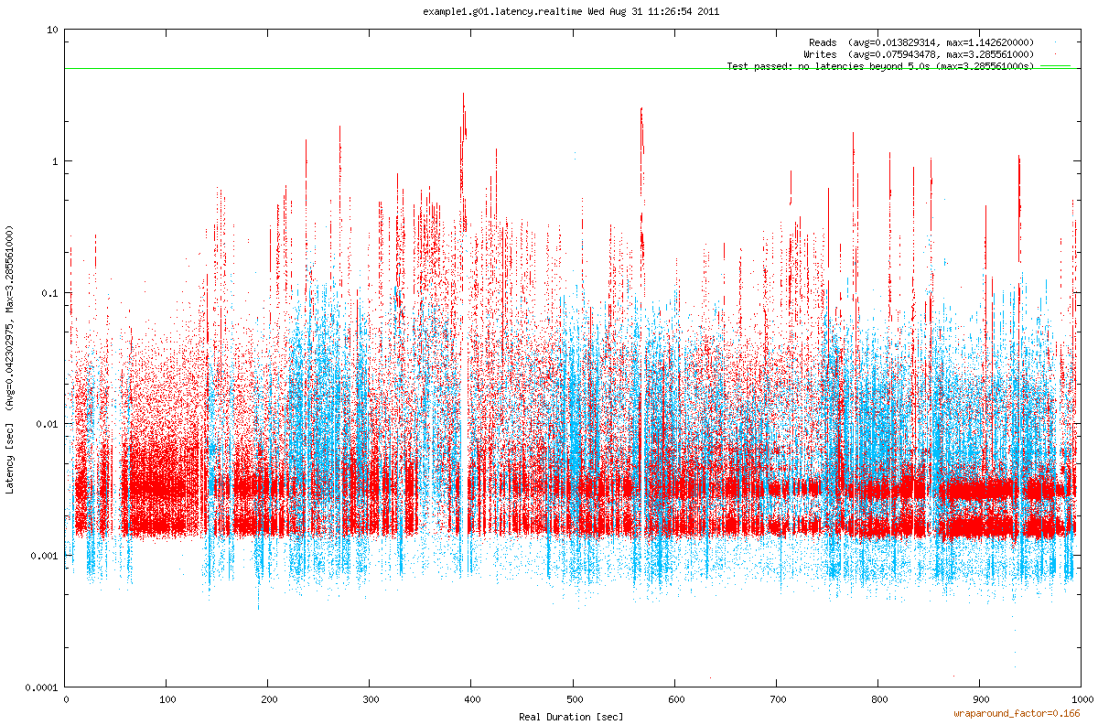
3.3. Human Interpretation of Results

Many existing benchmark tools try to deliver a single number as a result, such as x `hyper-ops` or x `hell-stones`. Such numbers cannot describe the reality, which has multiple dimensions and is much more complex. Sometimes, new dimensions are even discovered to have an influence.

By default, `blkreplay` does not produce any (single) number, but graphics, showing the behaviour over time (or in other dimensions). Although some of those graphics like `*thrp*` could be used for production of numbers, most of them serve as input to the human brain: the human "neural network" can be *trained* to detect **hidden properties** in your measurement data, which cannot be (easily) detected by current AI technology.

3.3.1. Sonar Diagrams

The following example shows the replay latencies of a natural load over real time. Note that the y axis is logarithmic scale (otherwise you would'nt see too much in the lower bands):



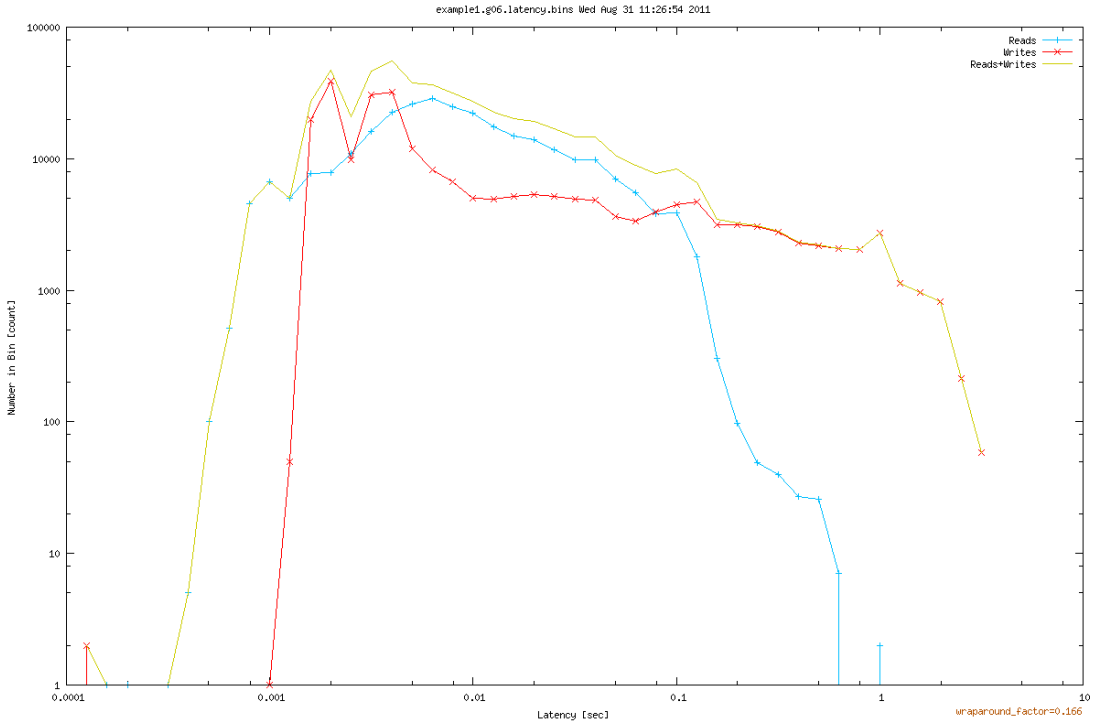
Reads are depicted as blue dots, while writes are colored red. In case of `--with-partial`, pushed-back requests are displayed additionally[26] in purple. Together, they form some kind of

[26] Notice that the set of purple "pushes" is always a *subset* of the "writes". They are only visible because they are printed later (hiding the previous red dots).

"clouds", showing some "density" both in time scale, as well as in the "latency scale".
It requires some experience to interpret sonar diagrams.

- As you can see, there are two main **"bands"** optically visible to the human interpretation. These look similar to "fish swarms" occuring in sonar diagrams; hence the name "sonar diagram".
- Most obviously, **cache hierarchies** can be identified visually, by looking at the "bands". In addition, the natural fluctuation of the load can be seen. Even better, it is quite interesting how the test candidate is responding to the fluctuation, even over time.
- Any differences between read and write behaviour will immediately catch your eye.
- The red "needles" are indications of queueing. In difference to a single "hang" which would just "fly above" other requests, intermediate latencies from "fellow requests" are also present, spanning almost the full range from "fast" requests to the queued ones.
- In case of the strongest needle, you can even see a "lash" behind it. There are no "faster" requests (not even reads) during this lash (i.e. the space below it appears empty - even as if there were an interruption in the time axis). This is an indication of a *global* queuing strategy.

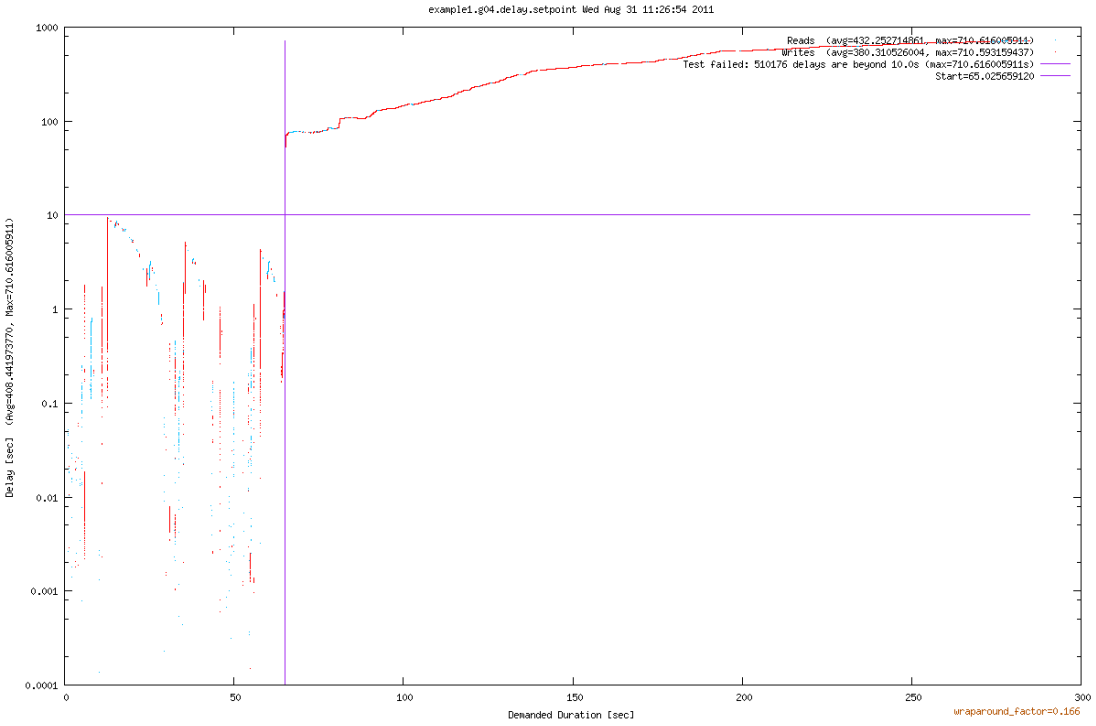
The same latencies are now displayed as a histogram. A histogram is a non-timely survey of the absolute frequencies occurring in the previous graphics.



You can identify the "bands" a second time by looking at the camel-like "humps". In addition to the sonar diagram, you can identify the "hight" and "thickness" of a hump more easily. Some very small additional humps can also be found, probably an indicator for further caches or probably for some scheduler timeouts.

3.3.2. Delay Diagrams

As already explained, delays are the difference between the *intended/demanded* starting time of an IO requests, and its *real* starting time. Huge delays are almost always an indication that the system cannot "catch up" with the application demands. Here is an example:

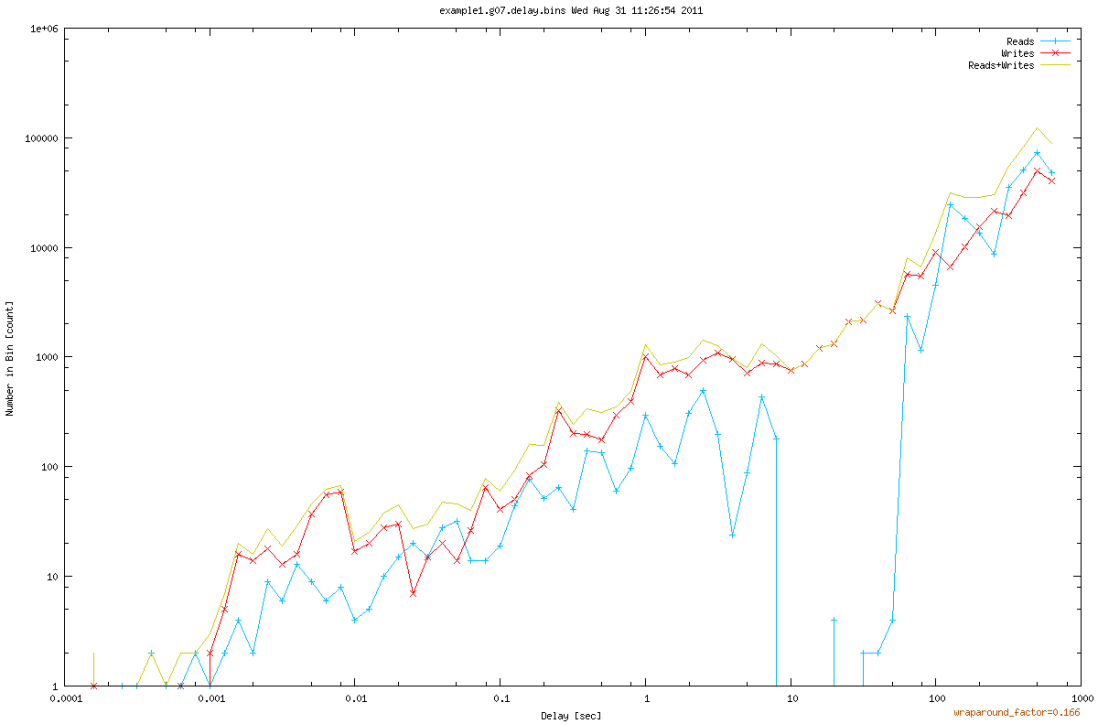


Here are some interpretations:

- You will immediately notice the "runaway tail" after the purple indicator line, which is just a strong indication that overall throughput is too low.

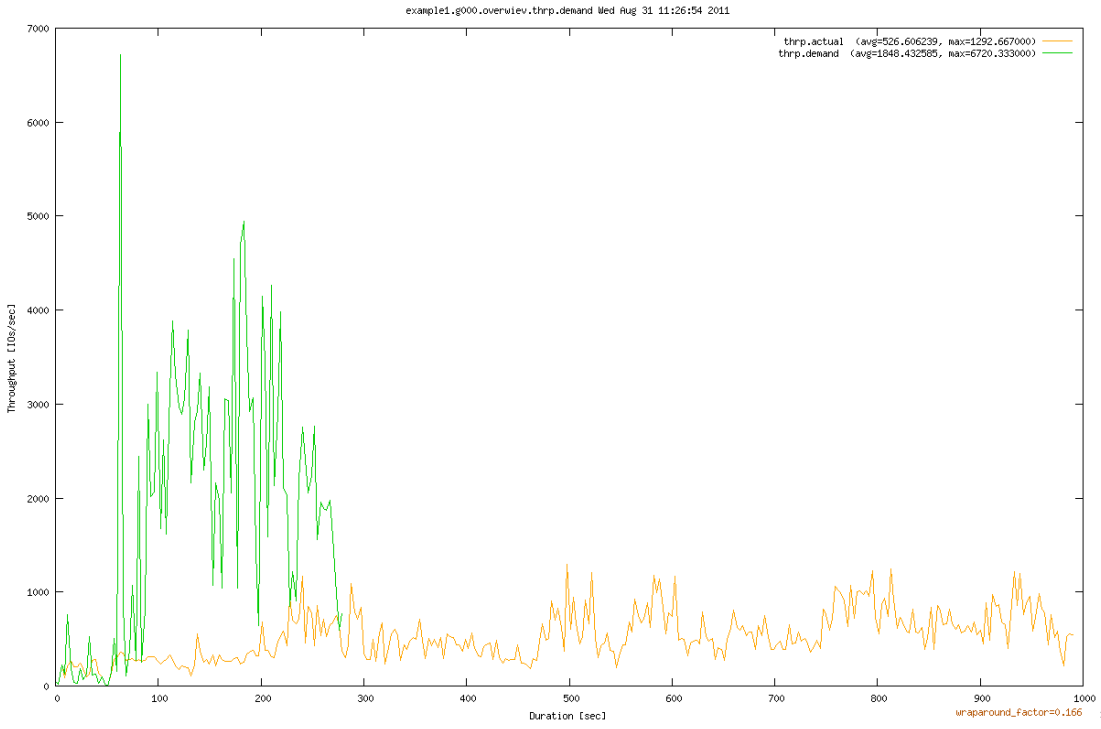
- Even before, you can see some "needles" with a "lash" behind. Another indication, even occurring in a section with rather "low" throughput demands.

As before, there is also a histogram variant of delays:



3.3.3. Throughput Diagrams

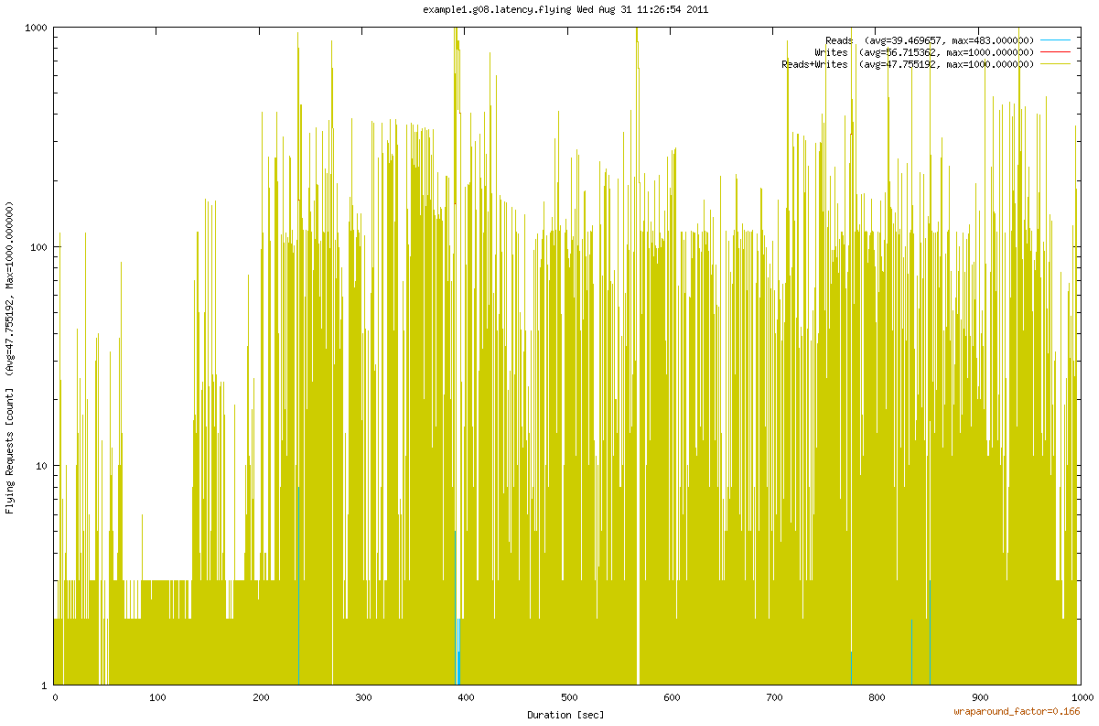
In order to compare this with the actual throughput, look at the following picture, where demanded throughput is colored green, while actually delivered throughput is orange:



This picture is a strong indication that single IOPS numbers are not sufficient for really describing the actual IO throughput behaviour of a system. Our timely plot can tell you a lot more about the system. More examples can be found in section 5.1.2.

3.3.4. Flying Diagrams

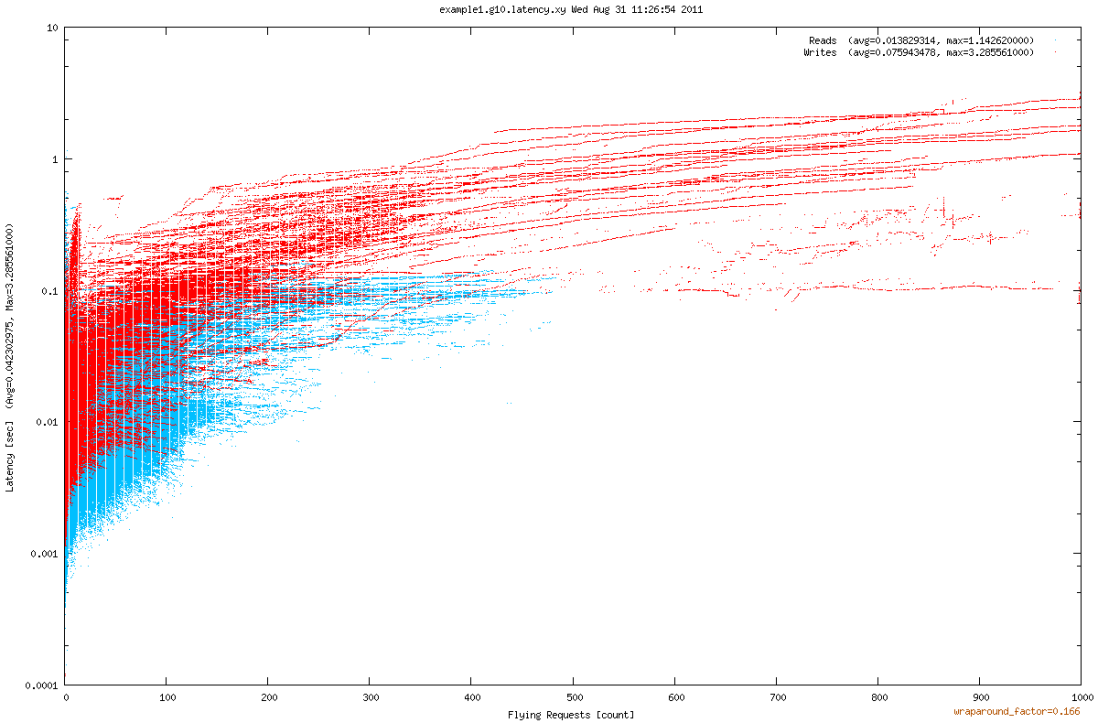
These are useful to inspect the actual request queue length over time. The following graphics exhibits the typical "jitter" of the queue length during replay of natural loads having a lot of conflicts, and using `--with-ordering`:



Notice: this is only the view of `blkreplay`. The physical device may be actually loaded with *less* requests in parallel, because intermediate layers (such as drivers, iSCSI, etc) may **limit** the IO parallelism. When `blkreplay` fires off a request, it need not hit the device immediately. There are a lot of opportunities to be queued for some time in another places.

3.3.5. Corrolation Diagrams

Most of them are for advanced users and experts. They can visualize internal relationships between interesting quantities, such as observed request queue length versus observed latency:



Rough interpretation: under overload, write requests cannot be started in time, and when they are started, they will take a long time. Read requests are seemingly behaving better.

3.4. Advanced Features

3.4.1. Modules

The script `tree-replay.sh` can be extended by plugins, aka modules. Some modules are delivered with the `blkreplay` suite, but you can add your own ones (and you should consider submitting them to the opensource project `blkreplay` at `github.com` if they could be useful for a broader audience).

Here is a list of already existing modules:

`iscsi_target_iet` When enabled, your `iet` iSCSI target will be automatically configured for you, automatically generating iSCSI `IQN` names for you. Usually this module is used in combination with `iscsi_initiator`.

`iscsi_initiator` When enabled, your `blkreplay` runs will automatically act on iSCSI `IQN` identifiers instead of on device names, transparently creating iSCSI connections for you. It can be used standalone, for example to connect to commercial storage boxes. When combined with module `iscsi_target_*`, even the `IQN` name generation will be automated transparently (in this case you just supply the device names present at the `iscsi_target` host).

`recreate_lvm` You just supply some physical devices (PVs) in `$replay_device_list`, and this module will transparently (re-)create an LVM volume group (VG) for you, together with some configurable number of logical volumes (LVs) inside it, optionally even with LVM striping.

`create_lv` Here you can put in your code for re-creating logical volumes on an arbitrary storage manager. You **need** this **absolutely** whenever you are benchmarking virtualized storage; see section 3.1.1.

`scheduler` By default (when enabled), the IO scheduler will be set to `noop` (or another value) on all relevant target machines.

`wipe` When enabled, all your devices will be filled with random data before the benchmark starts. **Always** use this when benchmarking virtualized storage; see section 3.1.1.

`bbu_megaraid` When enabled, your LSI megaraid conrollers will be programmed to enable/disable BBU write caching. The tool `MegaCli` must be installed.

`graph` By default (when enabled), the script `graph.sh` will be called automatically for you at the end of each `blkreplay` run.

`pipe_*` When enabled, these filtering modules modify the `*.load.gz` input at runtime before it reaches `blkreplay`. Following pipe filters are available:

- `pipe_repeat` Repeat the same input files forever, or until `$replay_duration` is reached. This is useful if your `*.load.gz` is too short for a longer replay. However keep in mind that the *workingset* will not increase, because exactly the *same* IO operations will be repeated after a while. In order to circumvent that, combine this module with `pipe_slip`.
- `pipe_slip` Add some offset to the sector# after a bunch of operations. Use this in combination with `pipe_repeat` to produce a "slowly moving" behaviour of the sector numbers, such that the next cycle will repeat the operations at a slightly higher position. Details see appendix A.3.2.
- `pipe_subst` Replace `R`[reads] with `W`[writes] or vice versa.
- `pipe_spread` Increase the region where the sector numbers will run about (corresponding to the size of a LV) by some factor. Details see appendix A.3.4.
- `pipe_resize` Increase the request size (#sectors) by some factor, or clip it to some bounds. Details see appendix A.3.5.
- `pipe_cmd` Insert arbitrary shell commands into the input pipeline.

Detailed documentation of the module parameters can be found in `default-*.conf`, or in appendix A.2.

In order to write your own module, just look at the existing ones and take them as a kind of "template". Modules are simply `/bin/bash` code residing in `scripts/modules/` and starting with a number[27], followed by an underscore, and finally *`modulename`*`.sh`. Any existing module is automatically `sourced` by `tree-replay.sh`, provided that some config file `default-`*`modulename`*`.conf` exists in your working directory (or in any parent directory down the `../` path).

Modules are sourced only once at the start of `tree-replay.sh`. Thus they should only define some shell `function`s and change some list variables. The shell functions will be called later for each leaf directory. The names of your functions should obey the following conventions:

`modulename``_prepare()` is called during the preparation phase. Typically, you can set some variables here, initialize some controllers, setup your network, create some iSCSI connections, or the like.

`modulename``_setup()` The setup phase starts after *all* prepare functions of all modules have been executed. For example, some device setup operations are only possible after some iSCSI connections have been already established. Therefore, the plugin methodology offers you a separate phase in order to guarantee some overall order on operations even from unknown foreign modules.

`modulename``_run()` Typically, you won't do anything here, because the running phase is usually the task of the `main` module. Only in very special cases, you would need to hook yourself into this phase.

`modulename``_cleanup()` Typically, you will de-configure something here, remove some temporary files, check your results, create some statistics, etc.

`modulename``_finish()` Typically, network connections etc will be shutdown only here, because some operations from the cleanup phase could be relying on them being active yet.

For each phase, there exists a variable *`phasename`*`_list` determining the order in which the functions will be called. In most cases, you will just *append* your function name to that list. In some cases, you may want to *pre*pend instead. Look into the existing code of small modules like `graph` to get an impression, and read the `main()` function in the `main` module.

3.5. Lowlevel Details and Expert Usage

Ordinary users should skip this section.

3.5.1. Internal Overhead

Some new SSDs are promising IO rates of more than 100.000 IOPS when operated over extremely fast IO channels. When you try to achieve such high rates, you may probably stumble over several difficulties.

For example, running too much threads / processes on the same file handle or on the same pipe can lead to serious lock contention in the Linux kernel, limiting overall throughput. Although some people might argue that `blkreplay` itself could produce an "artificial" bottleneck, please keep in mind that you need some *real applications* which were capable of producing / consuming such extremely high IO traffic. Real applications usually don't use AIO, where such problems occur less frequently. No wonder that benchmark results published on the internet are often based on AIO, which is far from practice.

Anyway, here is some advice how to deal with extremely high IO traffic. For "normal" IO rates in the region of a few 10.000, the following is usually not needed. Otherwise, you should be prepared to look into internals of the kernel, compile your own high-speed custom kernels with some debugging options switched off / better tuning of schedulers etc, and possibly use tools like `oprofile` to check what is really going on.

In order to determine the internal overhead of `blkreplay`, play around with the following parameters:

[27] The number is used to determine the ASCII sort order, and thus the order in which modules are `sourced`.

`--threads=...` (or `threads=...` as a shell variable in `*.conf`) The most important setting. Although counter-intuitive, too high numbers of threads may increase the overall overhead and thus reduce throughput. Be prepared to degrade this to rather low numbers in order to really achieve extremely high IO throughput. Usually, no more than 32 should be used.

`--dry-run` (or, as a shell variable, `dry_run` with underscore instead of minus) Omit any `read()` and `write()` operations, but continue to use `lseek64()` on the file handles. This way, you can determine the internal overhead in isolation.

`--fake-io` Additionally omit even `lseek64()` operations, as well as creation of header data. Compare results with `--dry-run`.

`--fan-out=...` In order to reduce parallelism on `pipe()` channels, `blkreplay` spawns internal distributor threads, forming a tree with limited fan-out degree. Too high numbers will increase pipe overhead, while too low numbers will increase thread and scheduling overhead. The optimum may depend on hardware (e.g. NUMA) and kernel version.

`--no-dispatcher` Omit distributor threads for the answer channels completely. When combined with high numbers of threads, this may increase the kernel overhead to almost unusable regions. Use this for kernel tuning.

`--bottleneck=...` Limit the number of flying IO requests. Has a similar effect than `--threads`, but is not the same. Can be used for determining internal kernel and even hardware overheads such as TLB thrashing (e.g. by keeping `--bottleneck` constant and increasing `--threads`). Could be useful for kernel and hardware tuning.
Hint: you can even set this to much higher numbers than `--threads`, in order to pre-fill the communication pipes with some ahead requests (minimizing the risk of waiting times). However, when the number is way too high, a *coprocess deadlock* could occur (see the literature about coprocesses).

`--speedup=...` A very dangerous option when used with natural loads. Leads to heavy distortions of the relationship between timely and positionly behaviour; in essence *a natural load will not remain a natural load any more*. For artificial loads, however, such as `*bursts*`, it is useful to reach top IOPS regions.

In any case, extremely high IOPS rates are likely to be reached only by running multiple `blkreplay` instances in parallel.

4. How to use `blktrace` for Recording of Natural Loads

The tool `blktrace`, as well as its friends like `blkparse`, are described elsewhere. Here are some additions from a practical viewpoint of IT operations:

- You will need a kernel with compile-time option `CONFIG_BLK_DEV_IO_TRACE` enabled. Although many newer Linux distros have it already enabled by default, you may stumble over elder systems / distributions where it was missing (so you may need to install and/or recompile a different kernel).
- Please note that some `man` pages about `blktrace` contain unclear / misleading information about the physical unit of the `-b` / `--buffer-size` option, which may lead to serious problems (including kernel hangups / crashes) if you (accidentally) try to allocate more kernel memory than physically available, and/or your production server is heavily loaded. Unless you are recording extremely high-performance loads, the default settings will succeed; so there will be no risk for IT operations.
- Tracing multiple devices in parallel is possible, but sometimes works differently than (inconsistently) described in the `man`, possibly leading to kernel problems (depending on versions etc). Before recording blocktraces at production sites, we recommend to test it somewhere in advance.

Please note that you need enough filesystem space for recording, residing on a *different* partition than you want to record the IO traffic! Otherwise, your measurements might be distorted.

You should record at least 8 hours of operation; better 24h.

Once `blktrace` has finished, some files *`yourname`*`.blktrace.{0..7}` (or other numbers, corresponding to the number of CPUs) will exist in the current working directory of your production machine. Please copy all of them to your workstation before proceeding (never risk more than necessary on production systems).

After copying, issue the following command:

```
/path/to/blkreplay/scripts/conv_blktrace_to_load.sh yourname
```

This will produce an output file *`yourname`*`.load.gz` in your working directory. You should seriously consider renaming it to *`some_better_name`*`.load.gz`.

The result file will contain as much IO requests as the script could find in the `blktrace` recording, but it will not contain any original timings (durations of the requests, aka latencies). The latency column is simply set to 0.0 everywhere.

There is a reason for it. Extracting timing information *reliably* from a `blktrace` recording is difficult. The problem is that each driver / subsystem in den kernel uses different trace action characters to indicate the start of an IO request (as well as for indicating various intermediate stages of request processing). Request completion is *another* event denoted by character '`C`'. Depending on kernel version / driver, the completion events may be even missing at all (some devices on very old kernels, which nevertheless occur in practice). More seriously, it is a hard task to determine pairs of events, i.e. determining which completion event belongs to which submission event. There is no concept of a "request ID" which would allow this easily. You have to guess matching pairs from their sector numbers / length etc. Guessing may be wrong due to ambiguities. In general, a completion could sometimes even belong to multiple submission events. In addition, when you start and/or end your `blktrace` recording at a heavily loaded *production* server (which is just the most interesting case), some pairs will very likely remain incomplete at the start and at the end.

Even under "normal" circumstances, we found many cases where non-matching events led to an incomplete `*.load.gz` result. Therefore we decided to omit timings from our default conversion script, at least for now[1]. However, there is another script which tries to *guess* the

[1] Please submit any improvements to the author as patches / git pull requests.

timings as best as it can:

```
/path/to/blkreplay/scripts/conv_blktrace_to_load_with_guessed_timing.sh
yourname
```

The latter script will name its output *yourname*`.guessed.load.gz,` which is different from the former one. There is a reason for it. Please don't remove the `*guessed*` infix when renaming that result file. We found cases where the difference was considerable.

Very often, some single guessed timings are obviously wrong. You can see it at the sonar diagram.

Please use the `*guessed*` version only for **analysis**, such as producing `*.latency.flying` or `*.smooth.latency.flying` from `graph.sh --static --dynamic` in order to determine the original IO parallelism and other interesting properties.

Only use the `*.load.gz` output stemming from `conv_blktrace_to_load.sh` as input for `blkreplay`. Never use `*.guessed.load.gz`. You may be missing some essential behaviour which could be crucial for detection of incidents!

When your load contains some interesting effects and could be of broader interest to a larger community, please contact the author of this paper by Email in order to publish your load at `www.blkreplay.org`. In particular, unusual and novel application areas, heavy ordinary workloads, and recordings of incidents, are of a broader interest.

5. Experiences with some Setups and some Loads

5.1. Overload Tests

Overload tests try to reveal the behaviour of a system under exceptionary conditions, similar to a worst-case scenario.

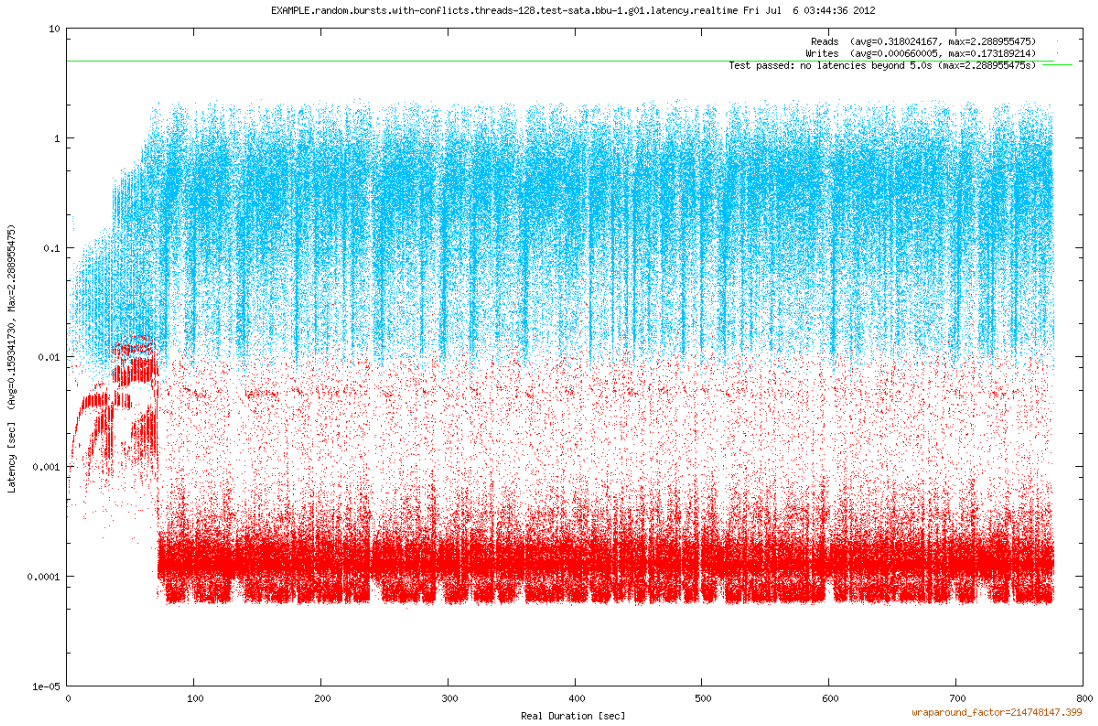
Overload tests can (and should) be done with natural loads (such as http://www.blkreplay.org/loads/natural/1and1/natural-derived/). In addition, it is one of the rare places where *artificial* loads can reach *some* justification.

5.1.1. Overload with Artificial Bursts

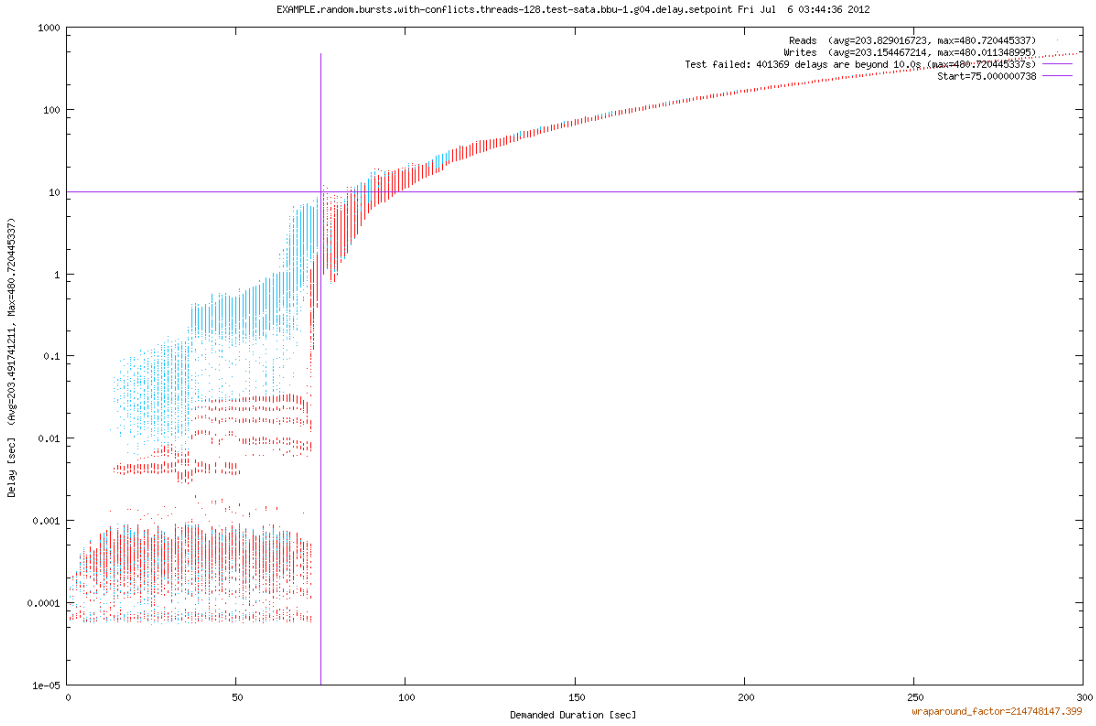
The following examples are using the artificial loads from http://www.blkreplay.org/loads/artificial/random.bursts/, which are rather "anal" ones (see file `00README` there). The load is linearly increasing over time, until the test candidate will "collapse" under the overload.

We use a single `blkreplay` instance, and compare between an old SATA RAID-6 and a newer RAID-6 with faster SAS disks (15k RPM). For now, we consistently use `--threads=128`. All results can be found in the directory `blkreplay.git/example-run/` and subdirs.

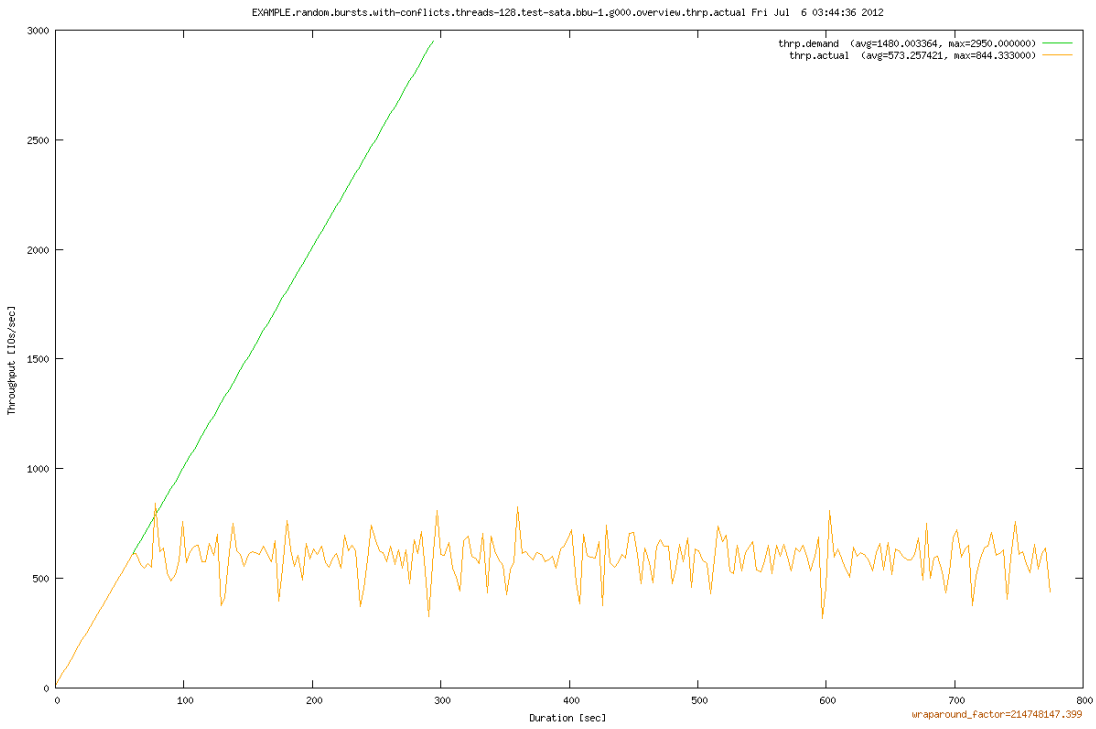
Here is the sonar diagram for the SATA system. It *should* have run 300s, but it actually run much longer in realtime:

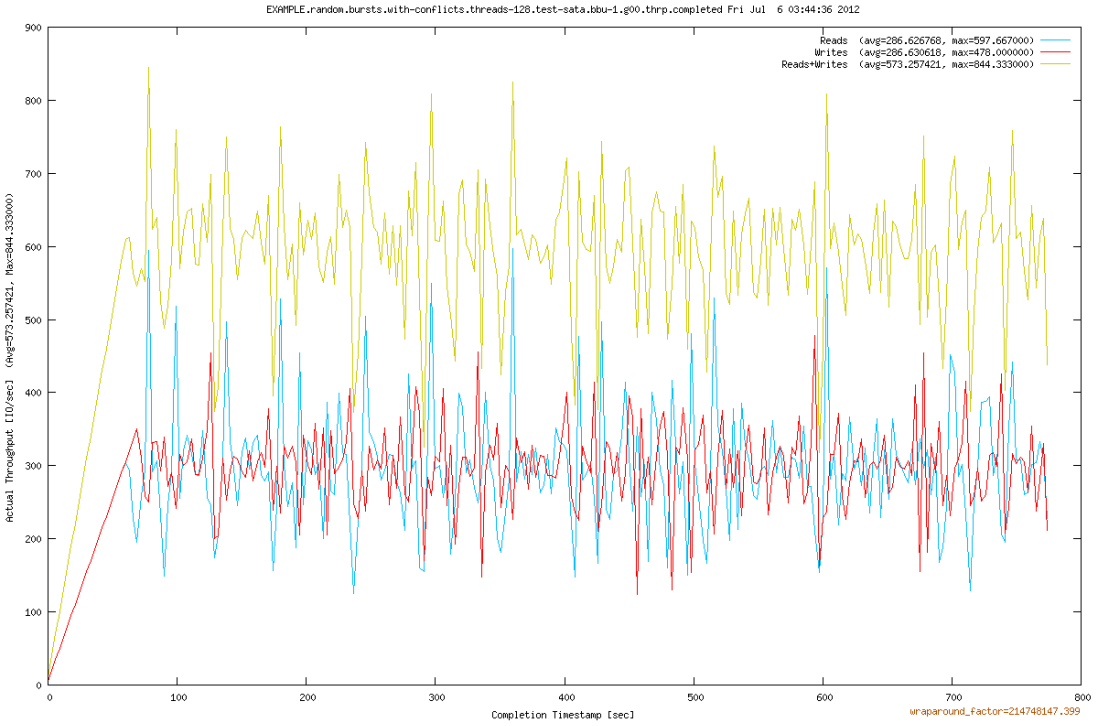


At the beginning, the load is rather low, and the system can catch up. Soon, it reaches a kind of "maximum throughput" where the latencies go up to about 1 second. You can see on the realtime x axis that the replay is delayed as a whole. This can be better seen at the delay diagram:

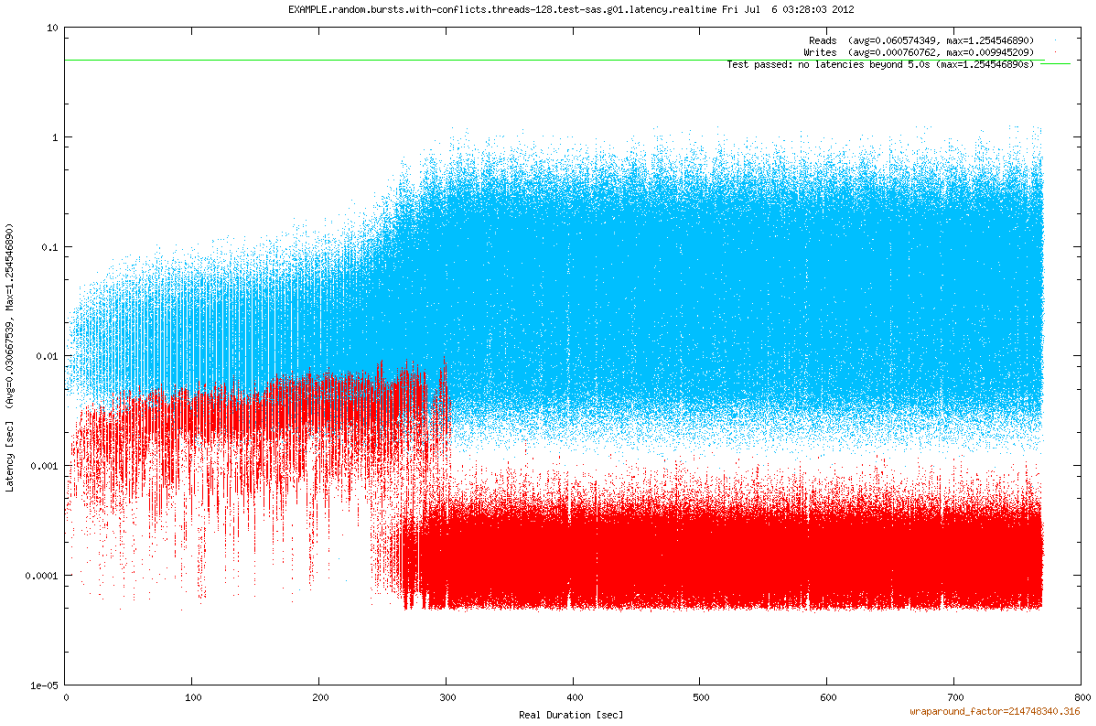


Here the x axis shows the *intended* starting time of each request, which is deliberately concentrated at whole seconds (with gaps inbetween), a property of of the `*bursts*` loads. The "collapsing point" is about 60s, which means that it roughly can deliver 600 random IOPS. This can be even better seen at the overwiew `thrp` comparing the actual with the demanded throughput, as well as the detailed `thrp` diagram of the actual throughput ordered by completion time of the requests:



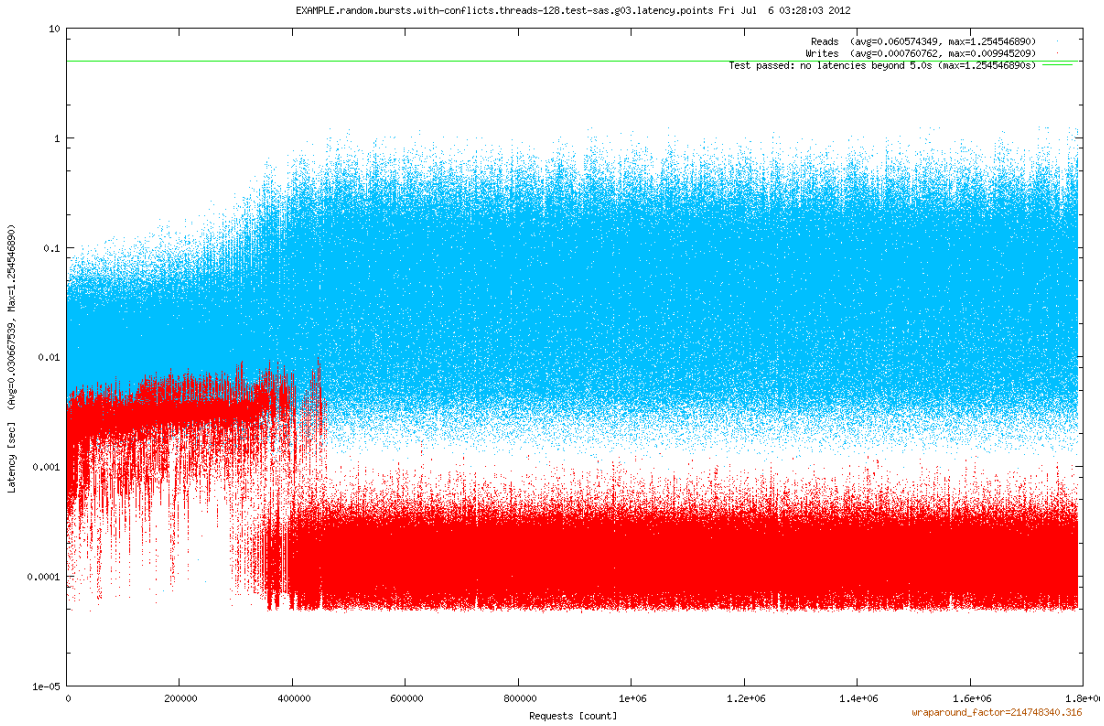


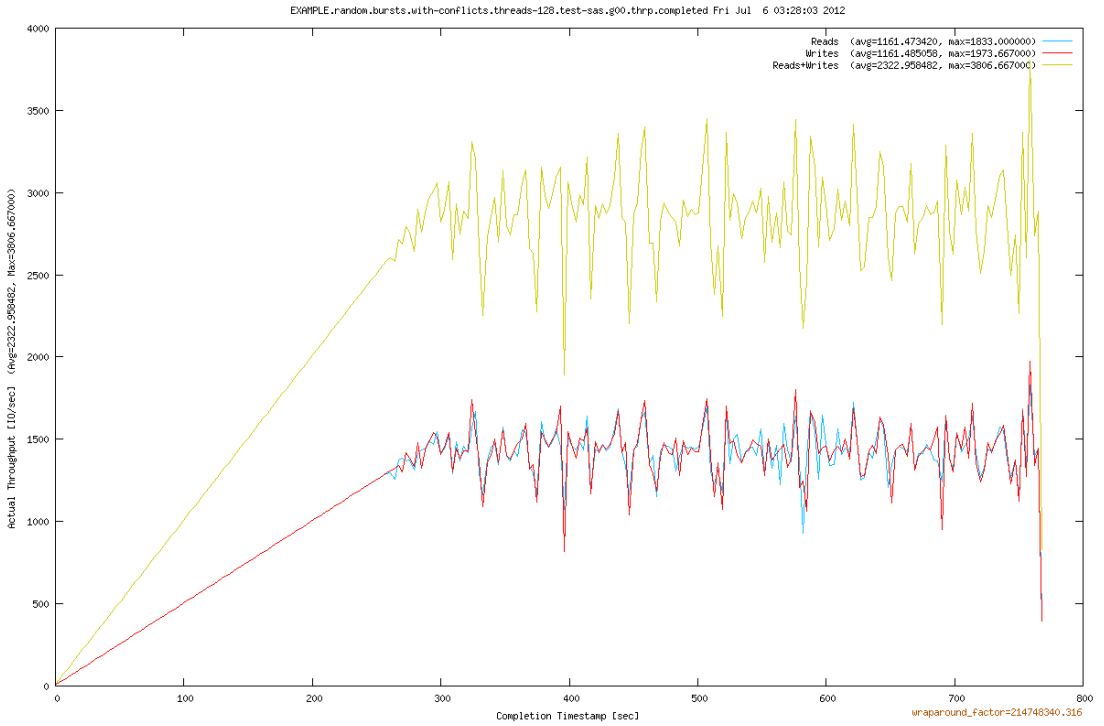
Now we turn to better hardware: the SAS system uses smaller disks with 15k RPM, and thus needs more spindles to reach the same capacity. If we would run the same test for 300s (as in the SATA case), we would not notice any collapsing. Thus we double `replay_duration` to 600s in order to be able to see the collapsing point. Please take this into account when looking at the following graphics:



Notice that the system is even able to reproduce the bursting behaviour of the load, much better than the SATA system. Only at higher request rates, some "blurring clouds" appear on the x axis, until it finally collapses under too high load. It is amazing to see the "binary" behaviour at the collapsing point: as soon as the actual throughput rate cannot catch up with the request rate, the internal queues will suddenly grow up to some internal maximum, and thus lead to

latencies around 0.1 seconds, which is better than SATA by an order of magnitude. Of course, the same effect can be observed at the delay diagram and the detailed throughput diagram:





5.1.2. Overload with (Derived) Natural Loads

Overload tests with natural loads are probably the most interesting ones. They can tell you a lot upon both the test candidate, as well as upon your application.

In many cases, natural loads are varying very much over time. Frequently, the variance in the IOPS rate can span an order of magnitude, or even more. Typical reasons are cron jobs or nightly backups on an otherwise (almost) idle server.

In order to find the peaks in the load, you have to draw `blktrace`s from production servers for at least 24h. After conversion to `*.load.gz`, it is rather cumbersome to find the load peaks by hand in order to get suitable input data for *overload* tests. Thus, we have developed the script `create_derived_load.sh` which will do the following for you:

1. It splits the input file(s) into snippets of equal length in realtime (by default, 5min length each).

2. It sorts the snippets according to IOPS in reverse order, i.e. the "heaviest" snippet will come first.

3. It distributes the snippets to a number of output files in a round-robin fashion (while adjusting the timestamps to the new order).

As a result, the output files will start with the heaviest IOPS load first, and later declining. This methodology has a number of advantages:

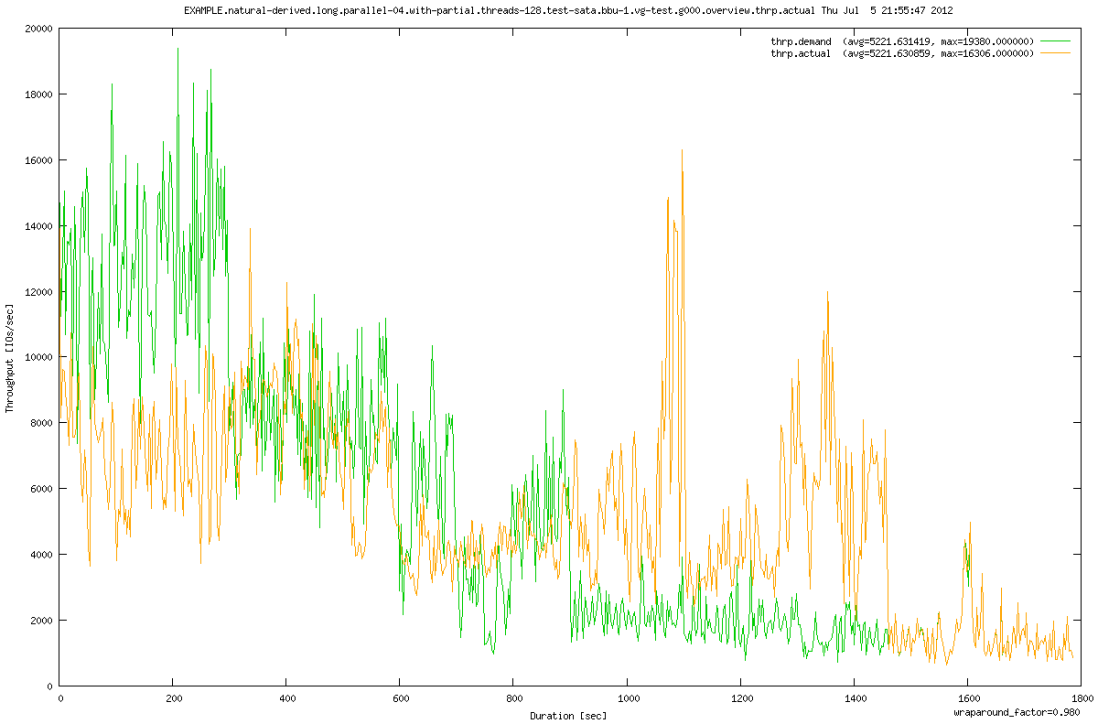
- The overload part is easily accessible at the beginning of each output file (interesting part first).

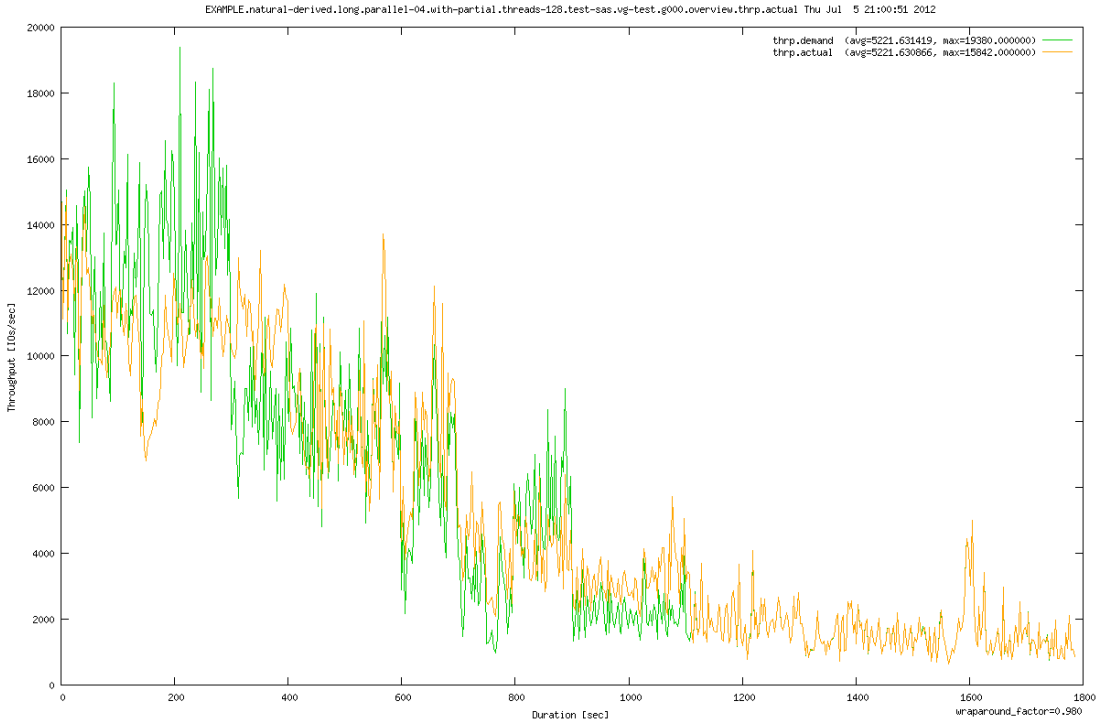
- Thanks to the distribution to many output files, higher replay parallelism is possible in an uncorrolated way.

- In case your test candidate should get overloaded: you can easily see its *recovery behaviour* from overload, thanks to the declination of the load over time. Will it recover at all, and how fast?

The latter is an important property of any enterprise-grade storage system.

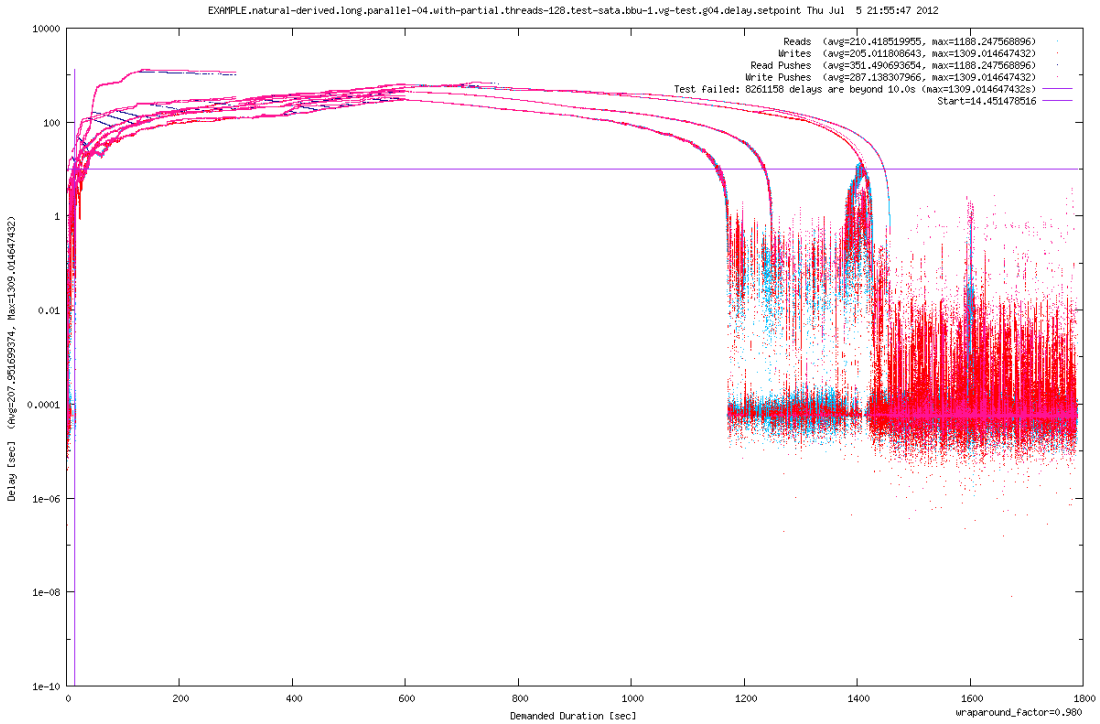
Here are some example comparisons between an old SATA system and a newer SAS system, both equipped with hardware RAID-6, and loaded with a replay parallelism of 4. You can find the complete results with additional graphics as a tarball at http://www.blkreplay.org/examples/.

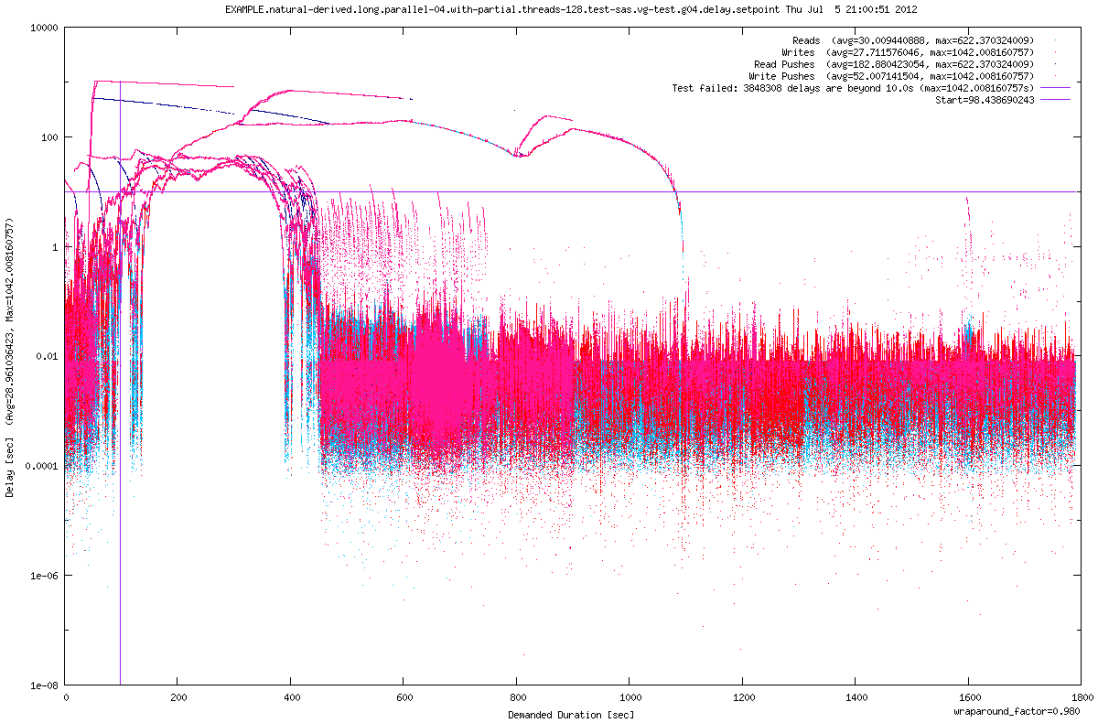




It is interesting to observe the behaviour of the SATA system in contrast to the high-performance SAS: at first, the real IOPS rate (orange) cannot catch up with the demanded rate (green). However, when the demanded rate decreases under the green line after some time, the orange line becomes higher than the green one until the backlog has been worked up. Finally, the system can (almost) run in sync with the lower demands, and the orange line is hiding the green one.

The same effects can also be seen in the corresponding delay diagrams, where you can even notice multiple top lines stemming from overlay of replays in parallel, and even watch their different fallback points:



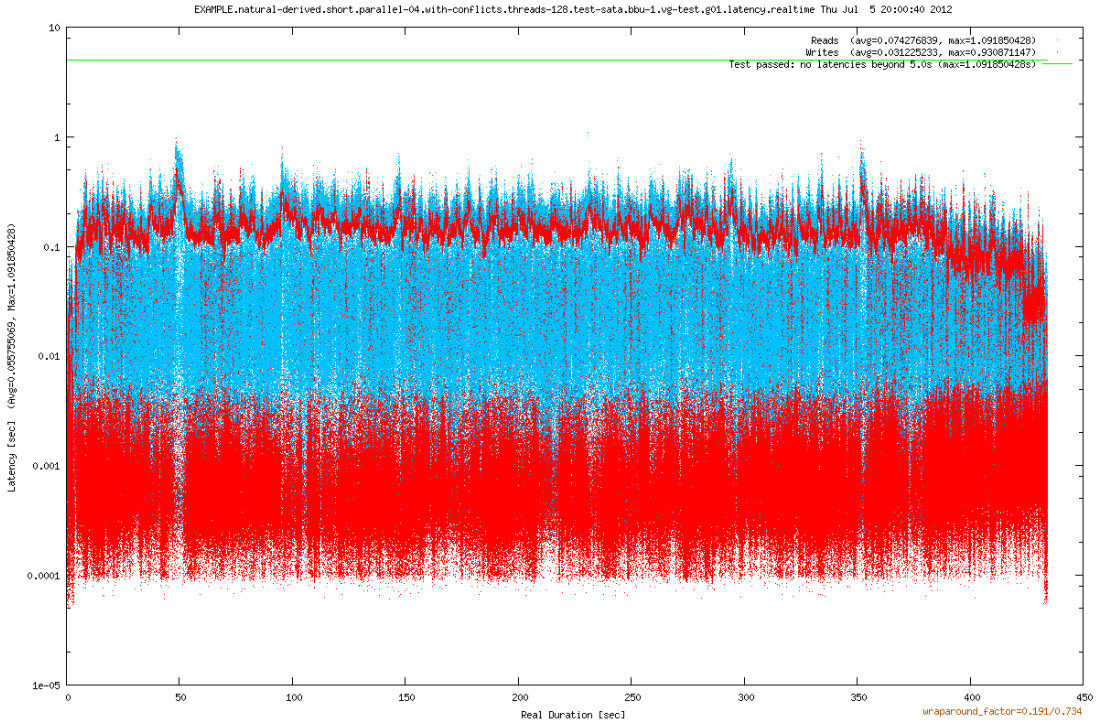


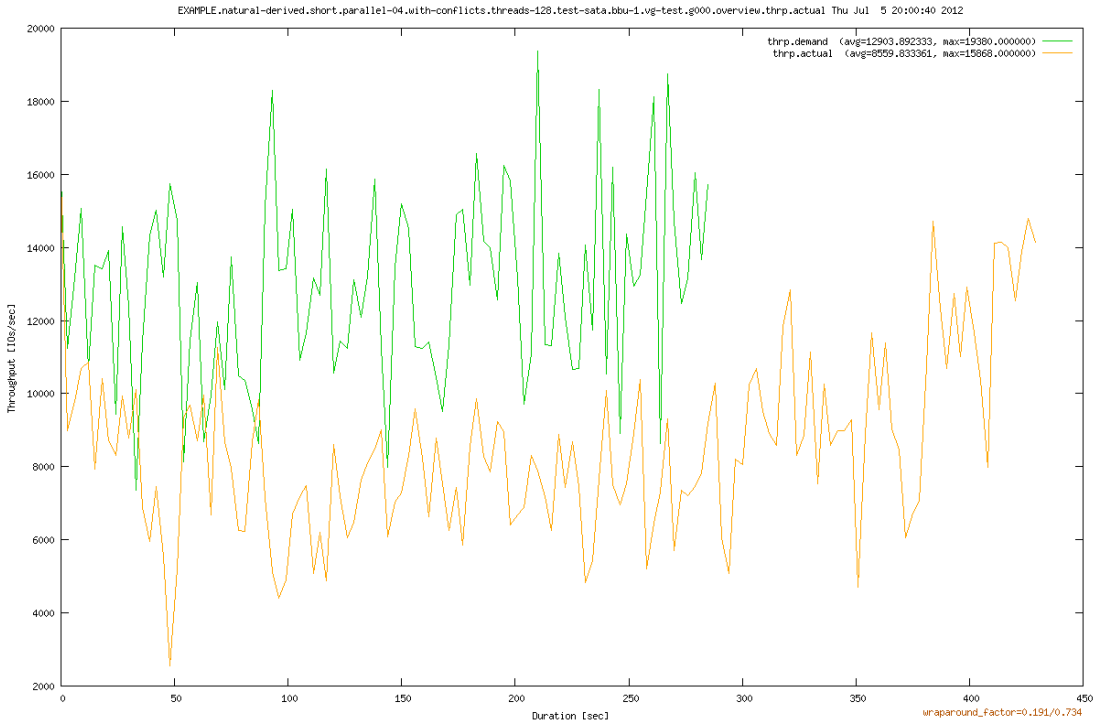
5.2. Influence of Replay Parameters

The following examples are run on the rather slow SATA system. By default, we use `threads=128` and 4 `blkreplay` instances, operating in parallel on 4 LVM devices created from the same volume group. Therefore, the total IO parallelism is $128 * 4 = 512$. Exceptions from that are denoted.

5.2.1. Influence of Request Ordering

We start with the simplest case: mode `--with-conflicts` fires up all requests without guaranteeing any order (besides that requests are never started too early), which may lead to damaged IO. Here is the sonar diagram as well as the throughput diagram:

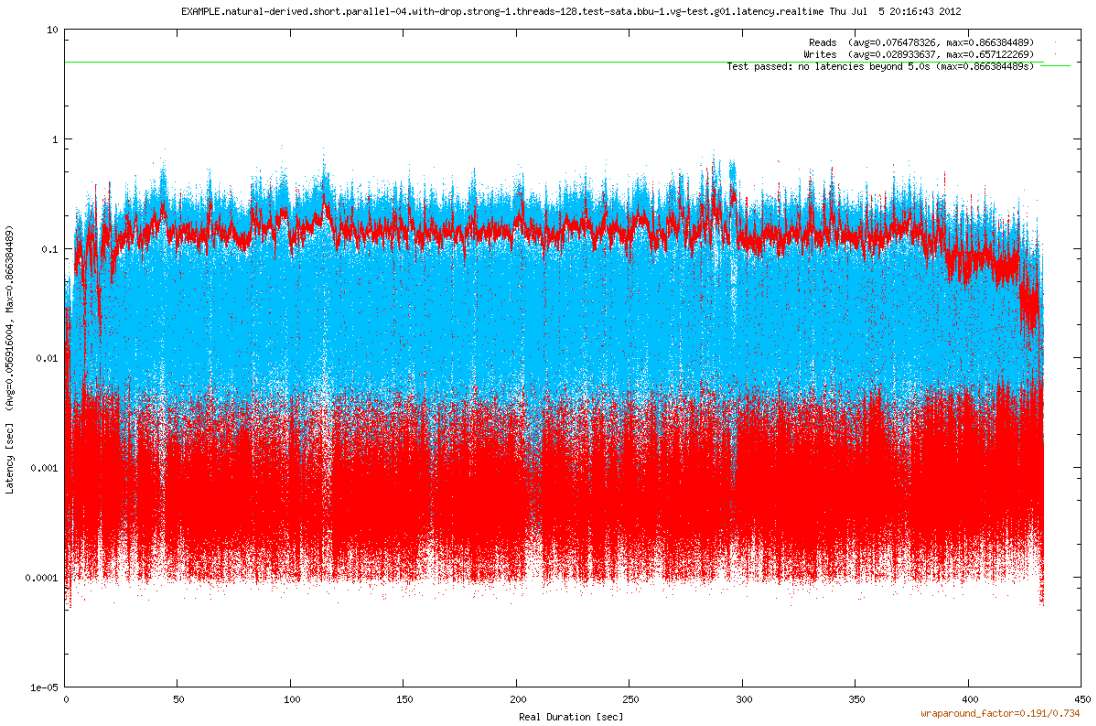


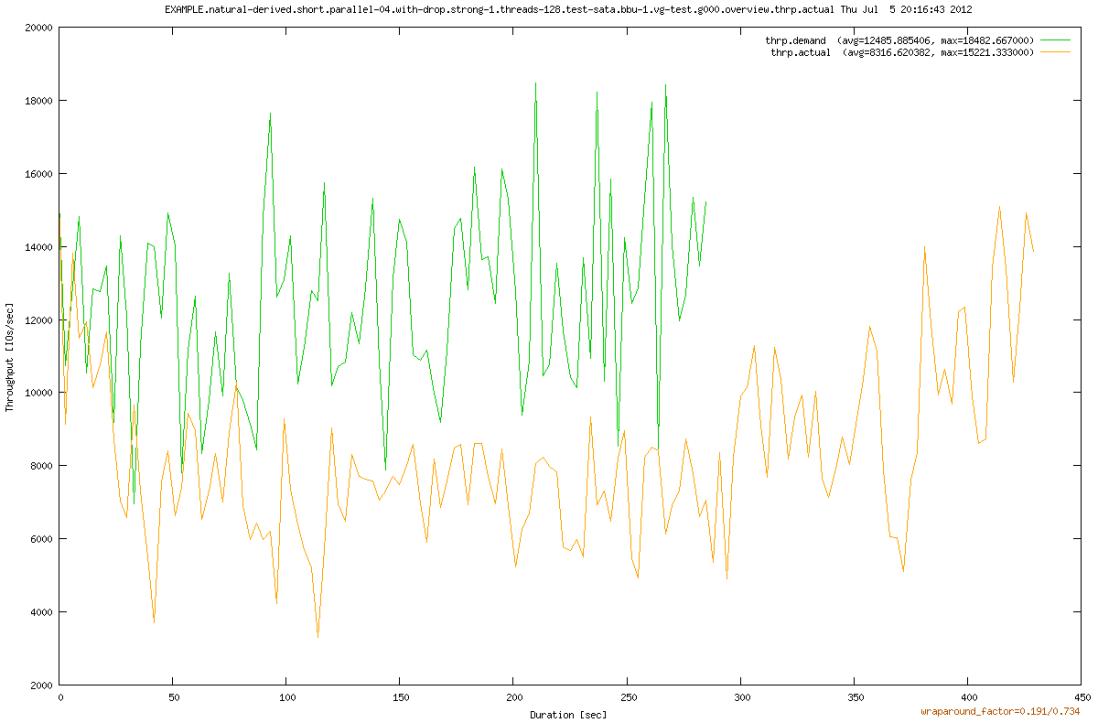


It is clear that the SATA cannot catch up with the high demands here. But why? Is it the damaged IO?

A first hint is the rather "thick" red "cloud carpet" in the latency diagram near 1s, indicating some queueing behaviour. Interestingly, this carpet falls a little down at the end, when the replay parallelism decreases due to different termination of different `blkreplay` instances.

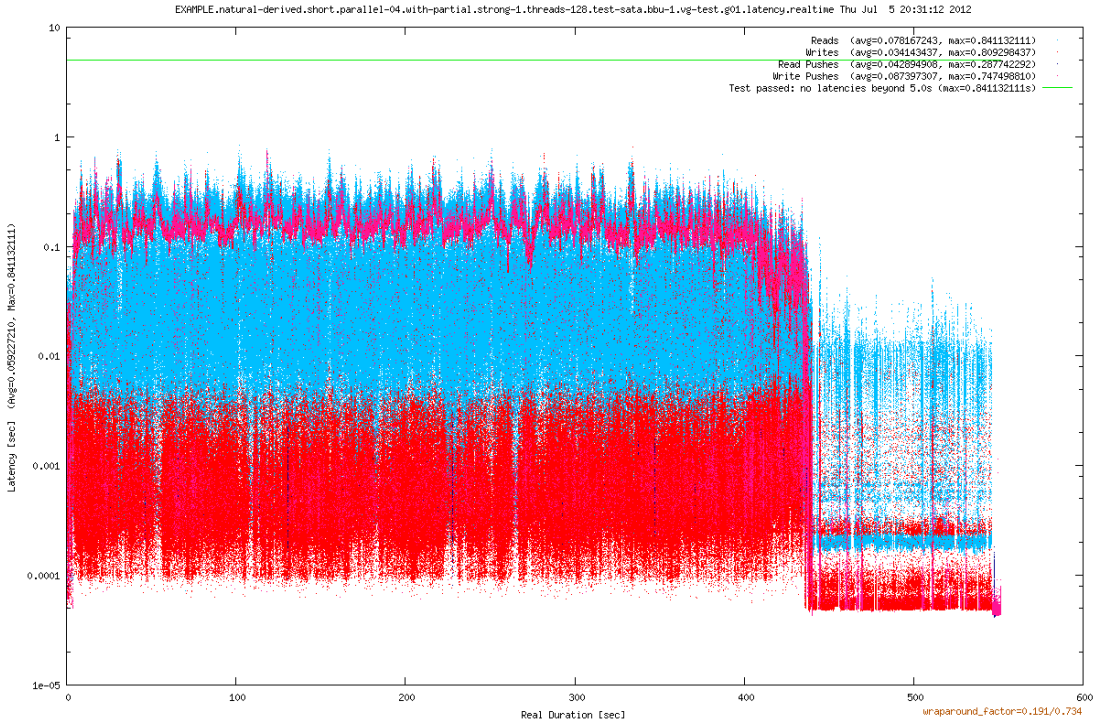
Now we turn to `--with-drop`. Surprisingly, there is only a small difference, although damaged IO is avoided at the cost of some lost requests:

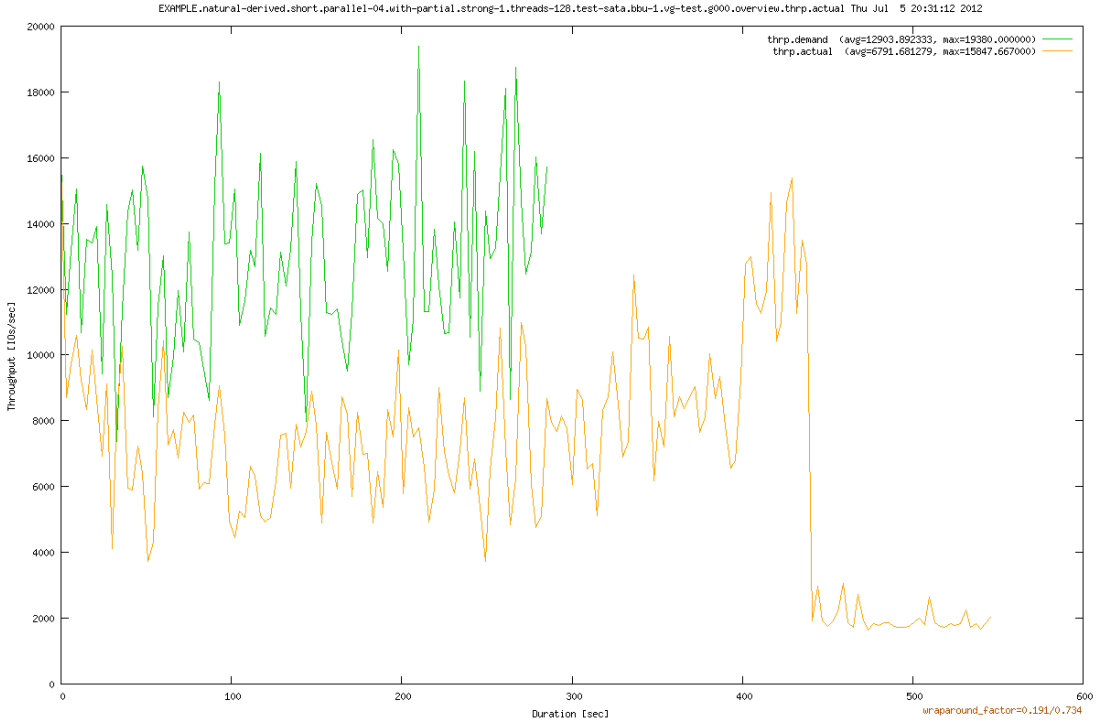




This is a clear indication that damaged IO cannot be the reason for the delays, since it is avoided by `--with-drop`.

Now we turn to `--with-partial`. Most of the time, there is almost the same behaviour as before. Only the tail is different, because the actual running times are more diverging between the four `blkreplay` instances started in parallel:





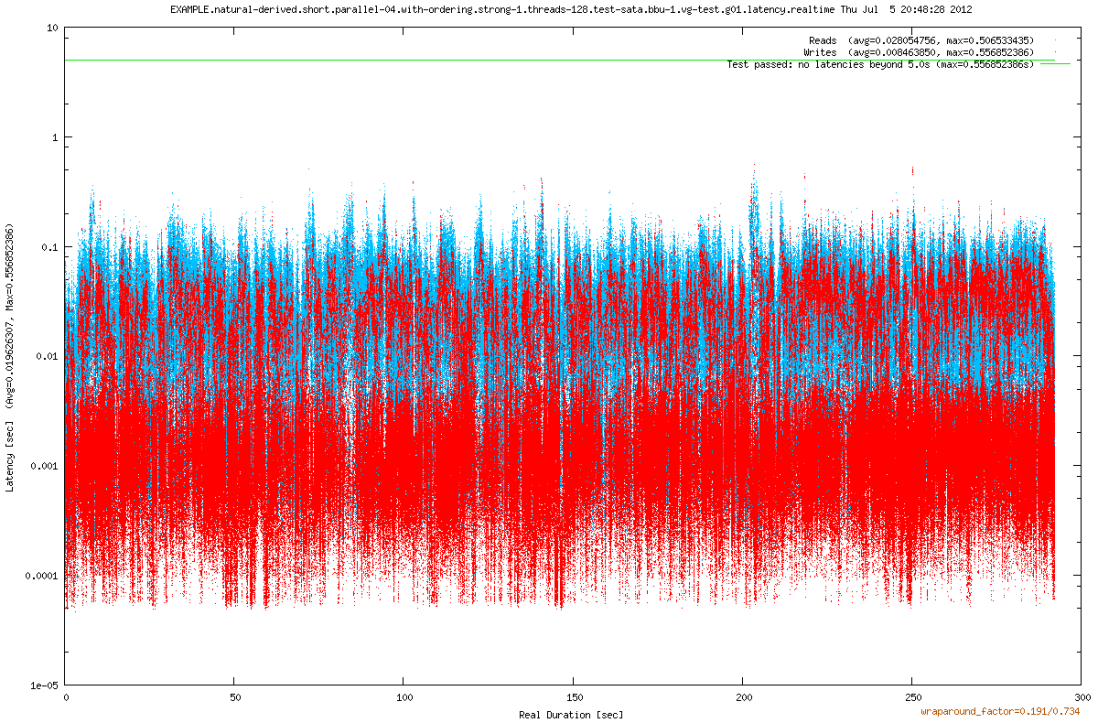
The tail may look disturbing to some people. It has a rather *low* IOPS rate. In case of delays, `blkreplay` should try to catch up as fast as possible, shouldn't it? What the heck is going on there? Is there a bug in `blkreplay`?

Examination of `*.lv-data3.replay.gz` from http://www.blkreplay.org/examples/ tells us that the real last tail consists almost exclusively of pushed-back requests. In detail, there are lots of requests depending on each other *transitively*, reaching over many generations. They have accumulated over a long time, because `--with-partial` is *defined* to do so. The tail is nothing but catch-up of pushed-back requests which could not be run earlier, in order to obey the storage semantics. Many of them are mutually hindering each other. The ordinary non-conflicting requests form a different class than then conflicting ones, leading to a 2-class society with different chances to be processed (unfairness). Thus one of the both classes finishes earlier than the other. This is a disadvantage of `--with-partial` you should know.

On the other hand, this can be turned into an advantage, because you can use `--with-partial` for detection of such a behaviour. Think of the race between Achilles and the turtle. If you find a much slower turtle, your test candidate has probably some problems with that *specific* load, but not necessarily with other loads at the same IOPS level.

Hint: the 2-class society caused by `--with-partial` is often visible in the delay diagrams. Check out yourself!

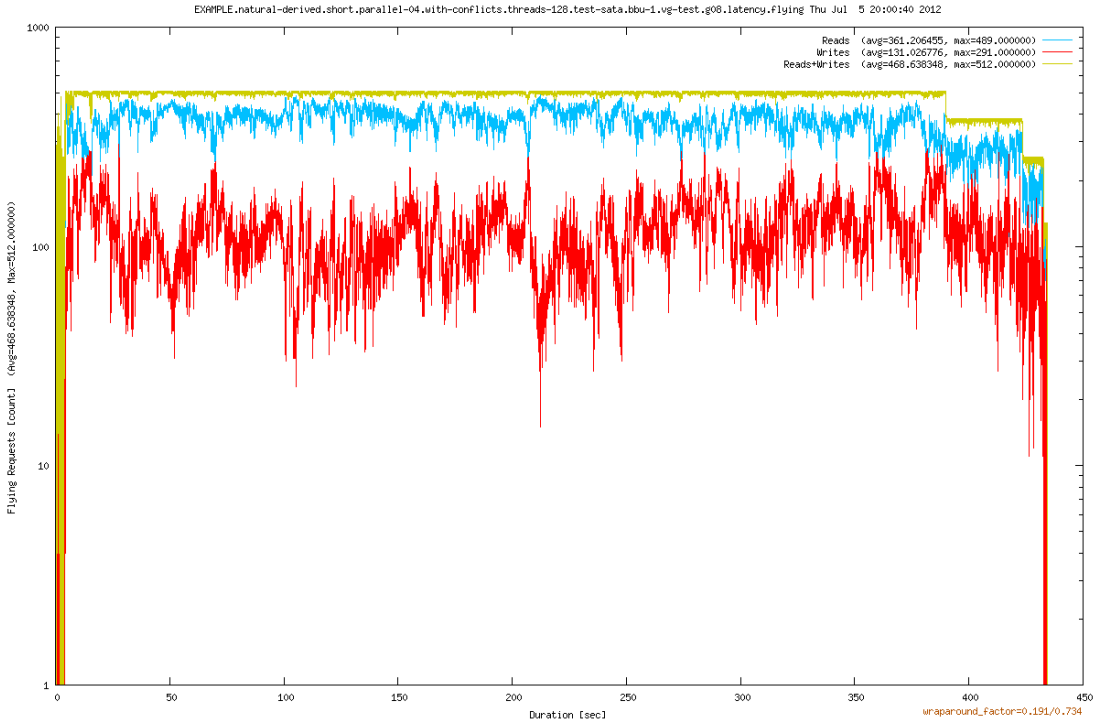
Let us return to the comparion of request ordering modes: more prominent differences can be found when we turn to `--with-ordering`. Now the system can catch up *in average* (but not in general at load peaks). The sonar diagram shows that the "thick clouds" at about 1s have fallen down and have become thinner in densitiy:

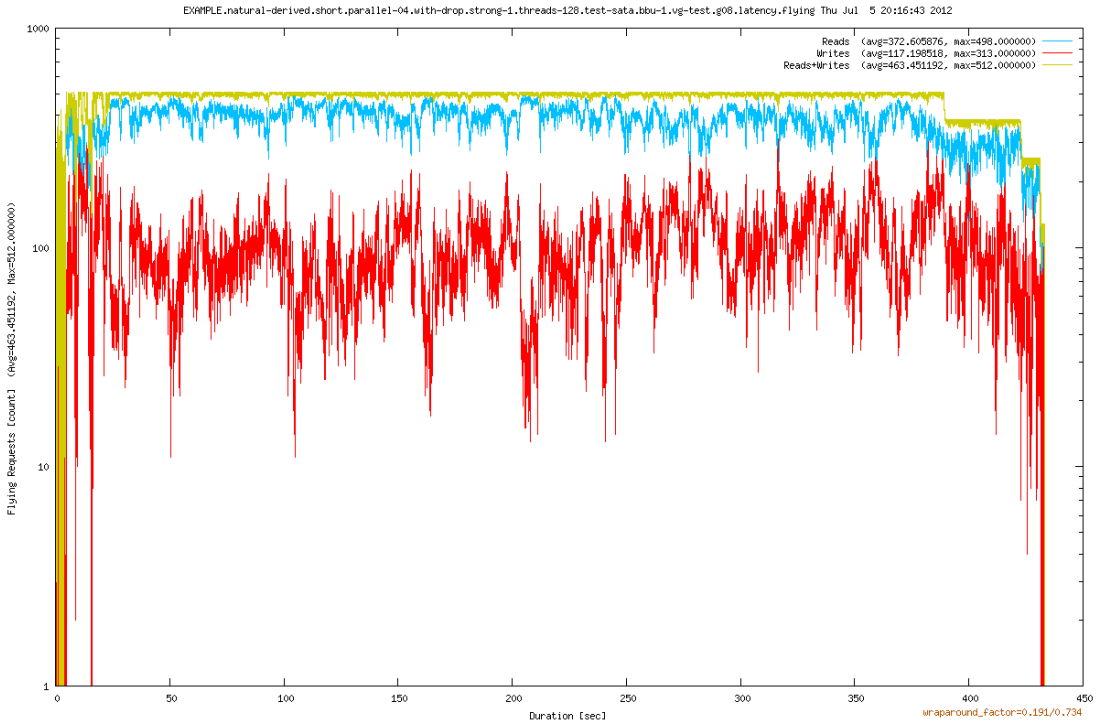


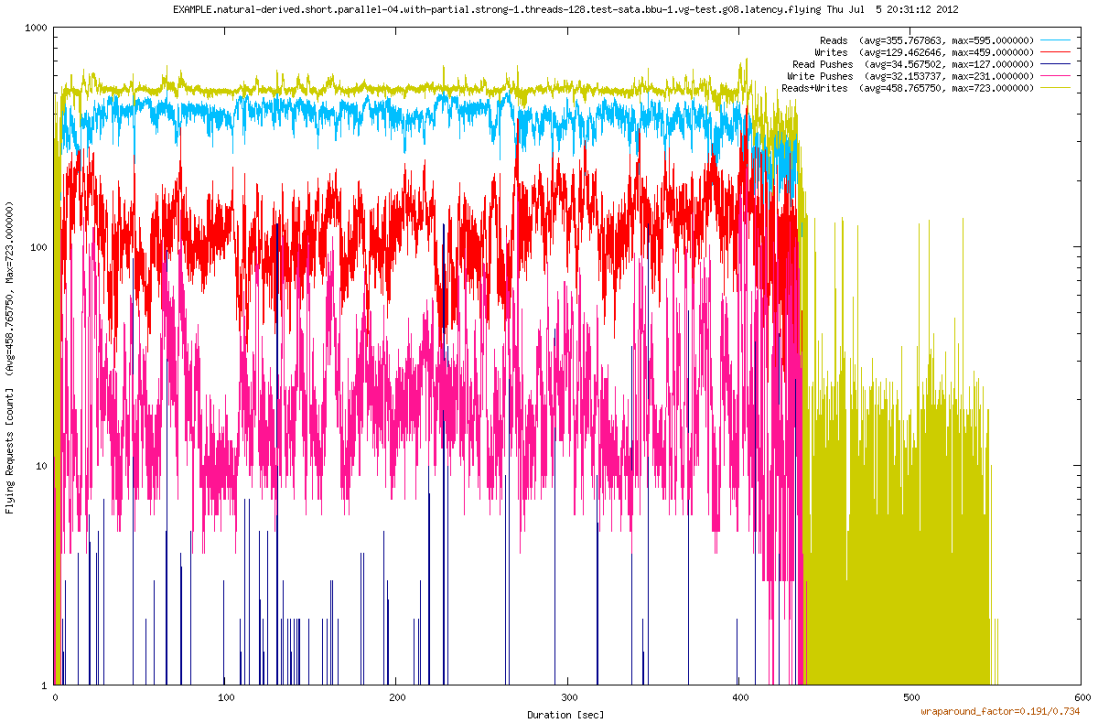
This rises the question for reasons. Why should `--with-ordering` be better than `--with-partial` or even `--with-drop`? Full ordering will usually *reduce* the IO parallelism when compared to

partial ordering. The full answer is in sections 5.2.2 and 5.2.4, but for now you can take the small "gaps" on the x axis as a hint that there must be something related to small micro-stalls, which seem to *accelerate* overall throughput in some **counter-intuitive** way. The micro-stalls are easily explainable by the submission delays caused by `--with-ordering` , reducing the IO parallelism in case of conflicts. But why does that *accelerate* the overall throughput?

Before proceeding, we take a closer look at the micro-stall effect. It is better visible in `*.latency.flying` showing the *actually achieved IO parallelism* during the replay. Here is the comparison between `--with-conflicts`, and `--with-drop`, and `--with-`partial:

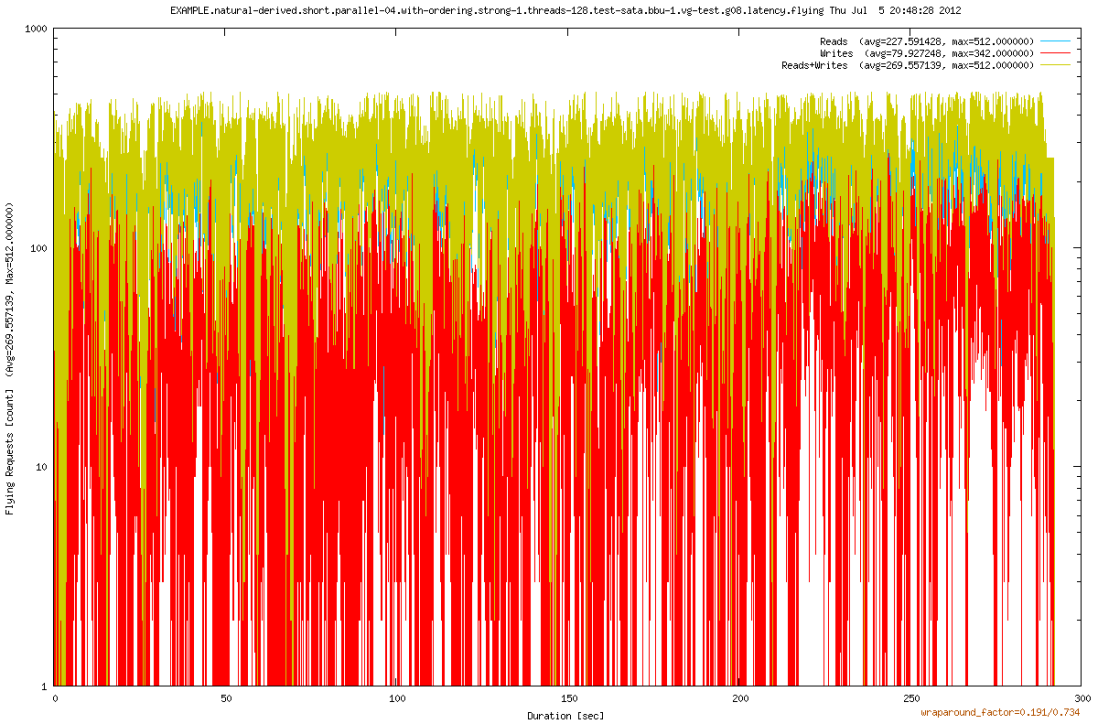






The sum of both reads and writes (olive line) indicates that the RAID controller is filled with a maximum of $128 * 4 = 512$ requests (caused by `threads=128` and a replay parallelism of 4) most of the time, except at the tail.

And now, compare these with `--with-ordering`:

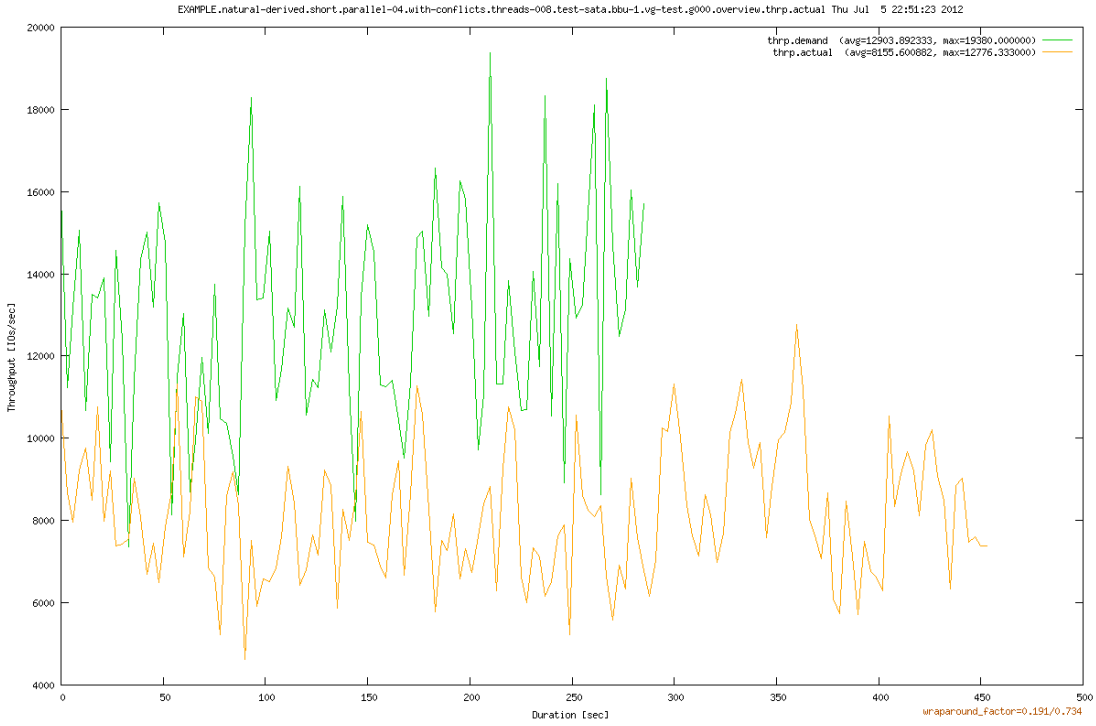


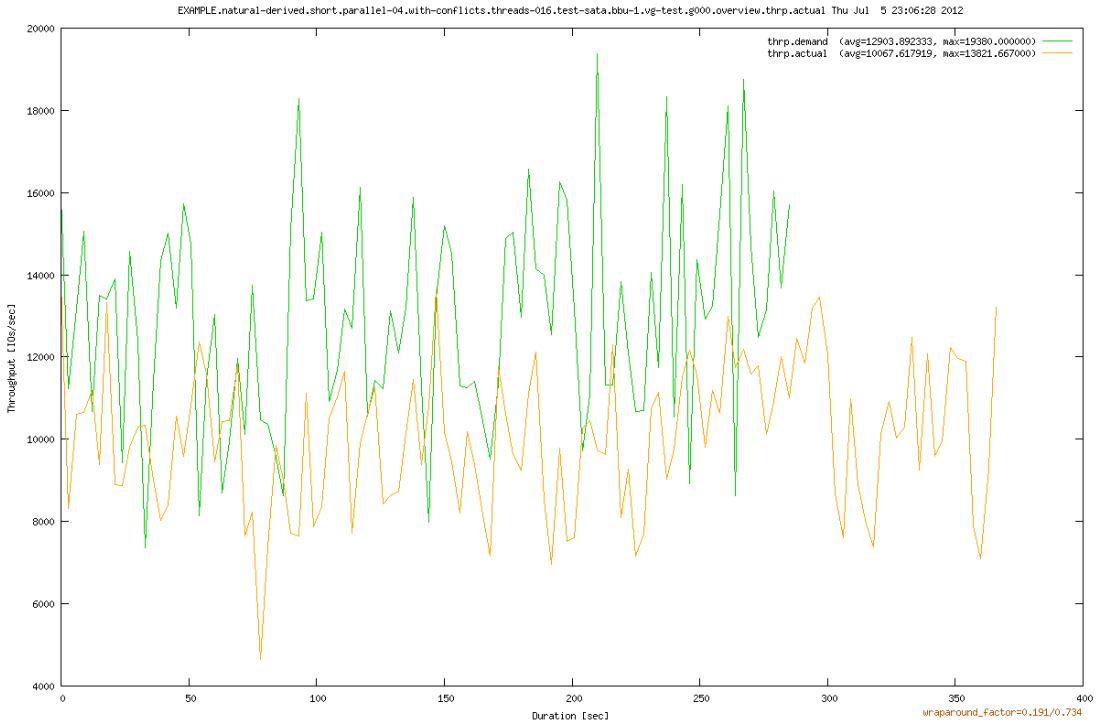
It is clear that by definition of `ordering`, some kind of stalls have to be introduced to guarantee some virtual "strict storage semantics". It is also clear that these micro-stalls will sometimes reduce the actual IO parallelism, which is clearly visible in the graphics. However, can these micro-stalls be made *responsible* for an *increase* of overall throughput?

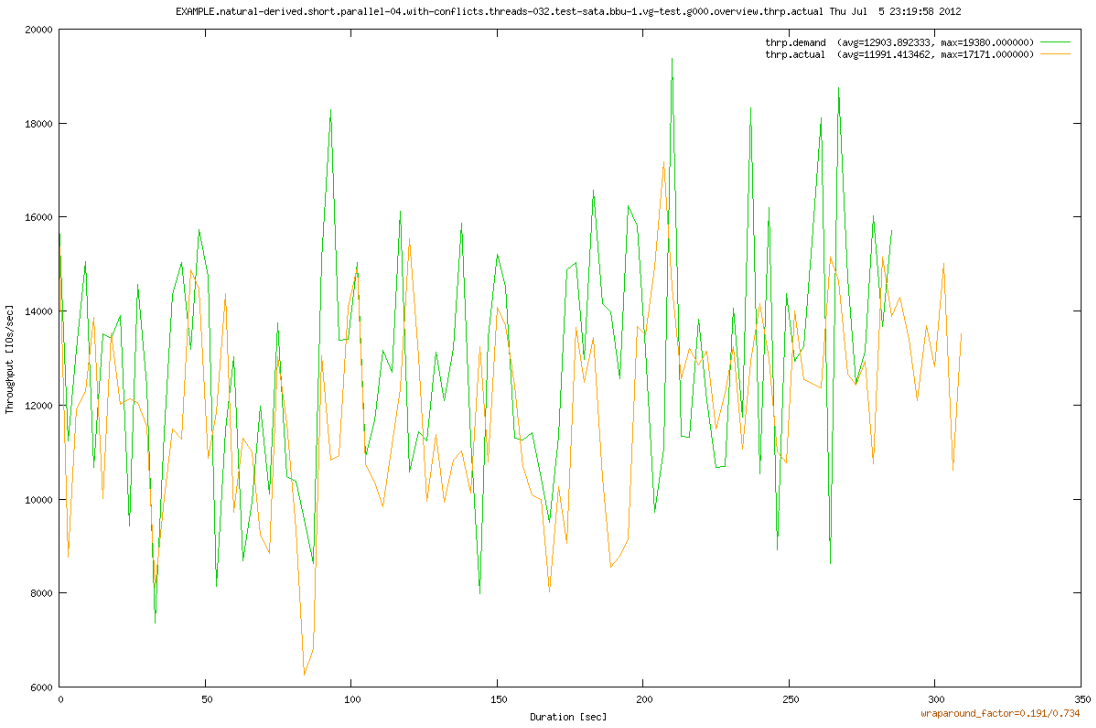
In order to investigate that, we turn to the influence of the `threads=` parameter.

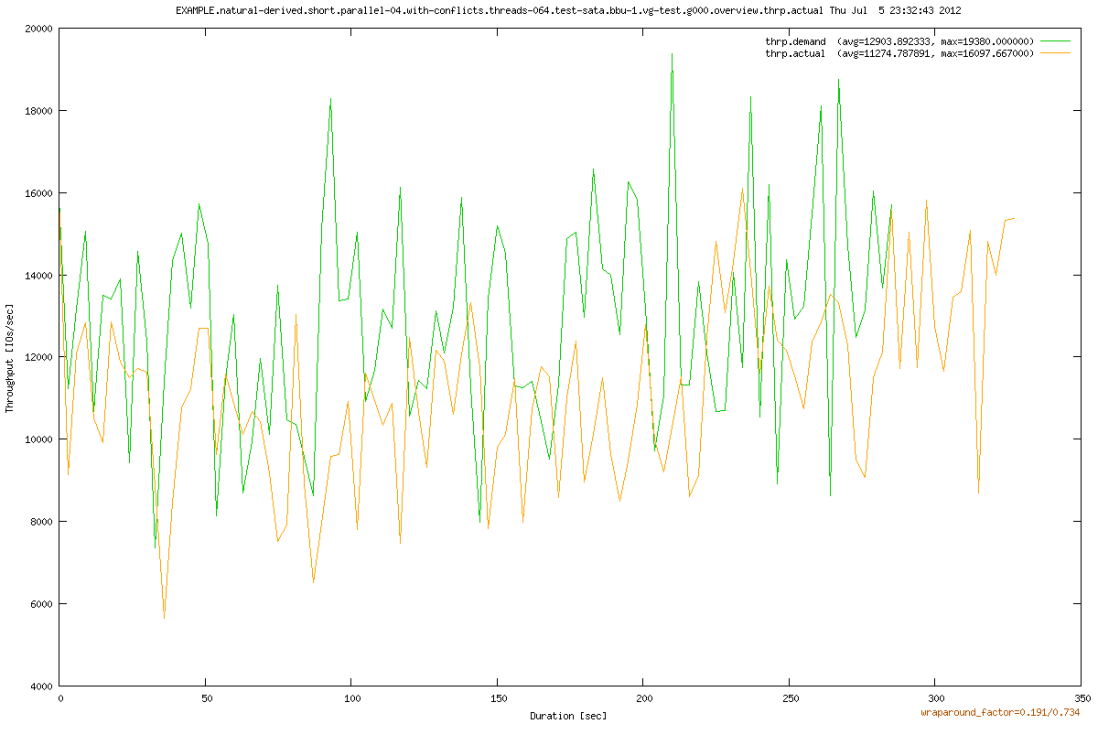
5.2.2. Influence of Number of Threads

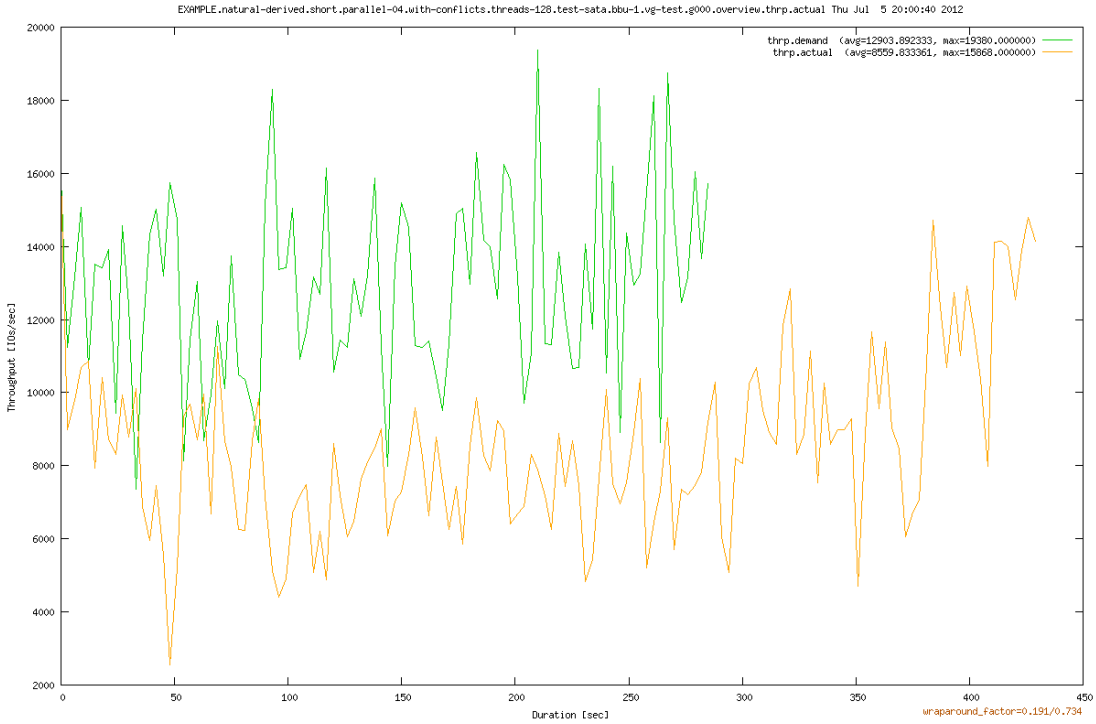
The following throughput graphics are a direct comparison between 8, 16, 32, 64, 128, 256 and 512 threads on the SATA system using `--with-conflicts`, which guarantees the best possible IO parallelism and avoids any non-linear influences from any conflict-handling strategies:

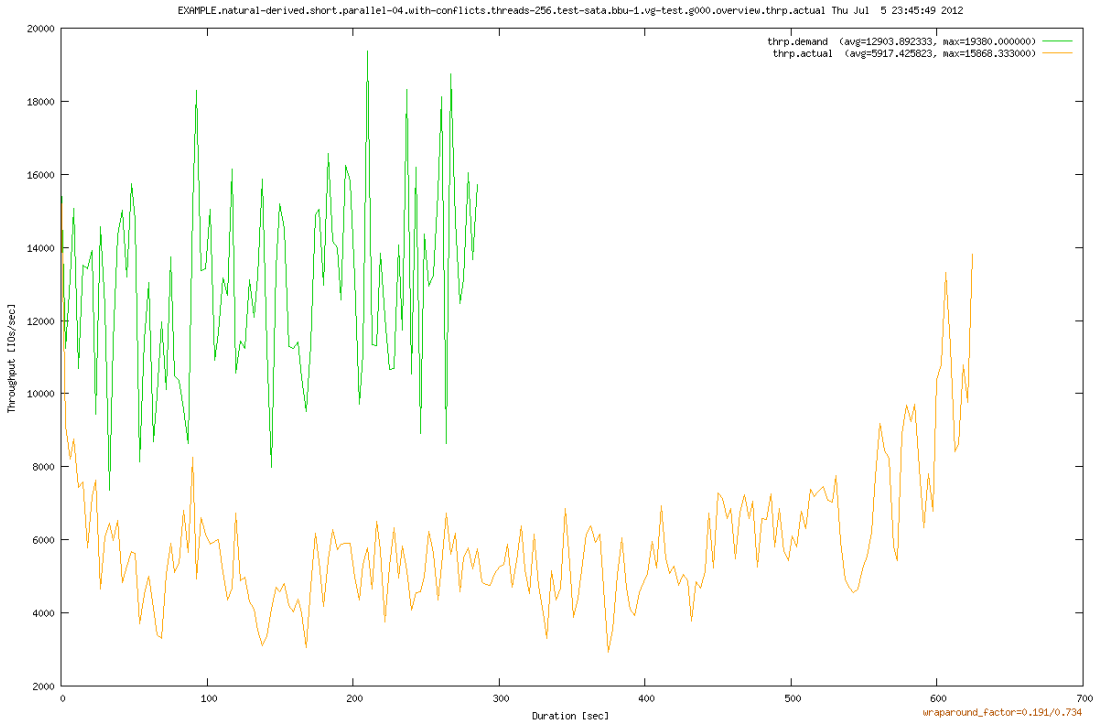


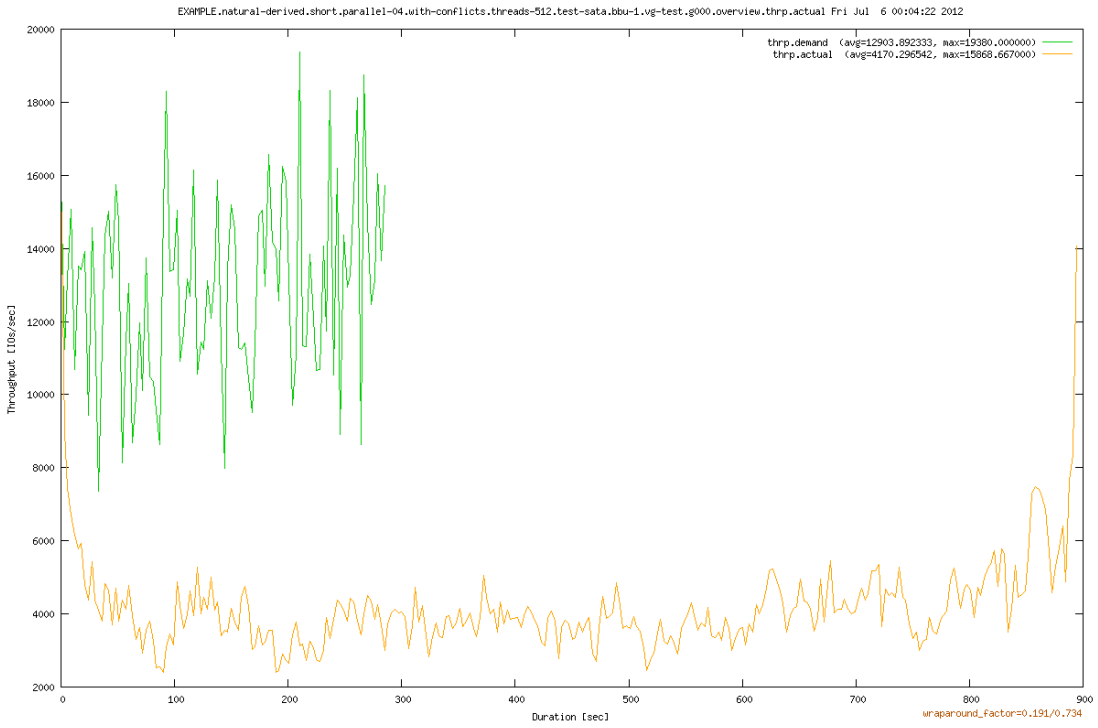



EXAMPLE.natural-derived.short.parallel-04.with-conflicts.threads-032.test-sata.bbu-1.vg-test.g000.overview.thrp.actual Thu Jul 5 23:19:58 2012
thrp.demand (avg=12903.892333, max=19380.000000)
thrp.actual (avg=11991.413462, max=17171.000000)
Throughput [IOs/sec]
Duration [sec]
wraparound_factor=0.191/0.734


EXAMPLE.natural-derived.short.parallel-04.with-conflicts.threads-064.test-sata.bbu-1.vg-test.g000.overview.thrp.actual Thu Jul 5 23:32:43 2012
thrp.demand (avg=12903.892333, max=19380.000000)
thrp.actual (avg=11274.787891, max=16097.667000)
Throughput [IOs/sec]
Duration [sec]
wraparound_factor=0.191/0.734


EXAMPLE.natural-derived.short.parallel-04.with-conflicts.threads-128.test-sata.bbu-1.vg-test.g000.overview.thrp.actual Thu Jul 5 20:00:40 2012
thrp.demand (avg=12903.892333, max=19380.000000)
thrp.actual (avg=8599.833361, max=15868.000000)
Throughput [IOs/sec]
Duration [sec]
wraparound_factor=0.191/0.734


EXAMPLE.natural-derived.short.parallel-04.with-conflicts.threads-256.test-sata.bbu-1.vg-test.g000.overview.thrp.actual Thu Jul 5 23:45:49 2012
thrp.demand (avg=12903.892333, max=19380.000000)
thrp.actual (avg=5917.425823, max=15868.333000)
Throughput [IOs/sec]
Duration [sec]
wraparound_factor=0.191/0.734



It is clear that a too low number of threads, such as 8, can hinder overall throughput (even when in reality the total IO parallelism is $8*4 = 32$ by taking the number of `blkreplay` instances into account). It is therefore no wonder that 16 through 64 threads perform better than 8. But, why is there a counter-productive break-down when increasing the number of threads after that? The effect is rather strong: 512 threads will decrease throughput almost by a factor of three when compared to the "best" setting. This is no peanuts! What the hell is going on here?

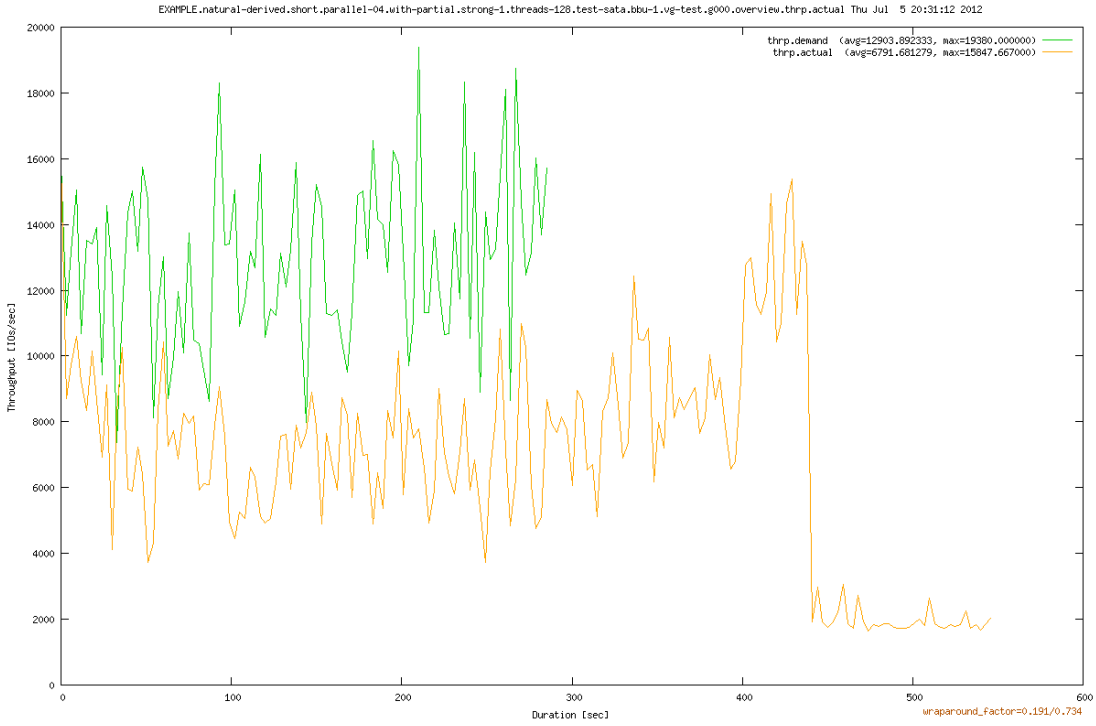
The answer can only be buried in the internals of the hardware and/or its firmware and/or its driver. Typically, some systems have problems when filled with too many requests in parallel. Here, we see a typical **non-linear behaviour** as explained in section 3.1.4.

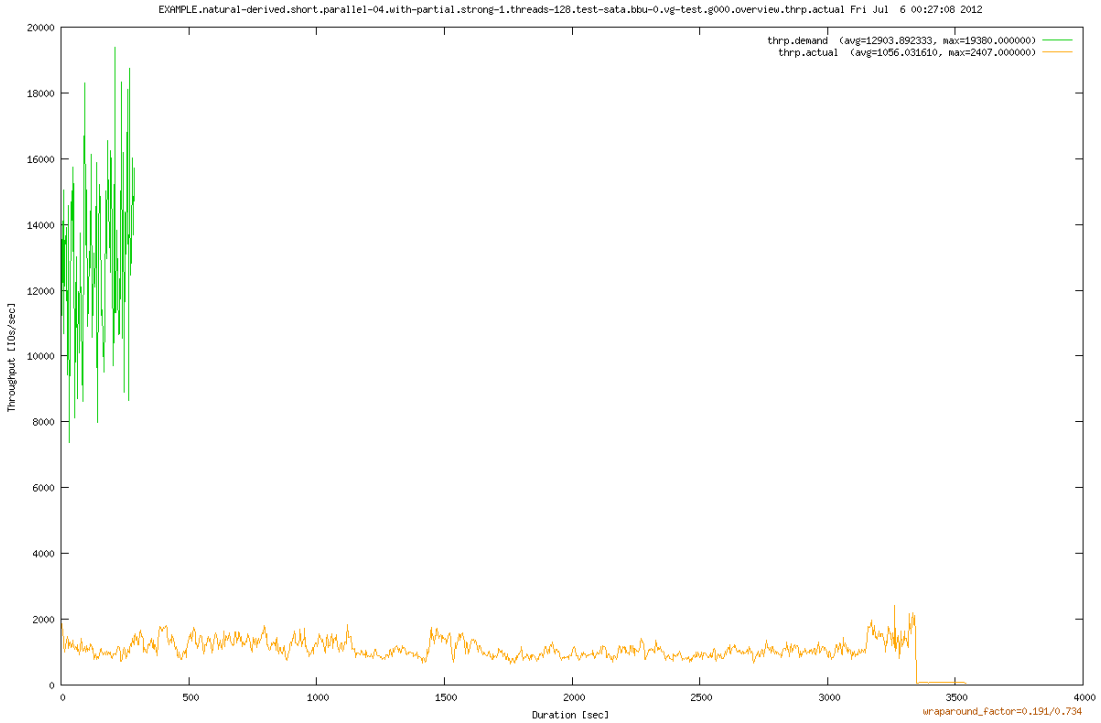
Consequences: the `threads=` parameter can be very important. And its "optimum" can non-linearly depend from the request ordering parameters. Here, you see a proof for the warnings posed in section 3.1.4.

Never adjust the `threads=` parameter to the value delivering the "best" performance when comparing different systems / vendors. Doing so will be a *lie(!)* to your audience! Always use the `threads=` setting which corresponds to your **real application** based on **knowledge** (by examining the submission IO parallelism at the original `blktrace` recording site, e.g. by looking at the `*.flying` graphics produced from the original `blktrace` recording), and **not** based on any assumptions which could be faked in any direction! Be responsible and tell your audience *why* you select a specific value, and what would happen if you selected a different one! Or, provide a series of different `threads=` settings telling the *whole* story!

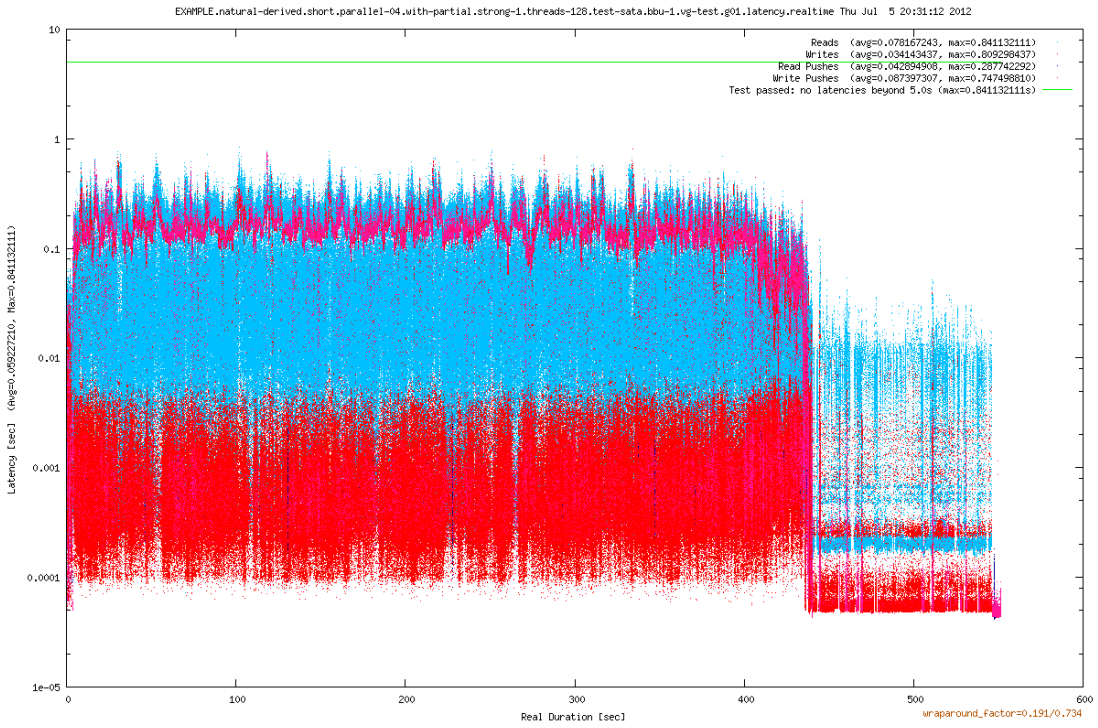
5.2.3. Influence of BBU units at RAID controllers

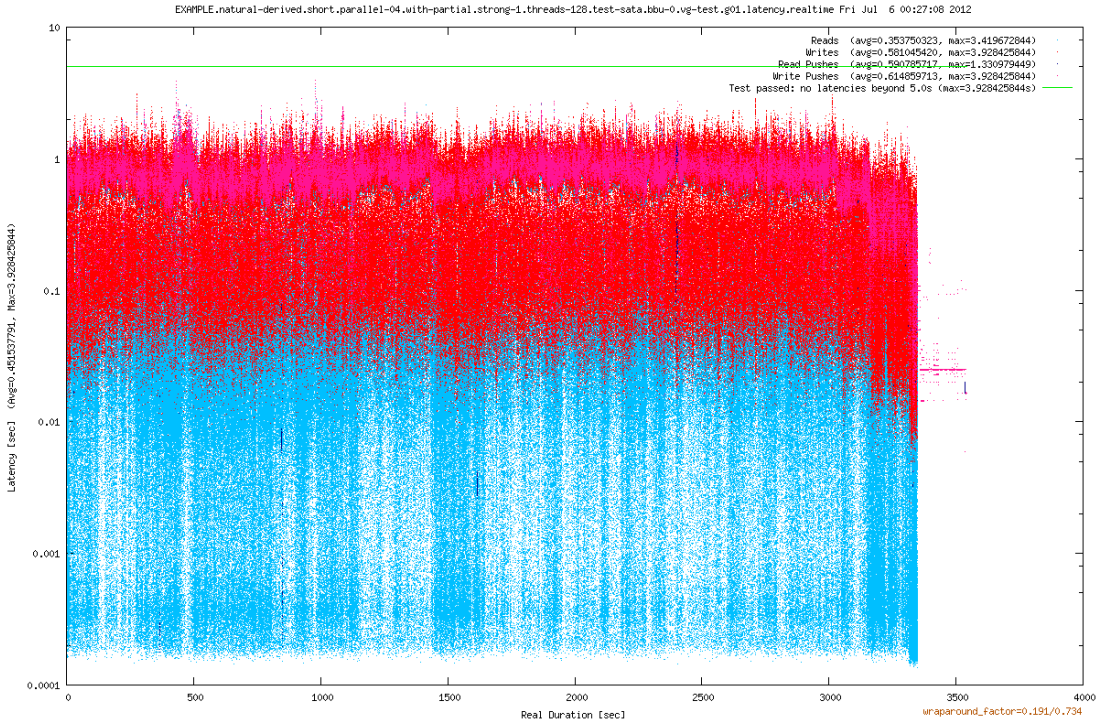
The following comparison is an example for the performace degradation resulting from defective (or missing) BBU units:





In essence, the missing BBU will slow down write requests. This can be easily seen when comparing the sonar diagrams (just look at the colors and where they appear):



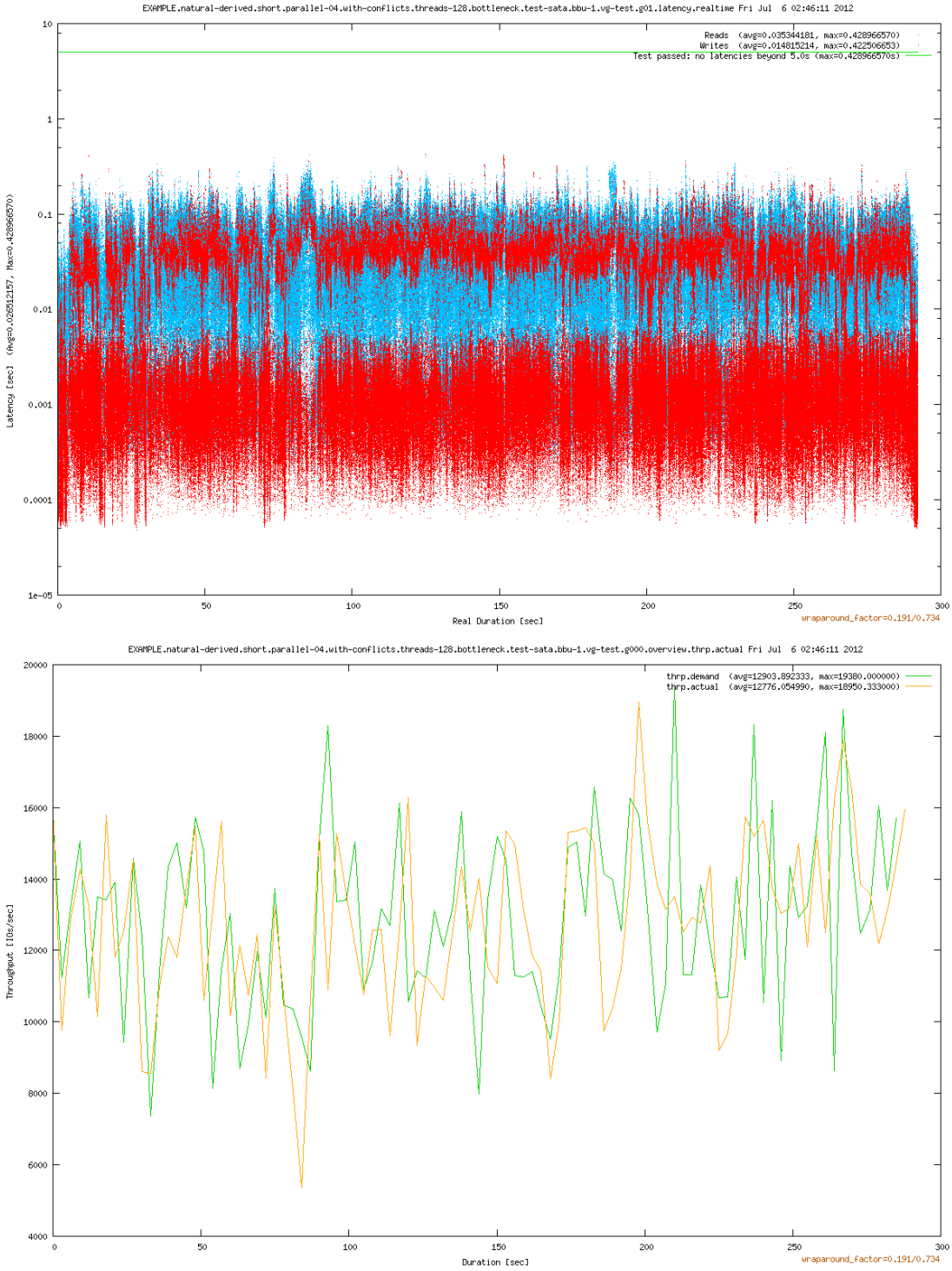


More interesting differences can be found in the tarball at http://www.blkreplay.org/examples/.

5.2.4. Influence of Bottlenecks

The expert option `--bottleneck=<number>` can be used to limit the number of requests submitted to the worker threads, which is usually higher than the number of threads. In each individual pipeline connecting the main thread to each worker thread, up to 8 requests may be queued in advance.

In the following example using `--with-conflicts` and `--threads=128`, we also limit the number of requests by `--bottleneck=128`, which leads to almost the same effect as the above discussion regarding `--with-ordering`:

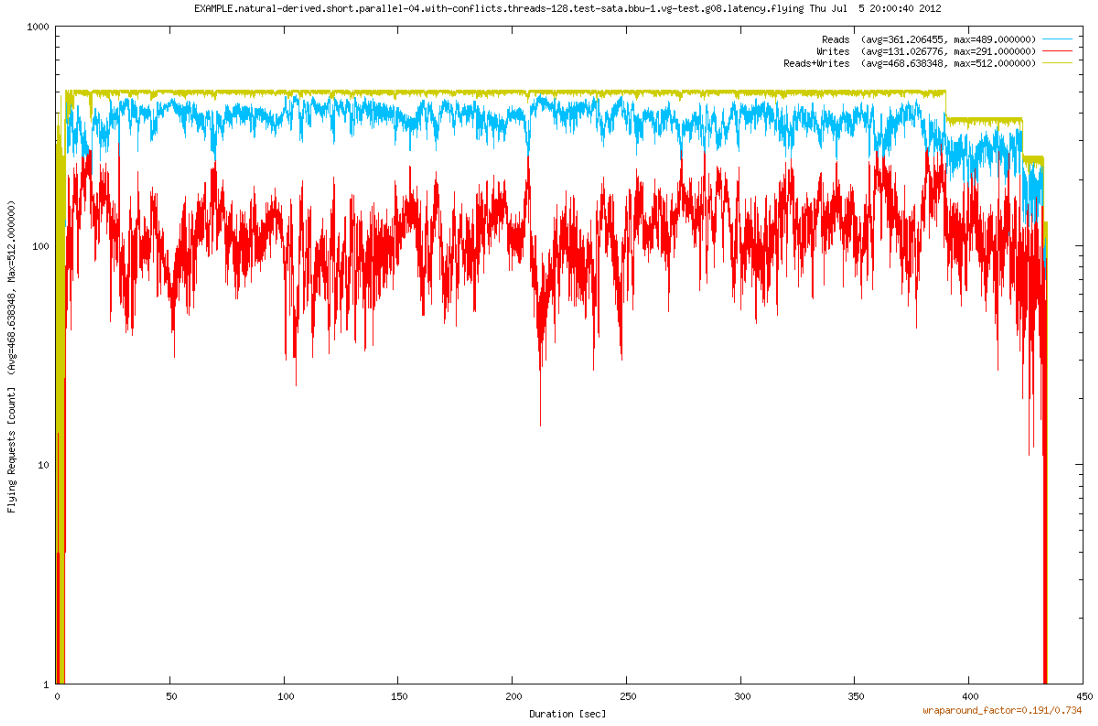


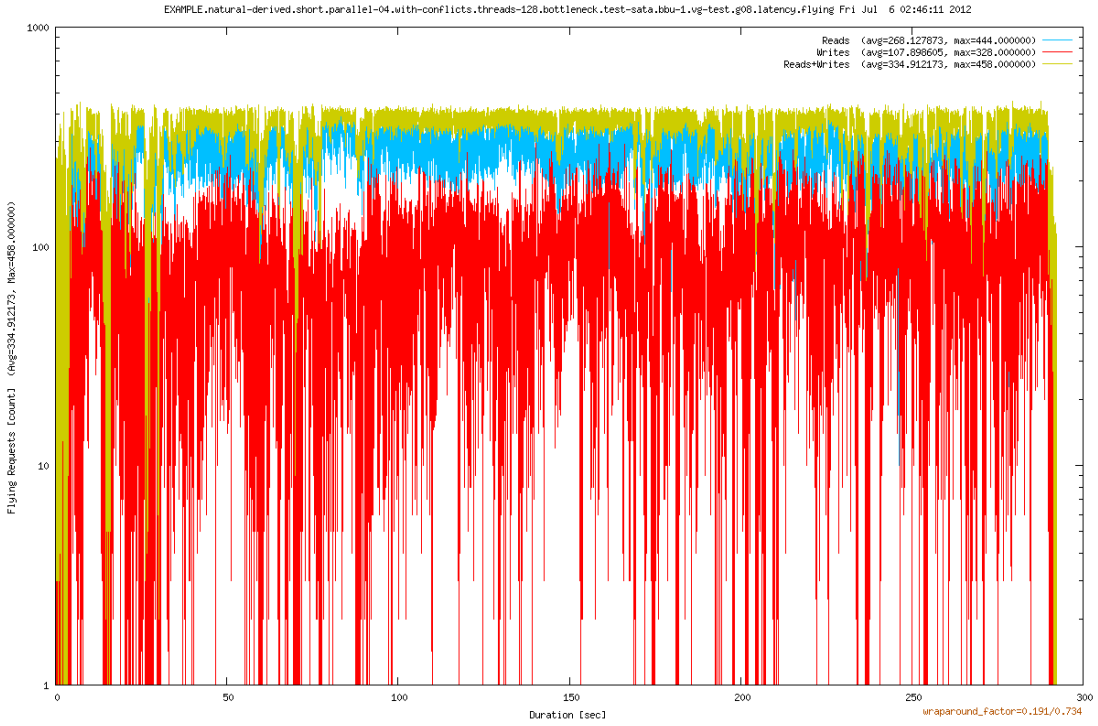
This result is unexpected. In theory, it should not make any difference whether there is 1 request in between the main thread and a worker, or many. However, in practice it makes a difference: the *communication latencies* over the pipelines are avoided when the pipeline remains always filled with some requests in advance.

When `--bottleneck` has exactly the same value as `--threads`, a "logical request" can be either working in a worker thread, or being processed by the main thread, or underway over the pipes. It cannot be present at more than one of these places at the same time.

An example, just for demonstration of the principle: Assume that the IO system can catch up (at least at the beginning), and that the IO latencies are *constant*. Assume that the communication latencies are exactly the same constant. What will be the effect? One half of the requests will always reside in the IO system, but the other half will reside in the pipe communication infrastructure, in average. Did you get the point?

Now, assume that both the IO latencies as well as the communication latencies are varying (e.g. due to kernel scheduling etc). Then the queue length might be staggering around, as observed in the following graphics (first is the default bottleneck setting `--bottleneck=$(( threads * 8 ))`, second is `--bottleneck=${threads}`):



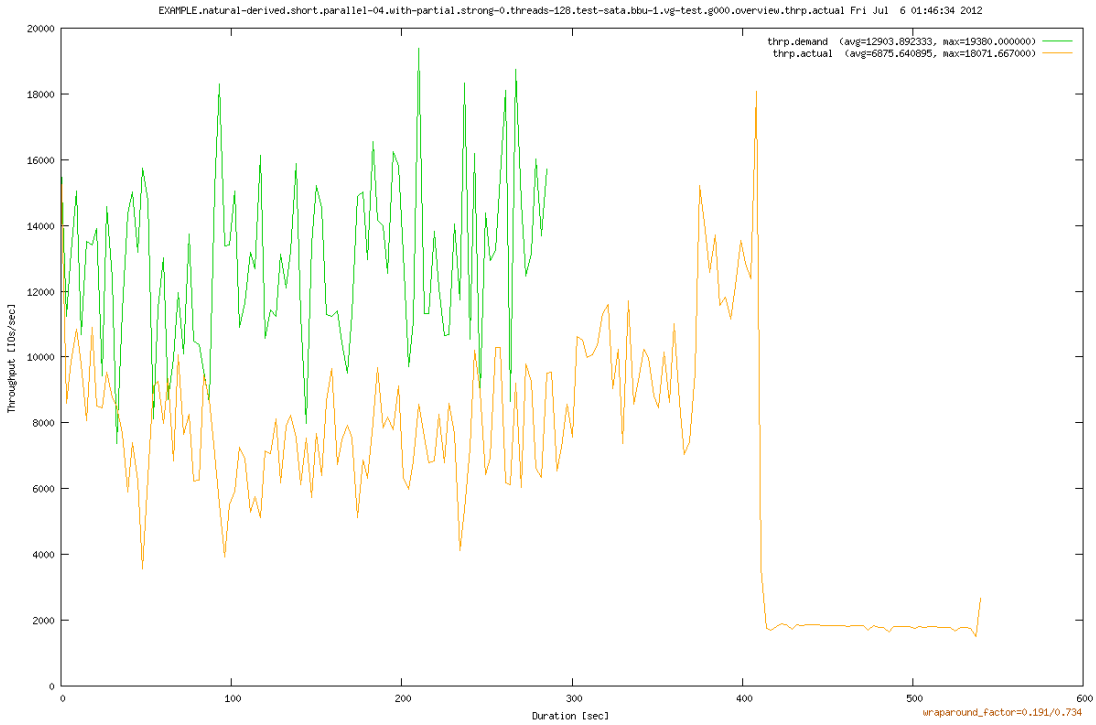


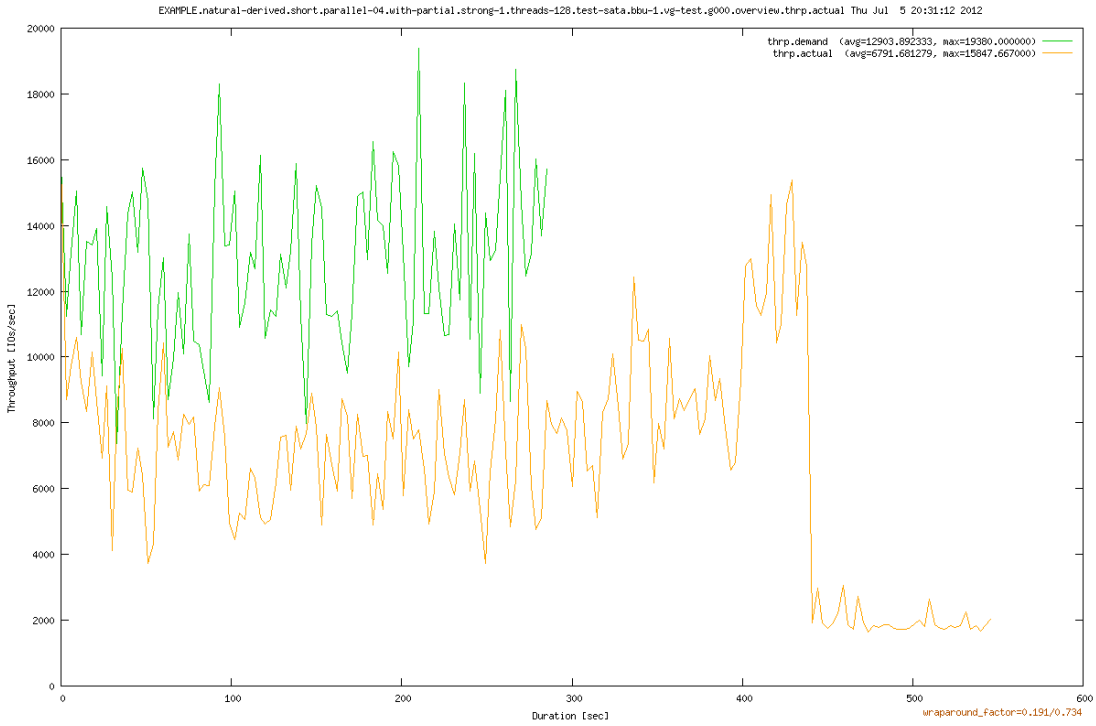
As before, the SATA system shows its counter-productive behaviour when too many IO requests are fired up in parallel. Once the number of requests has reduced, it works better. But somewhen it staggers to the other extreme, flipping around. And to the opposite.

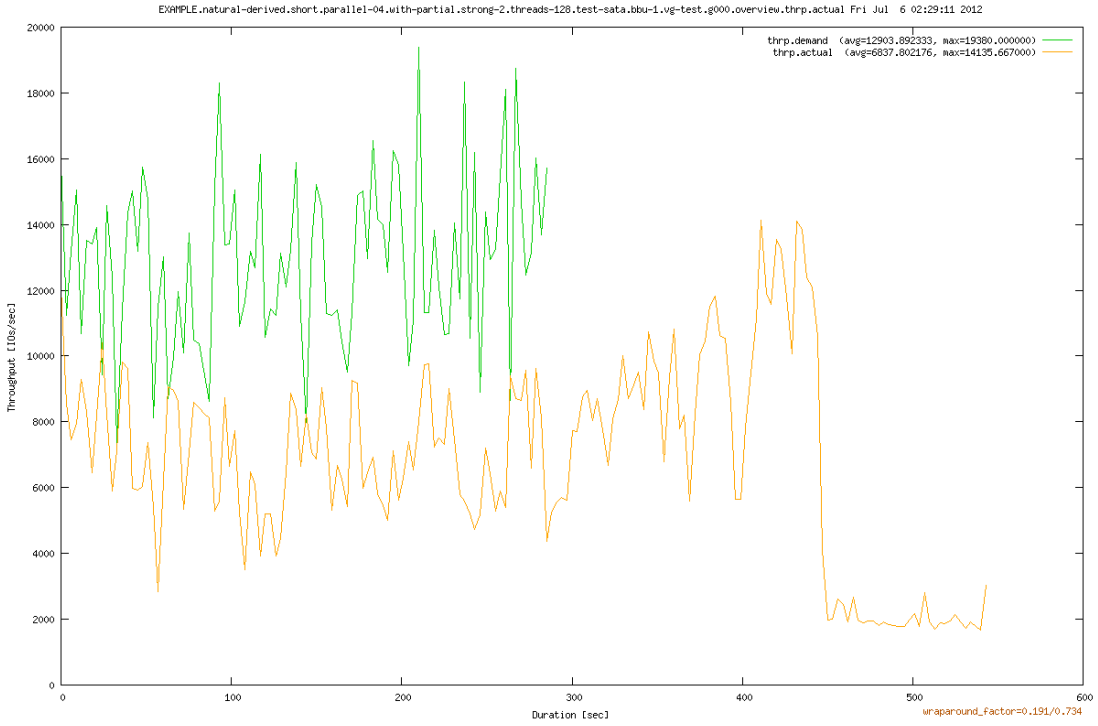
Interesting side note: similar behaviour can be observed in real-life production systems, because natural loads (as opposed to standard benchmark tests) often vary by orders of magnitude. When spuriously plagued by seemingly "unmotivated" incidents, look for places where non-linear and counter-productive behaviour may cause unexpected (and possibly self-amplifying) effects!

5.2.5. Influence of `strong` Mode

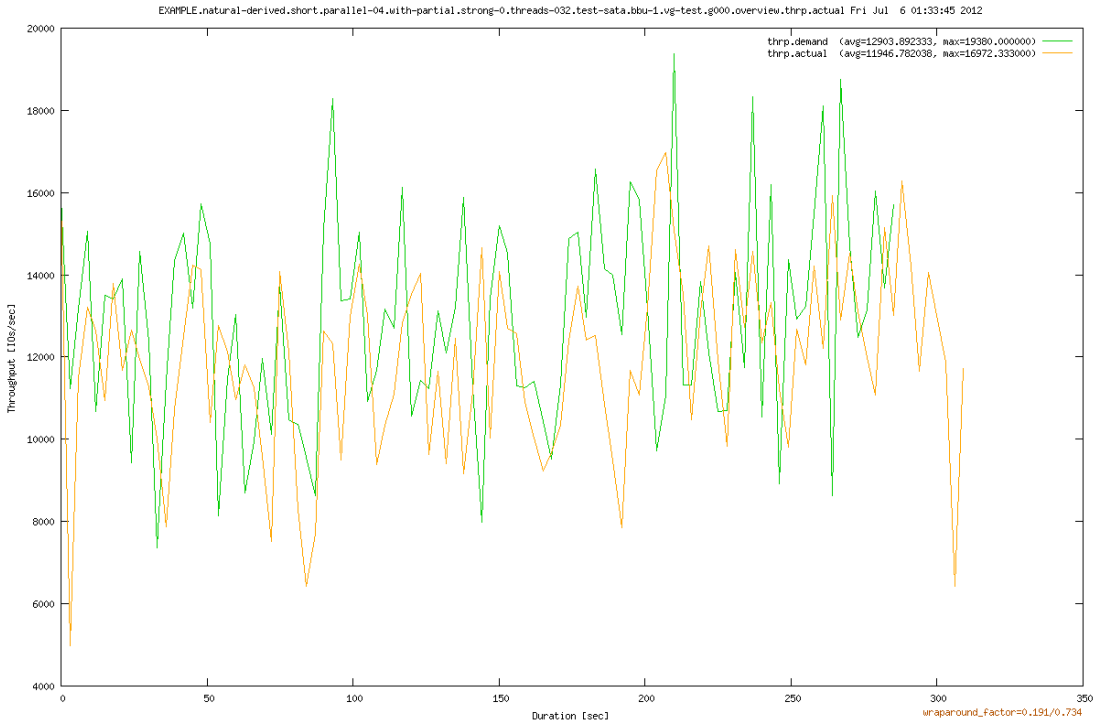
Here is a direct comparison between all three `--strong=` modes, when combined with `--with-partial`. The effect is highly dependable from the load, and usually rather small (when compared to the drastic effects demonstrated in some of the previous sections). But don't take this as given! Better check for it. First, we compare at `--threads=128`:

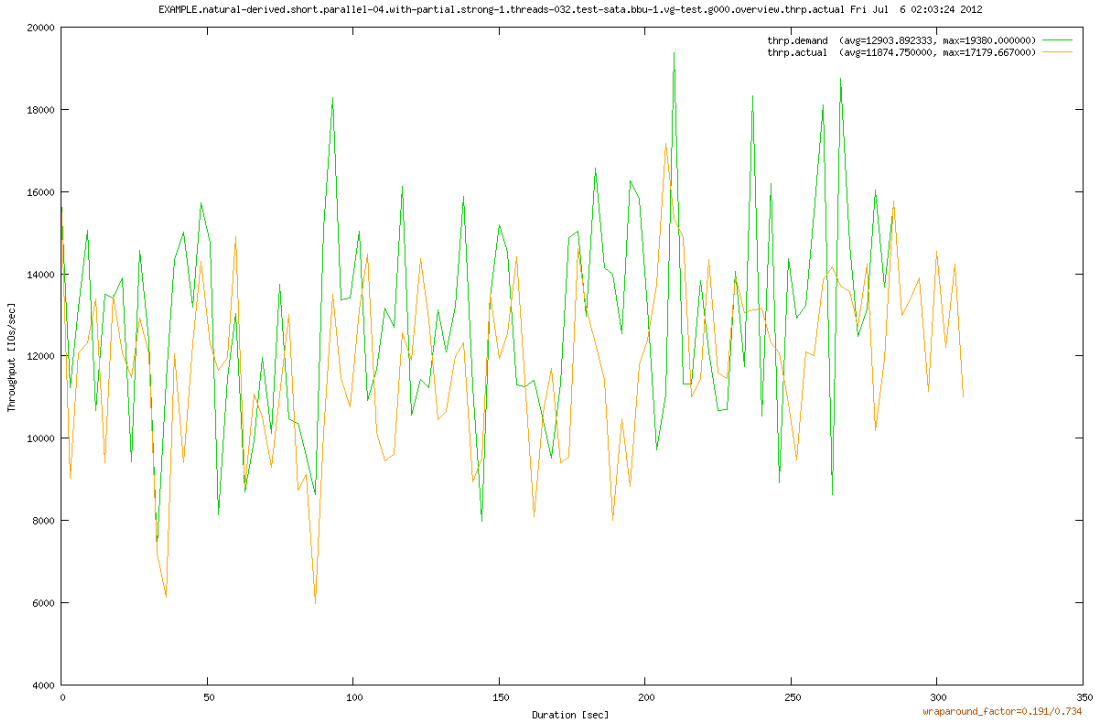


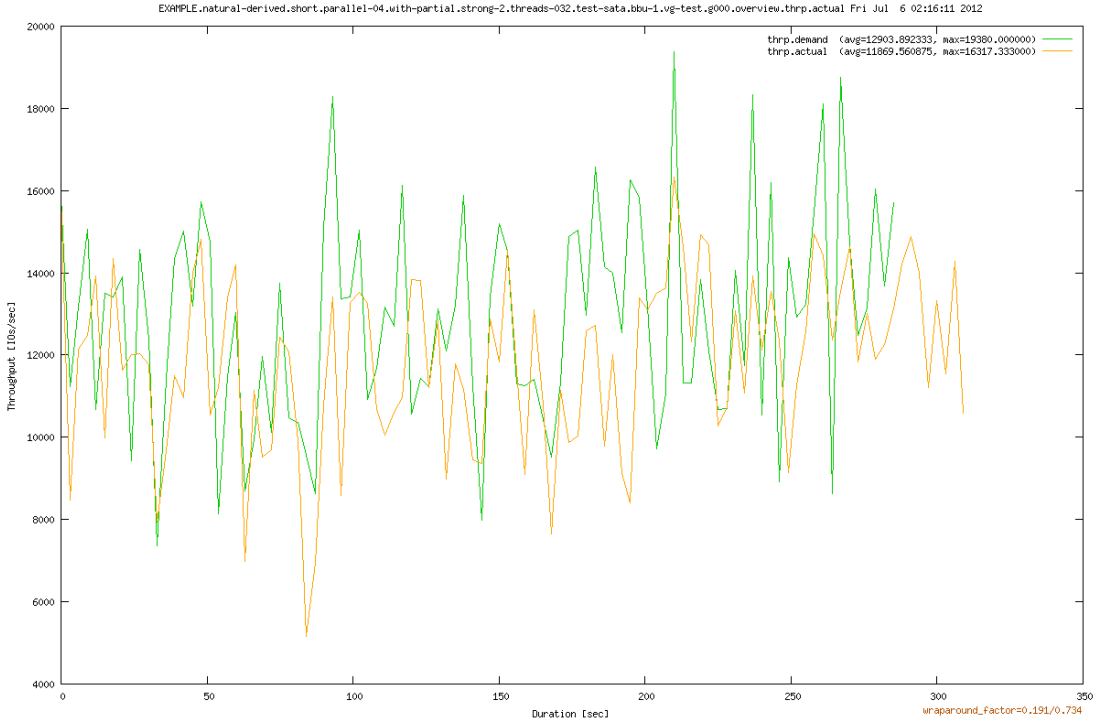




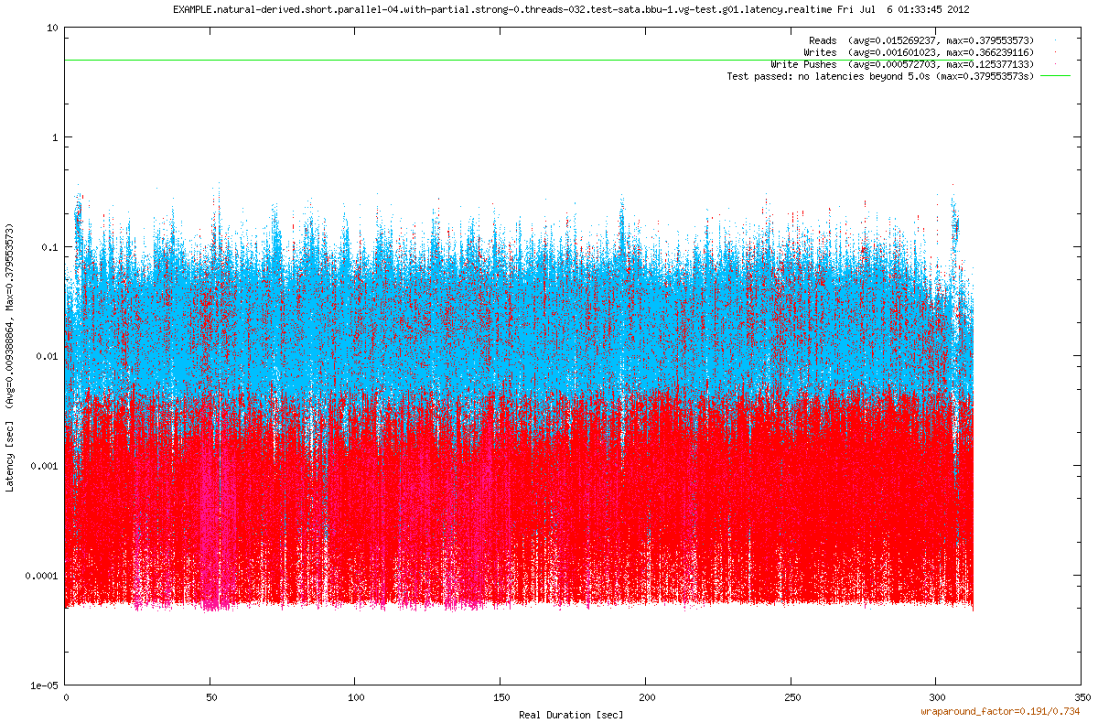
The "long tail" explained in section 5.2.1 is disturbing. The 2-class treatment inherent to `--with-partial` leads to a high number of *transitively* pushed-back requests, racing against the ordinary requests. This race is in some sense "non-deterministic" and may depend non-linearly from the `--strong=` mode. Therefore, we reduce the 2-class treatment by turning to `--threads=32`:

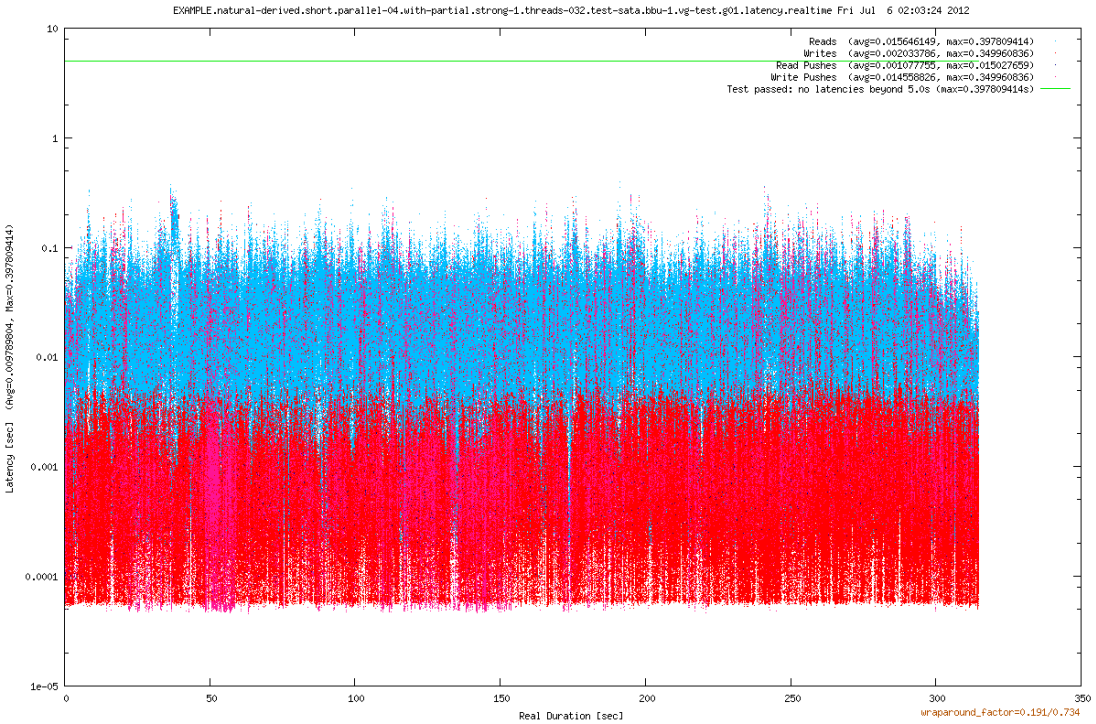


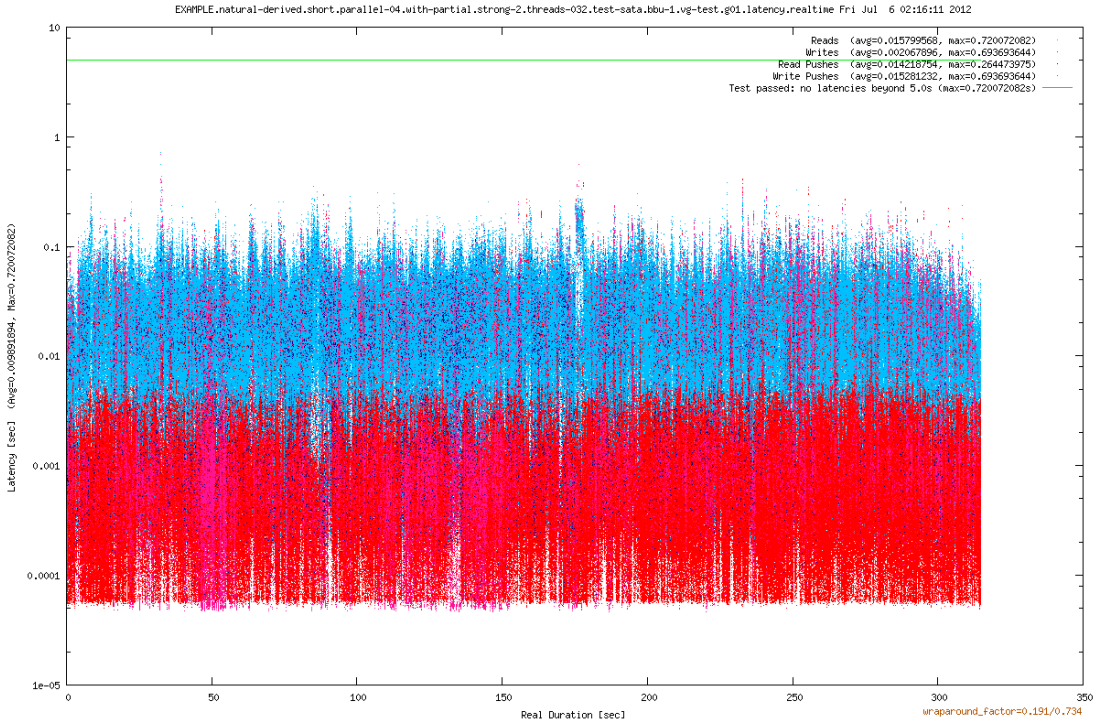




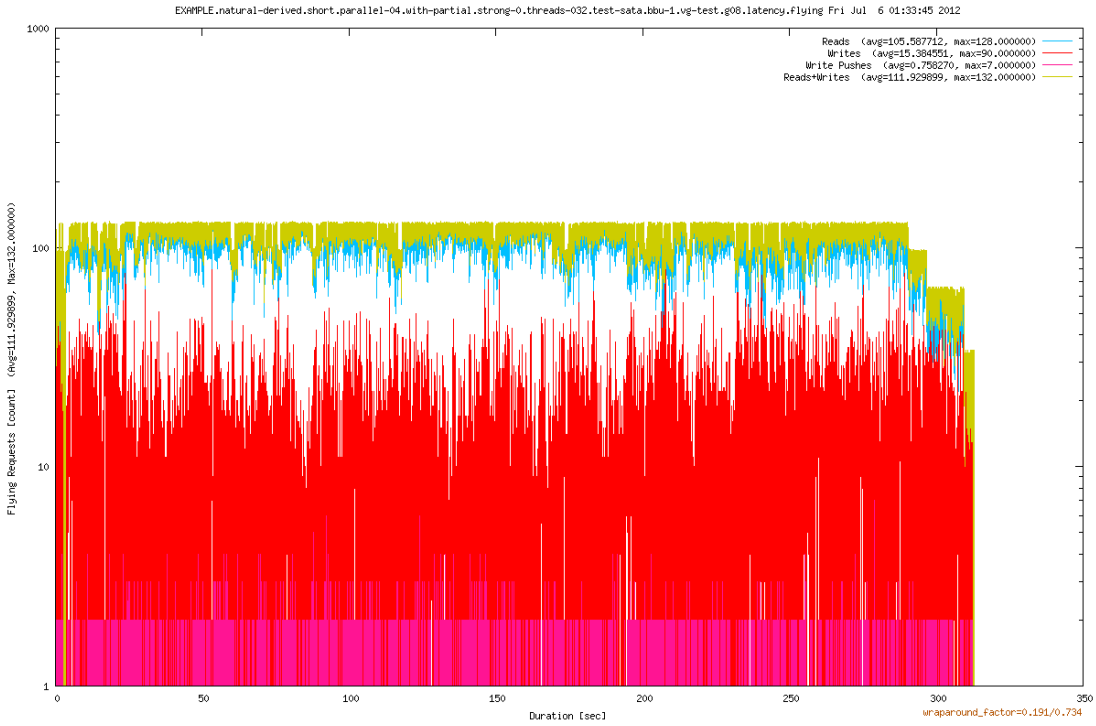
Now the tail effect has almost vanished, because the better throughput leads to less queueing, which in turn decrease the chances for a conflict to actually occur during the time window `--ahead_limit=` (default 1 second). In consequence, any self-amplifying "traffic jam effects" are much lower. As expected, the actual throughput decreases with **`strong`**er mode, but the effect is really tiny, probably below measurement tolerances. In order to really see something, we take a closer look at the sonar diagrams:

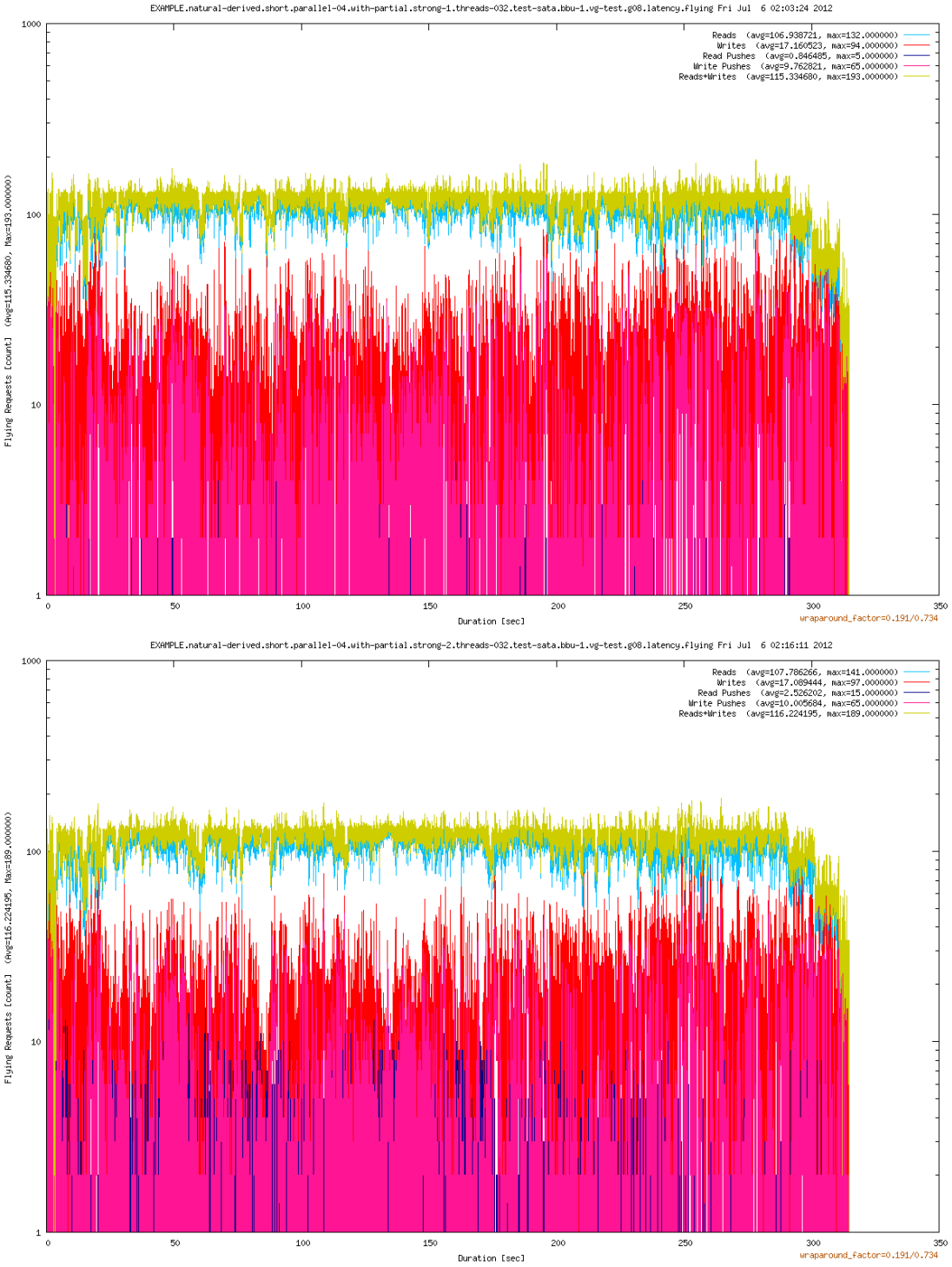

EXAMPLE.natural-derived.short.parallel-04.with-partial.strong-0.threads-032.test-sata.bbu-1.vg-test.g01.latency.realtime Fri Jul 6 01:33:45 2012
Reads (avg=0.015269237, max=0.379553573)
Writes (avg=0.001601023, max=0.366239116)
Write Pushes (avg=0.000572703, max=0.125377133)
Test passed: no latencies beyond 5.0s (max=0.379553573s)
Latency [sec] (Avg=0.009388864, Max=0.379553573)
Real Duration [sec]
wraparound_factor=0.191/0.734


EXAMPLE.natural-derived.short.parallel-04.with-partial.strong-1.threads-032.test-sata.bbu-1.vg-test.g01.latency.realtime Fri Jul 6 02:03:24 2012
Reads (avg=0.015646149, max=0.397809414)
Writes (avg=0.002033786, max=0.349960836)
Read Pushes (avg=0.001077755, max=0.015027659)
Write Pushes (avg=0.014558826, max=0.349960836)
Test passed: no latencies beyond 5.0s (max=0.397809414s)
Latency [sec] (Avg=0.009788804, Max=0.397809414)
Real Duration [sec]
wraparound_factor=0.191/0.734



As expected, the number of `Write Pushbacks` is increasing with `--strong=` mode, and `Read Pushbacks` come also into play. But the differences are minor. A litte more impressive are the differences at `*.latency.flying`, where the "staggering effect" is clearly increasing with growing `--strong=` mode:





More details can be explored when looking at the purple “Write Pushes” / deep-blue “Read Pushes” displayed in further graphics at http://www.blkreplay.org/examples/.

A. Config File Parameters

A.1. Basic Parameters

A.1.1. File `user_modules.conf`:

example–run/user_modules.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#####################################################################

## here you can add your own (local) modules.

# user_module_dir="/path/to/my/directory"
```

A.1.2. File `default-main.conf`:

example–run/default–main.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#####################################################################

## defaults for standalone tests (without modules)

## replay_host_list
##
## Whitespace-separated list of hostnames where blkreplay is run in parallel.
## Each host must be accessible via ssh, without password prompt.
## You may use advanced shell pattern syntax, such as "myhost{17..23}"

replay_host_list="host672␣host{675..679}"

## replay_device_list
##
## Whitespace-separated list of devices where blkreplay is run in parallel.
## You may use advanced shell pattern syntax, such as "/dev/drbd{0..3}"
##
## Notice: you will get the CARTESIAN PRODUCT of
## replay_host_list x replay_device_list
## i.e. all devices must occur on each host.
##
## If you need asymmetric combinations, you can omit (comment out)
## replay_device_list and instead denote each individual combination
## by the special syntax
## replay_host_list="host1:/dev/device1 host2:/dev/device2"
## (or similar)

replay_device_list="/dev/dm-{7,9}"
```

```
## input_file_list
##
## Whitespace-separated list of *.load.gz files.
## You may use ordinary shell pattern syntax, such
## as "bursts-ultrafine.readwrite.1b.*.load.gz"
## or advanced shell pattern syntax,
## like "bursts-ultrafine.readwrite.1b.{1..9}.load.gz" etc.
##
## If you provide less input files than needed by the cartesian product
## replay_host_list x replay_device_list, the same input files will be
## re-used in a round-robin fashion.
##
## HINT: you can provide an URL for downloading from the internet (even with
    wildcards),
## such as "http://www.blkreplay.org/loads/natural/1and1/misc/statistik.*.load.
    gz"
##
## WARNING! running many hosts on a single input file may lead to
## DISTORTIONS, since all load peaks will occur at the same time,
## and the disk seeks / their distances will be duplicated everywhere
## in exactly the same way. As a workaround, see parameter replay_delta below.
##
## WARNING! Mixing fundamentally different loads can lead to
## unintended results! (if you don't know what you are doing)

input_file_list="${base_dir}/example-load/artificial/bursts-ultrafine.readwrite
    .1b.1.load.gz"

## replay_max_parallelism
##
## When set to values > 0, this reduces the parallelism produced by replay_*
    _list
## to some smaller value.
##
## This is handy for comparisons between different degrees of replay parallelism
    ,
## without need for reconfiguring the host / devices list each time.

replay_max_parallelism=0 ________# means unlimited / only limited by replay_*
    _list

## output_label
##
## All output files are prefixed with this name.
## Useful for general description of projects etc.

output_label="MYPROJECT"

## output_add_*
##
## When set to 1 or 0, the output filename will (or will not) contain the
## corresponding information.
## Useful for detailed description of your results.
##
## WARNING! disabling the hostname / device can lead to name clashes
## (mutual overwrites) if you start a replay on multiple hosts
## (and/or on multiple devices) in parallel.

output_add_path=1   # the relative path of the leaf directory is added
output_add_host=1   # add hostname where this blkreplay instance is running
output_add_device=1 # add devicename where this blkreplay instance is running
output_add_input=0  # add the input file name

## replay_start
##
## Starting offset in the input file(s), measured in seconds.
## Often this is 0.
## Can be used to "zoom into" any "time window" in the input files
## (when combined with replay_duration)
##
## Notice: this is _uniformly_ for all input files. If you need
## indiviual time windows from each input file, just create specialized
## input files, e.g. using standard Unix tools like head(1) / tail(1) /
```

```
## awk(1) / perl / gzip etc.

replay_start=0

## replay_duration
##
## One of the most important parameters, measured in seconds.
##
## Please read the warnings in the documentation about unexpected
## effects of storage virtualization layers, caches etc when this
## parameter is too short.

replay_duration=3600

## replay_delta
##
## As said above, replaying the same input file many times in parallel
## can lead to unintended distortions. Often, you don't have enough
## independent input files to achieve high replay parallelism.
## As a workaround, you may "move on" the time window by the
## distance $replay_delta, i.e. the next host will replay a
## "later" portion of the same input file. Although this is worse
## than having completely independent / uncorrolated input files,
## this is by far better than "common mode".
##
## Warning! please check the length of your input file, whether
## (replay_start + replay_duration + n * replay_delta) fits into
## the total length. Otherwise, your load will be silently lower
## than intended.
##
## Hint: when needed, replay_delta should be as high as possible, in
## order to avoid repetition of the same parts over and over again.

replay_delta=0

## threads
##
## Replay parallelism. Must be between 1 and (almost) 65536.
##
## Typically, the number of threads is limiting the number of
## outstanding IO requests [aka "request queue depth"].
##
## Many people believe "the higher, the better". However,
## beware of hidden overheads (see PDF paper).
##
## Higher numbers will not always lead to better throughput:
## some devices / drivers / IO schedulers will even slow down when
## hammered with too many requests in parallel. Some of these
## bottlenecks may vary with the kernel version (see paper).
##
## Attention! Never pick this parameter "out of thin air".
##
## In order to approximate real life behaviour, you should DEFINITELY
## consider the ACTUAL threading behaviour of your application
## and try to approximate it!
##
## Using a totally different threads= parameter than occurring in
## practice / in YOUR accplication can easily lead to high distortions,
## and even to completely worthless FAKE RESULTS!
##
## If you have a "fork bomb" like Apache, use a high number of threads.
## Typically, a database like mysql has a relatively low number of threads.

threads=512

## strong
##
## Conflict handling. Determines the STORAGE SEMANTICS. Details see PDF doc.
##
## One of following conflict tables is used, depending on this setting:
##
## strong=0 :
##
```

```
##    conflict? | R | W |
##           ---+---+---+
##           R  | - | - |
##           W  | - | y |
##
## strong=1 :
##
##    conflict? | R | W |
##           ---+---+---+
##           R  | - | y |
##           W  | y | y |
##
## strong=2 :
##
##    conflict? | R | W |
##           ---+---+---+
##           R  | y | y |
##           W  | y | y |
##
```

strong=1

```
## cmode
##
## Shorthand for "conflict mode".
## See section about both timely and positionly overlapping
## in the PDF paper, aka "damaged IO".
##
## Following values are possible:
##
## with-conflicts:
##   No countermeasures against damaged IO are taken.
##   This can lead to the highest possible throughput, but your device
##   may behave incorrectly.
##
## with-drop:
##   In case of damaged IO, any conflicting requests are just dropped.
##   [conflicts are determined by the above strong= parameter]
##   This will minimize artificial delays, but at the cost of some
##   distortions from missing requests.
##
## with-partial:
##   In case of damaged IO, the conflicting requests are pushed back
##   to an internal pushback list and re-activated as soon as possible.
##   This gives the best possible throughput while avoiding artificial
##   delays as well as damaged IO.
##
##   Attention! this may INCREASE the IO parallelism [request queue depth]
##   because additional request slots must be reserved in advance
##   (see paper).
##
##   In addition, this may lead to a 2-class treatment of requests, because
##   the pushed-back requests have less chances and are queued more
##   intensively than ordinary requests. If you want to reveal whether your
##   test candidate already has some internal 2-class treatment, don't
##   use this mode (otherwise you cannot distinguish the reasons easily).
##
## with-ordering:
##   In case of damaged IO, the conflicting requests (as well as
##   any later requests) will be delayed until the conflict has gone.
##
##   This can lead to ARTIFICIAL DELAYS, because all following requests
##   are delayed as well.
##
##   This mode is useful for detection of massive problems in the hardware.
##   It is also useful for detection of 2-class request treatment in your
##   test candidate, because the artificial delays are "fairer" than
##   with-partial.
##
##   Since the micro-stalls introduced by this mode are more approximate
##   to "request queue staggering" occurring in practice, this mode
##   is also useful for avoidance of some distortions caused by overload.
##   In some cases, the artificial delays caused by this mode are even
```

```
##   BENEFICIAL!

cmode=with-partial

## vmode
##
## Shorthand for "verify mode".
##
## WARNING! switching on verify can lead to serious performance degradation
## (i.e. blkreplay itself may become a bottleneck)
##
## During verify mode, some temporary files
## /tmp/blkreplay.$$/{verify,completion}_table
## are used to store version information about written data.
##
## Depending on the size of the device, this can take considerable space.
## Depending on workingset behaviour, accesses to those temporary files
## can slow down blkreplay considerably (due to additional IO).
##
## Don't use verify mode for benchmarks!
## Use it only for checking / validation!
##
## Following values are possible:
##
## no-overhead:
##   No checks are done. No overhead.
##
## with-verify:
##   Whenver a sector is read which has been written before (some time
##   ago), the sector header is checked for any violations of the
##   storage semantics.
##
## with-final-verify:
##   In addition to with-verify, at the end all touched sector are
##   separately re-read and checked for any mismatches.
##
## with-paranoia:
##   Like with-final-verify, but in addition _all_ written sectors will
##   be _immediately_ re-read and checked.
##   This leads to high distortions of measurements results (because it
##   doubles the IO rates for all writes), but is useful to
##   check the storage semantics even more thoroughly.

vmode=no-overhead

## verbose_script
##
## When set to 1, make shell output more speaking.

verbose_script=0

## verbose
##
## When set to 1, make blkreplay output more speaking.

verbose=0

## start_grace
##
## Over slow networks, it may take some time until all pipes / buffers
## are filled. If the benchmark starts too early, artificial delays
## could result (because the "supply chain" is too slow).
##
## The real starting time of the benchmark will be after this
## (configurable) grace period. Defaults to 15s.
##
## You may need this also in case of laptop disks with a long spin-up
## time.

#start_grace=15

########################################################################
```

```
## some advanced parameters (experts only)

#replay_out_start=""
#omit_tmp_cleanup=0
#enable_compress_ssh=2

#no_o_direct=0 # extremely dangerous, leads to FAKE results! read the docs!
#o_sync=0      # leads to distortions
#speedup=1.0   # leads to heavy distortions of NATURAL loads
#dry_run=0
#fake_io=0
#ahead_limit=1.000000000
#simulate_io=0.001000000
#fan_out=8
#no_dispatcher=0
#bottleneck=0
```

A.2. Ordinary Module Parameters

The following list is in the order of activation.

A.2.1. File `default-recreate_lvm.conf`

example–run/default–recreate_lvm.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#####################################################################

## defaults for module recreate_lvm
##
## recreate_lvm: transparently (re)create LVM devices from a
## set of physical volumes.
##

## enable_recreate_lvm
##
## Set to 0 or 1. Enable / disable this module.

enable_recreate_lvm=0

## vg_name
##
## Name of the LVM volume group.
##
## All physical devices supplied in $replay_device_list will be
## used to create a single volume group out of them, on each host
## from $replay_host_list. Only the full cartesian product between
## $replay_host_list and $replay_device_list is supported.
##
## WARNING! any pre-existing logical volumes (LVs) in that
## volume group (VG) will be destroyed, and their data will be lost!
## Their old physical volumes (PVs) are also removed and destroyed!
##
## After creation of the LVs, $replay_device_list will be re-written
## to reflect the list of LVs thereafter.
##
## This means: you just supply the physical devices, the number of LVs
## to create, and the size of each.
## The names of the LVs are then created and maintained automatically for you.
## This can be even combined with the iSCSI modules etc, which will
## automatically take the LVs and work on them, instead of on the PVs.
```

```
vg_name="vg-test"

## lv_count
##
## Number of LVs to create.
##
## This determines the replay parallelism, when not limited
## by $replay_max_parallelism.

lv_count=1

## lv_size
##
## Size of each logical volume. Syntax see "man lvcreate".
##
## Of course, there must be enough space on the physical volumes :)

lv_size=1T

## lvm_striping
##
## Set to 0 or 1. Enable / disable LVM striping over multiple
## physical volumes.
##
## When you have multiple RAID sets, this can _tremendously_ improve
## IO performance!

lvm_striping=1

## lvm_stripesize
##
## For best performace, this should be equal to the RAID stripesize
## at physical level.

lvm_stripesize=64
```

A.2.2. File `default-create_lv.conf`

example–run/default–create_lv.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#####################################################################

## defaults for module create_lv
##
## create_lv: before filling all devices with random data, (re)create your LVs
##
## ATTENTION! Always use this module when running benchmarks on
## virtualized storage!
##
## Read the section about "Pitfalls from Storage Virtualization"
## in https://github.com/schoebel/blkreplay/raw/master/doc/blkreplay.pdf
##
## You *MUST* delete and re-create your LVs at every run on commercial
## storage boxes as well as other virtualized storage!

## enable_create_lv
##
## Set to 0 or 1. Enable / disable this module.

enable_create_lv=0

# implement the next function such that it can be called twice
# (even when no LVs exist any more) without causing errors.
```

```
#
# However, in case of errors just "return 1" to signal it!

function delete_lv
{
    echo "$FUNCNAME deleting all your LVs"
    # ensure that nothing can go wrong any more...
    replay_host_list=""

    # put in your code here
    # ...

    if (( failure )); then
        echo "Sorry. Some devices are left over. Remove them by hand!"
        return 1
    fi

    # success
    return 0
}

# this should also signal success by the return code.

function create_lv
{
    echo "$FUNCNAME creating all your LVs"

    # put in your code here
    # ...

    if (( failure )); then
        echo "Sorry. Setup failed."
        return 1
    fi

    # upon success: tell which devices can be used.
    replay_host_list="yourhost1:/dev/device1 yourhost2:/dev/device2"
    echo "replay_host_list='$replay_host_list'"
    return 0
}
```

A.2.3. File `default-iscsi_target_iet.conf`

example–run/default–iscsi_target_iet.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#########################################################################

## defaults for module iscsi_target_iet
##
## Automatic setup of iet (of course, it must be installed)

## enable_iscsi_target_iet
##
## Set to 0 or 1. Enable / disable this module.

enable_iscsi_target_iet=0

## iscsi_target and iscsi_ip
##
## Host where iet should be configuered.
##
## IMPORTANT!
##
## When you also use the module iscsi_initiator (which is often the case),
```

```
## you will usually configure $iscsi_target and $iscsi_ip _there_, not here.
##
## In order to avoid any mess by doubly overwriting
## these variables, they are commented out here.
## However, you may override them in your own *.conf files when necessary.

#iscsi_target="myserver"
#iscsi_ip=""

## iqn_base
##
## base name for automatic creation of iSCSI IQN identifiers.
##
## This module will automatically create IQNs for you, by taking
## this variable and appending device names (where slashes are
## substituted by underscores).
## When combined with module iscsi_initiator, you will not even
## have to mess with indidvidual IQN names, since the initiators
## will automatically "know" them.

iqn_base="iqn.2000-01.info.test:test"

## replay_host_list and replay_device_list
##
## IMPORTANT!
##
## These variables are usually configured in the main module, or in
## your own *.conf overrides.
##
## The MEANING of devices changes, as soon as you enable this module:
## Now the device names are referring to devices on the $iscsi_ip server!
##
## Don't mess that up.
##
## BEWARE: cartesian combinations of $replay_host_list with $replay_host_list
## will continue to take place.
##
## Thus, it is strongly recommended to use the syntax
## "hostname:devicename" as documented in default-main.conf
## in order to avoid re-using the same device twice (by accident).
```

A.2.4. File `default-iscsi_initiator.conf`

example–run/default–iscsi_initiator.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module iscsi_initiator
##
## Automatic setup of the linux iSCSI initiator (of course, it must be installed
    )

## enable_iscsi_initiator
##
## Set to 0 or 1. Enable / disable this module.

enable_iscsi_initiator=0

## iscsi_target
##
## hostname of the iSCSI target you want to connect with.
##
## Beware of firewall rules: unless you use $iscsi_ip below,
## this must be accessible from all your
```

```
## $replay_host_list machines, as well as from your workstation where
## tree-replay.sh is running. Setup your ssh correctly, such that
## no interactive questions will occur!

iscsi_target="myserver"

## iscsi_ip
##
## deviating IP or hostname of the iSCSI target in case of a different
## eth* interface for an internal storage network.
##
## This is only used for internal connections between the initiators
## and the target.
##
## The syntax ip:portnumber is also supported.
##
## When unset, this defaults to $iscsi_target

#iscsi_ip="10.0.0.1"

## replay_host_list and replay_device_list
##
## IMPORTANT!
##
## These variables are usually configured in the main module, or in
## your own *.conf overrides.
##
## The MEANING of "devices" changes, as soon as you enable this module:
## Now the device names are nothing but IQN names for iSCSI!
##
## Don't mess that up.
##
## BEWARE: cartesian combinations of $replay_host_list with $replay_host_list
## will continue to take place.
##
## Thus, it is strongly recommended to use the syntax
## "hostname:IQN.bla" as documented in default-main.conf
## in order to avoid re-using the same IQN specification twice (by accident).
##
## When combined with module iscsi_target_*, the meaning of the devices
## will change _again_: in this case they will denote devices (not IQNs)
## on $iscsi_ip (see also comments in default-iscsi_target_*.conf).
```

A.2.5. File `default-scheduler.conf`

example–run/default–scheduler.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#####################################################################

## defaults for module scheduler
##
## automatically set the IO scheduler on all relevant machines.

## enable_scheduler
##
## Set to 0 or 1. Enable / disable this module.
##
## WARNING! disabling this module can result in some "random" behaviour,
## depending on what is (accidentally) preset at your hosts.

enable_scheduler=1

## scheduler
##
```

```
## One of the values "noop", "deadline", "anticipatory", "cfq" etc.
## Defaults to "noop".

scheduler="noop"
```

A.2.6. File `default-wipe.conf`

example–run/default–wipe.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module wipe
##
## wipe: fill all devices with random data
##
## ATTENTION! Always use this module when running benchmarks on
## virtualized storage!
##
## Read the section about "Pitfalls from Storage Virtualization"
## in https://github.com/schoebel/blkreplay/raw/master/doc/blkreplay.pdf
##
## HINT: when combined with iSCSI, this is always run over it.
## There are cases (such as XenServer) where some intermediate layers
## _may_ introduce some additional storage virtualization.
##
## enable_wipe
##
## Set to 0 or 1. Enable / disable this module.

enable_wipe=0

## no further options
```

A.2.7. File `default-bbu_megaraid.conf`

example–run/default–wipe.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module wipe
##
## wipe: fill all devices with random data
##
## ATTENTION! Always use this module when running benchmarks on
## virtualized storage!
##
## Read the section about "Pitfalls from Storage Virtualization"
## in https://github.com/schoebel/blkreplay/raw/master/doc/blkreplay.pdf
##
## HINT: when combined with iSCSI, this is always run over it.
## There are cases (such as XenServer) where some intermediate layers
## _may_ introduce some additional storage virtualization.
```

```
##
## enable_wipe
##
## Set to 0 or 1. Enable / disable this module.

enable_wipe=0

## no further options
```

A.2.8. File `default-graph.conf`

example–run/default–graph.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module graph
##
## graph: automatically call graph.sh whenever results have been created.

## enable_graph
##
## Set to 0 or 1. Enable / disable this module.

enable_graph=1

## graph_options
##
## Commandline options for graph.sh. See documentation.

graph_options="--static␣--dynamic"

## sequential_mode
##
## When set to 1, all the gnuplot commands will run sequentially
## instead of in parallel.
##
## Recommended for large *.replay.gz file, since gnuplot may
## eat tons of memory (swapping).

sequential_mode=1

#######################################################################
##
## The following internal variables may be set to override the default values.
## Some of them are gnuplot commands / options. Please consult
## the docs for details.

#picturetype="png" # [ps|jpeg|gif|...]
#pictureoptions="small size 1200,800" # other values: "size 800,600" etc
#bad_latency="5.0" # seconds
#bad_delay="10.0" # seconds
#bad_ignore="1" # the n'th exceeding the limit
#ws_list="000 001 006 060 600" # list of workingset window sizes
#thrp_window=3 # window size for throughput computation
#turn_window=1 # window size for turn computation
#smooth_latency_flying_window=1 # window size for smoothing latency.flying

## colors (associative array)
## Default colors may be overridden.
## Use RGB values like in HTML/CSS

declare -A color

#color[reads]="#00BFFF" # DeepSkyBlue
```

```
#color[writes]="#FF0000" # red
#color[r_push]="#00008B" # DarkBlue
#color[w_push]="#FF1493" # DeepPink
#color[all]="#CDCD00" # yellow3

#color[ok]="#00EE00" # green2
#color[bad]="#A020F0" # purple
#color[border]="#000000" # black
#color[warn]="#B35200"
#color[fake]="#B35200"

#color[demand]="#00CD00" # green3
#color[actual]="#FFA500" # orange1

#color[000]="#EE0000" # red2
#color[001]="#228B22" # ForestGreen
#color[006]="#0000CD" # MediumBlue
#color[060]="#CD950C" # DarkGoldenrod3
```

A.3. Pipe Module Parameters

A.3.1. File `default-pipe_repeat.conf`

example–run/default–pipe_repeat.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module pipe_repeat
##
## pipe_repeat: repeat contents of the *.load.gz input files until forever.
##
## WARNING: always set $replay_duration, otherwise blkreplay will
## run forever.

## enable_pipe_repeat
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_repeat=0

## no further options
```

A.3.2. File `default-pipe_slip.conf`

example–run/default–pipe_slip.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#######################################################################

## defaults for module pipe_slip
##
## pipe_slip: increase the sector numbers after a bunch of operations.

## enable_pipe_slip
```

```
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_slip=0

## pipe_slip_every
##
## After this number of operations, the offset will be increased.

pipe_slip_every=32768

## pipe_slip_increase
##
## Increase the offset (sector#) by this number
## (every $pipe_slip_every steps).

pipe_slip_increase=256
```

A.3.3. File `default-pipe_subst.conf`

example–run/default–pipe_subst.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#########################################################################

## defaults for module pipe_subst
##
## pipe_subst: substitute reads/writes in *.load.gz input pipe.

## enable_pipe_subst
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_subst=0

## pipe_subst_from and pipe_subst_to
##
## Characters to replace. R = read, W = write.

pipe_subst_from="R"
pipe_subst_to="W"
```

A.3.4. File `default-pipe_spread.conf`

example–run/default–pipe_spread.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#########################################################################

## defaults for module pipe_spread
##
## pipe_spread: increase the region where the sector numbers
## are running about.
##
## WARNING! this will not increase the workingset behaviour,
## and has almost no effect on SSDs or virtual storage.
```

```
## However, it _may_ be useful for increasing the average seek distance
## at mechanical hard disks. Check the result!

## enable_pipe_spread
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_spread=0

## pipe_spread_factor
##
## How large the "working area" should get.

pipe_spread_factor=2.0

## pipe_spread_align
##
## Ensure that the result is aligned to these multiples of sectors.
## (every $pipe_spread_every steps).

pipe_spread_align=8 # corresponding to 4k MMU size

## pipe_spread_offset
##
## Additional offset (added to resulting sector#)

pipe_spread_offset=0
```

A.3.5. File `default-pipe_resize.conf`

example–run/default–pipe_resize.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

#########################################################################

## defaults for module pipe_resize
##
## pipe_resize: increase the request size (#sectors)
## by some factor, and/or tie to some limits.

## enable_pipe_resize
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_resize=0

## pipe_resize_factor
##
## How large the "working area" should get.

pipe_resize_factor=2.0

## pipe_resize_{min,max}
##
## Minimum / maximum size of the resulting request.

pipe_resize_min=0
pipe_resize_max=65536
```

A.3.6. File `default-pipe_cmd.conf`

example–run/default–pipe_cmd.conf

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer /  1&1 Internet AG
#
# Copying and distribution of this file, with or without modification,
# are permitted in any medium without royalty provided the copyright
# notice and this notice are preserved.  This file is offered as-is,
# without any warranty.

##############################################################################

## defaults for module pipe_cmd
##
## pipe_cmd: insert arbitrary shell commands into the
## input pipeline.

## enable_pipe_cmd
##
## Set to 0 or 1. Enable / disable this module.

enable_pipe_cmd=0

## pipe_cmd
##
## The command, will be executed in a subshell.

pipe_cmd="cat"
```

B. File Format

The `*.load.gz` and `*.replay.gz` files are in essence compressed CSV files, using ‘‘;’’ as delimiter. However, errors and comments can also occur in the file, violating the classical CSV format.

In order to extract classical CSV, use the following command:

```
zgrep ‘‘;’’ myname.replay.gz | less
```

The first line of the output contains the column names (and could be removed by inserting `| tail -n +2` into the pipeline). In *theory*, you could even use MS Excel to work on the above output. However, in practice the file will be so extremely large that interactive workstation tools will likely start swapping your system to death, or even bombing.

So the usual way to operate on the output is a “pipe and filters style”, using stream-processing tools like `sed`, `awk`, `perl`, and so on.

Both `*.load.gz` and `*.replay.gz` use the same format. However, `*.load.gz` ***must*** be sorted numerically according to the timestamp, while `*.replay.gz` is usually unsorted (roughly showing the order of request completion in most cases, which is different from the submission timestamp ordering). In order to re-use `*.replay.gz` again as input for another run of `blkreplay`, you must sort it with `sort -n`. In some cases, public `blktrace` recordings at `www.blktrace.org/loads/` will contain only constants 0.0 as delays and latencies.

Feel free to filter your results, or create your own artifical loads.
Enjoy.

C. Validation of the `blkreplay` Tool

Some effort has been spent to check that `blkreplay` really produces the intended kind of IO load on the target system, and that its measurement results are likely to be valid. However, there is no guarantee against hidden bugs, misconfigurations, or other types of accidents. See also the disclaimer of the GPL license.

The following tools try the *help* you by making `blkreplay` as transparent to the public as possible, but cannot form an absolute guarantee against all possible sources of errors.

Please report any bugs to the author, or, even better, send patches or git pull requests.

C.1. Running `blktrace` during `blkreplay`

The timeliness of IO requests is already measured and displayed by the `*.latency.*` graphics. The following steps will thus concentrate on *completeness*.

In order to check that all IO requests present in a `*.load.gz` file are actually applied to the test candidate, you can (and should) re-record the IO load during an execution of `blkreplay` via `blktrace`. The following steps are to be done manually:

1. Start `blktrace` manually, as described in chapter 4, *before* you start `tree-replay.sh`. Be sure to catch all involved devices on all involved hosts.
2. Either kill `blktrace` with Ctrl-C *after* tree-replay.sh is done, or use the `-w` option to specify a *strictly longer* (plus some safety margin) recording time.
   Attention! some versions of `blktrace` seem to loose some requests in some buffers, at least under some conditions. Usually the number of lost events is rather low, about 10. Keep that in mind if you later discover some purpoted discrepancies!
3. Copy the resulting `*.blktrace.[0-9]+` file to your workstation.
4. Invoke `/path/to/blkreplay/scripts/check_replay_against_blktrace.sh` *`myresult`*`.replay.gz` *`mytrace-prefix`* (exactly two parameters).
   Hint: the coincidence between *`myresult`*`.replay.gz` and *`myload`*`.load.gz` must be checked separately, using tools like `cut` and `diff`.
5. Enjoy the result in form of a wide side-by-side `diff` output.

Hint: no output means that no differences have been found. The timeliness as well as the order of requests is *not* checked. Any difference means that a request is missing somewhere, or has been altered in some way. Expect some minor differences (usully less than 1%, but sometimes more) from request splitting by some device drivers.

Keep in mind that sometimes `blktrace` looses some events, leading to "false positives". You should take this as a validation of the validation tool ;)

C.2. Verbosity Graphics

Experts only. It is easy to draw wrong conclusions from this.

Enable the following parameters: `verbose=10; enable_compress_ssh=1; graph_options="--static --verbose"` in your respective `*.conf` files.

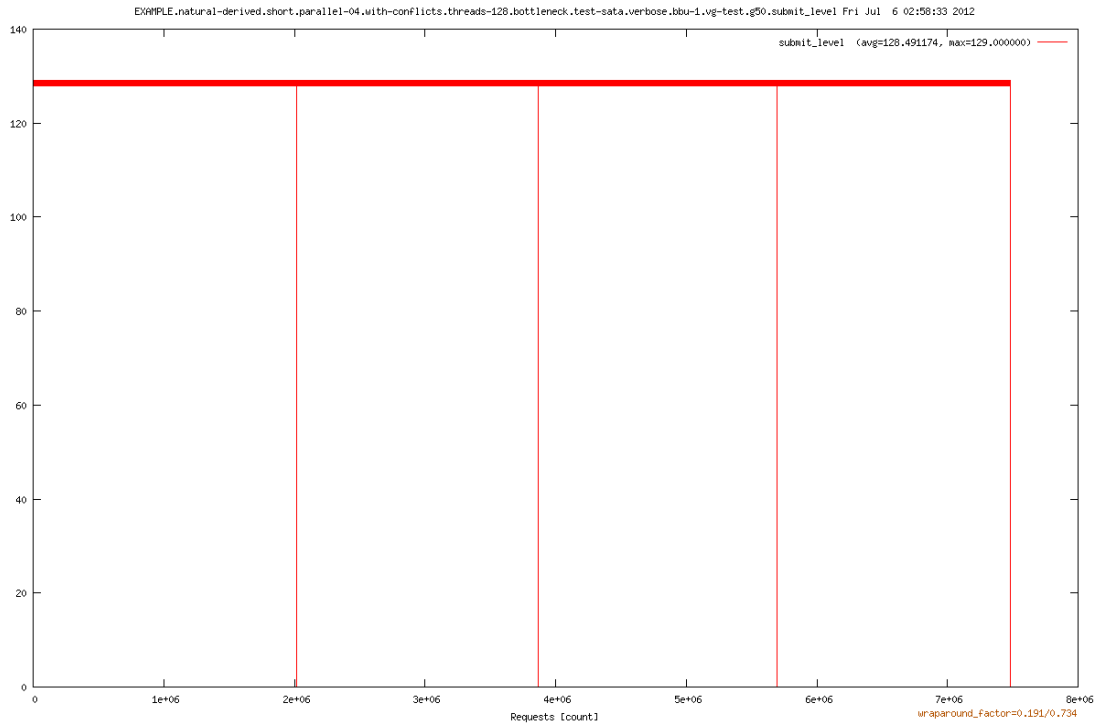
The resulting `*.replay.gz` file will now contain *huge masses* of lines starting with `INFO:` and carrying lots of internal variables, similar to debug / tracing output.

The sheer masses of data written to `stdout` (as such) my lead to distortions of your measurements, e.g. artificial delays or even hangs. Always compare

to `--verbose=0` under otherwise identical conditions, to be sure! In order to check whether `stdout` could have become a bottleneck, please search for the lines with `flush_total=n.nnnnnnnnn` (some timing with nanosecond resolution). In particular, the main thread (additionally indicated by `role='main'`) should have spent much less time in `fflush(stdout)` than total running time. When the network is the bottleneck, `enable_compress_ssh=1` may help; otherwise disabling could be better – please check.

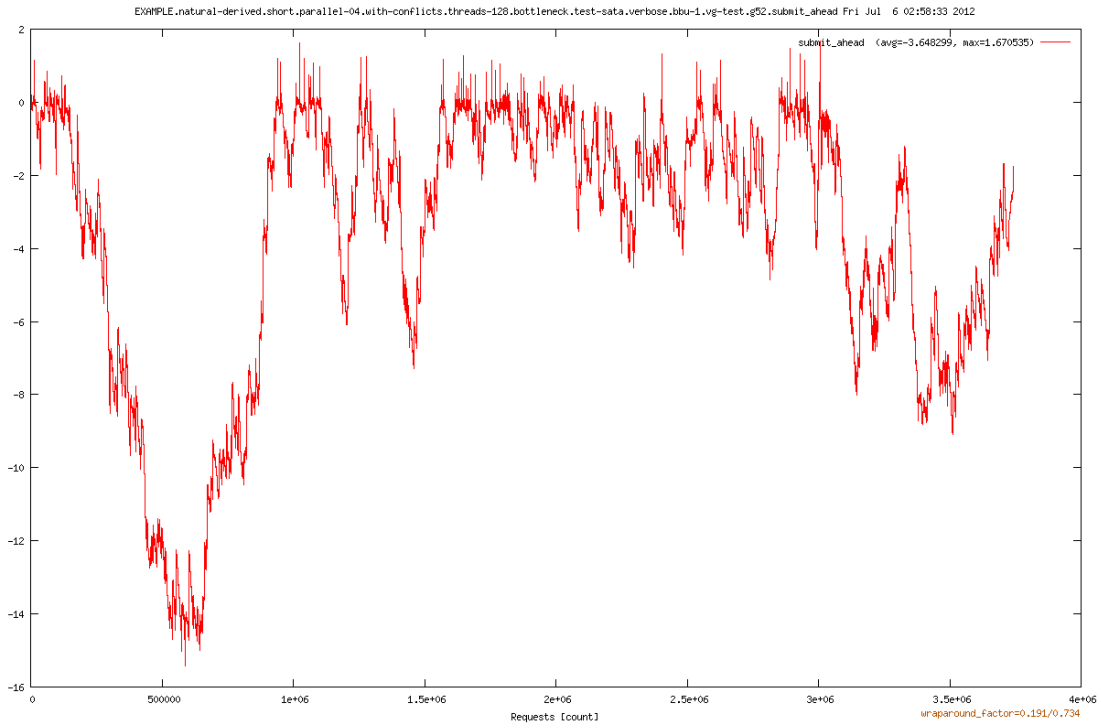
In addition, the following types of internal visualizations will be produced:

`*.submit_level.png`
Shows the actual filling level of the pipelines between the main thread and the workers. The x axis is in units of requests, and in *completion order* (different from the submission order).



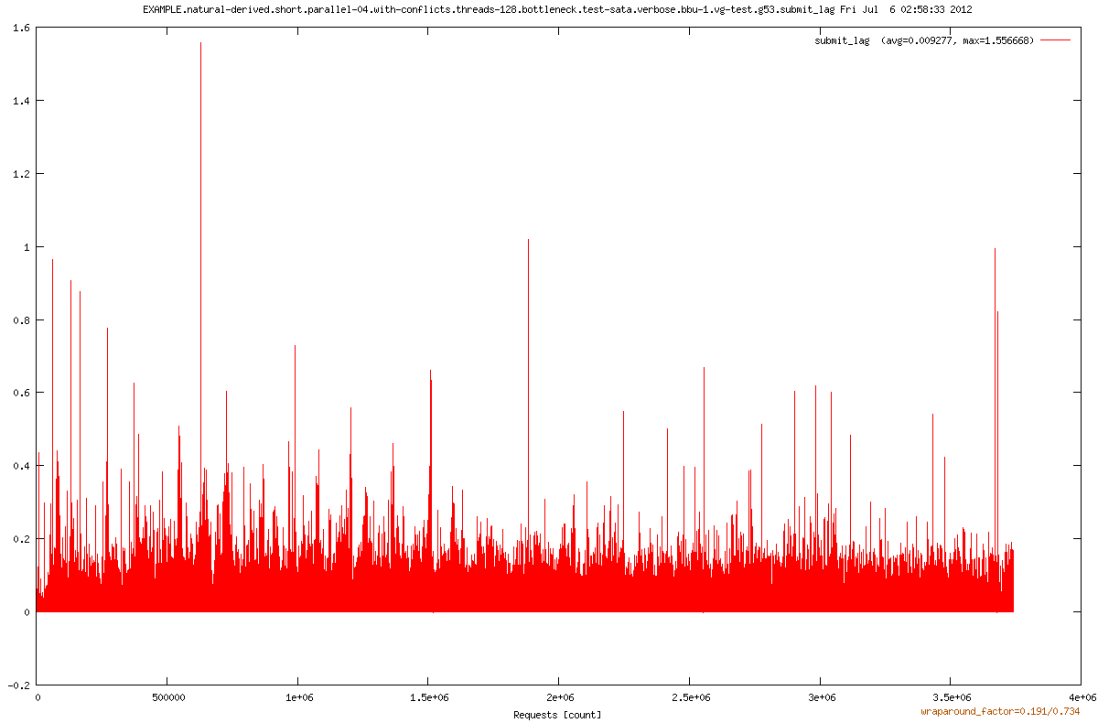
`*.pushback_level.png`
Shows the current number of requests on the pushback list. Only meaningful at `--with-partial`.

`*.submit_ahead.png`
Shows the time difference between submission to the pipelines (which is usually ahead of the execution time, but not always) and the intended execution timestamp.

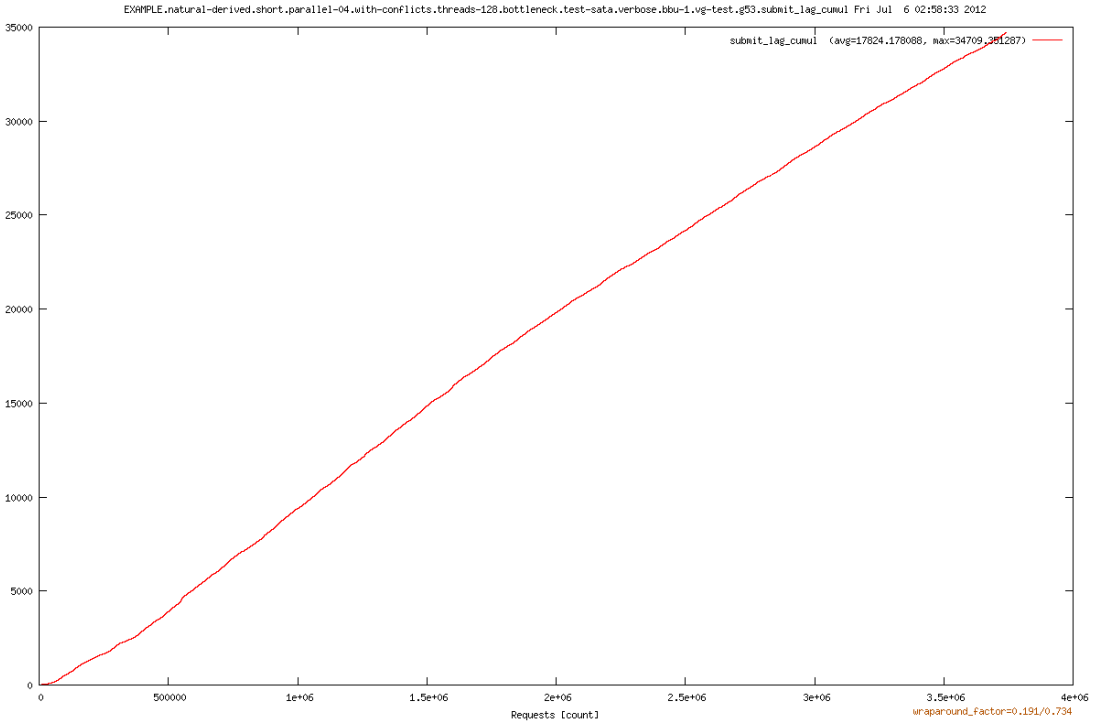


Don't draw wrong conclusions from negative values here! They *can* be a sign of "too late", but the *reason* is almost always in the test candidate, just when ordinary delays are produced.

`*.submit_lag.png`
The time difference between submission to the pipelines, and reception by the worker thread (aka communication latency).

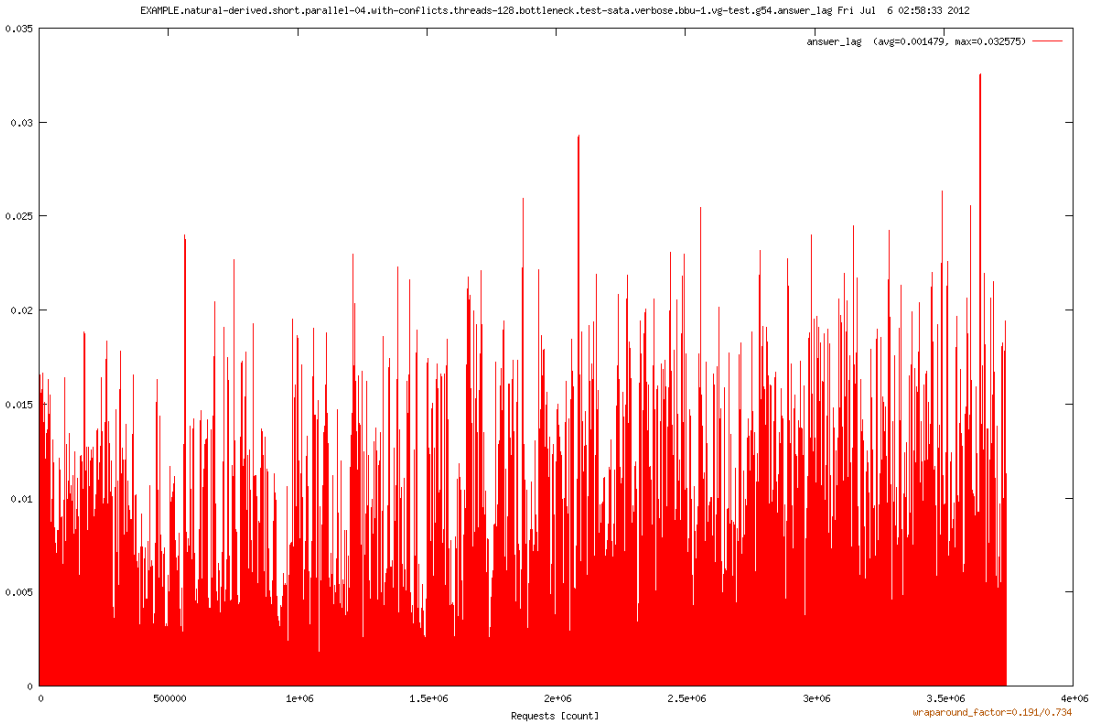


`*.submit_lag_cumul.png`
The same, but cumulized over time.

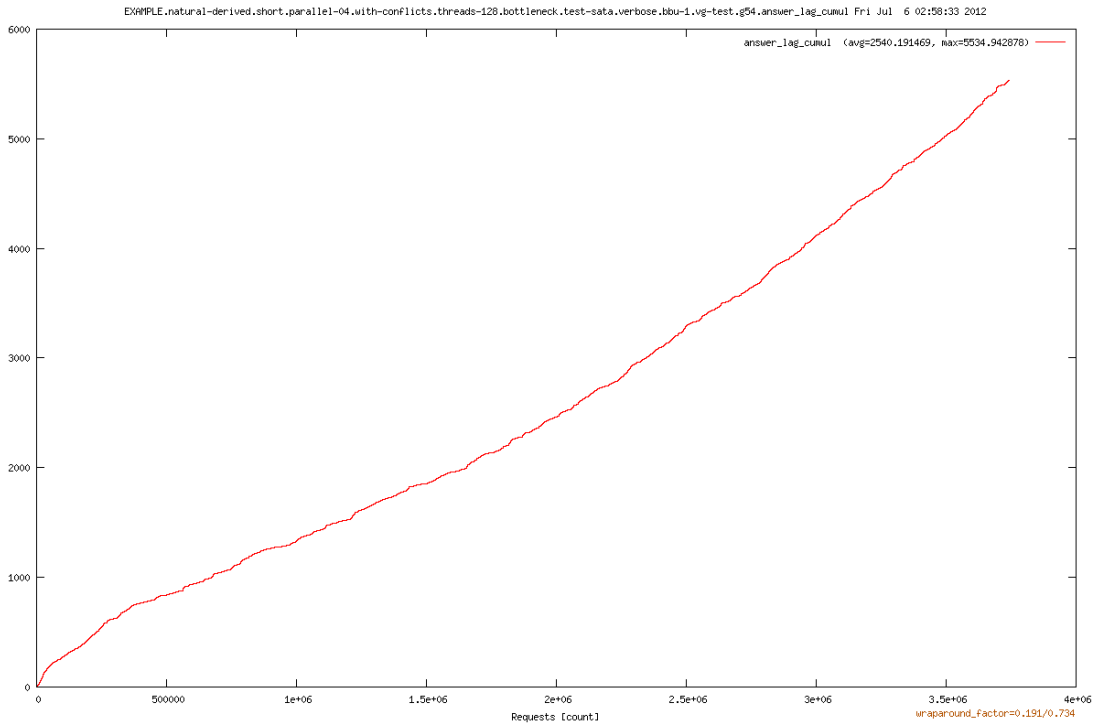


Notice than cumulation of time lags belonging to different threads is "unfair" by concept. Large values can regularly occur because of queueing in the submit pipelines. Complaints about high values make only sense in case of `$(( bottleneck <= threads ))`.

`*.answer_lag.png`
The time difference between submission to the answer pipeline caused by a worker thread, and reception by the main thread (aka communication latency).

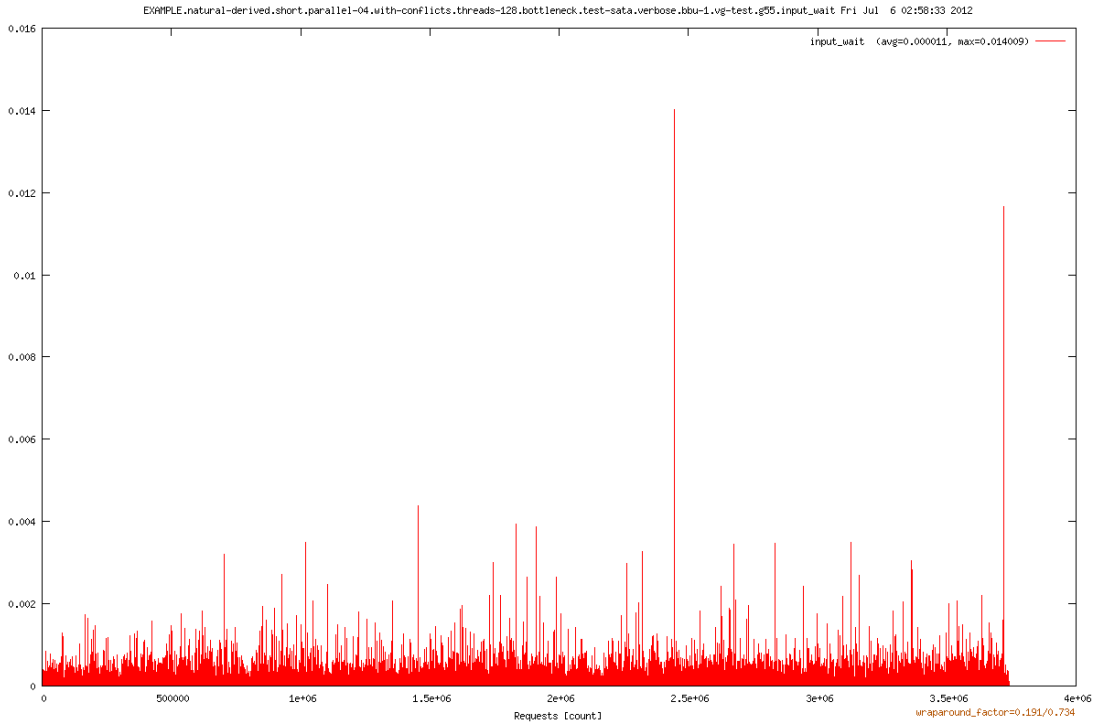


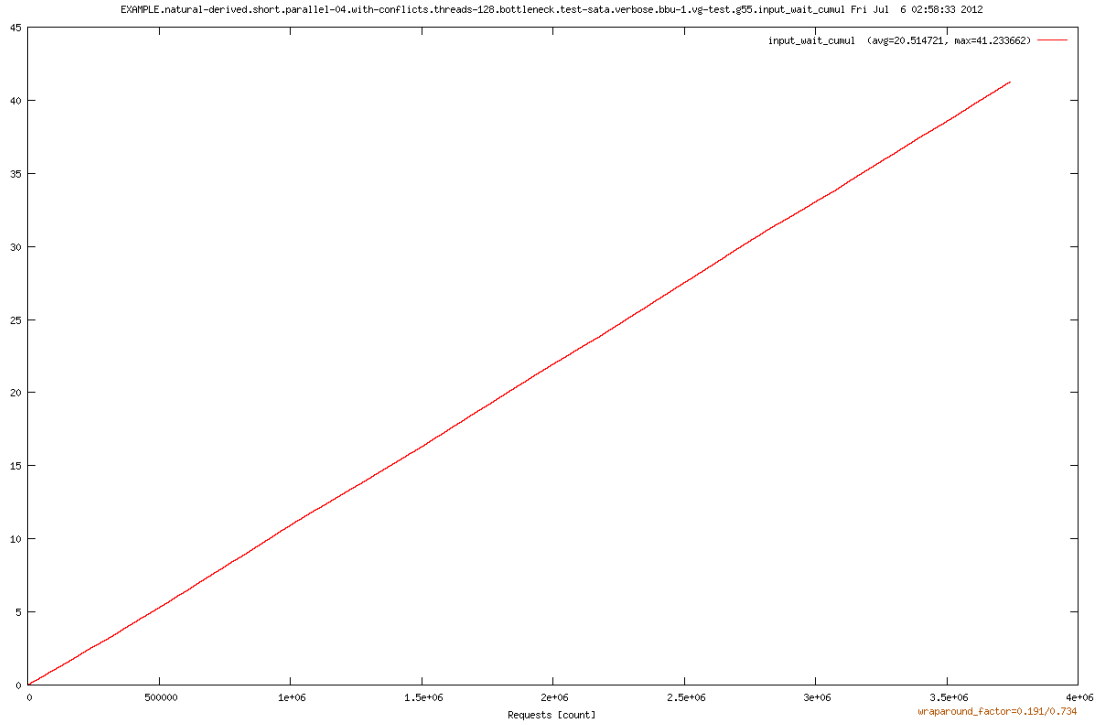
`*.answer_lag_cumul.png`
Dito, cumulated over time.



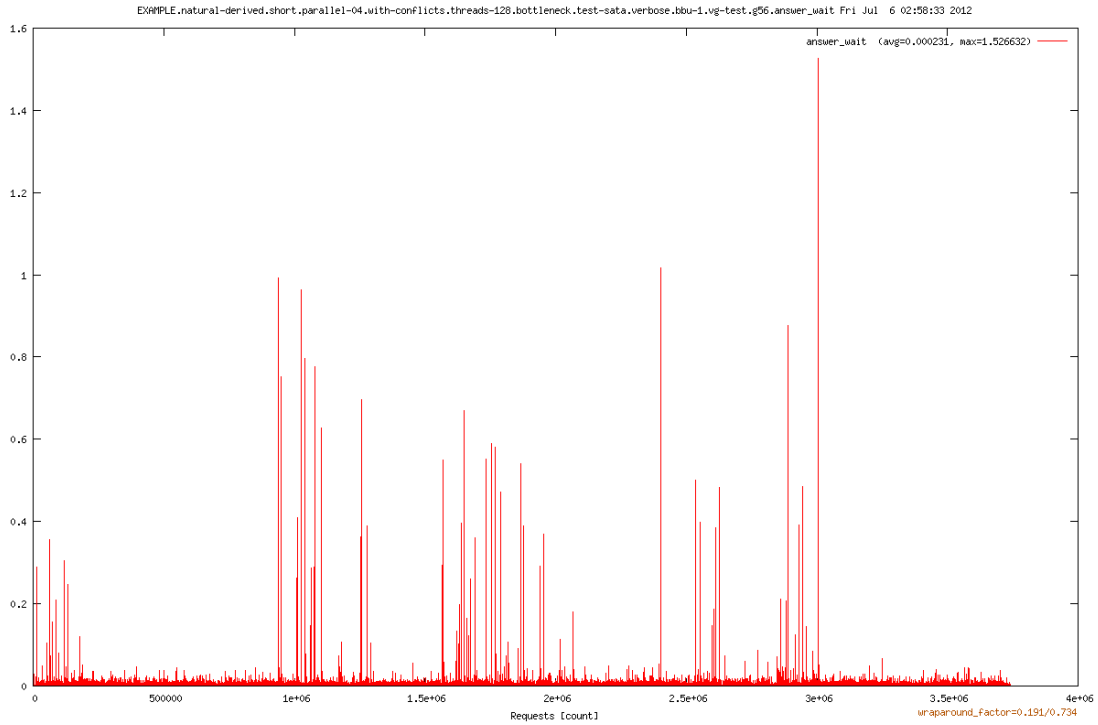
Similar warning as before: high values occur regularly in `--with-ordering` mode. Only be startled in *some* other cases!

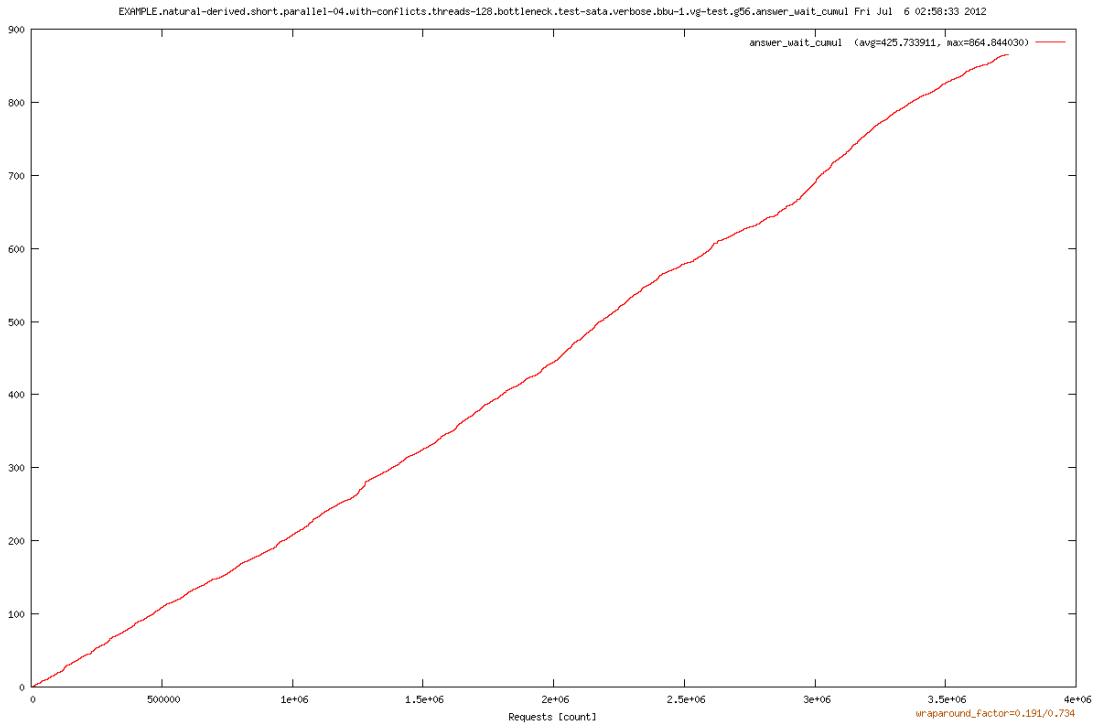
`*.input_wait[_cumul].png`
The time differences before and after `fgets()` (aka stalls in `stdin`). When these numbers get too high, this time you should be alerted quite rightly.





`*.answer_wait[_cumul].png`
The time difference of the main thread it is spending for answer wait. Not to be confused with communication latencies! High number will occur regularly. It is even a sign of good performance when the cumulation almost reaches the total processing time.




EXAMPLE.natural-derived.short.parallel-04.with-conflicts.threads-128.bottleneck.test-sata.verbose.bbu-1.vg-test.g56.answer_wait_cumul Fri Jul 6 02:58:33 2012
answer_wait_cumul (avg=425.733911, max=864.844030)
900
800
700
600
500
400
300
200
100
0
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
Requests [count]
wraparound_factor=0.191/0.734

D. GNU Free Documentation License

fdl.txt

```
                GNU Free Documentation License
                 Version 1.3, 3 November 2008


 Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
     <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
functional and useful document "free" in the sense of freedom: to
assure everyone the effective freedom to copy and redistribute it,
with or without modifying it, either commercially or noncommercially.
Secondarily, this License preserves for the author and publisher a way
to get credit for their work, while not being considered responsible
for modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense.  It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does.  But this License is not limited to software manuals;
it can be used for any textual work, regardless of subject matter or
whether it is published as a printed book.  We recommend this License
principally for works whose purpose is instruction or reference.


1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that
contains a notice placed by the copyright holder saying it can be
distributed under the terms of this License.  Such a notice grants a
world-wide, royalty-free license, unlimited in duration, to use that
work under the conditions stated herein.  The "Document", below,
refers to any such manual or work.  Any member of the public is a
licensee, and is addressed as "you".  You accept the license if you
copy, modify or distribute the work in a way requiring permission
under copyright law.

A "Modified␣Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary␣Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall
subject (or to related matters) and contains nothing that could fall
directly within that overall subject.  (Thus, if the Document is in
part a textbook of mathematics, a Secondary Section may not explain
any mathematics.)  The relationship could be a matter of historical
connection with the subject or with related matters, or of legal,
commercial, philosophical, ethical or political position regarding
them.

The "Invariant␣Sections" are certain Secondary Sections whose titles
```

are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover␣Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title␣Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title␣Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled␣XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve␣the␣Title" of such a section when you modify the Document means that it remains a section "Entitled␣XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add
   to it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page. If
   there is no section Entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on. These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles. Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant. To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version. Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity. If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the
Invariant Sections of all of the original documents, unmodified, and
list them all as Invariant Sections of your combined work in its
license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and
multiple identical Invariant Sections may be replaced with a single
copy. If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"
in the various original documents, forming one section Entitled
"History"; likewise combine any sections Entitled "Acknowledgements",
and any sections Entitled "Dedications". You must delete all sections
Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other
documents released under this License, and replace the individual
copies of this License in the various documents with a single copy
that is included in the collection, provided that you follow the rules
of this License for verbatim copying of each of the documents in all
other respects.

You may extract a single document from such a collection, and
distribute it individually under this License, provided you insert a
copy of this License into the extracted document, and follow this
License in all other respects regarding verbatim copying of that
document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, is called an "aggregate" if the copyright
resulting from the compilation is not used to limit the legal rights
of the compilation's users beyond what the individual works permit.
When the Document is included in an aggregate, this License does not
apply to the other works in the aggregate which are not themselves
derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one half of
the entire aggregate, the Document's Cover Texts may be placed on
covers that bracket the Document within the aggregate, or the
electronic equivalent of covers if the Document is in electronic form.
Otherwise they must appear on printed covers that bracket the whole
aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may
distribute translations of the Document under the terms of section 4.
Replacing Invariant Sections with translations requires special
permission from their copyright holders, but you may include
translations of some or all Invariant Sections in addition to the
original versions of these Invariant Sections. You may include a
translation of this License, and all the license notices in the
Document, and any Warranty Disclaimers, provided that you also include
the original English version of this License and the original versions
of those notices and disclaimers. In case of a disagreement between
the translation and the original version of this License or a notice
or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements",
"Dedications", or "History", the requirement (section 4) to Preserve
its Title (section 1) will typically require changing the actual
title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document
except as expressly provided under this License. Any attempt
otherwise to copy, modify, sublicense, or distribute it is void, and
will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license
from a particular copyright holder is reinstated (a) provisionally,

unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or␣any␣later␣version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive␣Multiauthor␣Collaboration␣Site" (or "MMC␣Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive␣Multiauthor␣Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible␣for␣relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and

```
license notices just after the title page:

    Copyright (c)  YEAR  YOUR NAME.
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
␣␣␣␣Free␣Documentation␣License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```