`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 0.11

Abstract

`blkreplay` is a GPL'ed userspace tool for testing 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` 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 boxes 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 test setup is also handled very carefully[2].

Furthermore, it can be used for tests of the **Linux kernel**, e.g. for testing device drivers, comparing different IO schedulers at different load patterns, determining the overhead of Linux `dm` targets, determining 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.

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.

[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.

2.2. Architecture of `blkreplay`

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

Ordinary production systems like web servers[2] 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.

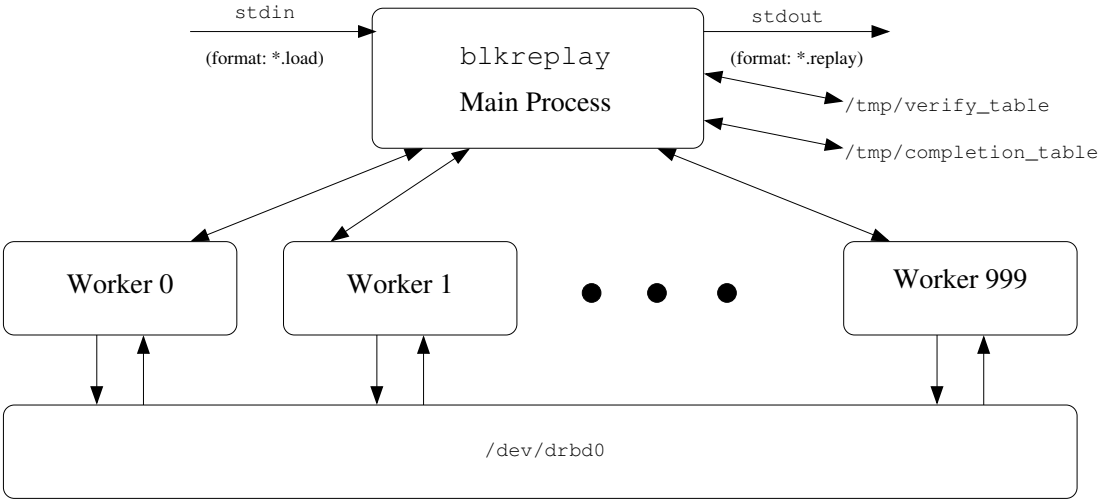
In order to simulate such a behaviour in the lab, there are principally two alternatives:

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).

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

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. Overlapping of IO Requests

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*.

[2] Under high load in datacenters, the behaviour of the `apache` web server may be characterized as similar to a "fork bomb". In contrast to classical toy fork bombs, these are generating heavy IO load in parallel.

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 files changes frequently. Completely unrelated requests (neither timely nor positionly overlapping) are even the most frequent case in most practical load scenarios at production sites. So, what about both kinds of overlapping at the same time?

The case of *both* timely and positionly overlapping (simultanously) 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 working with Direct IO, 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 concurrent reads in parallel to writes to/from the same sector[3].

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 must 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.

[3] In theory, concurrent reads from the same sector would be possible without causing harm to data integrity on the block device. However, 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. Therefore, this case also does not appear in practice.

[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*.

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. There are two modes of operation:

`with-drop` Whenever a new write request is conflicting with an old (already issued) request, it is simply dropped. This has no side effects, other than that some writes may be missing. Depending on properties of the load, the number of dropped writes may be rather high. Please check the end of the result file. In the statistics section, you will find the number of dropped requests. If this number is higher than, say, 5%, you should consider the next option.

`with-ordering` Notice that the main process spreads its IO requests to the worker processes in a round-robin fashion over their anonymous pipes. Whenever conflicting IO requests (in the sense of damaged IO) are detected, this spreading process will stop until the conflict has gone away. As a side effect, all following IO requests are also delayed, even if they don't conflict with anything else.

The latter can lead to *artificial delays*. Future versions of `blkreplay` are planned to optionally act smarter in such situations.

However, practical experience from many hardware tests shows that the current behaviour (artificial delays) seems to be an *advantage*. Whenever such stalls occur more than seldomly, they act as an **indicator** for massive IO problems of the test candidate. Thus we feel that this behaviour is more a feature than a bug.

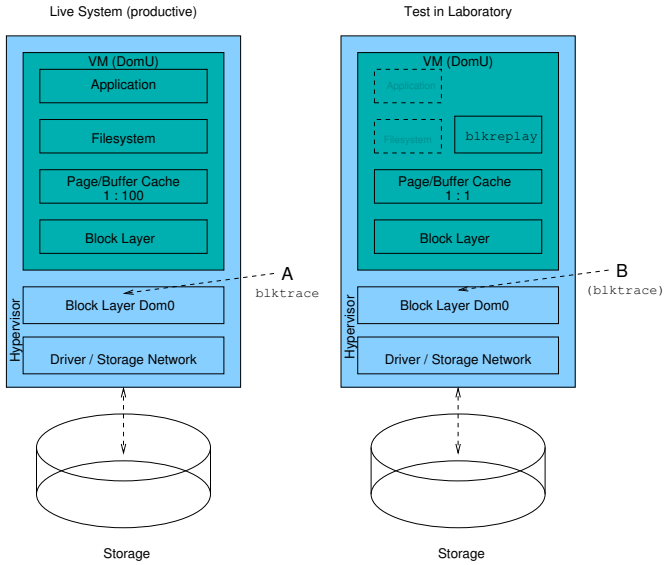
If you know what you are doing, you may switch on an option of `blkreplay` (see `cmode` in appendix A), which just *drops* conflicting requests instead of delaying them. As a consequence, stalls caused by potentially damaged IO are avoided, but at the cost of lost IO operations (deviating from the original recording[5]).

2.4. Mode of Operation

It is crucial to understand not only the concepts of `blkreplay` by itself, but also the *operating environment* where it is running. Otherwise, you can easily produce totally invalid fake results.

Please take a look at the following picture:

[5] Some loads don't suffer from repeated writes to the same place, but others do. Please check the output files of `blkreplay`. At the end, the number of dropped requests is displayed in the statistics section.



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 on 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 effect 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[6].

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 1:100 (or let it be only 1:1.1) would lead to heavy 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.

[6]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.

Notice that even by this measure, there are often 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[7]" 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`.

2.5. 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

[7] 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.

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.

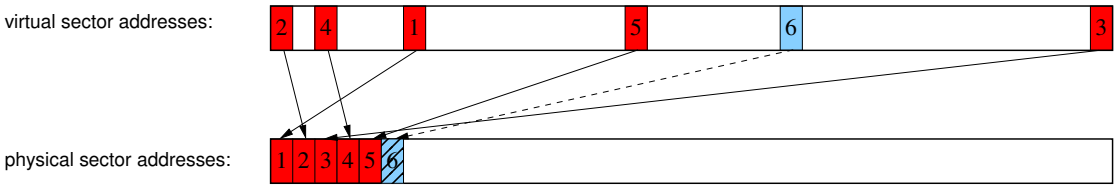
3.1.1. Pitfalls from Storage Virtualization

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):

[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.



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. 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.

[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

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] 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 black-box 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!

---

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.

[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.

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:

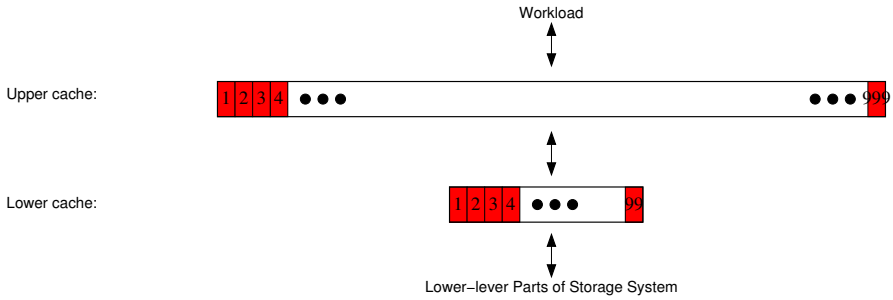
1. ignore all benchmark results from the time span before steady state has been reached.

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

[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.

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.



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.

[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.

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...

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.

[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.

[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".

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:

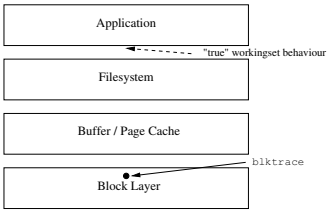
> A cache is called to be designed **efficiently**, iff at any point in time t the following condition holds:
> $|WS(t, \delta)| \leq |\text{cachesize}|$ for some $\delta \geq$ (time to fill the cache once).

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`:



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

[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.

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.

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 the *same* size as the original device where `blktrace` has taken its measurement from[17].

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,

[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 the **wraparound factor** related to your current replay device size. Therefrom you can easily compute the right device sizes.

[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**!

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.

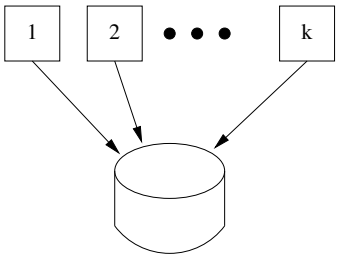
General rule: *never* expect any of these effects to be linear. Almost always, they are **non-linear**[20]. Anyone claiming something else must **prove** it!

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.



[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

[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.

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.

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.

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

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:

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 installed[23]. Go to subdirectory `src/` and type `make` there.

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.

[23] Just checkout the git repository of `blkreplay`. In addition, you need `gcc`, `make`, `gnuplot`, and some standard tools like `grep` / `awk` etc.

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*[24] 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, and directory ending with `.old` or including the substring `ignore` 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

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

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.

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.

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

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

Hint: 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.

Hint: 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.

Hint: 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.

Hint: 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.

Hint: 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.

Hint: 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).

Hint: *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

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`.

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`.

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[25]" 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 produces. In order additionally switch on any (or both), just add one of the options `--static` or `--dynamic`.

Both classes of graphics are described in the following.

3.2.4.1. Static Analysis

Here is a list of the meaning of different suffixes, together with a short description:

`*.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[26].

`*.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

[25] For many people; of course, there may be different needs. Feel free to rename your result files as you like.

[26] The slope of the `000` line is an indicator for the "repetitiveness" of the workload.

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.

`*.thrp.*.png`
: The throughput is displayed on the y axis. The requested throughput is red, while the actual throughput (only occurring in `*.replay.gz`) is green, for comparison[27].

3.2.4.2. Dynamic Analysis

Here is a list of the meaning of different suffixes, together with a short description:

`*.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.

`*.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* starting point, i.e. when the request *should have* started (excluding any delays). As before, multiple `*.replay.gz` are usually merged.

`*.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. May be used as a kind of "looking glass", to reveal more details from performance hot spots. Notice:

[27] Strictly speaking, the *actual* throughput is a dynamic result, which the *required* throughput is a static one. Although both may occur in one picture, `*.thrp.*` is nevertheless called a "static" graphics.

multiple concurrent replays are *not* merged on the x axis (as is the case with `*.realtime.png` and `*.setpoint.png`). Instead, the runs are just concatenated (pasted together sequentially).

`*.bins.png`
: The new y axis is now a *histogram* showing absolute frequency, while the x axis now carries the role of the former y axis. Example: `*.latency.bins.png` shows the latencies on the x axis (while formerly `*.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).

Hint: since the internal data format of `*.replay.gz` is the same as `*.load.gz`, you can use `/path/to/graph.sh --static *.replay.gz` to additionall 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

TBD

3.4. Advanced Features

TBD

3.5. Lowlevel Details and Expert Usage

TBD

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

TBD

5. Experiences with some Setups and some Loads

TBD

6. Internals of `blkreplay`

TBD

A. Config File Parameters

A.1. Basic Parameters

File `default-main.conf`:

```
#!/bin/bash
# Copyright 2010-2012 Thomas Schoebel-Theuer, sponsored by 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.
##
## 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.lo

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

output_label="MYPROJECT"

## 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. Many people think "the higher,
## the better". However, beware of hidden overheads.
##
## 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 vary with the kernel version.
##
## In order to approximate real life behaviour, you should consider
## the ACTUAL threading behaviour of your application.
##
## If you have a "fork bomb" like Apache, use a high number of threads.
## Typically, a database like mysql has a rather low number of threads.

threads=512

## cmode
##
## Shorthand for "conflict mode".
## See section about both timely and positionly overlapping
## in the 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, the conflicting request is just dropped.
##   This will minimize artificial delays, but at the cost of some
##   distortions from missing requests.
```

```
##
## 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
##   may be delayed as well.

cmode=with-drop

## 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

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

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

```
#replay_out_start=""
#speedup=1.0
#omit_tmp_cleanup=0
```

B. GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

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.