GemDisk: a GEOM Class to Emulate Disk Drives

Fabio Checconi* and Luigi Rizzo†

Abstract

Working on a novel, GEOM-based, disk scheduling infrastructure for FreeBSD, we are facing the problem of test coverage. In particular, it is quite difficult to foresee how a specific scheduling algorithm or optimization will behave with different disks and hardware configurations. Testing with real hardware would be the best thing to have; unfortunately it would require access to a lot of different, often expensive, media. So we are exploring the feasibility of using emulation, along with real hardware testing, to better understand the behavior of the different scheduling algorithms, and to prevent pitfalls in their deployment.

One result of our effort is GemDisk, a disk simulation tool which we are developing to mimic the behaviour of a disk unit (rotational media or even SSD) with configurable characteristics.

In this paper we present the status of our disk simulation tool, which is currently work in progress. We start giving some information on the state of the art of disk modelling, simulation, and parameter extraction. We then describe the architecture and the implementation of our simulator; finally we show some usage examples and evaluate its performance, discussing its limitations.

1 Introduction

Emulation has always been a powerful tool when designing and testing complex systems. Disk devices are complex systems and are no exception to that rule. The number of disk drives out in the world is so high that testing a software solution using them is just impossible; so there is the need to isolate their characteristics, and test the behavior of the software system under exam under the biggest possible amount of relevant configurations. This would still require a lot of expensive hardware, so one of the possible solutions to avoid unaffordable costs is using emulation.

Our recent work on disk scheduling put us in need of such an emulation system. The main limit we encountered with existing emulation/simulation approaches was their level of abstraction; we found either full system emulators, allowing the execution of a full operating system on top of them, but not offering an accurate model of disk performance, or disk simulators, very accurate in simulating disk devices, but highly inflexible or limited in the kind of system software they could be used to test.

Given these constraints and the limitation of previous solutions, we exploited the flexibility of the geom subsystem and the accuracy of DiskSim, a state-of-the-art disk simulator to create our own emulation approach. We are able to run unmodified applications on top of a user-specified mesh of real and emulated devices, using online simulation to drive the response times of the emulated devices.

2 Related Work

Storage systems simulation and modeling has a long history, and many of its aspects have been analyzed in the literature. Here we present the background we exploited designing and developing GemDisk.

2.1 Disk Modeling

Models for disk drives performance are in use since disk drives started to be used in storage systems. They take into account various aspects of the internal structure of the drives and of their observed behavior. The construction of an accurate model for an existing (yet old, as of now) disk drive is described in great detail in [11]. Another example of modeling can be found in [8].

Seek times are one of the critical components in determining the response time of a disk drive. In [11] the au-

*Scuola Superiore S. Anna, Pisa

†Dipartimento di Ing. dell'Informazione, Pisa

thors propose the following expression to calculate the seek time as a function of the seek distance d:

$$seekTime(d) = \begin{cases} T_0 & if \quad d = 1 \\ T_1 + T_2\sqrt{d} & if \ 1 < d < D \\ T_3 + T_4 d & if \quad d >= D, \end{cases}$$

where $T_i, i \in \{0, \ldots, 4\}$ and D are constants to be determined for each specific drive. This expression reflects a first phase of the seek operation, during which the head accelerates, and a second one, for long seeks, where the head moves at its maximum speed.

Rotational effects are kept into account by tracking the number of rotations completed from the beginning of the simulation, and using the distance from the position of the request to be served to calculate the rotational delay.

Transfer delays are considered proportional to the size of the requests, while caching effects require a split-down analysis of the various caching techniques drives can use; there are too many parameters to list them here, see [5] for a detailed discussion.

2.2 Simulation and Emulation

The model described in [11] has been implemented in the simulator described in [7]. Anyway the de-facto standard for research works is DiskSim [4], which we describe in more detail later.

The most relevant example of a timing-accurate storage emulator is the Memulator [6], which uses DiskSim, like GemDisk, to control the timing of the emulated device. The key difference between GemDisk and the Memulator is that the former is able to provide accurate simulation results (if the clients used for benchmarking do not use complex time handling techniques—an assumption that proved to be satisfied by many common benchmarking tools) even if the dimension of the emulated device is bigger than system RAM. Note also that RAM can be used as backing storage in GemDisk too, and the geom gate tools shipped with the system can be used to create a setup similar to the remote one implemented in Memulator[1].

Among others, in [3] the authors use an emulator similar to our one; they use it as a component of their system, consisting in an online filesystem and an offline analysis engine. In [13] the authors use an emulation scheme with virtual devices driven by an in-kernel emulator to validate their proposed handling of synchronous writes. Both these approaches are unable to cope with disk sizes exceeding the size of the RAM available in the system.

[1] In this setup the emulation is done on a remote machine, connected via a network interface to the system under test; this is useful if the load induced by the emulation component can interfere with the behavior of the application being tested.

System level emulators like QEMU [2] usually do not provide accurate timing for the storage components they emulate, as their focus is mainly in the throughput they can provide. However, more accurate emulators have been proposed, for example SimOS [10] used the simulator described in [7] to drive the timings of its storage components.

2.3 Parameter Extraction

Even the most accurate model needs an accurate knowledge of its parameters to produce accurate results. Various techniques have been proposed in the past, usually consisting in analyzing the response time of suitably crafted request patterns sent to the disk drive, eventually varying the initial state of the drive itself.

The various approaches in literature differ for the methodology of the various solicitations on the drive being analyzed. For example [12] uses only microbenchmarks consisting in read and write requests, being portable, but limited in the parameters that can be determined. On the other hand [14] uses SCSI commands to extract more detailed information from the drives. Finally, [5] presents DIXTrac, a tool for automated parameter extraction, designed to work together with DiskSim, providing it the parameters needed to simulate the analyzed drive.

3 Main Idea

The main idea behind GemDisk is providing virtual devices emulating the behavior of real ones, in a system running on bare metal. To do this we use geom_gate to create a geom provider which passes its requests to the userspace and has them served according to an user-level defined policy. The user-level demon controlling both the content and the timing of the requests reinjected back to the kernel uses an internal online simulator to make the provider react like a real device with the configured parameters.

As anticipated, the two dimensions controlled by the emulator are the handling of the backing store for the emulated device, and its temporal behavior. This is reflected by the architecture of our solution, a multi-threaded application running in background as a demon, handling the incoming requests (going from the geom gate to userspace) and reinjecting them (from userspace to the geom gate) after a) their content is read/written as requested, and b) the internal simulator says it is the time to do so. We will see how this two-dimensional split can cause problems with requests which the em-

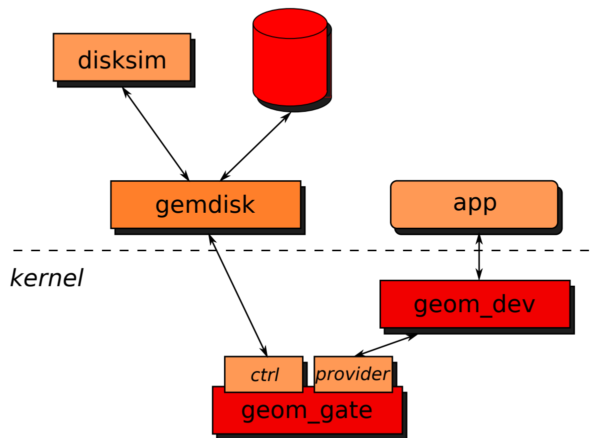


Figure 1: Architecture.

ulated device is ready to complete, but still have their backing store not ready to pass them back to the gate.

Figure 1 shows the architecture of the system. Requests flow from the application to the geom device node on top of a geom gate provider. The geom gate is connected, via a control interface (implemented using character device operations, as described in Section 4.1) to the userspace GemDisk demon. The GemDisk demon uses information coming both from actual storage and from a DiskSim instance to answer the requests coming from the gate. Using geom_gate allows us to use only userspace components; we will see that this has its good and bad sides.

4 Implementation

Now that we have seen the overall design of GemDisk, we are ready to see how it works. First we give a brief description of the components we have reused (i.e., geom_gate and DiskSim) work, then we describe some of the features/internals of DiskSim; then we give a detailed description of GemDisk. Finally, 4.4 contains a description of one of the most relevant problems we faced, and of our solution to it.

4.1 Geom Gates

Geom gate is a GEOM class[2] that can be used to forward block I/O requests from the kernel space to userspace, where they can be served by a regular application. It can be used, for example, to forward the I/O traffic of a GEOM provider to remote hosts, or to use non-GEOM devices as backends for GEOM providers.

The kernel-side of geom gate exports a character device-based interface, controlled via `/dev/ggctl`. Commands are given opening the controlling device and using `ioctl()` calls on it. The operations it supports are:

- `G_GATE_CMD_CREATE`, allocating a unit number and creating a new provider that can be accessed from userspace via geom gate; (geom dev will take care of creating a character device associated with it, once it tastes the newly created provider.)
- `G_GATE_CMD_DESTROY`, removing the created provider, and signaling the destruction to any userspace application serving it;
- `G_GATE_CMD_CANCEL`, canceling requests already completed by the userspace but not yet dispatched by the gate;
- `G_GATE_CMD_START`, returning the next request to serve from a specified geom gate device. If a request is already available it is returned immediately, otherwise the `ioctl()` sleeps until the gate provider receives a new one;
- `G_GATE_CMD_DONE`, signaling the completion of a request from userspace. This call is used to reinject requests into the gate and thus into GEOM.

GemDisk, as any application acting as an userspace request proxy using geom gate, creates one or more such devices and sits in a loop to receive and serve requests coming from the geom gate.

4.2 DiskSim

DiskSim [4] is a disk simulator, originally developed for research purposes. It can be configured to simulate all the components of a storage system, from disks to controllers, busses, up to schedulers and disk arrays. Recently, as part of the work described in [1], an SSD extension to DiskSim has been published.

DiskSim is structured internally as an event-driven simulator, and can be compiled as a library to be included in system-level simulators or—as in our case—emulators. The exported interfaces allows the external emulator to insert requests into the DiskSim simulated system and signals their completion by calling a function registered during initialization. The simulation loop is ran by the emulator, that has the responsibility of updating simulation time too.

Almost all the simulation parameters can be configured, from the details of the disk models used to the interconnection of the system components. Gemdisk reuses all the DiskSim configuration machinery as-is, thus everything documented in [4] still holds and can be applied to GemDisk configuration files.

[2] For an introduction to GEOM see Chapter 19 in [9].

```
struct request {
    struct g_gate_ctl_io ggio;
    struct aiocb aiocb;
    struct disksim_request dsim;
    int flags;
    int error;
    TAILQ_ENTRY(request) qlink;
};

struct device {
    uint32_t unit;
    char path[MAX_BPATH];
};
```

Figure 2: Request Descriptor.

4.3 GemDisk

GemDisk is a userspace geom gate client, structured as a multithreaded application. Its components are:

- the main thread, which initializes the application and then starts executing the DiskSim main loop;
- one or more *gatekeepers*, one per gate device, each one listening for the requests coming from the gate geoms, and sending them to the components that process them;
- a single (optional) *reaper*, which waits for the completion of the synchronous requests coming from the backing store.

4.3.1 Data Structures

The two main data structures used in GemDisk are shown in Figure 2. Each request has associated a `struct request` object, specifying its geom gate descriptor, its IOCB for asynchronous I/O, and the descriptor used for DiskSim simulation. The `flags` field is used to keep track of the state of the request, and `error` to store any eventual error encountered processing the request. Requests are queued before being submitted to DiskSim, using `qlink` to build a TAILQ. The data buffer is shared among `ggio` and `aiocb`, and it is allocated and resized dynamically.

Each device has its own descriptor, a `struct device` object, which associates each emulated device to its backing store and geom gate unit.

4.3.2 Main Thread

The (pseudo-)code for the main thread is shown in Figure 3. It first initializes the shared memory area used to communicate with clients (see Section 4.4), then initializes the DiskSim engine; then, for each device specified by the user on the command line, it creates a geom gate and a gatekeeper for it (calling `gate_create()`) and starts the reaper thread. Finally it enters the main simulation loop, which processes the requests to be passed to DiskSim, runs the simulator until it has handled all its internal events that are in the past with respect to the simulation time, and waits for new events to be generated (i.e., for a new request to be submitted, if GemDisk is idle).

```
gemdisk_serve(void)
{
    shm_init();
    disksim_init();

    for (devno = 0; devno < devnum; devno++)
        gate_create(devno);

    if (npaths)
        pthread_create(&tid, 0, gemdisk_reaper, 0);

    if (!verbose)
        daemon(0, 0);

    simtime = gettime();

    while (!stop_all) {
        process_queue();
        process_events();
        wait_next_event();
    }
}
```

Figure 3: Main Thread.

4.3.3 Gatekeepers

Figure 4 shows the structure of a thread handling the requests coming from a geom gate. Reinjecting requests is not done from this same thread.

The body of the thread allocates a request and then calls the `G_GATE_CMD_START` ioctl on the geom_gate control device; as a result, it receives in `rq->ggio` all the needed information to process the request (or an error code). The request is first allocated with a data buffer of the same size of a sector on the emulated device; then, if the `ioctl()` call fails and indicates that it was due to lack of memory, the data buffer of the request is resized and the `ioctl()` is retried. Requests are allocated dynamically, and their data buffer is allocated separately to allow the resize step described above.

The `enqueue()` function is the most interesting call here. Its body is shown in Figure 5. This function handles the newly arrived request both from the I/O and the simulator perspective. If there is a backing file descriptor, an IOCB is prepared to do the needed I/O from it, and the IOCB is submitted asynchronously; then the DiskSim side is handled: the relevant fields of the DiskSim request are filled and the request itself is enqueued in a dispatch queue. We will see why it is not possible to immediately hand the request to DiskSim in Section 4.4.

```
gemdisk_gatekeeper(void *arg)
{
    struct device *dev;
    struct request *rq;

    for (;;) {
        rq = get_request(dev->unit);
once_again:
        g_gate_ioctl(G_GATE_CMD_START, &rq->ggio);
        switch (rq->ggio.gctl_error) {
        case 0:
            break;
        case ECANCELED:
            /* Exit gracefully. */
                        handle_exit();
        case ENOMEM:
            /* Buffer too small. */
            resize_request(rq);
                        goto once_again;
        }

        switch (rq->ggio.gctl_cmd) {
        case BIO_READ:
            if ((size_t)ggio->gctl_length >
                 sectorsize)
                resize_request(rq);
                        enqueue(rq, true);
            break;
        case BIO_DELETE:
        case BIO_WRITE:
            enqueue(rq, false);
            break;
        }
        }
}
```

Figure 4: Gatekeeper.

About synchronization, the `requests` condition variable is used to signal to the main thread that new requests are available from the dispatch queue, to wake it up in case it went idle.

4.3.4 Reaper

The structure of the (only) reaper thread is shown in Figure 6. It only waits for asynchronous I/O requests to complete and, when the handling of the I/O side of a request is handled, it tries to reinject it into the gate that generated it.

To keep things simple, we use a timeout instead of signals to terminate gracefully the application, so we have a `sleep()` loop here.

Figure 7 shows how the reinjection works. `try_reinject_request()` can be called both from the main thread and from the reaper thread, and the request is actually reinjected only after both have called this function. To keep track of the state of the request we use its `flags` field, and we call `reinject_request()` only if we see both `RQ_DSIM_COMPLETED` and `RQ_IO_COMPLETED` set.

Reinjecting just means calling the `G_GATE_CMD_DONE` `ioctl()` on the geom gate control device, to pass back the request to its gate.

```
enqueue(struct request *rq, bool read)
{

    if (rq->iocb.aio_fildes != -1) {
        rq->iocb.aio_buf = rq->ggio.gctl_data;
        rq->iocb.aio_offset = rq->ggio.gctl_offset;
        rq->iocb.aio_nbytes = rq->ggio.gctl_length;

        if (read)
            aio_read(&rq->iocb);
        else
            aio_write(&rq->iocb);
        }
    } else
        rq->flags |= RQ_IO_COMPLETED;

    rq->dsim.flags = read ? DISKSIM_READ :
        DISKSIM_WRITE;
    rq->dsim.blkno = rq->ggio.gctl_offset /
        sectorsize;
    rq->dsim.bytecount = rq->ggio.gctl_length;

    pthread_mutex_lock(&gemdisk_mutex);
    rq->drq.start = gettime();
    TAILQ_INSERT_TAIL(&queue, rq, qlink);
    pthread_cond_signal(&requests);
    pthread_mutex_unlock(&gemdisk_mutex);
}
```

Figure 5: Enqueueing Requests.

```
gemdisk_reaper(void *arg)
{
    struct request *rq;
    struct aiocb *iocb;
    int error;

    while (!stop_all) {
        error = aio_waitcomplete(&iocb, TIMEOUT);
        if (error == -1 && errno == EAGAIN) {
            sleep(1);
            continue;
        }
        if (error == -1 && errno == EINPROGRESS)
            continue;
        if (error == -1 && !iocb)
            break;

        error = error < 0 ? errno : 0;
        try_reinject_request(rq, error,
            RQ_IO_COMPLETED);
        pthread_cond_signal(&io_completed);
    }

    return (NULL);
}
```

Figure 6: Reaper.

```
reinject_request(struct request *rq, int error)
{
        struct g_gate_ctl_io *ggio = &rq->ggio;

        ggio->gctl_error = error;
        g_gate_ioctl(G_GATE_CMD_DONE, ggio);
        put_request(rq);
}


try_reinject_request(struct request *rq, int error,
    int flag)
{

    pthread_mutex_lock(&gemdisk_mutex);
    if (!(rq->flags & RQ_COMPLETED_MASK) && flag ==
        RQ_DSIM_COMPLETED)
        early_request(rq);

    rq->flags |= flag;

    if ((rq->flags & RQ_COMPLETED_MASK) ==
        RQ_COMPLETED_MASK)
        reinject_request(rq, !rq->error ? error :
            rq->error);
    else
        rq->error = error;
    pthread_mutex_unlock(&gemdisk_mutex);
}
```

Figure 7: Reinject Mechanism.

4.4 About Time

The single most complex issue to face while implementing GemDisk was timing. An ideal emulator would serve the requests exactly at the time the real hardware would do, so we used real time as a baseline for the emulator. Anyway, in particular when emulating fast drives, the backing store might be slower than the emulated hardware, and it may happen that a request that should have been completed in the emulated system is still under service in the real one.

Please note that most of the benchmarks which act at the block level, reading or writing directly to the device, do not care about the contents of the requests; in this case the user can just specify an empty backing store, bypassing completely the problem, and getting the proper timing from the emulator.

Anyway in the general case this is not true; to “solve” this problem we decided to *slow down* the perceived time for the applications under test. This was done hijacking some of the libc-provided syscalls related to time manipulation for the client processes. An example of how emulation time evolves, with respect to real time, during an emulation session is shown in Figure 8. The two instants where emulated time stops are when a request is ready to be completed in the emulated system, but it has not completed its I/O.

A process issuing synchronous requests only, which is blocked waiting for the completion of the outstanding request, cannot even notice this temporary halt of

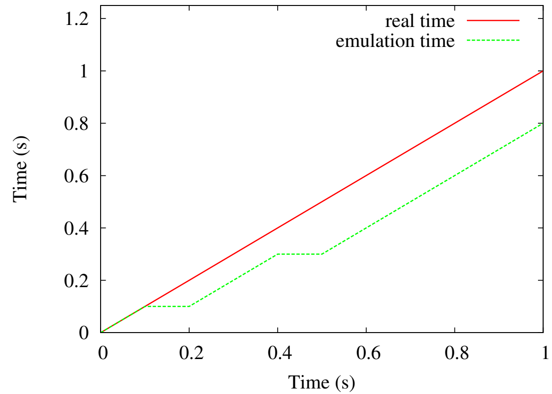


Figure 8: Flowing of Time.

system time (of course unless it communicates with external applications, with strict clock synchronization requirements). Processes issuing asynchronous requests see a slower system when emulated time is blocked, but generally can still make good use of the timing information they read (being them benchmark applications, they measure time mostly to generate I/O at given rates or to measure latency/throughput of the emulated device, thus they act like they were executed in a system running at emulation time).

Figure 9 shows how the emulated time is slowed down. `struct gemdisk_shtable` stores two values, the delta between the real time and the emulated time, and the last valid emulation time. Only one of the two values is used by clients: if the emulated time is growing then `delta` is even (its last bit is 0) and the emulated time is stopped at the value contained in `last`; otherwise the client can calculate the current time from `gettimeofday()`, summing `delta` to the returned value.

To synchronize GemDisk with its clients we use a seqlock, implemented using the `version` field of `struct gemdisk_shtable`. While spinning in userspace is not a good idea, here we can limit the number of retries of the reader, returning the last read value in case the writer has been preempted while inside its critical section.

The instance of `struct gemdisk_shtable` used to communicate among GemDisk and its clients is exported/accessed using System V shared memory. To avoid the need to modify clients to take into account this dedicated time handling we implemented our modified timing-related functions in a shared library, to be loaded using `LD_PRELOAD` to override the libc version of each call.

```
struct gemdisk_shtable {
        uint64_t delta;
        uint64_t last;
        uint32_t version;
};

early_request(struct request *rq)
{
        struct timeval tv, tv2;
        uint64_t last, delta;

        gettimeofday(&tv, NULL);

        last = (uint64_t)(gemdisk_gettime() * 1000);
        delta = gemdisk_shtable->delta | 1;
        write_shtime(last, delta);

        while ((rq->flags & RQ_COMPLETED_MASK) !=
              RQ_IO_COMPLETED &&
             !stop_all)
                cond_wait(&gemdisk_io_completed);

        gettimeofday(&tv2, NULL);
        delta = (delta + tv2.tv_sec * 1000 +
              tv2.tv_usec -
             tv.tv_sec * 1000 - tv.tv_usec) & ~1ULL;

        write_shtime(last, delta);
}
```

Figure 9: Timing Functions.

5 Usage Scenarios

5.1 Timing-only Emulation

The first, simple usage of GemDisk is the emulation of the timings of direct disk accesses which do not use the data they read/write. This may seem a useless scenario, in fact a lot of microbenchmarking activity needed to tune a disk scheduler does not use anything it reads or write; of course using a filesystem is not possible.

To support this mode of operation, GemDisk allows the creation of emulated devices with no backing store. The libc hijacking code is not needed because the I/O side of requests is never dispatched, so it can never be late.

5.2 In-memory Emulation

If the application under test uses the data it reads or writes, then a backing store must be specified. Anyway using a ramdisk allows bypassing the libc hijacking code, because accessing the backing store is fast (and unless emulating a flash memory it's unlikely to take more in the real system than in the emulated one). The big limitation of this configuration is that the size of the emulated devices cannot exceed the RAM size available on the system.

5.3 Disk-backed Emulation

The complete configuration allows using a file/device as backing store, thus it does not limit the maximum size for the emulated device. In this case time for the clients needs to be virtualized. Here the drawback is that all the applications doing I/O on the emulated devices, and the ones communicating with them need to be synchronized on the same clock driven by the emulator. This prevents the usage of GemDisk in distributed systems.

6 Experimental Evaluation

To evaluate the effectiveness of our emulation approach, we tested GemDisk accuracy in two different usage scenarios, with two different synthetic workloads injected into the system running the emulator. We focused on accuracy because, by design, GemDisk is able to emulate devices of arbitrary bandwidth requirements, as the "time stealing" mechanism makes the device appear to be faster than it really is according to real time. Apart from that, we used [6] as a model for the evaluation of GemDisk.

In our experiments we used a low-end setup, composed by an Asus EEEPC, equipped with an Atom N270 1.60GHz CPU, 1GB RAM and a Western Digital WD3200BEKT hard disk, with 16MB of cache, working at 7200RPM. Since the CPU pressure of the testing application is an issue that must be solved separately (e.g., moving the emulation component to a remote box) we used I/O bound workloads to test the emulator. Our basic aim was at evaluating the timing behavior of the emulated device, comparing it with the reference one provided by a DiskSim simulation of the same workload used for testing.

We generated a couple of different workloads, a *random* one, characterized by zero probability of local or sequential access, and a *mixed* one, with 0.3 probability of local access (i.e., within 500LBNs from the previous request, 0.2 probability of sequential access, and 0.5 probabilty of random access. In both cases the request size was uniformly distributed in $[2\text{KiB}, 130\text{KiB}]$, the interarrival times were uniformly distributed in $[0, 500\text{s}]$, and the number of generated requests was 500.

6.1 Emulation Accuracy

The first measure of emulation accuracy we did consist in comparing the service time of each request in GemDisk with the service time of the same request in the DiskSim simulation, that we called *emulation error*. To do that we developed a simple synthetic trace replay tool that we used to inject requests in the simulator and,

in a later step, in the gate device connected to the emulator.

For the simulator we considered the service time as the difference from the simulation time when the request was injected into DiskSim, and its completion time; for GemDisk we considered the difference between when the request reached the emulator and the time when it was inserted back into the gate device.

Figure 10 shows the distributions of the errors obtained during our measurements; in particular Figures 10(a) and 10(b) show the emulation error, in the case of a virtual device not backed by real storage. We can see that most of the samples are around zero, meaning no difference between the completion time in the simulator and the one in the emulator.

The figures show that a significative part of the samples completes earlier in the emulator than in the simulator. This is most likely due to the fact that the injection tool in some cases is not able to respect the arrival time of the requests and issues them with some delay, causing a discrepancy between the arrivals seen from the internal simulator with respect to the ones seen in the simulator-only environment. Even if we need to work more on this issue, this behavior seems related more to the way used to inject requests in the system rather than to the emulator design.

On the other hand, errors do not exceed a few ticks, which is compatible with both timer accuracy and preemptions which can cause the GemDisk threads to be descheduled when requests have to be reinjected back to the gate device.

With *"stolen time" error* we indicated the error in the completion time obtained at the reinjection in the gate, but when the time stealing mechanism was active (i.e., with the emulated device backed by real storage). Figures 10(c) and 10(d) show that there is no significative difference with respect to the emulation error with no backing device, indicating that, at least for simple applications the time stealing mechanism is a viable approach. Even if this was not the goal of this tests, the trace contained some amount of parallelism in the requests, and the trace replay tool used asynchronous I/O, so this behavior was not obtained in the most favorable conditions for the mechanism to work.

The last accuracy index we considered was what we called the *user perceived error*, i.e., the difference between the completion times in an emulated system with no backing store and no time stealing, and a system with an emulated drive with backing store and time stealing. Figures 10(e) and 10(f) show that the time stealing mechanism does add some noise, but it remains under the few milliseconds. As we have also seen previously, in our tests there was no substantial accuracy degradation deriving from the use of this mechanism.

7 Availability

`gemdisk` is available at:
`http://feanor.sssup.it/~fabio/freebsd/gemdisk/`

8 Future Work

GemDisk is still work in progress. There is still room for improvements with respect to its accuracy; moreover, we need to test it under more workloads, to better understand its limitations, and we need to extend the time virtualization mechanism to cover more possible users. Another topic we still have to explore is how the same time warping mechanism can be used to provide emulation with multiple instancies of GemDisk and/or on distributed systems.

9 Conclusion

In this paper we described GemDisk, a GEOM-based timing-accurate disk emulator that can be used to test system performance with a variety of storage system configurations, without the need of buying and assembling the real hardware.

We have also presented what, to the best of our knowledge, is the first attempt to let the storage emulation component drive the time perceived by the applications under test.

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46.

[3] Peter Bosch and Sape J. Mullender. Cut-and-paste file-systems: integrating simulators and file-systems. In *Proceedings of the 1996 USENIX Technical Conference*, pages 307–318. USENIX, 1995.

[4] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, May 2008.

[5] Jiri Schindler Gregory and Gregory R. Ganger. Automated disk drive characterization. Technical report, 1999.

[6] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate storage emulation. In *In Proceedings of the FAST 2002 Conference on File and Storage Technology)*, pages 75–88. USENIX Association, 2002.

[7] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical report, Dept. of Computer Science, Dartmouth College, 1994.

[8] Edward Kihyen Lee. *Performance modeling and analysis of disk arrays*. PhD thesis, 1993. Chair-Katz, Randy H.

[9] The FreeBSD Documentation Project. *FreeBSD Handbook*. 2009. `http://www.freebsd.org/doc/en/books/handbook/index.html`.

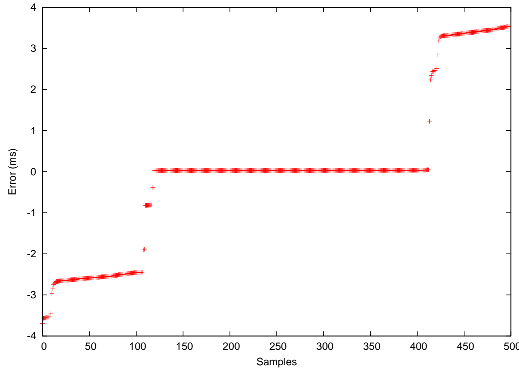
[10] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7:78–103, 1997.

[11] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27:17–28, 1994.

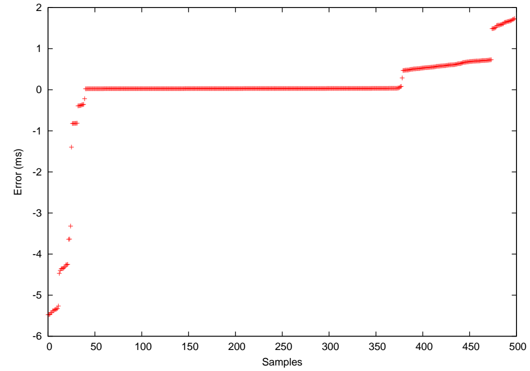
[12] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Micro-benchmark based extraction of local and global disk characteristics. Technical report, Berkeley, CA, USA, 2000.

[13] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file systems for a programmable disk. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 29–43, Berkeley, CA, USA, 1999. USENIX Association.

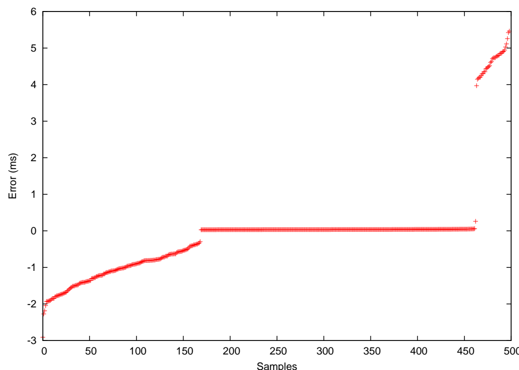
[14] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters, 1995.



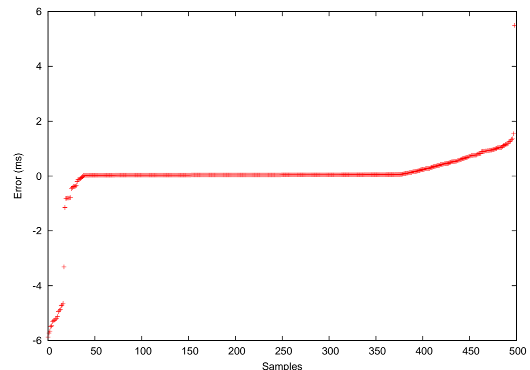
(a) Random, emulation error.



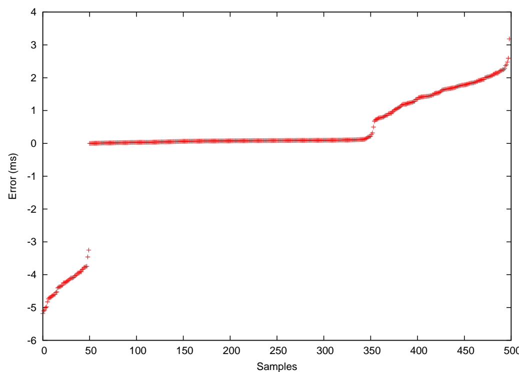
(b) Mixed, emulation error.



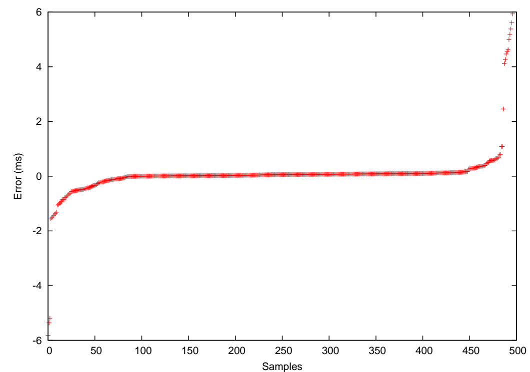
(c) Random, "stolen time" error.



(d) Mixed, "stolen time" error.



(e) Random, user perceived error.



(f) Mixed, user perceived error.

Figure 10: Various error figures obtained with the test workloads.