GEOM_SCHED: A Framework for Disk Scheduling within GEOM

Luigi Rizzo and Fabio Checconi

May 8, 2009

GEOM_SCHED
A framework for disk scheduling within GEOM

Luigi Rizzo

Dipartimento di Ingegneria dell'Informazione
via Diotisalvi 2, Pisa, ITALY

Fabio Checconi

SSSUP S. Anna,
via Moruzzi 1, Pisa, ITALY

Summary

- Motivation for this work
- Architecture of GEOM_SCHED
- Disk scheduling issues
- Disk characterization
- An example anticipatory scheduler
- Performance evaluation
- Conclusions

Motivation

- Performance of rotational media is heavily influenced by the pattern of requests;
- anything that causes seeks reduces performance;
- scheduling requests can improve throughput and/or fairness;
- even with smart filesystems, scheduling can help;
- FreeBSD still uses a primitive scheduler (elevator/C-LOOK);
- we want to provide a useful vehicle for experimentation.

Where to do disk scheduling

To answer, look at the requirements. Disk scheduling needs:

- geometry info, head and platter position;
    - necessary to exploit locality and minimize seek overhead;
    - known exactly only within the drive's electronics;
- classification of requests;
    - useful to predict access patterns;
    - necessary if we want to improve fairness;
    - known to the OS but not to the drive.

Where to do disk scheduling

Possible locations for the scheduler:

- Within the disk device
  - has perfect geometry info;
  - requires access to the drive's firmware;
  - unfeasible other than for specific cases.
- Within the device driver
  - lacks precise geometry info.
  - feasible, but requires modification to all drivers;
- Within GEOM
  - lacks precise geometry info;
  - can be done in just one place in the system;
  - very convenient for experimentations.

Why GEOM_SCHED

Doing scheduling within GEOM has the following advantages:

- one instance works for all devices;
- can reuse existing mechanisms for datapath (locking) and control path (configuration);
- makes it easy to implement different scheduling policies;
- completely optional: users can disable the scheduler if the disk or the controller can do better.

Drawbacks:

- no/poor geometry and hardware info (not available in the driver, either);
- some extra delay in dispatching requests (measurements show that this is not too bad).

Part 2 - GEOM_SCHED architecture

- GEOM_SCHED goals
- GEOM basics
- GEOM_SCHED architecture

GEOM_SCHED goals

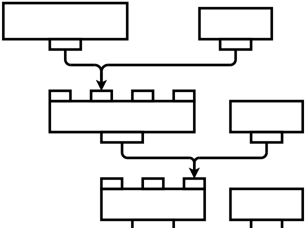
Our framework has the following goals:

- Support for run-time insertion/removal/reconfiguration;
- support for multiple scheduling algorithms;
- production quality.

GEOM Basics

Geom is a convenient tool for manipulating disk I/O requests.

- Geom modules are interconnected as nodes in a graph;
- Disk I/O requests (”bio’s”) enter nodes through ”provider” ports;
- arbitrary manipulation can occur within a node;
- if needed, requests are sent downstream through ”consumer” ports;
- one provider port can have multiple consumer ports connected to it;
- the top provider port is connected to sources (e.g. filesystem);
- the bottom node talks to the device driver.



Disk requests

A disk request is represented by a struct bio , containing control info, a pointer to the buffer, node-specific info and glue for marking the return path of responses.

```
struct bio {
        uint8_t bio_cmd;                        /* I/O operation. */
        ...
        struct cdev *bio_dev;                   /* Device to do I/O on. */
        long    bio_bcount;                     /* Valid bytes in buffer. */
        caddr_t bio_data;                       /* Memory, superblocks, indire
        ...
        void    *bio_driver1;                   /* Private use by the provider
        void    *bio_driver2;                   /* Private use by the provider
        void    *bio_caller1;                   /* Private use by the consumer
        void    *bio_caller2;                   /* Private use by the consumer
        TAILQ_ENTRY(bio) bio_queue;             /* Disksort queue. */
        const char *bio_attribute;              /* Attribute for BIO_[GS]ETATT
        struct g_consumer *bio_from;            /* GEOM linkage */
        struct g_provider *bio_to;              /* GEOM linkage */
        ...
};
```

Adding a GEOM scheduler

Adding a GEOM scheduler to a system should be as simple as this:

- decide which scheduling algorithm to use (may depend on the workload, device, ...);
- decide which requests we want to schedule (usually everything going to disk);
- insert a GEOM_SCHED node in the right place in the datapath.

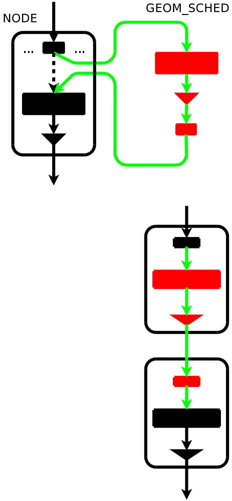
Problem: current "insert" mechanisms do not allow insertion within an active path;

- must mount partitions on the newly created graph to use of the scheduler;
- or, must to devise a mechanism for transparent insertion/removal of GEOM nodes.

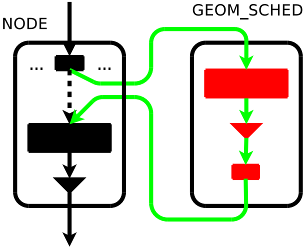
Transparent Insert

Transparent insertion has been implemented using existing GEOM features (thanks to phk's suggestion):

- create new geom, provider and consumer;
- hook new provider to existing geom;
- hook new consumer to new provider;
- hook old provider to new geom.



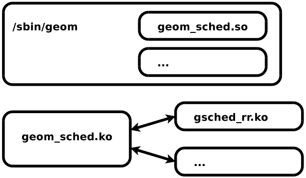
Transparent removal



Revert previous operations:

- hook old provider back to old geom;
- drain requests to the consumer and provider (careful!);
- detach consumer from provider;
- destroy provider.

GEOM_SCHED architecture



GEOM_SCHED is made of three parts:

- a userland object (geom_sched.so), to set/modify configuration;
- a generic kernel module (geom_sched.ko) providing glue code and support for individual scheduling algorithms;
- one or more kernel modules, implementing different scheduling algorithms (gsched_rr.ko, gsched_as.ko, ...).

GEOM_SCHED: geom_sched.so

geom_sched.so is the userland module in charge of configuring the disk scheduler.

```
# insert a scheduler in the existing chain
geom sched insert &lt;provider&gt;

# before:  [pp --> gp  ..]
# after:   [pp --> sched_gp --> cp]   [new_pp --> gp ... ]

# restore the original chain
geom sched destroy &lt;provider&gt;.sched.
```

GEOM_SCHED: geom_sched.ko

geom_sched.ko:

- provides the glue to construct the new datapath;
- stores configuration (scheduling algorithm and parameters);
- invokes individual algorithms through the GEOM_SCHED API;

```
  geom{}         g_sched_softc{}      g_gsched{}
+----------+    +---------------+   +-------------+
|  softc *-|--->| sc_gsched   *-|-->|  gs_init    |
|  ...      |    |               |   |  gs_fini    |
|          |    | [ hash table] |   |  gs_start   |
+----------+    |               |   |  ...        |
                |               |   +-------------+
                |               |
                |               |     g_*_softc{}
                |               |   +-------------+
                | sc_data     *-|-->|  algorithm- |
                +---------------+   |  specific   |
                                    +-------------+
```

Scheduler modules

Specific modules implement the various scheduling algorithms, interfacing with geom_sched.ko using the GEOM_SCHED API

```
/* scheduling algorithm creation and destruction */
typedef void *gs_init_t (struct g_geom *geom);
typedef void gs_fini_t (void *data);

/* request handling */
typedef int gs_start_t (void *data, struct bio *bio);
typedef void gs_done_t (void *data, struct bio *bio);
typedef struct bio *gs_next_t (void *data, int force);

/* classifier support */
typedef int gs_init_class_t (void *data, void *priv, struct thread *tp
typedef void gs_fini_class_t (void *data, void *priv);
```

GEOM_SCHED API, control and support

- gs_init() : called when a scheduling algorithm starts being used by a geom_sched node.
- gs_fini() : called when the algorithm is released.
- gs_init_class() : called when a new client (as determined by the classifier) appears.
- gs_fini_class() : called when a client (as determined by the classifier) disappears.

GEOM_SCHED API, datapath

- gs_start() : called when a new request comes in. It should enqueue the request and return 0 on success, or non-zero on failure (meaning that the scheduler will be bypassed, in this case bio->bio_caller1 is set to NULL).
- gs_next() : called i) in a loop by g_sched_dispatch() right after gs_start(); ii) on timeouts; iii) on 'done' events. Should return immediately, either a pointer to the bio to be served or NULL if no bio should be served now. Always return an entry if available and the "force" argument is set.
- gs_done() : called when a request under service completes. In turn the scheduler should either call the dispatch loop to serve other pending requests, or make sure there is a pending timeout to avoid stalls.

Classification

- Schedulers rely on a classifier to group requests. Grouping is usually done basing on some attributes of the creator of the request.
- long term solution:
  - add a field to the struct bio (cloned as other fields);
  - add a hook in g_io_request() to call the classifier and write the "flowid".
- For backward compatibility, the current code is more contrived:
  - on module load, patch g_io_request to write the "flowid" into a seldom used field in the topmost bio;
  - when needed, walk up the bio chain to find the "flowid";
  - on module unload, restore the previous g_io_request.
- this is just experimental, but lets us run the scheduler on unmodified kernels.

Part 3 - disk scheduling basics

Disk scheduling basics

Back to the main problem, disk scheduling for rotational media (or any media where sequential access is faster than random access).

- Contiguous requests are served very quickly;
- non contiguous requests may incur rotational delay or a seek penalty.
- In presence of multiple outstanding requests, the scheduler can reorder them to exploit locality.
- Standard disk scheduling algorithm: C-SCAN or "elevator";
- sort and serve requests by sector index;
- never seek backwards.

Disksort (and its API)

- bioq_disksort is a data structure that implements the C-SCAN algorithm;
- provides an API to force ordering;

- bioq_disksort() performs an ordered insertion;
- bioq_first() return the head of the queue, without removing;
- bioq_takefirst() return and remove the head of the queue, updating the 'current head position' as bioq->last_offset = bio->bio_offset + bio->bio_length;
- bioq_insert_tail() insert an entry at the end. It also creates a 'barrier' so all subsequent insertions through bioq_disksort() will end up after this entry;
- bioq_insert_head() insert an entry at the head, update bioq->last_offset = bio->bio_offset so that all subsequent insertions through bioq_disksort() will end up after this entry;
- bioq_remove() remove a generic element from the queue, act as bioq_takefirst() if invoked on the head of the queue.

Capture

- Requests are sorted by position, so a greedy, sequential client can "capture" the disk;

```
     offset --->
+----------------------------------------------+
|  WWWWW....      XXX...    YY....             |
+----------------------------------------------+
```

- likely to happen with writers, which are asynchronous;
- can be addressed by advancing the 'current' head position after a few sequential requests;
- the trick still does not protect from scattered request patterns.

Deceptive Idleness

- Readers tend to be synchronous: no request is sent before the previous one is complete;

  ```
       offset --->
  +--------------------------------------------+
  |  Aaaaaaa...    Bbbbbb...                   |
  +--------------------------------------------+
  ```

  Arrival order: A B a b a b ...
  Service order: A [seek] B [seek] a [seek] b ...
- the stream of requests from a process doing synchronous I/O is never seen as continuously backlogged by the scheduler.
- the interval between subsequent requests from the same client is called "think time".

Possible Solution: Anticipation

Basic idea: wait a bit before serving non contiguous requests, just in case a contiguous one comes soon.

- Useful with synchronous clients;
- may cause unnecessary idleness;
- may need some tuning of parameters (estimate the think time, don't wait much longer than that);
- helps fair schedulers to distribute disk bandwidth.

Addressing Fairness

Goal: assign resources according to some specific allocation pattern.

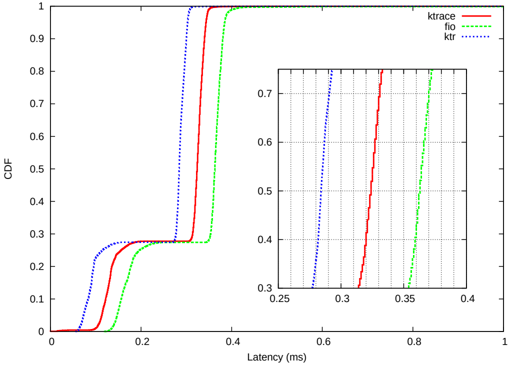
- Actual allocation should be independent from requests from competing clients (isolation);
- actual allocation should not alter the rate of our requests (impossible to achieve with synchronous clients);
- usually addressed by controlling the service delay experienced by our requests;
- same as the other two problems, relies on classification of requests.

Part 4 - disk characterization

Some measurements to analyse the behaviour of different schedulers.

- Characterize disk (and device driver) behaviour;
- important to design and understand the behaviour of scheduling algorithms.

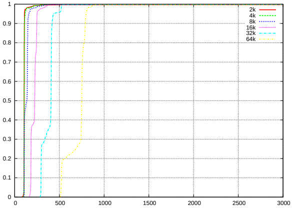
How to do measurement ?

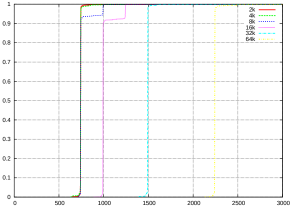


- Userland, ktrace, ktr ?
- small difference even with 2k blocks;
- userland is often good enough;
- be careful to discard outliers (initial seeks, scheduling artifacts, etc.)

Latency vs blocksize, streaming

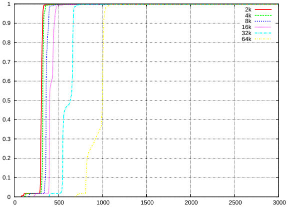
- limited by the disk/interface/bus throughput;
- latency also grows with the blocksize.
- left: 250GB SATA, 7200 RPM, peak 88MB/s;
- right: 250MB, ATA+USB, 700 RPM, USB2 peak 27MB/s

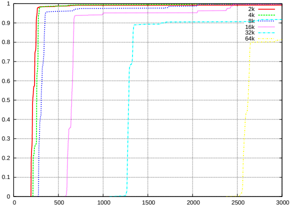




Latency vs blocksize, streaming(2)

- Two more disks:
- left: 160GB laptop, 19MB/s; right: 320MB 7200 RPM SATA, peak 75MB/s

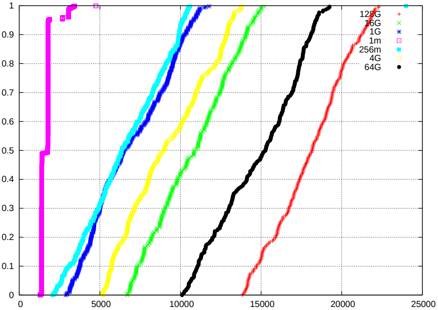




Delay vs seek distance

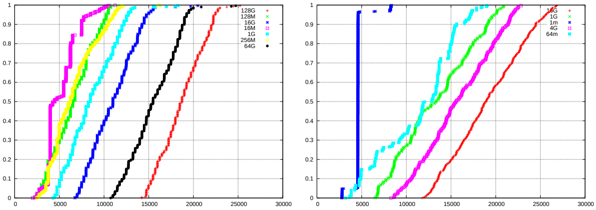
Seek delays have 3 parts:

- Acceleration/settle time;
- moving (proportional to distance)
- rotational delay.
- below: 250GB Sata, 7200 RPM



More delay vs seek distance

left: USB, 7200 RPM; right: laptop, 3600 rpm



Remarks on measurements

- we don't have exact geometry info, so we cannot easily predict the exact seek latency;
- media has variable throughput (and probably variable density);
- beware of caching;
- we don't know caching/readahead policies.
- Some measurement can be made at runtime and used to tune the scheduler.

Part 5 - an example disk scheduler

Example scheduler: gsched_rr

- Per-client queues sorted using C-SCAN;
- Round robin between queues;
- Anticipation on the queue currently under service;
- Bounded number of requests for each queue.
- Parameters:

```
kern.geom.sched.rr.wait_ms          5
kern.geom.sched.rr.bypass           0
kern.geom.sched.rr.w_anticipate     1
kern.geom.sched.rr.quantum_kb    8192
kern.geom.sched.rr.quantum_ms      50
kern.geom.sched.rr.queue_depth      1
```

Exported sysctl's

There are a few sysctl's exported by geom schedulers, for stats and debugging

```
kern.geom.sched.requests:      total requests
kern.geom.sched.in_flight:     requests in flight
kern.geom.sched.in_flight_w:   writes in flight
kern.geom.sched.in_flight_b:   bytes in flight
kern.geom.sched.in_flight_wb:  write bytes in flight
kern.geom.sched.done:          completed requests
kern.geom.sched.algorithms:    registered algorithms
kern.geom.sched.debug:         verbosity
kern.geom.sched.expire_secs:   classifier hash expire
```

gsched_rr performance

Some preliminary results on scheduler's performance in some easy cases (the focus here is on the framework).
Measurement is using multiple dd instances on a filesystems, all speeds in MiB/s.

- two greedy readers, throughput improvement
  NORMAL: 6.8 + 6.8 ; GSCHED_RR: 27.0 + 27.0
- one greedy reader, one greedy writer, capture effect
  NORMAL: R: 0.234 W:72.3 ; GSCHED_RR: R:12.0 W:40.0
- multiple greedy writers, only small loss of througput
  NORMAL: 16+16; RR: 15.5 + 15.5
- one sequential reader, one random reader (fio)
  NORMAL: Seq: 4.2 Rand: 4.2; RR: Seq: 30 Rand: 4.4

Conclusions

- We have presented GEOM_SCHED, a framework for disk scheduling within GEOM;
- extremely simple to use and non intrusive
- Already able to give performance improvements in simple cases
- no or small regression in generic case (low overhead)
- need some autotuning to achieve better performance
- open to experimentation (e.g. readahead in geom ?)

Questions ? luigi@freebsd.org
Code: http://info.iet.unipi.it/ luigi/FreeBSD/