GPU Job Scheduling in DRM: Past, Present and Future

Philipp Stanner
Red Hat
September 2025

Who?

- Philipp Stanner
- Kernel Engineer at Red Hat's "GPU & Accelerators" team
- One of three DRM GPU Scheduler maintainers
- phasta@kernel.org
- OFTC: phasta (I've got no bouncer. If I'm offline, I'm offline)

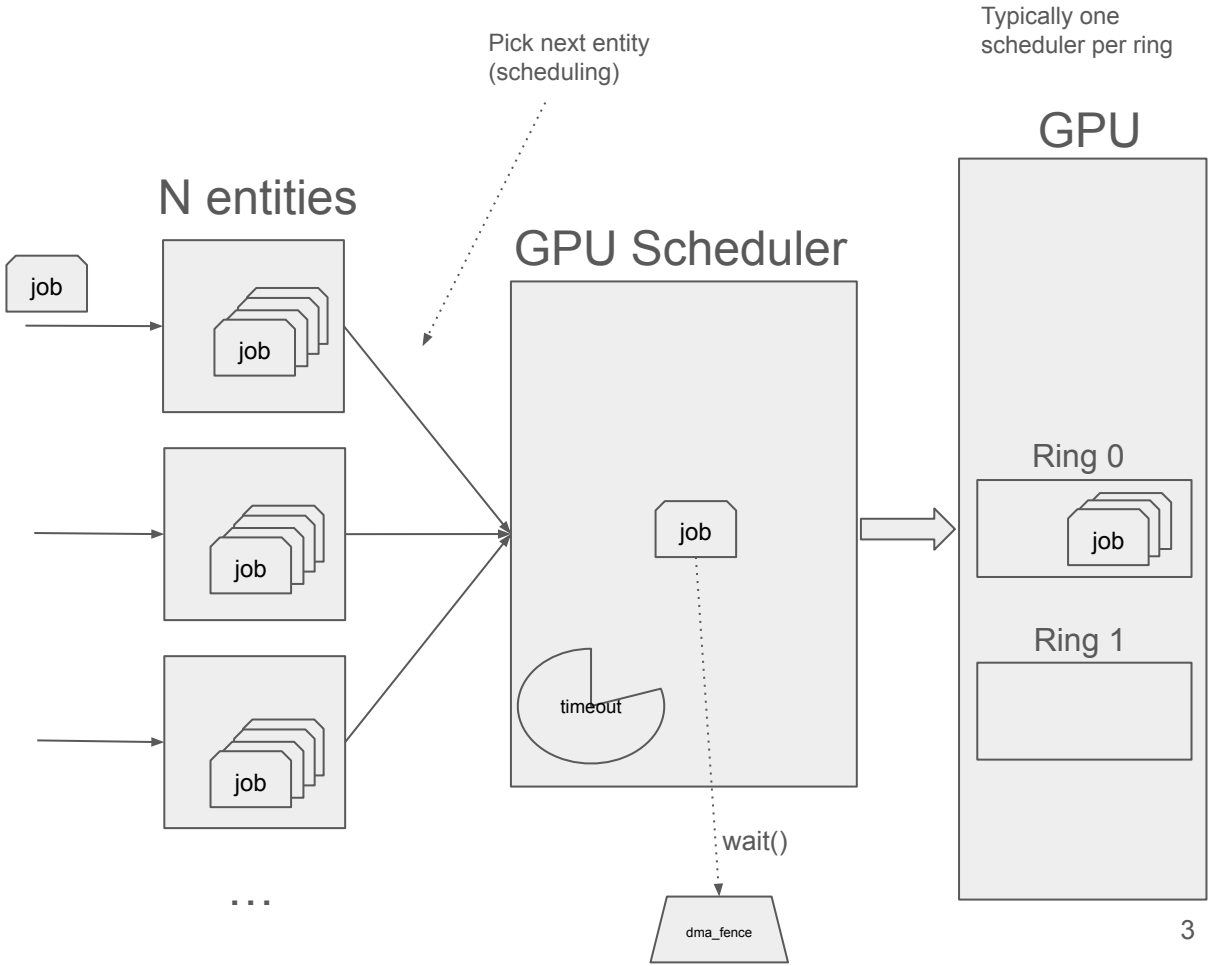
What's the Scheduler?

Load-Balancer: GPU doesn't have infinite capacity for jobs.

Dependency-Manager: runs jobs in (per-entity) submission order, but waits until a job's (external) dependencies are fulfilled (e.g., a VM_EXEC job only runs after an associated VM_BIND is finished).

"**Scheduler**" because it schedules between entities (job containers), each of which can correspond to a userspace counterpart. Can't interrupt jobs like a **C**PU scheduler can.

Timeout handler: Guarantees forward progress in case a job takes too long / forever.



Timeouts

Broken `while(true)` job blocks ring forever

Scheduler must guarantee forward progress.

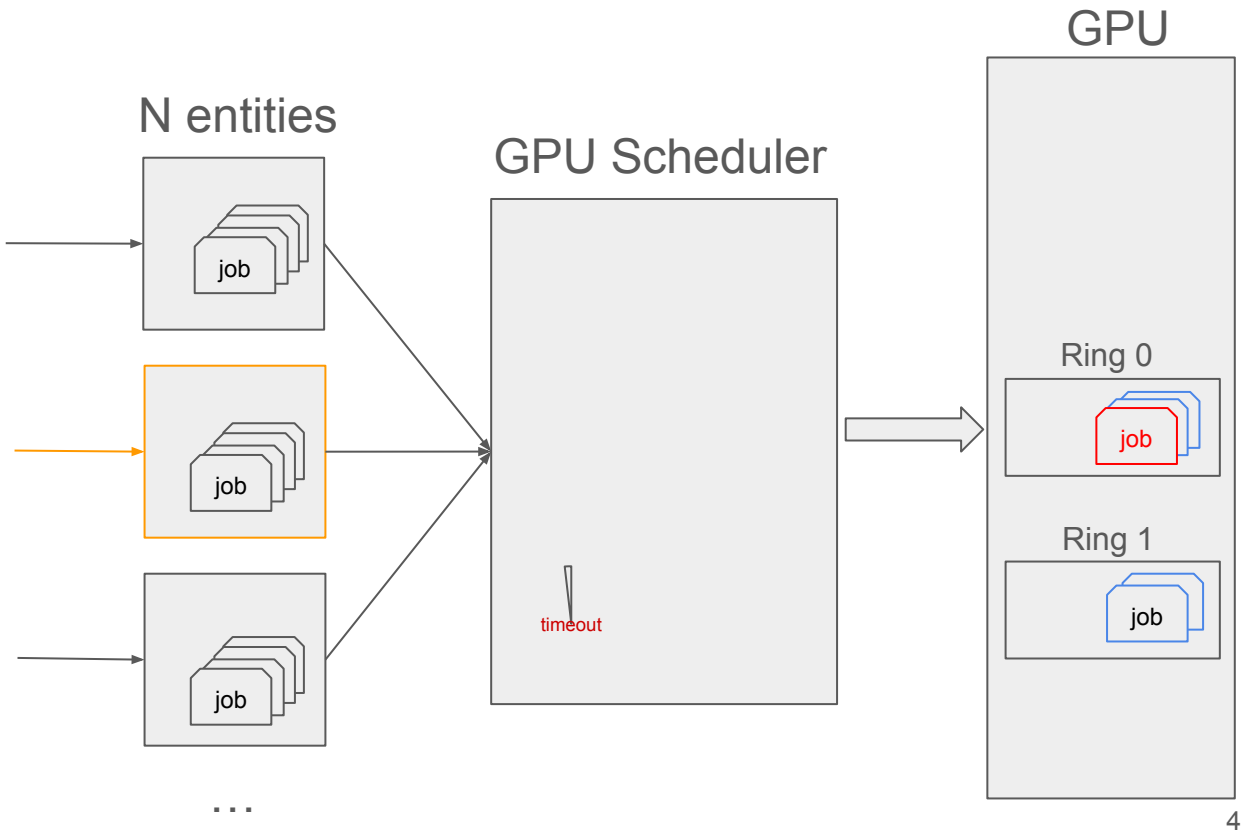
Scheduler can't just cancel the broken job on the GPU.

GPU needs to be **reset**, killing all "innocent" jobs in the rings.

Then, entity the job stemmed from has to be killed, with the associated userspace handle.

Resubmitting innocent jobs is difficult. Scheduler currently has no infrastructure for that (deprecated).

Timeouts can be **false-positives**: GPU didn't hang, was just slow.



Many (solvable?) problems

"My God, it's full of race conditions!"
– 2024: A Kernel Odyssey

- In general, scheduler issues are / were:
  - Race conditions
  - Broken (and even missing) locking
  - dma_fence signalling issues
  - Refcounting issues
  - Legacy code (e.g., old scheduling policies. Currently cleaned up by **Tvrtko Ursulin (Igalia)**)
  - Missing documentation
  - No concept for job-resubmissions
  - Unclear / unenforced lifetimes of scheduler objects
- In case you're searching for work: Code base contains many FIXMEs :)

Problems 1: Abusing API-Internals

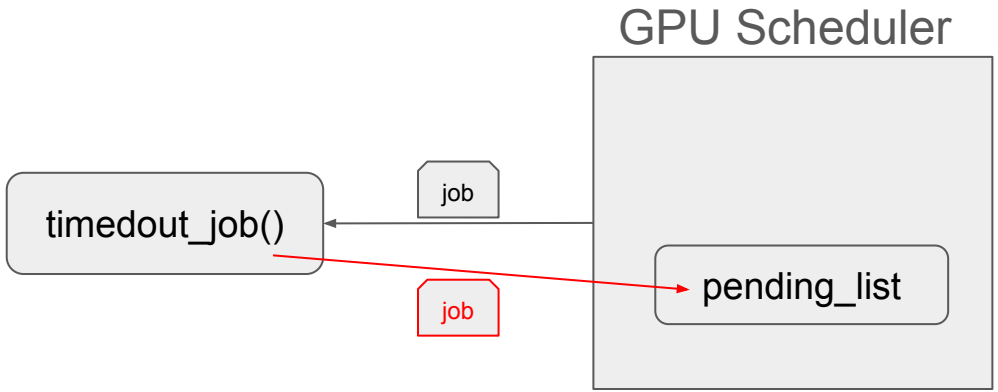
Scheduler informs driver via **timedout_job()** callback that a certain job caused a timeout.

Some drivers have mechanisms to detect that this was a false-positive timeout.

To the right:

Code from a driver's callback. Driver decides to just add the job back into the scheduler's **internal data structure**.

Without taking the list-lock! 😫




```
out_no_timeout:
        list_add(&sched_job->list, &sched_job->sched->pending_list);
        return DRM_GPU_SCHED_STAT_NOMINAL;
}
```

Unfortunately, there are **many** drivers that access various scheduler internals.

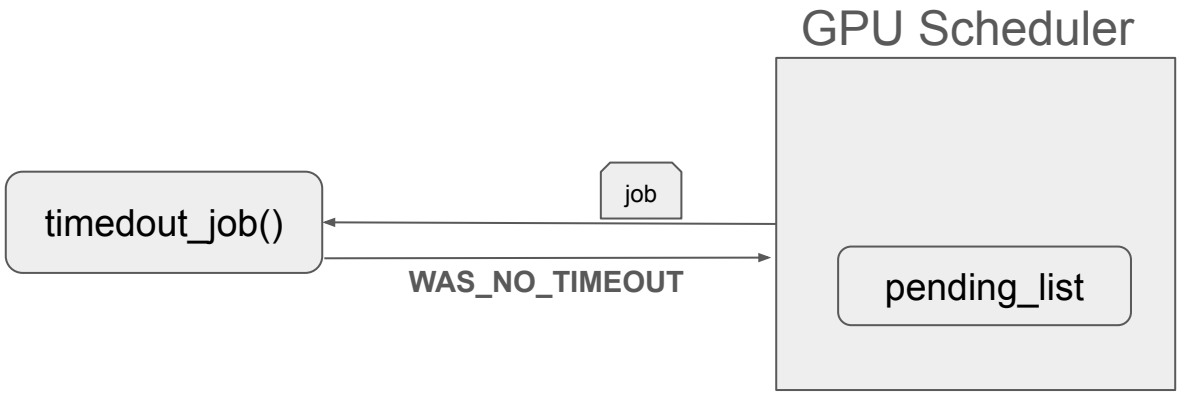
API Internals - Solution

Solved by **Maíra Canal (Igalia)** (61ee19dedb8d)

Timeout callback now informs Scheduler via **return code** about false-positives.

Problem was solvable in central infrastructure without too much effort.

⇒ Desirable behavior in DRM :)



Problems 2: Lack of Documentation

Scheduler code line calls into the driver callback.

Driver is expected to return a dma_fence with its reference count set to >= 2: One for itself, one for Scheduler.

Scheduler cannot do that itself (racy).

This was not documented.

```
fence = sched->ops->run_job(sched_job);
complete_all(&entity->entity_idle);
drm_sched_fence_scheduled(s_fence, fence);

if (!IS_ERR_OR_NULL(fence)) {
      /* Drop for original kref_init of the fence */
      dma_fence_put(fence);

      r = dma_fence_add_callback(fence, &sched_job->cb,
                                 drm_sched_job_done_cb);
```

Problems 3: Code Quality

"Implicit refcounting" is used in the Scheduler.

"I know that there is a reference still around somewhere, so this is fine."

The code on the right is not a "bug", i.e., it cannot fault. But it's still not good :(

```
fence = sched->ops->run_job(sched_job);
complete_all(&entity->entity_idle);
drm_sched_fence_scheduled(s_fence, fence);

if (!IS_ERR_OR_NULL(fence)) {
      /* Drop for original kref_init of the fence */
      dma_fence_put(fence);

      r = dma_fence_add_callback(fence, &sched_job->cb,
                                 drm_sched_job_done_cb);
```

^ UAF?!?

Unsolvable Problems

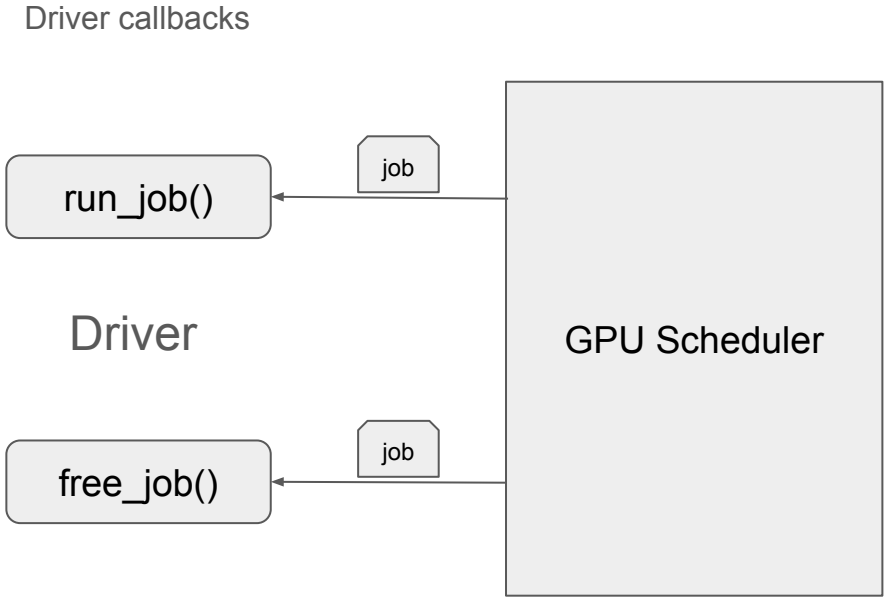
Scheduler pushes jobs to GPU with **run_job()**. Once Scheduler is done with the job, it frees it with **free_job()**.

Consequence: Scheduler de facto responsible for *job lifetime*. But the driver *allocates* the jobs…

free_job() is bad because of

- potential races with driver
- potential memory leaks
- more work to implement

Also, it's *unnecessary:* The Scheduler should just be a queue for jobs.
(Idea: **Christian König (AMD)**)



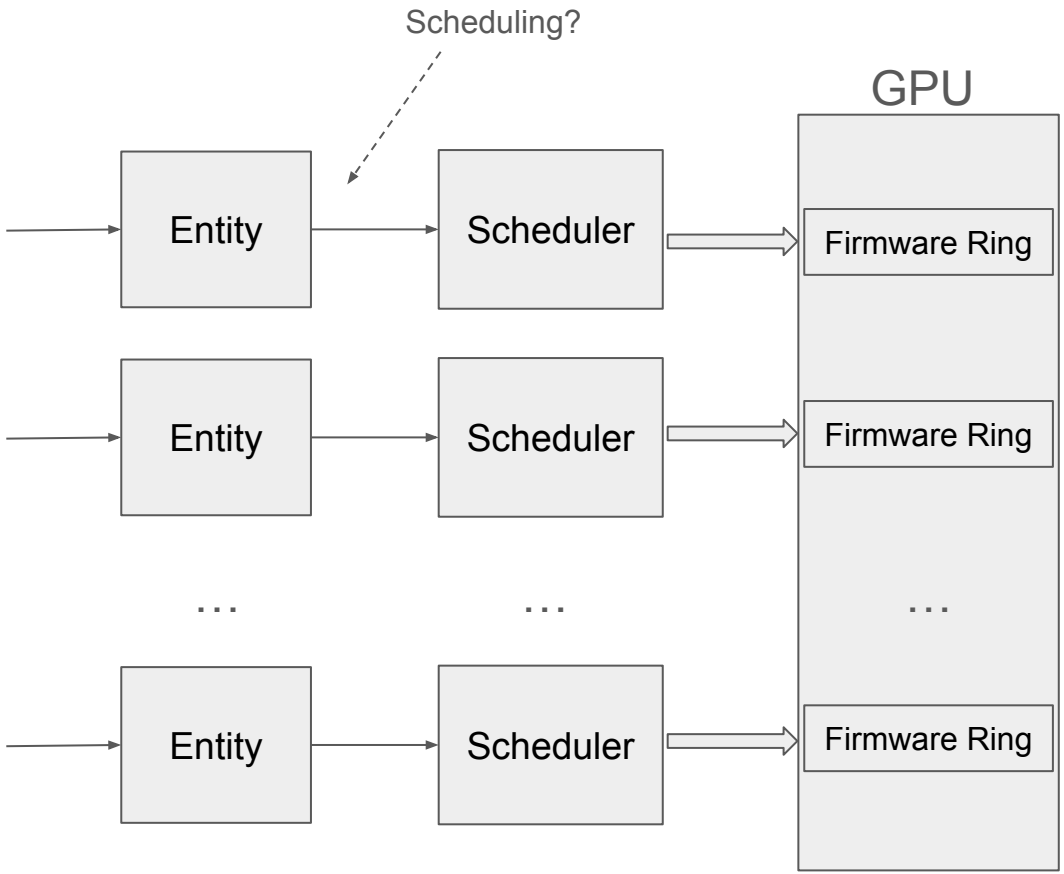
Non-Scheduling “Scheduler”

Many GPUs don’t have *hardware rings* anymore, but an arbitrary number of *firmware rings*.

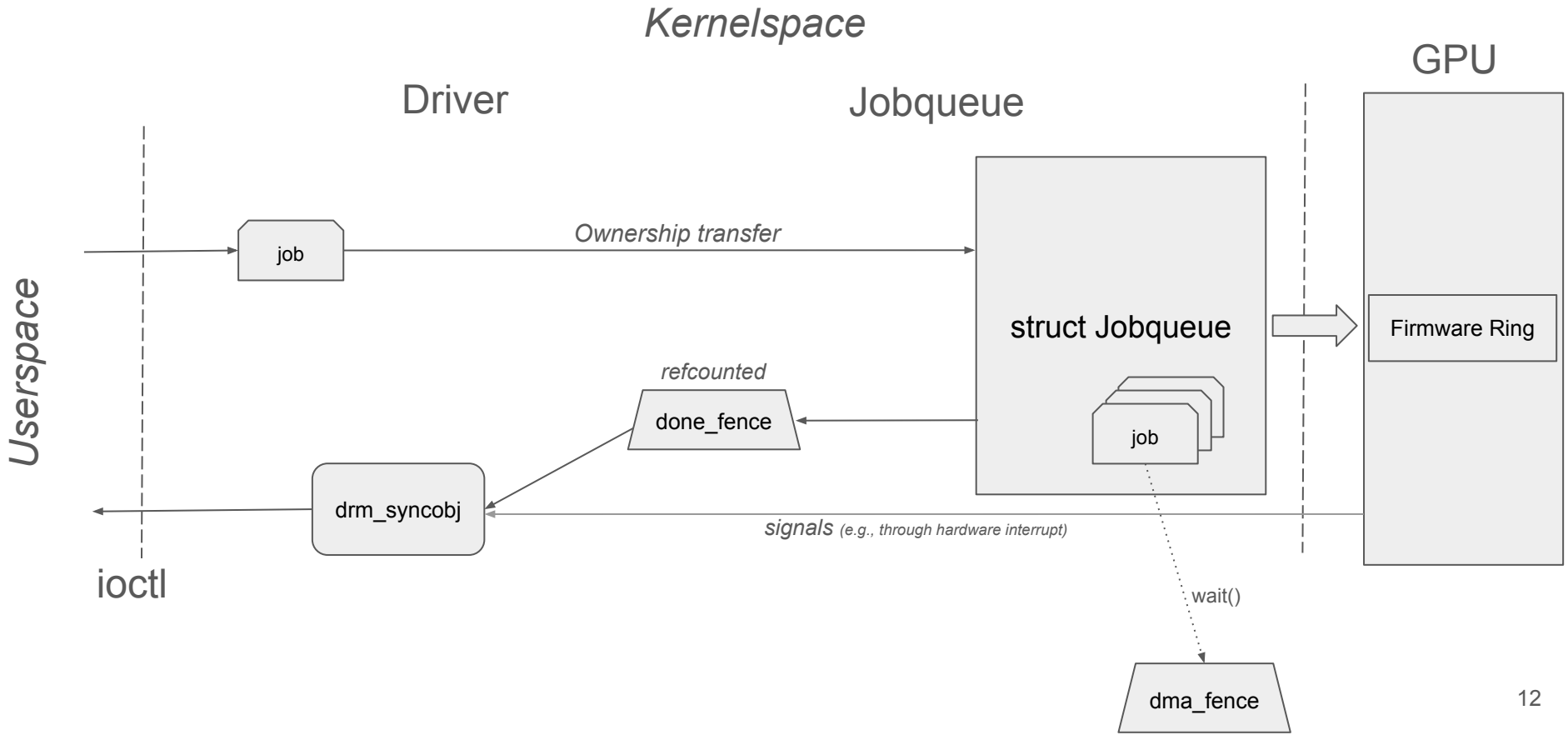
Rework by **Matthew Brost (Intel)** in 2023: Allow a 1:1 entity:scheduler relationship so drm_sched is usable with firmware rings.

Back then seemed like a good idea, but:

Scheduler now is not a scheduler anymore (for many drivers).



DRM Jobqueue



Idea

- New hardware / drivers do firmware scheduling:
  Nova, Tyr, Asahi, …
- We just need a “job queue”, not a “scheduler”
- A job queue can
  - leave drm_sched’s legacy problems behind
  - leave unnecessary features behind (red-black-tree for scheduling, job-resubmits on timeout, entities, …)
  - take lessons from 10 years of drm_sched

Programming Language: Rust

Why Rust?

- Nova, Tyr, Asahi are written in Rust already
- Rust can help with drm_sched-like problems: UAFs, refcounting, clear ownership rules
- Strong type system can prevent drivers from misusing APIs (e.g., touching internal lists)

Without a (Rust) Jobqueue, those drivers would need Rust abstractions on top of the “broken” drm_sched.

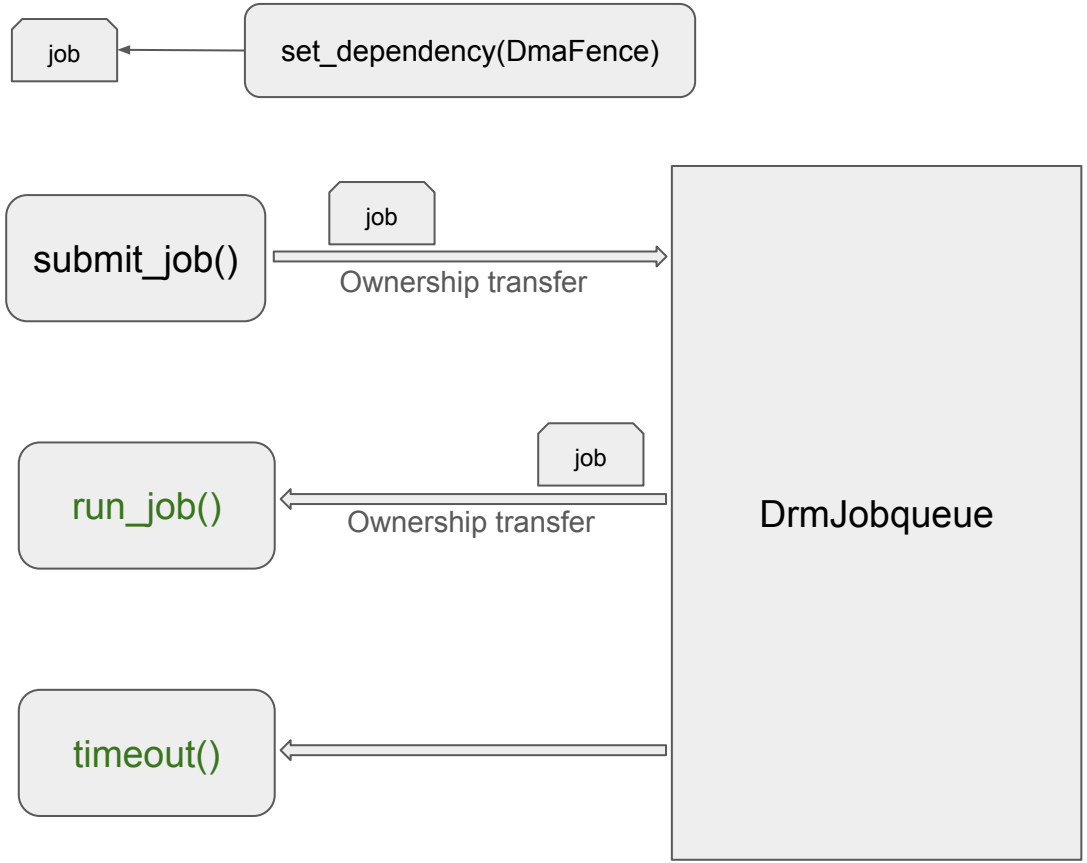
Functions and Driver Callbacks

set_dependency() specifies a DmaFence the Jobqueue has to wait to get signalled before running that job.

Driver creates a job and submits it through **submit_job()**. This transfers ownership to the Jobqueue.

Once it's time, Jobqueue calls driver's **run_job()** callback, transfering the ownership back.

In case, Jobqueue instructs the driver to take appropriate timeout actions through **timeout()**.



Advertisement

Workshops

- GPU Recovery: 30 Sep 2025, 11:25
- GPU Scheduler: 1 Oct 2025, 14:05

Summary

- drm_sched accumulated many problems over 10 years
- Some are solvable
- Some aren’t with reasonable effort (object lifetimes, locking?)
- Many drivers just need a job queue, not a scheduler
- DRM Jobqueue in Rust: successor *only* for firmware scheduling
- Drivers are accessing API internals…

Many thanks to:

- Danilo Krummrich
- the scheduler contributors
- the XDC organizers

Appendix (Not part of talk)

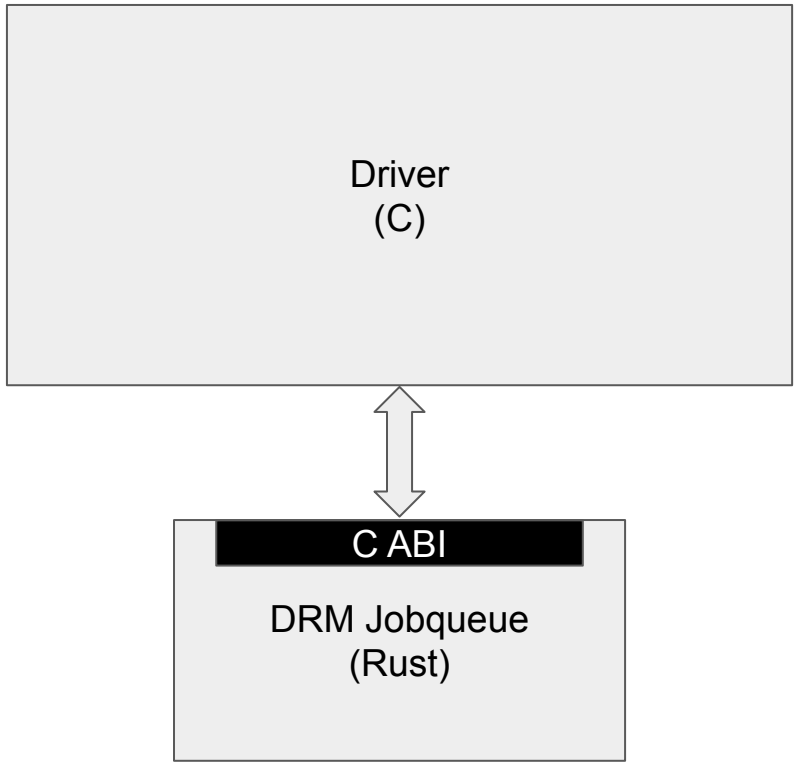
Distant Future

Rust can expose C-ABI functions.

Hypothetically, existing C drivers *with firmware scheduling* could be ported from drm_sched to drm_jobqueue.

Benefits:

- Fewer legacy problems
- drm_sched could focus more on hardware scheduling.



Lack of Documentation

```
/**
 * drm_sched_start - recover jobs after a reset
 *
 * @sched: scheduler instance
 * @full_recovery: proceed with complete sched restart
 *
 */
void drm_sched_start(struct drm_gpu_scheduler *sched, bool full_recovery)

/**
 * drm_sched_wqueue_start - start scheduler submission
 *
 * @sched: scheduler instance
 */
void drm_sched_wqueue_start(struct drm_gpu_scheduler *sched)
```

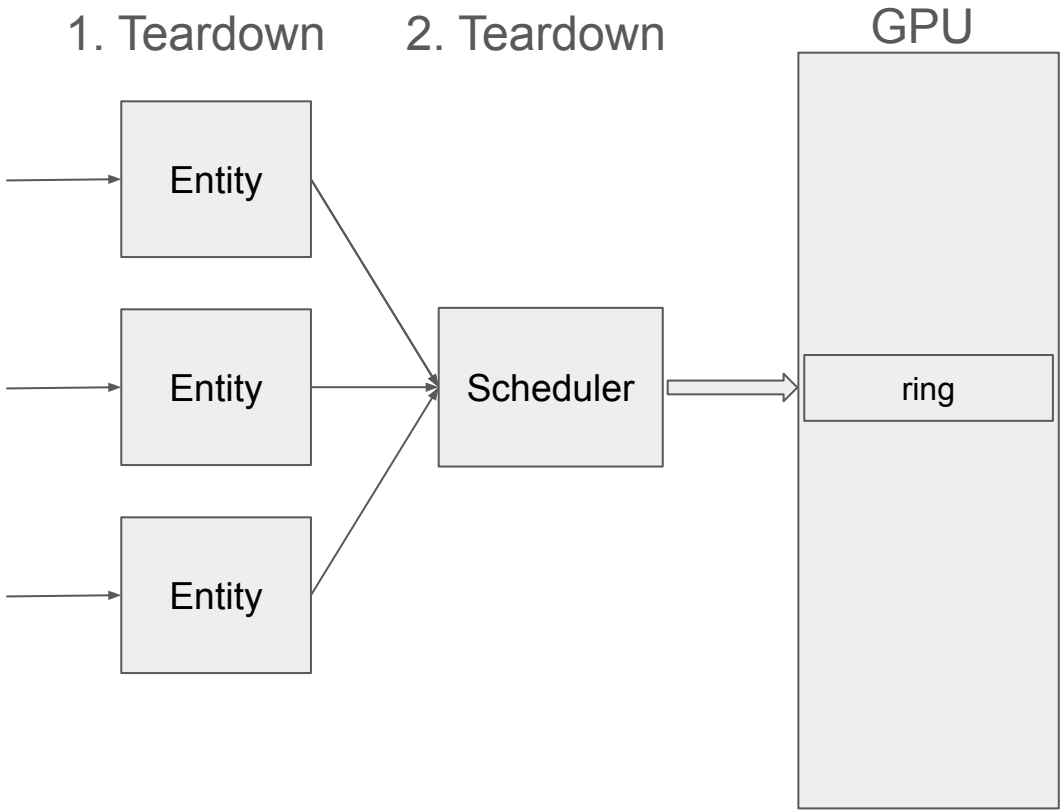
Plot twist: These functions aren’t necessary to *start* the scheduler during driver init. Only needed for GPU resets.
The actual “start function” is not called “_start”: `drm_sched_init()`

Lifetime Issues

Entities must not live longer than the Scheduler.

This rule “is in the code” and was clearly intended by drm_sched’s designers, but was neither **documented**, nor **enforced** in DRM**.**

Some drivers seem(ed?) to have outliving entities.



```
void drm_sched_fini(struct drm_gpu_scheduler *sched)
{
        struct drm_sched_entity *s_entity;
        int i;

        drm_sched_wqueue_stop(sched);

        for (i = DRM_SCHED_PRIORITY_KERNEL; i < sched->num_rqs; i++) {
                struct drm_sched_rq *rq = sched->sched_rq[i];

                spin_lock(&rq->lock);
                list_for_each_entry(s_entity, &rq->entities, list)
                        /*
                        * Prevents reinsertion and marks job_queue as idle,
                        * it will be removed from the rq in drm_sched_entity_fini()
                        * eventually
                        */
                        !!!ENTITY LOCK IS MISSING!!!
                        s_entity->stopped = true;
                spin_unlock(&rq->lock);
                kfree(sched->sched_rq[i]);
        }

. . .
```

```
void drm_sched_entity_push_job(struct drm_sched_job *sched_job)
{
        [...]

        /* first job wakes up scheduler */
        if (first) {
                struct drm_gpu_scheduler *sched;
                struct drm_sched_rq *rq;

                /* Add the entity to the run queue */
                spin_lock(&entity->lock);
                if (entity->stopped) {
                        spin_unlock(&entity->lock);

                        DRM_ERROR("Trying to push to a killed entity\n");
                        return;
                }

        rq = entity->rq;
        sched = rq->sched;

        spin_lock(&rq->lock);
        drm_sched_rq_add_entity(rq, entity);

. . .
```

A driver programmer was working around race conditions in a driver.
Apparently, the driver violated (violates?) the (undocumented) rule of entities having to be torn down before their scheduler.

That driver problem was addressed in `drm_sched_fini()` – but a lock was *forgotten* (*cough cough*). It's still broken:

- It's still racy because a lock is missing
- The lock **cannot be taken** because that would lead to lock-inversion ⇒ **deadlock**
- If the driver would follow the life time rules, no locks in drm_sched_fini() were necessary in the first place

That's one of the hard-to-solve scheduler problems. Which drivers follow the rules? Check them all, repair the broken ones, then remove that workaround from `drm_sched_fini()`.

General Code Problems

One example: problematic lock names

Solved by **Tvrtko Ursulin (Igalia)** (f93126f5d559)

```
spin_lock(&entity->rq_lock);
spin_lock(&entity->rq->lock);
```

Timeouts

Broken `while(true)` job blocks ring 0 forever

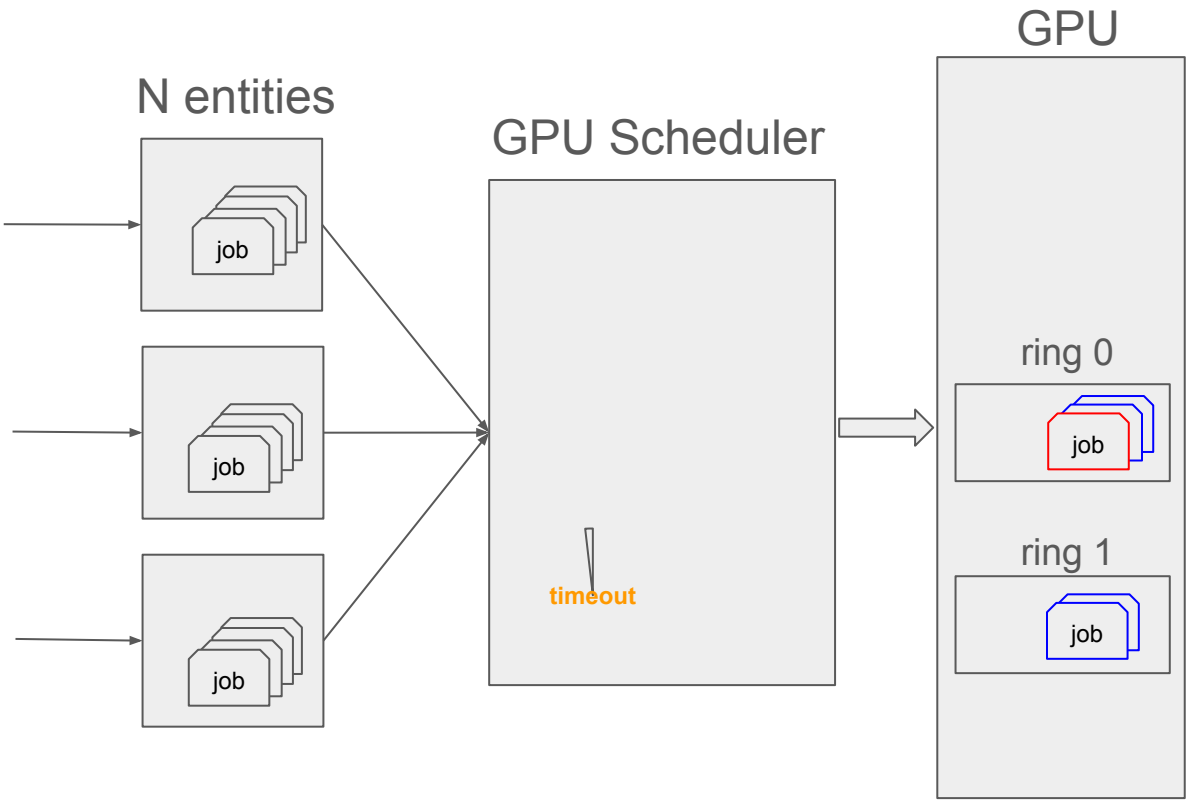
Scheduler must guarantee forward progress.

Problem: Linux kernel is not in charge of the “programs”. All controlled by userspace.

Scheduler can’t just cancel the broken job on the GPU.

GPU needs to be **reset**, killing all “innocent” jobs.

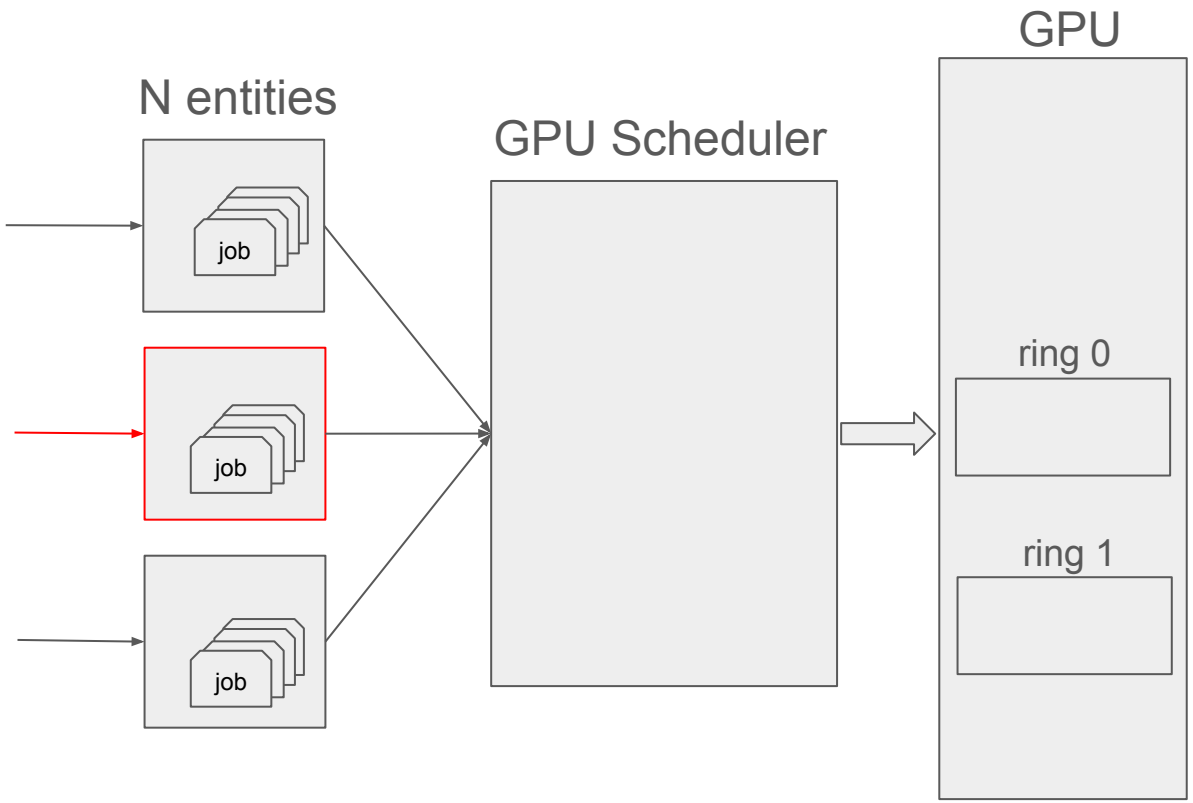
Innocent jobs on ring 1, served by a different scheduler (not depicted), get killed, too.



GPU got reset

Timeouts

1. Stop the scheduler.
2. Reset the GPU
3. Find the entity the guilty job belongs to.
4. Kill that entity. That will affect the associated userspace party.



Timeouts

1. Start the scheduler again
2. Ideally resubmit other “innocent” jobs that died on the GPU when it was reset.

Problems:

- Timeout handling is **very complicated**.
- Racy, can trigger **false positives**: Scheduler indicates timeout, but GPU load was just very high etc.
- There is no clear solution for resubmitting jobs: `drm_sched_resubmit_jobs()` currently deprecated).