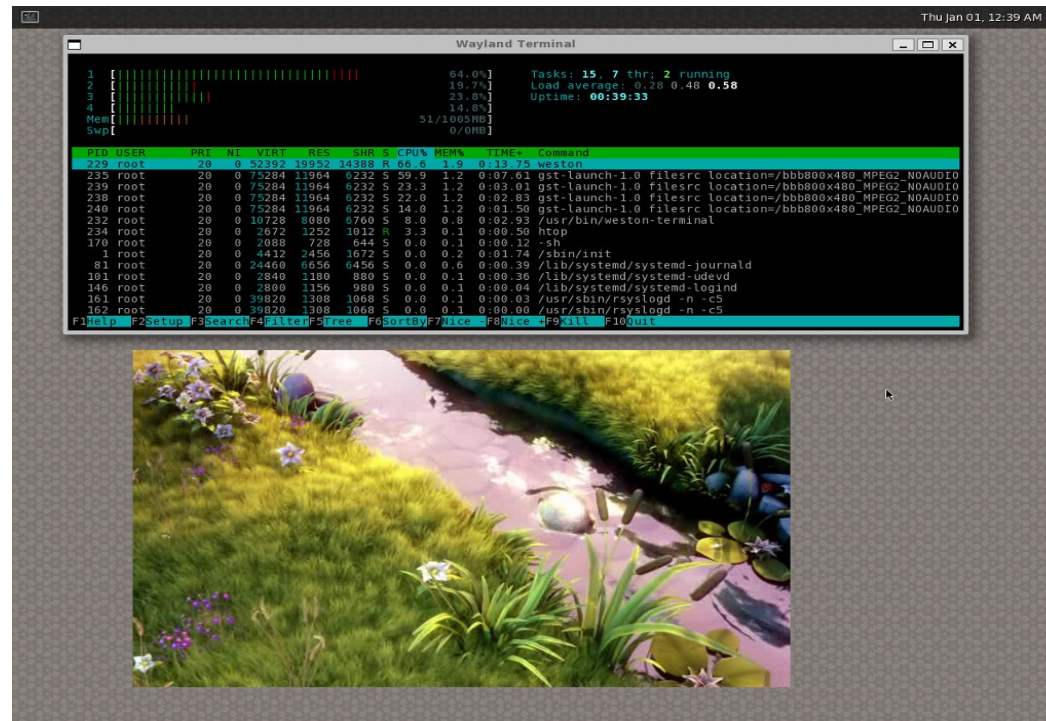


# Next-Generation DMABUF

## How To Efficiently Play Back Video on Embedded Systems




Wayland Terminal

```
Thu Jan 01, 12:39 AM
```

```
1 [|||||] 64.0%] Tasks: 15. 7 thr; 2 running
2 [|||||] 19.7%] Load average: 0.28 0.48 0.58
3 [|||||] 23.8%] Uptime: 00:39:33
4 [|||||] 14.8%]
Mem [|||||] 51/1005MB]
Swp [|||||] 0/0MB]
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
220	root	20	0	52392	19357	14388	R	66.6	1.9	0:18.75	weston
235	root	20	0	75284	11964	6232	S	59.3	1.2	0:07.01	gst-launch-1.0 filesrc location=/bbb800x480_MPEG2_NOAUDIO
239	root	20	0	75284	11964	6232	S	23.3	1.2	0:03.01	gst-launch-1.0 filesrc location=/bbb800x480_MPEG2_NOAUDIO
238	root	20	0	75284	11964	6232	S	22.0	1.2	0:02.83	gst-launch-1.0 filesrc location=/bbb800x480_MPEG2_NOAUDIO
240	root	20	0	75284	11964	6232	S	14.0	1.2	0:01.50	gst-launch-1.0 filesrc location=/bbb800x480_MPEG2_NOAUDIO
232	root	20	0	10728	8080	6760	S	5.0	0.8	0:02.93	/usr/bin/weston-terminal
234	root	20	0	2672	1252	1012	R	3.3	0.1	0:00.50	htop
170	root	20	0	2088	728	644	S	0.0	0.1	0:00.12	-sh
1	root	20	0	4412	2456	1672	S	0.0	0.2	0:01.74	/sbin/init
81	root	20	0	24460	6656	6456	S	0.0	0.6	0:00.39	/lib/systemd/systemd-journald
101	root	20	0	2840	1180	880	S	0.0	0.1	0:00.36	/lib/systemd/systemd-udev
146	root	20	0	2800	1156	980	S	0.0	0.1	0:00.04	/lib/systemd/systemd-logind
161	root	20	0	39820	1308	1068	S	0.0	0.1	0:00.03	/usr/sbin/rsyslogd -n -c5
162	root	20	0	39820	1308	1068	S	0.0	0.1	0:00.00	/usr/sbin/rsyslogd -n -c5



Embedded Linux Conference Europe

Edinburgh, 2013-10-25

Lucas Stach <l.stach@pengutronix.de>

Philipp Zabel <p.zabel@pengutronix.de>



# Agenda

- Simple videoplayback using Gstreamer
- Adding hardwareunits in the mix
- DMA-BUF – why and how
- Current DMA-BUF flaws  
→ our solution

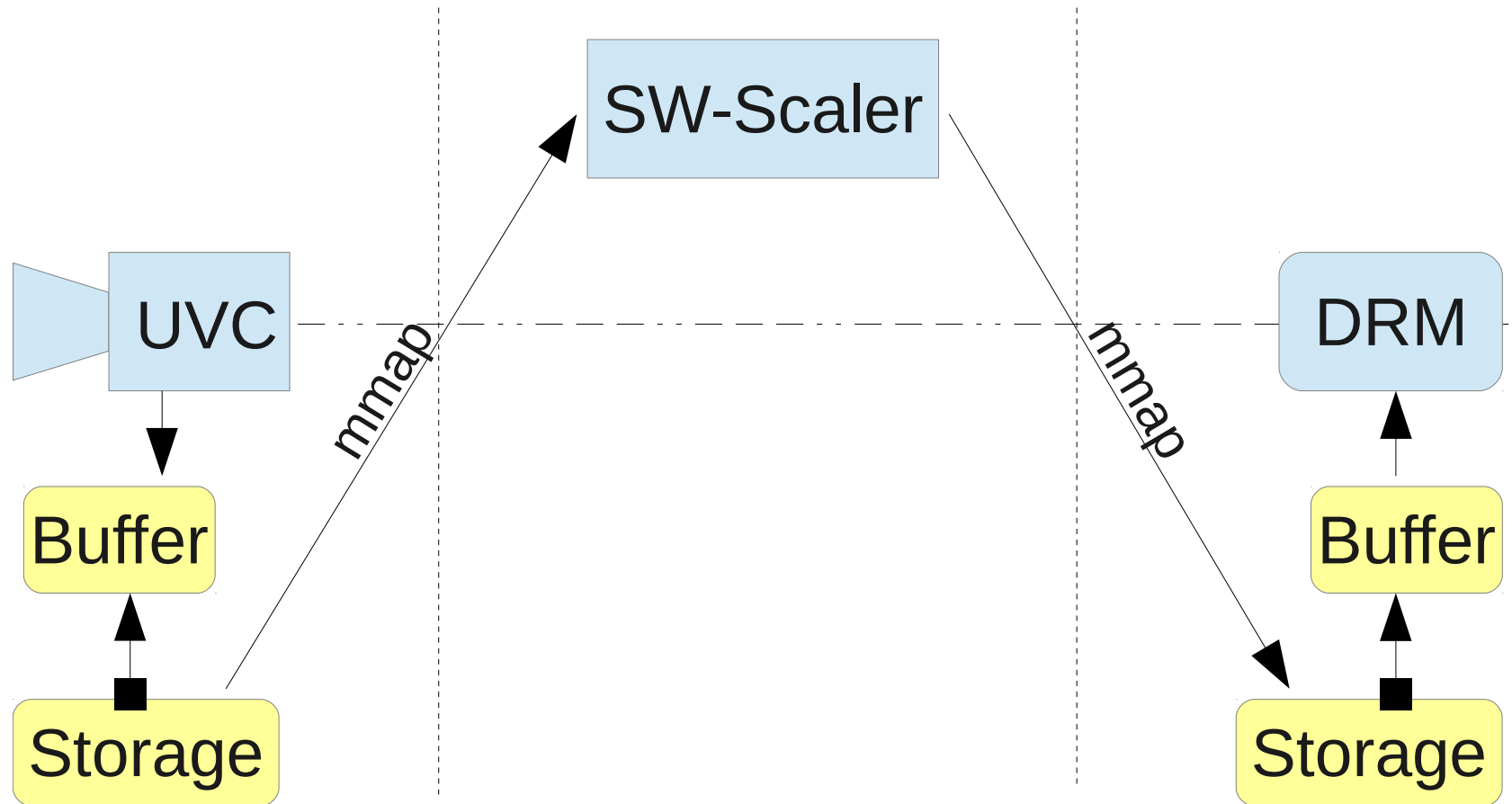




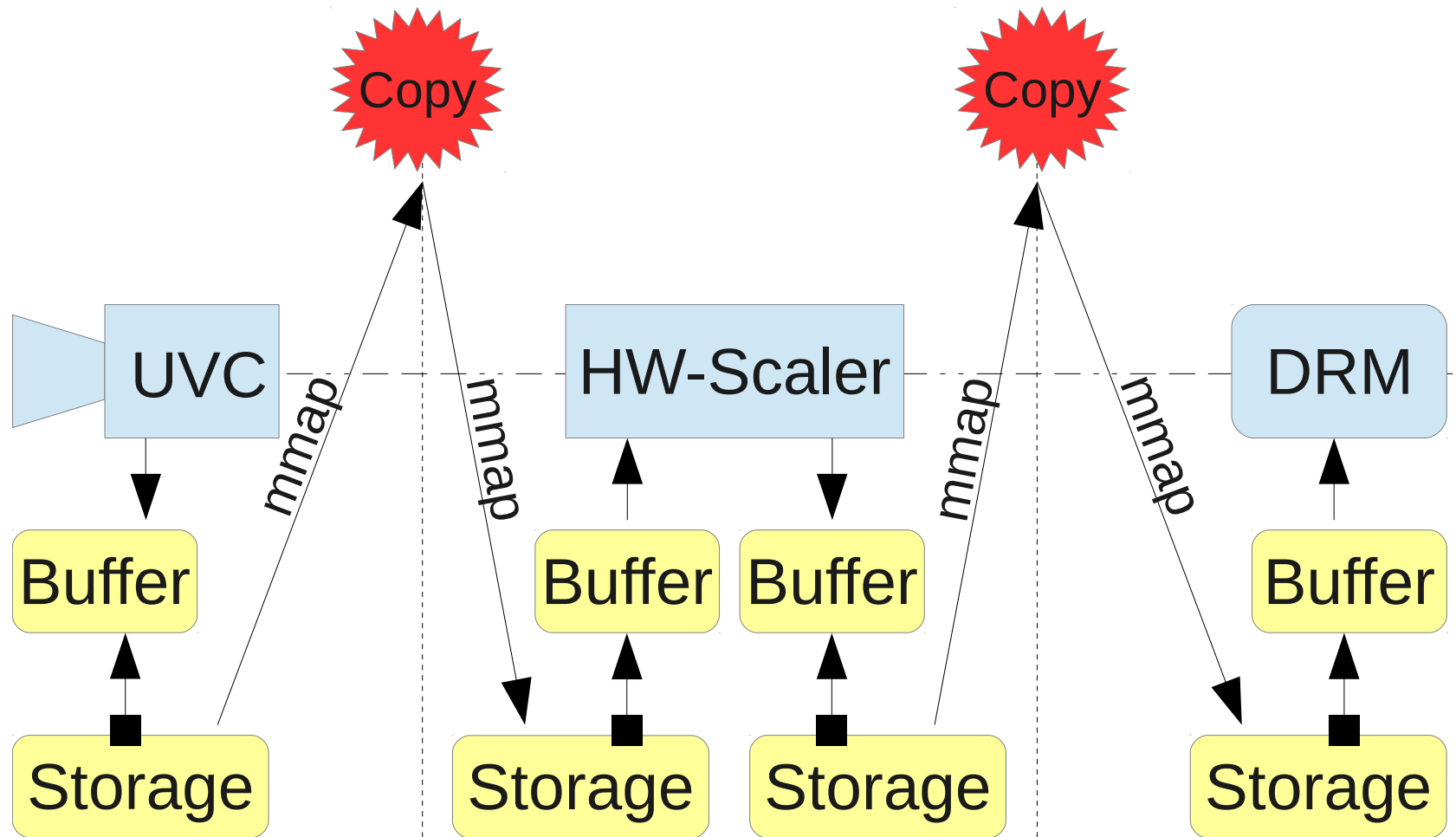
# *gstreamer*



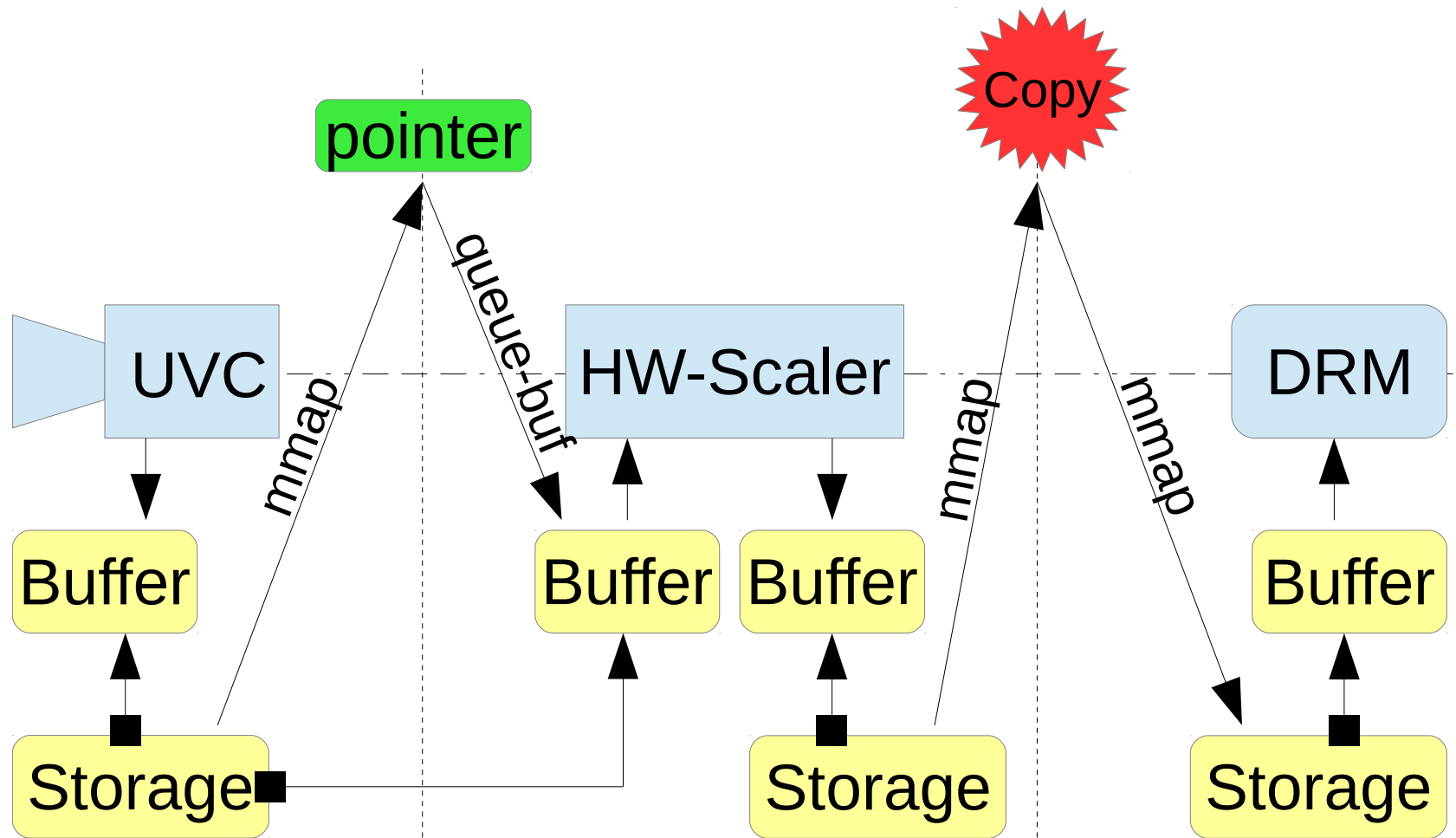
# GStreamer software pipeline



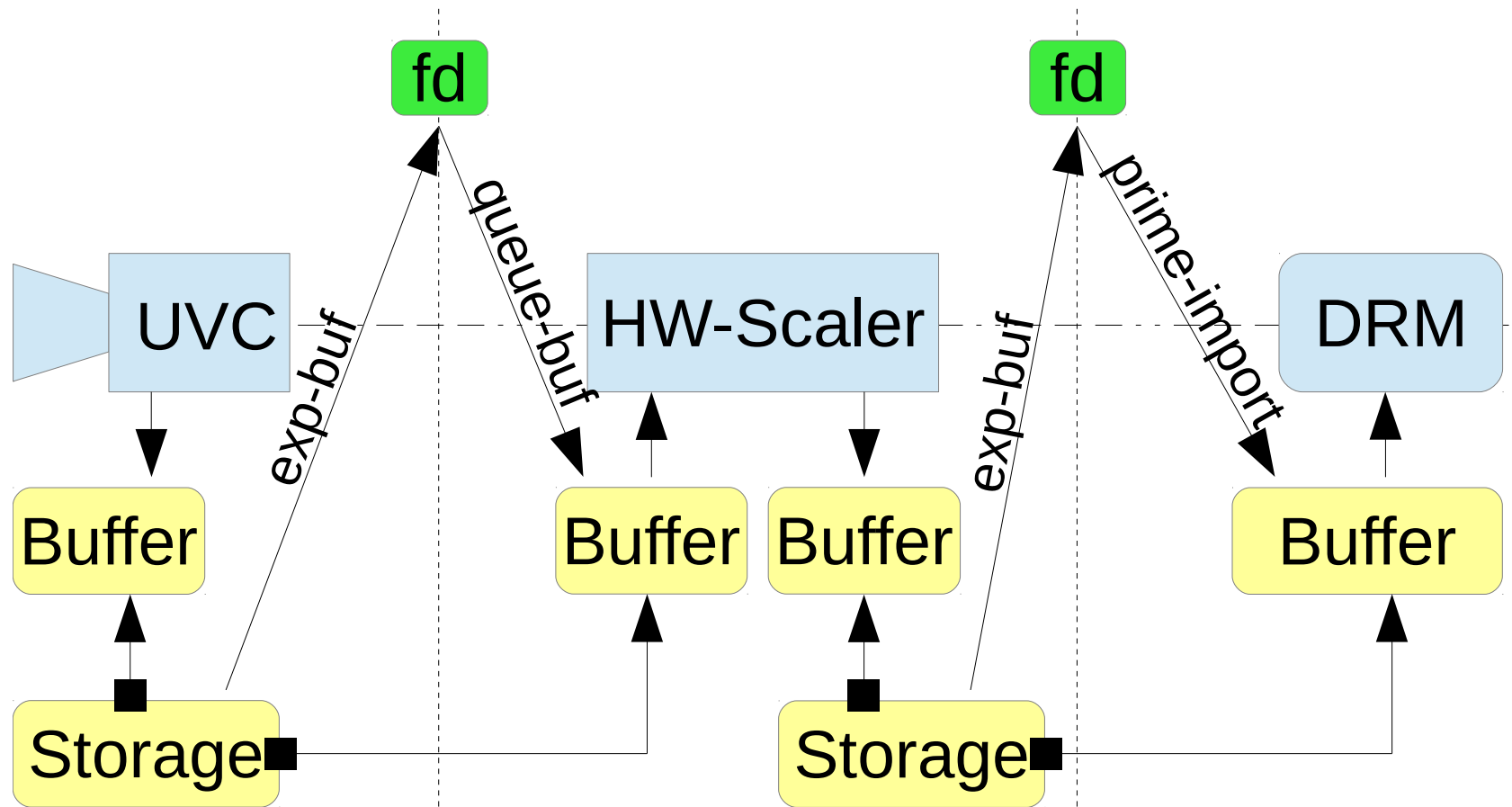
# Now add another HW element



# Video4Linux UserPTR



# Introducing DMABUF



# Fundamental DMABUF API

```
struct dma_buf_attachment *  
dma_buf_attach(struct dma_buf *dmabuf, struct device *dev);
```

```
struct dma_buf_attachment {  
    struct dma_buf *dmabuf;  
    struct device *dev;  
    struct list_head node;  
    void *priv;  
};
```

```
void dma_buf_detach(struct dma_buf *dmabuf,  
                   struct dma_buf_attachment *dmabuf_attach);
```





# Fundamental DMABUF API

```
struct sg_table *  
dma_buf_map_attachment(struct dma_buf_attachment *,  
                        enum dma_data_direction);
```

```
void  
dma_buf_unmap_attachment(struct dma_buf_attachment *,  
                          struct sg_table *,  
                          enum dma_data_direction);
```



Sounds like a good idea and reasonably easy,  
but ...



# Possible memory constraints

- different DMA windows
- contiguous vs. paged
- different MMU page sizes

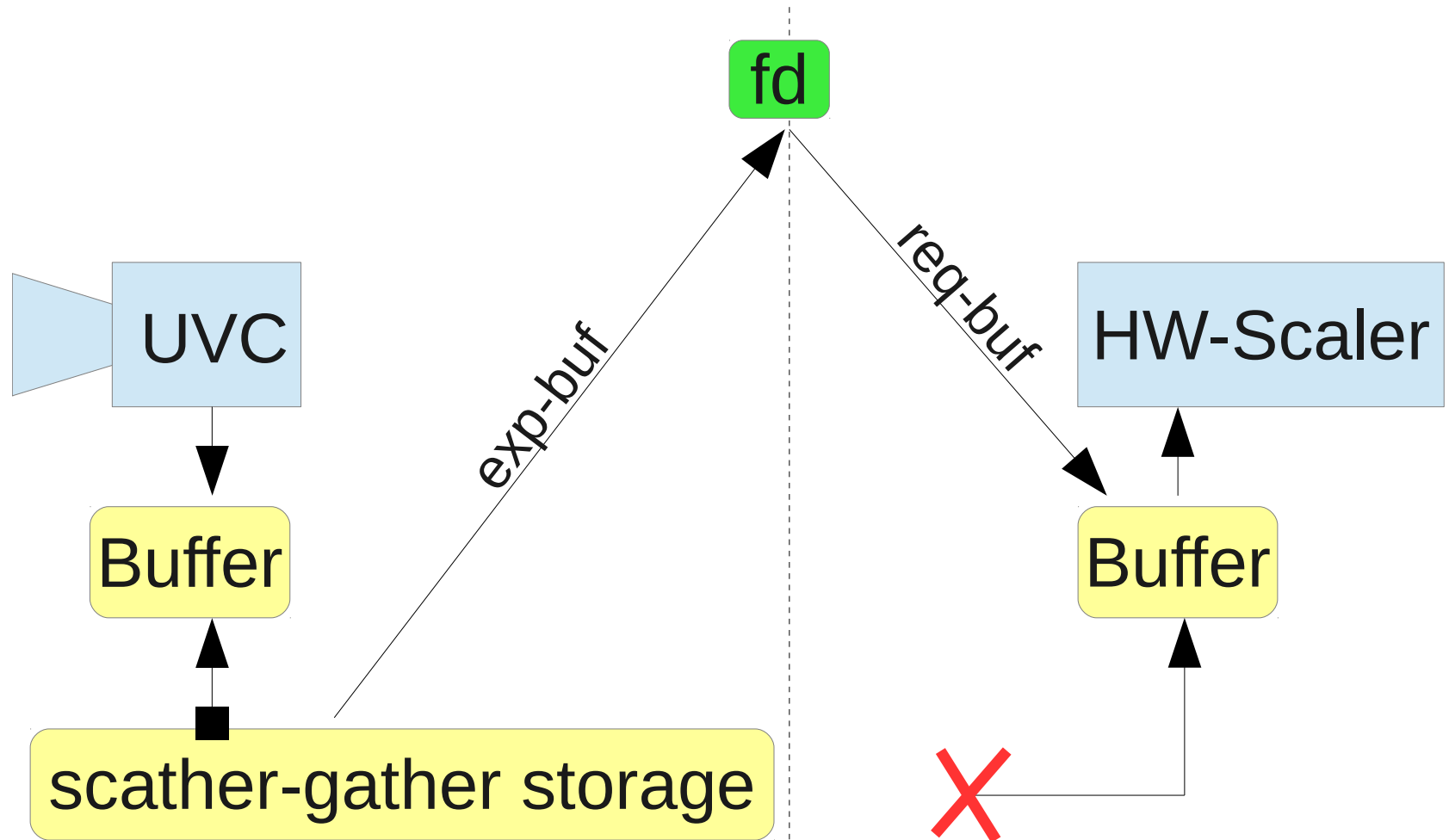


# Common restriction on embedded systems

- devices unable to do scatter-gather DMA
  - no IOMMU available
- DMA memory needs to be physically contiguous



# Mixed systems...



# Our solution

## Transparent backing store migration



# Prerequisites

- drivers need to be able to describe their device's DMA capabilities
- commonly known: `dma_mask`
- there's more:

```
struct device_dma_parameters {  
    unsigned int    min_segment_size;  
    unsigned int    max_segment_size;  
    unsigned long   segment_boundary_mask;  
    unsigned int    max_segments;  
};
```

# Prerequisites

- drivers need a more generic way for allocating backing store
- traditional DMA-API:

*void \**

```
dma_alloc_attrs(struct device * dev, size_t size, dma_addr_t  
               *dma_handle, gfp_t flag, struct dma_attrs * attrs)
```

What's wrong with that?





# Prerequisites

- new way to allocate DMA memory

```
int  
arm_dma_alloc_sgtable(struct device *dev, size_t size,  
                      struct sg_table *sgt, gfp_t gfp,  
                      struct device_dma_parameters *dma_parms);
```

```
struct sg_table {  
    struct scatterlist {  
        unsigned long page_link;  
        unsigned int length;  
        dma_addr_t dma_address;  
    } *sgl;  
    unsigned int nents;  
};
```



# Prerequisites

- map for device with well-known DMA-API

*int*

```
dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,  
           enum dma_data_direction dir, struct dma_attrs *attrs)
```

- map for CPU with new function

*void \**

```
dma_cpumap_sgtable(struct device *dev, struct sg_table *sgt,  
                  pgprot_t prot);
```



# Migration

- `dma_buf_map_attachment`
  - current storage compatible with attachment?
    - Yes
      - return `sg_table`
    - No
      - wait for other maps to go away
      - **reallocate storage**



# Reallocation

- try to find storage dma parameters compatible with all currently attached devices

*int*

```
dma_coalesce_constraints(int num_parms,  
                        struct device_dma_parameters **in_parms,  
                        struct device_dma_parameters *out_parms)
```

- if not possible use parameters from device currently trying to map and exporter only
- last resort: parameters from mapping device only
- use parameters to alloc new storage



# Migration

- `dma_buf_map_attachment`
  - current storage compatible with attachment?
    - Yes
      - return `sg_table`
    - No
      - wait for other maps to go away
      - reallocate storage
      - **move current content to new storage**



# Move buffer content

- simple and almost always working:
  - map both buffers to CPU
  - memmove()
- exporter is free to implement optimized move
  - examples:
    - GPU behind MMU can blit content
    - usage of dedicated on-chip DMA engines



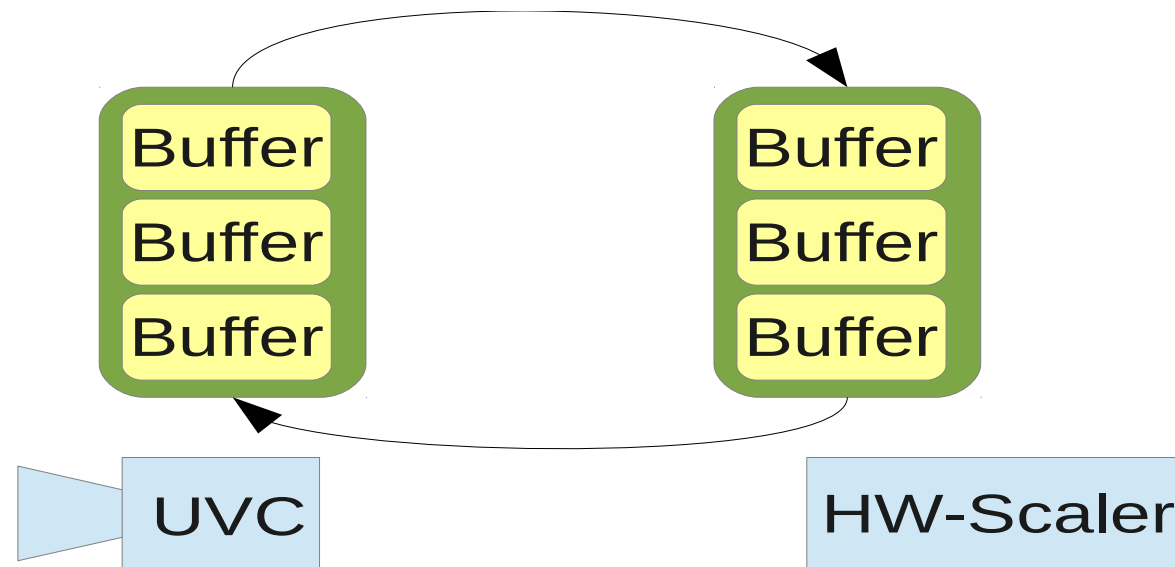
# Migration

- `dma_buf_map_attachment`
  - current storage compatible with attachment?
    - Yes
      - return `sg_table`
    - No
      - wait for other maps to go away
      - reallocate storage
      - move current content to new storage
      - return `sg_table` to new storage



# Why isn't this dead slow?

- GStreamer reuses allocated buffers – and you should too





## Corner cases

- sharing a buffer between devices with no overlap in `device_dma_parameters`
  - will work, but leads to ping-pong
- devices with memory not accessible to CPU and no way to migrate a buffer on it's own
  - Do you know of any real world example?
  - If you can't access a common memory region, why are you sharing a buffer?



# Possible optimization

- Delay allocation to last possible point in time  
→ alloc when first user wants to read/write
  - Userspace hands buffer handle to all devices before starting the pipeline  
→ all users attach before usage
- exporter is able to allocate matching storage right from the start

