Extending RTAI Linux with Fixed-Priority Scheduling with Deferred Preemption

Mark Bergsma, Mike Holenderski, Reinder J. Bril, Johan J. Lukkien

System Architecture and Networking
Department of Mathematics and Computer Science
Eindhoven University of Technology

This work was conducted within the ITEA CANTATA project at TU/e

Outline

- Context
- Problem description
- Solution direction: FPDS + reservations
- Extending RTAI Linux with FPDS
- Measurements
- Evaluation and conclusions

Context

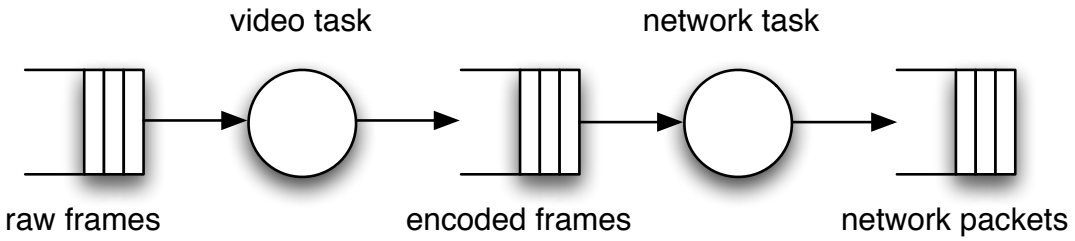
- Surveillance multimedia streaming application:
  - Computation and data intensive tasks
  - Dependent tasks (producer/consumer relation)
  - Networked system: several cameras + server
  - Cost-constrained system
- Camera platform
  - Processor + memory (cache, local, global) + bus + network
- Application: two main tasks
  - Video task (processor + memory + bus)
  - Network task (processor + memory + bus + network)

We conducted this work in the context of the CANTATA project, which has the aim of demonstrating surveillance multimedia streaming application.

The system is comprised of several cameras + server, communicating over the network.

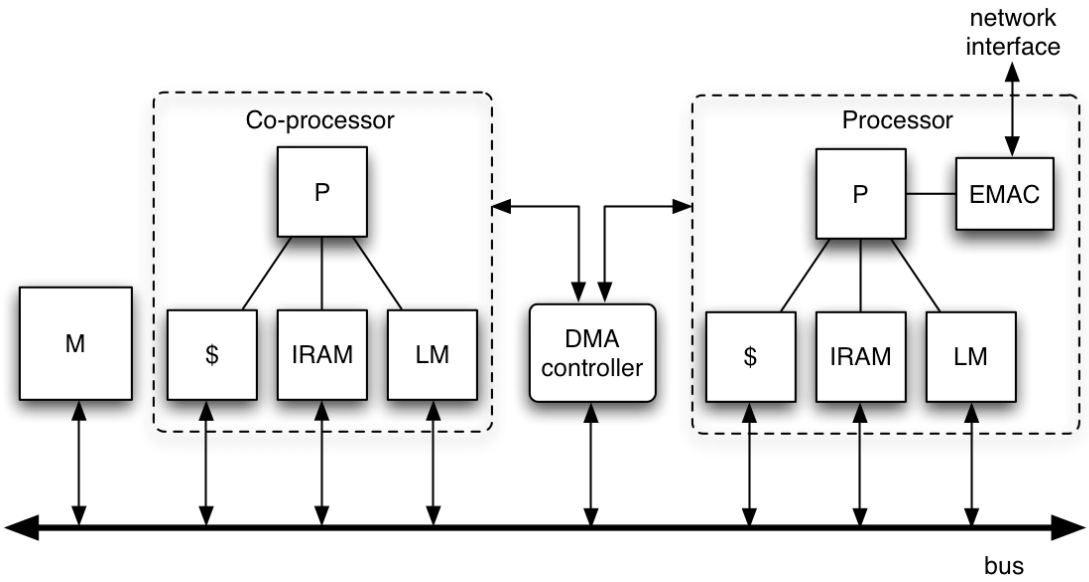
We can identify two main tasks in the application:
video processing task ... invoked periodically upon video frame arrival
video transmission task ... invoked by the video processing task

Application example



Here we have a picture of the application where video and network tasks **communicate** via **buffers** in the **memory**.

Platform example

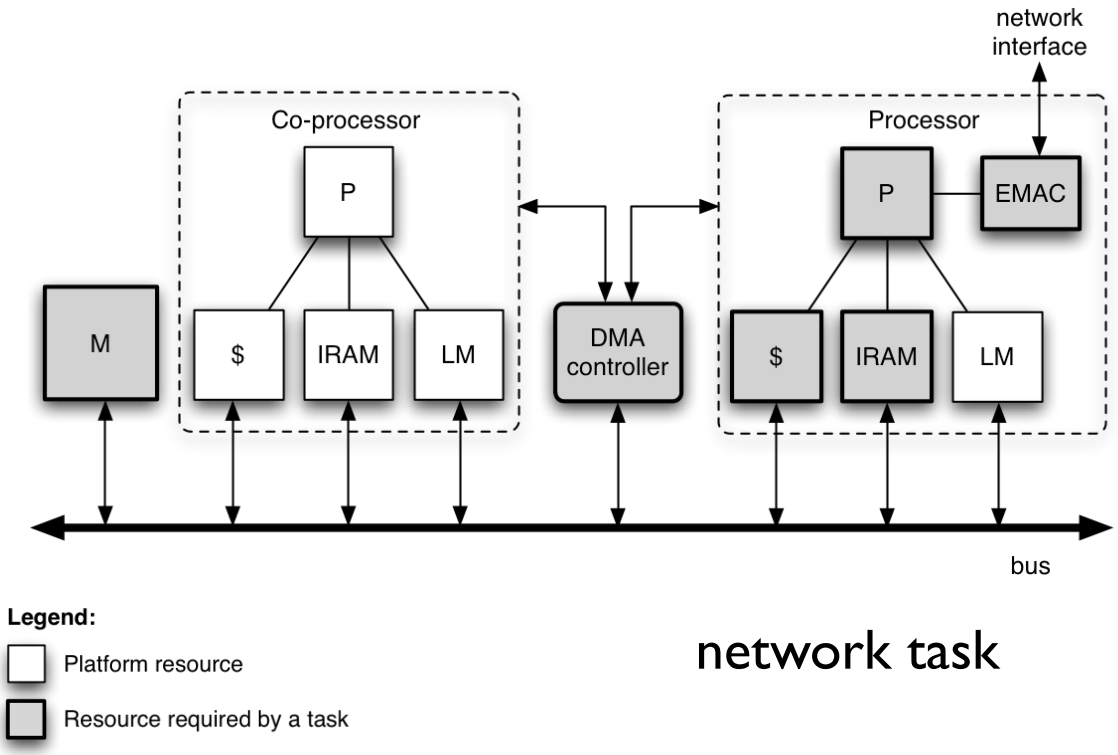


The tasks execute on a platform consisting of a processor, a co-processor and memory, connected via a bus with a DMA controller, which is used by the tasks for moving data between the memory and the processors.

Video task and Network task execute on the main Processor

They move data from main Memory to IRAM and LM (Local Memory) and back, using DMA channels.

Platform example



5

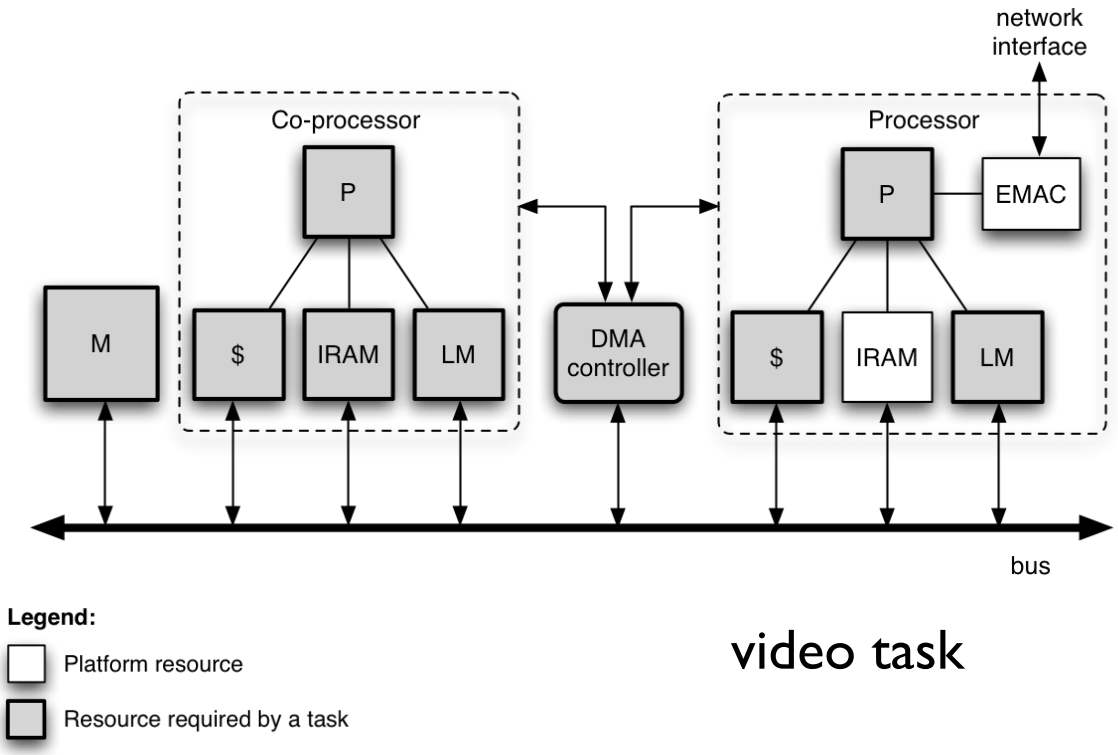
5

The tasks execute on a platform consisting of a processor, a co-processor and memory, connected via a bus with a DMA controller, which is used by the tasks for moving data between the memory and the processors.

Video task and Network task execute on the main Processor

They move data from main Memory to IRAM and LM (Local Memory) and back, using DMA channels.

Platform example



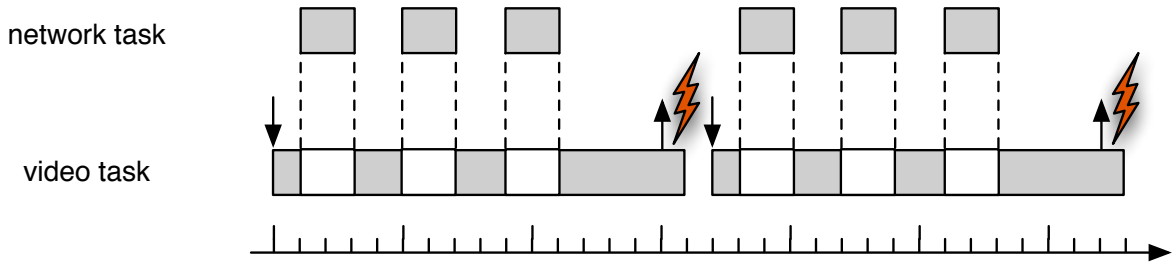
5

The tasks execute on a platform consisting of a processor, a co-processor and memory, connected via a bus with a DMA controller, which is used by the tasks for moving data between the memory and the processors.

Video task and Network task execute on the main Processor

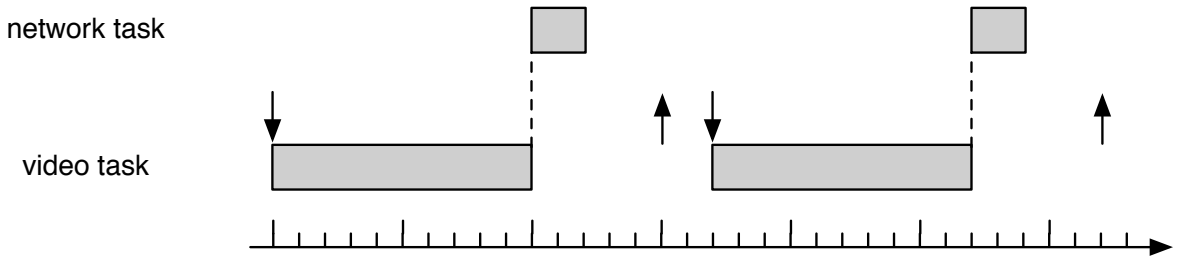
They move data from main Memory to IRAM and LM (Local Memory) and back, using DMA channels.

Problem illustration (FPPS)



Our industrial partner started with FPPS, however, high overheads due to the frequent preemptions, lead to deadline misses -> FPNS.

Problem illustration (FPNS)



This became problematic with fluctuating network bandwidth, e.g. due to congestion.

Since the network task is released **after** the **non-preemptive** video processing task has finished, it may be **assigned** the **processor** when the network is not available, and **therefore** it may not make optimal use of the processor

Problem

Video task is **greedy** and **non-preemptive**

+

Network availability **fluctuates** (e.g. congestion)

Network task cannot make optimal use of the **processor**

We can summarize the problem as: the combination of ... and fluctuating ... leading to the network task not being able to ...

Goal

- Optimize network usage while guaranteeing processor to the video processing task
  - by allowing limited preemptions of the video processing task, while guaranteeing its effectiveness

Solution direction

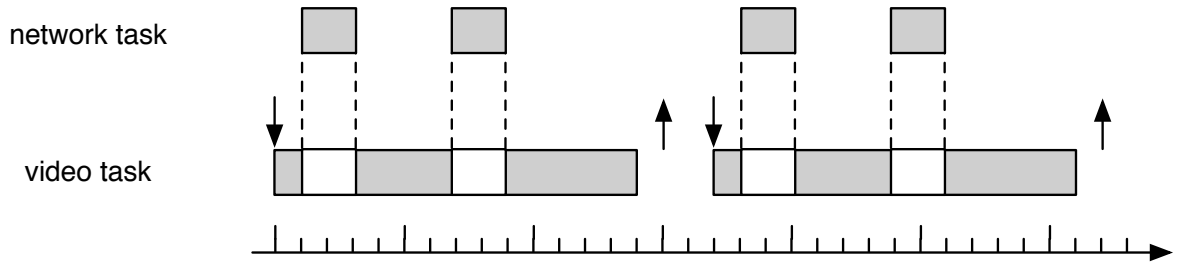
- **Network** task: increase the frequency of processor availability
  - Resolved by means of **FPDS**
    - Introduce preemption points, at appropriate places
      - minimizing context switch overhead
- **Video** task: still guarantee sufficient processing time
  - Resolved by means of **reservations**
    - Bound the interference of the network task

We want to match the bursty availability of the network with the resource provisioning to the network task.

Wrap the network task in a reservation driven by a deferrable or sporadic server (bandwidth preserving).

In this presentation we focus on FPDS, without going into detail about reservations.

Solution illustration



If the combination of FPDS and reservations is applied properly, everything works out.

Outline

- Context
- Problem description
- Solution direction: FPDS + reservations
- **Extending RTAI Linux with FPDS**
- Measurements
- Evaluation and conclusions

12

RTAI Linux

- Real-time extension of Linux
  - Supports FPPS
  - Primitives have a short and bounded latency
- Basis for the ITEA2/CANTATA framework, meant to be used by our industrial partner
- This work focuses on an **efficient** implementation of FPDS in RTAI Linux.

13

RTAI Architecture

- RTAI is a hypervisor between hardware and Linux
- RTAI scheduler
  - FPPS
    - Co-operative scheduling for tasks with equal priority
  - Linux scheduler is treated as a low priority soft task
  - Support for **periodic** and one-shot tasks, task suspension, timed sleep
  - Priority inheritance for shared resources

14

Intercepts all interrupts and dispatches them to the appropriate subsystems. In particular the timer interrupts, allowing RTAI to schedule real-time tasks and provide timeliness guarantees.

The hard real-time tasks are scheduled directly by RTAI, while the Linux scheduler takes care of scheduling soft tasks, which is itself treated as a low priority soft task.

RTAI Task

- Fixed priority, 16 bit, 0 the highest priority
- Periodic tasks implemented as a loop:

```
while (true) {
  ...
  rt_task_wait_period();
}
```

- Task Control Block (TCB)
  - Contains all administrative task info, including task state field
  - Resides in the kernel space

15

RTAI Scheduler

- *Ready queue*: priority queue sorted by task priority
- Periodic tasks reside in the *waiting queue* (called timed tasks): priority queue sorted by release time
- Scheduler invocation `rt_schedule()`:
  1. Update current time
  2. Move tasks from waiting to ready queue
  3. Select highest priority task from ready queue
  4. Context switch to the selected task
- Implemented by two similar functions:
  - `rt_timer_handler()` triggered periodically (by the timer)
  - `rt_schedule()` event triggered

16

The waiting queue can be an ordered list or a red-black tree.

Event triggered may be synchronous with execution of tasks (task suspension, timed sleep) or due to other interrupts

Requirements for the extension

- Compatible
  - Conservative with no effect on existing functionality
- Efficient
  - Low overhead of the scheduler and preemption points
- Maintainable
  - Easily integrated with future releases of RTAI

Compatible: want to preserve the existing functionality

Efficient: low overhead (both processor and memory)

Maintainable: changes to the RTAI code should be minimal

Extend Task Control Block (TCB)

- `preemptible` (Boolean)
  - true: task is preemptive
  - false: task is non-preemptive between preemption points
- `RT_FPDS_YIELDING` (Boolean)
  - flag in the task state
  - Indicates to the scheduler that the non-preemptive task is at a preemption point
- Invariant:
  - `RT_FPDS_YIELDING` ⇒ ¬ `preemptible`

Our first step is to extend the TCB with two boolean flags: ...

Additional primitive and scheduler modification

- `rt_fpds_yield()`:
  1. `RT_FPDS_YIELDING := true;`
  2. `rt_schedule();`
  3. `RT_FPDS_YIELDING := false;`
- `rt_schedule()`:
  1. Update current time
  2. Move tasks from waiting to ready queue
  3. **if (`preemptible` ∨ `RT_FPDS_YIELDING`)**
  4. Select highest priority task from ready queue
  5. Context switch to the selected task

19

We introduce an additional primitive, `rt_fpds_yield()`, which wraps the call to the scheduler between setting and clearing the ...

The added if statement in `rt_schedule()` the makes sure that **preempting task** will be switched in only if the **currently running task** at a preemption point, or if it is preemptive.

Both the preemptible and RT_FPDS_YIELDING pertain to the current task.

Optimize preemption point

- Avoid calling the scheduler when no new pending tasks
- Extend Task Control Block
  - `should_yield` (Boolean)
    - The should_yield flag is set to indicate that the task has to call `rt_fpds_yield()` upon next preemption point
- Add primitive
  - `fpds_pp()`:

```
1. if (rt_should_yield())
2.      rt_fpds_yield();
```

  - First step: `rt_should_yield()` is a system call

Our second step is to optimize the preemption point, and avoid calling the scheduler when there are no pending higher priority tasks which can preempt the current task.

`should_yield` is set by the scheduler and indicates that

Initially the `should_yield` was a flag in the TCB, requiring a system call to inspect it.

Optimize preemption point

- `rt_schedule()`:
  1. Update current time
  2. Move tasks from waiting to ready queue
  3. **if (higher priority task ready ∧ ¬ `preemptible`)**
  4. **`should_yield` := true**
  5. if (`preemptible` ∨ `RT_FPDS_YIELDING`)
  6. Select highest priority task from ready queue
  7. Context switch to the selected task

Now the scheduler also needs to set the `should_yield` flag when a higher priority task is ready AND the current task is not preemptible .

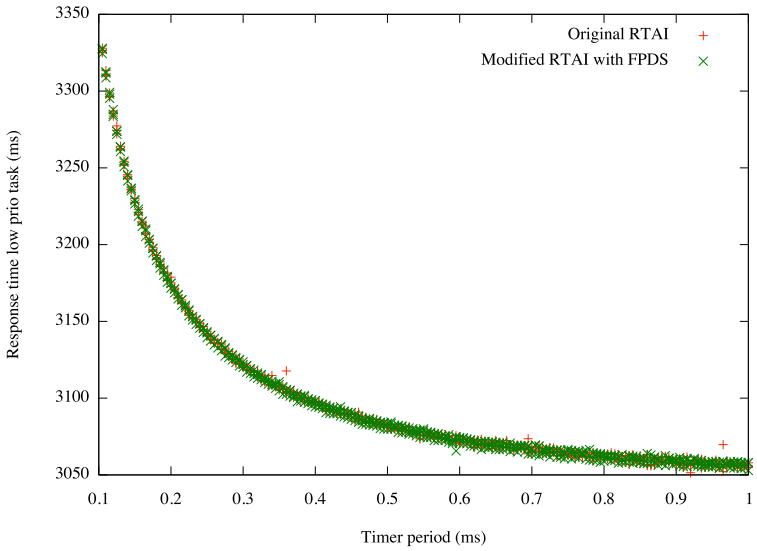
Experimental results

- Synthetic task set: two tasks
- Measured the response time of the low priority task under RTAI and RTAI+FPDS
- Tasks implemented in user space
  - Benefit from address space protection
  - Preemption point includes a system call overhead

22

The higher priority task was there to create work for the scheduler.

Scheduler overhead



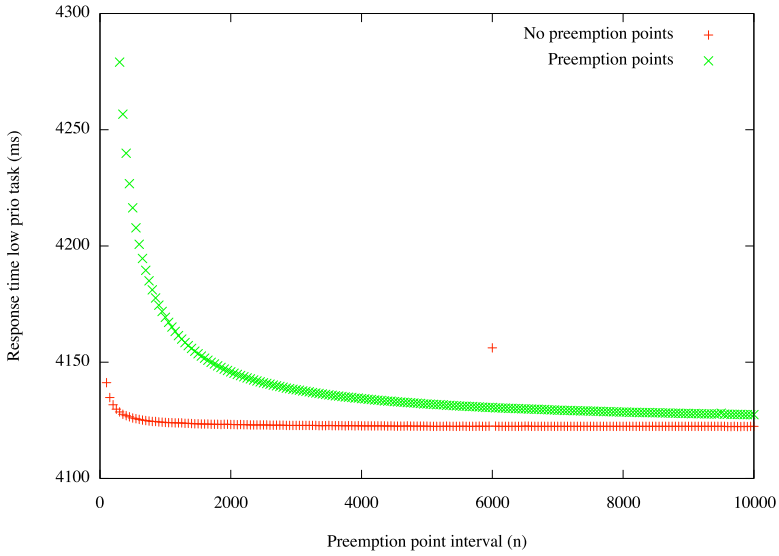
- No measurable scheduler overhead
  - Limited to a single if, with 3 variables and evaluating to false

We ran the same task set under stock RTAI (indicated by red crosses) and the modified RTAI (indicated by green crosses covering the red ones).

Two tasks: h and l
T_h = T_sched (so that a new task arrives at every scheduler invocation)

The overhead is lost in the noise of the measurements.

Preemption point overhead



- Preemption point overhead: 440us
  - Conform to the 434 us system call overhead in RTAI

For measuring the overhead of the preemption point, we compared the response time of the lower priority task with and without preemption points, running under the modified RTAI.

434 us system call suggests that the preemption point overhead can be reduced if a system call can be avoided.

```
Two tasks: l and h
l for FPDS:
for K/M {
        for i<M {
                counter := counter + 1;
        }
        rt_fpds_pp();
}

l for FPPS:
for K/M {
        for M {
                counter := counter + 1;
        }
}
```

Avoid the system call overhead

- Place the `should_yield` flag in the user space
- Similar to Litmus RT: soft real-time extension of the Linux kernel [Dr. James H. Anderson & Students]
- Measurements showed that the preemption point overhead was reduced to a few cycles

If the should_yield flag is stored in the kernel space, then checking the `should_yield` flag will require a system call. The system call can be avoided by storing the flag in the user space of the current task (similar to Litmus).

We have implemented the `should_yield` in user space.

Evaluation and conclusions

✓ Compatible
- Inserting preemption points and setting the task to non-preemptive (between the preemption points) are optional

✓ Efficient
- Low processor overhead of the scheduler and preemption points
- Low memory overhead (two additional integers in TCB)
- Note: blocking due to non-preemptive subtasks is intrinsic to FPDS (taken care of in the schedulability analysis)

✓ Maintainable
- 106 lines added/modified

26

Future Work

- Monitoring
  - Longest subtask will not interfere with higher priority tasks
  - Critical path through the task graph will not interfere with lower priority tasks
- Reservations

27

Questions?

Elaborate on the optional preemption points:

- Initially optional preemption points were meant to allow tasks adapt their control path.
In systems which incur large context switch overheads, such as the original platform of our industrial partner, where a preemption would invalidate the instruction pipeline and ongoing DMA transfers, one may want to refrain from starting a computation which will be aborted in case the task is preempted, and later restarted. The spare capacity can be used for other more useful computations.