Quality-of-Service and Resource management support in Task-Centric Models

Artur Podobas, Mats Brorsson, Vladimir Vlassov
{podobas,matsbror,vladv}@kth.se

KTH Information and
Communication Technology

Contributions

- Show that it is possible (with benefits) to achieve QoS in user-space with task-centric programming models
- Increase the resource awareness of task-centric runtime systems
- Empower the task-centric programming with timing constrained tasks
- Reduced power consumption

Outline

- What is QoS?
- Task-Centric scheduling and QoS-awareness
- Timing as a QoS constraint
- Current ideas and implementation
- Preliminary Results
- Conclusions

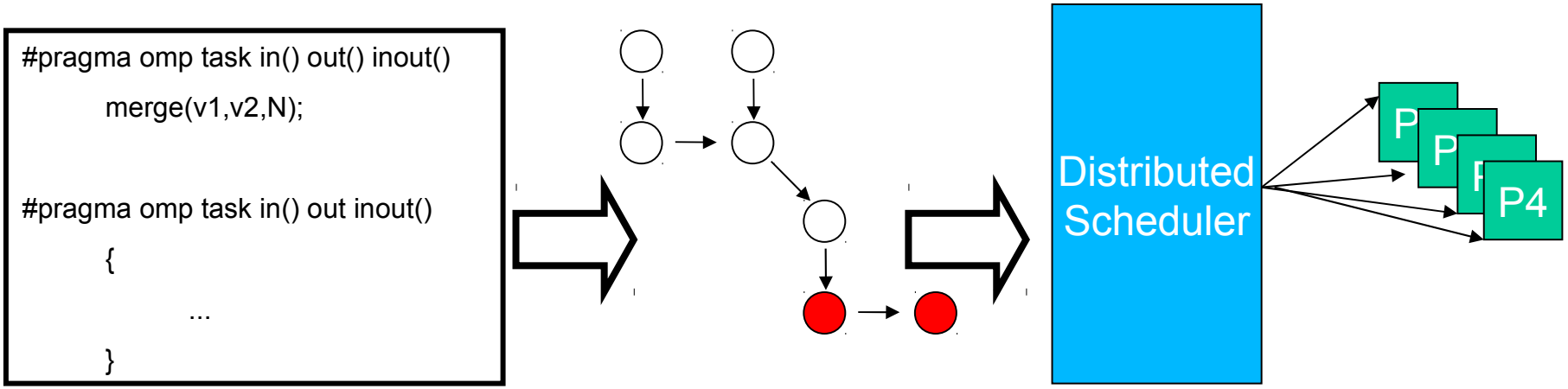
What is Quality-of-Service?

- Maximize the user's perceived experience
- QoS-needs exist in all system abstraction layers
  - Multimedia, Web browsers,...
  - Operating System,...
  - NoC interconnects,...
- Often combined with Resource-management
  - Enough resources to satisfy application QoS
  - …but also not too many to prevent degradation of other applications or to limit power consumption

Task-Centric scheduling and QoS-awareness

The task-centric paradigm:

- Exploiting dynamic parallelism within an application
  - Programmer exposes available parallelism encapsulated as *tasks*
  - A task can dynamically generate new tasks
- The task-centric scheduler distributes work across the acquired resources



Task-Centric scheduling and QoS-awareness

- There already exists tons of research concerning QoS and resource-management
  - Are they not adaptable to the task-centric paradigm?

- Not necessarily...
  - Existing solutions are within kernel-, hypervisor- or middleware-space
  - They do not assume multiple layers of scheduling (OS/user-level run-time for multiprogrammed workloads)

Task-Centric scheduling and QoS-awareness

- However, a task-centric runtime system contains a scheduler:
  - A distributed scheduler that assigns tasks to cores and which may also control resources (preferably in cooperation with the OS)
  - Middleware can get incorrect readings of an application's resource usage
  - The task-centric runtime system knows what tasks exist, will exist, and about history

Timing as a QoS constraint – Soft real-time systems

- We chose to use timing to specify QoS demands of the application:
  - Let the programmer specify the timing behavior of tasks
  - The timing should be specified so-that violating the constraint will result in a degraded experience for the user
- The timing-constraints will guide the scheduler in taking decisions
  - Tasks with tightest timing constraints execute first
  - Allow the scheduler to drop tasks predicted to violate their timing constraints
  - Predict what resources can be turned off to save power and when additional resources are needed

Timing as a QoS constraint

- Extensions to the existing OpenMP directive to support timing-behavior

#pragma omp task **deadline**(time) **release_after**(time)
ON_ERROR(OMP_SKIP | OMP_NO_SKIP)

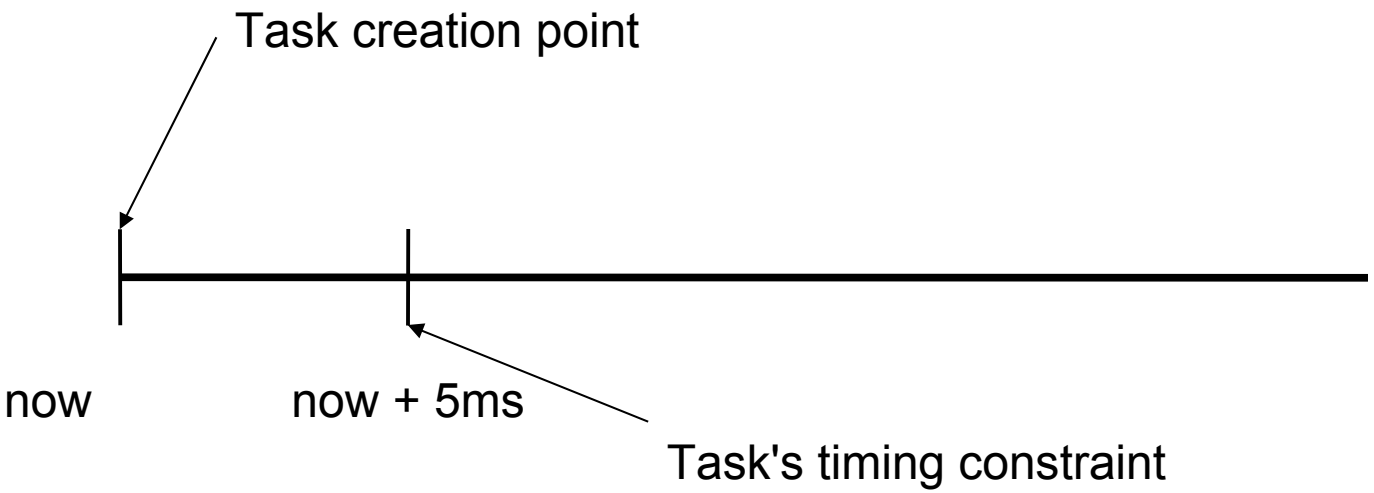
deadline() – Specifies the latest time a task should finish executing.

release_after() – Specified the earliest time a task can start executing.

ON_ERROR() - Specifies if this task may or may not be dropped

Timing as a QoS constraint

```
now =  omp_get_wtime();
#pragma omp task deadline(now + 5 ms)
                fft_array();
```



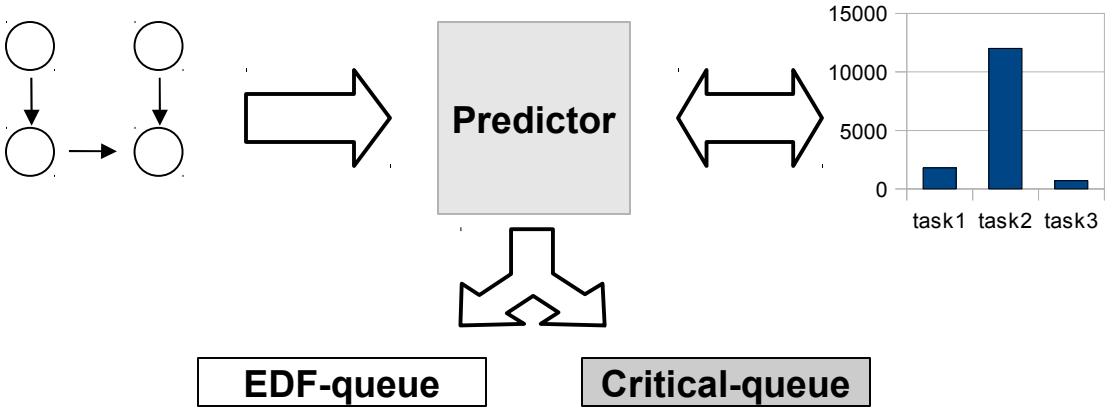
Current ideas and implementation

- The two global goals of the runtime scheduler are:
  - Strive to minimize the amount of tasks violating their timing constraints
  - Re-actively or pro-actively conserve resource according to the needs of the application (throughout execution) to reduce power consumption

Current ideas and implementation

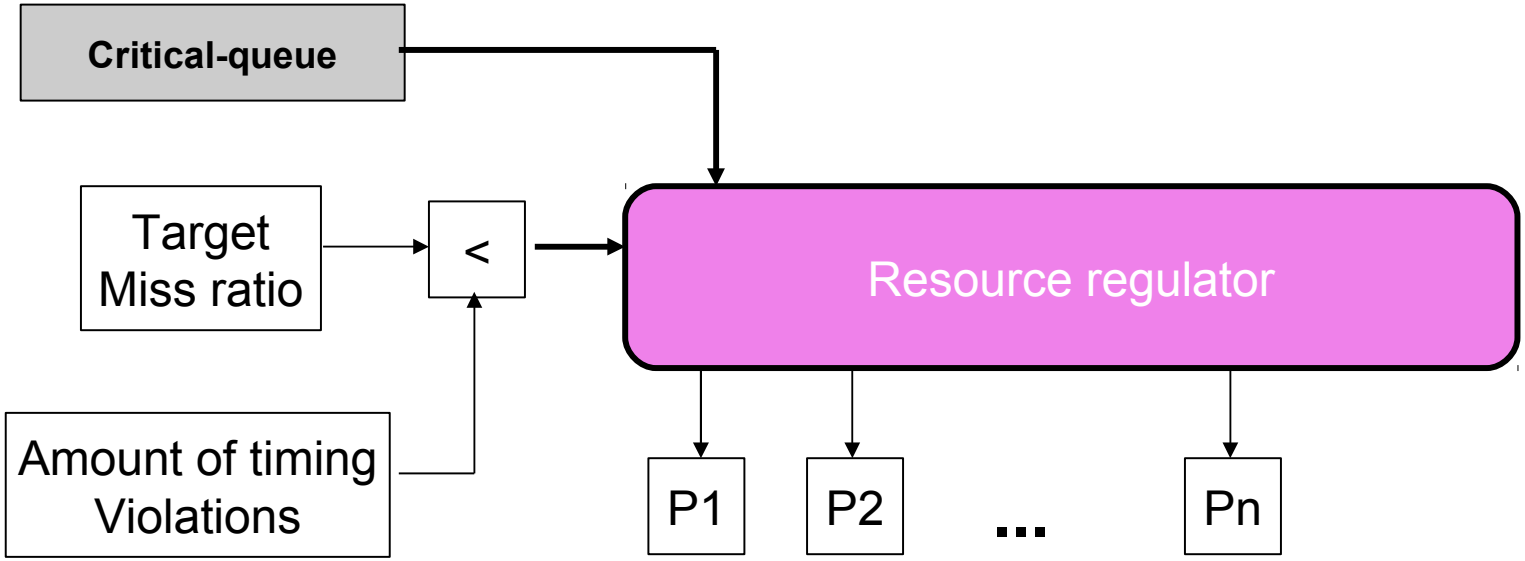
- Goal 1: ***"Strive to minimize the amount of tasks violating their timing constraints"***

- Solution:
  - Integrate an Earliest-Deadline-First, queuing policy to ensure that the scheduler always executes tasks with the earliest deadline
  - Integrate an critical queue that handle tasks that, according to their history and timing constraints, might miss their deadline



Current Ideas and Implementation

- Goals 2: ***"Re-actively or pro-actively conserve resources according to the needs of the application (throughout execution)"***
- Current Solution:
  - A "fuzzy-logic" approach monitoring the critical-queue and current timing violations to (de-) active resources

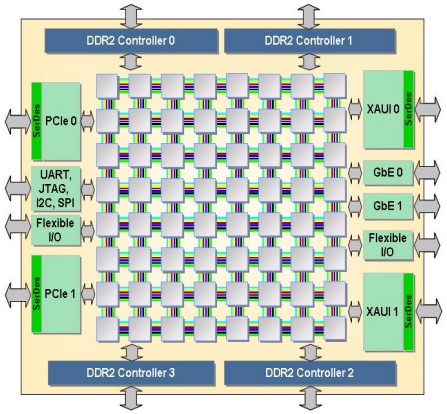


Current Ideas and Implementation

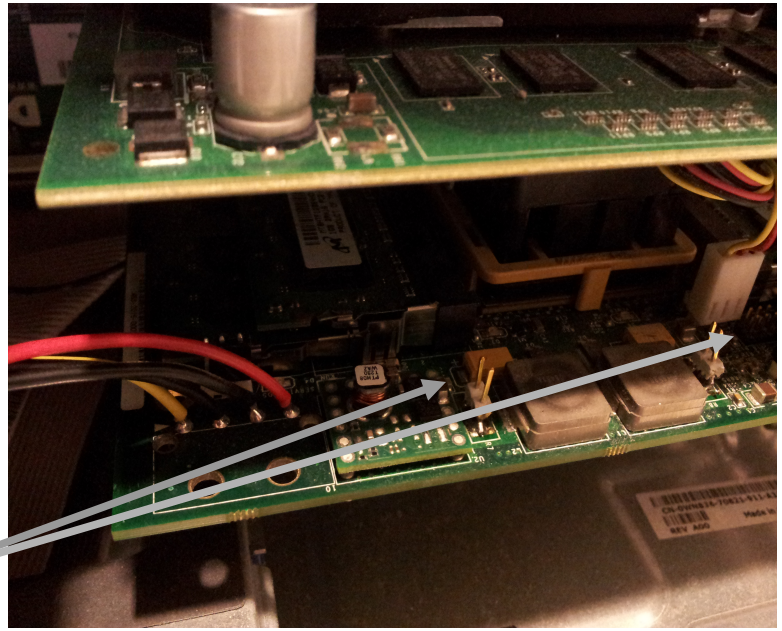
- We are using the Nanos++ runtime library (under the OmpSs programming model)
  - Plug-in based customization
  - Compiler assisted development using the Mercurium compiler
  - Existing debugging tool-chains: Paraver

Preliminary results

- We ported Nanos++ to the TilePRO64 processor
- The TilePRO64:
  - 64 small but energy efficient cores
  - VLIW
  - 700 MHz clock frequency



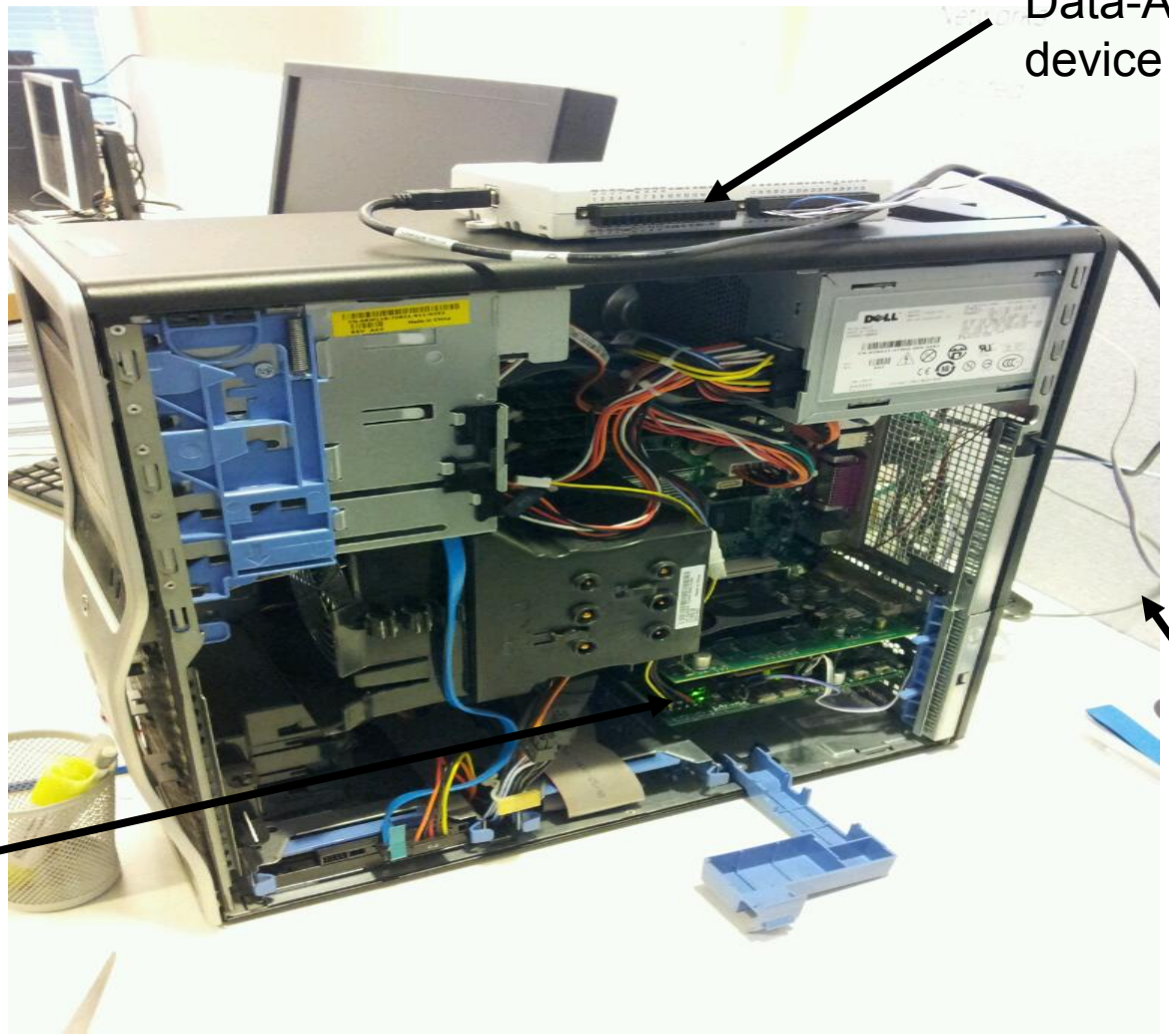
- We soldered and attached a National Instruments Data-acquisition (NI USB-6210) device to the TilePRO64's power pins



Soldered header-pins

encore

Preliminary results

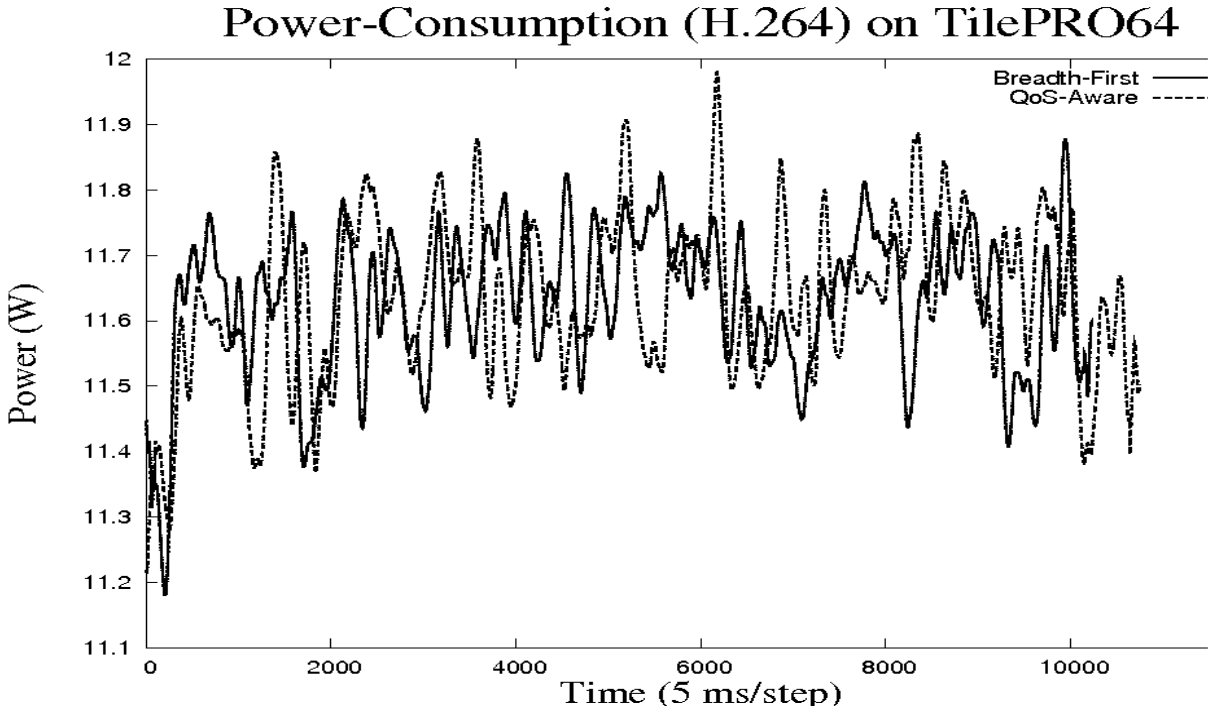


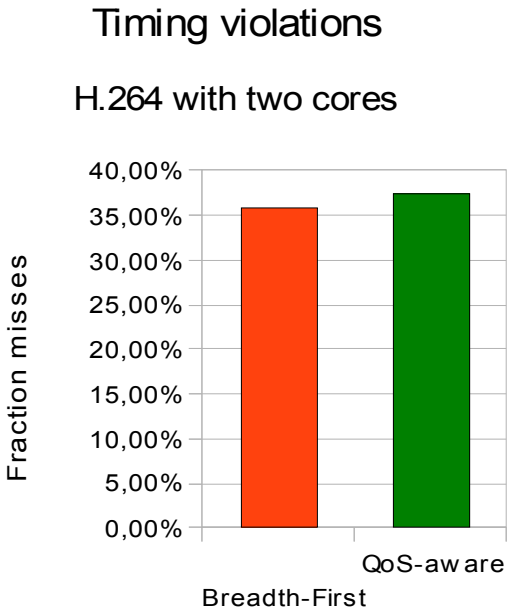
Preliminary results

- We executed the H.264 video decoder on the TilePRO64
- For the ***QoS-aware*** scheduler, we set a target of **<2%** timing violations
- We compare the results against a timing-unaware scheduler, the **Breadth-First scheduler**
- In these examples, only the **deadline()** clause was used
  - No task dropping
- Three scenarios:
  1. Not enough resources to meet timing constraints
  2. Enough resources to meeting timing constraints
  3. All resources available; letting the scheduler decide.

Preliminary results

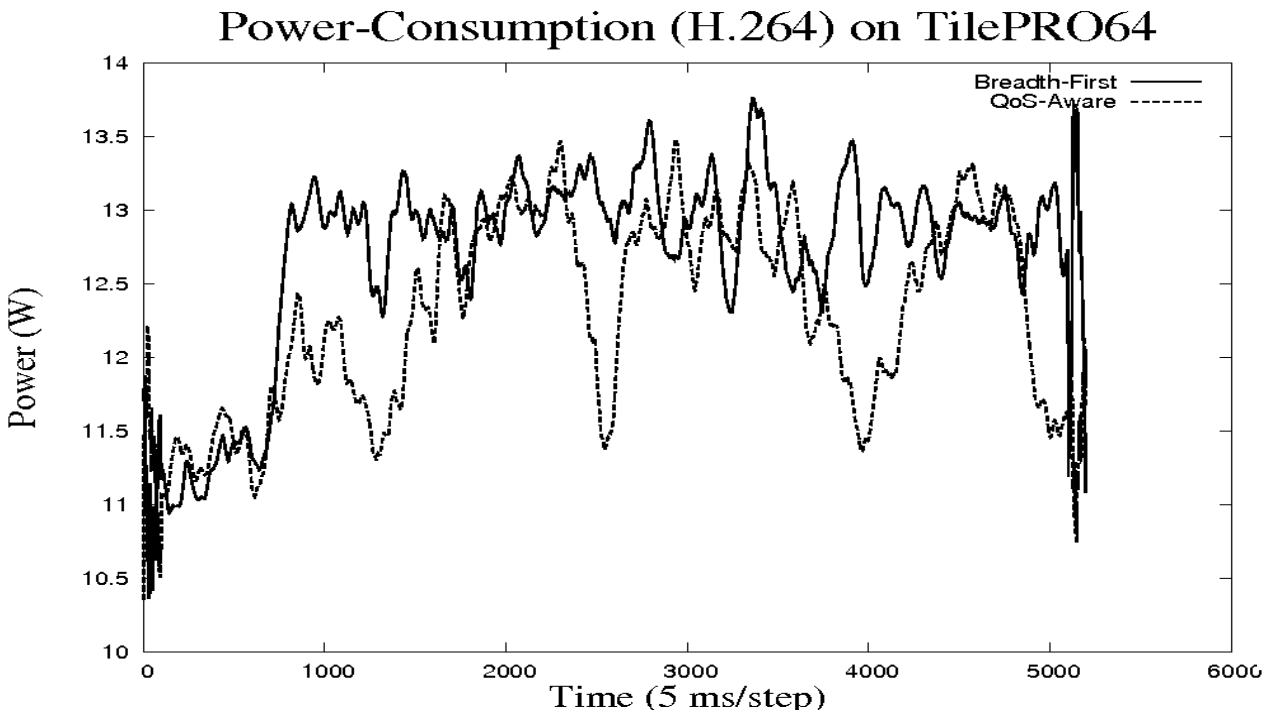
- H.264 running an HD movie with two cores.
  - Timing constraint set towards a 10 fps execution
- Overall power consumption ***increase***: 0.7%
  - Investigation need, but likely due to complexity of scheduler

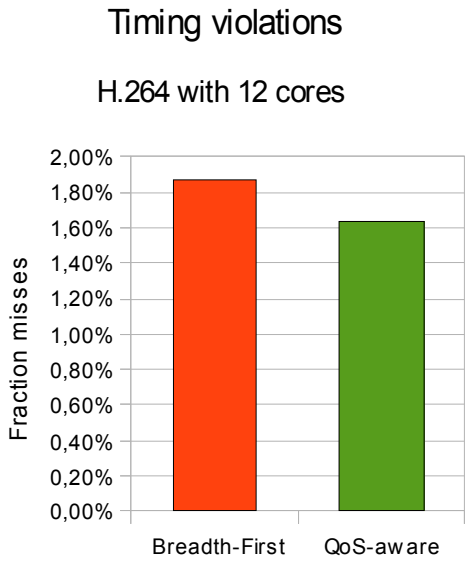




Preliminary results

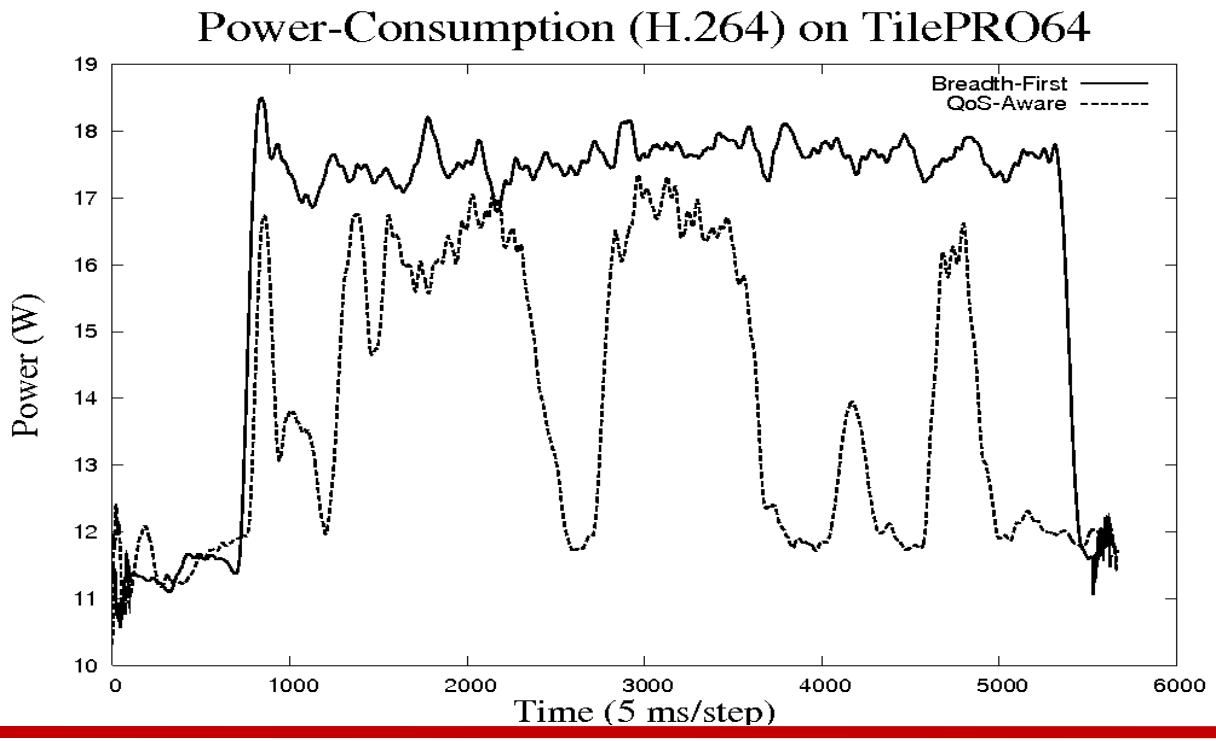
- H.264 running a HD movie with 12 cores
  - Timing constraints set towards a 10 fps execution
- Overall power consumption ***decrease***: ~5%

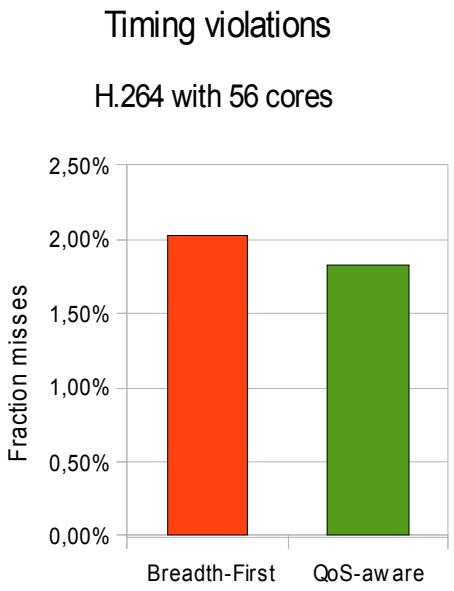




Preliminary results

- H.264 decoder running a movie with 56 cores
  - Timing constraints set towards a 10 fps execution
- Overall power consumption ***decrease***: ~17%





Conclusions

- In majority of cases, power consumption is decreased compared to a timing un-aware scheduler
- A scheduler that guarantees that tasks with earliest deadline are executed first
- User-friendliness and portability increase; let the runtime system decide about resources

Future work

- Future work include
  - Refining the resource controlling model.
  - Further decrease the overhead of our scheduling policy
  - Evaluate on more benchmarks

Acknowledgments

- Thanks to C.C. Chi and prof B. Juurlink of TU Berlin for the OmpSs version of H.264
- Thanks to M. Själander and S. McKee's team of Chalmers for the support concerning the power measurements

Thank you