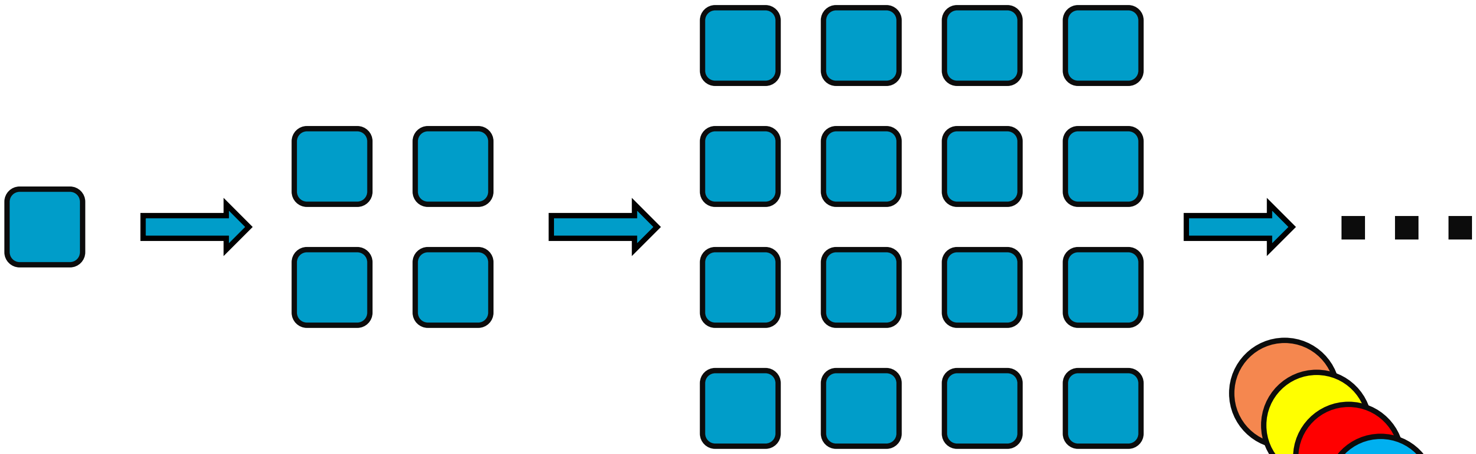
CISTER - Research Center in
Real-Time & Embedded Computing Systems

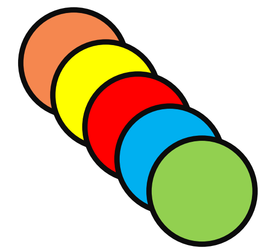
Semi-Partitioned Scheduling of Fork-Join Tasks using Work-Stealing

Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho
EUC 2015

Context

- Evolution from uni to multi/manycores

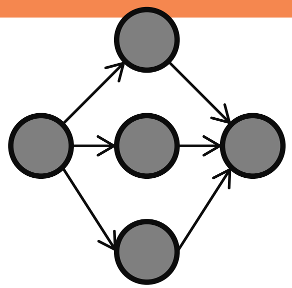




Set of tasks

- Scheduling in multiprocessors
  - When and where
- Scheduling Approaches
  - Global, partitioned, semi-partitioned

Context



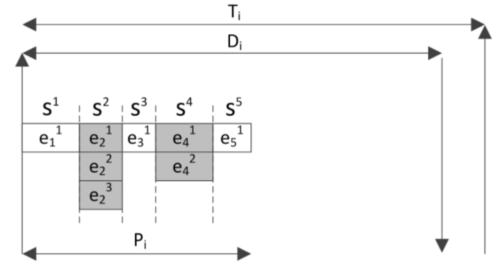
- What about parallel tasks?
- Parallel frameworks used to exploit parallelism
  - Implicit parallelism
  - Explicit parallelism
  - Many use work-stealing
- Work-stealing
  - Reduces task contention
  - Load balances the workloads
  - Preserves data locality
  - Not ready for real-time systems

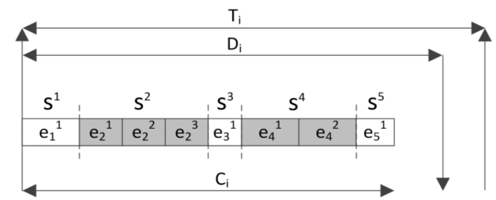
Contributions

- Scheduling Fork/Join tasks using semi-partitioned scheduling
- Work-stealing may reduce average response-time
  - Execute other tasks or save energy consumption
- Controlled stealing allows the policy to be used in RT systems

System Model

- Fork/join tasks
- Constrained-deadline model



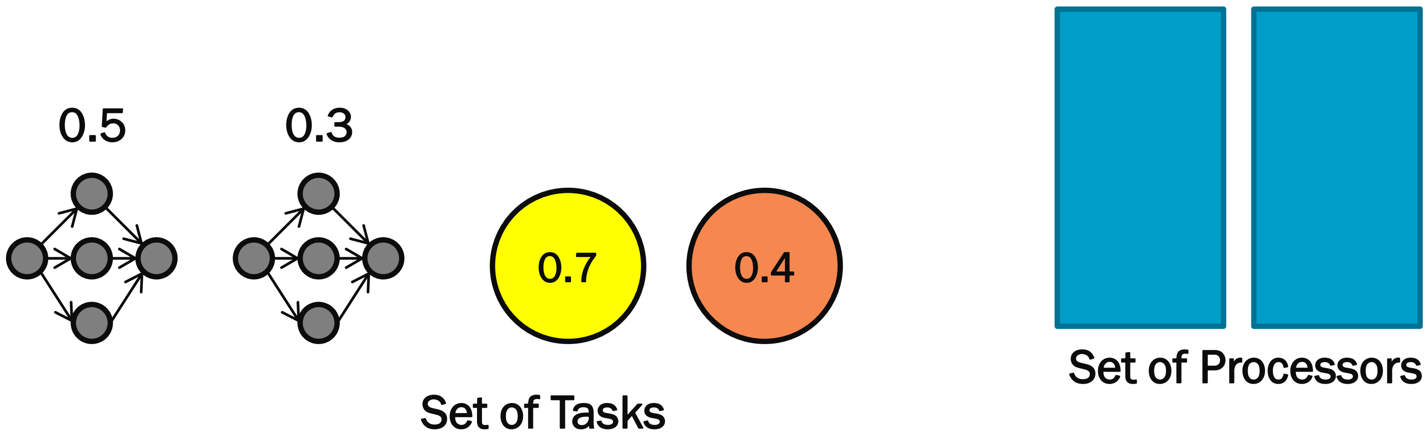


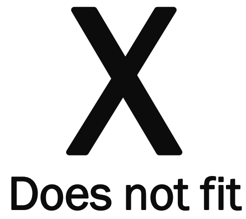
- Homogeneous processors
- Fully preemptive EDF scheduler on each core
- Assumptions
  - Task density **is not** greater than 1
  - Decomposition approaches can be used for conversion
    - Task structure must be preserved

Proposed approach

- Phase 1 - Task assignment
  - Select migrating and non-migrating tasks
    - Task density
    - Demand of each core after task assignment
  - Sequential tasks are evaluated first
    - Increasing the probability of having parallel tasks as migrating tasks
  - First-Fit Decreasing (FFD) to partition tasks into cores

Phase 1 – Task Assignment





- Output
  - Set of non-migrating tasks
  - Set of candidate migrating tasks

Phase 2 – Offline Scheduling

- Determine the execution pattern of each migrating task
- Each migrating task is treated as a multiframe task
    - i.e. $_{11}$= ((3, 0,0,0),5,6), $_{12}$= ((0, 3,3,3),5,6),

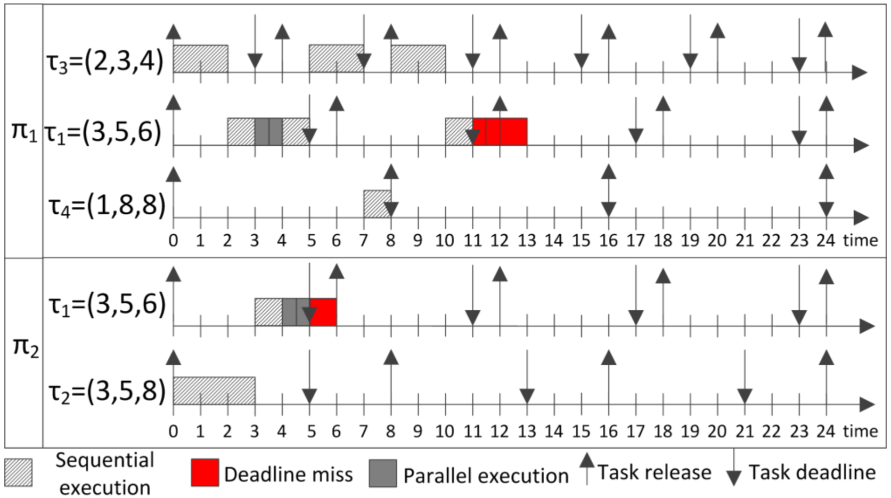
- For each core we check the largest number of jobs that can be executed without violating schedulability
    - Starts at $k_i = H/T_i$ jobs and it decrements a unit at a time
    - For each k_i jobs we check the valid execution patterns for that core
    - Stops when an execution pattern is found with k_i jobs or no pattern exists

Phase 3 – Online Scheduling

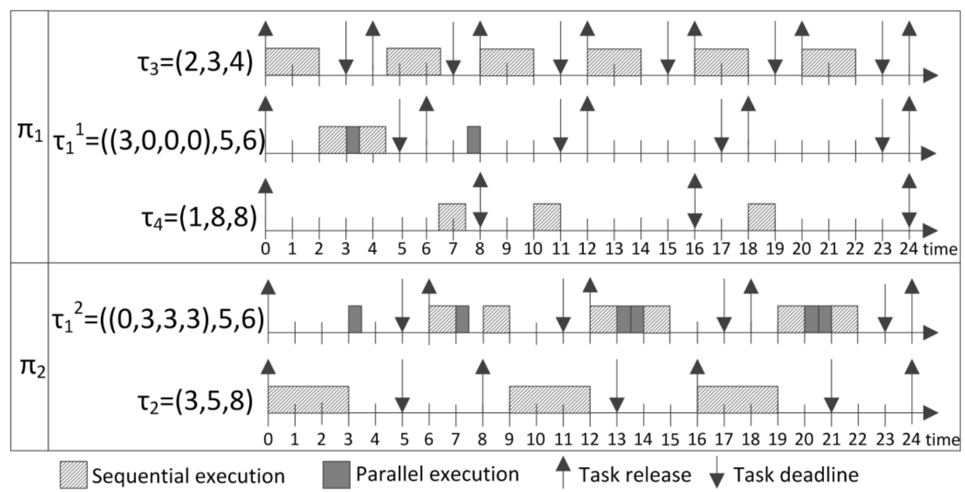
- Apply work-stealing among cores that share a copy of the task
  - Reduce the average response-time of the tasks in the system
  - Controlled number of migrations due to the task to core mapping

- Rules for stealing work:
  - A core must be idle in order to steal
  - Workload is stolen from the deque of another core
  - Highest priority sub-task must be chosen (#MT > 1)
  - Admission control is performed before stealing

Example

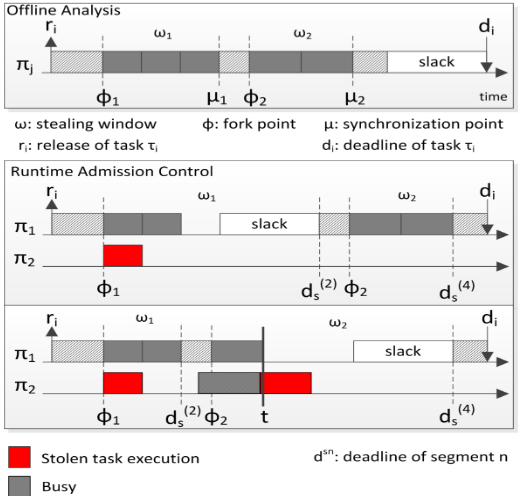


- $\lambda_1 = 0.6$
- $\lambda_2 = 0.6$
- $\lambda_3 = 0.66$
- $\lambda_4 = 0.125$



Schedulability Analysis

- Offline phases
  - Based on demand bound function (DBF)
  - Both types of tasks are considered
    - Non-migrating: standard DBF
    - Migrating: modified DBF that considers the execution patterns
- Online Phase
  - Admission control
    - Slack and stealing windows

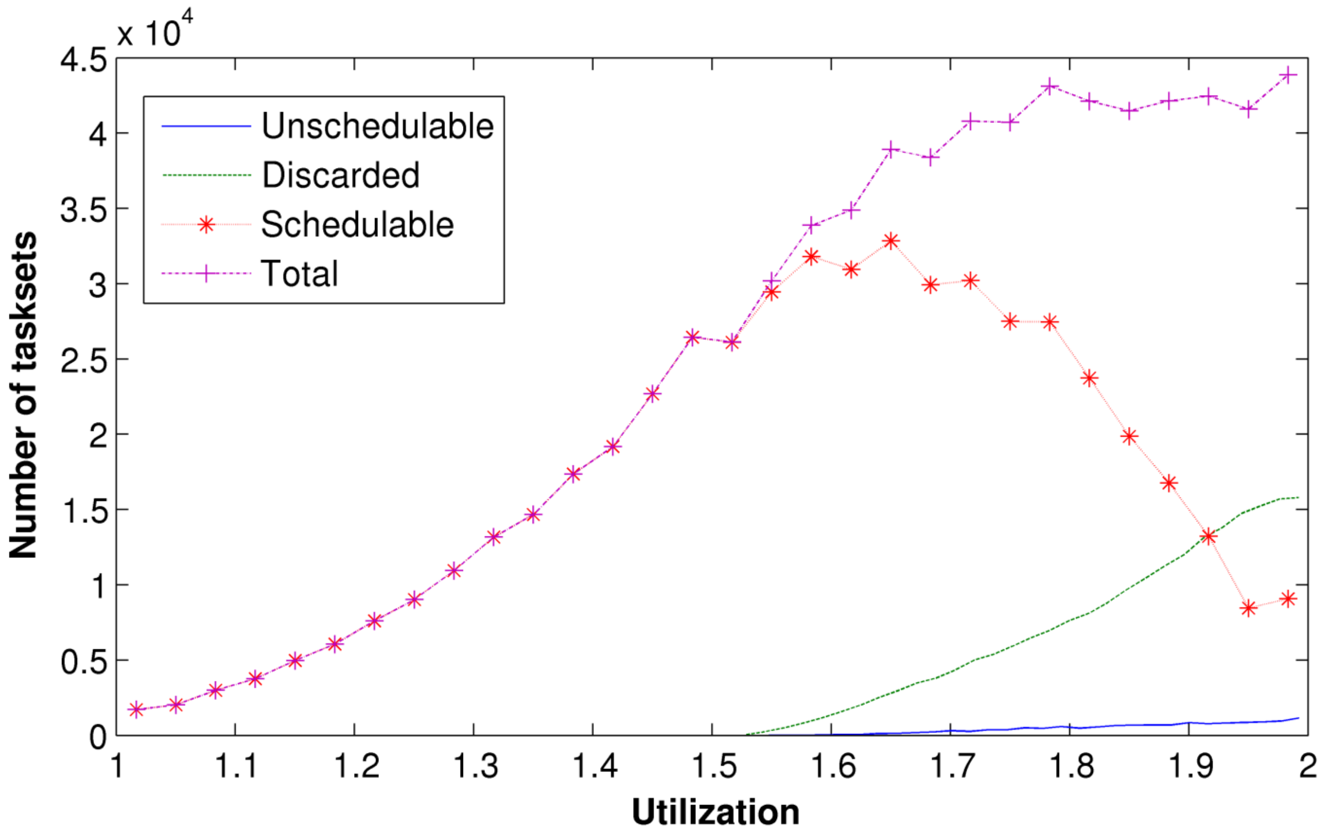


Results

- Random task generation
  - Tasks can be sequential or parallel
  - Number of segments k is chosen from (1,3,5,7)
  - Number of sub-tasks varies in the interval [k,10]
  - Each sub-task has a max_Ci_subtsk = 2
  - Period is generated in the interval:
    - [C_i, n_{subtsk} * max_Ci_subtsk * 2]
  - 1000 task sets are generated for 2 and 4 cores

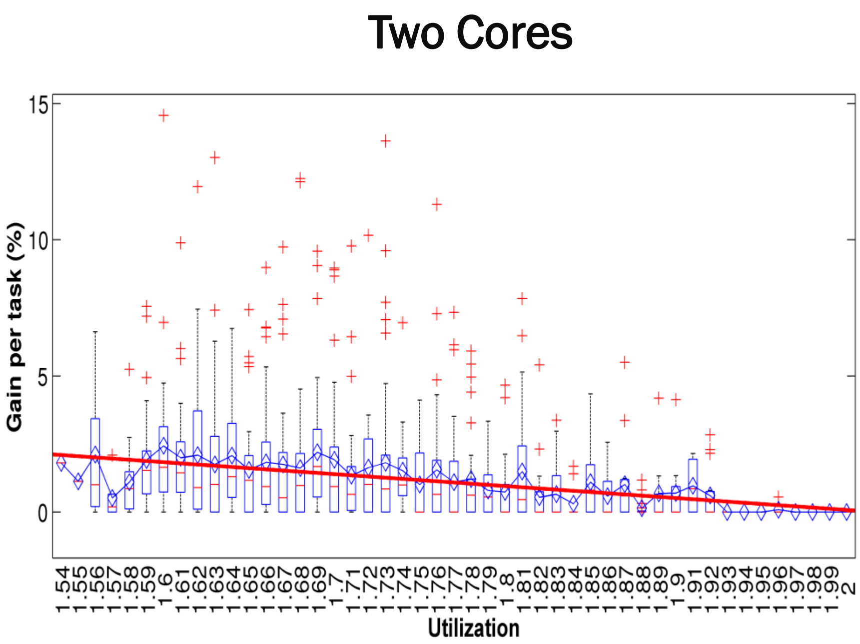
- We measure the gain obtained for each task set in terms of average worst-case response time
  - Using a WS approach versus a non-WS approach

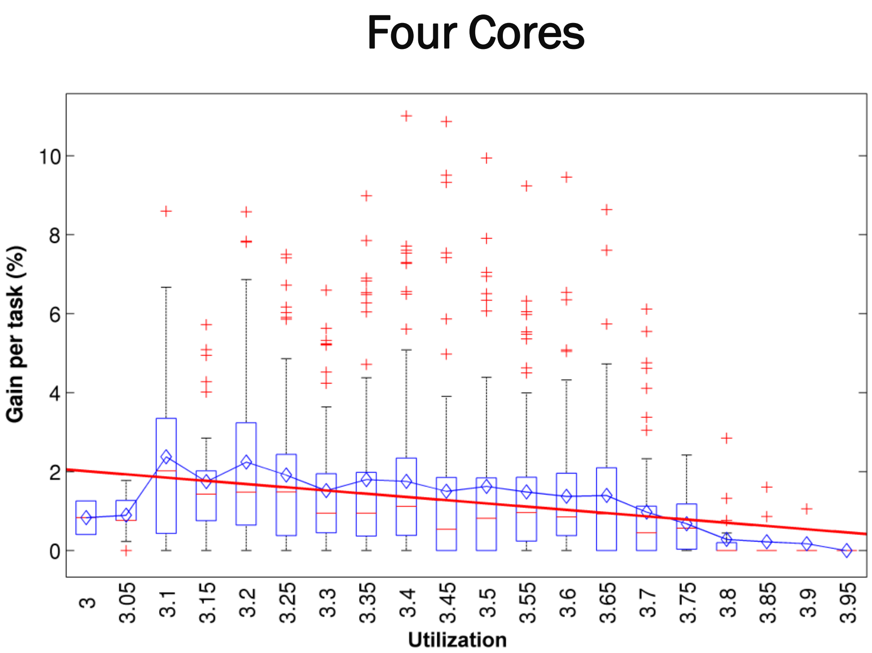
Results - Generation profile



Results

- Improvement in terms of average worst-case response time per task





Overheads

- Cores that share a task have a local copy of the task
  - Platform dependent due to memory constraints
  - Local copies prevent having to fetch code + data

- Stealing may cause interference on the shared bus
- Stealing costs are supported by the idle core
- The number of data transfers can be bounded
  - Worst-case depends on the number of sub-tasks and the number of cores that share a task

- Online admission test
  - Time instant and available slack

Conclusion

- Framework for scheduling parallel tasks on multicore platforms
- Combining semi-partitioning and work-stealing
  – Decrease the average worst-case RT of tasks
  – Bound the number of migrations
- Future work
  – Scalability of the approach
  – Different allocations heuristics
  – Better mechanism for pattern detection

Questions