Semi Partitioned Hard Real-Time Scheduling Under Locked Cache Migration in Multi-Cores

By

Mayank Shekhar[1]

Harini Ramaprasad[1]

Frank Mueller[2]

Abhik Sarkar[2]

1: Southern Illinois University Carbondale

2: North Carolina State University

Introduction

- Multi-core architectures
  - Provide more computational power
  - Have increased power/energy efficiency
  - Increasingly used in Real-Time/Embedded Systems
- Hurdles in using multi-cores in real-time systems
  - Unpredictability in execution increases
  - Inefficient Scheduling leads to 'hot spots'

Background

Categories of multi-core scheduling algorithms are

- Partitioned -Tasks statically assigned to cores
  - Advantages:
    - No migration overhead
    - Low on-line overhead
    - Uni-processor algorithms can be reused
  - Disadvantages:
    - Optimal task allocation is an NP-hard problem
    - Could lead to poor load balancing
    - Unable to use free processing time on idle cores

Background(2)

- Global – Dynamic Task allocation
  - Advantages :
    - Increased Utilization
    - Improved load balancing
  - Disadvantages :
    - Higher scheduling overhead
    - Migration overhead (Ex- Cache to cache migration)
    - Guaranteeing Predictability is challenging

Background(3)

- Semi-Partitioning :
  - Most tasks statically partitioned onto cores
  - Few tasks migrate among group of cores
  - Advantages :
    - Less scheduling overhead compared to global
    - Less migration overhead compared to global
    - Improved load balancing compared to Partitioning
    - Increased utilization compared to Partitioning

Related Work

- Andersson et al., Kato et al. and Dorin et al. have proposed semi-partitioning algorithms in past
  - These algorithms aim at reducing migration overhead
  - Constants added to WCET for migration overhead
  - Don’t consider cache content migration.
- Sarkar et al. proposed
  - proactive, push-based migration mechanisms for bus-based multi-core architectures
  - mechanisms for locked cache migration

Current work

- Cache based migration not trivial in multi-cores
- In our current paper we
  - Explicitly consider cache related factors
  - Reduce on-line overhead by offline decision making
  - Use push-based mechanism for cache-content migration

Assumptions-Architectural Model

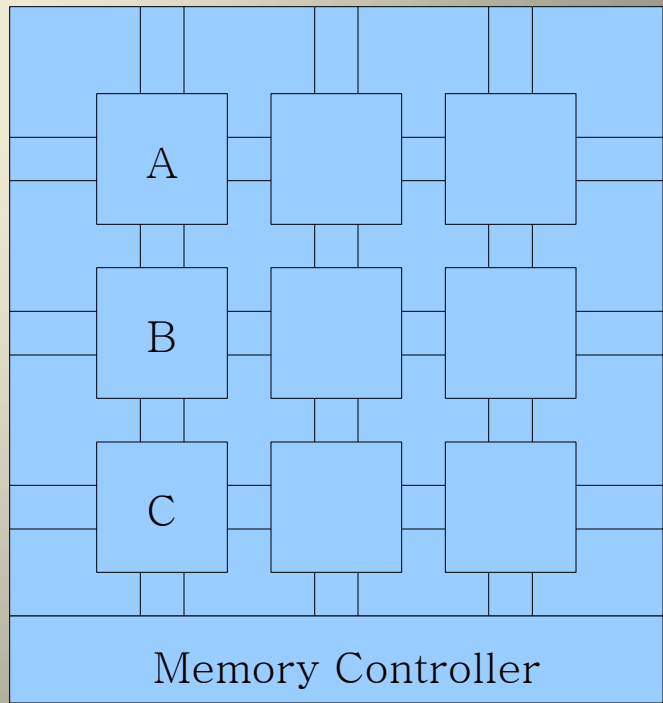
- Homogeneous multi-cores
- Private lockable K-way set associative caches on each core
- A 2-D mesh-based Network On Chip(NoC)
  - Dedicated bi-directional channel for cache to cache transfers
  - No interference with channels for main memory traffic.
  - Ex- TilePro64-64 core architecture

Assumptions-Task Model

- Periodic hard real-time tasks
- Relative deadlines <= Periods
- Independent tasks
- Can lock cache footprints
- Unlocked memory lines bypass cache
- K-1 ways lockable by partitioned tasks
- No tasks wrap around
- One migrating task per core
- WCET independent of core location

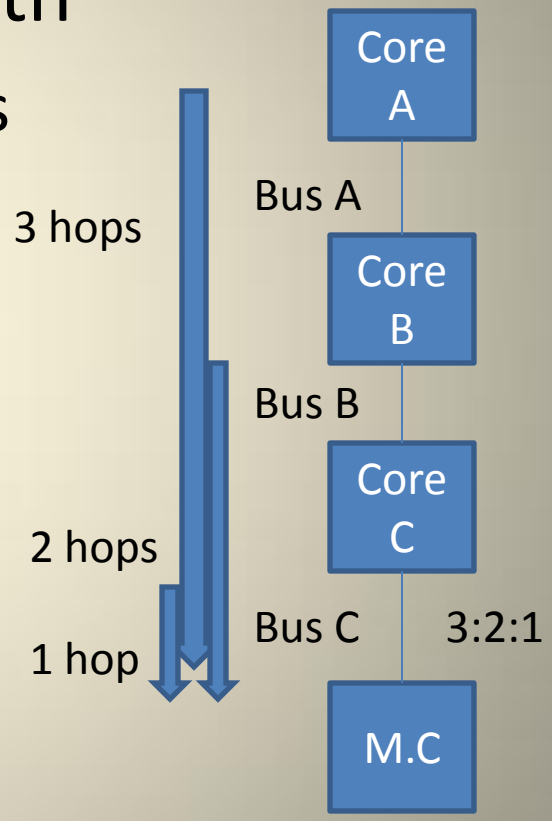
Weighted TDM approach

- Memory requests are
  - Statically routed along straight path to the memory controller
  - Arbitrated using TDM approach

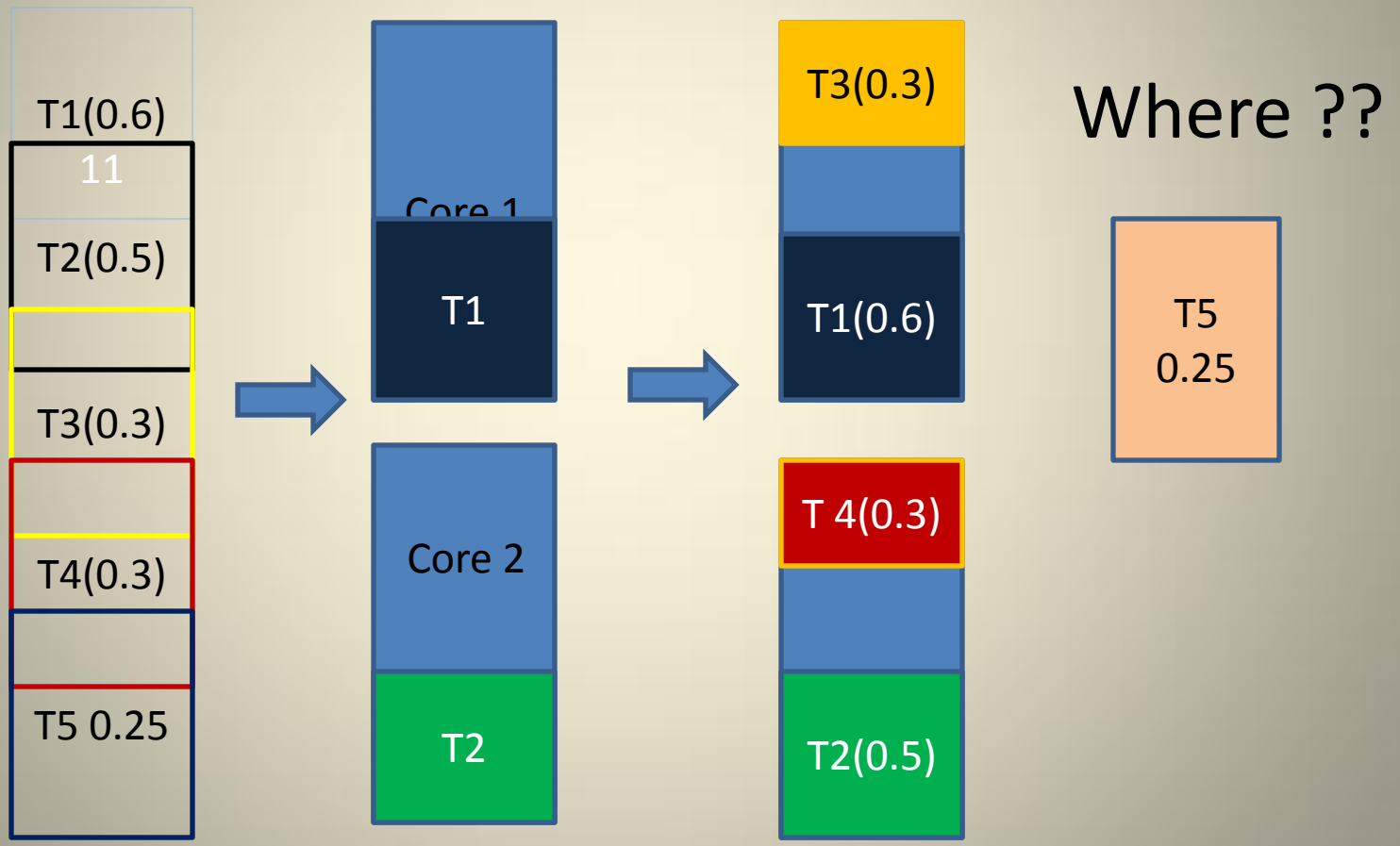


Weighted TDMA

- Main memory traffic bandwidth
  - Proportional to number of hops
  - Noc latency across bus C for core A,B and C is 2, 3 and 6
- Total NoC latency for
  - Traffic from core A is 5 cycles
  - Traffic from core B is 6 cycles
  - Traffic from core C is 6 cycles

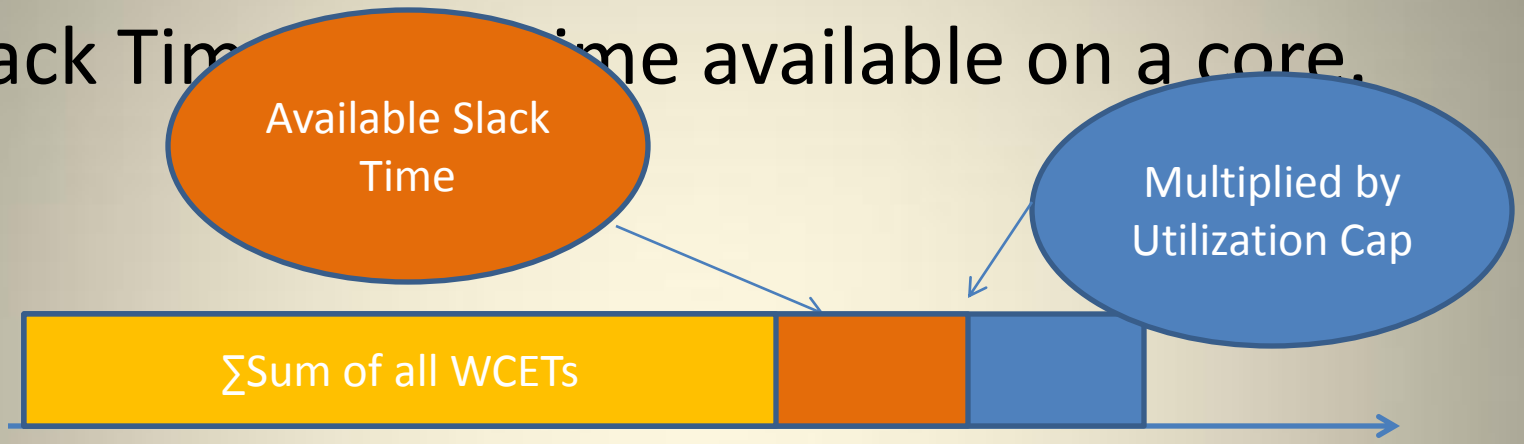


Algorithm(Partitioning)



Theory Involved

- Slack Tim ime available on a core.



- Theorem : A migrating task T gets highest priority for time δ within any time interval equal to the shortest relative deadline

Theorem Implication

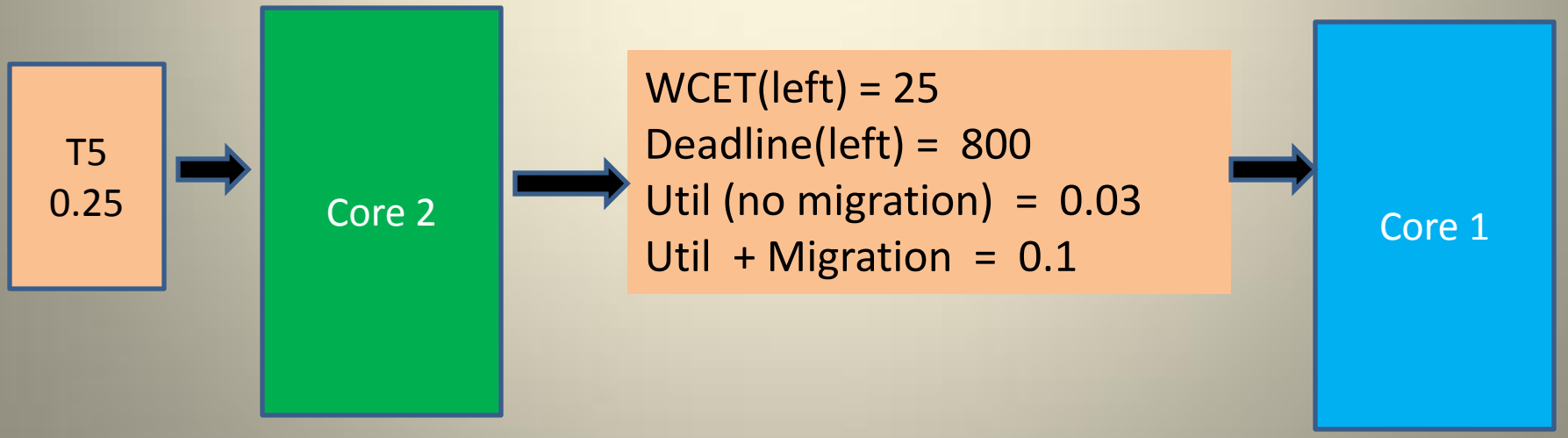
- T gets highest priority upon arrival to a core
- Its remaining utilization decreases with each migration
- On last core, it executes as normal task
- Schedulability of non-migrating tasks on those cores not affected

Algorithm(Migration)

Tasks	T1	T2	T3	T4	T5
Periods	1000	1000	1000	1000	1000
WCET	600	500	300	300	225

Core 1
Slack Time = 100

Core 2
Slack Time = 200



Algorithm

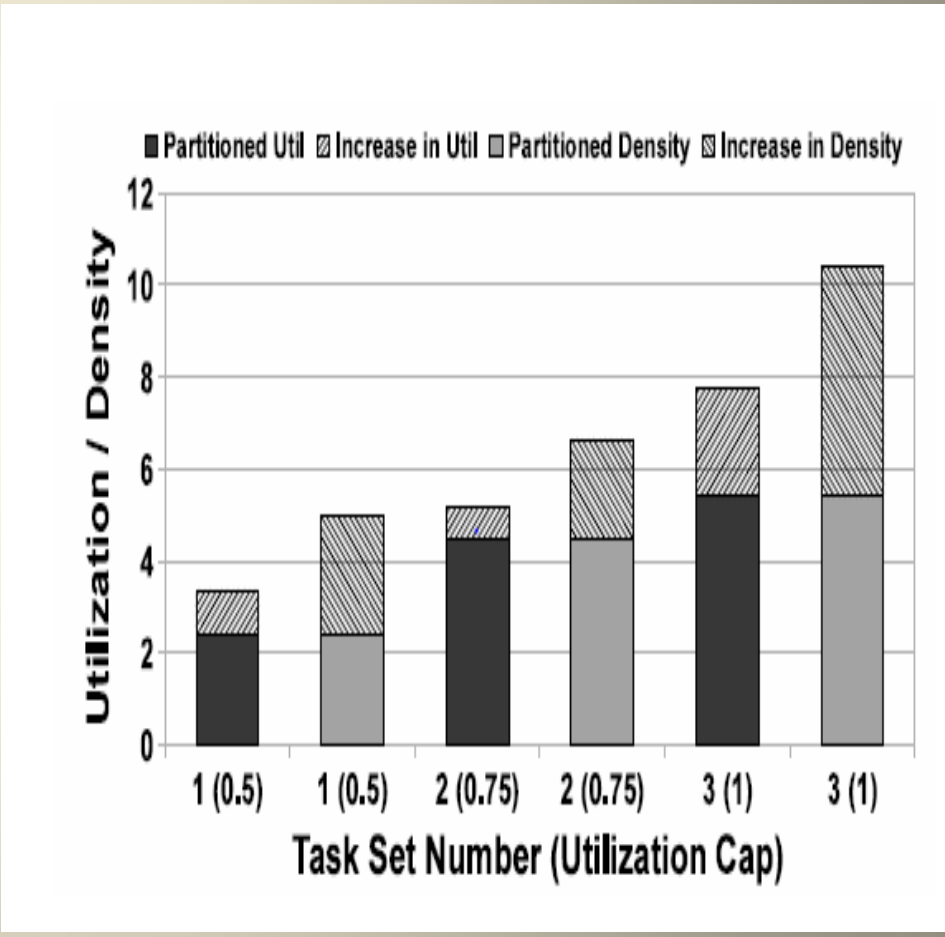
- Migration overhead consists of
  - Read latency at source core
  - Write latency at target core
  - Transfer time of cache lines from source to target core
  - Time to return to the first core
- Assign migrating task to cores such that migration overhead is minimized

Experimental Setup

Processor	In-order
Cache Line Size	32 Bytes
L1 D-Cache Size/Associativity	256KB/4-way
L1 hit latency	1 cycle
Number of Cores	9
External Memory Latency	72 cycles

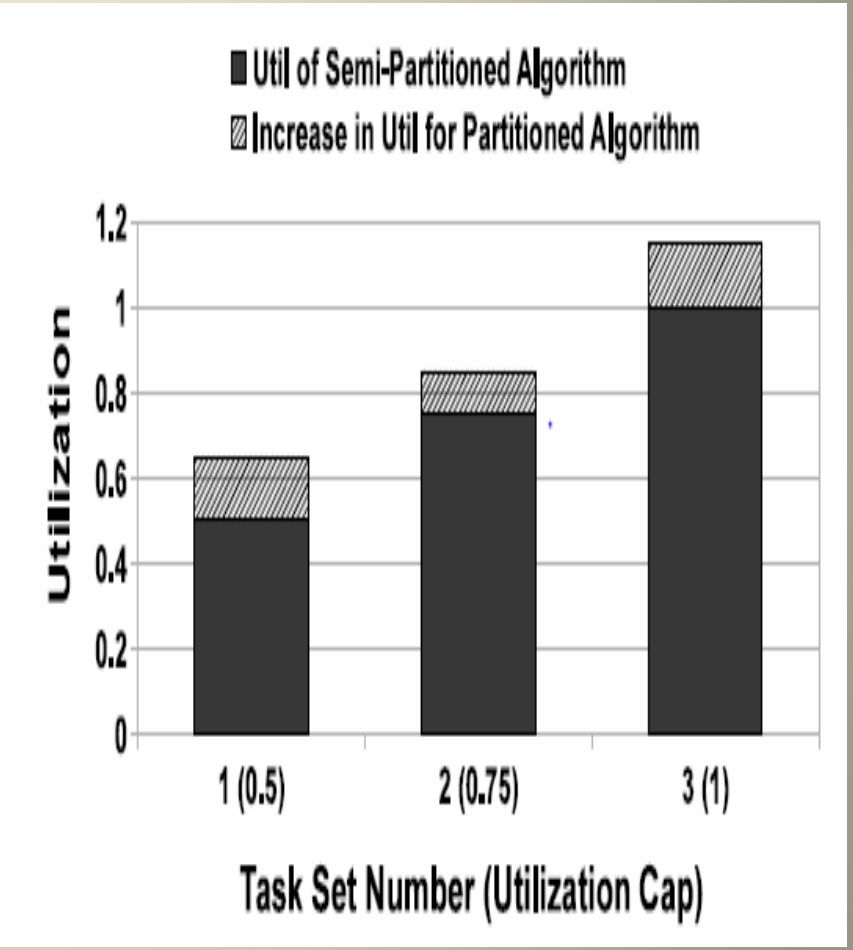
Experimental Results

- Used DSPStone Benchmarks
- For Utilization caps of 0.5,0.75 and 1
  - Comparing with purely partitioned approach.
- We show the increase in utilization and density



Experimental Results

- Compares utilization cap needed by Partitioning and Semi-Partitioning to schedule same task set
- Lower Utilization cap can lead to save power eventually



Conclusion

- Compared to purely partitioned approach we achieve
  - an average increase in utilization of 36.75%
  - an average increase in density of 78.32%

THANK YOU