Opportunistic Competition Overhead Reduction for Expediting Critical Section in NoC based CMPs

Yuan Yao and Zhonghai Lu

KTH Royal Institute of Technology

Stockholm, Sweden

ISCA 2016, Seoul, Korea, 2016-06-20.

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction
- Problem
- Design
- Experiments
- Summary

ROYAL INSTITUTE OF TECHNOLOGY

Introduction

- For multi-threaded shared variable applications, **entering and executing** critical section that contains shared data need to be synchronized and must be **mutually exclusive**, meaning that only one thread can enter and run a critical section at a time.

- As previous studies [1, 2, 3, *etc.*] show, the time spent in executing critical sections by different threads is usually the most significant source of serialization in parallel applications.

1, M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, *"Accelerating Critical Section Execution with Asymmetric Multi-core Architectures,"* in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.

2, J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, *"Bottleneck Identification and Scheduling in Multithreaded Applications,"* in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

3, E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, *"Parallel Application Memory Scheduling,"* in International Symposium on Microarchitecture (MICRO), 2011.

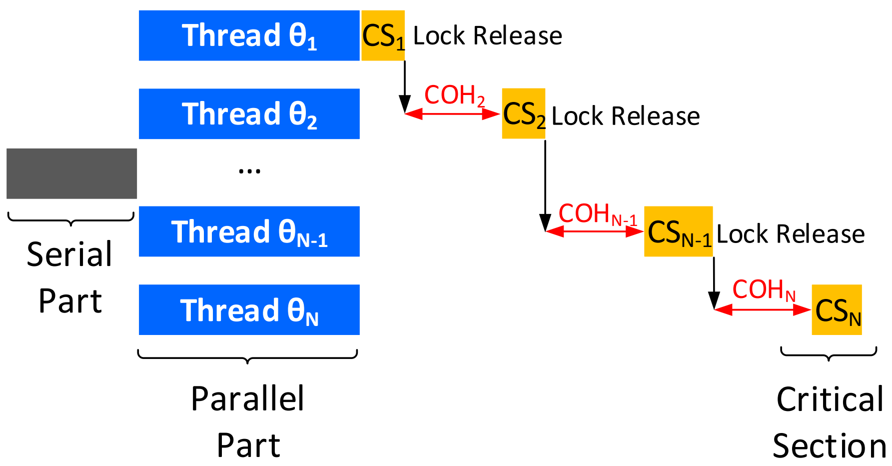
ROYAL INSTITUTE OF TECHNOLOGY

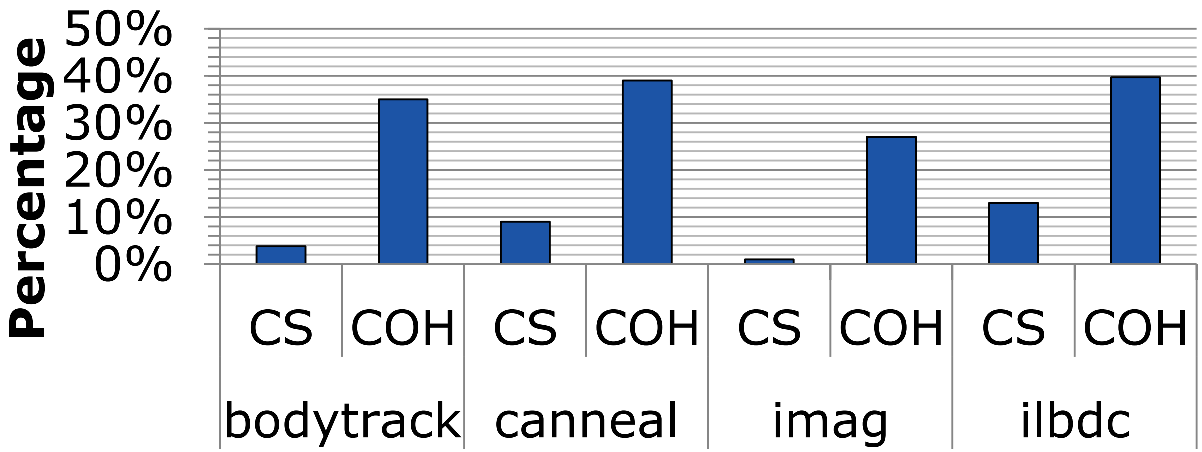
Problem

- However, performance of multi-threaded shared variable applications is **not only** limited by serialized critical section execution, **but also** by **competition overhead** (COH) which threads experience in order to **enter** critical sections.

- As the number of concurrent threads grows, such competition overhead may <u>exceed</u> the time spent in executing critical section, and become the dominating factor limiting the performance of parallel applications.

Problem – Cont.





Percentage of CS execution and COH in ROI finish time

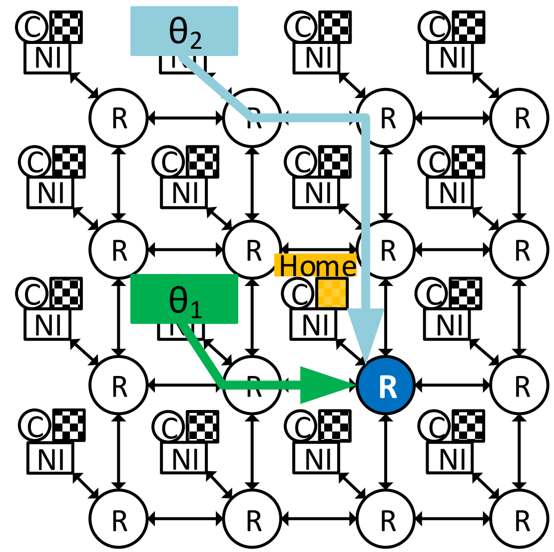
- Consisting of only a few lines of code, critical section itself usually takes very limited time to execute.
- However, threads may spend more time competing with each other to enter into critical sections.

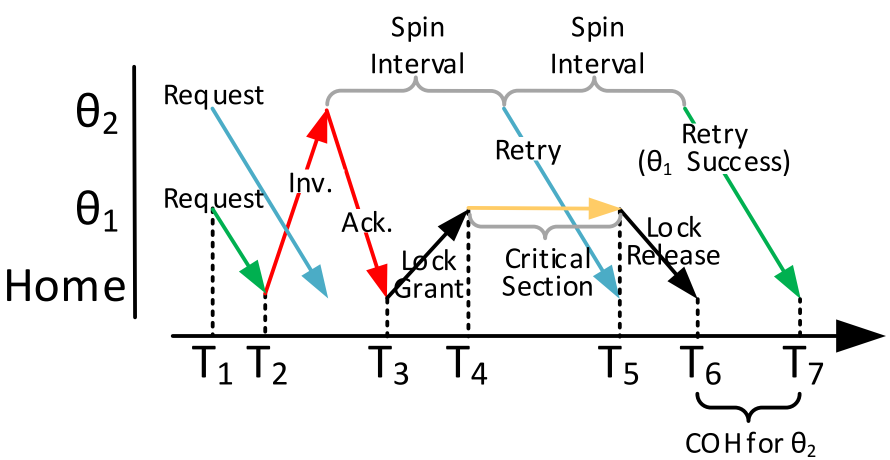
Queue spin-lock in state-of-the-art OS

- In OS, locking primitives are provided to support critical section synchronization for multi-threaded programs.
- Different threads compete with each other to lock the critical section through the locking functions.
- Most state-of-the-art OSes such as Linux 4.2 and Unix BSD 4.4 adopt **queue spinlock** primitive, which comprises a low-overhead **spinning phase** and a high-overhead **sleep phase**.
  - In the low-overhead spinning phase, a thread spins for locking critical section access.
  - If the critical section cannot be obtained after a certain times of spins, the thread registers its request to a lock queue and enters the high-overhead sleep phase.

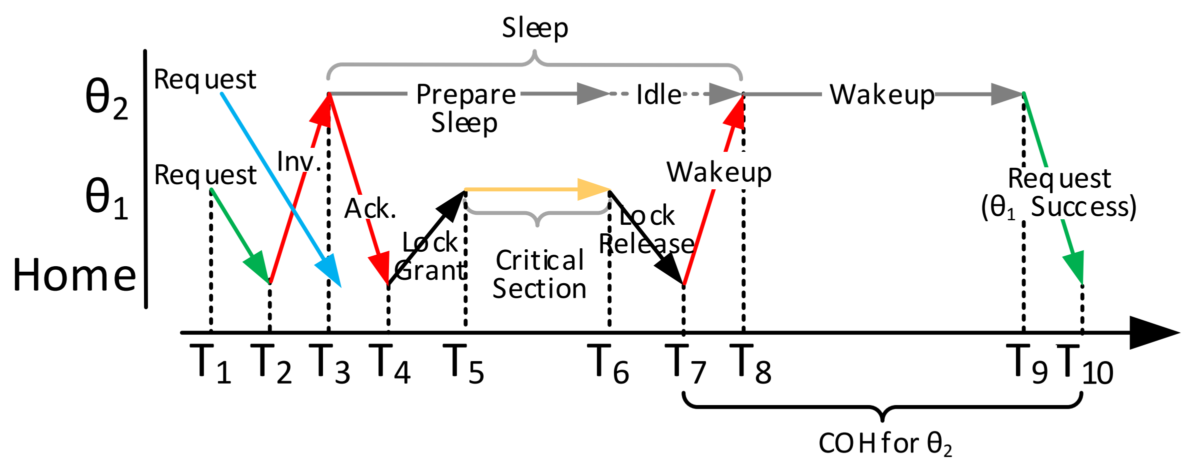
Queue spin-lock

- Assume that two threads (θ_1 and θ_2) in two different nodes compete for the same lock variable at the home node.
- Assume that θ_2's core is a sharer of the lock variable at the home node at time T1.





Low-overhead spinning phase



High-overhead sleep phase

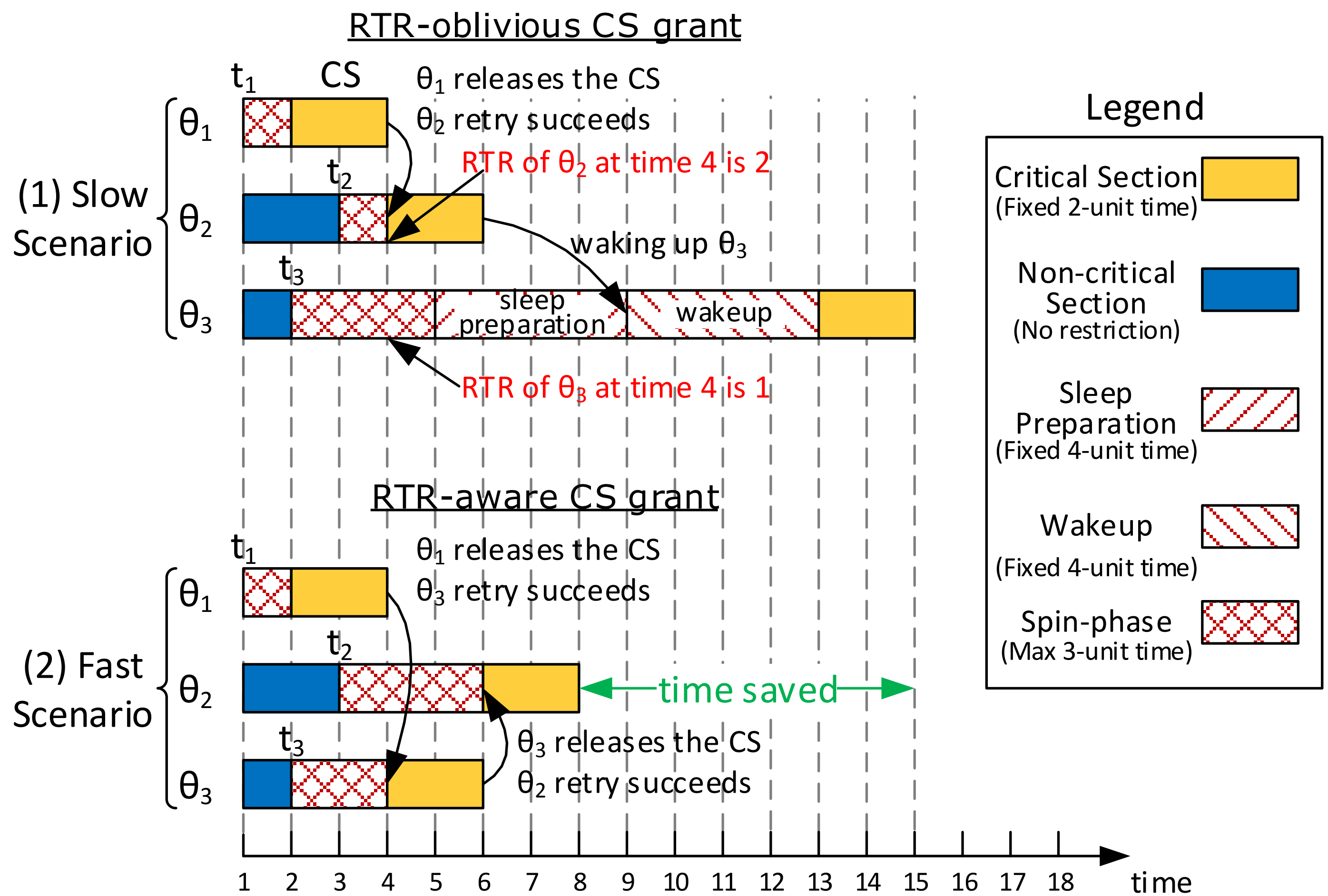
ROYAL INSTITUTE OF TECHNOLOGY

Design – Opportunistic competition overhead reduction

- Our idea
  - We develop a **software-hardware** cooperative mechanism to opportunistically reduce competition overhead (COH) by
    - maximizing the chance that a thread gets access to critical section during the low-overhead spinning phase.
    - Meanwhile, minimizing the chance that a thread gets access to critical section during the high-overhead sleep phase.

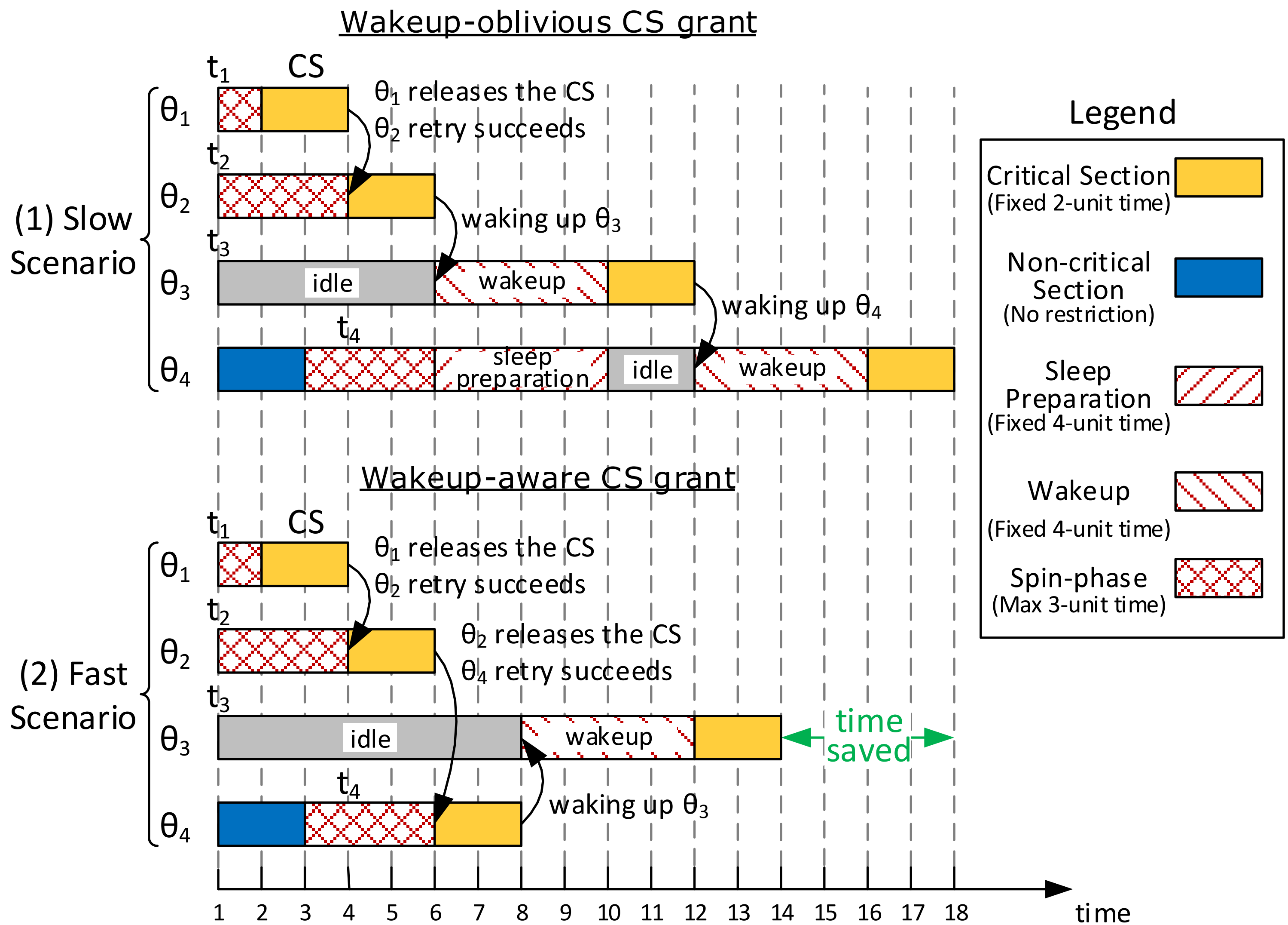
Concept – Least RTR, first grant

- We check the remaining times of retry (RTR) in a thread's spinning phase.
- RTR-oblivious CS grant may result in slow scenario.
- RTR-aware CS grant leads to fast scenario.



Concept – Wakeup request, last grant

- We postpone the critical section grant to a thread that has already been in the sleep phase.
- Wakeup-oblivious CS grant may result in slow scenario.
- Wakeup-aware CS grant leads to fast scenario.



Software level – Modification of the OS locking primitives

- We modified the default queue spinlock lock/unlock primitives.

```
function q_spinlock_lock(shared_lock *lock) {
  int c = 0;
  /* Spinning phase begins */
  for (i=0; i < MAX_SPIN_COUNT; i++) do {
    int RTR = MAX_SPIN_COUNT - i;
    write_local_reg(RTR, get_thread_PCB()->PROG);
    c = atomic_try_lock(lock); /* Atomic locking */
    if (!c) return 0;/* Atomic locking succeeds, return */
    cpu_relax();/* Otherwise, delay and retry the locking */
  }
  sys_futex(lock, FUTEX_WAIT);
}
```

```
function q_spinlock_unlock(shared_lock *lock) {
  atomic_release(lock);
  get_thread_PCB()->PROG++;
  write_local_reg(get_thread_PCB()->PROG);
  sys_futex(lock, FUTEX_WAKE);
}
```

- No need of modifying application software.

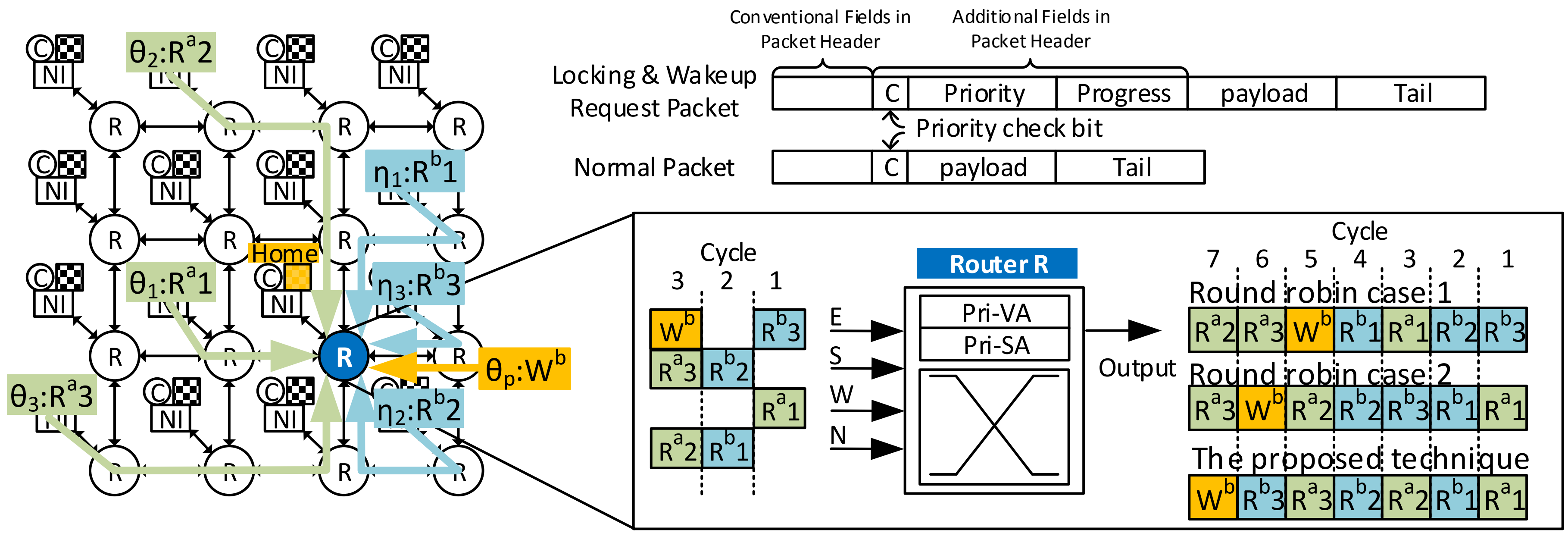
```
function pthread_mutex_lock(shared_lock *lock)
{
  …
  q_spinlock_lock(lock);
  …
}
```

```
function pthread_mutex_unlock(shared_lock *lock)
{
  …
  q_spinlock_unlock(lock);
  …
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Hardware level – Starvation-free prioritization in the NoC

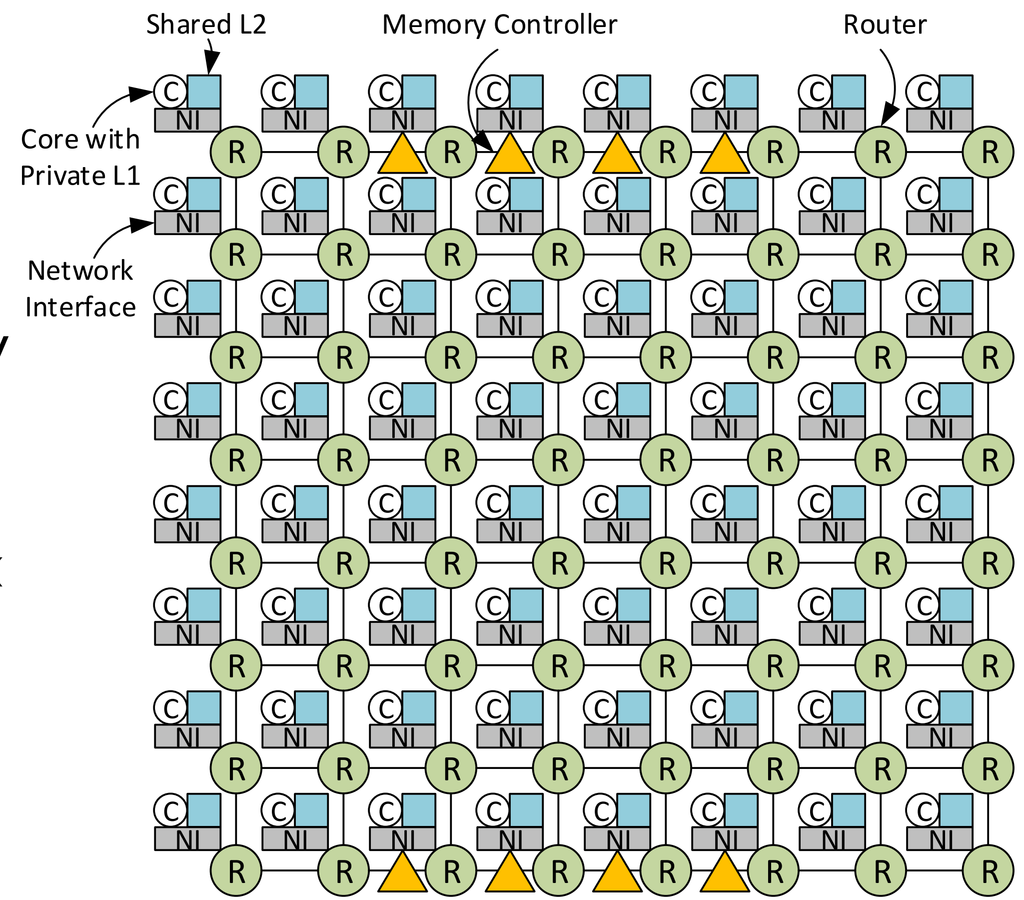
- RTR: 1,2,3
- Progress: a,b
- R: lock request
- W: wakeup



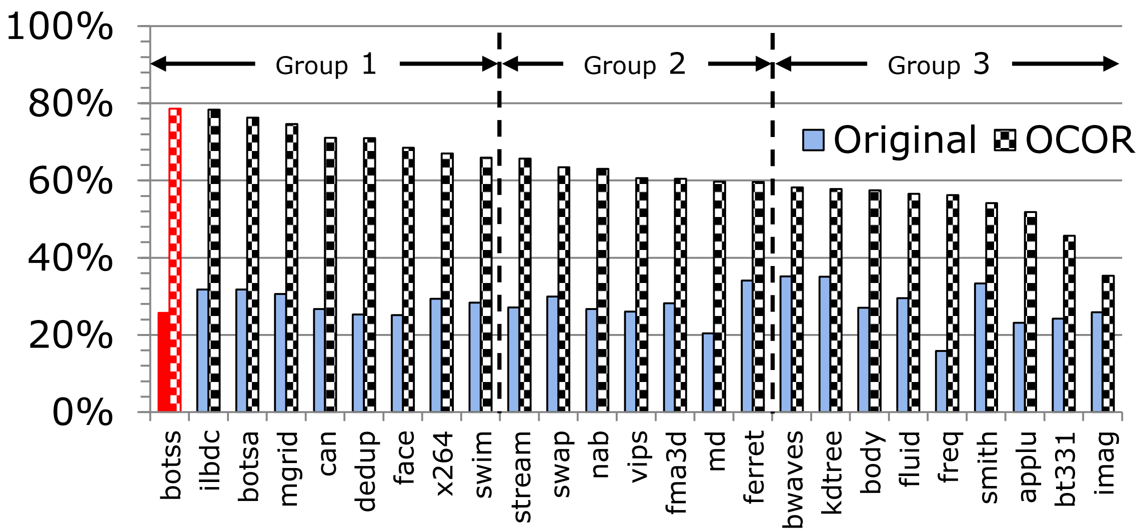
- A prioritization mechanism in VA (VC Allocation) and SA (Switch Allocation) to speed up least-RTR packets and slow down wakeup-request packets.
- To avoid starvation, packets from a slower progressing thread are prioritized over packets from a faster progressing thread.
  - Thread execution progress is obtained by get_thread_PCB()->PROG.

Experimental setup

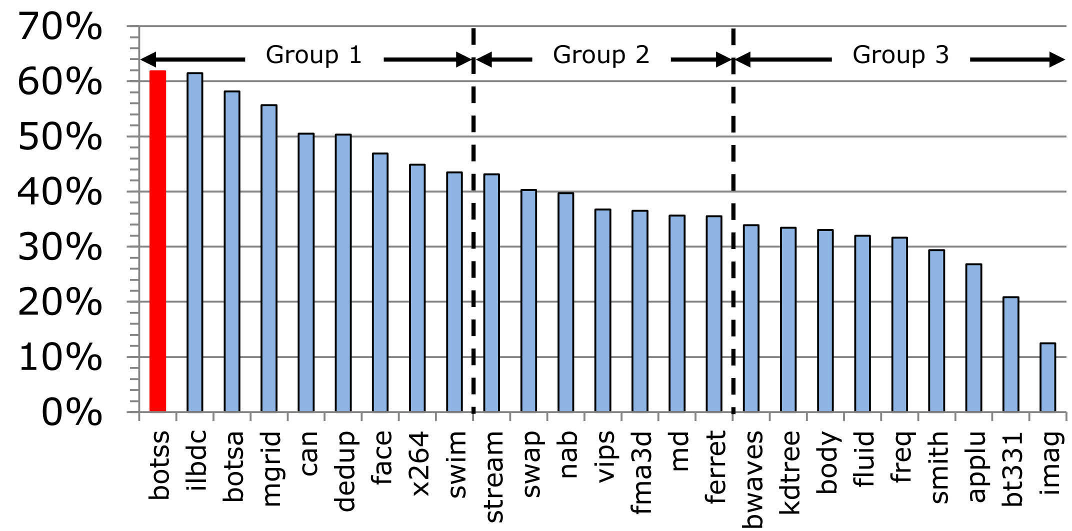
- Simulator: GEM5.
- Benchmark: PARSEC (11 programs) and SPEC OMP2012 (all 14 programs).
  - No *Blackscholes,* because it only uses barrier for sync.
- Software-level modification:
  - Queue spinlock (`mutex`) in Linux 4.2.
    - Locking/unlocking functions used in pthread and OpenMP libraries.
- Hardware-level modification:
  - VA and SA of the GARNET router.



Experimental results – COH reduction



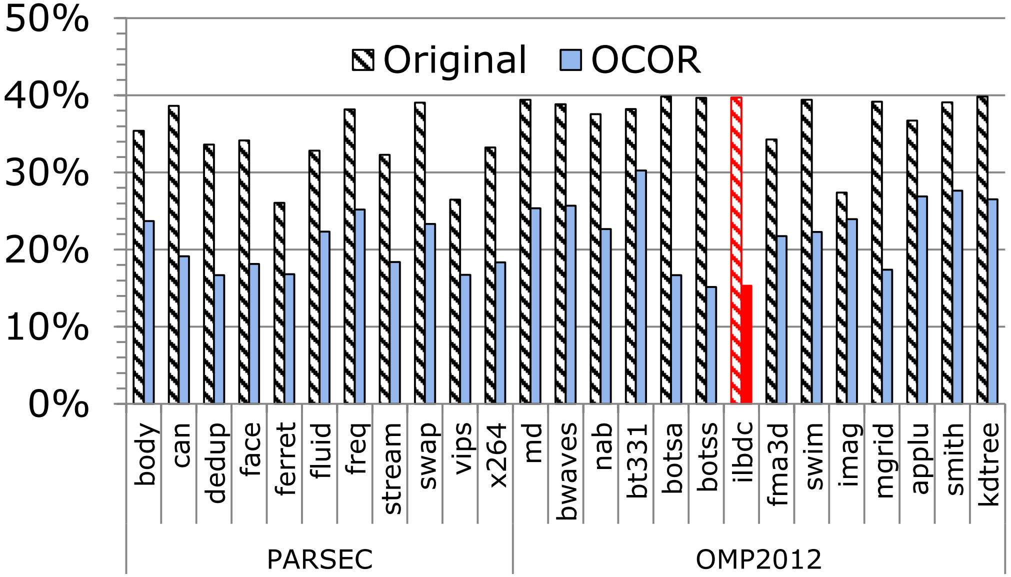
CS access percentage in spinning phase



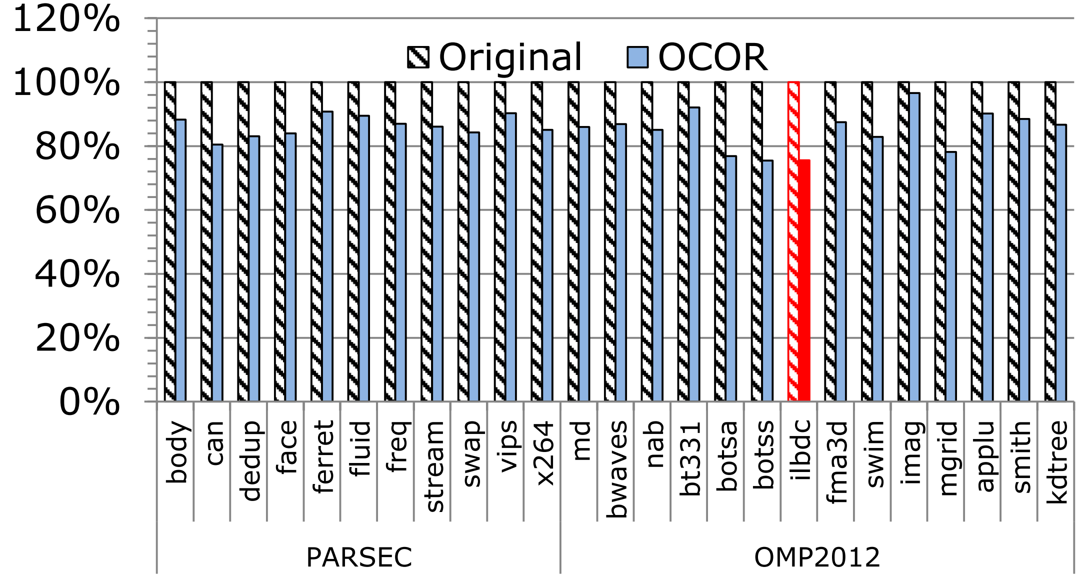
COH reduction with OCOR

- OCOR (Opportunistic Competition Overhead Reduction) increases the chance of a thread securing critical section in the low-overhead spinning phase.
  - Based on critical section access rate and network utilization, we divide all benchmarks into 3 groups (more details in the paper).
- With OCOR, COH is constantly reduced across all benchmarks.
  - Maximum reduction in *botss* (61.8%).
  - Average reduction reaches 40.4% for PARSEC, 39.3% for OMP2012 programs.

Experimental results – ROI acceleration



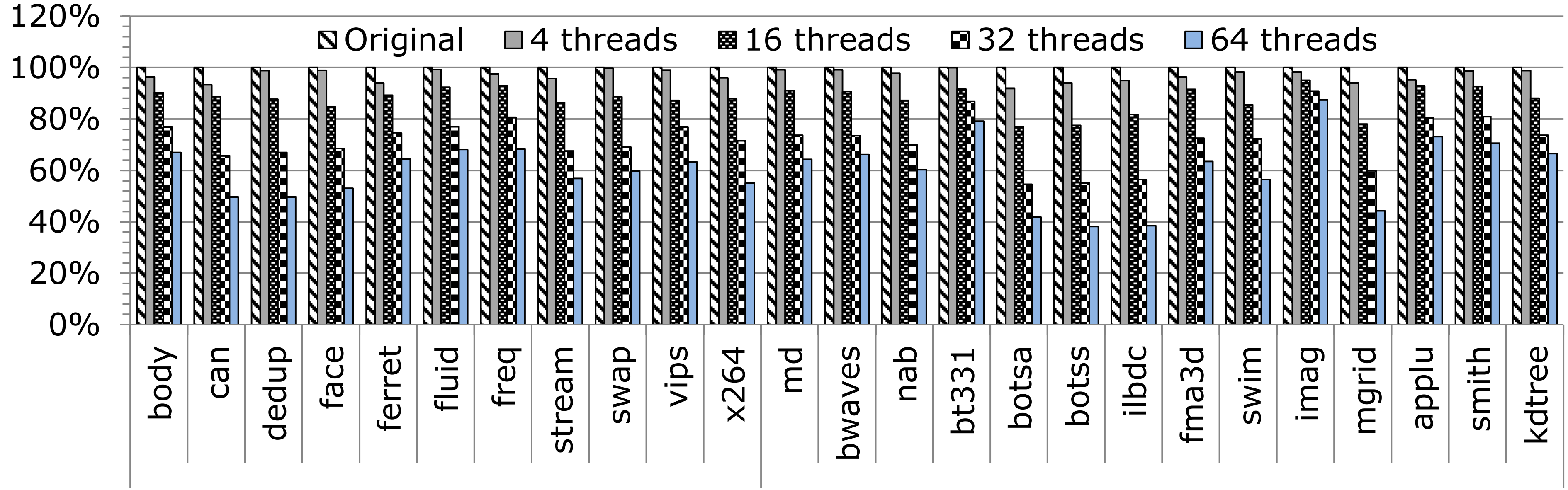
COH percentage in ROI finish time



Comparison in ROI finish time

- OCOR constantly reduces COH percentage in ROI finish time.
- OCOR thus accelerates application ROI execution.
  - Maximum reduction in *libdc* (24.5%).
  - Average reduction reaches 13.7% for PARSEC, 15.1% for OMP2012.

Experimental results – Scalability



COH percentage for all benchmarks running in 4, 16, 32, 64 threads

- We normalize the competition overhead without OCOR to 100%.
- OCOR constantly reduces the competition overhead across all benchmarks.
- The more threads spawned, the larger COH reduction achieved.

ROYAL INSTITUTE OF TECHNOLOGY

Summary

- Problem: **competition overhead** is a major source of thread's blocking time, exceeding the execution time of CS itself.
- Central idea: **opportunistically**
  - maximize the chance that a thread wins the CS access in the **low-overhead spinning phase**;
  - minimize the chance that a thread wins the CS access in the **high-overhead sleep phase**.
- Approach: a **software-hardware** cooperative technique that can effectively reduce the competition overhead of threads accessing critical sections.
- Experimental results: Our technique significantly **reduces** the competition overhead, **improves** the ROI finish time, and achieves **scalable** gains across all benchmark programs.

Thank you!

Q & A