EMERALDS: a small-memory real-time microkernel

By Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin

Outline

- Introduction
- Requirements
- EMERALDS Overview
- CSD Scheduler
- Semaphore implementation
- Inter-task communication
- Conclusion

Introduction

- EMERALDS = Extensible Microkernel for Embedded, ReAL-time Distributed Systems
- Most RTOS were designed for powerful systems
- Real-time embedded controllers are getting popular, running on top of minimal hardware
  - Implement core OS services using optimized, carefully crafted code
- EMERALDS's approach: new OS schemes and algorithms
  - Small kernel and application code size
  - Priori knowledge of task communication and execution pattern to reduce overhead

Requirements

- Slow processing speed (15-25 MHz) for single chip micro-controllers
- Limited ROM/RAM size (32-128 kbyte)
  - 20 kbytes RTOS kernel code size
- Uniprocessor or distributed nodes
  - Low speed field bus network (1-2 Mbps)
- Design goal
  - 10-20 concurrent, periodic real-time tasks
  - Interrupt and I/O services
  - No disk or file system support
  - Task synchronization, communication and clock service

EMERALDS Overview

- A micro-kernel RTOS written in C++
- Features supported
  - Multi-threaded processes
  - IPC (message passing, mailboxes and shared memory)
  - Semaphores and condition variables
  - Communication protocol stacks
  - Optimized context switching and interrupt handling
  - User level device drivers

Scheduling in embedded applications

- 1st try: cyclic time-slice scheduling
  - Calculate entire schedule offline
  - Modification of schedule due to task characteristic change is difficult and costly
  - High-priority aperiodic task has poor response time
  - Result in large time-slice schedules if work load contains short and long period tasks

CSD Scheduler

- Priority driven scheduler: a combination of EDF and RM
- 2 components of task scheduler overhead
  - Run-time overhead
  - Schedulability overhead

Run-time overhead

- Time consumed by executing scheduler code
  - Parsing queue tasks, add/delete tasks from queue
  - Blocking overhead (Δt_b)
  - Unblocking overhead (Δt_u)
  - Selection overhead (Δt_s)
- Total overhead in each period when blocking system calls are used
  - $\Delta t = 1.5(\Delta t_b + \Delta t_u + 2\Delta t_s)$
- Total workload utilization
  - $U = \sum (C_i + \Delta t)/P_i,\ i = 1 \sim n$

Overhead comparison between EDF and RM

- EDF overhead (single, unsorted queue)
  - Δt_b, Δt_u: O(1)
  - Δt_s: O(n)
- RM overhead (single, sorted queue)
  - Δt_b: O(n) (set pointer point to next ready task)
  - Δt_u: O(1)
  - Δt_s: O(1)
- RM has significant less run-time overhead than EDF since
  - $\Delta t = 1.5(\Delta t_b + \Delta t_u + 2\Delta t_s)$

Schedulability overhead

- Defined as 1 – U^*, where U^* is the idea schedulable utilization
- Utilization $U = \sum c_i/P_i$, $i = 1 \sim n$
- $U^* = 1$ for EDF $\rightarrow$ no schdulability overhead
- $U^* = 0.88$ (0.69!?) for RM

CSD: a mixed approach

- CSD = Combined Static/Dynamic
- Given a workload, it may be feasible under EDF but not RM
- Identify the “trouble maker”: the task that is schedulable in EDF but infeasible in RM
  - Use **EDF** for tasks that has higher rate up to the troublesome task
  - Use **RM** for the remaining tasks

Example workload

i	1	2	3	4	5	6	7	8	9	10
P_i (ms)	4	5	6	7	8	20	30	50	100	130
c_i (ms)	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.5

CSD (con't)

- Maintain 2 queues of tasks
  - Dynamic priority (DP) queue for EDF
  - Fixed priority (FP) queue for RM
- DP queue is given higher priority
  - Execute tasks in DP queue if at least one task is ready
  - Otherwise pick one from FP queue with highest priority ready task

Run-time overhead of CSD

- 4 possible cases
  - DP task blocks: Δt_b = O(1), Δt_s = O(r)
  - DP task unblocks: Δt_u = O(1), Δt_s = O(r)
  - FP task blocks: Δt_b = O(n-r), Δt_s = O(1)
  - FP task unblocks: Δt_u = O(1), Δt_s = O(r)
- Total CSD overhead
  - $\Delta t_b + \Delta t_{s\_block} + \Delta t_u + \Delta t_{s\_unblock}$
  - DP: O(1) + O(r) + O(1) + O(r) = 2O(r)
  - FP: O(n-r) + O(1) + O(1) + O(r) = O(n)

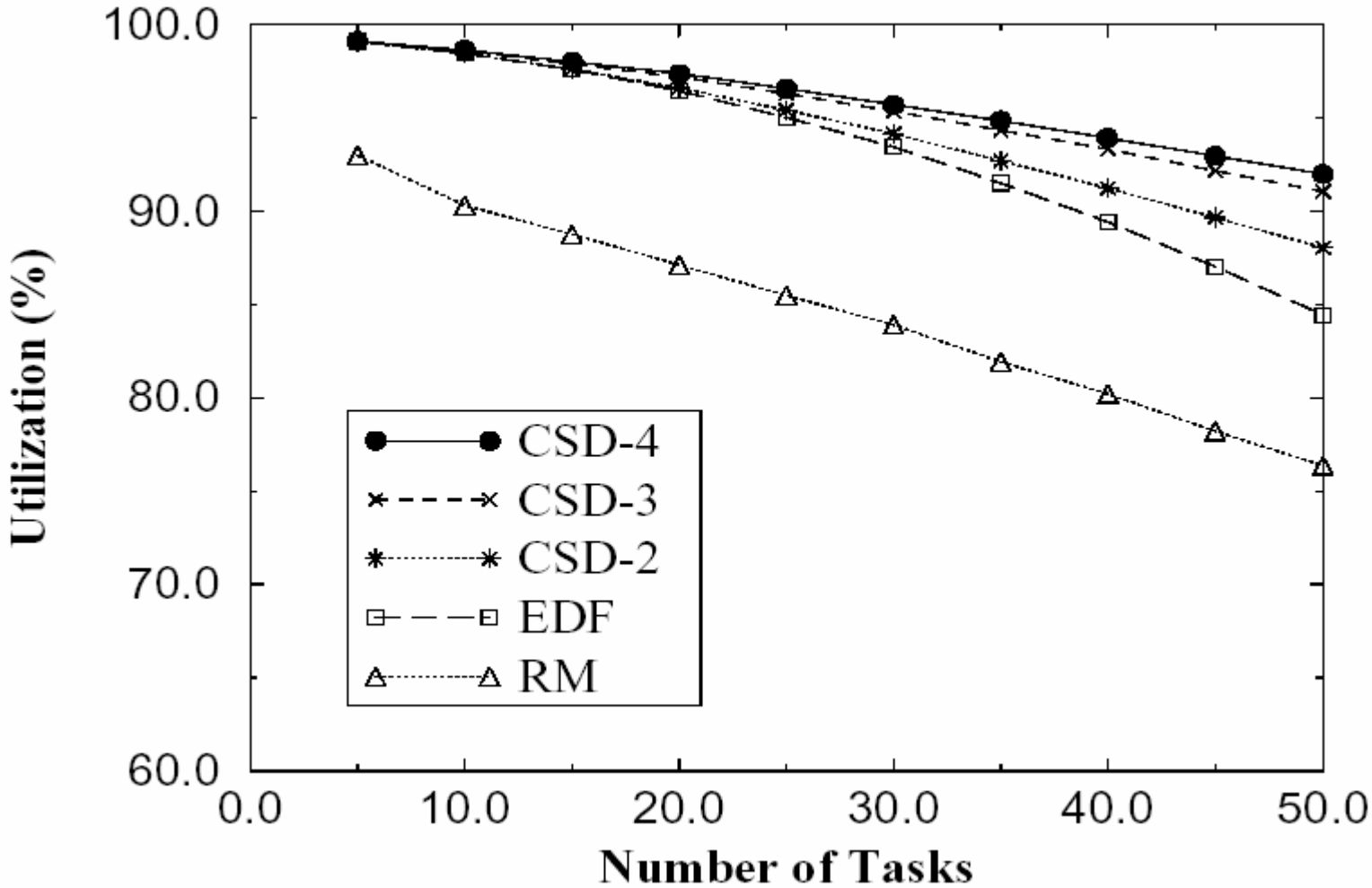
Reducing CSD run-time overhead

- CSD reduces runtime overhead by keeping DP queue length short
- As number of tasks increases, performance degrades rapidly
- Solution: split DP queue into sub-queues
- CSD-3: 2 DP queues (DP1 & DP2) + 1 FP queue
  - DP1 contains tasks having the shortest periods, reduction in overhead greatly improves performance

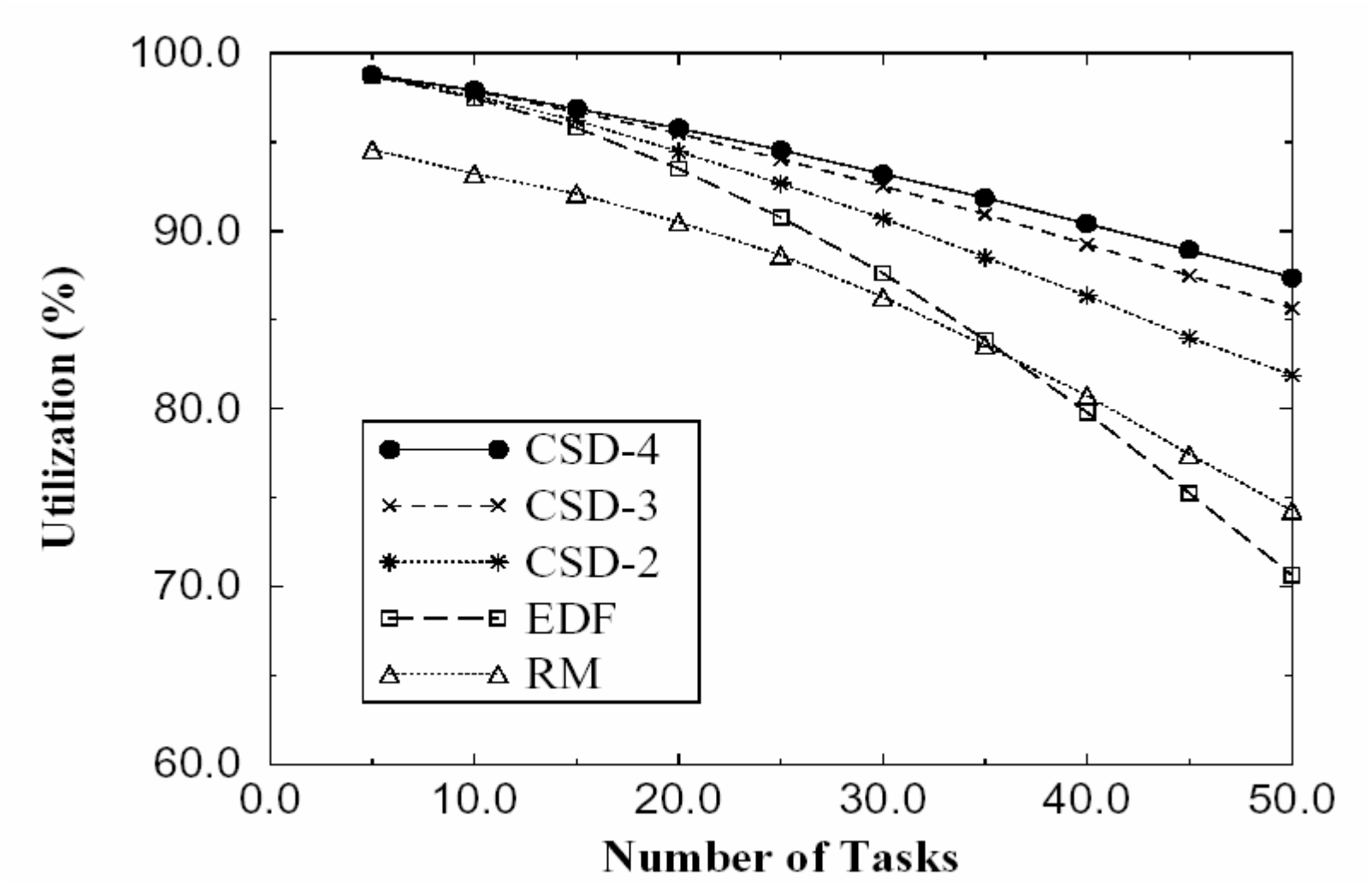
Task allocating into sub queues

- When tasks have different periods
  - Keep only few tasks in DP1 to keep $\Delta t(DP1)$ small
  - Multiple DP queues will result in non-zero schedulability overhead
- Beyond CSD-3
  - Increase in schedulability overhead will exceed reduction in runtime overhead as number of queues gets larger
  - 2 extreme cases: 1 queue and n queues are equivalent to RM

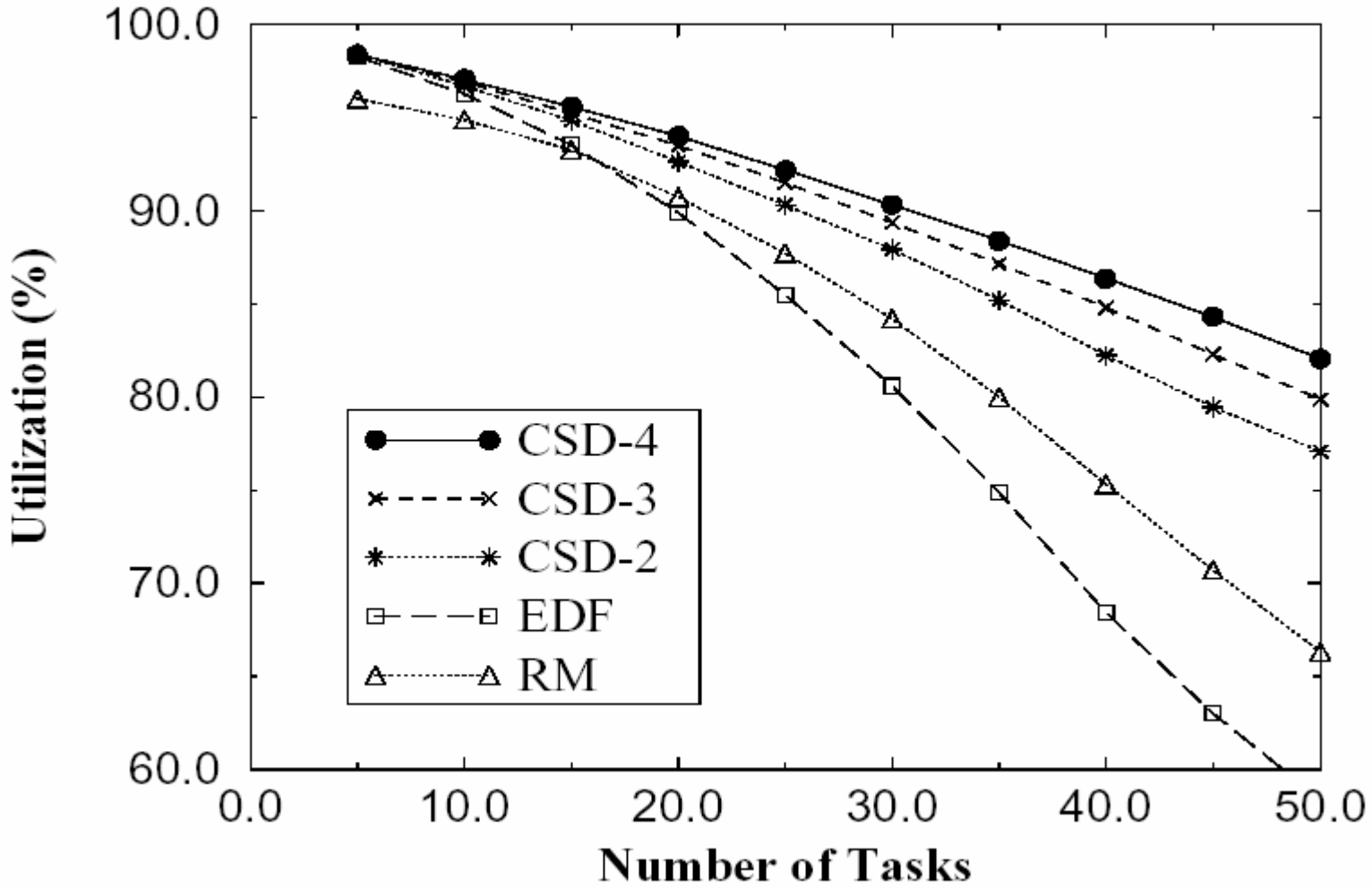
CSD Performance



CSD Performance (period scale down by 2)



CSD Performance (period scale down by 3)



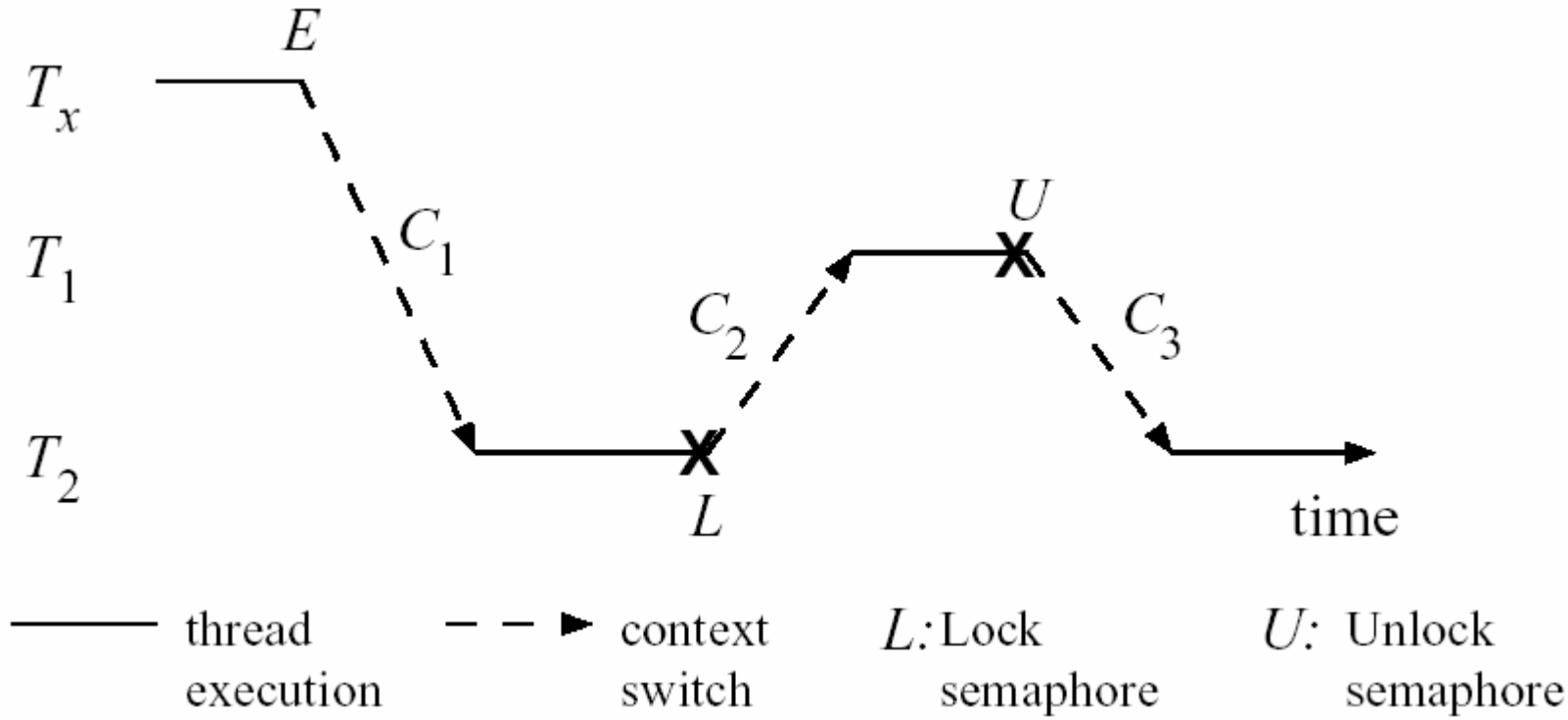
Semaphore Implementation

- Standard semaphore implementation

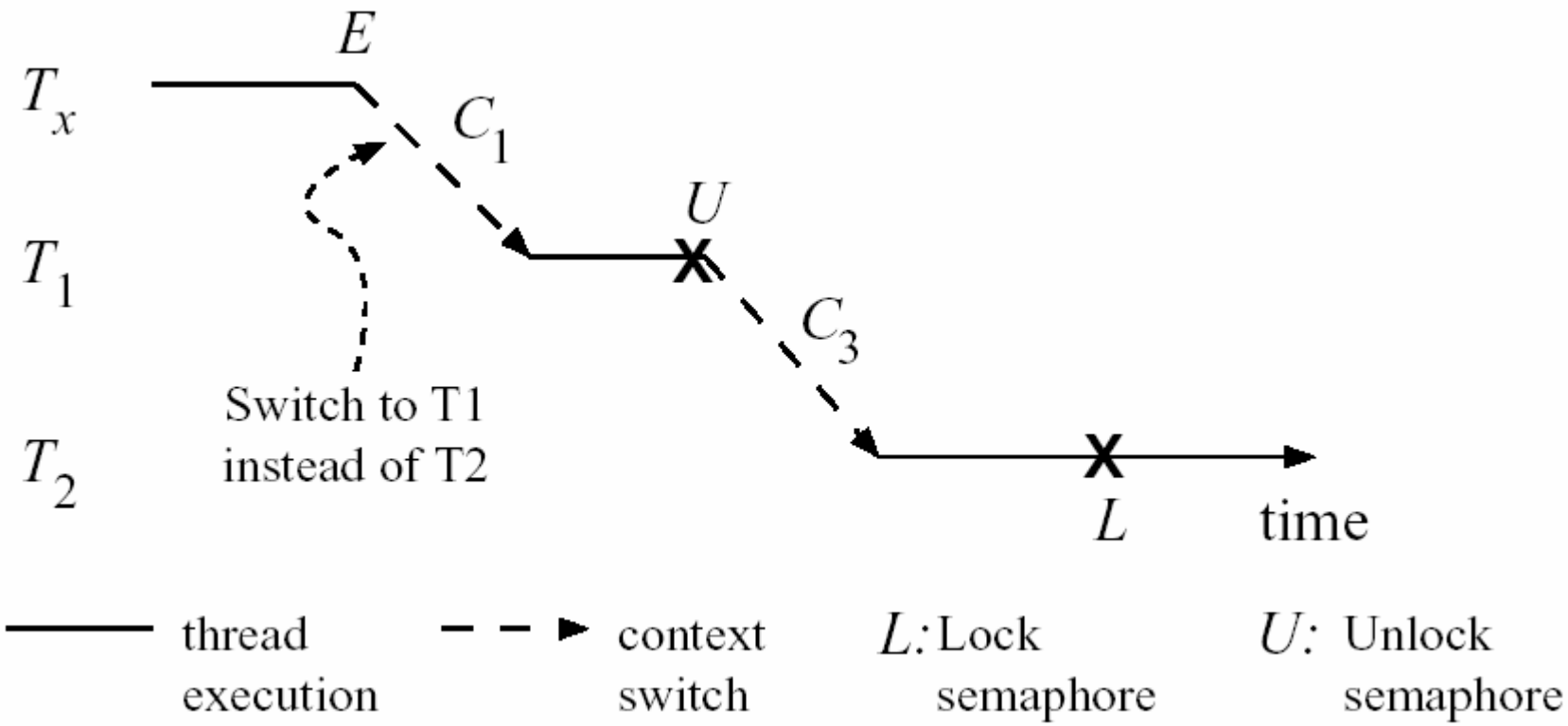
```
If (sem locked)
{
        do priority inheritance;
        add caller thread to wait queue;
        block; //and wait for sem to be released
}
Lock sem
```

Semaphore Implementation (Con't)

- Need Priority inheritance to avoid unbounded priority inversion



Reduce context switches



Reduce context switches (Con't)

- Rethink priority inheritance for both DP and FP tasks
- Letting lower priority task who holds semaphore continue to execute
- Passing extra parameter to indicate which semaphore to lock when calling acquire_sem()

Reducing Context switches for FP tasks

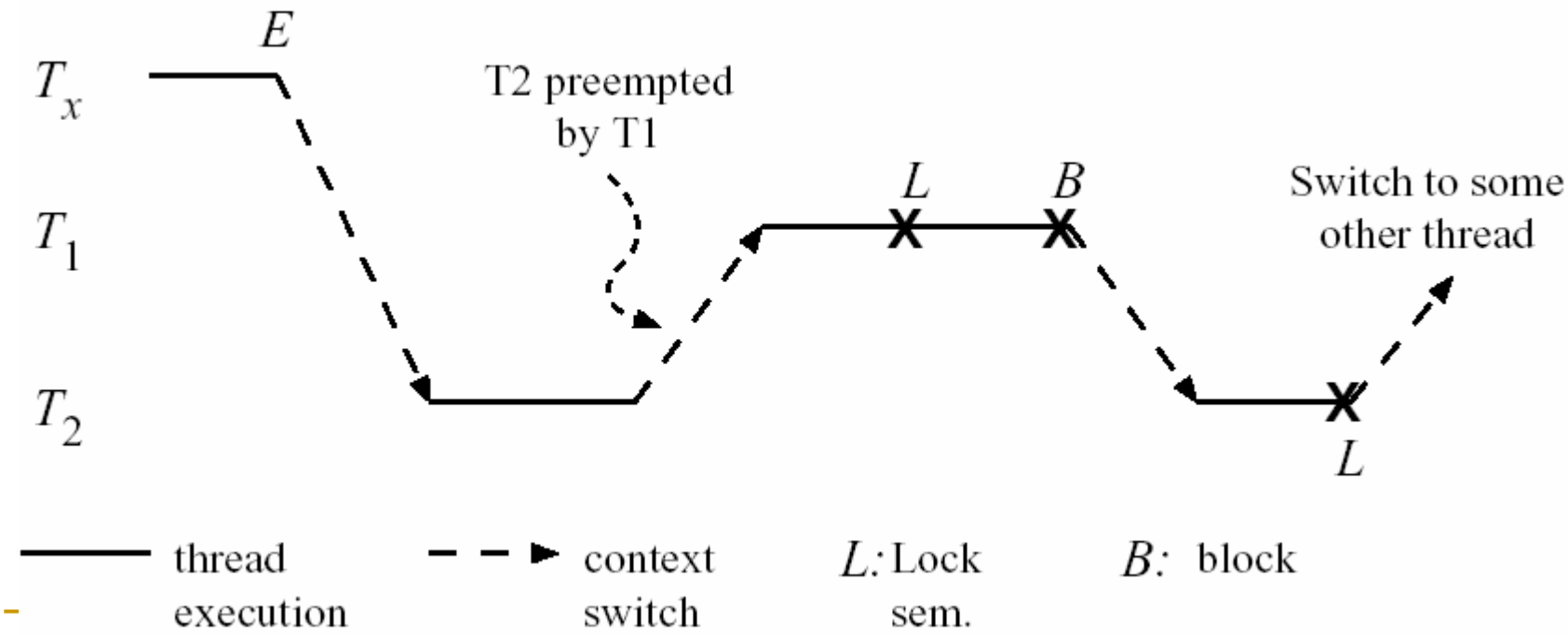
- O(n-r) complexity for normal queue operation
- Optimizing priority inheritance to O(1)
  - Insert T_1 directly ahead of T_2 (T_1 inherits T_2's priority)
  - Swap position in queue between T_1 and T_2 (return T_1 to its original priority)
- If thread T_3 came in while T_1 inherits T_2's priority
  - Swap between T_1 and T_3, and put T_2 back
- Require FP to keep ready and blocked tasks in the same queue

Analysis of new scheme

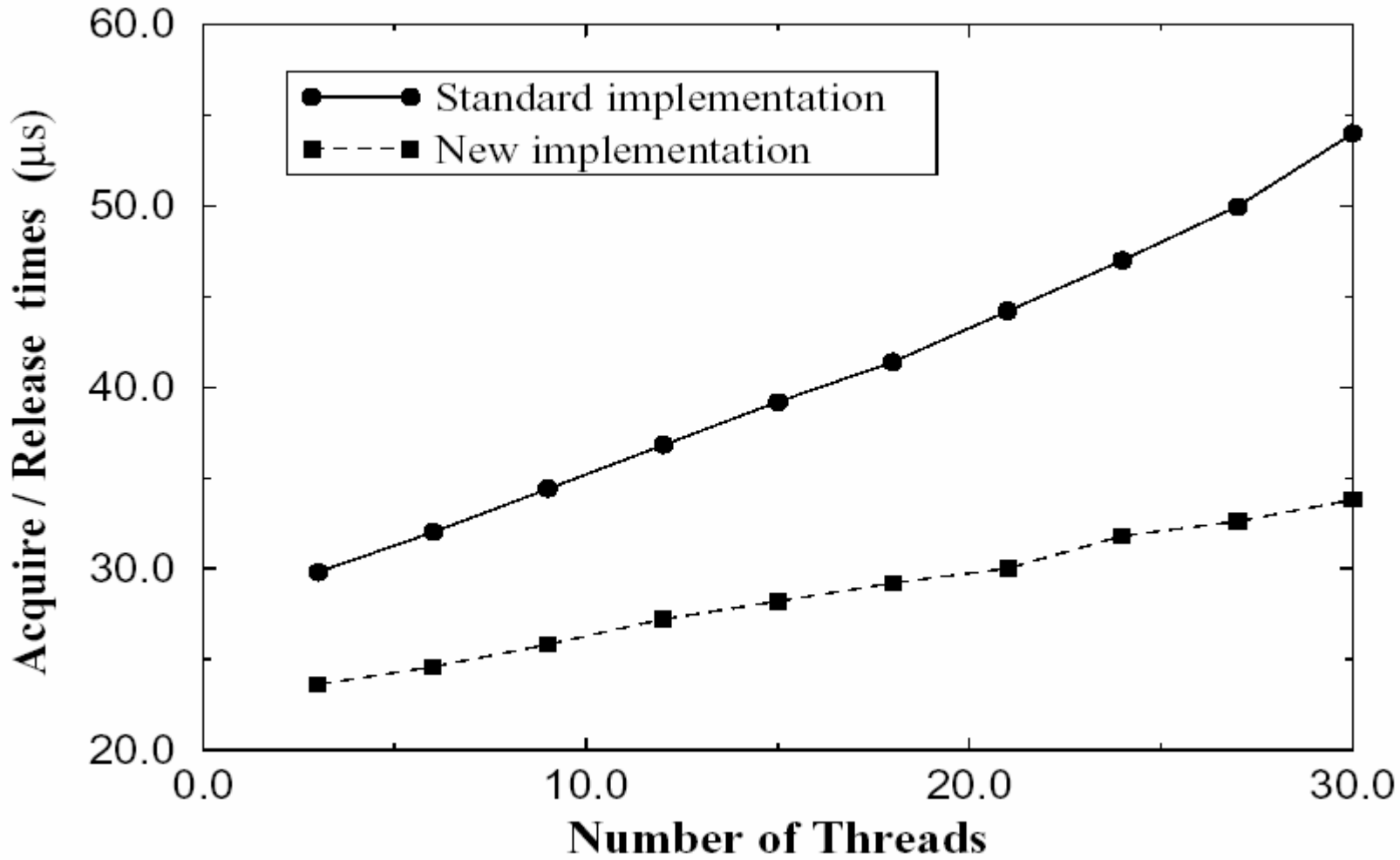
- What if thread T_2 does not block on acquire_sem?
- Is it save to delay T_2?

What if T1 has higher priority?

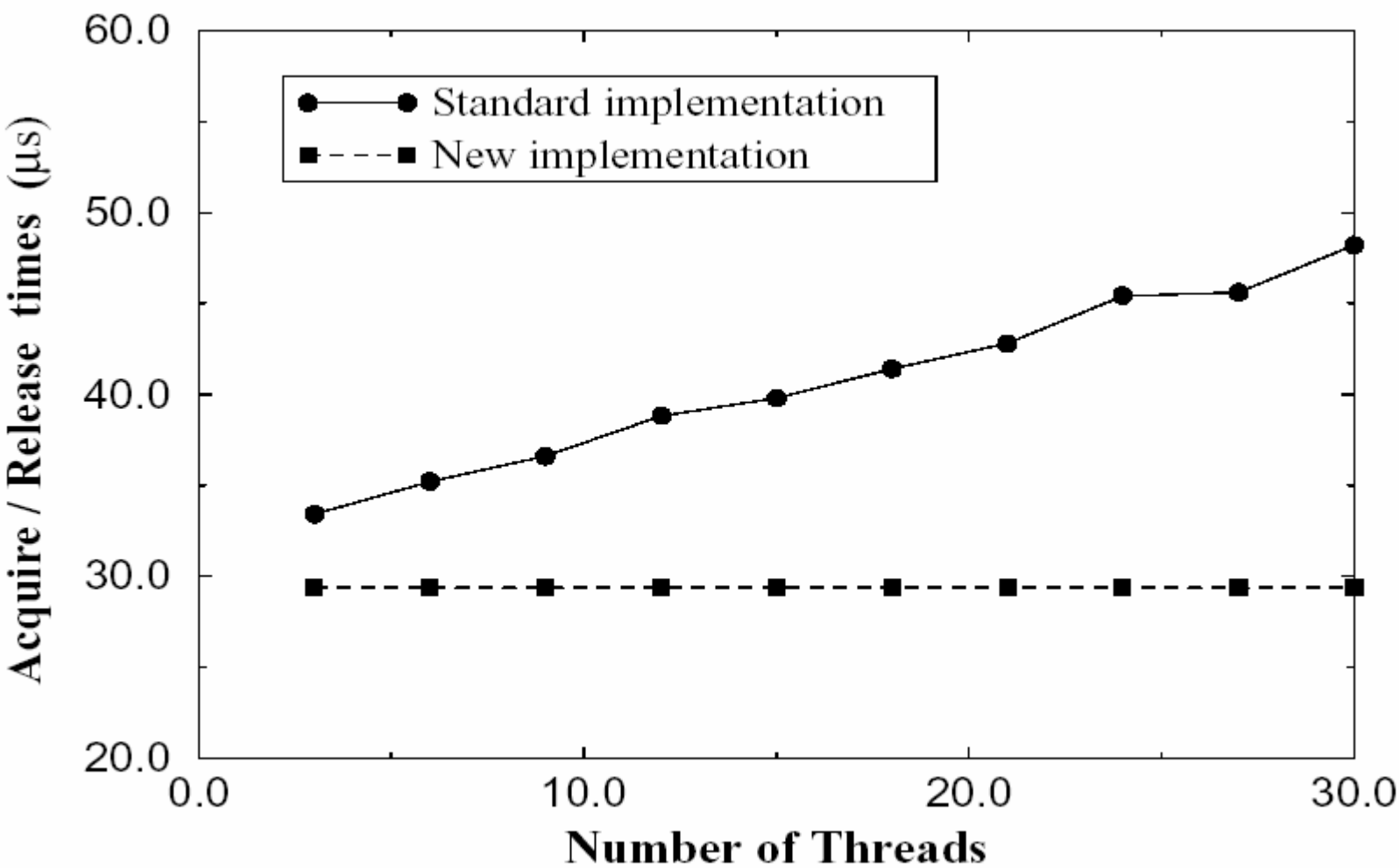
- T_2 incurs full overhead of acquire_sem()
- Sol: block T_2 when higher priority thread locks S, and unblock T_2 when S is released



Semaphore scheme performance (DP)



Semaphore scheme performance (FP)



Inter-task communication

- Traditional Mechanism: mailbox
  - Invoke system call to send message
  - High overhead
  - Need to send same message multiple times to multiple tasks
- Solutions
  - Global variables
  - State message passing

State message semantics

- Solve single writer, multiple reader problem
- A mailbox is associated with a writer
  - One writer sends message to SMmailbox
  - Multiple readers can receive message
  - New message overwrites old one
  - Reads do not consume message
  - Both reads and writes are non-blocking

State message implementation

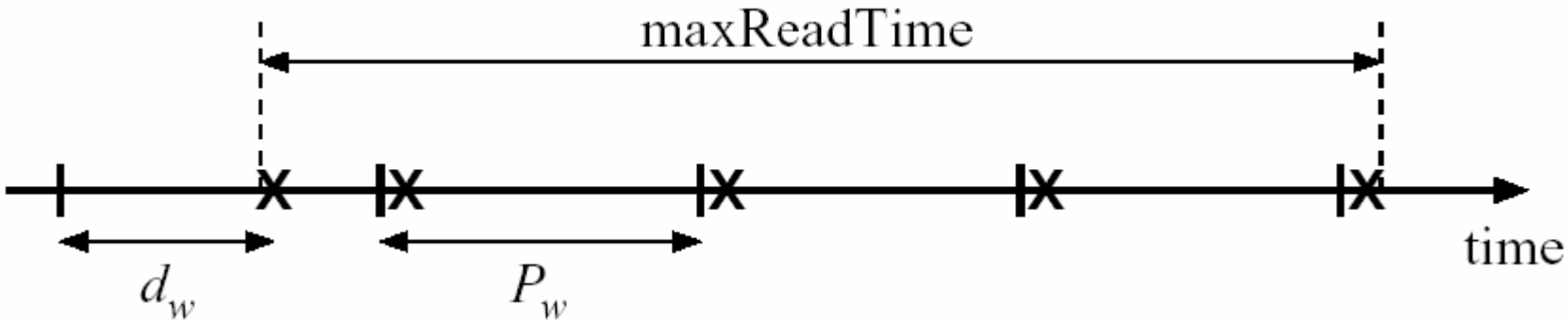
- B: max number of bytes CPU can read/write
- L: message length
- Simple case: L ≤ B
- If L > B
  - assign N-deep circular buffer to each state message
  - Each message has a 1-byte index

Calculate buffer depth N

- x: number of write operations can occur while reader is blocked
  - $N = \max(2, x_{max} + 1)$
- $maxReadTime = d – (c - c_r) = (d – c) + c_r$
  - d: deadline
  - c: execution time
  - c_r: read execution time

Calculate max write times

- x_{max} -1 = floor[(maxReadTime – (P_w - d_w))/P_w]
  - p_w: writer's period
  - d_w: writer's deadline



Conclusions

- EMERALDS provides key OS services with significantly lower overhead
  - CSD scheduler creates balance between static and dynamic scheduling
  - New semaphore reduces context switch and priority inheritance overhead by 20~30%
  - State-message paradigm incurs ¼~1/5 overhead
- Future work
  - Networking issues