A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real-Time Systems

Hyoseung Kim
Arvind Kandhalu
Prof. Raj Rajkumar

Electrical and Computer Engineering
Carnegie Mellon University

Why Multi-Core Processors?

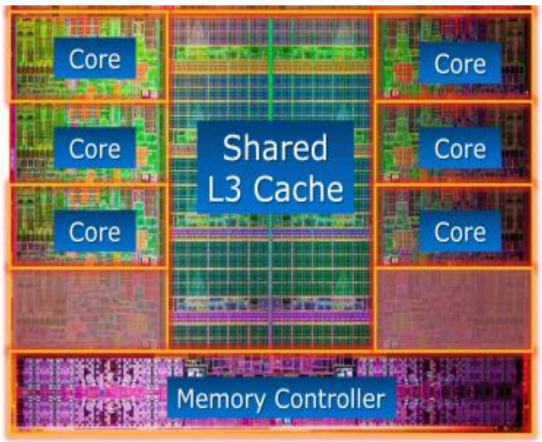
- **Processor development trend**
  - Increasing overall performance by integrating multiple cores

- **Embedded systems: Actively adopting multi-core CPUs**
  - **Automotive:**
    - Freescale i.MX6 Quad-core CPU
    - Qorivva Dual-core ECU

  - **Avionics and defense:**
    - COTS multi-core processors
    - ex) Rugged Intel i7-based single board computers

Multi-Core CPUs for Real-Time Systems

- **Large shared cache in COTS multi-core processors**



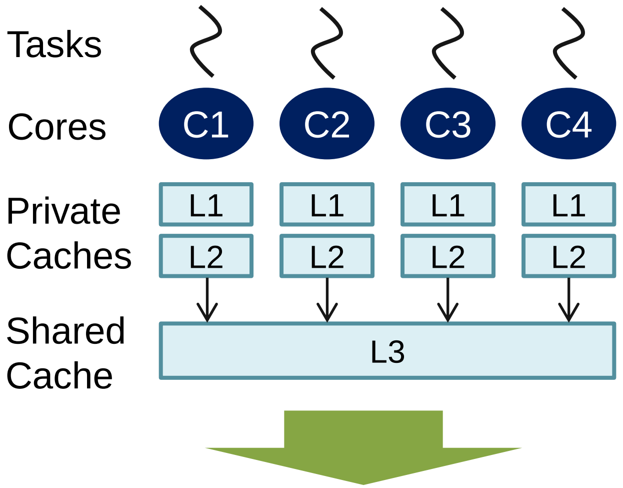
Intel Core i7
8-15 MB L3 Cache

Freescale i.MX6
1MB L2 Cache

- **Use of shared cache in real-time systems**
  - Reduce task execution time
  - Consolidate more tasks on a single multi-core chip processor
  - Implement a cost-efficient real-time system

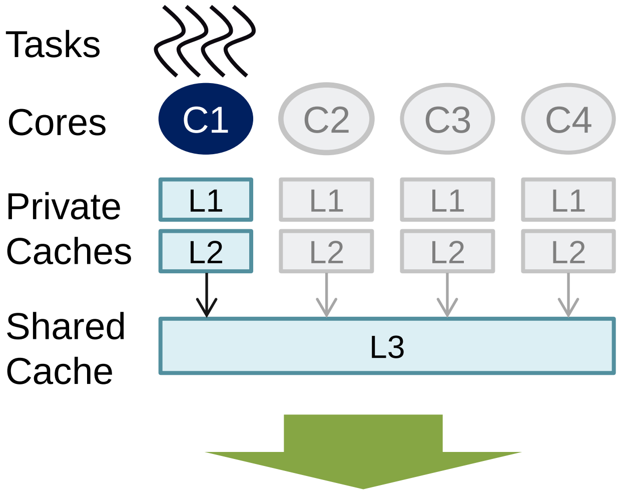
Uncontrolled Shared Cache

1. Inter-core Interference



40% Slowdown*

2. Intra-core Interference



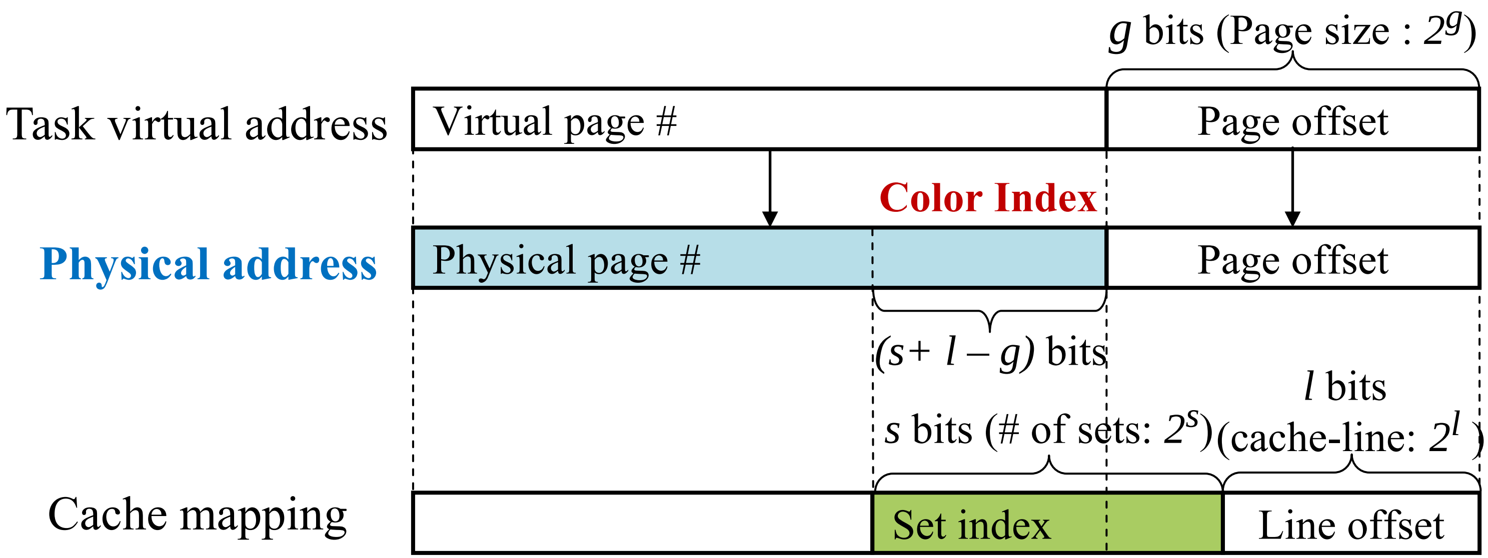
27% Slowdown*

Uncontrolled use of shared cache
→ Severely degrade the predictability of real-time systems

* PARSEC Benchmark on Intel i7

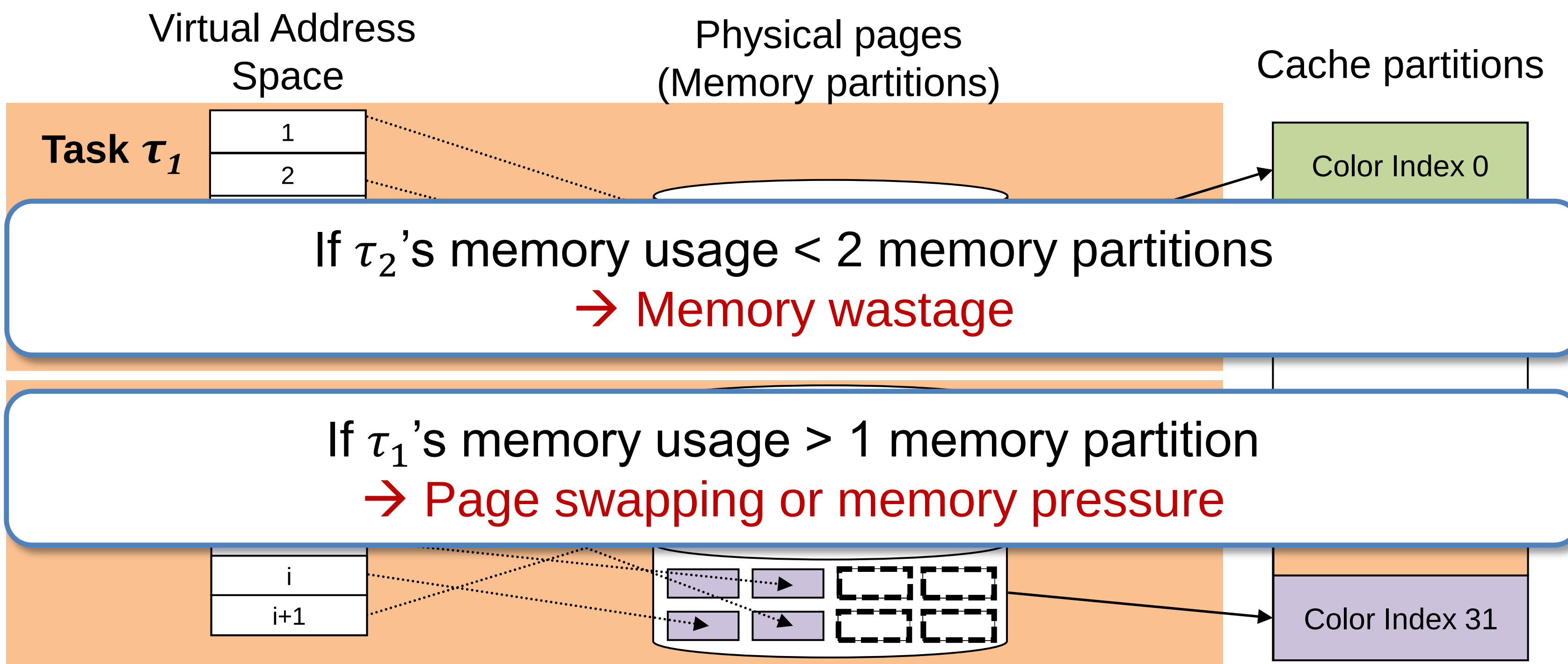
Cache Partitioning

- **Page coloring (S/W cache partitioning)**
  - Can be implemented on COTS multi-core processors
  - Provides cache performance isolation among tasks



Problems with Page Coloring (1/2)

1. **Memory co-partitioning problem**
   - Physical pages are grouped into memory partitions
   - **Memory usage ≠ Cache usage**

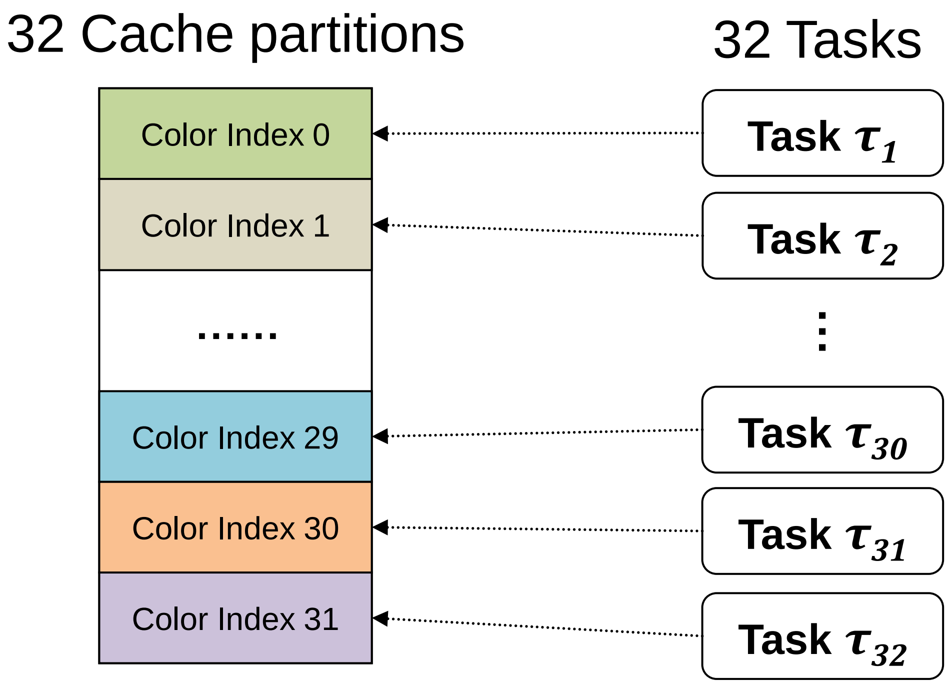


If τ_2's memory usage < 2 memory partitions
→ Memory wastage

If τ_1's memory usage > 1 memory partition
→ Page swapping or memory pressure

Problems with Page Coloring (2/2)

2. **Limited number of cache partitions**
   - Results in degraded performance as the number of tasks increases
   - The number of tasks cannot exceed the number of cache partitions



Our Goals

- **Challenges**
  - **Uncontrolled shared cache:** Cache interference penalties
  - **Cache partitioning (page coloring):**
    - Memory co-partitioning → Memory wastage or shortage
    - Limited number of cache partitions

- **Key idea:** Controlled sharing of partitioned caches while maintaining timing predictability

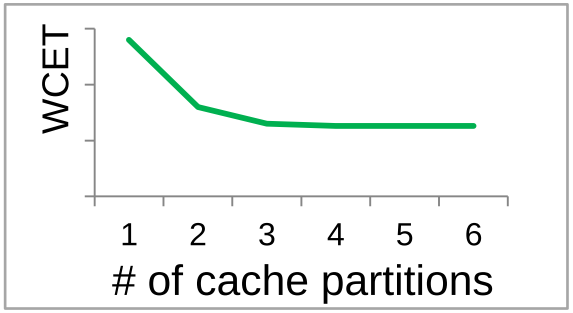
1. Provide predictability on multi-core real-time systems
2. Mitigate the problems of memory co-partitioning, limited partitions
3. Allocate cache partitions efficiently

Outline

- Motivation
- **Coordinated Cache Management**
  - System Model
  - Per-core Cache Reservation
  - Reserved Cache Sharing
  - Cache-Aware Task Allocation
- **Evaluation**
- **Conclusion**

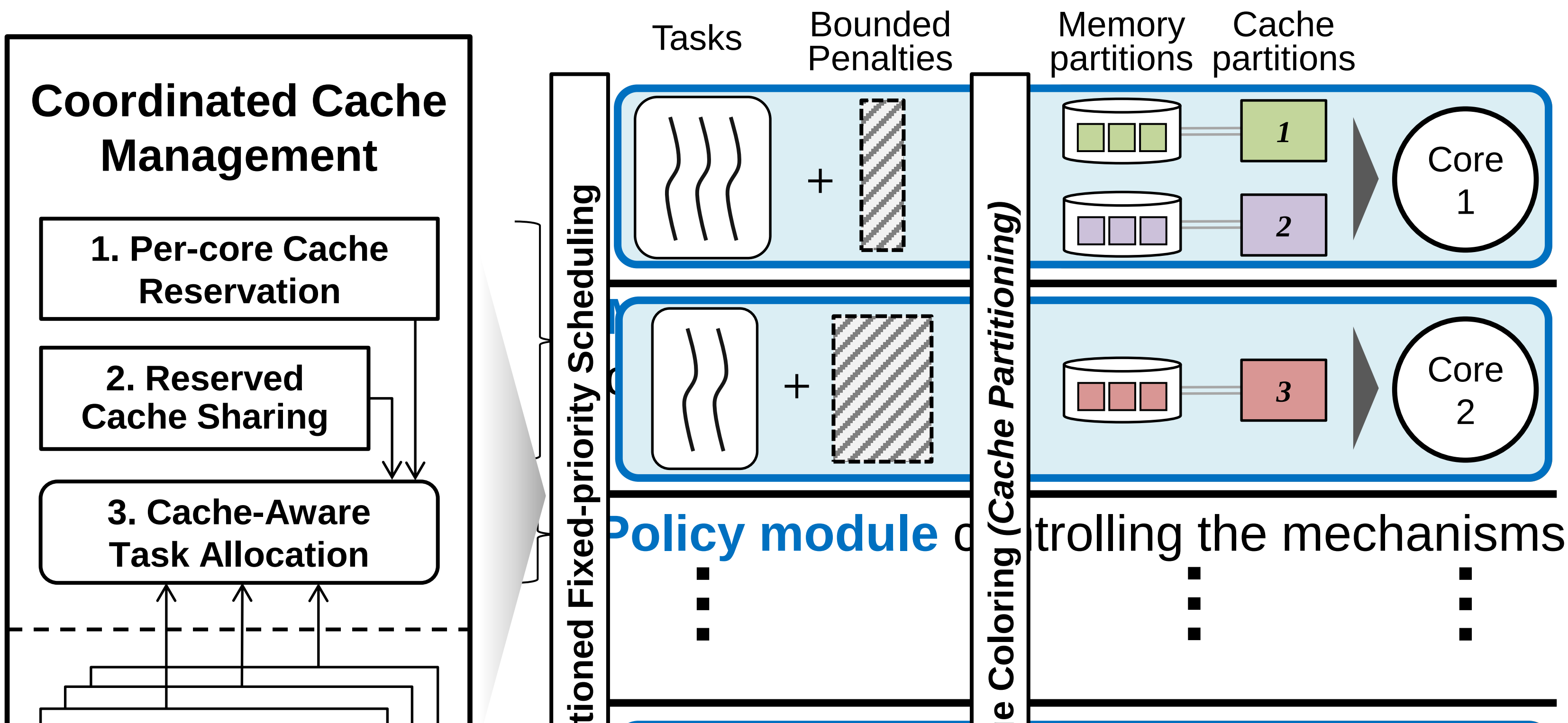
System Model

- **Task Model** $\tau_i: (C_i^p, T_i, D_i, M_i)$
  - C_i^p: Worst-case execution time (WCET) of task τ_i, when it runs alone in a system with p cache partitions
    → C_i^p is non-increasing with p
  - T_i: Period of task τ_i
  - D_i: Relative deadline of task τ_i
  - M_i: Maximum physical memory requirement of task τ_i



- **Partitioned fixed-priority preemptive scheduling**
- **Assumptions**
  - Tasks do not self-suspend
  - Tasks do not share memory

Coordinated Cache Management

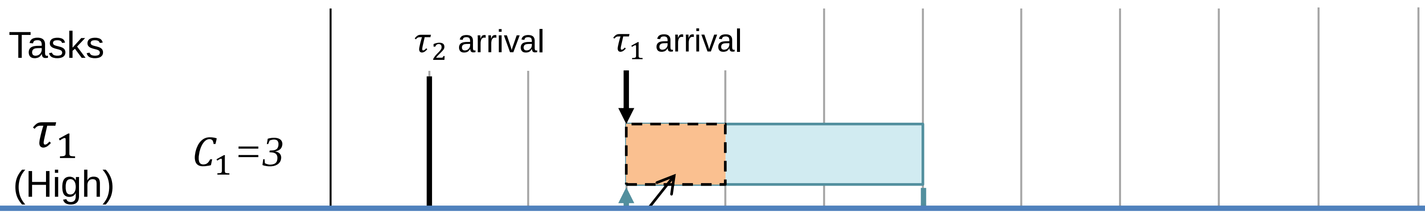


Reserved cache sharing: Mitigate the problems with page coloring

Considerations
1. Preserving schedulability
2. Guaranteeing memory requirements

Intra-Core Cache Interference

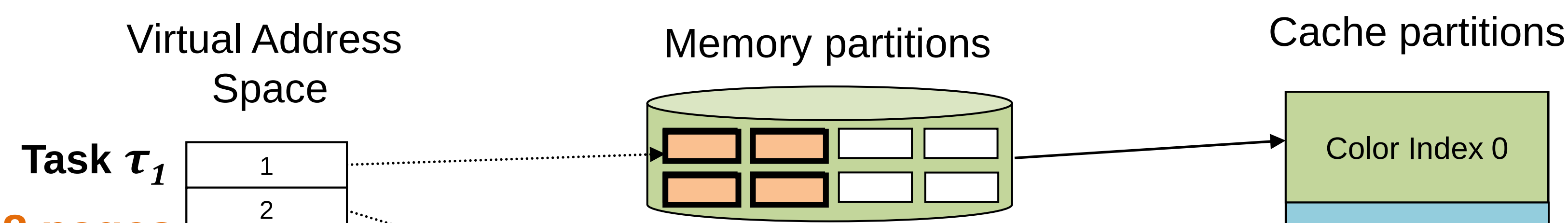
1. **Cache warm-up delay**
   - Occurs at the beginning of each period of a task
   - Caused by the executions of other tasks while the task is inactive
2. **Cache-related preemption delay**
   - Occurs when a task is preempted by a higher-priority task
   - Imposed on the preempted task



Our RT-test:
- Bounds intra-core cache interference
- Independent of specific cache analysis used
- Allows estimating WCET in isolation from others

Page Allocation for Cache Sharing

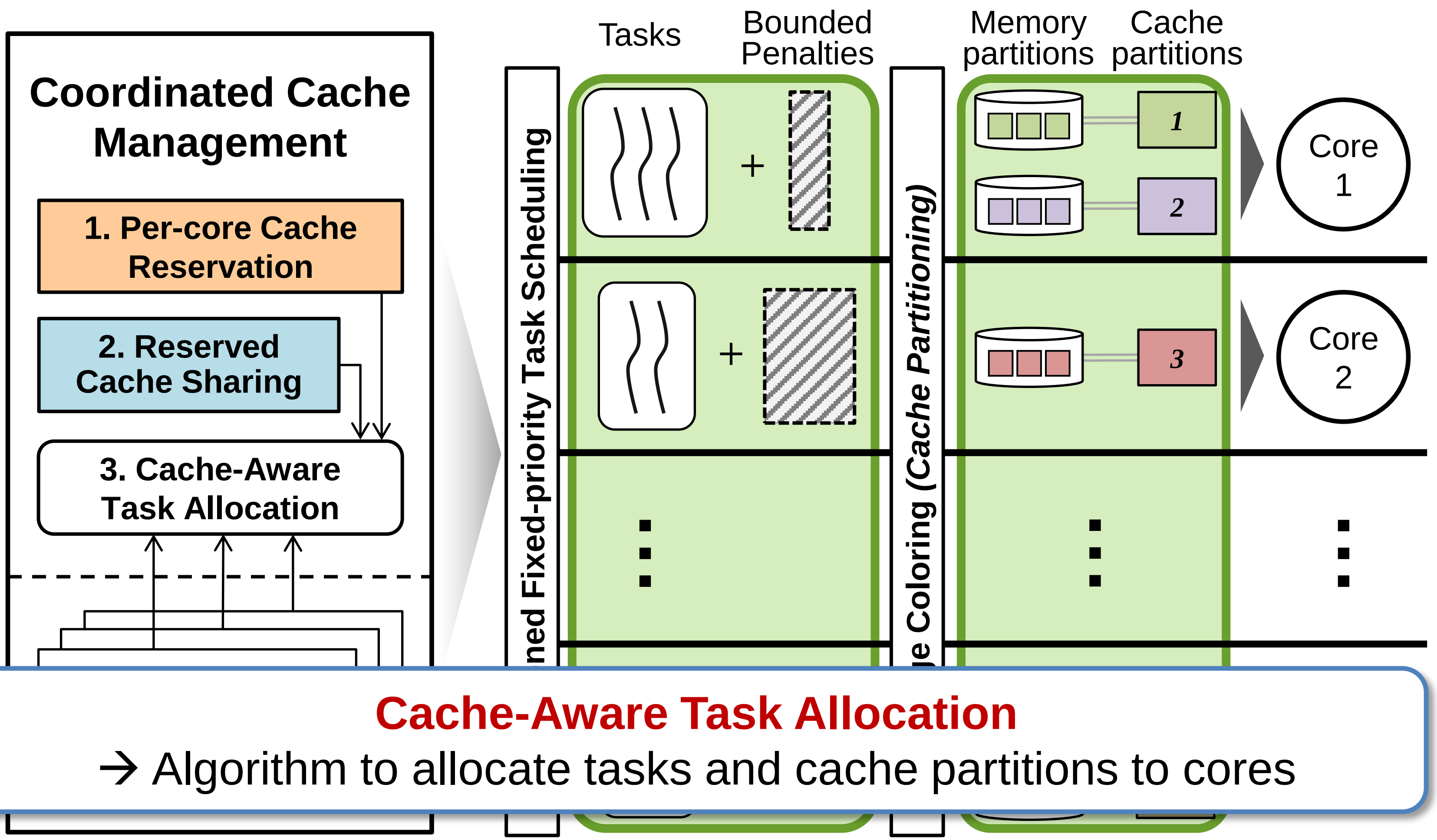
- **Sharing cache partitions = Sharing memory partitions**
  - Cache sharing can be restricted by task memory requirements
  - Depends on how pages are allocated
- **Our approach**
  - Allocate pages to a task from memory partitions in round-robin order



→ Bounds the worst-case memory usage in a memory partition

→ Developed a memory feasibility test for cache-partition sharing

Coordinated Cache Management



Cache-Aware Task Allocation

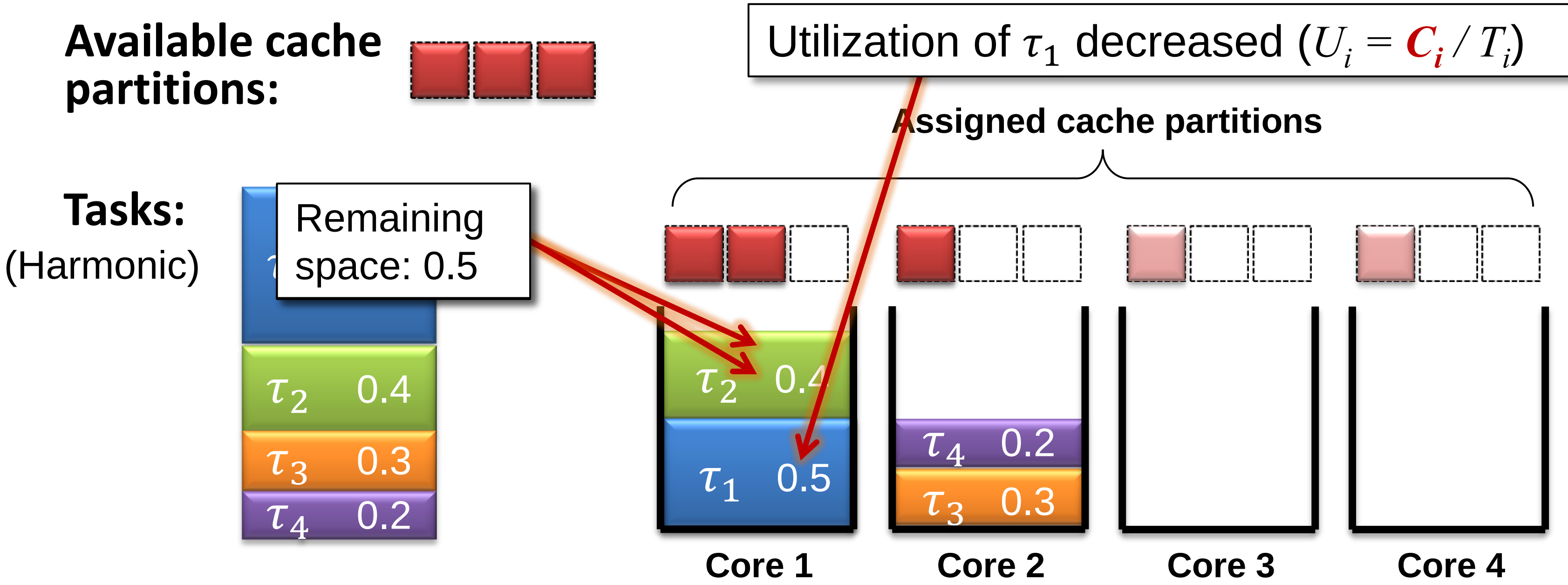
→ Algorithm to allocate tasks and cache partitions to cores

Cache-Aware Task Allocation (1/2)

- **Objectives**
  - Reduce the number of cache partitions required for a given taskset
    - Remaining cache partitions { Non-real-time tasks / Saving CPU usage
  - Exploit the benefits of cache sharing
- **Our approach**
  - Based on the BFD (best-fit decreasing) bin-packing heuristic
    - Load concentration is helpful for cache sharing
  - Gradually assign caches to cores while allocating tasks to cores
    - Use cache reservation and cache sharing during task allocation

Cache-Aware Task Allocation (2/2)

- **Step 1:** Each core is initially assigned zero cache partitions
- **Step 2**: Find a core where a task fits best
- **Step 3**: If not found, try to find the best-fit core for the task, assuming each core has 1 more cache partition than before
- **Step 4:** Once found, the best-fit core is assigned the task and the assumed cache partition(s)

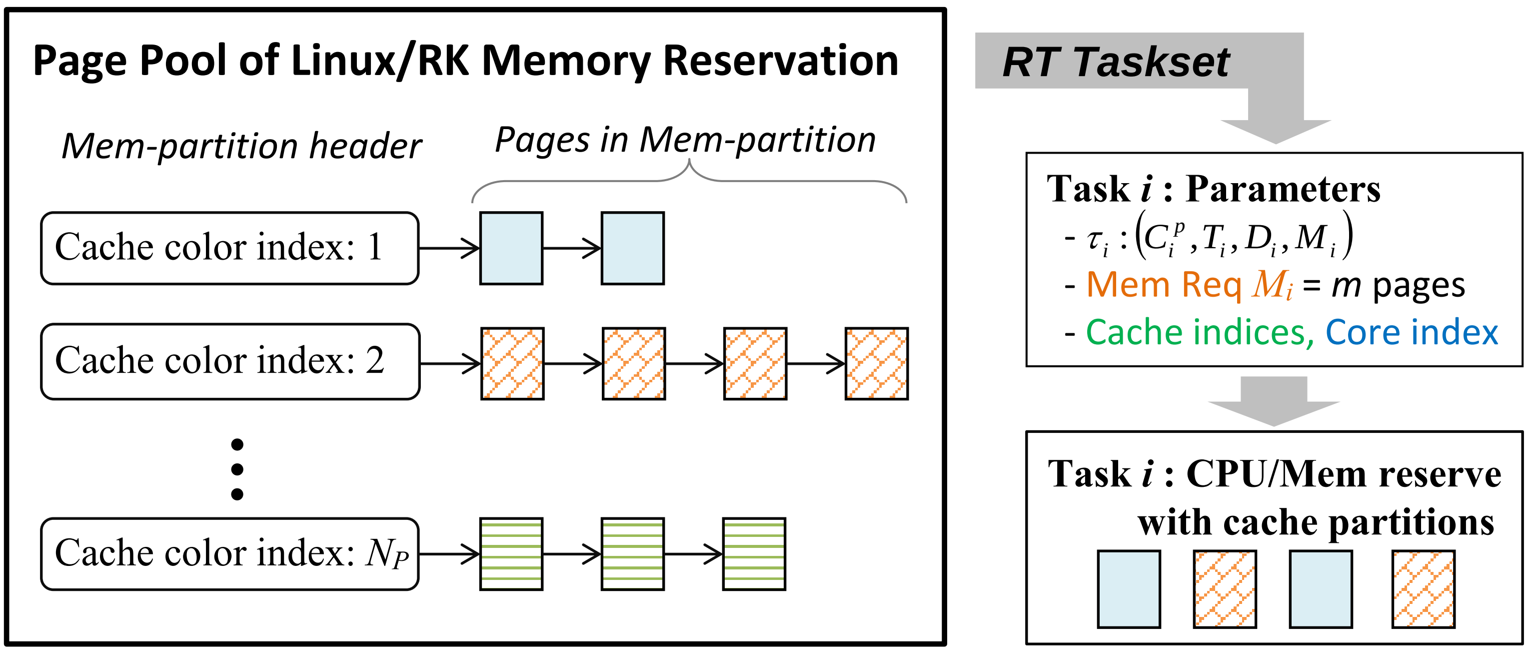


Outline

- Motivation
- Coordinated Cache Management
  - Task model
  - Per-core Cache Reservation
  - Reserved Cache Sharing
  - Cache-Aware Task Allocation
- Evaluation
- Conclusion

Implementation

- **Based on Linux/RK Memory Reservation**
  – Page pool stores unallocated physical pages
  – Classifies pages into memory partitions with their color indices



Experimental Setup

- **Target system and system parameters**
  - Implemented in **Linux/RK** (Linux 2.6)
  - Intel i7-2600 quad-core processor → $N_C = 4$ cores
  - 8 MB shared L3 cache → $N_P = 32$ cache partitions
  - Physical memory (M_{total}): 1GB → Size of a mem-partition: 32MB; 2GB → 64MB
  - Number of tasks: $n = \{8, 12, 16\}$
    - Task functions are from the PARSEC benchmarks
    - Mixture of cache-sensitive and cache-insensitive tasks
    - C_i^p and M_i for tasks are estimated ahead of time

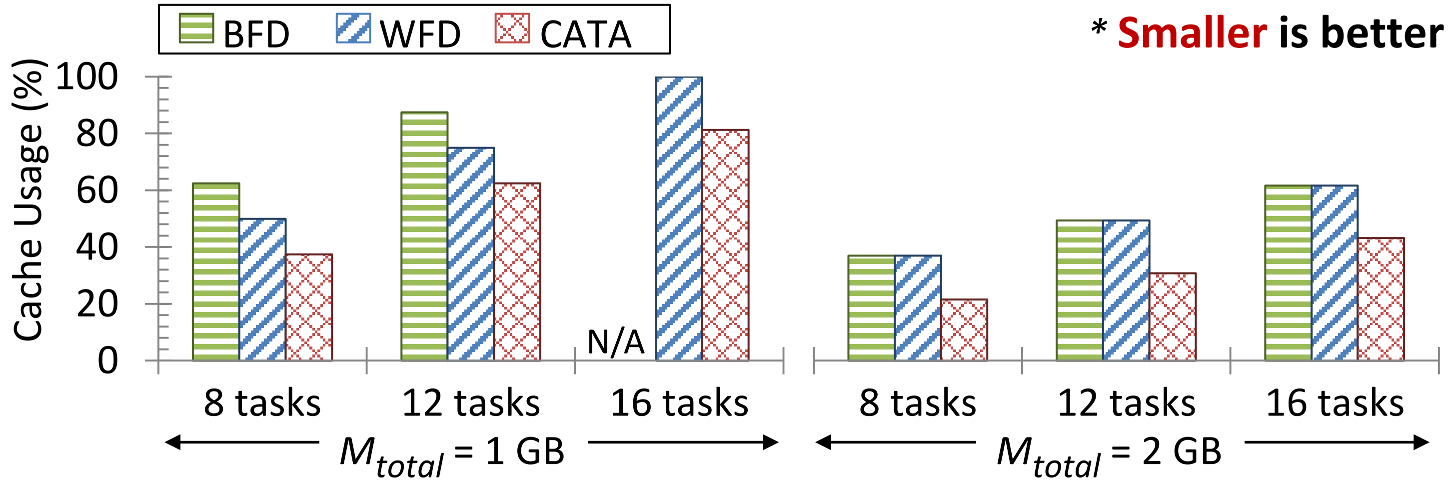
Evaluation Methodology

- **Metrics**
  1. Cache partition usage
  2. CPU utilization

- **Evaluated schemes**
  1. **BFD**: Best-Fit Decreasing + Page Coloring
  2. **WFD**: Worst-Fit Decreasing + Page Coloring
     - No cache partition sharing
  3. **CATA:** Our scheme (<u>**C**</u>ache-<u>**A**</u>ware <u>**T**</u>ask <u>**A**</u>llocation)

Cache Partition Usage

- Minimum amount of cache required to schedule given tasksets

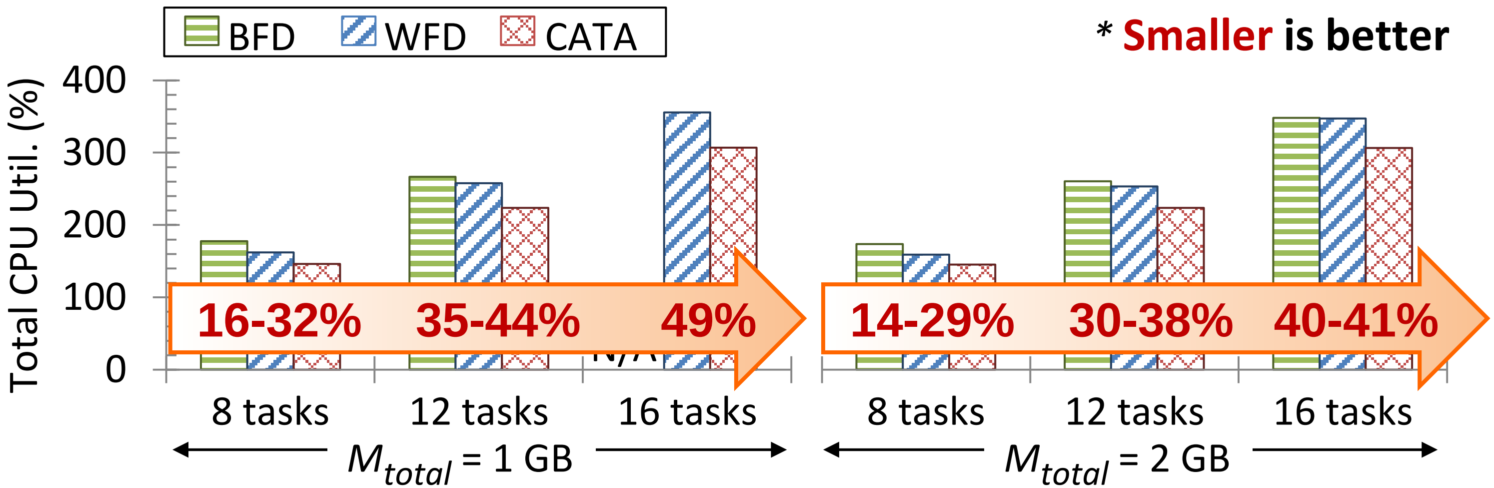


CATA requires **12-25% fewer cache partitions** than BFD and WFD

Fewer cache partitions → Fewer memory partitions
→ Mitigates the memory wastage of page coloring

CPU Utilization

- Total accumulated CPU utilization required to schedule given tasksets
  - Same number of cache partitions is used ($N_P = 32$)



CATA requires **14-49% less CPU utilization** than BFD and WFD

More number of tasks → Larger utilization benefit

→ Mitigates the limited availability of cache partitions

Our scheme
- Efficient allocation of cache partitions
- Mitigates the two problems with page coloring

Conclusions

- **Multi-core CPUs for real-time systems**
  - **Uncontrolled shared cache:** temporal interference among tasks
  - **Page coloring:** memory wastage/shortage, limited partitions
- **Coordinated OS-Level Cache Management**
  - **No** special H/W support, **No** modifications to application S/W
  - **Per-core cache reservation & Reserved cache sharing**
    - Preserves task schedulability
    - Guarantees task memory requirements
  - **Cache-aware task allocation**
    - Determines efficient task and cache allocation
    - Yields 9-18% improvement in utilization on real platforms

Linux/RK

- https://rtml.ece.cmu.edu/redmine/projects/rk/

Home Projects Help

RK

Overview Activity News Wiki Forums Repository

Overview

RK (Resource Kernel) is a real-time kernel (operating system) that provides timely, guaranteed and enforced access to system resources for applications. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

Members

Manager: Hyoseung Kim
Developer: Hyoseung Kim
Reporter: Hyoseung Kim

What is Linux/RK?

Linux/RK stands for Linux/Resource Kernel, which incorporates real-time extensions to the Linux kernel to support the abstractions of a resource kernel. Linux/RK is developed by the Real-time and Multimedia Systems Laboratory led by Prof. Raj Rajkumar at Carnegie Mellon University. Current ongoing research topics include

- multi-core reservation
- reservation for multi-core memory hierarchy (cache, DRAM banks, etc)
- reservation for parallel task model
- heterogeneous multi-core architectures
- low-power management

Carnegie Mellon University

Real-Time and Multimedia Systems Lab

- x86 (32/64bit)
- ARM (Cortex-A9)
- Global/Partitioned scheduling
- CPU/Mem reservation
- Cache/Bank coloring
- Task profiling mechanism