Demo II

- In class on **4/6** and **4/8**
- **12 min** per team
  - 10-min presentation + 2-min Q&A
- Substantial progress towards final demo.
- Submit on Canvas before the class of your presentation.
  - Slides
  - Video as backup
Parallel Real-Time Systems for Latency-Critical Applications

Chenyang Lu
CSE 520S
Cyber-Physical Systems (CPS)

Since the application interacts with the physical world, its computation must be completed **under a time constraint**.

CPS are built from, and depend upon, the seamless integration of computational algorithms and physical components. [NSF]
Parallelism Improves RTHS Accuracy

A RTHS simulates a nine stories building, with first story damper

- Previously, sequential processing power limits a rate of 575Hz
- Parallel execution now allows a rate of 3000Hz
Parallelism Improves RTHS Accuracy

A RTHS simulates a nine stories building, with first story damper

- Previously, sequential processing power limits a rate of 575Hz
- Parallel execution now allows a rate of 3000Hz

- Reduction in error for acceleration and displacement
- Parallelism increases accuracy via faster actuation and sensing
Cyber-Physical Systems (CPS)
Interactive Cloud Services (ICS)

Need to respond within 100ms for users to find responsive*.

* Jeff Dean et al. (Google) "The tail at scale." Communications of the ACM 56.2 (2013)
Interactive Cloud Services (ICS)

Need to respond **within 100ms** for users to find responsive*. E.g., web search, online gaming, stock trading etc.

* Jeff Dean et al. (Google) "The tail at scale." Communications of the ACM 56.2 (2013)
Real-Time Systems

The performance of the systems depends not only upon their functional aspects, but also upon their temporal aspects.

Real-time performance:
1) Provide hard guarantee of meeting jobs’ deadlines (e.g. CPS)
2) Optimize latency-related objectives for jobs (e.g. ICS)
New Generation of Real-Time Systems

Characteristics:
- New classes of applications with complex functionalities
- Increasing computational demand of each application
- Consolidating multiple applications onto a shared platform
- Rapid increase in the number of cores per chip

**Demand:** leverage parallelism within the applications, to improve real-time performance and system efficiency
State of the Art

- **Real-time systems**
  - Schedule multiple sequential jobs on a single core
  - Schedule multiple sequential jobs on multiple cores

- **Parallel runtime systems**
  - Schedule a single parallel job
  - Schedule multiple parallel jobs to optimize fairness or throughput

- **New**: parallel real-time systems for latency-critical applications
Challenges for Parallel Real-Time Systems

**Theory**
How to provide real-time performance for multiple parallel jobs?

**Systems**
How to build parallel real-time systems that are efficient and scalable?

**Develop provably good and practically efficient real-time systems for parallel applications**
Parallel Job – Directed Acyclic Graph (DAG)

Naturally captures programs generated by parallel languages such as Cilk Plus, Thread Building Blocks and OpenMP.

**Node**: sequential computation

**Edge**: dependence between nodes

**Work** $C_i$: execution time on one core
Parallel Job – Directed Acyclic Graph (DAG)

Naturally captures programs generated by parallel languages such as Cilk Plus, Thread Building Blocks and OpenMP.

**Node**: sequential computation

**Edge**: dependence between nodes

**Work** $C_i$: execution time on one core

**Span** (critical-path length) $L_i$: execution time on $\infty$ cores

$C_i = 18$

$L_i = 9$
Parallel Real-Time Task Model

A task periodically releases DAG jobs with deadlines.

Task 1

Job 1

Job 2

\[ D_i = 12 \]

deadline \( D_i = \) period
Parallel Real-Time Task Model

A task periodically releases DAG jobs with deadlines.

\[ D_i = 12 \]

**Task 1**

**Job 1**

**Job 2**

Deadline \( D_i = \text{period} \)

- worst-case span \( L_i \)
- worst-case work \( C_i \)
Parallel Real-Time Task Model

A task periodically releases DAG jobs with deadlines.
Multiple tasks scheduled on multi-core system.

Goal of system: guarantee all tasks can meet all their deadlines.
Federated Scheduling

For parallel tasks, FS has the best bound in term of schedulability

FS assigns $n_i$ dedicated cores to each parallel task

$$n_i = \left\lfloor \frac{C_i - L_i}{D_i - L_i} \right\rfloor$$

- deadline $D_i = \text{period}$
- worst-case span $L_i$
- worst-case work $C_i$

$n_i$ – the minimum #cores needed for a task to meet its deadline
Empirical Comparison

FS platform

- Middleware platform providing FS service in Linux
- Work with GNU OpenMP runtime system
- Run OpenMP programs with minimum modification

Compare with our Global Earliest Deadline First platform (GEDF)

- Linux kernel 3.10.5 with LITMUS$^\text{RT}$ patch
- 16-core machine with 2 Intel Xeon E5-2687W processors
- GCC version 4.6.3. with OpenMP
- Each data point has 100 task sets
- Each task is randomly generated with parallel for-loops
Empirical Comparison

Fraction of Task Sets Missing Deadlines

Better performance

• Linux kernel 3.10.5 with LITMUS\textsuperscript{RT} patch
• 16-core machine with 2 Intel Xeon E5-2687W processors
• GCC version 4.6.3. with OpenMP
• Each data point has 100 task sets
• Each task is randomly generated with parallel for-loops

\[
\text{Normalized system utilization} = \frac{\sum_{i} C_i}{D_i} \quad m: \#\text{cores}
\]

GEDF
FS

Normalized System Utilization

Harder to schedule
Empirical Comparison

- Linux kernel 3.10.5 with LITMUS\textsuperscript{RT} patch
- 16-core machine with 2 Intel Xeon E5-2687W processors
- GCC version 4.6.3. with OpenMP
- Each data point has 100 task sets
- Each task is randomly generated with parallel for-loops

52% tasks sets become schedulable under FS

Better performance
Summary of Federated Scheduling

For parallel real-time systems with guarantee of meeting deadlines, Federated Scheduling has:
- the best theoretical bound in term of schedulability
- better empirical performance compared to GEDF

RTHS has used FS platform to improve system performance

The End?
Issue with the Classic System Model

The classic system model uses the worst-case work for analysis. The worst-case work is significantly larger than the average work. → The average system utilization is very low in practice.

To guarantee that all tasks can meet all deadlines at all cases.

Most cases  Work 10ms

Very rare cases  Work 100ms
Mixed-Criticality in Cars

Features with different criticality levels:

- Safety-critical features
- Infotainment features

Display system with Car Navigation and Infotainment
**Toy Example of MC System**

**High criticality task** deadline 40ms

- Most-case work 10ms
- Most cases

- Worst-case work 100ms
- Very rare cases

**Low criticality task** deadline 40ms

- Most-case work 80ms
- core 1
- core 2
- core 3

- 100ms
Most-Case vs. Worst-Case Scenarios

Single-criticality systems:

need to model **worst-case scenario**

**Most cases**
- core 1: 10ms
- core 2: 80ms
- core 3
- core 4
- core 5

**Very rare cases**
- core 1
- core 2
- core 3
- core 4
- core 5

- 100ms
- 80ms
MC Model Improves Resource Efficiency

Mixed-criticality system:
Provide different levels of real-time guarantees

Most cases: guarantee that both high and low-criticality tasks meet deadlines

Very rare cases: only guarantee that high-criticality tasks meet deadlines

Graph showing:
- Core 1: 10ms
- Core 2: 80ms
- Core 3: 100ms

Legend:
- Green: overrun
- Blue: guaranteed time
- Red: not guaranteed time
MCFS Algorithm at a High Level

For each parallel task, calculate and assign:

(1) dedicated cores in typical-state

\[ m \text{ cores} \]

\[
\text{Typical-state (most cases)}
\]

\[
\text{High-Criticality}
\]

\[
\text{Low-Criticality}
\]

\[
\text{High-Criticality}
\]
MCFS Algorithm at a High Level

For each parallel task, calculate and assign:

(1) dedicated cores in typical-state
(2) dedicated cores in critical-state
MCFS Algorithm at a High Level

For each parallel task, calculate and assign:

(0) virtual deadline
(1) dedicated cores in typical state
(2) dedicated cores in critical state

If a job has not completed by its virtual deadline, it transitions to critical-state.

Virtual deadline

Typical-state (most cases)  Critical-state (rare case)

High-Criticality

Low-Criticality

High-Criticality

$m$ cores
MCFS Algorithm at a High Level

MCFS jointly assigns virtual deadlines and cores to maximize utilization while guaranteeing task deadlines.

1. Dedicated cores in typical state
2. Dedicated cores in critical state

If a job has not completed by its virtual deadline, it transitions to critical-state.

$m$ cores
MCFS Implementation

In typical-state, MCFS assigns dedicated cores to all tasks.

High-Criticality

OpenMP Runtime

Low-Criticality

OpenMP Runtime

High-Criticality

OpenMP Runtime

Linux
MCFS Implementation

In critical-state, MCFS increases cores assigned to high-crit. tasks.
MCFS Implementation

Put additional HC threads to sleep on higher priority cores.
Empirical Evaluations

Fraction of tasks with no deadline miss (per criticality)

- Linux with RT_PREEMPT patch version 4.1.7-rt8
- 16-core machine with 2 Intel Xeon E5-2687W processors
- GCC version 4.6.3. with OpenMP
- Each data point has 100 task sets
- Each task is randomly generated with parallel for-loops
Empirical Evaluations

Fraction of tasks with no deadline miss (per criticality)

- Linux with RT_PREEMPT patch version 4.1.7-rt8
- 16-core machine with 2 Intel Xeon E5-2687W processors
- GCC version 4.6.3. with OpenMP
- Each data point has 100 task sets
- Each task is randomly generated with parallel for-loops
**Issue with the Analysis of Parallel Jobs**

Centralized greedy scheduler

- Threads get work (nodes) from a *centralized* queue

Bottleneck for scalability of large scale systems

Implicit assumption of parallel real-time scheduling theory: when a thread (core) is allowed to work on a job, it must be able to find the available nodes *immediately* (within bounded time)
Issue with the Analysis of Parallel Jobs

Centralized greedy scheduler
- Threads get work (nodes) from a centralized queue

Randomized work-stealing
- Threads usually get work \textit{locally};
- If local queue is empty, it \textit{steals randomly} from another queue

<table>
<thead>
<tr>
<th>Predictable</th>
<th>Bounded worst-case</th>
<th>Unbounded worst-case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable</td>
<td>Does not scale well</td>
<td>Good scalability</td>
</tr>
</tbody>
</table>
Empirical Comparisons

Randomized work-stealing for *large-scale soft* real-time system?

- FS Implementations (with scheduling overheads incorporated):
  - FSCG with centralized greedy scheduler in GNU OpenMP
  - FSWS with randomized work-stealing in GNU Cilk Plus

- Linux with RT_PREEMPT patch version r14
- 32-core machine with 4 Intel Xeon E5-4620 processors
- GCC 5.1 with OpenMP, Cilk Plus
- Each data point is one task set
- Each task is randomly generated using benchmark program Heat
Empirical Comparisons

Randomized work-stealing for *large-scale soft* real-time system?

- **Better performance**

FSCG and FSWS
- Same computation
- Same resources
- Only difference: *internal scheduling* of parallel tasks
Empirical Comparisons

Randomized work-stealing for *large-scale soft* real-time system?

- Better performance
- FSCG and FSWS
  - Same computation
  - Same resources
  - Only difference: *internal scheduling* of parallel tasks

<table>
<thead>
<tr>
<th>Percentage of Utilization</th>
<th>Deadline Miss Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>0.4</td>
<td>0</td>
</tr>
<tr>
<td>0.6</td>
<td>0</td>
</tr>
<tr>
<td>0.8</td>
<td>0</td>
</tr>
</tbody>
</table>

- RTWS
- RTCG
Empirical Comparisons

Randomized work-stealing for large-scale soft real-time system?

The benefit of scalability in work-stealing dominates the increased variation in parallel execution times.

FSCG and FSWS
- Same computation
- Same resources
- Only difference: *internal scheduling* of parallel tasks
Outline

- Contributions

- System Guaranteed to Meet Deadlines for Parallel Jobs in CPS

- System Optimized to Meet Target Latency for ICS

- Future Work
System for Interactive Cloud Services

**Online system:** do not know when jobs arrive

**Objective:** optimize latency-related objectives for the service
e.g., average latency, max latency
System for Interactive Cloud Services

**Online system:** do not know when jobs arrive

**Objective:**

maximize the number of jobs that meet a target latency $T$
Workload Distribution Has a Long Tail

- Large jobs must run in parallel to meet target latency
- Always run large jobs in full parallelism?

Bing search workload

Target latency

Job Sequential Execution Time (ms) (work)
Parallelize Large Jobs According to Load

**Tail-Control Strategy:** when load is low, run all jobs in parallel; when load is high, run large jobs sequentially.

Latency = Processing Time + Waiting time

**At low load:**
processing time dominates latency

**At high load:**
waiting time dominates latency

---

Miss 0 request

Miss 1 request
The Inner Workings of Tail-Control

We implement tail-control algorithm in the runtime system of Intel Thread Building Block and evaluate on Bing search workload.
The Inner Workings of Tail-Control

We implement tail-control algorithm in the runtime system of Intel Thread Building Block and evaluate on Bing search workload.

![Graph showing the comparison between default work-stealing and tail-control. The x-axis represents current time in seconds, and the y-axis represents request latency in milliseconds. The graph includes data points for both methods, with a target latency line marked in purple at the 30ms mark.]
The Inner Workings of Tail-Control

We implement tail-control algorithm in the runtime system of Intel Thread Building Block and evaluate on Bing search workload.

![Diagram showing request latency over time with target latency highlighted.](image-url)
We implement tail-control algorithm in the runtime system of Intel Thread Building Block and evaluate on Bing search workload.
Conclusion

Exploit the untapped efficiency in parallel computing platforms and drastically improve the real-time performance of applications.

- **System Guaranteed to Meet Deadlines for CPS**
  - Develop provably good schedulers for parallel applications
  - Incorporate real-time scheduling into parallel runtime system
  - Improve system efficiency by dealing with uncertainty in jobs
  - Address system scalability issue due to internal scheduling

- **System Optimized to Meet Target Latency for ICS**
  - Design and implement strategy to optimize real-time performance
References


