Parallel Real-Time Systems
for Latency-Critical Applications

Chenyang Lu
CSE 520S

Washington University in St. Louis
Cyber-Physical Systems (CPS)

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

NSF Cyber-Physical Systems Program Solicitation:
CPS are built from, and depend upon, the seamless integration of computational algorithms and physical components.

^ Robert L. and Terry L. Bowen Large Scale Structures Laboratory at Purdue University
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
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
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

\[ C_i = 18 \]
\[ L_i = 9 \]

**Work** \( C_i \): execution time on one core

**Span** (critical-path length) \( L_i \): execution time on \( \infty \) cores
Parallel Real-Time Task Model

A task periodically releases DAG jobs with deadlines.

deadline $D_i = \text{period}$
Parallel Real-Time Task Model

A task periodically releases DAG jobs with deadlines.

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$^{RT}$ 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$^{RT}$ 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

Normalized System Utilization

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

GEDF
FS

Fraction of Task Sets

Normalized System Utilization

Harder to schedule
Empirical Comparison

- Linux kernel 3.10.5 with LITMUS$^{RT}$ 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

![Graph showing fraction of task sets missing deadlines vs normalized system utilization.](image)

- Better performance
- 52% tasks sets become schedulable under FS
- Harder to schedule
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**.
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

0 10ms 40

0 80ms 40

0 100ms 40
Most-Case vs. Worst-Case Scenarios

Single-criticality systems:

need to model **worst-case scenario**

Most cases

<table>
<thead>
<tr>
<th>Core</th>
<th>Most Cases</th>
<th>Very Rare Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1</td>
<td>10ms</td>
<td>Core 1</td>
</tr>
<tr>
<td>Core 2</td>
<td>80ms</td>
<td>Core 2</td>
</tr>
<tr>
<td>Core 3</td>
<td></td>
<td>Core 3</td>
</tr>
<tr>
<td>Core 4</td>
<td></td>
<td>Core 4</td>
</tr>
<tr>
<td>Core 5</td>
<td></td>
<td>Core 5</td>
</tr>
</tbody>
</table>
MC Model Improves Resource Efficiency

**Mixed-criticality system:**

- Provide different levels of real-time guarantees

**Core 1:**
- 10ms

**Core 2:**
- 80ms

**Core 3:**

**Overrun**
- 100ms

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

**Very rare cases:**
- Only guarantee that **high-criticality tasks** meet deadlines
MCFS Algorithm at a High Level

For each parallel task, calculate and assign:

(1) dedicated cores in typical-state
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.

Virtual deadline

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

High-Criticality

Low-Criticality

High-Criticality

$m$ cores
MCFS Implementation

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

High-Criticality

Low-Criticality

OpenMP Runtime

OpenMP Runtime

High-Criticality

OpenMP Runtime

HC thread

LC thread

cores
MCFS Implementation

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

High-Criticality

Low-Criticality

High-Criticality

OpenMP Runtime

OpenMP Runtime

OpenMP Runtime

HC thread

cores

more HC thread
MCFS Implementation

Put additional HC threads to sleep on higher priority

High-Criticality

Low-Criticality

High-Criticality

OpenMP Runtime

OpenMP Runtime

OpenMP Runtime

MCFS

Linux

HC thread

Sleeping HC thread

LC thread
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 \textit{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 locally;
- If local queue is empty, it steals randomly from another queue

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

Graph:
- X-axis: Normalized System Utilization
- Y-axis: Deadline Miss Ratio
- Two lines: FSWS and FSCG
- Higher load

39
Empirical Comparisons

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

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?

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

Job Sequential Execution Time (ms) (work)

Target latency
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

\[ \text{time} \]

\[ \text{target} \]
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.
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
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.

≥100

- ○ default work-stealing
- • tail-control

Target Latency
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.

Target Latency ≥ 100

- default work-stealing
- tail-control

![Graph showing request latency vs. current time in seconds with target latency marker]
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

- J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill and C. Lu, Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks, RTAS’16. Outstanding Paper Award