Timely Fine-grained Interference-sensitive Run-time Adaptation of Time-triggered Schedules

Stefanos Skalistis, Angeliki Kritikakou

December 2019
Time-critical systems

Embedded systems:
Time-critical systems

Embeddedsystems:
• perform a set of tasks
• within certain amount of time (deadlines).

<table>
<thead>
<tr>
<th>Task n</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 sec</td>
</tr>
<tr>
<td></td>
<td>7 sec</td>
</tr>
<tr>
<td></td>
<td>3 sec</td>
</tr>
</tbody>
</table>
Time-critical systems

Embedded systems:
- perform a set of tasks
- within certain amount of time (deadlines).

Typical Application Domains:
- Avionics (Fly-by-wire)
- Automotive (Airbag)
- Medical (X-Ray)
- Power Production (Nuclear)
Time-critical systems

Embedded systems:
• perform a set of tasks
• within certain amount of time (deadlines).

Typical Application Domains:
• Avionics (Fly-by-wire)
• Automotive (Airbag)
• Medical (X-Ray)
• Power Production (Nuclear)

Deadlines

<table>
<thead>
<tr>
<th>Task n</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 sec</td>
</tr>
<tr>
<td></td>
<td>7 sec</td>
</tr>
<tr>
<td></td>
<td>3 sec</td>
</tr>
</tbody>
</table>

… enormous impact on everyday life and society!
Motivation

Scalability via parallelization:
- Cloud applications
- Mobile applications

Time-critical systems:
- Time guarantees rely on Worst Case Execution Time (WCET)
Motivation

Scalability via parallelization:
• Cloud applications
• Mobile applications

Time-critical systems:
• Time guarantees rely on Worst Case Execution Time (WCET)
• Timing Interferences due to arbitrated resource sharing

“One-out-of-m processor” problem:
#Cores increase ⇒ Sequential schedules better than parallel
Timing interference

Interference affects Worst-Case Execution Time (WCET):

• Task deadline
• Latency guarantees
Timing interference

Interference affects Worst-Case Execution Time (WCET):

- Task deadline
- Latency guarantees

\[ WCET = WCET_{iso} + \text{Worst-Case Interference} \]
Timing interference

Interference affects Worst-Case Execution Time (WCET):

- Task deadline
- Latency guarantees

WCET = WCET_{iso} + Worst-Case Interference

Example:

- Four identical tasks
- Interference delay (1 time unit)

<table>
<thead>
<tr>
<th></th>
<th>WCET_{iso}</th>
<th>WCMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Timing interference

Interference affects Worst-Case Execution Time (WCET):

• Task deadline
• Latency guarantees

\[ \text{WCET} = \text{WCET}_{iso} + \text{Worst-Case Interference} \]

Example:

• Four identical tasks
• Interference delay (1 time unit)

<table>
<thead>
<tr>
<th>WCET_{iso}</th>
<th>WCMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

29/04/2020
Timing interference

Interference affects Worst-Case Execution Time (WCET):

- Task deadline
- Latency guarantees

WCET = WCET_{iso} + Worst-Case Interference

Example:

- Four identical tasks
- Interference delay (1 time unit)
Timing interference

Interference affects Worst-Case Execution Time (WCET):

• Task deadline
• Latency guarantees

\[ \text{WCET} = \text{WCET}_{iso} + \text{Worst-Case Interference} \]

Example:

• Four identical tasks
• Interference delay (1 time unit)

\[
\begin{array}{c|c|c}
\text{Core1} & \text{Core2} & \text{Core3} \\
\hline
\text{\textcolor{brown}{v_1}} & \text{\textcolor{brown}{v_3}} & \text{\textcolor{brown}{v_2}} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>\text{WCET}_{iso}</th>
<th>\text{WCMA}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Interference affects Worst-Case Execution Time (WCET):

- Task deadline
- Latency guarantees

\[ \text{WCET} = \text{WCET}_{iso} + \text{Worst-Case Interference} \]

Example:

- Four identical tasks
- Interference delay (1 time unit)

<table>
<thead>
<tr>
<th>WCET_{iso}</th>
<th>WCMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Timing interference

Interference affects Worst-Case Execution Time (WCET):

- Task deadline
- Latency guarantees

\[ \text{WCET} = \text{WCET}_{iso} + \text{Worst-Case Interference} \]

Example:

- Four identical tasks
- Interference delay (1 time unit)

WCET can grow up to 7x WCET\(_{iso}\)
Interference-sensitive approaches

Reduce impact of interference:

\[ isWCET = WCET_{iso} + \text{Interference}(...) \]
Interference-sensitive approaches

Reduce impact of interference:

\[ i_{\text{is}} \text{WCET} = \text{WCET}_{\text{iso}} + \text{Interference}(\ldots) \]

Contention-free approaches:

• Rely on spatio-temporal isolation to avoid interferences
• *E.g.* Resource partitioning, PREM / AER-based
Interference-sensitive approaches

Reduce impact of interference:

\( isWCET = WCET_{iso} + \text{Interference}(\ldots) \)

Contention-free approaches:

- Rely on spatio-temporal isolation to avoid interferences
- *E.g.* Resource partitioning, PREM / AER-based

Context-based approaches:

- Rely on knowledge of upper-bound of interference
  - A-priori limit of interference
  - Calculation of interference (Schedule, Co-runners, Task-overlapping)
- *E.g.* Budget-based, Analytical approaches

Such *isWCET are valid only* on the given context!
Interference-sensitive approaches

Reduce impact of interference:

\[ isWCET = WCET_{iso} + \text{Interference(…)} \]

Contention-free approaches:

- Rely on spatio-temporal isolation to avoid interferences
- *E.g.* Resource partitioning, PREM / AER-based

Context-based approaches:

- Rely on knowledge of upper-bound of interference
  - A-priori limit of interference
  - Calculation of interference (Schedule, Co-runners, Task-overlapping)
- *E.g.* Budget-based, Analytical approaches

*Interference-sensitive rely on time-triggered execution*
Time-triggered (TT) Execution Model

Tasks: fixed start time
- Tasks are executed precisely at that time
Time-triggered (TT) Execution Model

Tasks: *fixed start time*
- Tasks are executed precisely at that time

Scheduler:
- Simply waits for the time to arrive
Time-triggered (TT) Execution Model

Tasks: **fixed start time**
- Tasks are executed precisely at that time

Scheduler:
- Simply waits for the time to arrive

Very easy to implement:
- Just Lookup table

<table>
<thead>
<tr>
<th>Core</th>
<th>Task</th>
<th>Time</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>κ₀</td>
<td>τ₀</td>
<td>2</td>
<td>τ₂</td>
</tr>
<tr>
<td>κ₁</td>
<td>τ₁</td>
<td>0</td>
<td>τ₅</td>
</tr>
<tr>
<td>κ₀</td>
<td>τ₂</td>
<td>4</td>
<td>τ₄</td>
</tr>
<tr>
<td>κ₁</td>
<td>τ₃</td>
<td>6</td>
<td>τ₁</td>
</tr>
<tr>
<td>κ₀</td>
<td>τ₄</td>
<td>7</td>
<td>τ₀</td>
</tr>
<tr>
<td>κ₁</td>
<td>τ₅</td>
<td>3</td>
<td>τ₃</td>
</tr>
</tbody>
</table>

29/04/2020
isWCET will (almost) never happen

- Tasks rarely execute with WCET_{iso}
- Interference is not increased
isWCET will (almost) never happen

- Tasks rarely execute with WCET$_{iso}$
- Interference is not increased
isWCET will (almost) never happen

- Tasks rarely execute with WCET_{iso}
- Interference is not increased

Execution gaps ⇒ performance loss
isWCET will (almost) never happen

- Tasks rarely execute with WCET$_{iso}$
- Interference is not increased

**Execution gaps ⇒ performance loss**

- Execute less critical / sporadic tasks
- Increase QoS
- Increase Fault-tolerance
**isWCET will (almost) never happen**

- Tasks rarely execute with WCET$_{iso}$
- Interference is not increased

**Execution gaps ⇒ performance loss**

- Execute less critical / sporadic tasks
- Increase QoS
- Increase Fault-tolerance

*Need for interference-sensitive run-time adaptation*
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:
  • Safely, without introducing timing-anomalies
  • Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique (isRA)
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique (isRA)
Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique (isRA)

- **Key Idea (safety):** preserve the partial order of tasks

![Diagram showing a timeline with tasks on Core 0 and Core 1, and arrows indicating the partial order of tasks over time.](image-url)
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique \((isRA)\)

- **Key Idea (safety):** preserve the partial order of tasks
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique (isRA)

- **Key Idea (safety)**: preserve the partial order of tasks
Interference Sensitive Run-time Adaptation

Given a TT-schedule how can we adapt:

- Safely, without introducing timing-anomalies
- Minimal overhead in terms of controller’s WCET and actual performance

Distributed adaptation technique (isRA)

- **Key Idea (safety):** preserve the partial order of tasks
- **Result:** no additional interference can be introduced
Encoding of scheduling dependencies

Distributed adaptation technique

- **Key Idea (overhead):** encode timing dependencies in bit vectors
- **Result:** minimal and tightly bounded checking time
Encoding of scheduling dependencies

Distributed adaptation technique

- **Key Idea (overhead):** encode timing dependencies in bit vectors
- **Result:** minimal and tightly bounded checking time
Encoding of scheduling dependencies

Distributed adaptation technique

• **Key Idea (overhead):** encode timing dependencies in bit vectors
• **Result:** minimal and tightly bounded checking time

![Diagram showing notification and ready vectors with time stamps and bit vectors]

- Core 1
  - \( \tau_1 \)
  - \( \tau_5 \)
  - \( \tau_3 \)

- Core 0
  - \( \tau_0 \)
  - \( \tau_2 \)
  - \( \tau_4 \)

Time stamps: 0, 1, 2, 3, 4, 5, 6, 7

Bit vectors:
- \( \{01\} \)
- \( \{11\} \)
- \( \{11\} \)
- \( \{11\} \)
Encoding of scheduling dependencies

Distributed adaptation technique

- **Key Idea (overhead):** encode timing dependencies in bit vectors
- **Result:** minimal and tightly bounded checking time

![Diagram showing scheduling dependencies and vectors](image-url)
**Key Idea:** Reschedule as-soon-as-possible when tasks are ready

**Three phases:**

- **Ready Phase:** Continuously test if task is ready, according to status bit vector
- **Execute Phase:** Execute the task
- **Notify Phase:** Notify other cores (through status) when tasks finish, according to the notify vector.
Execution starts:

- Core 0 executes $\tau_0$
- Core 1 executes $\tau_1$
Execution starts:
- Core 0 executes $\tau_0$
- Core 1 executes $\tau_1$

Core 0 finishes:
- Core 0 notifies respective cores according to the notification vector – i.e. itself (01)
**isRA Example**

**Execution starts:**
- Core 0 executes $\tau_0$
- Core 1 executes $\tau_1$

**Core 0 finishes:**
- Core 0 notifies respective cores according to the notification vector – i.e. itself (01)
- Core 0 tests if $\tau_2$ is ready, by comparing its status with the ready mask.
Execution starts:
- Core 0 executes $\tau_0$
- Core 1 executes $\tau_1$

Core 0 finishes:
- Core 0 notifies respective cores according to the notification vector – i.e. itself (01)
- Core 0 tests if $\tau_2$ is ready, by comparing its status with the ready mask.
**isRA Example**

**Execution starts:**
- Core 0 executes \( \tau_0 \)
- Core 1 executes \( \tau_1 \)

**Core 0 finishes:**
- Core 0 notifies respective cores according to the notification vector – i.e. itself (01)
- Core 0 tests if \( \tau_2 \) is ready, by comparing its status with the ready mask.
isRA Example

Core 1
status_1: 00

Core 0
status_0: 01

τ_0 → τ_1 → τ_2 → τ_3

{00} (11){10} (00)
{00}{01} {11} (00)

{00} {10}

time
Core 1 finishes:

- Core 1 notifies respective cores according to the notification vector – i.e. both cores (11)
- Core 0 tests if $\tau_2$ is ready
- Core 1 tests if $\tau_3$ is ready
Core 1 finishes:

- Core 1 notifies respective cores according to the notification vector – i.e. both cores (11)
- Core 0 tests if $\tau_2$ is ready
- Core 1 tests if $\tau_3$ is ready
Both cores are ready:

Core 1
status_1: 10

Core 0
status_0: 11

\[
\begin{array}{c}
\tau_0 \\
\tau_1 \\
\tau_2 \\
\tau_3
\end{array}
\]

\[
\begin{array}{c}
\{00\} \\
\{00\}(01) \\
\{11\} \\
\{10\} \\
\{00\}
\end{array}
\]
Both cores are ready:

- Reset the bits of their status according to the ready mask
- Execute their next tasks
Both cores are ready:

- Reset the bits of their status according to the ready mask
- Execute their next tasks

Tasks $\tau_1$ and $\tau_2$ did not overlap!
isRA Example

Both cores are ready:

- Reset the bits of their status according to the ready mask
- Execute their next tasks

Tasks $\tau_1$ and $\tau_2$ did not overlap!

Thus guaranteeing no additional interference
Concurrency

Cores writes to other cores statuses:
• Use of protection mechanism during writes
• E.g. semaphores, mutexes, locks, etc.

Two flavours of isRA:
• isRA-GLO (IEEE D&T): uses a single protection mechanism
  • Critical sections are serialized ⇒ performance penalty
  • Easier Response Time Analysis
• isRA-FG (RTSS): uses multiple protection mechanism
  • Critical sections are parallelized ⇒ better performance
  • Harder Response Time Analysis
Theoretical Results

- **Parametric WCET for the phases**
  - In terms of incoming / outgoing edges of task

- **Including these cost up front is safe**
  - Any scheduling algorithm can be used (as long as deadlines are met)

- *isRA-FG is free from timing anomalies*
Experimental results

- **Target architecture**: 8-core DSP TMS320C6678
- **Benchmarks**: 3 applications from StreamIT, 18 different configurations
  - 1x2, 1x4, 1x8: Single instance parallelized in 2, 4, 8 cores
  - 2x2, 4x4, 8x8: 2, 4, 8 parallel instances (each one sequential)
- **WCET acquisition**: measurement-based with –O0 and without data-caches
- **Data-placement**: controller-data in MSM, app. data in main memory
- **Evaluation criteria**:
  - Scalability in the number of cores
  - Comparison of parallel execution vs multiple sequential
Results: Single parallelized instance
Results: Single parallelized instance

isRA-FG improves TT performance by >50%
Results: *isRA-FG* vs *isRA-GLO*
Results: *isRA-FG vs isRA-GLO*

*isRA-FG outperforms isRA-GLO*
Results: *Single vs Multiple instances*

![Graph showing makespan for FFT, DCT, and MERGE operations with single and multiple instances.](image-url)

- **isRA-FG**
- **isRA-GLO**
- **isTT**

- **8x8**
- **8*(1x8)**

- **FFT**
- **DCT**
- **MERGE**
Results: *Single vs Multiple instances*

Multiple instances is better
Thank You

Questions?