Introduction to Multiprocessor Real-Time Scheduling

Foundations of Cyber-Physical Systems

Björn Brandenburg & Rupak Majumdar
# Three Kinds of Multiprocessors

<table>
<thead>
<tr>
<th></th>
<th>Proc. 1</th>
<th>Proc. 2</th>
<th>Proc. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Identical</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 GHz</td>
<td>2 GHz</td>
<td>2 GHz</td>
</tr>
<tr>
<td></td>
<td>FPU</td>
<td>FPU</td>
<td>FPU</td>
</tr>
<tr>
<td><strong>Uniform Heterogeneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 GHz</td>
<td>1 GHz</td>
<td>500 MHz</td>
</tr>
<tr>
<td></td>
<td>FPU</td>
<td>FPU</td>
<td>FPU</td>
</tr>
<tr>
<td><strong>Unrelated Heterogeneous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 GHz</td>
<td>3 GHz</td>
<td>500 MHz</td>
</tr>
<tr>
<td></td>
<td>FPU</td>
<td>large cache</td>
<td>I/O coproc.</td>
</tr>
</tbody>
</table>

**identical:**
- all processors have equal **speed** and **capabilities**

**uniform heterogeneous (or homogenous):**
- all processors have equal **capabilities**
- but **different speeds**

**unrelated heterogeneous:**
- no regular relation assumed
- tasks may not be able to execute on all processors
What makes multiprocessor scheduling hard?

“Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.” [emphasis added]

Scheduling Approaches

Partitioned Scheduling
- task **statically** assigned to cores
- One ready queue **per core**
- uniprocessor scheduler on each core

Global Scheduling
- jobs **migrate** freely
- All cores serve **shared** ready queue
- requires new schedulability analysis
Global Scheduling — Dhall Effect

Uniprocessor Utilization Bounds

- EDF = 1
- Rate-Monotonic (RM) = $\ln 2$

Question: What are the utilization bounds on a multiprocessor?

- Notation: $m$ is the number of processors
- Intuition: would like to fully utilize all processors!

Guesses?

- Global EDF = ?
- Global RM = ?

Dhall Effect — Illustration

A Difficult Task Set

- **m + 1** tasks

- First **m** tasks — (\(T_i\) for \(1 \leq i \leq m\)):
  - Period = 1
  - WCET: \(2\varepsilon\)

- Last task \(T_{m+1}\):
  - Period = \(1 + \varepsilon\)
  - WCET = 1

Total utilization?
Dhall Effect — Implications

Utilization Bounds
- For $\varepsilon \rightarrow 0$, the utilization bound approaches 1.
- Adding processors makes no difference!

Global vs. Partitioned Scheduling
- Partitioned scheduling is easier to implement.
- Dhall Effect shows limitation of global EDF and RM scheduling.
- Researchers lost interest in global scheduling for ~25 years.

Since late 1990ies...
- It’s a limitation of EDF and RM, not global scheduling in general.
- In last decade, much work on global scheduling.
Partitioned Scheduling

Reduction to $m$ uniprocessor problems

- Assign each task \textit{statically} to one processor
- Use uniprocessor scheduler on each core
  - Either fixed-priority (P-FP) scheduling or EDF (P-EDF)

Find task mapping such that

- No processor is \textit{over-utilized}
- Each partition is \textit{schedulable}
  - \textit{trivial for implicit deadlines} & EDF
Connection to Bin Packing

**Bin packing decision problem**

Given a number of bins $B$, a bin capacity $V$, and a set of $n$ items $x_1, \ldots, x_n$ with sizes $a_1, \ldots, a_n$, does there exist a packing of $x_1, \ldots, x_n$ that fits into $B$ bins?

**Bin packing optimization problem**

Given a bin capacity $V$ and a set of $n$ items $x_1, \ldots, x_n$ with sizes $a_1, \ldots, a_n$, assign each item to a bin such that the number of bins is minimized.
Bin-Packing Reduction

**Bin packing decision problem**

*Given a number of bins* \( B \), *a bin capacity* \( V \), *and a set of* \( n \) *items* \( x_1, \ldots, x_n \) *with sizes* \( a_1, \ldots, a_n \), *does there exist a packing of* \( x_1, \ldots, x_n \) *that fits into* \( B \) *bins?*

1) Normalize sizes \( a_1, \ldots, a_n \) *and capacity* \( V \)
   - assume *unit-speed* processors

2) Create an implicit-deadline sporadic task \( T_i \) *for each item* \( x_i \)
   - with utilization \( u_i = a_i / V \)
   - Pick period arbitrarily, scale WCET appropriately

3) Is the resulting task set *feasible* under *P-EDF* on *B* processors?
   - Hence, finding a valid partitioning is NP-hard.
Upper Utilization Bound

**Theorem**: there exist task sets with utilizations arbitrarily close to \((m+1)/2\) that cannot be partitioned.


A difficult-to-partition task set

- **m + 1** tasks

- For each \(T_i\) for \(1 \leq i \leq m + 1\):
  - Period = 2
  - WCET: \(1 + \varepsilon\)
  - Utilization: \((1 + \varepsilon) / 2\)

Partitioning not possible

- Any two tasks together over-utilize a single processor by \(\varepsilon\)!
- Total utilization = \((m + 1) \cdot (1 + \varepsilon) / 2\)
Partitioning in Practice (I)

*Empirical approach*

Heuristics are *cheap*, just try to partition and see how far we get…
Partitioning in Practice (I)

binpacking heuristics comparison (P-EDF), using Emberson et al. (2010) tasks for m=16, periods=logunif, tasks-per-core=3, and tasks=48

Bottom line: heuristics work well most of the time (for independent tasks).
Partitioning in Practice (II)

difficulty of binpacking (P-EDF), using Emberson et al. (2010) tasks with \( m=16 \), and periods=logunif

Bottom line: larger number of tasks \( \rightarrow \) easier to partition.
Improving Upon Partitioning

Worst-Case Loss

- Partitioning may cause almost up to **50% utilization loss**!
- For **pathological task sets**, the system is half-idle!
- It gets much more difficult for non-independent task sets
  ‣ Locks, precedence, etc.

Can’t we do better?

- Can we achieve a utilization bound of \( m \)?
- Avoid **offline** assignment phase?
- Global scheduling…
Global Scheduling

General Approach
- At each point in time, assign each job a priority
- At any point in time, schedule the $m$ highest-priority jobs

Implementation
- Conceptually a globally shared ready queue
- Actual implementation can differ
- efficient & correct: ongoing research

Challenges
- migrations require coordination
- cache affinity
- lock contention
- e.g., see Linux
Classification of Scheduling Policies

**Task-Level Fixed-Priority (FP) Scheduler** *(static priorities)*
- Each *task* is assigned a fixed priority
- All jobs (of a task) have the same priority
- Example: Rate-Monotonic Scheduling

**Job-Level Fixed-Priority (JLFP) Scheduler** *(dynamic priorities)*
- The priority of each task *changes over time*.
- The priority of a job does *not* change.
- Example: EDF

**Job-Level Dynamic-Priority (JLDP) Scheduler**
- No restrictions.
- The priority of each job changes over time.
- Priorities are a function of *time, job identity*, and *system state*. 
Unknown Critical Instant

Critical Instant
- Job release time such that response time is maximized.
- Exists unless system is over-loaded.

Uniprocessor
- Liu & Layland: synchronous release sequence yields worst-case response-times
  - synchronous: all tasks release a job at time 0
  - assuming constrained deadlines and no deadline misses

Multiprocessors
- No general critical instant is known!
- It is not necessarily the synchronous release sequence.
- A G-EDF example…
The synchronous release sequence is not always the worst case!
Non-Optimality of Global EDF

Uniprocessor
- EDF is optimal

Multiprocessor
- G-EDF is not optimal (w.r.t. meeting deadlines)
- Key problem: **sequentiality** of tasks
  - Two processors available for $T_5$, but it can only use one.
Non-Optimality of G-JLFP Scheduling

Any Job-Level Fixed-Priority Scheduling Policy is not optimal

- Example: two processors, three tasks
  - Period 15, WCET = 10
  - Synchronous release at time 0
- One of the three jobs is scheduled last under any JLFP policy
  - Deadline miss inevitable!
Global JLDP Example

- Job priority changes
  - $T_1$ scheduled on processor 1
  - $T_2$ scheduled on processor 2
  - $T_3$ scheduled on processor 1

- Release and deadline markers
  - $T_1$ released at time 0, deadline at time 10
  - $T_2$ released at time 5, deadline at time 15
  - $T_3$ released at time 10, deadline at time 20

- Completion markers
  - $T_1$ completed at time 10
  - $T_2$ completed at time 15
  - $T_3$ completed at time 20

- G-JLDP examples
  - Global JLDP (G-JLDP)

- Diagram indicating job priority changes
G-EDF is a JLFP Policy

- Can (pseudo-)deadlines be used to schedule correctly?
- **Yes**, but deadlines alone are not enough.
  - Need to break jobs into “smaller pieces”.
  - Need appropriate **tie-breaking rules**.
- $\text{PD}^2$
Optimal Multiprocessor Scheduling

G-EDF

Pfair / PD²
Optimal Multiprocessor Scheduling

**Pfair**
- Notion of “fair share of processor”—always proportional to utilization
- If a schedule is pfair, then no implicit deadline will be missed.

**PD^2**
- Constructs a pfair schedule.
- Splits jobs into unit-sized subtasks
  - Each subtask has its own deadline
- Uses two deadline tie-breaking rules
PD² Illustration

Brief PD² Overview
- split jobs into subtasks
- assign subtask release times and deadlines
  ‣ pfair windows

Successor Bit
- 1 if subtask window overlaps with that of successor
- 0 otherwise

Group Deadline
- intuitively, how far does the “ripple effect” of *not immediately scheduling this subtask* extend

Runtime Scheduler
- earliest *pseudo* deadline first
- first, tie-break in favor of *non-zero successor bit*
- second, tie-break in favor of *later group deadline*
Optimal Online Scheduling of Sporadic Tasks with Arbitrary Deadlines

Is it possible to extend $P_{fair}/P_{D^2}$ to support arbitrary deadlines?
Optimal Online Scheduling of Sporadic Tasks with Arbitrary Deadlines

**Theorem**: there does not exist an *online* scheduler that *optimally* schedules sporadic tasks with constrained deadlines.

Non-Existence of Optimal Online Schedulers for General Sporadic Tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET</th>
<th>Deadline</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$T_3$</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>$T_4$</td>
<td>2</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>$T_5$</td>
<td>2</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>$T_6$</td>
<td>4</td>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>

which job goes next?
Non-Existence of Optimal Online Schedulers for General Sporadic Tasks

If $T_5$ goes first, then $T_6$ can miss its deadline.

New jobs at time 6.

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET</th>
<th>Deadline</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$T_3$</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>$T_4$</td>
<td>2</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>$T_5$</td>
<td>2</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>$T_6$</td>
<td>4</td>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>
Non-Existence of Optimal Online Schedulers for General Sporadic Tasks

If $T_6$ goes first, then $T_5$ can miss its deadline.

<table>
<thead>
<tr>
<th>Task</th>
<th>WCET</th>
<th>Deadline</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$T_2$</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$T_3$</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>$T_4$</td>
<td>2</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>$T_5$</td>
<td>2</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>$T_6$</td>
<td>4</td>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>

New jobs at time 5.
Non-Existence of Optimal Online Schedulers for General Sporadic Tasks

If $T_5$ goes first, then $T_6$ can miss its deadline.

If $T_6$ goes first, then $T_5$ can miss its deadline.

The task set is **feasible**, but correct decision requires **knowledge of future arrivals**!
Clustered Scheduling

A hybrid / generalization of global and partitioned scheduling.
Clustered Scheduling

smaller clusters = harder bin packing instance

larger clusters = higher overheads

partitioned scheduling

clustered scheduling

global scheduling
Semi-Partitioned Scheduling

another generalization of partitioned scheduling

High-Level Idea

Partition as far as possible, then *split* remaining tasks into *sub-tasks with jitter*.

Example: “Each job of split task executes for 10ms on core 1 and for remaining 4 on core 2.”
Semi-Partitioned Scheduling

*another generalization of partitioned scheduling*

**Partition first**
- Assign each task statically to a processor if possible
- Keep track which tasks could not be assigned (if any)
- Details vary according to specific **semi-partitioned** algorithm

**Split remaining tasks across multiple processors**
- Split each unassigned task into multiple “portions” or “chunks”
- Distribute portions/chunks among multiple processors
  - E.g., split each job into **subjobs with precedence constraints**
  - Alternatively, do not migrate jobs, but vary a task’s processor assignment over time (soft real-time)
Summary

Approaches
- Partitioned
- Global
- Hybrid
  - Clustered
  - Semi-Partitioned
  - Arbitrary Processor Affinities...

Priorities
- Task-Level Fixed Priority
- Job-Level Fixed Priority
- Job-Level Dynamic Priority

Optimal Online Scheduling
- Implicit deadlines: requires global job-level dynamic priority scheduler
- Constrained deadlines: does not exist
- Arbitrary deadlines: does not exist