On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study

Sathish Gopalakrishnan
The University of British Columbia
(based on work by others at the University of North Carolina)
Focus of this Talk

- Multicore platforms are predicted to get much larger in the future.
  » 10s or 100s of cores per chip, multiple hardware threads per core.

Research Question: How will different real-time scheduling algorithms scale?

» Scalability is defined w.r.t. schedulability (more on this later).
Outline

- **Background.**
  - Real-time workload assumed.
  - Scheduling algorithms evaluated.
  - Some properties of these algorithms.

- **Research questions addressed.**

- **Experimental results.**

- **Observations/speculation.**

- **Future work.**
Real-Time Workload Assumed in this Talk

- Set $\tau$ of periodic tasks scheduled on $M$ cores:

$$T = (2,5)$$

$$U = (9,15)$$

One Core Here
Real-Time Workload Assumed in this Talk

- Set $\tau$ of periodic tasks scheduled on $M$ cores:
  - Task $T = (T.e, T.p)$ releases a job with exec. cost $T.e$ every $T.p$ time units.
  - $T$'s utilization (or weight) is $U(T) = T.e/T.p$.
  - Total utilization is $U(\tau) = \Sigma T.e/T.p$.
Real-Time Workload Assumed in this Talk

- Set $\tau$ of periodic tasks scheduled on $M$ cores:
  - Task $T = (T.e, T.p)$ releases a job with exec. cost $T.e$ every $T.p$ time units.
  - $T$'s utilization (or weight) is $U(T) = T.e/T.p$.
  - Total utilization is $U(\tau) = \sum_T T.e/T.p$.

**Diagram:**

- $T = (2,5)$
- $U = (9,15)$
Real-Time Workload Assumed in this Talk

- Set $\tau$ of periodic tasks scheduled on $M$ cores:
  - Task $T = (T.e, T.p)$ releases a job with exec. cost $T.e$ every $T.p$ time units.
    - $T$'s utilization (or weight) is $U(T) = T.e/T.p$.
    - Total utilization is $U(\tau) = \sum_T T.e/T.p$.
  - Each job of $T$ has a deadline at the next job release of $T$. 

![Diagram showing task T and its utilization U with intervals and deadlines]
Real-Time Workload Assumed in this Talk

Set $\tau$ of periodic tasks scheduled on $M$ cores:

- Task $T = (T.e, T.p)$ releases a job with exec. cost $T.e$ every $T.p$ time units.
  - $T$'s utilization (or weight) is $U(T) = T.e/T.p$.
  - Total utilization is $U(\tau) = \sum T.e/T.p$.
- Each job of $T$ has a deadline at the next job release of $T$. 

![Diagram showing T = (2,5) and U = (9,15)]
Real-Time Workload Assumed in this Talk

- Set $\tau$ of periodic tasks scheduled on $M$ cores:
  - Task $T = (T_e, T_p)$ releases a job with exec. cost $T_e$ every $T_p$ time units.
  - Task $T$'s utilization (or weight) is $U(T) = \frac{T_e}{T_p}$.
  - Total utilization is $U(\tau) = \sum_T \frac{T_e}{T_p}$.
  - Each job of $T$ has a deadline at the next job release of $T$.

This is an *earliest-deadline-first* schedule. Much of our work pertains to EDF scheduling...

- To show: much of our work pertains to EDF scheduling...

---

**Example**

- $T = (2,5)$
- $U = (9,15)$
- One Core Here

---

**Real-Time Scalability**

9
Scheduling vs. Schedulability

W.r.t. scheduling, we actually care about **two** kinds of algorithms:

- **Scheduling algorithm** (of course).
  - **Example**: Earliest-deadline-first (EDF): Jobs with earlier deadlines have higher priority.

- **Schedulability test**.

![Diagram](Image)
Multiprocessor Real-Time Scheduling

Two Approaches:

Partitioning

Steps:
1. Assign tasks to processors (bin packing).
2. Schedule tasks on each processor using a uniprocessor algorithm.

Global Scheduling

Important Differences:
- One task queue.
- Tasks may migrate among the processors.
Scheduling Algorithms Considered

- **Partitioned EDF:** PEDF.
- **Preemptive & Non-preemptive Global EDF:** GEDF & NP-GEDF.
- **Clustered EDF:** CEDF.
  - Partition onto clusters of cores, globally schedule within each cluster
Scheduling Algorithms (Continued)

- **PD²**, a global *Pfair* algorithm.
  - Schedule jobs one quantum at a time at a “uniform” rate.
    - May preempt and migrate jobs frequently.

- **Staggered PD²**: *S-PD²*.
  - Same as PD² but quanta are “staggered” to avoid excessive bus contention.
Under partitioning & most global algorithms, overall utilization must be capped to avoid deadline misses. Due to connections to bin-packing. Exception: Global "Pfair" algorithms do not require caps. Such algorithms schedule jobs one quantum at a time. May therefore preempt and migrate jobs frequently. Perhaps less of a concern on a multicore platform. Under most global algorithms, if utilization is not capped, deadline tardiness is bounded. Sufficient for soft real-time systems.

3 tasks with parameters (2,3) on two processors...

On Processor 1

On Processor 2

T = (2,3)

U = (2,3)

V = (2,3)
Schedulability

- **HRT**: No deadline is missed.
- **SRT**: Deadline tardiness is bounded.
- For some scheduling algorithms, utilization loss is inherent when checking schedulability.

» That is, schedulability cannot be guaranteed for all task systems with total utilization at most M.
Example: Partitioning three tasks with parameters (2,3) on two processors will overload one processor.

In terms of bin-packing...

```
Processor 1
0

1

Processor 2

Task 1

Task 2

Task 3
```
# Schedulability Summary

<table>
<thead>
<tr>
<th></th>
<th>HRT</th>
<th>SRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEDF</td>
<td>util. loss</td>
<td>util. loss (same as HRT)</td>
</tr>
<tr>
<td>GEDF</td>
<td>util. loss</td>
<td>no loss</td>
</tr>
<tr>
<td>NP-GEDF</td>
<td>util. loss</td>
<td>no loss</td>
</tr>
<tr>
<td>CEDF</td>
<td>util. loss</td>
<td>util. loss (not as bad as PEDF)</td>
</tr>
<tr>
<td>PD²</td>
<td>no loss</td>
<td>no loss</td>
</tr>
<tr>
<td>S-PD²</td>
<td>slight loss</td>
<td>no loss</td>
</tr>
<tr>
<td></td>
<td>(must shrink periods by one quantum)</td>
<td></td>
</tr>
</tbody>
</table>
Real-Time Scalability

GEDF SRT Example

Earlier example with GEDF…

Tardiness is at most one quantum.

T = (2,3)

U = (2,3)

V = (2,3)
Outline

- Background.
  - Real-time workload assumed.
  - Scheduling algorithms evaluated.
  - Some properties of these algorithms.
- Research questions addressed.
- Experimental results.
- Observations/speculation.
- Future work.
Research Questions

- In *theory*, PD² is always preferable.
  - It is optimal (no utilization loss).

Focus of this Talk: An Experimental comparison of these scheduling algorithms on the basis of *schedulability*.

- Do migrations really matter on a multicore platform with a *shared cache*?  
- As multicore platforms get larger, will global algorithms *scale*?
Test System

- **HW platform:** Sun Niagara (UltraSPARC T1).

  - 4 HW threads per core
  - 16K (8K) L1 instr. (data) cache per core
  - Shared 3MB L2

- 1.2 GHz “RISC-like” cores.
- Relatively simple, e.g., no instr. reordering or branch prediction.
- Caches somewhat small compared to Intel.

- OS has 32 “logical CPUs” to manage.
- **Far larger than any system considered before in RT literature.**
- **Note:** CEDF “cluster” = 4 HW threads on a core.
Test System (Cont’d)

- Operating System: **LITMUS[^RT]**: **L**inux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems.
  - Developed at UNC.
  - Extends Linux by allowing different schedulers to be linked as “plug-in” components.
  - Several (real-time) synchronization protocols are also supported.
  - Code is available at [http://www.cs.unc.edu/~anderson/litmus-rt/](http://www.cs.unc.edu/~anderson/litmus-rt/)

[^RT]: **RT** stands for Real-Time.
Methodology

- Ran several hundred (synthetic) task sets on the test system.
- Collected 70 GB of raw overhead samples.
- Distilled expressions for average (for SRT) and worst-case (for HRT) overheads.
- Conducted schedulability experiments involving 8.5 million randomly-generated task sets with overheads considered.

Note: This step is offline. It does not involve the Niagara.
Kinds of Overheads

- **Tick scheduling overhead.**
  » Incurred when the kernel is invoked at the beginning of each quantum (timer “tick”). A quantum is 1ms.

- **Release overhead.**
  » Incurred when the kernel is invoked to handle a job release.

- **Scheduling overhead.**
  » Incurred when the scheduler (in the kernel) is invoked.

- **Context-switching overhead.**
  » Non-cache-related costs associated with a context switch.

- **Preemption/migration overhead.**
  » Costs incurred upon a preemption/migration due to a loss of cache affinity.

These overheads can be accounted for in schedulability tests by **inflating job execution costs**. (Doing this correctly is a little tricky.)
Kernel Overheads

- Most overheads were small (2-15μs) except worst-case overheads impacted by global queues.

  » Most notable: Worst-case scheduling overheads for PD\(^2\), S-PD\(^2\), and GEDF/NP-GEDF:

<table>
<thead>
<tr>
<th>Alg</th>
<th>Scheduling Overhead (in μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>32.7</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>43.1</td>
</tr>
<tr>
<td>GEDF/NP-GEDF</td>
<td>55.2+.26N (N = no. of tasks)</td>
</tr>
</tbody>
</table>
Preemption/Migration Overheads

Obtained by measuring synthetic tasks, each with a 64K working set & 75/25 read/write ratio.

» **Interesting trends:** PD\(^2\) is terrible, staggering really helps, preempt. cost ≈ mig. cost per algorithm, but algorithms that migrate have higher costs.

### Worst-Case Overheads (in µs)

<table>
<thead>
<tr>
<th>Alg</th>
<th>Overall</th>
<th>Preemption</th>
<th>Intra-Cluster Mig</th>
<th>Inter-Cluster Mig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>681.1</td>
<td>649.4</td>
<td>654.2</td>
<td>681.1</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>104.1</td>
<td>103.4</td>
<td>103.4</td>
<td>104.1</td>
</tr>
<tr>
<td>GEDF</td>
<td>375.4</td>
<td>375.4</td>
<td>326.8</td>
<td>321.1</td>
</tr>
<tr>
<td>CEDF</td>
<td>171.6</td>
<td>171.6</td>
<td>167.3</td>
<td>---</td>
</tr>
<tr>
<td>PEDF</td>
<td>139.1</td>
<td>139.1</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Schedulability Results

- Generated random tasks using 6 distributions and checked schedulability using “state-of-the-art” tests (with overheads considered).
  - 8.5 million task sets in total.

- Distributions:
  - Utilizations uniform over
    - [0.001, 0.01] (light),
    - [0.1, 0.4] (medium), and
    - [0.5, 0.9] (heavy).
  - Bimodal with utilizations distributed over either [0.001, 0.05) or [0.5, 0.9] with probabilities of
    - 8/9 and 1/9 (light),
    - 6/9 and 3/9 (medium), and
    - 4/9 and 5/9 (heavy).
Schedulability Results

- Generated random tasks using 6 distributions and checked schedulability using “state-of-the-art” tests (with overheads considered).
  - 8.5 million task sets in total.

- Distributions:
  - Utilizations uniform over
    - [0.001,01] (light),
    - [0.1,0.4] (medium), and
    - [0.5,0.9] (heavy).

- Bimodal with utilizations distributed over either [0.001,05) or [0.5,09] with probabilities of
  - 8/9 and 1/9 (light),
  - 6/9 and 3/9 (medium), and
  - 4/9 and 5/9 (heavy).

【will only show graphs for these】
HRT Summary

- **PEDF usually wins.**
  - Exception: Lots of heavy tasks (makes bin-packing hard).

- **S-PD^2 usually does well.**
  - Staggering has an impact.

- **PD^2 and GEDF are quite poor.**
  - PD^2 is negatively impacted by high preemption and migration costs due to aligned quanta.
  - GEDF suffers from high scheduling costs (due to the global queue).
HRT, Bimodal Light

PEDF performs pretty well if most task utilizations are low.

S-PD² performs pretty well too.
HRT, Bimodal Light

bimodally distributed in [0.001, 0.5] (8/9) and [0.5, 0.9] (1/9)
In this and the next slide, as the fraction of heavy tasks grows, the gap between S-PD$^2$ and PEDF narrows.
HRT, Bimodal Medium

bimodally distributed in [0.001, 0.5] (6/9) and [0.5, 0.9] (3/9)
HRT, Bimodal Heavy

bimodally distributed in [0.001, 0.5] (4/9) and [0.5, 0.9] (5/9)

utilization cap

schedulability

P-EDF  C-EDF  G-EDF  PFAIR  S-PFAIR

Real-Time Scalability
SRT Summary

- **PEDF** is not as effective as before, but still OK in light-mostly cases.
- **CEDF** performs the best in most cases.
- **S-PD²** still performs generally well.
- **GEDF** is still negatively impacted by higher scheduling costs.
  - Note: SRT schedulability for GEDF entails no utilization loss.
  - Note: NP-GEDF and GEDF are about the same.
- **Note:** The scale is different from before.
PEDF and CEDF perform well if tasks are mostly light.

Note: S-PD² never performs really badly in any experiment.
SRT, Bimodal Light

bimodally distributed in $[0.001, 0.5]$ (8/9) and $[0.5, 0.9]$ (1/9)
This and the next slide show that as the frequency of heavy tasks increases, PEDF degrades. CEDF isn’t affected by this increase much.
SRT, Bimodal Medium

bimodally distributed in [0.001, 0.5] (6/9) and [0.5, 0.9] (3/9)
SRT, Bimodal Heavy

bimodally distributed in [0.001, 0.5] (4/9) and [0.5, 0.9] (5/9)

schedulability

utilization cap

P-EDF  C-EDF  G-EDF  PFAIR  S-PFAIR  G-NP-EDF
Outline

- Background.
  - Real-time workload assumed.
  - Scheduling algorithms evaluated.
  - Some properties of these algorithms.
- Research questions addressed.
- Experimental results.
- Observations/speculation.
- Future work.
Global algorithms are really sensitive to how shared queues are implemented.

- Saw 100X performance improvement by switching from linked lists to binomial heaps.
- Still working on this...
- **Speculation:** Can reduce GEDF costs to close to PEDF costs for systems with \( \leq 32 \) cores.

**Per algorithm, preempt. cost \( \approx \) mig. cost.**

- Due to having a shared cache.
- One catch: Migrations increase **both** costs.

**Quantum staggering is very effective.**
No one “best” algorithm.

Intel has claimed they will produce an 80-core general-purpose chip. If they do…

» the cores will have to be simple $\Rightarrow$ high execution costs $\Rightarrow$ high utilizations $\Rightarrow$ PEDF will suffer;

» “pure” global algorithms will not scale;

» some instantiation of CEDF (or maybe CS-PD$^2$) will hit the “sweet spot”.
Future Work

- Thoroughly study “how to implement shared queues”.
- Repeat this study on Intel and embedded machines.
- Examine mixed HRT/SRT workloads.
- Factor in synchronization and dynamic behavior.

» In past work, PEDF was seen to be more negatively impacted by these things.
Thanks!

- Questions?
SRT Tardiness, Uniform Medium

uniformly distributed in [0.1, 0.4]

utilization cap

tardiness (in ms)

C-EDF  G-EDF  G-NP-EDF
Measuring Overheads

- Done using a UNC-produced tracer called Feather-Trace.
  - http://www.cs.unc.edu/~bbb/feathertrace/
- Highest 1% of values were tossed.
  - Eliminates “outliers” due to non-deterministic behavior in Linux, warm-up effects, etc.
- Used worst-case (average-case) values for HRT (SRT) schedulability.
- Used linear regression analysis to produce linear (in the task count) overhead expressions.
Obtaining Kernel Overheads

- Ran 90 (synthetic) task sets per scheduling algorithm for 30 sec.
- In total, over 600 million individual overheads were recorded (45 GB of data).
# Kernel Overheads (in $\mu$s)

$(N = \text{no. of tasks})$

## Worst-Case

<table>
<thead>
<tr>
<th>Alg</th>
<th>Tick</th>
<th>Schedule</th>
<th>Context SW</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD$^2$</td>
<td>11.2 +.3N</td>
<td>32.7</td>
<td>3.1+.01N</td>
<td>---</td>
</tr>
<tr>
<td>S-PD$^2$</td>
<td>4.8+.3N</td>
<td>43.1</td>
<td>3.2+.003N</td>
<td>---</td>
</tr>
<tr>
<td>GEDF</td>
<td>3+.003N</td>
<td>55.2+.26N</td>
<td>29.2</td>
<td>45+.3N</td>
</tr>
<tr>
<td>CEDF</td>
<td>3.2</td>
<td>14.8+.01N</td>
<td>6.1</td>
<td>30.3</td>
</tr>
<tr>
<td>PEDF</td>
<td>2.7+.002N</td>
<td>8.6+.01N</td>
<td>14.9+.04N</td>
<td>4.7+.009N</td>
</tr>
</tbody>
</table>

## Average

<table>
<thead>
<tr>
<th>Alg</th>
<th>Tick</th>
<th>Schedule</th>
<th>Context SW</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD$^2$</td>
<td>4.3+.03N</td>
<td>4.7</td>
<td>2.6+.001N</td>
<td>---</td>
</tr>
<tr>
<td>S-PD$^2$</td>
<td>2.1+.02N</td>
<td>4.2</td>
<td>2.5+.001N</td>
<td>---</td>
</tr>
<tr>
<td>GEDF</td>
<td>2.1+.002N</td>
<td>11.8+.06N</td>
<td>7.6</td>
<td>5.8+.1N</td>
</tr>
<tr>
<td>CEDF</td>
<td>2.8</td>
<td>6.1+.01N</td>
<td>3.2</td>
<td>16.5</td>
</tr>
<tr>
<td>PEDF</td>
<td>2.1+.002N</td>
<td>2.7+.008N</td>
<td>4.7+.005N</td>
<td>4+.005N</td>
</tr>
</tbody>
</table>
# Kernel Overheads (in $\mu$s)

(N = no. of tasks)

## Worst-Case

<table>
<thead>
<tr>
<th>Alg</th>
<th>Tick</th>
<th>Schedule</th>
<th>Context SW</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD$^2$</td>
<td>11.2+.3N</td>
<td>32.7</td>
<td>3.1+.01N</td>
<td>---</td>
</tr>
<tr>
<td>S-PD$^2$</td>
<td>4.8+.3N</td>
<td>43.1</td>
<td>3.2+.003N</td>
<td>---</td>
</tr>
<tr>
<td>GEDF</td>
<td>3+.003N</td>
<td>55.2+.26N</td>
<td>29.2</td>
<td>45+.3N</td>
</tr>
<tr>
<td>CEDF</td>
<td>3.2</td>
<td>14.8+.01N</td>
<td>6.1</td>
<td>30.3</td>
</tr>
<tr>
<td>PEF</td>
<td>2.7+.002N</td>
<td>8.6+.01N</td>
<td>14.9+.04N</td>
<td>4.7+.009N</td>
</tr>
</tbody>
</table>

## Average

<table>
<thead>
<tr>
<th>Alg</th>
<th>Tick</th>
<th>Schedule</th>
<th>Context SW</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD$^2$</td>
<td>4.3+.03N</td>
<td>4.7</td>
<td>2.6+.001N</td>
<td>---</td>
</tr>
<tr>
<td>S-PD$^2$</td>
<td>2.1+.02N</td>
<td>4.2</td>
<td>2.5+.001N</td>
<td>---</td>
</tr>
<tr>
<td>GEDF</td>
<td>2.1+.002N</td>
<td>11.8+.06N</td>
<td>7.6</td>
<td>5.8+.1N</td>
</tr>
<tr>
<td>CEDF</td>
<td>2.8</td>
<td>6.1+.01N</td>
<td>3.2</td>
<td>16.5</td>
</tr>
<tr>
<td>PEF</td>
<td>2.1+.002N</td>
<td>2.7+.008N</td>
<td>4.7+.005N</td>
<td>4+.005N</td>
</tr>
</tbody>
</table>
Obtaining Preemption/Migration Overheads

- Ran 90 (synthetic) task sets per scheduling algorithm for 60 sec.
- Each task has a 64K working set (WS) that it accesses repeatedly with a 75/25 read/write ratio.
- Recorded time to access WS after preemption/migration minus “cache-warm access”.
- In total, over 105 million individual preemption/migration overheads were recorded (15 GB of data).
### Preemption/Migration Overheads (in \(\mu s\))

(N = no. of tasks)

#### Worst-Case

<table>
<thead>
<tr>
<th>Alg</th>
<th>Overall</th>
<th>Preemption</th>
<th>Intra-Cluster Mig</th>
<th>Inter-Cluster Mig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>681.1</td>
<td>649.4</td>
<td>654.2</td>
<td>681.1</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>104.1</td>
<td>103.4</td>
<td>103.4</td>
<td>104.1</td>
</tr>
<tr>
<td>GEDF</td>
<td>375.4</td>
<td>375.4</td>
<td>326.8</td>
<td>321.1</td>
</tr>
<tr>
<td>CEDF</td>
<td>171.6</td>
<td>171.6</td>
<td>167.3</td>
<td>---</td>
</tr>
<tr>
<td>PEDF</td>
<td>139.1</td>
<td>139.1</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

#### Average

<table>
<thead>
<tr>
<th>Alg</th>
<th>Overall</th>
<th>Preemption</th>
<th>Intra-Cluster Mig</th>
<th>Inter-Cluster Mig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>172</td>
<td>131.4</td>
<td>141.8</td>
<td>187.6</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>89.3</td>
<td>86.2</td>
<td>87.8</td>
<td>90.2</td>
</tr>
<tr>
<td>GEDF</td>
<td>73</td>
<td>95.1</td>
<td>73.5</td>
<td>72.6</td>
</tr>
<tr>
<td>CEDF</td>
<td>67</td>
<td>78.5</td>
<td>64.8</td>
<td>---</td>
</tr>
<tr>
<td>PEDF</td>
<td>72.3</td>
<td>72.3</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Preemption/Migration Overheads (in \( \mu s \))
(N = no. of tasks)

**Worst-Case**

<table>
<thead>
<tr>
<th>Alg</th>
<th>Overall</th>
<th>Preemption</th>
<th>Intra-Cluster Mig</th>
<th>Inter-Cluster Mig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>681.1</td>
<td>649.4</td>
<td>654.2</td>
<td>681.1</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>104.1</td>
<td>103.4</td>
<td>103.4</td>
<td>104.1</td>
</tr>
<tr>
<td>GEDF</td>
<td>375.4</td>
<td>375.4</td>
<td>326.8</td>
<td>321.1</td>
</tr>
<tr>
<td>CEDF</td>
<td>171.6</td>
<td>171.6</td>
<td>167.3</td>
<td>---</td>
</tr>
<tr>
<td>PEDF</td>
<td>139.1</td>
<td>139.1</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Average**

<table>
<thead>
<tr>
<th>Alg</th>
<th>Overall</th>
<th>Preemption</th>
<th>Intra-Cluster Mig</th>
<th>Inter-Cluster Mig</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD(^2)</td>
<td>172</td>
<td>131.4</td>
<td>141.8</td>
<td>187.6</td>
</tr>
<tr>
<td>S-PD(^2)</td>
<td>89.3</td>
<td>86.2</td>
<td>87.8</td>
<td>90.2</td>
</tr>
<tr>
<td>GEDF</td>
<td>73</td>
<td>95.1</td>
<td>73.5</td>
<td>72.6</td>
</tr>
<tr>
<td>CEDF</td>
<td>67</td>
<td>78.5</td>
<td>64.8</td>
<td>---</td>
</tr>
<tr>
<td>PEDF</td>
<td>72.3</td>
<td>72.3</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
This is the easiest case for partitioning, so PEDF wins.

\textbf{S-PD}^2\textsuperscript{2} does pretty well too.
HRT, Uniform Light

uniformly distributed in $[0.001, 0.1]$
HRT, Uniform Medium

Similar to before.

Utilizations aren’t high enough to start causing problems for partitioning.
HRT, Uniform Medium

uniformly distributed in [0.1, 0.4]
HRT, Uniform Heavy

Utilizations are high enough to cause problems for partitioning.

S-PD² wins now.
HRT, Uniform Heavy

uniformly distributed in [0.5, 0.9]
SRT, Uniform Light

uniformly distributed in [0.001, 0.1]

PEDF wins, S-PD² performs pretty well.
SRT, Uniform Light

uniformly distributed in [0.001, 0.1]

utilization cap

schedulability

P-EDF
C-EDF
G-EDF
G-NP-EDF
S-PFAIR
PFAIR
CEDF really benefits from using a “no utilization loss” schedulability test within each cluster.
SRT, Uniform Medium

uniformly distributed in [0.1, 0.4]
SRT, Uniform Heavy

GEDF and NP-GEDF actually win in this case.

CEDF and S-PD^2 perform pretty well.

PEDF loses.
SRT, Uniform Heavy

uniformly distributed in [0.5, 0.9]
On the Implementation of Global Real-Time Schedulers

Simon Fraser University
April 15, 2010

Sathish Gopalakrishnan
The University of British Columbia

Work supported by IBM, SUN, and Intel Corps., NSF grants CNS 0834270, CNS 0834132, and CNS 0615197, and ARO grant W911NF-06-1-0425.
UNC’s Implementation Studies (I)

Calandrino et al. (2006)

➢ Are commonly-studied RT schedulers implementable?
➢ In Linux on common hardware platforms?


UNC’s Implementation Studies (I)

Calandrino et al. (2006)

➡ Are commonly-studied RT schedulers implementable?
➡ In Linux on common hardware platforms?

Intel 4x 2.7 GHz Xeon SMP
(few, fast processors; private caches)

UNC’s Implementation Studies (I)

Calandrino et al. (2006)

- Are commonly-studied RT schedulers implementable?
- In Linux on common hardware platforms?

partitioned EDF

2 x global EDF

2 x PFAIR


UNC’s Implementation Studies (I)

Are commonly-studied RT schedulers implementable in Linux on common hardware platforms?

“for each tested scheme, scenarios exist in which it is a viable choice”


UNC’s Implementation Studies (II)

Brandenburg et al. (2008)

→ What if there are many slow processors?


Tuesday, April 5, 2011
**UNC’s Implementation Studies (II)**

**Brandenburg et al. (2008)**

- What if there are many slow processors?
- Explored scalability of RT schedulers on a Sun Niagara.

---


UNC’s Implementation Studies (II)

Brandenburg et al. (2008)

→ What if there are many slow processors?
→ Explored scalability of RT schedulers on a Sun Niagara.

G-EDF: high overheads, low schedulability.


Today’s discussion

How to implement global schedulers?


Today’s discussion

How to implement global schedulers?

→ Explore how implementation tradeoffs affect schedulability.


Today’s discussion

How to implement global schedulers?
- Explore how **implementation tradeoffs** affect **schedulability**.
- Case study: **nine G-EDF variants** on a Sun Niagara.


Tuesday, April 5, 2011
Design Choices
Design Choices

- When to schedule.
- Quantum alignment.
- How to handle interrupts.
- How to queue pending jobs.
- How to manage future releases.
- How to avoid unnecessary preemptions.
Scheduler Invocation
Scheduler Invocation

Event-Driven

- on job release
- on job completion
- preemptions occur immediately

On the Implementation of Global Real-Time Schedulers
Scheduler Invocation

**Event-Driven**
- on job release
- on job completion
- preemptions occur immediately

**Quantum-Driven**
- on every timer tick
- easier to implement
- on release a job is just enqueued; scheduler is invoked at next tick
Quantum Alignment

Aligned
- Tick synchronized across processors.
- **Contention** at quantum boundary!

![Diagram showing quantum boundary and processors](image)
Quantum Alignment

**Staggered**
- Ticks spread out across quantum.
- **Reduced** bus and lock contention.
- Additional **latency**.

**Aligned**
- Tick **synchronized** across processors.
- **Contention** at quantum boundary!
Quantum Alignment

**Staggered**
- Ticks spread out across quantum.
- **Reduced** bus and lock contention.
- Additional **latency**.

**Aligned**
- Tick **synchronized** across processors.
- **Contention** at quantum boundary!
Quantum Alignment

Staggered
- Ticks spread out across quantum.
- **Reduced** bus and lock contention.
- Additional **latency**.

Aligned
- Tick **synchronized** across processors.
- **Contention** at quantum boundary!
Staggered
- Ticks spread out across quantum.
- Reduced bus and lock contention.
- Additional latency.

Aligned
- Tick synchronized across processors.
- Contention at quantum boundary!
Interrupt Handling
Interrupt Handling

Global interrupt handling.
➡ Job releases triggered by interrupts.
➡ Interrupts may fire on any processor.
➡ Jobs may execute on any processor.
➡ Thus, in the worst case, a job may be delayed by each interrupt.
Interrupt Handling

Global interrupt handling.
- Job releases triggered by interrupts.
- Interrupts may fire on any processor.
- Jobs may execute on any processor.
- Thus, in the worst case, a job may be delayed by each interrupt.

Dedicated interrupt handling.
- Only one processor services interrupts.
- Jobs may execute on other processors.
- Jobs are not delayed by release interrupts.
- Well-known technique; used in the Spring kernel (Stankovic and Ramamirtham, 1991).
- How does it affect schedulability?

Ready Queue
On the Implementation of Global Real-Time Schedulers

**Ready Queue**

Globally-shared priority queue.

- Problem: **hyper-period boundaries**.
- Problem: **lock contention**.
- Problem: **bus contention**.
Ready Queue

Globally-shared priority queue.

- Problem: **hyper-period boundaries**.
- Problem: **lock contention**.
- Problem: **bus contention**.

Requirements.

- **Mergeable** priority queue: release \( n \) jobs in \( O(\log n) \) time.
- **Parallel** enqueue / dequeue operations.
- Mostly **cache-local** data structures.
Ready Queue

Globally-shared priority queue.

➡ Problem: hyper-period boundaries.
➡ Problem: lock contention.
➡ Problem: bus contention.

In this study, we consider three queue implementations.

Coarse-Grained Heap  Hierarchical Heaps  Fine-Grained Heap
Ready Queue: Coarse-Grained Heap

Binomial heap + single lock.
- Lock used to synchronize all G-EDF state.
- **Mergeable** queue.
- No parallel updates.
- No cache-local updates.
- Low locking overhead
  (only single lock acquisition).
Ready Queue: Hierarchical Heaps

Per-processor queues + master queue.

- Each queue protected by a lock.
- Master queue holds min element of each per-processor queue.
- **Global, sequential** dequeue operations.
- **Mostly-local** enqueue operations.
**Ready Queue: Hierarchical Heaps**

**Per-processor queues + master queue.**
- Each queue protected by a lock.
- Master queue holds min element of each per-processor queue.
- **Global, sequential** dequeue operations.
- **Mostly-local** enqueue operations.

**Locking.**
- Dequeue: top-down.
- Enqueue: bottom-up.
- Enqueue may have to drop lock, retry.
- Additional complexity wrt. dequeue (see paper).
- Bottom line: **expensive**.
Ready Queue: Fine-Grained Heap

Parallel binary heap.
- One lock per heap node.
- Proposed by Hunt et al. (1996).
- **Not mergeable**.
- **Parallel enqueue / dequeue**.
- **No cache-local data**.

Ready Queue: Fine-Grained Heap

**Parallel binary heap.**
- One lock per heap node.
- Proposed by Hunt et al. (1996).
- **Not mergeable.**
- **Parallel enqueue / dequeue.**
- **No cache-local data.**

**Locking.**
- Many lock acquisitions.
- Atomic **peek+dequeue** operation needed to check for preemptions.

Additional Components

Release queue.
- Support mergeable queues.
- Support dedicated interrupt handling.

Job-to-processor mapping.
- Quickly determine whether preemption is required.
- Avoid unnecessary preemptions.
- Used to linearize concurrent scheduling decisions.
Implementation in LITMUS$^{RT}$
**LiMUSRT**

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

---

**LiMUSRT**

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

---

**UNC’s Linux patch.**

- Used in several previous studies.
- On-going development.
- Currently, based off of Linux 2.6.24.
On the Implementation of Global Real-Time Schedulers

**LitmusRT**
Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

**Linux Testbed for Multiprocessor Scheduling in Real-Time systems**

**UNC’s Linux patch.**
- Used in several previous studies.
- On-going development.
- Currently, based off of Linux 2.6.24.

**Scheduler Plugin API.**
- scheduler_tick()
- schedule()
- release_jobs()
## Considered G-EDF Variants

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>FEm</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>HEm</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>CE1</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
<tr>
<td>FE1</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
<tr>
<td>CQ1</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>dedicated</td>
</tr>
<tr>
<td>S-CQ1</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>dedicated</td>
</tr>
</tbody>
</table>
## Considered G-EDF Variants

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>HEm</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>FEm</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
</tbody>
</table>
### Considered G-EDF Variants

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>HEm</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>FEm</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
</tbody>
</table>

Baseline from (Brandenburg et al., 2008)
No fine-grained heaps + quantum-driven scheduling. (Parallel updates not beneficial due to quantum barrier.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>HEm</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>FEm</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
</tbody>
</table>
### Considered G-EDF Variants

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>HEm</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>FE1</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
<tr>
<td>CE1</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
<tr>
<td>CQ1</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>dedicated</td>
</tr>
<tr>
<td>S-CQ1</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>dedicated</td>
</tr>
<tr>
<td>FE1</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
</tbody>
</table>
No hierarchical heaps + dedicated interrupt handling.
(Hierarchical heaps not beneficial if only one proc. enqueues.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Ready Q</th>
<th>Scheduling</th>
<th>Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEm</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CQm</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>global</td>
</tr>
<tr>
<td>S-CQm</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>global</td>
</tr>
<tr>
<td>HE1m</td>
<td>hierarchical</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>FE1m</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>global</td>
</tr>
<tr>
<td>CE1</td>
<td>coarse-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
<tr>
<td>CQ1</td>
<td>coarse-grained</td>
<td>quantum (aligned)</td>
<td>dedicated</td>
</tr>
<tr>
<td>S-CQ1</td>
<td>coarse-grained</td>
<td>quantum (staggered)</td>
<td>dedicated</td>
</tr>
<tr>
<td>FE1</td>
<td>fine-grained</td>
<td>event-driven</td>
<td>dedicated</td>
</tr>
</tbody>
</table>
Schedulability Study
Objective

Compare the discussed implementations in terms of the ratio of randomly-generated task sets that can be shown to be schedulable under consideration of system overheads.
Scheduling Overheads
Scheduling Overheads

**Release overhead.**
- The cost of a one-shot timer interrupt.

**Scheduling overhead.**
- Selecting the next job to run.

**Context switch overhead.**
- Changing address space.
Scheduling Overheads

Release overhead.
→ The cost of a one-shot timer interrupt.

Scheduling overhead.
→ Selecting the next job to run.

Context switch overhead.
→ Changing address space.

Tick overhead.
→ Cost of a periodic timer interrupt.
→ Beginning of a new quantum.

Preemption and migration overhead.
→ Loss of cache affinity.
→ Known from (Brandenburg et al., 2008).
IPI Latency

**Inter-processor interrupts (IPIs).**
- Interrupt may be processed by a processor different from the one that will schedule a newly-arrived job.
- Requires notification of remote processor.
- **Event-based scheduling incurs added latency.**
Test Platform

**LITMUS**

- UNC’s Linux-based Real-Time Testbed

**Sun UltraSPARC T1 “Niagara”**

- 8 cores, 4 HW threads per core = 32 logical processors.
- 3 MB shared L2 cache
Test Platform

**LITMUS**
- UNC's Linux-based Real-Time Testbed

**Sun UltraSPARC T1 “Niagara”**
- 8 cores, 4 HW threads per core = 32 logical processors.
- 3 MB shared L2 cache

**Overheads**
- Traced overheads under each of the plugins.
- Collected more than 640,000,000 samples (total).
- Computed worst-case and average-case overheads.
- Over 20 graphs; see online version.

**Outliers**
- Removed top 1% of samples to discard outliers.
Example: Tick Overhead

“Higher is worse.”
Example: Tick Overhead

![Graph showing quantum-driven and event-driven tick overhead]

- Quantum-Driven
- Event-Driven

Overhead (us) vs. number of tasks:
- CEm tick overhead (worst-case)
- CE1 tick overhead (worst-case)
- FEm tick overhead (worst-case)
- FE1 tick overhead (worst-case)
- CQm tick overhead (worst-case)
- CQ1 tick overhead (worst-case)
- HEm tick overhead (worst-case)
- HEm tick overhead (worst-case)
Example: Release Overhead

![Graph showing release overhead for different schedulers: Event-Driven and Quantum-Driven. The x-axis represents the number of tasks, and the y-axis represents the overhead in microseconds (us). The graph includes lines for CEm release overhead (worst-case), CE1 release overhead (worst-case), FEm release overhead (worst-case), FE1 release overhead (worst-case), CQm release overhead (worst-case), CQ1 release overhead (worst-case), and HEm release overhead (worst-case).]
Study Setup

**Methodology.**
- Randomly generate task set.
- Apply overheads (for each G-EDF implementation).
- Test whether task set can be claimed schedulable (for each G-EDF implementation).
Study Setup

**Methodology.**
- Randomly generate task set.
- Apply overheads (for each G-EDF implementation).
- Test whether task set can be claimed schedulable (for each G-EDF implementation).

**Schedulability.**
- Hard real-time: worst-case overheads, no tardiness.
- Soft real-time: average-case overheads, bounded tardiness.
Study Setup

Methodology.
- Randomly generate task set.
- Apply overheads (for each G-EDF implementation).
- Test whether task set can be claimed schedulable (for each G-EDF implementation).

Schedulability.
- Hard real-time: worst-case overheads, no tardiness.
- Soft real-time: average-case overheads, bounded tardiness.

Task set generation.
- Six utilization distributions (uniform and bimodal).
- Three period distributions (uniform).
- Over 300 graphs; see online version.
Results

"Higher is better."
Interrupt Handling

utilization uniformly in [0.1, 0.4]; period uniformly in [10, 100]

Dedicated interrupt handling was generally preferable (or no worse).
Quantum Staggering

utilization uniformly in [0.001, 0.1]; period uniformly in [10, 100]

Aligned → Zero Overheads → Staggered

Staggered quanta were generally preferable (or no worse).
Quantum- vs. Event-Driven

utilization uniformly in [0.1, 0.4]; period uniformly in [10, 100]

Event-driven scheduling was preferable in most cases.
Choice of Ready Queue (1)

The coarse-grained ready queue performed better than the hierarchical queue.
The **fine-grained ready queue** performed marginally better than the coarse-grained queue if used together with **dedicated interrupt handling**.
Conclusion
Summary of Results

Implementation choices can impact schedulability as much as scheduling-theoretic tradeoffs.

Unless task counts are very high or periods very short, G-EDF can scale to 32 processors.
Recommendation

Best results obtained with combination of:

- fine-grained heap
- event-driven scheduling
- dedicated interrupt handling
Future Work

Platform.
➡ Repeat study on embedded hardware platform.

Implementation.
➡ Simplify locking requirements.
➡ Parallel mergeable heaps?

Analysis.
➡ Less pessimistic hard real-time G-EDF schedulability tests.
➡ Less pessimistic interrupt accounting.