Multicore Linux and Middleware

Chenyang Lu
CSE 520S
Motivation and Contributions

- Trend towards multi-processor and multi-core platforms affects both OS and middleware
  - Techniques designed for uni-processors need revisiting

- Contributions to real-time systems on multi-core platforms
  - A performance evaluation of relevant Linux features
  - MC-ORB middleware designed for multi-core platforms
  - Evaluation of MC-ORB’s multi-core aware RT performance
Background and Related Work

- Linux 2.6 introduced SMP and multi-core support
  - Linux 2.6.23 added the Completely Fair Scheduler (CFS)
  - However, many deployed platforms predate 2.6.23
  - We studied Linux 2.6.17 as a representative compromise

- We assume unmodified COTS Linux as our middleware design point, for highly portable real-time performance

- The differing trade-offs for uni-processor vs. multi-processor platforms motivate new middleware designs
We first evaluated clock differences between cores

- How well do platform/Linux maintain synchronization?
- We used RDTSC instruction to record clock ticks on each core

We bounced a message back and forth between two cores

- Used arrival TSCs \((x, y, z)\) to measure round trip delay (RTD)
- The results show that the cores’ frequencies were well matched
We then estimated the cores’ temporal offsets as
\[ \delta_0 = 2y_1 - x_0 - z_0 \; ; \; \delta_1 = 2y_0 - x_1 - z_1 \]

Insight 1

- Though frequencies matched well, avg. offset was \(\sim 1.3\mu s\)
- Motivates measuring offsets in our subsequent analyses
Linux Performance: Load Balancing

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Utilization</th>
<th>Imbalances detected in 5 min</th>
<th>Overhead per imbalance (ns)</th>
<th>Overhead (total μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum</td>
<td>Mean</td>
</tr>
<tr>
<td>10</td>
<td>0.6</td>
<td>211</td>
<td>405</td>
<td>983</td>
</tr>
<tr>
<td>30</td>
<td>0.6</td>
<td>210</td>
<td>566</td>
<td>1178</td>
</tr>
<tr>
<td>10</td>
<td>1.0</td>
<td>588</td>
<td>536</td>
<td>854</td>
</tr>
<tr>
<td>30</td>
<td>1.0</td>
<td>596</td>
<td>671</td>
<td>1124</td>
</tr>
</tbody>
</table>

- Can thread affinity thwart (bad) Linux rebalancing?
  - We ran sets of 10 vs. 30 tasks (all bound to one core to prevent rebalancing), with total utilizations of 0.6 vs. 1.0

- Insight 2
  - Though overhead is small and amortized, compiling kernels with rebalancing off appears to be a preferable method.
Linux Performance: Migration Strategies

- Two key migration strategies
  - Thread migrates itself
  - Separate manager thread migrates it

- State of the migrating thread influences the mechanisms & cost
  - Thread core affinity mask is updated
  - For running thread, may involve kernel run queues & scheduler
  - Self migration *always* entails migrating a running thread

Case 1: a running thread modifies its own affinity

Case 2: a separate manager thread modifies a *running* thread’s affinity

Case 3: a separate manager thread modifies a *sleeping* thread’s affinity
Linux Performance: Migration Costs

Insight 3

- Every strategy risks a non-negligible thread migration cost
- Motivates binding task threads into core-specific thread pools
- Motivates an ORB architecture with a separate manager thread (next)

![Graphs showing migration delays](image)

- Self migration (~ 16 to 45 µs)
- Manager migrates running thread (~ 18 to 36 µs)
- Manager migrates sleeping thread (~ 4 to 10 µs)
Conventional Middleware Architecture

- Traditional single-CPU approach benefits from leader/followers to *reduce costly hand-offs*
  - E.g., TAO, nORB
- However, multiple cores increase *risk of migration*

1. Leader invokes TA (and AC) for task
2. Picks new leader
3. New leader may need to move old
4. Old leader runs the task (on the appropriate core)
MC-ORB Middleware Architecture

- In contrast, MC-ORB’s threading architecture *leverages* hand-offs to avoid thread migrations
  - Key trade off: copying/locking costs vs. migration costs

1. Request is queued
2. Manager thread reads requests in priority order
3. Invokes TA w/AC
4. Manager picks thread from pool
5. Thread runs task
To gauge performance costs of our middleware architecture we examined four key features

- Allocate on same vs. other core (as manager thread)
- Thread available vs. migration needed
- Reallocation is vs. is not required to allocate task
- New task is admitted vs. rejected

We evaluated our middleware architecture both with (MC-ORB) and without (MC-ORB*) rejection

- MC-ORB* compared to nORB (designed for uniprocessors)
- Varied utilization granularity & magnitude (10 task sets)
- We measured how many of the task sets missed a deadline
### Overheads for MC-ORB’s Extensions (μs)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Minimum</th>
<th>Mean</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>55</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>58</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>64</td>
<td>121</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>235</td>
<td>289</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
<td>50</td>
<td>107</td>
</tr>
</tbody>
</table>

**Scenarios used for Overhead Evaluation**

1. New task on same core as manager
2. New task on different core (similar cost to 1)
3. (Sleeping) thread moved from other core to run new task
4. (All) running tasks reallocated to make room for new task
5. The new task is rejected (low cost, but it’s pure overhead)
### Fraction of Workloads w/ Deadline Misses

<table>
<thead>
<tr>
<th>Total Utilization</th>
<th>ORB</th>
<th>Balance Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>1.4</td>
<td>nORB</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>MC-ORB*</td>
<td>0</td>
</tr>
<tr>
<td>1.5</td>
<td>nORB</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>MC-ORB*</td>
<td>0</td>
</tr>
<tr>
<td>1.6</td>
<td>nORB</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>MC-ORB*</td>
<td>0.3</td>
</tr>
</tbody>
</table>

- With rejection, >94% of tasks were *admitted* by MC-ORB and all admitted tasks met all deadlines.

- Without rejection (where + shows need for AC) MC-ORB*
  - Worked better with less balanced workloads
  - Outperformed nORB in 6 cases (green)
  - Performed the same as nORB in 4 cases (grey)
  - Underperformed nORB in 2 cases (red)
Concluding Remarks

- **COTS OS evaluations**
  - Measurement on specific target platforms is crucial
  - Behaviors of hardware and OS mechanisms are important

- **Middleware architectures**
  - OS evaluations establish design trade-off parameters
  - Prior design decisions may be reversed on new platforms

- **Performance evaluations bear out our new design**
  - Even w/out admission control, MC-ORB architecture helps
  - W/ AC admitted high utilization, and met all deadlines
Paper


- Slides based on Chris Gill’s RTCSA’09 presentation.