Adaptive QoS Control for Real-Time Systems

Chenyang Lu
CSE 520S

Challenges for Real-Time Systems

- Classical real-time scheduling theory relies on accurate knowledge about workload and platform.

New challenges under uncertainties

- Maintain robust real-time properties in face of:
  - unknown and varying workload
  - system failure
  - system upgrade
- Certification and testing of real-time properties of adaptive systems

Challenge 1: Workload Uncertainties

- Task execution times
  - Heavily influenced by sensor data or user input
  - Unknown and time-varying
- Disturbances
  - Aperiodic events
  - Resource contention from subsystems
  - Denial of Service attacks
- e.g., SCADA for power grid management, total ship computing environment

Challenge 2: System Failure

- Only maintaining functional reliability is not sufficient.
- Must also maintain robust real-time properties!

Example: nORB Middleware

<table>
<thead>
<tr>
<th>Application</th>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker thread</td>
<td>Conn. thread</td>
<td>Operation Request Lanes</td>
</tr>
<tr>
<td>CORBA Objects</td>
<td>T1: 2 Hz T2: 12 Hz</td>
<td>Timer thread</td>
</tr>
<tr>
<td>Priority queues</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Challenge 3: System Upgrade

- Goal: Portable application across HW/OS platforms
  - Same application “works” on multiple platforms
- Existing real-time middleware
  - Support functional portability
  - Lack QoS portability: must manually reconfigure applications on different platforms to achieve desired QoS
  - Profile execution times
  - Determine/Implement allocation and task rate
  - Test/analyze schedulability
  - Time-consuming and expensive!
Challenge 4: Certification

- Uncertainties call for adaptive solutions.
- But...
- Adaptation can make things worse.
- Adaptive systems are difficult to test and certify.

![An unstable adaptive system](image)

Adaptive QoS Control

- Develop software feedback control in middleware
  - Achieve robust real-time properties for many applications
  - Apply control theory to design and analyze control algorithms
  - Facilitate certification of embedded software

Sensor/human input? Disturbance?

Applications

- Adaptive QoS Control Middleware
- Drivers/OS/HW?
- Available resources? HW failure?

Maintain QoS guarantees
- w/o accurate knowledge about workload/platform
- w/o hand tuning

Adaptive QoS Control Middleware

- FCS/nORB: Single server control
- FC-ORB: Distributed systems with end-to-end tasks

Feedback Control Real-Time Scheduling

- Developers specify
  - Performance specs
    - CPU utilization = 70%; Deadline miss ratio = 1%.
  - Tunable parameters
    - Range of task rate: digital control loop, video/data display
    - Quality levels: image quality, filters
    - Admission control
  - FCS guarantees specs by tuning parameters based on online feedbacks
    - Automatic: No need for hand tuning
    - Transparent from developers
    - Performance Portability!

A Feedback Control Loop

![A Feedback Control Loop](image)

The FC-U Algorithm

- $U_s$: utilization reference
- $K_c$: control parameter
- $R(0)$: initial rate

1. Get utilization $U(k)$ from Utilization Monitor.
2. Utilization Controller:
   - $B(k+1) = B(k) + K_c(U_s - U(k))$ /* Integral Controller */
3. Rate Actuator adjusts task rates
   - $R_i(k+1) = [B(k+1)/B(0)]R_i(0)$
4. Inform clients of new task rates.
The Family of FCS Algorithms

- **FC-U** controls utilization
  - Performance spec: \( U(k) = U_s \)
  - Meet all deadlines if \( U_s \) ≤ schedulable utilization bound
  - Relatively low utilization if utilization bound is pessimistic
- **FC-M** controls miss ratio
  - Performance spec: \( M(k) = M_s \)
  - High utilization
  - Does not require utilization bound to be known a priori
  - Small but non-zero deadline miss ratio: \( M(k) > 0 \)
- **FC-UM** combines FC-U and FC-M
  - Performance specs: \( U_s, M_s \)
  - Allow higher utilization than FC-U
  - No deadline misses in “nominal” case
  - Performance bounded by FC-M

Control Analysis

- Rigorously designed based on feedback control theory
- Analytic guarantees on:
  - Stability
  - Steady state performance
  - Transient state: settling time and overshoot
  - Robustness against variation in execution time
- Do not assume accurate knowledge of execution time


Dynamic Response

Stability

Steady state error

Transient State

Steady State

Settling time

Time

FCS/nORB Architecture

Implementation

- Running on top of COTS Linux
- Deadline Miss Monitor
  - Instrument operation request lanes
  - Time-stamp operation request and response on each lane
- CPU Utilization Monitor
  - Interface with Linux /proc/stat file
  - Count idle time: “Coarse” granularity: jiffy (10 ms)
  - Only controls server delay

Offline or Online?

- Offline
  - FCS executed in testing phase on a new platform
  - Turned off after entering steady state
  - No run-time overhead
  - Cannot deal with varying workload
- Online
  - Run-time overhead (actually small...)
  - Robustness in face of changing execution times
Set-up

- OS: Redhat Linux
- Hardware platform
  - Server A: 1.8GHz Celeron, 512 MB RAM
  - Server B: 1.99GHz Pentium 4, 256 MB RAM
  - Same client
  - Connected via 100 Mbps LAN
- Experiment
  1. Overhead
  2. Steady execution time (offline case)
  3. Varying execution time (on-line case)

Server Overhead

- Overhead: FC-UM > FC-M > FC-U
- FC-U increases CPU utilization by <1% for a 4s sampling period.

Performance Portability

Steady Execution Time

- Same CPU utilization (and no deadline miss) on different platforms w/o hand-tuning!

Steady-state Deadline Miss Ratio

Server A

- FC-M enforces miss ratio spec
- FC-U, FC-UM causes no deadline misses

Steady-State CPU Utilization

Server A

- FC-U, FC-UM enforces utilization spec
- FC-M achieves higher utilization

Robust Guarantees

Varying Execution Time

- Same CPU utilization and no deadline miss in steady state despite changes in execution times!
Tolerance to Load Increase

- **Surprise**: server crashes under FC-M when execution time increases
  - FCS/nORB threads run at real-time priority
  - Kernel starvation when CPU utilization reaches 100%
- **Tolerance margin** of load increase
  - FC-U, FC-UM: margin = \( \frac{1}{U_s} - 1 \)
    - \( U_s=70\% \) $\Rightarrow$ Server can tolerate \( (1/0.7-1) = 43\% \) of increase in execution time
  - FC-M: small and "unknown" margin
    - Inappropriate middleware-level service when execution time can increase unexpectedly

Summary of Experimental Results

- FCS algorithms enforce specified CPU utilization or miss ratio in steady state
  - Experimental validation of control design and analysis of FCS
- **Performance Portability**: FCS/nORB achieves the same performance guarantee when
  - platform changes
  - execution time changes (within tolerance margin)
- **Overhead** acceptable $\Rightarrow$ FCS can be used online

Summary: FCS/nORB

- FCS/nORB supports **robust, performance-portable** real-time software
  - Program application once $\Rightarrow$ runs on multiple platforms with robust performance guarantees
  - FCS/nORB 1.0 release: [http://deuce.doc.wustl.edu/FCS_nORB](http://deuce.doc.wustl.edu/FCS_nORB)
- Next: FC-ORB
  - Handle end-to-end tasks
  - Fault tolerance

Reference