Dynamic CPU Management for Real-Time, Middleware-Based Systems

Eric Eide
John Regehr

Tim Stack
Jay Lepreau

University of Utah, School of Computing

May 27, 2004
Monoliths → Modules

- **Old: Monolithic RT systems**
  - Made by one organization
  - One-off products
  - Closed and static
  - Hard to extend and integrate

- **New: Modular RT systems**
  - Made by many organizations
  - Product lines and families
  - Open and dynamic
  - Hard to compose and control
Modularity: Middleware

- Common backplane
- Relieve developers of tedious low-level detail
- Fewer bugs, shorter time to market
- Reusable across multiple products, product families
Modularity: Separate RT

- Separate application logic from RT logic
  - Reuse of task parts
  - Modularize RT parts for understandability
- Separation enables remodularization
- **But... how?**
Modularity: Specifying RT

- Need composability
  - Reservations are composable, but not enough

- Need adaptability
  - …for complex parts
    - data-, mode-, or configuration-dependent demand
  - …for open systems
    - unknown agents and resources before deployment
  - …because prediction is hard
Idea: “CPU Broker”

- Service for managing processor resources on a host
- Mediates between RT tasks and RT OS
- Broker is a negotiator, not a scheduler
Idea 1: Separate Concerns

- Separate application logic from RT logic
- Separate per-task and global decision makers
- Separate negotiation from scheduling
- Manage both middleware-based and other applications
Idea 2: Dynamic/Adaptive

- Dynamic monitoring of applications
- Changing task sets
- External inputs to the broker at run time
- Set and change broker configuration at run time
Idea 3: Open Framework

- Parts are objects
  - Plug-in per-task “advocates”
  - Plug-in global policies
- Connect to other QoS management systems
Contributions

- Architecture addressing critical software engineering challenges
  - …moving from reservations to negotiations
  - …supporting more modular RT abstractions
- Implementation on COTS MW and RTOS
- Evaluation
  - …with synthetic RT applications
  - …with a distributed RT military application
Related Work

- Our focus: address SE challenges
- Build on previous work
  - Feedback-driven scheduling
    - QoS Manager [Abeni and Buttazzo, ’99]
  - RT middleware
    - RT CORBA [OMG], feedback [Lu et al., ’03]
  - MW-based QoS architectures
    - DQM [Brandt et al., ’98], QuO [Zinky et al., ’97]
Design Overview

- Within the broker:
  - Advocates
  - Policies
  - Scheduler proxy

RTOS
Advocates

- Per-application adaptation
- Request CPU resources for a task
- Chain to build up complex behaviors
- Goal: match expected task demand
Topmost Advocate

Task

Advocate

- Input CPU consumed
  - “status”
- Output request for a periodic reservation
  - “advice” (C, P)
- Predict future need
  - …library of advocates
  - …or, write your own
  - can collocate with task

Task data (e.g., mode)

OS data (e.g., getrusage)

status

status and advice
Subsequent Advocates

- Modify advice or perform side effect
  - cap request (min/max)
  - talk to other MW
  - watchdog timer
- Chaining allows
  - reuse of advocates
  - dynamic configuration
  - … and reconfiguration

Advocate

status, advice

status, new advice

status, new advice

…

Other MW, e.g. Load Predictor
Policies

- Global adaptation
- Collect advice from advocates
- Request reservations
- Propagate information back to advocates
- Goal: negotiate to resolve contention
Encapsulate negotiation
  - ...hard or soft RT
Multiple policies via partitions
Setting up policies
  - ...use library policies
  - ...or, write your own
Set/reset policy data at run time, e.g.
  - ...importances, weights
Putting It All Together

- **Tasks**: status
- **Advocates**: predictions
- **Policies**: allocations
- **Scheduler proxy**
- **RTOS**: reservations
- **Propagate changes**
Implementation

- COTS RT scheduling and accounting
  - *TimeSys Linux*
- Open architecture, dynamic control
  - CORBA
- Separate application and RT logic
  - QuO
  - *TimeSys Linux*
TimeSys Linux

- **Core services**
  - Reservation-based scheduling
  - High-resolution CPU usage timers

- **Good abstractions**
  - Multiple threads can share one reservation
  - Reservations can be inherited
  - Thread→reservation associations can be created and manipulated “externally”

- …allow easy monitoring and manipulation
Advocates and policies: CORBA objects
- Easy to assemble
- Easy to invoke at run time
- In one process or multiple processes

Basis for custom advocates and policies
- Dynamic loader for custom objects

Basis for connecting to MW-based tasks

...allows open and dynamic architecture
QuO

- Apps not designed *for* the CPU Broker

- Use QuO delegates
  - Improved integration with MW-based tasks
    - Sync’ed with task cycle
    - Customizable
  - … but, small source changes required

Real-Time App
  - Object Implementation
  - QuO Delegate

CPU Broker

*Periodic CORBA requests*
Process Advocate

- Need to manage non-MW applications, too
- Use “proc_advocate”
  - Transparent to task
    - No MW required
    - Entirely reusable
  - …but, less access to task state
    - E.g., period

Real-Time App

CPU Broker

proc_advocate

TimeSys Linux
Using the CPU Broker

- **Scripted, interactively**
  - Command-line tool
  - “set priority of task mplayer to 5”

- **Programmatically**
  - E.g., with QARMA [Gillen et al., 2004]
Evaluation

- Overhead
- *Synthetic applications (correct operation)*
- UAV application

- Experiments performed in Utah’s Emulab
  - 850 MHz Pentium III, 512 MB RAM
  - TimeSys Linux/NET 3.1.214, Red Hat 7.3
Measured Overhead

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Monitor+Broker CPU Time (μs)</th>
<th>Monitor Only Real Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way QuO delegate</td>
<td>1742</td>
<td>1587</td>
</tr>
<tr>
<td>1-way QuO delegate</td>
<td>1716</td>
<td>660</td>
</tr>
<tr>
<td>In-broker proc. advocate</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

“load” “latency”

- Reasonable and small for many RT applications
- Further optimizations possible
UAV Military Simulation

- Distributed application
  - Soft real-time
- Broker applied at ATR
  - Java process
  - Multi-threaded
  - Irregular CPU demand
- Goals
  - Ensure ATR meets deadlines
  - Allow high system utility
Broker Extension

- Custom advocate for ATR
  - Predict GC cycles from recent CPU demand
- Other broker objects from our library
Custom advocate accurately predicts demand, allowing high system utility
## Resilience to CPU Load

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unloaded, Baseline</th>
<th>CPU Load</th>
<th>CPU Load, With Broker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames processed</td>
<td>432</td>
<td>320</td>
<td>432</td>
</tr>
<tr>
<td>Avg. FPS</td>
<td>1.84</td>
<td>1.32</td>
<td>1.81</td>
</tr>
<tr>
<td>Min. FPS</td>
<td>1.67</td>
<td>0.45</td>
<td>1.11</td>
</tr>
<tr>
<td>Max. FPS</td>
<td>2.00</td>
<td>2.01</td>
<td>1.99</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>0.09</td>
<td>0.34</td>
<td>0.09</td>
</tr>
<tr>
<td>Alerts received</td>
<td>76</td>
<td>50</td>
<td>76</td>
</tr>
<tr>
<td>Avg. latency</td>
<td>127.67</td>
<td>1560.44</td>
<td>325.72</td>
</tr>
<tr>
<td>Min. latency (ms)</td>
<td>101.00</td>
<td>362.00</td>
<td>145.00</td>
</tr>
<tr>
<td>Max. latency (ms)</td>
<td>193.00</td>
<td>3478.00</td>
<td>933.00</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>33.46</td>
<td>961.62</td>
<td>153.60</td>
</tr>
</tbody>
</table>
Conclusions

- Dynamic RT systems face critical design-time and run-time challenges
- Our CPU Broker successfully addresses many of these challenges
  - specifications: separated and consolidated
  - dynamic: negotiations atop reservations
  - open framework: extension and integration
- → More modular and understandable RT
Open Source

- CPU Broker available online
- http://www.cs.utah.edu/flux/alchemy/