Achieving Bounded End-to-End Latencies with Real-time Linux and Realtime CORBA

Gautam H. Thaker
Patrick J. Lardieri
Donald Krecker
Keith O’Hara
Michael Price
Chuck Winters

LM Advanced Technology Labs
Camden, NJ
{gthaker,plardier}@atl.lmco.com
Purpose and Outline

■ Purpose
  — Experimental work to apply distributed scheduling theory to RT-CORBA-based DRE systems on (Real-time) Linux platforms

■ Outline
  — Research Vision
  — Evaluation of Linux QoS mechanism
  — Evaluation of QoS mechanisms suitability for distributed realtime scheduling
    ■ Comparison between theory, simulation, and testbed
Answers Sought

- Can we predict worst case end to end latencies for a set of distributed task chains executing on a distributed system?
- Can we realize such systems using RT CORBA on appropriate RT OS?
- How well our predicted worst case end to end latencies match measured values?
- What can we do to close the gap if these do not agree?
Research Vision
**Research Challenge**

- **Grand Architectural Vision for a standards compliant COTS-based infrastructure that enables predictable systems**
  - Focus on bounded time operations
  - Multi-property problem extension is more challenging

- **Critical enabling technologies**
  - Middleware (includes QoS and networks)
  - Task and network scheduling theory
  - System and software engineering methods and tools

*Challenge is essentially the problem of distributed resource management (specification and allocation)*
ATL Research Focus

- Architectural vision and associated research problems
  - Resource-requirements language
  - Performance-requirements language
  - Resource-capacity language
  - QoS mechanisms

- Holistic “end-to-end system” resource management

Supports proactive and reactive Resource-management approaches
Evaluation of Linux QoS Mechanism
Experimental approach

- Simple test design
- Focus on worst case behavior
- Establish upper bounds
- Conduct large sample tests
- Verify against simulations
- Collect comprehensive data
- Share results and work with vendors/technologists to explain and improve performance

ATL has conducted over 2000 experiments and collected results on its QoS Technology website
Measuring Scheduling Jitter

1  for i = 1 to N
2    getTime(before)
3    nanosleep(period)
4    getTime(after)
5    record(after - before)

System Conditions

— No System Load
— Competing CPU intensive non-realtime processes
— Competing I/O intensive non-realtime processes
Linux 2.4.7

- Scheduler:
  - O(n) -- variable time and not scalable
  - 100 priorities (0 - 99)

- Kernel Preemptibility:
  - Not preemptible
  - No protection for kernel or user priority inversion
  - Kernel routines are not schedulable
  - Low resolution nanosleep() - clock granularity

---

**Linux v2.4.7 Periodic Test - No Load**

**System:** Pentium II 450 MHz, 256M RAM, SCSI

**[1.7.2002.kjo]**

**min:** 19848 **max:** 20153 **mean:** 20000 **var:** 91.0841 **n:** 100000
Linux 2.5.20

Scheduler

- O(1) – constant time and scalable
- SMP processor affinity
- 100 priorities (0 - 99)

Kernel Preemptibility

- Small regions of non-preemptibility marked by spinlocks
- No protection for kernel or user-land priority inversion

Kernel routines are not schedulable

Low nanosleep() resolution due to clock granularity
Timesys Linux/GPL

Scheduler
   O(1) – constant-time and scalable
   Variable number of priorities (4 - 32,768)

Kernel Preemptibility
   Fully preemptible using mutexes to guard critical sections
   Protection against kernel-level priority inversions via priority inheritance

Kernel routines are schedulable

Low resolution nanosleep() due to clock granularity
Timesys Linux/Real-time

Scheduler
  O(1) – constant-time and scalable
  Variable number of priorities (4 - 32,768)

Kernel Preemptibility
  Fully preemptible using mutexes to guard critical sections
  Protection against kernel and user-land priority inversions via priority inheritance

Kernel routines are schedulable

High resolution nanosleep() using ktimer kernel module
ATL QoS Performance Evaluation

Middleware Comparator Analysis Tool
Evaluation of QoS Mechanisms Suitability for Distributed Realtime Scheduling
**Problem Introduction**

- **Multiple, interconnected nodes**
- **Multiple task chains**
  - Each task has its own period and deadline
  - Each task consists on “n” subtasks
  - Completion of subtask(j) signals release of subtask(j+1)
- **Notation:**
  - task(task#, subtask#)
  - cpu(task#, subtask#) = execution demand of subtask, etc.
Theory and Practice of End-to-End Scheduling

Sources of Techniques
- Scheduling Literature
- ATL Innovations

Types of Experiments
- Scheduling Analysis Algorithms
- Discrete Event Simulations
- RT System Prototypes (OS and MW)

Bounds Differences
- Theoretical
- Modeled
- Oversimplification
- Measured

Types of Results
- Comparisons
- Pitfalls
- Suggestions

Research Goals
- RT System Design Guidance
- RT System Customer Confidence

Test Cases:
- Random Workloads from J. Sun’s Thesis
- ATL-Developed Configurations
- Prototype C4ISR Problem
Experimental Evaluations

- **Uni-processor case**
  - Four periodic tasks scheduled by different techniques
    - application critical priority order,
    - rate monotonic,
    - deadline monotonic

- **Distributed (three processor) case**
  - Three periodic tasks; each task is a chain consisting of three subtasks
    - proportional deadline monotonic
Uni-Processor, Priority Scheduling

Comparing theory, simulation and implementation for worst case execution times for task chains for 4 tasks/1 cpu test. criticality monotonic. 2002-05-06 gthaker@atl.lmco.com

Application Critical
Monotonic
Prioritization:
All deadlines not met

OK (reduced margin)
Not-OK
OK (with Margin)
Uni-Processor, Rate Monotonic Scheduling

Rate Monotonic Prioritization:
All deadlines not met
Uni-Processor, Deadline Monotonic Scheduling

Comparing theory, simulation and implementation for worst case execution times for task chains for 4 tasks/1 cpu test. Deadline monotonic. 2002-05-06 gthaker@atl.imco.com

Deadline Monotonic Prioritization:
All deadlines met
Multi-processor Test Case

Node 0:
- task(0,0)
- task(0,2)
- task(2,0)
- task(2,2)

Node 2:
- task(1,0)
- task(1,2)

Node 3:
- task(0,1)
- task(2,1)
- task(1,1)

Task 0
- period = 104
- deadline = 88
- cpu(0,0) = 7
- cpu(0,1) = 8
- cpu(0,2) = 25

Task 1
- period = 30
- deadline = 30
- cpu(1,0) = 9
- cpu(1,1) = 5
- cpu(1,2) = 11

Task 2
- period = 54
- deadline = 54
- cpu(2,0) = 10
- cpu(2,1) = 6
- cpu(2,2) = 14

• TCP based tests - application manipulates priorities directly
• RT-CORBA based tests extend examples/RTCORBA/Activity and use thread pools, lanes and bands
Summary of Multi-Processor Results

Currently working to overcome this case

OK
Examining Task 0 and 2 Completion Times

There is minimal to modest disagreement due to communication costs being ignored in simulation for the time being.
Conclusions

■ Scheduling Theory
  — Advances in theoretical estimation of worst case end-to-end response times reduce pessimism
  — Experiments suggest additional enhancements possible (future work)
  — Can be usefully applied to small and medium scale systems

■ QoS mechanisms
  — Real-time Linux variants sufficiently enforce fixed priority scheduling throughout the OS for hard real-time systems

■ RT-CORBA
  — RT-CORBA mechanisms (thread pools, lanes, etc) sufficiently enforce fixed priority scheduling for hard real time systems
  — Careful configuration required to prevent priority inversions
Future Work

- **Synchronization Protocols**
  - Release Guard (from the literature)
- **Analytical Techniques**
  - ATL developed extensions that reduce pessimism
  - Further explore off-line, on-line & scaling issues
- **Integration of these improved techniques with TAO**
  - Real-time scheduling service (Kokyu)
    - Automatic generation of all svc.conf.x files with proper parameters for lanes, bands etc.
    - Explore use of these techniques in CORBA Components and Model Driven Architecture
- **Integrate testbed with our SCIOP (GIOP over SCTP) work**
- **Compare and contrast with dynamic scheduling (edf etc.)**