# A Brief History of Real-Time Linux

Sven-Thorsten Dietrich

Montavista Software, Inc.

EB II Room 1226

April 12, 2006: 1:50 - 2:45

Raleigh, NC

# Real Time Overview

## Real-Time Linux Background

- Real-Time Linux Evolution
- Real-Time Linux Enablers

## Real-Time Inhibitors

- Interrupt Latency
- Kernel Locking
- Legacy Locking

## Real-Time Kernel

- Interrupt Handlers, PI Mutex
- Performance / Benchmarks
- Acceptance
- Virtualization

# Evolution of Linux

## Early Linux Not Designed for Real-Time Processing

- Early Linux (1.x Kernel) installations on retired Windows PCs
  - Old/Obsolete hardware useful under Linux due to efficiency of O/S
  - Linux outperformed Windows in reliability and uptime (still does)
- Linux Design: Fairness, Throughput and Resource-Sharing
  - Basic Unix development design principles applied in Kernel
  - Heavily (over)-loaded systems continue to make progress
  - Does not drop network connections or starve users / applications
- Fairness- and Resource-Sharing Design is Linux's Strength
  - contributed to make Linux competitive and popular in the enterprise-server and development-application environments
  - Gave rise to RedHat and others.
  - Essential to the evolution of Linux, endemic of UNIX legacy

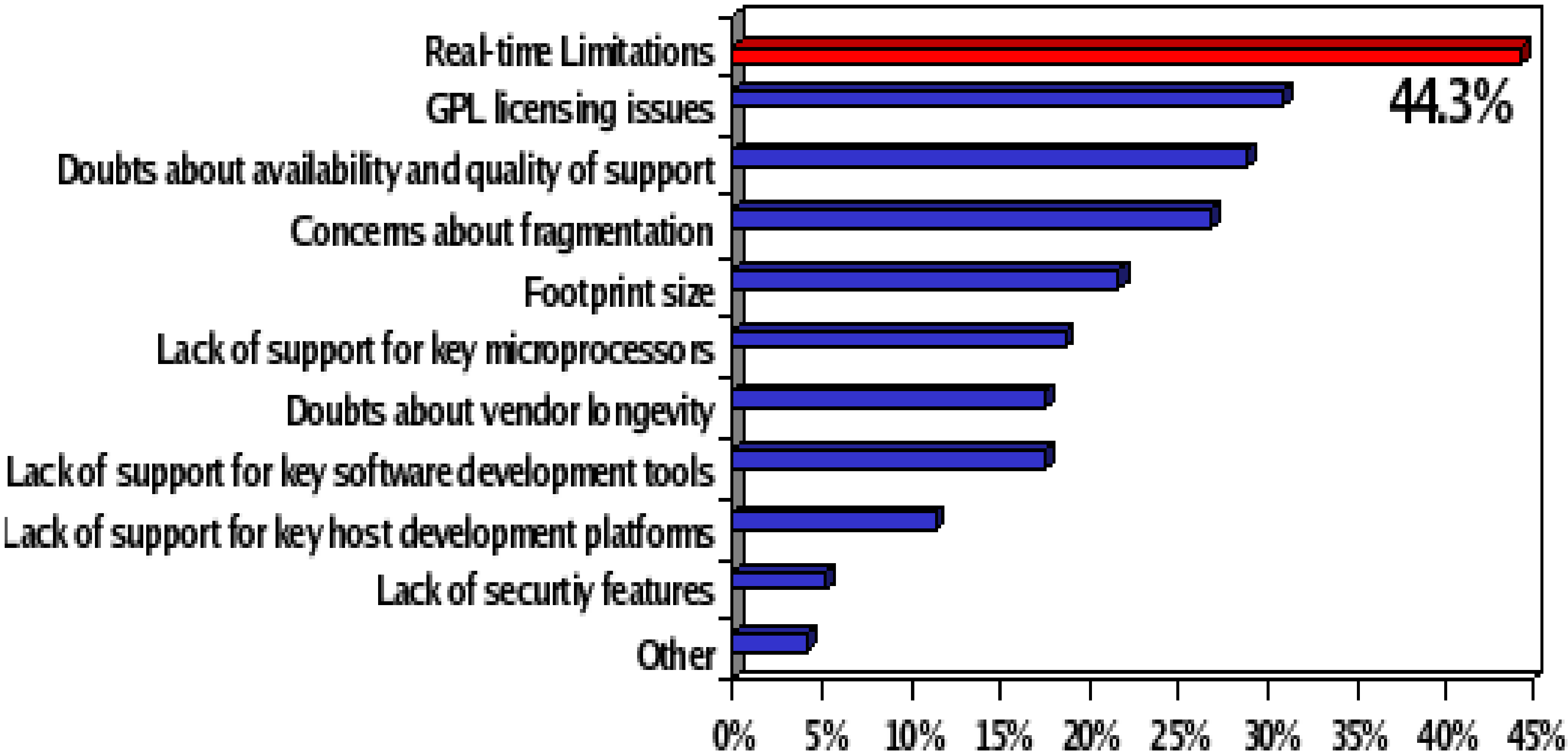
# Why Linux in Real-Time Systems?

## Not because of the Kernel’s Real-Time Performance!

- UNIX-legacy Operating Systems were designed with operating principles focused on **throughput** and **progress**
  - User tasks should not stall under heavy load
  - System resources must be shared fairly between users
- Fairness, progress and resource-sharing conflict with the requirements of time-critical applications
  - VIP vs. General Admission
- UNIX systems (and Linux) are historically not Real-Time OS
- Linux has lagged many commercial Unix's in Real-Time performance-enhancement and Real-Time capabilities
  - Solaris, LynxOS, QNX
  - SCO

# Why Real-Time in Linux Systems (Embedded)

## Most Important Factors Inhibiting Linux Adoption




***Data from VDC, "Linux's Future in the Embedded Systems Market", June 2004***

# Real-Time in Handheld & Embedded Systems

## Cost / Performance / Power / Weight Compromise

- Competitive, High-Volume, Low-margin Markets
- Maximum Feature-set, Add-ons, Responsive UI feel
- Device specs: minimal CPU & Memory & Battery Powered
- Minimal CPU = High CPU utilization
- High CPU load + Time-Critical functionality → RT specs
- Real-time Requirements will **never** be alleviated by Improvements in Hardware Performance / Efficiency
  - Software utilizing latest hardware technologies easily keep up with, and usually out-paces, advances in hardware technology
  - If you don't believe that, go shopping (for a mobile phone)

# Real-Time Linux 2.6 Enablers

## Pro-Audio Performance Requirements

- Audio Community Involved in Kernel-Preemption since 2.2
- Audio Community strongly Endorsing RT technology

## Embedded Application Domain

- Single-Chip, Mobile Applications (Wireless / Cellular Handsets)
- Predictable OS performance eliminates HW design uncertainty
  - Reliable Prototyping and Improved Product Scheduling

## Multimedia Carrier (QOS) Application Domain

- Telephony, Audio / Video / Multimedia / Home Entertainment

## Fine-Granular Preemption improves SMP scalability

- Mainstreaming of SMP Technology
  - Dual / Quad / Octa - Core Intel, AMD, PPC, Arm

# Real-Time and Linux Kernel Evolution

## Gradual Kernel Optimizations over Time

- SMP Critical sections (Linux 2.x)
- Low-Latency Patches (Linux 2.2)
- Preemption Points / Kernel Tuning (Linux 2.2 / 2.4)
- Preemptible Kernel Patches (Linux 2.4)
- Fixed-time “O(1)” Scheduler (Linux 2.6)
- Voluntary Preemption (Linux 2.6)

## In 2003-04 Linux 2.6 RT Technology Regressed

- Early Linux 2.6 Real-Time Performance was worse than 2.4 Kernel Performance
- Audio Community and others balked at moving to 2.6 Kernel Base
- What Happened?

# Real-Time Inhibitor: Critical Section Locking

## Linux 2.6 Kernel Critical Sections are Non-Preemptible

- Critical sections protect shared resources, e.g. hardware registers, I/O ports, and data in RAM
- Critical sections are shared by Processes, Interrupts and CPUs.
- Effective protection is provided by the Spin-Lock Subsystem
- Critical sections must be locked and unlocked
- Locked critical sections are not preemptible
- Linux 2.6 Kernel has 11,000 critical sections
- Exhaustive Kernel testing to identify worst-case code paths
- Labor-intensive cleanup of critical sections
- No control over 3rd party drivers
- Worst-case after cleanup still not acceptable
- Maintenance, community education, policing / regression testing

# Real-Time Inhibitor: Interrupt Handlers

## Linux 2.6 Kernel: Unbounded IRQ subsystem latencies

- Task-Preemption latency increases with hardware-interrupt load
- Interrupts cannot be preempted
- No Priorities for Interrupts
  - IRQ Subsystem always preempts tasks unconditionally
- Unbounded SoftIRQ subsystem (“Bottom Half Processing”)
  - Activated by HW IRQs (Timers, SCSI, Network)
  - SoftIRQs re-activate, iterate
- Driver-level adaptations
  - Network Driver NAPI adaption reduces D.o.S. effects of high packet loads

# Real-Time Inhibitor: Legacy Locking

## Existing Locking Subsystems are not Priority-Aware

- System semaphore
  - Counting semaphore used to wake multiple waiting tasks
  - No support for priority inheritance
  - No priority ordering of waiters
- Big Kernel Lock (BKL)
  - Originally non-preemptible, now preemptible using system semaphore
  - Can be released by blocking tasks, re-acquired upon wake-up
  - No priority-awareness, or priority inheritance for contending tasks
- RCU (Read-Copy-Update) Locks in Network subsystem
  - Read-optimized cached locking requiring race-free invalidation
- Read – Write Locks
  - Classical blocking / starvation issues with no priority awareness

# The Fully Preemptible Linux Kernel

## Dramatic Reduction in 2.6 Preemption Latencies

- Multiple Concurrent Tasks in Independent Critical Sections
- Generally Fully Preemptible → "No Delays"
  - Non-preemptible: Interrupt off paths and lowest-level interrupt management
  - Non-preemptible: Scheduling and context switching code

## Design Flexibility

- Provides Full Access to Kernel Resources to RT Tasks
- Supports existing driver and application code
- User-space Real-Time

## Optimization Flexibility

- RT Tasks designed to use Kernel-resources in managed ways can reduce or eliminate Priority-Inheritance delays

## Adequate Instrumentation

- Latency timing, latency triggers & stack tracing, histograms

# Linux Real-Time Technology Overview

## Linux 2.6 Kernel Real-Time Technology Enhancements

- Preemptible Interrupt Handlers in Thread Context
- IRQ-Disable Virtualization for Drivers
  - IRQ threads disabled without masking hardware
- Integrated Kernel Mutex with Priority Inheritance (PI)
  - Preemptible PI Mutex protects Kernel Critical Sections
- PI Mutex Substituted for Non-Preemptible Kernel (SMP) Locks
  - Big Kernel Lock (BKL) converted to PI Mutex
  - Spin-Locks converted to PI Mutex
  - Read-Write Locks converted to PI Mutex
  - RCU Preemption Enhancements
- High Resolution Timers
- User-Space Mutex
  - Robustness / Dead-Owner / Priority Queuing

# Thread-Context Interrupt Handlers

## Real-Time Solution: Interrupts in Thread Context

- Functionality of IRQ Handlers does not require IRQ context
  - no special “IRQ Mode” Instructions
  - IRQ Thread can have private stack
  - Default: IRQs run in threads
- Demote IRQ Handler Execution to Thread-based function
  - IRQ Handlers scheduled in bounded-time by O(1) scheduler
  - Inter-leaving of RT and IRQ tasks
  - Real-Time tasks at Higher Priority than IRQ handlers
- Incoming IRQ returns immediately
  - IRQ activates corresponding Handler-thread (wake_up_process)
- RT IRQs operate in Vacated IRQ execution-space
  - RT IRQs do not contend with common IRQs - Runs at IRQ Priority
  - RT IRQ Latency Predictable
  - Subject to Minimal Variation
- Promoted SoftIRQ Daemon Processes ALL Bottom-half activity
  - SoftIRQs Preemptible

# Thread-Context Interrupt Handlers

## Threaded IRQs Pros

- RT IRQs do not contend with common IRQs
- IRQ Processing does not Interfere with task execution
- Flexible priority assignment
  - can be arranged to emulate hardware-based priorities
- Interrupts run fully preemptible

## Threaded IRQs Cons

- IRQ-Thread Overhead
  - Scheduler must run to activate IRQ Threads
- IRQ Thread Latency
  - IRQs no longer running at the highest priority
  - Full task switch required to handle IRQ
  - Response-Time / Throughput tradeoff

# Priority-Inheriting Kernel Mutex

## New Kernel+Userspace Synchronization Primitive

- Fundamental RT Technology
  - Preemptible alternative to spin-locked / non-preemptible regions
  - Expands on “Preemptible Kernel” Concept
  - Spinlock typing preserved (maps spin_lock to RT or non-RT function )
- Enabler for User-space Real-Time Condition Variables & Mutexes
- Priority Inheritance
  - Eliminate Priority Inversion delays
- Priority-ordered O(1) Wait queues
  - Constant-time Waiter-list processing
  - Minimize Task Wake-Up latencies
- Deadlock Detect
  - Identify Lock-Ordering errors
  - Reveal Locking cycles

# PICK_OP: Spinlock->Mutex Function Mapping

## Preprocessor determines static function mapping

- Compile-time mapping, based on declared type

```
#define spin_lock(lock)  PICK_OP(raw_spinlock_t, spin, _lock, lock)
```

PICK_OP

```
#define PICK_OP(type, optype, op, lock) \
      do { \
          if (TYPE_EQUAL((lock), type)) \
              _raw_##optype##op((type *)(lock)); \
          else if (TYPE_EQUAL(lock, spinlock_t))     \
              _spin##op((spinlock_t *)(lock));   \
          else __bad_spinlock_type();                \
      } while (0)
```

TYPE_EQUAL

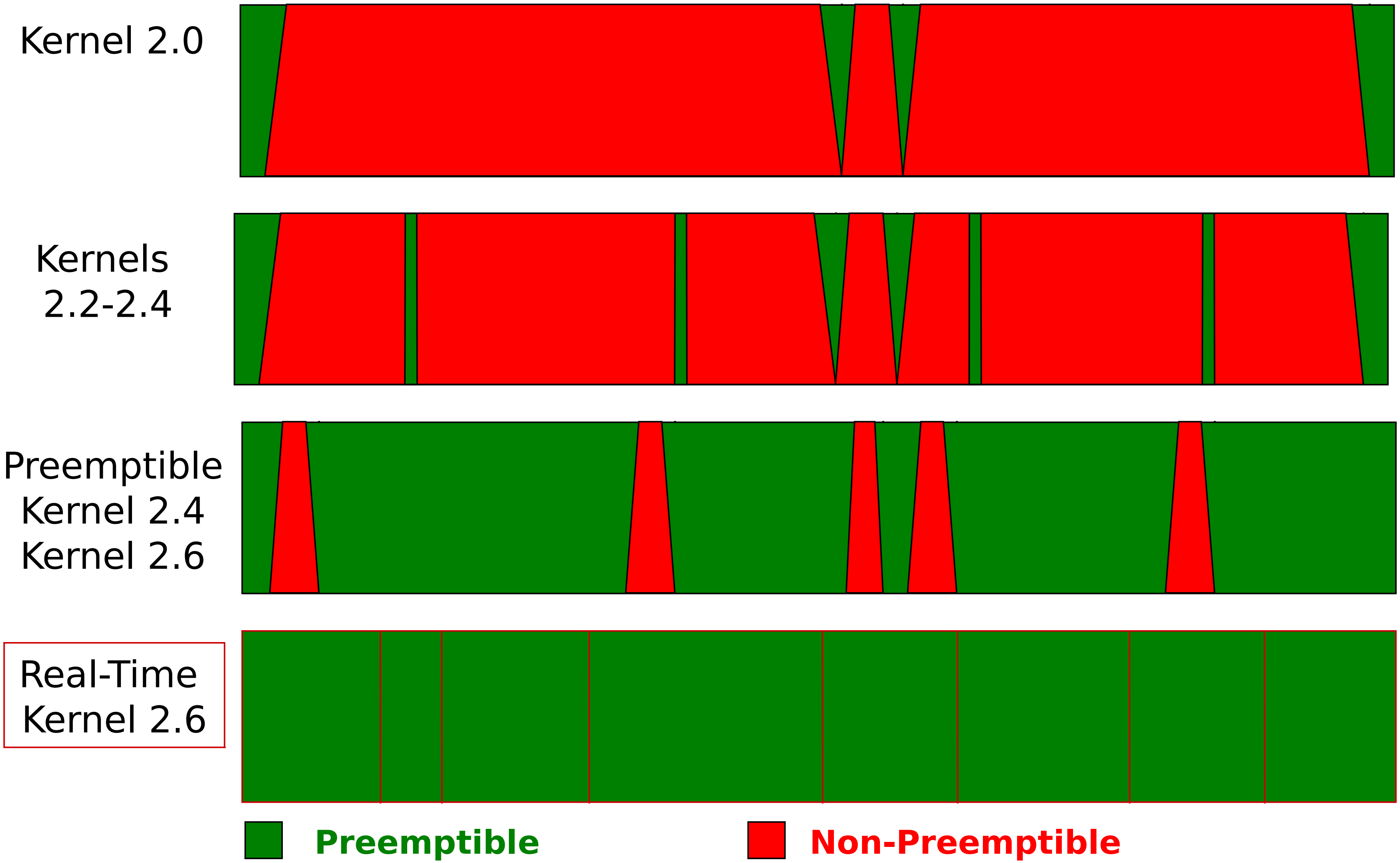
```
#define TYPE_EQUAL(lock, type) \
          __builtin_types_compatible_p(typeof(lock), type *)
```

# Real-Time Linux Kernel Evolution

## Gradual SMP-Oriented Linux Kernel Optimizations

| | |
|---|---|
| Early Kernel 1.x | No Kernel preemption |
| SMP Kernel 2.x | No Kernel preemption, “BKL” SMP Lock |
| SMP Kernel 2.2 - 2.4 | No preemption, Spin-locked Critical Sections |
| “Preempt” Kernel 2.4 | Kernel Preemption outside Critical Sections<br>Spin-locked Critical Sections |
| Current Kernel 2.6 | Kernel Preemption outside Critical Sections,<br>Preemptible “BKL”, O(1) Scheduler |
| “RT-Preempt” Kernel | Kernel Critical sections Preemptible<br>IRQ Subsystem Prioritized and Preemptible<br>Mutex Locks with Priority Inheritance<br>High-Resolution Timers |

# Kernel Evolution: Preemptible Code



# Real-Time Linux 2.6 Performance

## Real-Time Linux 2.6 Kernel Performance

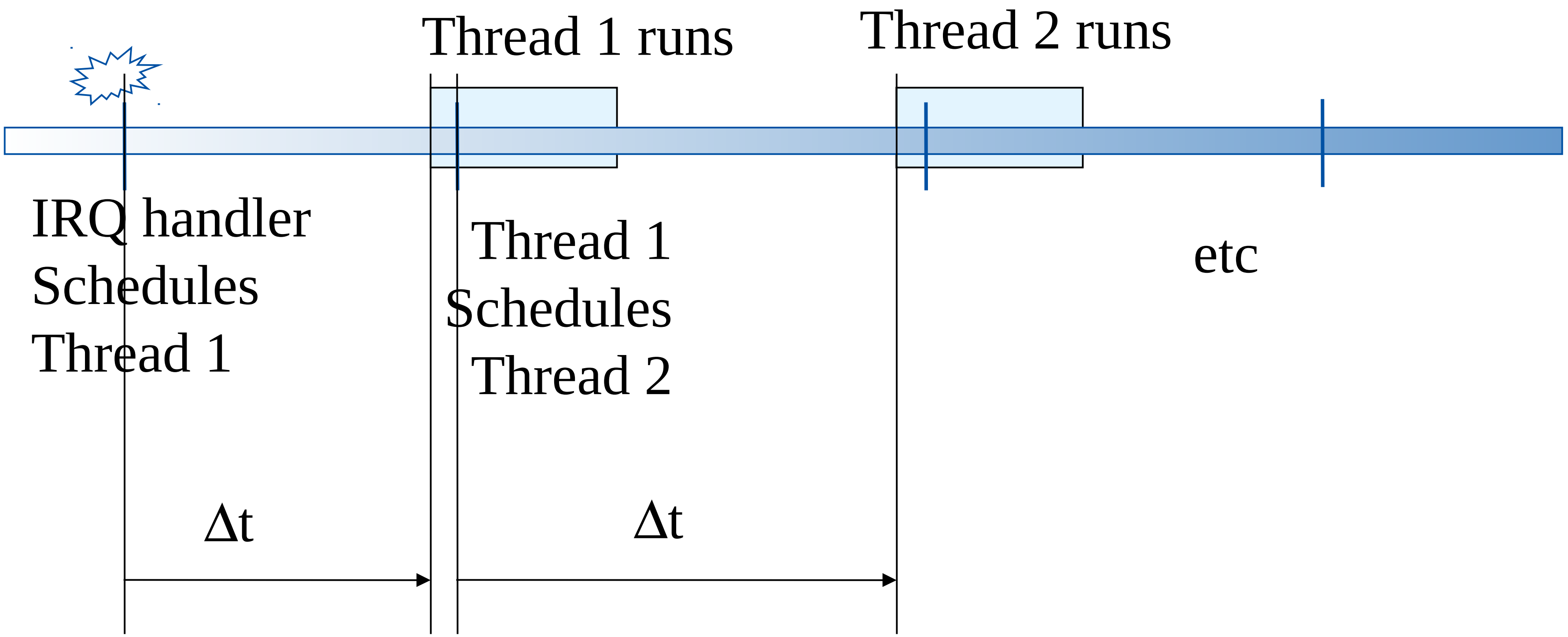
- Far exceeds most stringent Audio performance requirements
- Enables sub-millisecond control-loop response
- Enables Hard Real Time for qualified RT-aware Applications

## SMP Kernel Performance

- SMP-safe code is by definition preemptible
- Any code that allows concurrent execution by multiple CPUs, also allows context switching and therefore preemption
- Increased preemptible code surface in the Kernel also increases SMP throughput / efficiency

# FRD

## Fast Real-time Domain Measurement tool



# Benchmarks

**Target machine:**

- Intel® Celeron® 800 MHz

**Workload applied to the target system:**

- Lmbench
- Netperf
- Hackbench
- Dbench
- Video Playback via MPlayer

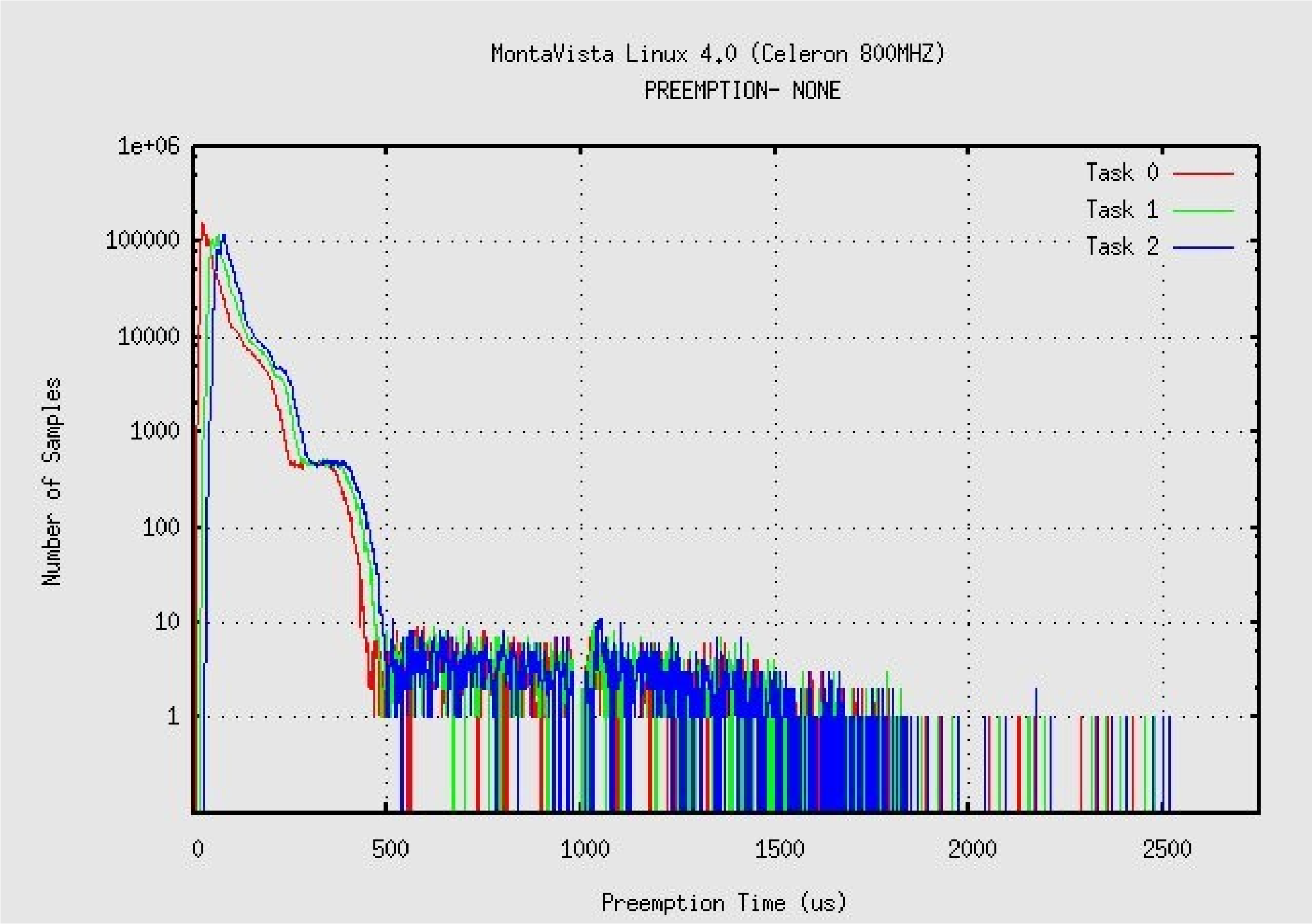
**CPU utilization during test:**

- 100% most of the time

**Test Duration:**

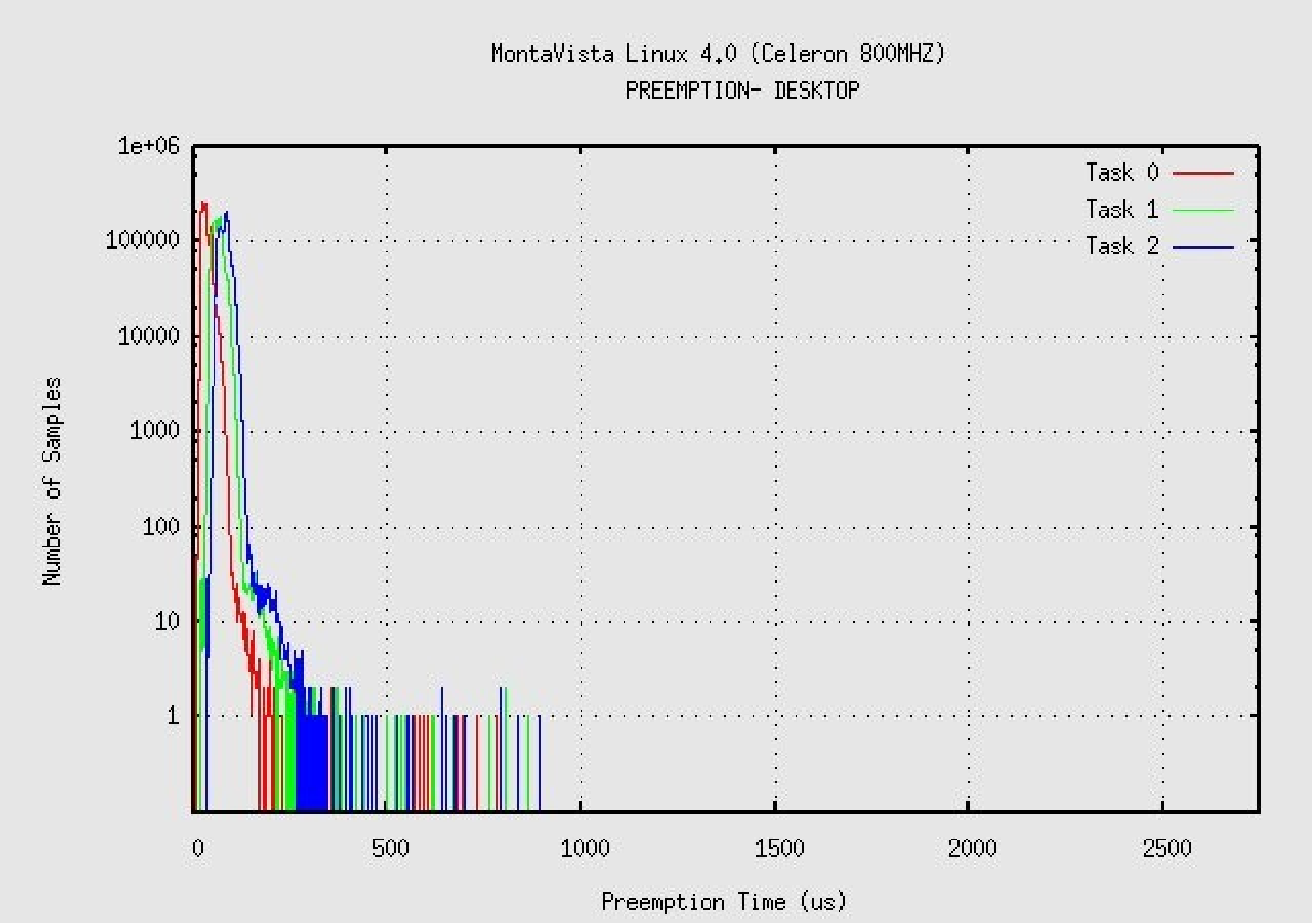
- 20 hours

# Linux 2.6 Kernel – No Preemption



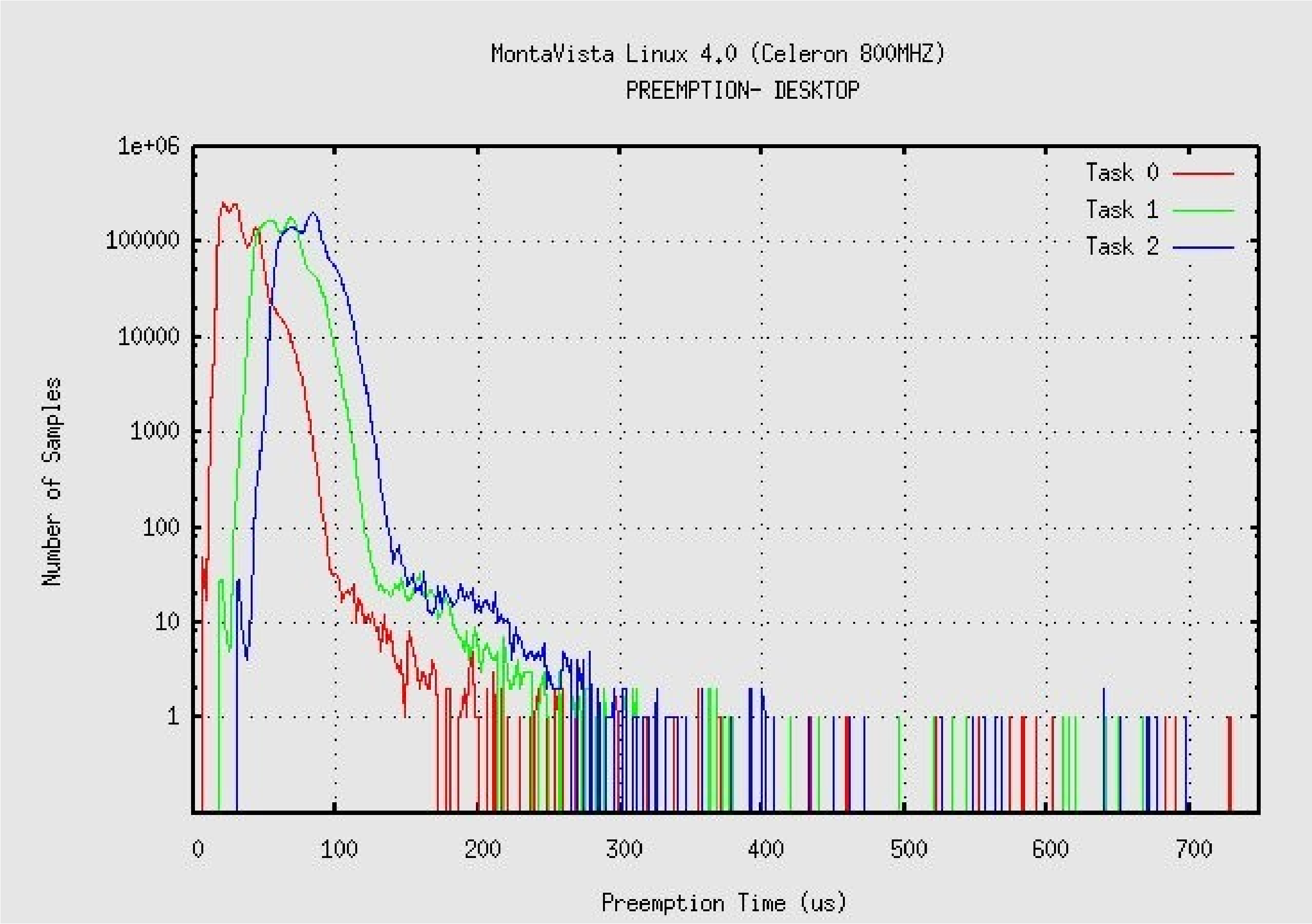

Source:

# Linux 2.6 Kernel – Preemptible Kernel



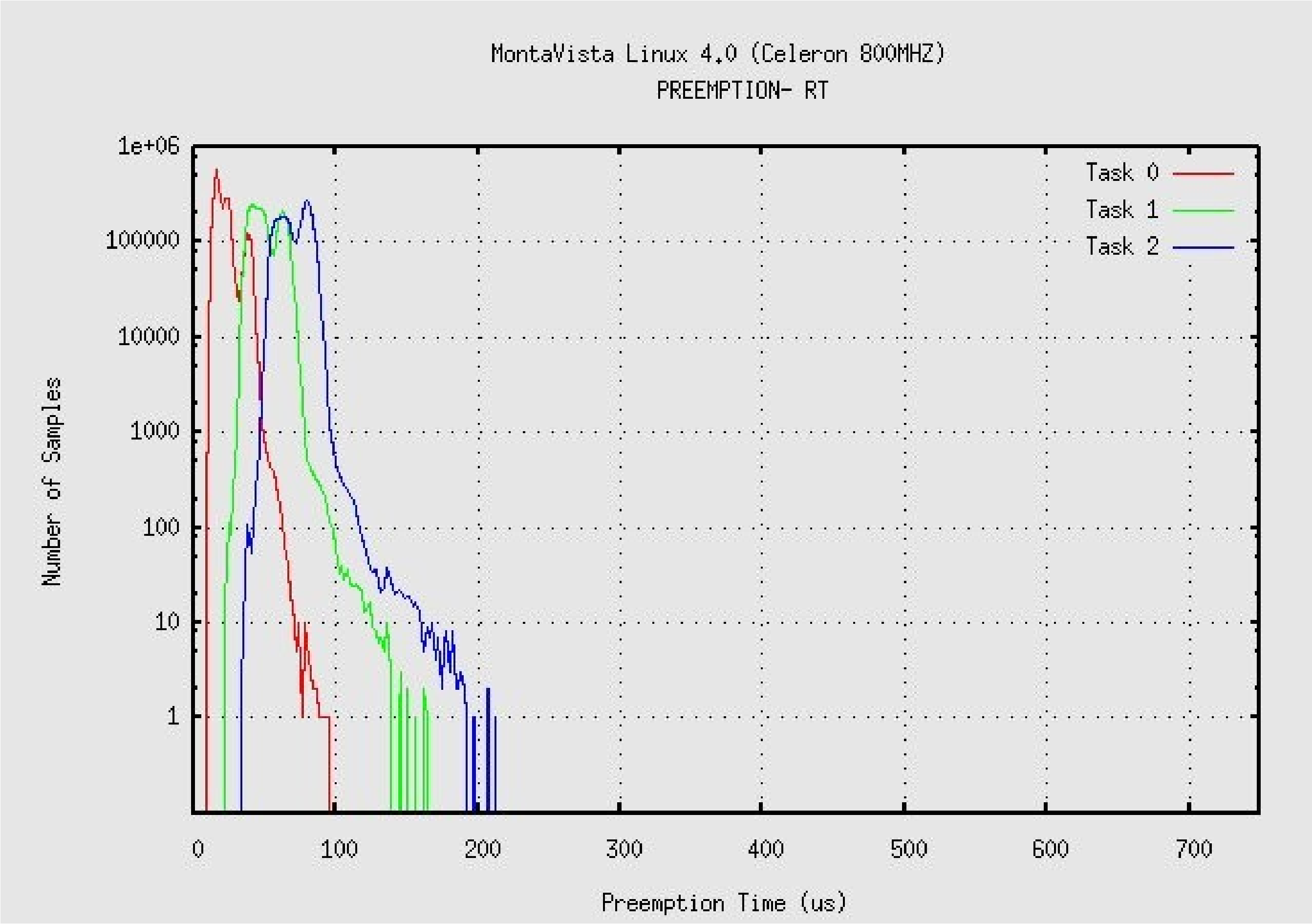

Source:

# Linux 2.6 Kernel – Preemptible Kernel (scaled)



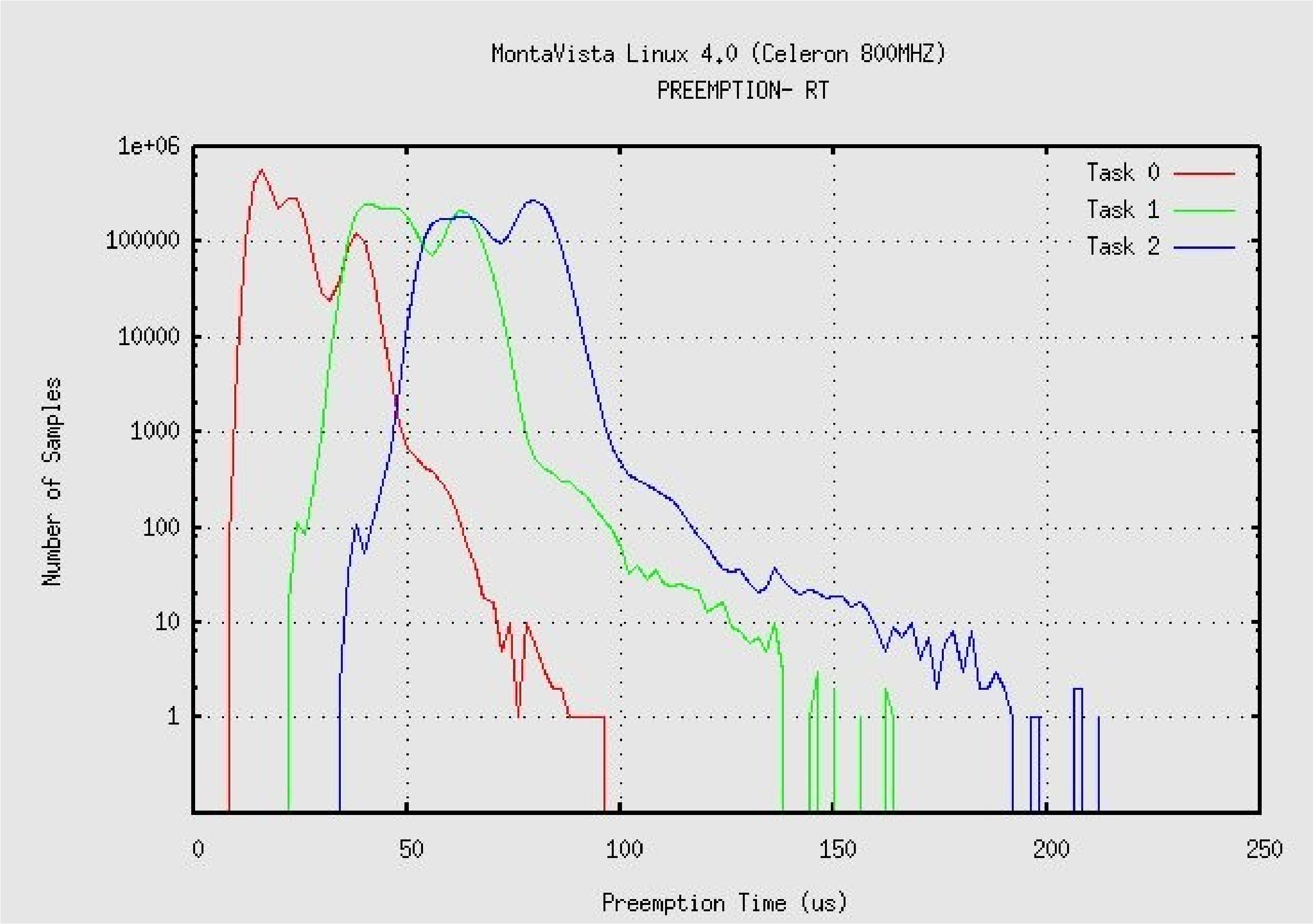

Source:

# Linux 2.6 Kernel – Real-Time Preemption




Source:

# Linux 2.6 Kernel – Real-Time Preemption (scaled)




Source:

## Real-Time Linux 2.6 IRQ Latency

### Linux-2.6.12-rc6-RT vs. Adeos / I-Pipe

| Kernel | sys load | Aver | Max | Min | StdDev |
|---|---|---|---|---|---|
| | None | 13.9 | 55.5 | 13.4 | 0.4 |
| | Ping | 14.0 | 57.9 | 13.3 | 0.4 |
| Vanilla-2.6.12-rc6 | lm. + ping | 14.3 | 171.6 | 13.4 | 1.0 |
| | lmbench | 14.2 | 150.2 | 13.4 | 1.0 |
| | lm. + hd | 14.7 | 191.7 | 13.3 | 4.0 |
| | None | 13.9 | 53.1 | 13.4 | 0.4 |
| | Ping | 14.4 | 56.2 | 13.4 | 0.9 |
| with RT-V0.7.48-25 | lm. + ping | 14.7 | 56.9 | 13.4 | 1.1 |
| | lmbench | 14.3 | 57.0 | 13.4 | 0.7 |
| | lm. + hd | 14.3 | 58.9 | 13.4 | 0.8 |
| | None | 13.9 | 53.3 | 13.5 | 0.8 |
| | Ping | 14.2 | 57.2 | 13.6 | 0.9 |
| with Ipipe-0.4 | lm.+ ping | 14.5 | 56.5 | 13.5 | 0.9 |
| | lmbench | 14.3 | 55.6 | 13.4 | 0.9 |
| | lm. + hd | 14.4 | 55.5 | 13.4 | 0.9 |

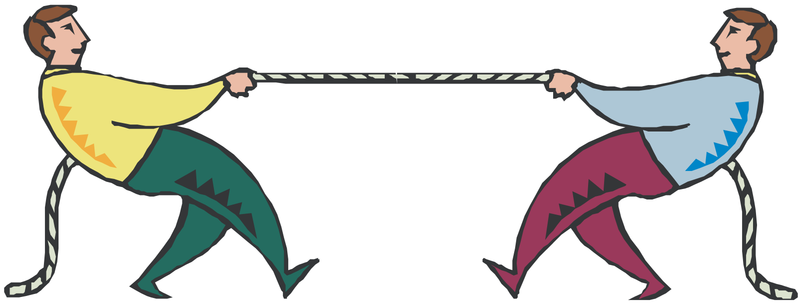
**(Source: OperSys.Com Benchmarks)**

# Real-Time Response vs. Throughput

## Efficiency and Responsiveness are Inversely Related

- Overhead for Real-Time Preemption
  - Mutex Operations more complex than spinlock operations
  - Priority Inheritance on Mutex increases task switching
  - Priority Inheritance increases worst-case execution time
  - Interrupt overhead
    - Additional Task Switching
    - Interrupt Preemption → Interrupt throughput reduction

**Throughput**



**High responsiveness**

# Real-Time Systems Design

- Design flexibility allows much better worst case scenarios
  - Task Independence
  - Kernel resources utilized in managed ways
  - Delays eliminated or reduced

# Real-Time Linux 2.6 Acceptance

## Community Status

- RT Kernel Patch: development-stable in community
- Generic Implementation facilitates Portability, Stability
  - Intel, AMD 32-bit and 64-bit
  - Arm
  - PPC

## Real-Time Linux 2.6 Technology Confidence

- RT Preemption reveals Hard-to-find SMP Bugs
  - Concurrency bugs easier to trace on UP Systems
  - Sanctioned by Kernel Summit as Constructive R & D
- Growing Community awareness of Performance Issues
- Audiophile Linux Distributions shipping RT Kernel

# Real-Time and Linux Kernel Evolution

## Today

- Real–Time Preemption
- User-Space Robust Mutex
- High Resolution Timers

## Future Innovation

- RT “awareness” extensions to Power-management subsystem
  - Quick CPU Power+Freq Ramp for RT Tasks
  - Power Level Scheduling Classes
- Virtualization / Hypervisor support for Real-Time
  - Rate-Monotonic scheduling of RT tasks in independent VMs
  - Per-VM QoS guarantees

# Linux Real-Time is Open Source

## Ongoing Real-Time Development

- Patch against current Community Kernel (2.6.16)
- Maintained by Ingo Molnar / RedHat
- Contributions from Community
- Architectures: i386, x86_64, PPC 32/64, Arm

## Download from:

http://people.redhat.com/~mingo/realtime-preempt/

# More Information

Questions?

Contact me: sven@mvista.com

MontaVista Software

- www.mvista.com