Responsive and Enforced Interrupt Handling for Real-Time System Virtualization

Hyoseung Kim*  Shige Wang†  Raj Rajkumar*

* Carnegie Mellon University
† General Motors R&D
Workload Consolidation

- Multi-core CPUs for embedded real-time systems
  - Automotive:
    - Freescale i.MX6 4-core CPU
    - NVIDIA Tegra K1 platform
  - Avionics and defense:
    - Rugged Intel i7 single board computers
    - Freescale P4080 8-core CPU

- Consolidation of real-time applications onto a single hardware platform
  - Reduces the number of CPUs and wiring harness among them
  - Leads to a significant reduction in cost and space requirements
Benefits of Real-Time Virtualization

- **Barrier to consolidation**
  - Each app. could have been developed independently by different vendors
    - Heterogeneous S/W infrastructure
    - Bare-metal / Proprietary OS
    - Linux / Android
  - Different license issues

- **Consolidation via virtualization**
  - Each application can maintain its own implementation
  - Minimizes re-certification process
  - IP protection, license segregation
  - Fault isolation
Scheduling in Virtualization

- **Two-level hierarchical scheduling structure**
  - Task scheduling and VCPU scheduling

  ![Diagram showing two-level hierarchical scheduling structure]

- **Real-time hierarchical scheduling**
  - **Budget** and replenishment period for each VCPU
  - Various budget replenishment policies (e.g., deferrable server)
Interrupt Handling in Virtualization

- VM 1
  - Task
  - Task
  - Task
  - Task
  - Guest OS
  - Task Scheduler
  - Interrupt Service Routine
  - VCPU
  - VCPU
  - Hypervisor
  - I/O device (e.g., sensor)

- VM 2
  - Task
  - Task
  - Task
  - Task
  - Guest OS
  - Task Scheduler
  - Interrupt Service Routine
  - VCPU
  - VCPU

- Interrupt-triggered task
- Virtual interrupt
- Physical interrupt
Requirements for Interrupt Handling

• R1: Responsive and bounded interrupt handling time
  – Timing penalties to interrupt handling in virtualization

• R2: Protect real-time tasks from interrupt storms
  – Task schedulability should be guaranteed

• R3: Support unmodified guest OSs
  – Many commercial RTOSs are closed-source
# Previous Work

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Priority based sched.</td>
<td>VCPU temporal isolation</td>
<td>Bounded Interrupt handling</td>
</tr>
<tr>
<td>[1]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[2]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[3]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ours</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Our Approach

• **vINT**: an analyzable interrupt handling framework for real-time system virtualization
  – Provides *responsive*, *bounded*, and *enforced* interrupt handling
  – Does not require any change to the guest OS code
    • Easily applicable to virtualizing proprietary, closed-source RTOSs

• Contributions
  – vINT framework design
  – Analysis on interrupt handling time and VCPU/task schedulability
  – Implementation and case study on the KVM hypervisor of Linux/RK
Outline

• Introduction

• vINT Framework
  – System model
  – Problems with interrupt handling
  – vINT details
  – Analysis

• Evaluation

• Conclusion
System Model (1)

- Partitioned fixed-priority scheduling for both VCPUs and tasks
  - Widely supported in many real-time OSs and hypervisors
  - e.g., OKL4, PikeOS, ...

- VCPU \( v_i: (C^v_i, T^v_i) \)
  - \( C^v_i \): Maximum execution budget
  - \( T^v_i \): Budget replenishment period

- VCPU budget replenishment policies
  - Deferrable server & sporadic server

- Task \( \tau_i: (C_i, T_i) \)
  - \( C_i \): Worst-case execution time (WCET)
  - \( T_i \): Minimum inter-arrival time

Any task or OS code can execute only if the corresponding VCPU has a non-zero remaining budget
**System Model (2)**

- **Physical interrupt** $I_{pi}^{pi}: (C_{pi}^{pi}, T_{pi}^{pi})$
  - A signal issued from a hardware device to a PCPU
  - Handled by the corresponding ISR of the hypervisor

- **Virtual interrupt** $I_{vi}^{vi}: (C_{vi}^{vi}, T_{vi}^{vi})$
  - A software signal from the hypervisor to a VCPU
  - Handled by the ISR of the guest OS while consuming the VCPU budget

Min. inter-arrival time expected at design time → Interrupt storms may happen at runtime

Diagram:
- **PCPU 1**
  - Hypervisor
  - VCPU $v_1$
  - Task $\tau_1$

- **Physical Intr.**
  - VM Exit
  - ISR
  - VM Enter

- **Virtual Intr.**
  - VM Exit
  - ISR
  - Task $\tau_2$

- **Trap**
  - EOI
  - VM Enter

**Interrupt-triggered execution flow**
Problems with Virtual Interrupts (1)

- Virtual interrupt
  - Main difference between interrupt handling in virtualized and non-virtualized environments

- Problem 1: **Timing penalties** to virtual interrupt handling
  - VCPU budget depletion and VCPU preemption
Problems with Virtual Interrupts (2)

- Problem 2: Virtual interrupt storms
  - VCPU typically has a *fraction* of physical CPU time as its budget
  - Negative impact of virtual interrupt storm can be much significant than physical interrupt storms

- Prior work developed for non-virtualized systems
  - Cannot address virtual interrupt storms due to the unawareness of the passage of physical time within a VM
vINT Overview

- Conceptually splits virtual interrupt handling from the VCPU of regular tasks in an analyzable way
  - Used pseudo-VCPU abstraction
  - Prioritizes virtual interrupt handling
  - Does not require any guest OS modification
Pseudo-VCPU Parameters

• Same types of parameters as a regular VCPU: \( (C_p^v, T_p^v) \)
• Budget replenishment period \( T_p^v \)
  – Equal to or greater than the minimum inter-arrival time of the associated interrupt
• Execution budget \( C_p^v \)

\[
C_p^v = \left[ \frac{T_p^v}{T_i^v} \right] C_i^v + \sum_{\substack{I_j^v \in V(I_i^v) \wedge \text{pseudo}(I_j^v) = \emptyset}} \left[ \frac{T_i^v}{T_i^v} \right] C_j^v
\]

Sum of execution times of ISR and interrupt-triggered task

Extra budget to reduce blocking time on interrupt handling
Pseudo-VCPU Realization

- Pseudo-VCPU does not have an execution context
  - vINT handles a virtual interrupt as if it was handled in its pseudo-VCPU

vINT checks the remaining budget of the corresponding pseudo-VCPU

vINT let VCPU $v_1$ override the budget and priority of the pseudo-VCPU

vINT supports nested interrupt handling by using an EOI signal
Analysis

• Scope of our analysis
  – Interrupt handling time
  – VCPU schedulability
  – Task schedulability

• Considers four different use cases

<table>
<thead>
<tr>
<th>VCPU budget replenish policies</th>
<th>With vINT</th>
<th>Without vINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferrable server</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Sporadic server</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
Interrupt Handling Time Analysis

- **Interrupt handling time**
  - Sum of physical and virtual interrupt response times

- **Physical interrupt response time**
  \[ W_{i}^{pi,n+1} = C_{i}^{pi} + \sum_{I_{h}^{pi} \in P(I_{i}^{pi}) \land \pi_{h} > \pi_{i}^{pi}} \left[ \frac{W_{i}^{pi,n}}{T_{h}^{pi}} \right] C_{h}^{pi} \]

- **Virtual interrupt response time**

  **[ without vINT ]**
  \[ W_{j}^{vi,n+1} = C_{j}^{vi} + \sum_{I_{h}^{vi} \in V(I_{j}^{vi}) \land \pi_{h} > \pi_{j}^{vi}} \left[ \frac{W_{j}^{vi,n} + J_{h}}{T_{h}^{vi}} \right] C_{h}^{vi} \]

  Delay from VCPU budget depletion

  Delay from time-triggered tasks

  **[ vINT ]**
  \[ W_{j}^{vi,n+1} = C_{j}^{vi} + B_{p,j}(W_{j}^{vi,n}) + \sum_{I_{h}^{vi} \in P(v_{p})} \left[ \frac{W_{j}^{vi,n}}{T_{h}^{vi}} \right] C_{u}^{pi} \]

  Delay from higher-priority interrupt handling
Outline

• Introduction

• vINT Framework

• Evaluation
  – Performance characteristics of vINT
  – Implementation
  – Case study

• Conclusion
Performance Characteristics of vINT

- **Purpose**: Empirically investigate the performance characteristics and benefits of vINT

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSbase</td>
<td>Deferrable Server without vINT (baseline)</td>
</tr>
<tr>
<td>SSbase</td>
<td>Sporadic Server without vINT (baseline)</td>
</tr>
<tr>
<td>DSvINT</td>
<td>Deferrable Server with vINT</td>
</tr>
<tr>
<td>SSvINT</td>
<td>Sporadic Server with vINT</td>
</tr>
</tbody>
</table>

- **Experimental setup**
  - Used randomly-generated task sets and interrupt sets
  - **Metrics**
    - Percentage of schedulable task sets
    - Percentage of serviceable interrupt sets
Experimental Results (1)

- Interrupts with short inter-arrival times
  - Task schedulability

![Graph showing task schedulability](image)

- Interrupt service rate

![Graph showing interrupt service rate](image)

vINT has benefits in both task scheduling and interrupt handling.
Experimental Results (2)

- WCET of interrupt handlers
  - Task schedulability
    ![Graph showing task schedulability](image1)
    - vINT shows slightly lower task schedulability
  - Interrupt service rate
    ![Graph showing interrupt service rate](image2)
    - But vINT provides significantly higher interrupt service rates
Case Study

• System configuration
  – Hypervisor: KVM of Linux/RK
    • Chosen for convenience
    • vINT applied to a Gigabit PCI NIC
  – Guest VM
    • OS: Unmodified Linux kernel 3.10
    • Tasks: Netperf (network benchmark tool), Mplayer (movie player), Busyloop (background task)
Netperf Round-Trip Latency

- Highly affected by system’s interrupt handling time

- Netperf with vINT: handles 95% of round-trips in 200 μsec
- Netperf without vINT: only 50% during that time
Mplayer QoS under Interrupt Storms

- Measured fps (frames-per-second) of video playback
  - MPEG2 video stream recorded in 29.97 fps
  - X-axis: total VCPU budget assigned

- Mplayer with vINT: nearly unaffected
- Mplayer without vINT: dropped from 29.97 fps to 6 fps
Conclusions

• **vINT**: an interrupt handling framework for RT virtualization
  – Provides *responsive and bounded* interrupt handling time
  – Protects real-time tasks from *interrupt storms*
  – Supports *unmodified guest OSs*

• **Analysis and Experimental Results**
  – Timely interrupt handling and good task schedulability in most cases
  – A system designer can choose a trade off between task schedulability and interrupt handling time for each interrupt

• **Implementation and Case study**
  – KVM + Linux/RK: [https://rtml.ece.cmu.edu/redmine/projects/rk/](https://rtml.ece.cmu.edu/redmine/projects/rk/)

• **Future Work**
  – Memory interference, efficient VCPU resource allocation