Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems

Robert Kaiser

Distributed Systems Lab
University of Applied Sciences
Wiesbaden, Germany

IIES 08, Glasgow, 1-Apr-2008
1. Introduction

2. Timing properties of virtual machines
   - Hierarchical scheduling
   - Proportional share scheduling
   - Effect on latency
   - Overhead vs. Latency

3. Supporting real-time systems
   - Workload classification
   - Requirements of different load classes
   - Scheduling infrastructure

4. Conclusion
1 Introduction

2 Timing properties of virtual machines
   - Hierarchical scheduling
   - Proportional share scheduling
   - Effect on latency
   - Overhead vs. Latency

3 Supporting real-time systems
   - Workload classification
   - Requirements of different load classes
   - Scheduling infrastructure

4 Conclusion
Motivation (1)

- Embedded systems are becoming more powerful
- Disproportionate relation of performance vs. cost (e.g. 2 x cost ⇒ 10 x performance)
- Using this potential makes sense..
  - .. economically: same/more performance at lower cost
  - .. technically: fewer components
    ⇒ fewer points of failure
- More functionality per controller
- Complexity must be managed, though...
Motivation(2)

- Software of embedded systems becomes more complex
- Applications from different suppliers..
  - .. may serve different (unrelated) purposes
  - .. may not be aware of each other
  - .. need to be securely isolated (liability, safety, security)
- ⇒ A case for virtualisation!
A Case for Virtualisation

- Complex embedded systems need management
  - Applications must be kept from interfering with each other
    → Need fault containment
  - Applications may be programmed against different OS APIs
    → Need to support multiple OSs in parallel

⇒ Similar requirements as server consolidation

- Many embedded systems already have the necessary resources (e.g. MMU/MPU)
- But can virtualisation be applied to embedded systems ”as is”?
  - Workload is often known in advance
    → Dynamic resource management is less important
  - Many applications have soft/hard timing requirements

⇒ Need to support real-time guest systems!
Outline

1. Introduction

2. Timing properties of virtual machines
   - Hierarchical scheduling
   - Proportional share scheduling
   - Effect on latency
   - Overhead vs. Latency

3. Supporting real-time systems
   - Workload classification
   - Requirements of different load classes
   - Scheduling infrastructure

4. Conclusion
Hierarchical scheduler

- Virtual machine monitor acts as a scheduler
- Virtual machines host guest OSs which –again– act as schedulers
- \( \rightarrow \) VMM is the first level in a scheduler hierarchy
- VMM is unaware of this: ”VMs are just processes”
- Goal: share the CPU among VMs
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
- Smaller quanta: → better approximation
- ... but also: more overhead
  - Scheduler invocations
  - Cache & TLB losses
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
  - Smaller quanta: → better approximation
  - ... but also: more overhead
    - Scheduler invocations
    - Cache & TLB losses
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
- Smaller quanta: → better approximation
- ... but also: more overhead
  - Scheduler invocations
  - Cache & TLB losses
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
- Smaller quanta: → better approximation
- ... but also: more overhead
  - Scheduler invocations
  - Cache & TLB losses
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
- Smaller quanta: → better approximation
- ... but also: more overhead
  - Scheduler invocations
  - Cache & TLB losses
Proportional share scheduling

- Ideal: VMs continuously share CPU
- Reality: approximation by time multiplexing
- Smaller quanta: → better approximation
- ... but also: more overhead
  - Scheduler invocations
  - Cache & TLB losses
Effect on latency

- Comparing RT process running on real hardware vs. virtual machine
- Virtual machine may experience a "blackout"
- Worst case delay: $T_{del} = (N - 1) \cdot T_{vm} + N \cdot T_{sw}$
Effect on latency

- Comparing RT process running on real hardware vs. virtual machine
- Virtual machine may experience a "blackout"
- Worst case delay: \( T_{del} = (N - 1) \cdot T_{vm} + N \cdot T_{sw} \)
Effect on latency

- Comparing RT process running on real hardware vs. virtual machine
- Virtual machine may experience a "blackout"
- Worst case delay: $T_{del} = (N - 1) \cdot T_{vm} + N \cdot T_{sw}$
Simulated overheads as function of quantum

- **Simulated\textsuperscript{a} Scheduler:** VTRR (proportional share)

\[ \text{Simulated overheads as function of quantum} \]

\[ \text{Overhead [%]} \quad \text{Delay bound [ms]} \]

\[ \text{Quantum [µs]} \]

\( f_{\text{flood}} \quad f_{\text{avg}} \quad \text{delay} \)

(3 VMs, CPU: Celeron@2.5GHz, 64k WSS)

\( \Rightarrow \) Observation: for quanta \( \leq 1 \text{ ms} \):
- overhead becomes significant
- Delay: proportional to quantum and number of VMs
- Here: for quanta \( \geq 1 \text{ ms} \)

\( \Rightarrow \) Delay \( \geq 5 \text{ ms} \)
- Contemporary RTOS: \( \sim 5-10\mu\text{s} \)

\( \text{aSee: Kaiser: Empirische Ermittlung} \)

\( \text{Cache-bedingter Umschaltverluste} \)
Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.

For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.

⇒ For these, virtualisation is applicable “as is”

But: there also exist applications for which such delay/jitter is unacceptable.

How to support these, too?

How to integrate this variety of requirements?
Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.

For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.

⇒ For these, virtualisation is applicable “as is”

But: there also exist applications for which such delay/jitter is unacceptable.

How to support these, too?

How to integrate this variety of requirements?
Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.

For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.

⇒ For these, virtualisation is applicable "as is"

But: there also exist applications for which such delay/jitter is unacceptable.

How to support these, too?

How to integrate this variety of requirements?
Upshot

- Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.
- For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.
- ⇒ For these, virtualisation is applicable "as is"
- **But**: there also exist applications for which such delay/jitter is unacceptable.
- How to support these, too?
- How to integrate this variety of requirements?
Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.

For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.

⇒ For these, virtualisation is applicable "as is"

But: there also exist applications for which such delay/jitter is unacceptable.

How to support these, too?

How to integrate this variety of requirements?
Using proportional share scheduling, the delay/jitter imposed by virtualisation is severe, but bounded.

For many practical real-time applications, a ten to hundred millisecond delay or jitter is entirely acceptable.

⇒ For these, virtualisation is applicable "as is"

**But:** there also exist applications for which such delay/jitter is unacceptable.

How to support these, too?

How to integrate this variety of requirements?
Mixture of workloads to support

- Real-time: must\(^1\) or should\(^2\) meet deadlines
- Two subclasses:
  - Time-driven: static schedule, typically periodic
  - Event-driven: scheduled in response to (unpredictable) events, assumed to be sporadic
- Non real-time: no need to meet deadlines, instead, try to utilise all resources
- Assumption: Each class uses a specific OS API
- \(\Rightarrow\) Guests and their VMs as a whole can be classified as one of:
  1. Time-driven, real-time
  2. Event-driven, real-time
  3. Non real-time

\(^1\) = "hard’ real-time
\(^2\) = "soft” real-time
Supporting non real-time VMs

- No deadlines to meet → VM delay not a problem
- If multiple non-RT VMs exist: must share CPU evenly
- Should utilise all available resources → self-suspend when not ready
- (→ VMM schedule becomes unpredictable)
- Allocation of time to real-time VMs is designed for the worst case
- Worst case occurs rarely (if at all)
- In the average case, real-time VMs will not use all of their allocated time
  ⇒ Dynamically re-assign unused time to non real-time VMs
- Real-time VMs take priority over non real-time VMs
- May need provisions to avoid starvation, though..
Supporting time-driven VMs

- Cause of VM delay: VMM schedule and local schedules not correlated
- Synchronise VMM schedule and local schedules of time-driven VMs
- Define VMM schedule to "enclose" all time-driven local schedules
- Restrictions:
  - Local schedules must not overlap
  - Local schedules must use same (or harmonic) periods
- Low jitter (e.g. for PLCs)

Resulting "super schedule" is strictly a function of time
Supporting event-driven VMs

- Event-driven VMs need access to CPU at arbitrary times
- ⇒ Need ability to preempt current VM
- Conflicts with time-driven VMs
- Two choices:
  - Give event-driven VMs precedence over time-driven VMs
    → Time-driven VMs experience jitter and delays
  - Give time-driven VMs precedence over event-driven VMs
    → Event-driven VMs are delayed
- Classical dilemma: no generic solution (for uniprocessor architectures)
- Approach must be flexible enough to allow both choices on a case by case basis
Combining time-driven, priority-based and prop. share

- Assign static priorities to VMs
- Assign VMs to *time domains* \((\tau_i)\) (represented by a set of ready queues, one per priority)
- A VM is scheduled iff
  - its time domain is *active*, and
  - it has the highest priority

Up to **two** time domains can be active at a time:
- \(\tau_0\): *background* domain, always active
- \(\tau_i, i = 0..N\): *foreground* domain, cyclically switched

VMs from foreground and background domain compete by priority
VMs of same domain and priority are scheduled to proportional share
Semantics of priorities and time domain

- **time-driven VMs:**
  - Foreground domain, individual, high priorities
  - ⇒ individual, time-driven schedules

- **event-driven VMs:**
  - Background domain, individual, high priorities
  - ⇒ can request CPU any time (iff sufficient priority)

- **non real-time VMs:**
  - Background domain, common, low priority
  - ⇒ share all unused CPU resources

![Diagram showing priority levels and time domain](image-url)
Semantics of priorities and time domain

- **time-driven VMs:**
  - Foreground domain, individual, high priorities
  - ⇒ individual, time-driven schedules

- **event-driven VMs:**
  - Background domain, individual, high priorities
  - ⇒ can request CPU any time (iff sufficient priority)

- **non real-time VMs:**
  - Background domain, common, low priority
  - ⇒ share all unused CPU resources
Semantics of priorities and time domain

- **time-driven VMs:**
  - Foreground domain, individual, high priorities
  - \(\Rightarrow\) individual, time-driven schedules

- **event-driven VMs:**
  - Background domain, individual, high priorities
  - \(\Rightarrow\) can request CPU any time (iff sufficient priority)

- **non real-time VMs:**
  - Background domain, common, low priority
  - \(\Rightarrow\) share all unused CPU resources
Implementation details

- First implementation: "PikeOS"
  µKernel (Sysgo AG)
- (Round robin instead of proportional share)
- Use cases so far: RTOS & Linux, PLC & Linux
- Priorities control (individually) precedence of time-driven vs. event-driven VMs
  → O(1) complexity
- High priority VMs trusted not to exceed their budgets
Outline

1. Introduction

2. Timing properties of virtual machines
   - Hierarchical scheduling
   - Proportional share scheduling
   - Effect on latency
   - Overhead vs. Latency

3. Supporting real-time systems
   - Workload classification
   - Requirements of different load classes
   - Scheduling infrastructure

4. Conclusion
Summary

- Applying virtualisation to embedded systems makes sense
- Current virtualisation approaches have limited real-time support
- Proposed approach can improve it (with some restrictions)
- Coexistence of time-driven and event-driven subsystems remains problematic
- Proposed approach can be implemented efficiently by means of the outlined scheduler infrastructure
Further work

- Use Xen as testbed
- Introduce budgets to limit high priority time consumption
- Multiprocessor (MultiCore) support
  - MultiCore CPUs are attractive for embedded systems
  - Decouple time-driven and event-driven VMs
  - Enable parallel programming ("coscheduling") under guest OS control
- Apply schedulability analysis to guest OSs
  - RMS: already covered (though not in this paper)
  - EDF: TBD
- Improve estimates for cache-related switch overheads
  - Validate simulation-based estimation under realistic workloads
  - Estimates should also be applicable processor migration cost
Thank you for your attention!

Any Questions?