Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

16.11.2018
HYPERVISOR DETERMINISM ON MODERN SOC

Robert Kaiser · Computer Engineering · RheinMain University of Applied Sciences

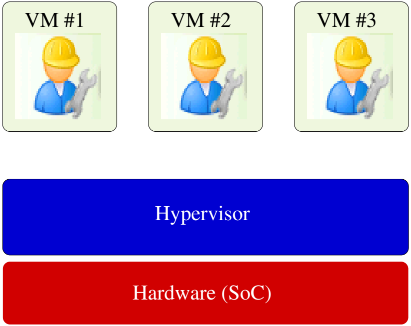
01
INTRODUCTION

WHY HYPERVISORS IN EMBEDDED SYSTEMS

- Origin: Server consolidation → also useful for *complex* embedded systems for:
    - Safety:
        - Mixed criticality systems
        - Software redundancy (e.g. multi version dissimlar code)
        - Online monitoring (e.g. for graceful degradation)
    - Security:
        - MILS Systems
        - Online monitoring (e.g. for intrusion detection)
    - Efficiency: Improve resource utilization
    - Combine different levels of real-time requirements
    - (Legal reasons: License isolation)

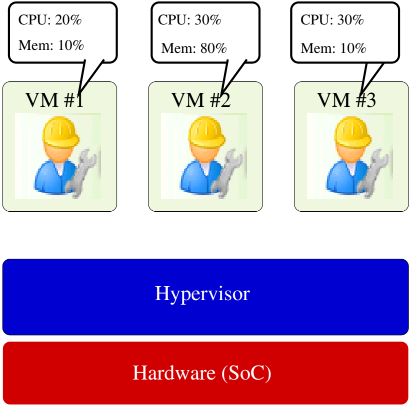
A HYPERVISOR PROVIDES ...

- ... multiple instances of the underlying physical machine
- ... each with its own subset of system resources (→ isolated and independent)
- ... each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



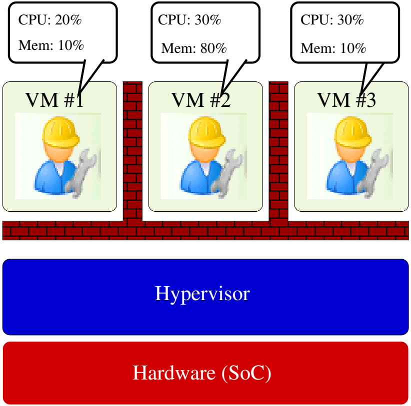
A HYPERVISOR PROVIDES ...

- ... multiple instances of the underlying physical machine
- ... each with its own subset of system resources (→ isolated and independent)
- ... each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



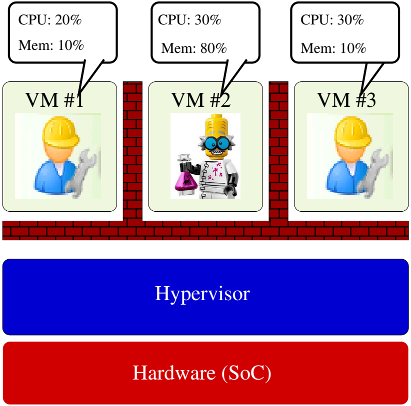
A HYPERVISOR PROVIDES ...

- ... multiple instances of the underlying physical machine
- ... each with its own subset of system resources (→ isolated and independent)
- ... each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



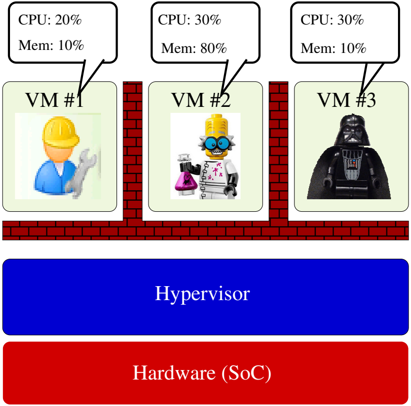
A HYPERVISOR PROVIDES ...

- ... multiple instances of the underlying physical machine
- ... each with its own subset of system resources (→ isolated and independent)
- ... each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



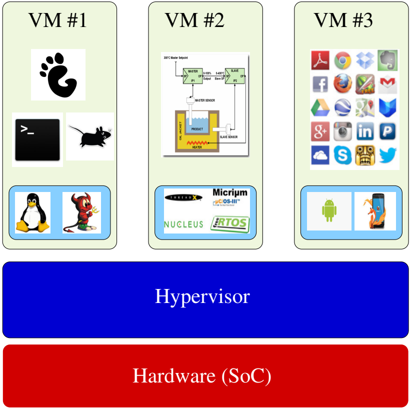
A HYPERVISOR PROVIDES …

- … multiple instances of the underlying physical machine
- … each with its own subset of system resources (→ isolated and independent)
- … each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



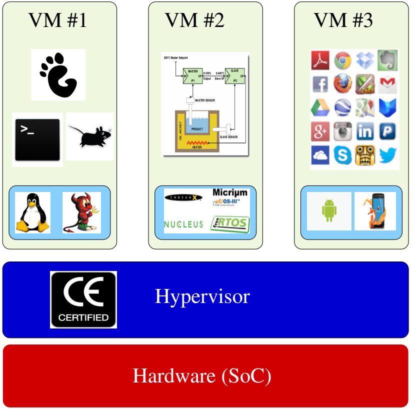
A HYPERVISOR PROVIDES …

- … multiple instances of the underlying physical machine
- … each with its own subset of system resources (→ isolated and independent)
- … each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



A HYPERVISOR PROVIDES ...

- ... multiple instances of the underlying physical machine
- ... each with its own subset of system resources (→ isolated and independent)
- ... each can run its own specialised OS w/ apps
- Sole mandatory trusted code for all: the hypervisor



HYPERVISOR: DEFINITION

- From Wikipedia, paraphrasing [PG74]:
  "A virtual machine monitor (VMM, also called hypervisor) is the piece of software that provides the abstraction of a virtual machine. There are three properties of interest when analyzing the environment created by a VMM:"
    - Equivalence / Fidelity:
      "A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly."
    - Resource control / Safety:
      "The VMM must be in complete control of the virtualized resources."
    - Efficiency / Performance:
      "A statistically dominant fraction of machine instructions must be executed without VMM intervention."

“ESSENTIALLY IDENTICAL“ ...

- *“Any program run under the hypervisor should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies.“*

⇒ Determinism is normally not within the scope of Hypervisors.

⇒ **Scope of Hypervisors must be extended.**

02

REQUIREMENTS AND NON-REQUIREMENTS

FEATURES WE WANT

- Safe isolation between VMs
  - Achieved by *resource partitioning*
- Interaction between VMs
  - Any interaction must be under hypervisor's control
- Temporal determinism
  - Requires extension (see above) → subject of this talk

FEATURES WE WANT

- Safe isolation between VMs ✓
  - Achieved by *resource partitioning*
- Interaction between VMs
  - Any interaction must be under hypervisor's control
- Temporal determinism
  - Requires extension (see above) → subject of this talk

FEATURES WE WANT

- Safe isolation between VMs ✓
  - Achieved by *resource partitioning*
- Interaction between VMs ✓
  - Any interaction must be under hypervisor's control
- Temporal determinism
  - Requires extension (see above) → subject of this talk

FEATURES WE WANT

- Safe isolation between VMs ✓
  - Achieved by *resource partitioning*
- Interaction between VMs ✓
  - Any interaction must be under hypervisor's control
- Temporal determinism ✗
  - Requires extension (see above) → subject of this talk

FEATURES WE MAY NOT NEED

- Virtual memory
  - Two aspects: virtual addressing and protection
  - For partitioning: just having protection suffices
  - Some low end SoCs lack mapping capability (only MPU)
- Dynamic reconfiguration
  - I.e. changing a VM's allocated resources at run-time
  - Live migration

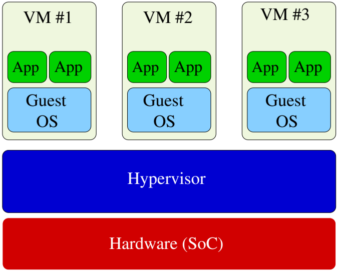
  → Significant complexity in HV **and** VM
  - Needed (if at all) only by best effort (non-realtime)VMs
- "Standard" ABI compatibility
  - e.g. IA32 / 64 / Windows
  - Often irrelevant for SoCs

03

HYPERVISOR BASICS

HYPERVISOR BASICS: ARCHITECTURAL CLASSIFICATION

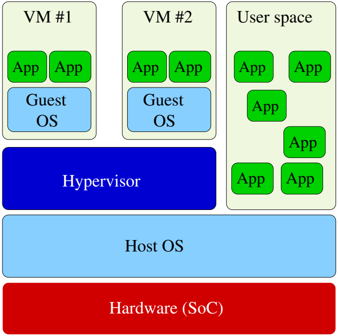
- Type 1: Run on bare metal
- Classical: Hardware assisted, full virtualization
  architecture must be “virtualisable“ (according to [PG74])
- Paravirtualisation & microkernels
  Privileged software (kernel) needs to be adapted
- *Binary rewriting*, aka “just in time paravirtualisation“:
  Full virtualisation for non-virtualisable architectures
- Special cases
  e.g. leveraging ARM TrustZone



HYPERVISOR BASICS: ARCHITECTURAL CLASSIFICATION

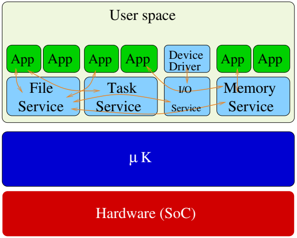
- Type 2: Run on another OS

- One more layer in scheduler hierarchy → needs to be controlled for determinism
- Benefits for embedded systems are questionable (See “Standard“ ABI compatibility)



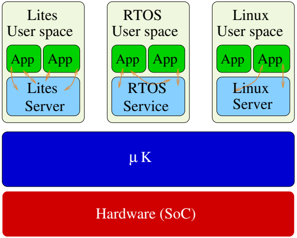
MICROKERNELS VS. HYPERVISORS

- Microkernels: Different origin …
  - Minimise privileged (e.g. kernel) code
  - Seperate policy from mechanism
  - IPC as central (only) service
- … but similar results
  - E.g. the Lites Server [Hel94] and L4Linux [HHL$^+$97] **were** paravirtualised UNIX kernels
- Generally more flexible
- No significant differences wrt. scheduling / determinism (however: ongoing attempts to push scheduling policy out of the kernel [GGB$^+$17])



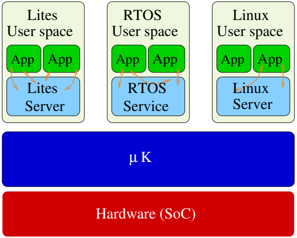
MICROKERNELS VS. HYPERVISORS

- Microkernels: Different origin …
  - Minimise privileged (e.g. kernel) code
  - Seperate policy from mechanism
  - IPC as central (only) service
- … but similar results
  - E.g. the Lites Server [Hel94] and L4Linux [HHL$^+$97] **were** paravirtualised UNIX kernels
- Generally more flexible
- No significant differences wrt. scheduling / determinism (however: ongoing attempts to push scheduling policy out of the kernel [GGB$^+$17])



MICROKERNELS VS. HYPERVISORS

- Microkernels: Different origin …
  - Minimise privileged (e.g. kernel) code
  - Seperate policy from mechanism
  - IPC as central (only) service
- … but similar results
  - E.g. the Lites Server [Hel94] and L4Linux [HHL+97] **were** paravirtualised UNIX kernels
- Generally more flexible
- No significant differences wrt. scheduling / determinism (however: ongoing attempts to push scheduling policy out of the kernel [GGB+17])



HYPERVISOR CPU ALLOCATION

- 1. Static allocation of CPUs to VMs
- Hardware as well as software solutions exist

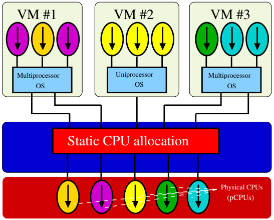
⊕ No[1] or very little software required

⊕ ~~No~~ Very little interference between VMs (e.g. system bus / L3 cache may still be shared)

⊕ No CPU sharing between VMs

→ Uniprocessor scheduling theory directly applicable

→ Easy to mix real-time and non-realtime VM payloads



⊕ Amenable to heterogenous multicore SoCs

⊖ Inflexible: #Cores $\geq$ #VMs → Can lead to poor utilization

⊖ Sharing, if needed, can be difficult

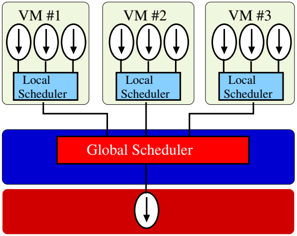
---

[1] for hardware solutions

HYPERVISOR CPU ALLOCATION

- 2. Hierarchical scheduler
- Dynamic allocation of cores to VMs

- ⊕ More flexible
- ⊕ Controlled sharing of CPU and other resources possible
- ⊕ Better utilisation of resources
- ⊕ Applicable to uniprocessor systems
- ⊖ More interference between VMs
- ⊖ Scheduling needs more consideration



04

HYPERVISOR SCHEDULING

TIME FROM A VM'S POINT OF VIEW

- Computation time and observed “wall clock time“ differ
  - Slowdown due to virtualization (e.g. trap & emulate)
    - Makes virtual processor run slower
    - Compensate by allocating more budget

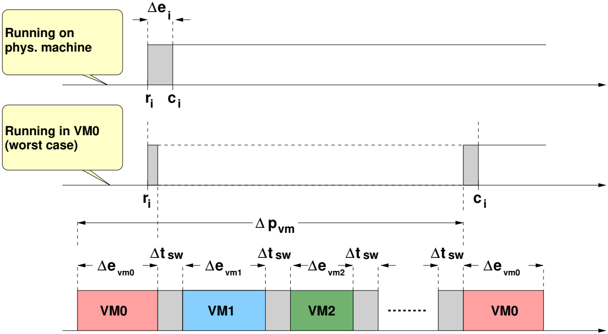
    → No problem for determinism
  - Slowdown due to sharing of CPU with other VMs
    - Causes “Blackouts“
      (CPU not available when VM has work to do)
    - Need to adapt hypervisor scheduling to either avoid or cope
  - Slowdown due to pollution of shared Caches / TLBs by other VMs
    - Makes virtual processor run slower following VM switch
    - Effect decays as CPU is “owned“ by VM for some time
    - May also leak information about other VM (*covert channel*)
    - Need to adapt hypervisor scheduling to either avoid or cope

TIME FROM A VM'S POINT OF VIEW

- Computation time and observed "wall clock time" differ
  - Slowdown due to virtualization (e.g. trap & emulate)
    - Makes virtual processor run slower
    - Compensate by allocating more budget
    - → No problem for determinism ✓
  - Slowdown due to sharing of CPU with other VMs
    - Causes "Blackouts"
      (CPU not available when VM has work to do)
    - Need to adapt hypervisor scheduling to either avoid or cope
  - Slowdown due to pollution of shared Caches / TLBs by other VMs
    - Makes virtual processor run slower following VM switch
    - Effect decays as CPU is "owned" by VM for some time
    - May also leak information about other VM (*covert channel*)
    - Need to adapt hypervisor scheduling to either avoid or cope

EFFECT ON LATENCY

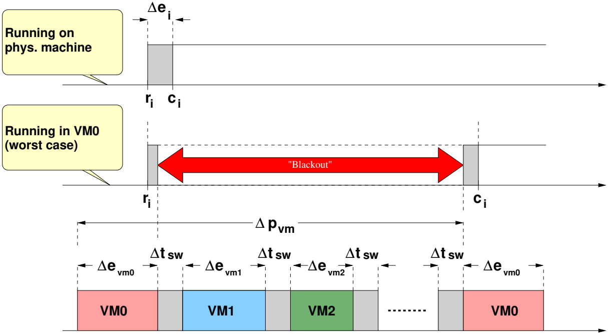
- Real-time process running in VM may experience a “blackout“
- Worst case delay: $\Delta t_{del_j} = \Delta t_{sw} + \sum_{i \neq j} \Delta e_{vm_i} + \Delta t_{sw}$



⇒ Imposed jitter/delay is severe, but bounded

EFFECT ON LATENCY

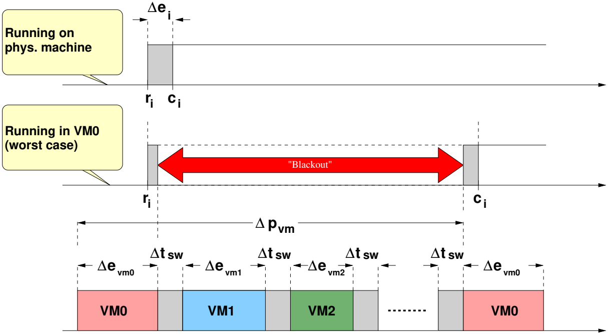
- Real-time process running in VM may experience a "blackout"
- Worst case delay: $\Delta t_{del_j} = \Delta t_{sw} + \sum_{i \neq j} \Delta e_{vm_i} + \Delta t_{sw}$



⇒ Imposed jitter/delay is severe, but bounded

EFFECT ON LATENCY

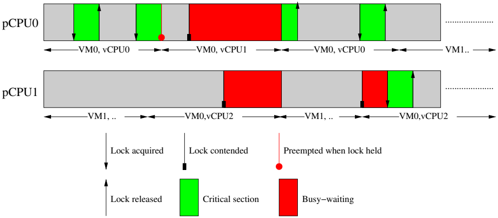
- Real-time process running in VM may experience a “blackout“
- Worst case delay: $\Delta t_{del_j} = \Delta t_{sw} + \sum_{i \neq j} \Delta e_{vm_i} + \Delta t_{sw}$



⇒ **Imposed jitter/delay is severe, but bounded**

LOCK HOLDER PREEMPTION PROBLEM

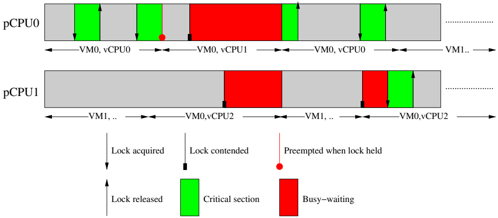
- Cause: Virtual CPU (*vCPU*) being preempted while holding a spinlock
- Guest OS is unaware of vCPU ↔ pCPU mapping
- May cause excessive CPU waste
- Similar to priority inversion problem



- **Countermeasures:**
  - “Helping“ [FB08]: complex interaction patterns
  - **At VM scheduler level: always co-schedule VMs**

LOCK HOLDER PREEMPTION PROBLEM

- Cause: Virtual CPU (*vCPU*) being preempted while holding a spinlock
- Guest OS is unaware of vCPU ↔ pCPU mapping
- May cause excessive CPU waste
- Similar to priority inversion problem



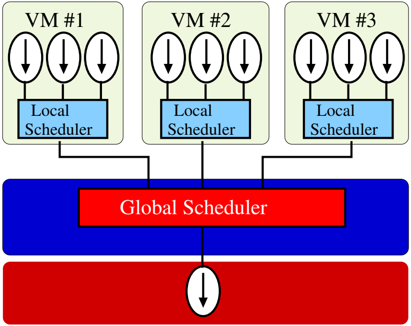
- **Countermeasures:**
  - “Helping“ [FB08]: complex interaction patterns
  - **At VM scheduler level: always co-schedule VMs**

05

HYPERVISOR SCHEDULERS

PROPORTIONAL SHARE SCHEDULING

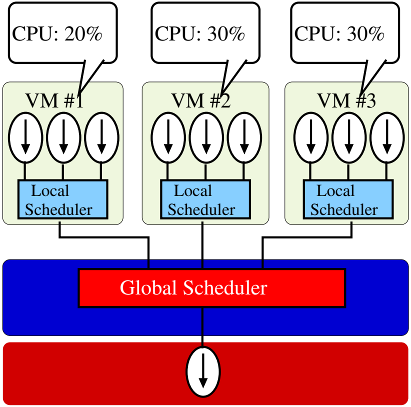
- Default strategy with most hypervisors:
- Pfair / proportional share scheduling
- VMs receive share (percentage) of CPU(s)
- Idealised assumption: VM “sees“ slower CPU, available at any time



→ Slowdown can (in theory) be compensated by allocating sufficient budget

PROPORTIONAL SHARE SCHEDULING

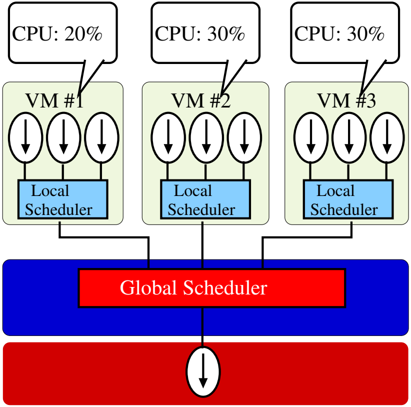
- Default strategy with most hypervisors:
- Pfair / proportional share scheduling
- VMs receive share (percentage) of CPU(s)
- Idealised assumption: VM “sees“ slower CPU, available at any time



→ Slowdown can (in theory) be compensated by allocating sufficient budget

PROPORTIONAL SHARE SCHEDULING

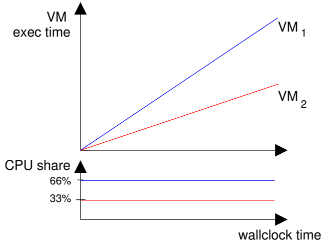
- Default strategy with most hypervisors:
- Pfair / proportional share scheduling
- VMs receive share (percentage) of CPU(s)
- Idealised assumption: VM “sees“ slower CPU, available at any time



→ Slowdown can (in theory) be compensated by allocating sufficient budget

PROPORTIONAL SHARE SCHEDULING

- Idealised situation ...
- ... is approximated
- Precision increases with frequency
- Limited by switch cost

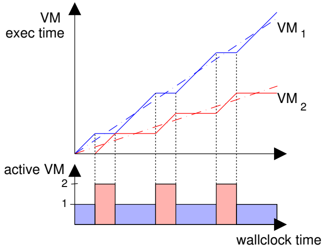


- Rule of thumb: Switch frequencies above 1-10kHz lead to excessive overhead

→ Applicable to best effort and “slow“ real-time systems

PROPORTIONAL SHARE SCHEDULING

- Idealised situation ...
- ... is approximated
- Precision increases with frequency
- Limited by switch cost

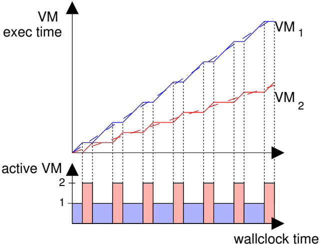


- Rule of thumb: Switch frequencies above 1-10kHz lead to excessive overhead

→ Applicable to best effort and “slow“ real-time systems

PROPORTIONAL SHARE SCHEDULING

- Idealised situation ...
- ... is approximated
- Precision increases with frequency
- Limited by switch cost

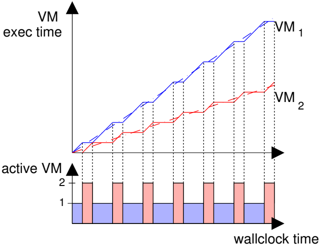


- Rule of thumb: Switch frequencies above 1-10kHz lead to excessive overhead

→ Applicable to best effort and “slow“ real-time systems

PROPORTIONAL SHARE SCHEDULING

- Idealised situation ...
- ... is approximated
- Precision increases with frequency
- Limited by switch cost

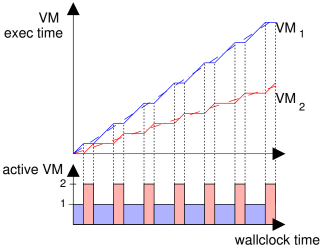


- Rule of thumb: Switch frequencies above 1-10kHz lead to excessive overhead

→ Applicable to best effort and “slow“ real-time systems

PROPORTIONAL SHARE SCHEDULING

- Idealised situation ...
- ... is approximated
- Precision increases with frequency
- Limited by switch cost



- Rule of thumb: Switch frequencies above 1-10kHz lead to excessive overhead

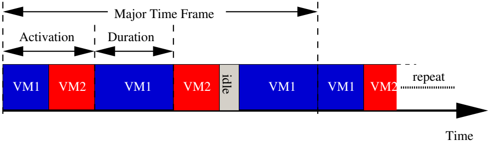
→ Applicable to best effort and “slow“ real-time systems

PROPORTIONAL SHARE SCHEDULING

- Shortcomings from a real-time perspective:
  - Scheduler is work conserving: idle VMs give up their time slice
    - Makes actual allocation unpredictable
    - Breaks (to some extent) VM isolation by opening covert channels
  - Generally no admission test

    → per VM absolute budget could change at any time
- Targeted at best effort, greedy VMs

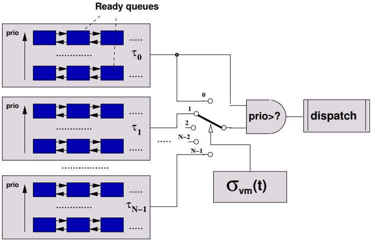
TIME PARTITIONING



- VMs have statically configured time slots (*durations*) within periodic *major time frame*
- (Different modes are possible)
- Time slots are enforced: exceeding budget is a violation
- Non-work conserving: VMs must “burn“ budget when idle
- Simple enough for formal reasoning
- Targeted at real-time systems

PIKEOS SCHEDULER [KF14]

- Assign priority ranges to VMs
- Assign *time domains*[2] (τ_i) to VMs
- A VM is scheduled iff
    - it has the highest priority, **and**
    - its time domain is *active*



- Up to **two** time domains can be active at a time:
    - τ_0: *background* domain, always active
    - τ_i, $i = 1..N$: *foreground* domain, switched by partition schedule
- VMs from foreground or background domain selected by priority
- Guaranteed time partitioning, but also work conserving:
- Over-allocated budget not used by high priority, foreground real-time VMs falls back to low priority, background best-effort VMs

---

[2] (represented by a set of ready queues, one per priority)

06

HIERARCHICAL SCHEDULING

SUPPORTING DIFFERENT VM CLASSES

- VM scheduler must be aware of the nature of task sets executing in a VM
- **Real-time:** must or should [3] 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** (best effort): no need to meet deadlines, instead: try to utilise all resources (“greedy“)
- Assumption: Each class uses a specific OS API

⇒ Guests and their VMs as a whole can be classified as one of:

1. Time-driven, real-time (TRT)
2. Event-driven, real-time (ERT)
3. Non real-time (NRT)

- VM scheduler must guarantee sufficient resources for all real-time guests

---

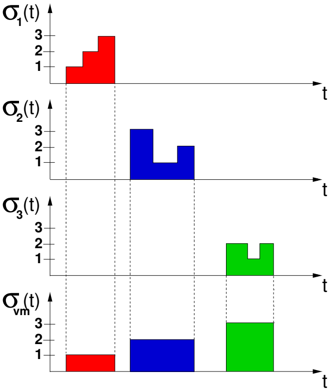
[3] “must“ = “hard“, “should“ = “soft“ real-time

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

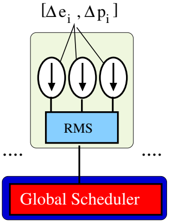
- Event-driven VMs need access to CPU at arbitrary times

$\Rightarrow$ Need ability to preempt current VM

- Conflicts with time-driven VMs
- Two choices:
  - Give event-driven VMs precedence over time-driven VMs
    $\rightarrow$ Time-driven VMs experience jitter and delays
  - Give time-driven VMs precedence over event-driven VMs
    $\rightarrow$ 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
- How to derive necessary period/budget for real-time VMs?

PROXY MODEL

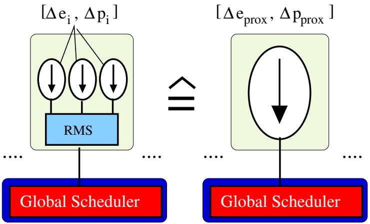
- VM's point of view:
  1. $P = \{1, .., n\}$: Set of periodic tasks with:
     - Execution time Δe_i
     - Period Δp_i
  2. Scheduled, e.g. by RMS (fixed prio)



- **Abstraction:** equivalent representation at the hypervisor level: <u>one</u> periodic proxy process with parameters Δe_{prox}, Δp_{prox}

PROXY MODEL

- VM's point of view:
  1. $P = \{1, .., n\}$: Set of periodic tasks with:
     - Execution time Δe_i
     - Period Δp_i
  2. Scheduled, e.g. by RMS (fixed prio)



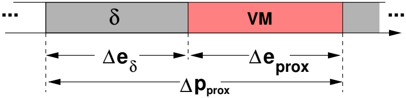
- **Abstraction:** equivalent representation at the hypervisor level: <u>one</u> periodic proxy process with parameters Δe_{prox}, Δp_{prox}

RMS INTEGRATION

- RMS rule:

$$\Delta p_i > \Delta p_j \Leftrightarrow prio(i) < prio(j)$$

- Concept of the proxy process:



1. To guarantee timeliness on the local level, set: $\Delta p_{prox} \leq min(\Delta p_i) \forall i \in P$
2. Map the time quantum **not** used by the proxy to a "*parasite process*" δ, in the worst case of higher priority.
3. Since $\Delta p_\delta \leq \Delta p_{prox} \leq min(\Delta p_i) \forall i \in P$, δ can be added to the set P as new task having the highest priority.

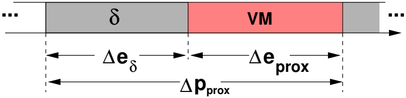
⇒ **RMS remains applicable to new set**

RMS INTEGRATION

- RMS rule:

$$\Delta p_i > \Delta p_j \Leftrightarrow prio(i) < prio(j)$$

- Concept of the proxy process:



1. To guarantee timeliness on the local level, set: $\Delta p_{prox} \leq min(\Delta p_i) \forall i \in P$
2. Map the time quantum **not** used by the proxy to a "*parasite process*" δ, in the worst case of higher priority.
3. Since $\Delta p_\delta \leq \Delta p_{prox} \leq min(\Delta p_i) \forall i \in P$, δ can be added to the set P as new task having the highest priority.

$\Rightarrow$ **RMS remains applicable to new set**

UTILIZATION BOUNDS

Question: What are the lowest upper bounds on utilization of the process set $\{\delta, 1, ..., n\}$

- Derivation analogous to Liu/Layland (see [KZ09]):

$$U_{min} = n \cdot \left(\sqrt[n]{\frac{2}{U_\delta + 1}} - 1 \right)$$

$$\lim_{n \to \infty} U_{min} = \ln \left(\frac{2}{U_\delta + 1} \right)$$

- Where: $U_\delta = \frac{\Delta e_\delta}{\Delta p_{prox}}$ is the CPU utilization by the parasite process δ

POXY TASK PARAMETERS

- Proxy period (see above): $\Delta p_{prox} = min(\Delta p_i)$
- Proxy execution time:

$$\begin{aligned} \Delta e_{prox} + \Delta e_{\delta} &= \Delta p_{prox} (\text{see above}) \\ \Rightarrow U_{prox} &= \frac{\Delta e_{prox}}{\Delta p_{prox}} = 1 - U_{\delta} \\ \Rightarrow \Delta e_{prox} &= 2 \cdot \Delta p_{prox} \cdot \left(1 - \frac{1}{\left(\frac{U}{n} + 1\right)^{n}}\right) \\ \lim_{n \to \infty} \Rightarrow \Delta e_{prox} &= 2 \cdot \Delta p_{prox} \cdot \left(1 - e^{-U}\right) \end{aligned}$$

⇒ **Can compute $\Delta e_{prox}, \Delta p_{prox}$ for all VMs using RMS**

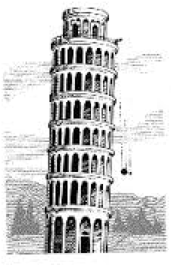
⇒ Input to global scheduler, planning using RMS or EDF

A similar derivation is also possible for EDF [Kai09]

07

CONTEXT SWITCH COST

SLEDGEHAMMER APPROACH



- *Estimate* switch cost
- Describe switch behaviour with a simple model
- "Calibrate" model with experimentally gathered data
- Cons/Pros:
    - ⊖ imprecise
    - ⊖ no proof (only empirical evidence)
    - ⊕ simple computation
    - ⊕ only superficial platform information required
- At any rate: better than neglecting ...

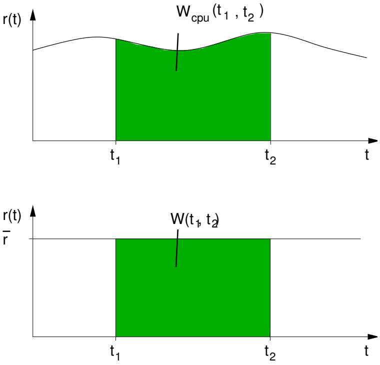
MODEL

- Computational Power, or Progress rate: $r(t)$
- Work (or Service): $W_{cpu}(t)$

$$W_{cpu}(t_1, t_2) = \int_{t_1}^{t_2} r(\tau) d\tau$$

- Equivalent constant workload (i.e. $r(t) = \bar{r}$):

$$W(t_1, t_2) = (t_2 - t_1) \cdot \bar{r}$$



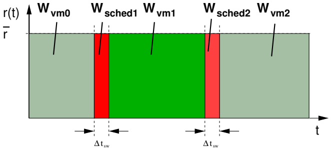
SWITCH OVERHEADS

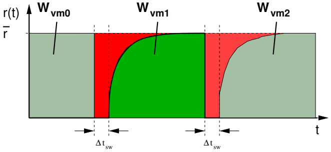
- Execution time lost by task switching
  - Activity of other proceses (<u>not</u> overhead)
  - Scheduler activity (Overhead)
- Assumption: Fixed scheduler execution time ($= \Delta t_{sw}$)

⇒ Cost per scheduler invocation:

$$W_{sched} = \Delta t_{sw} \cdot \bar{r}$$

- On process switch[4]: more overhead
- Both can be accounted to processes





---

[4] (Note: Process switch $\neq$ scheduler invocation)

PROCESS SWITCH COST

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

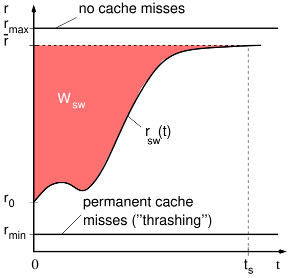
- Process switch cost: caused by Cache-/TLB-misses
- No discrete time window but "slowdown" of CPU, i.e. temporarily lowered progress rate

$\Rightarrow$ Cost per process switch:

$$W_{sw}(t) = t \cdot \bar{r} - \int_0^t r_{sw}(\tau) d\tau$$

- Relative loss:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t \frac{r_{sw}(\tau)}{\bar{r}} d\tau$$



Process switch at $t = 0$

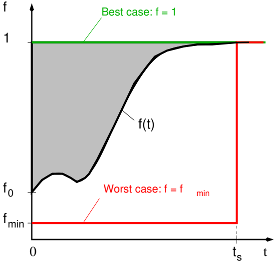
PAYLOAD SHARE

- Share of computational power consumed by payload:

$$f(t) := \frac{r_{sw}(t)}{\bar{r}}$$

- Thus:

$$O_{sw}(t) = 1 - \frac{1}{t} \int_0^t f(\tau) d\tau$$



Process switch at $t = 0$

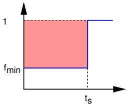
- Problem: $f(t)$ is unknown, however
  - best case: $f(t) = 1$
  - worst case: $f(t) = f_{min} > 0$
  - realistic case: somewhere between ..

APPROXIMATING PAYLOAD SHARE

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

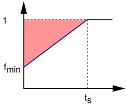
- Use time-dependent functions $f(t)$, e.g.:
- "cache flooding" (worst case) ...

$$f_{flood}(t) = \begin{cases} f_{min}, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



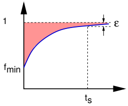
- ... or linear ...

$$f_{lin}(t) = \begin{cases} f_{min} + \frac{1-f_{min}}{t_s} \cdot t, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases}$$



- ... or exponential function ...

$$f_{exp}(t) = 1 + (f_{min} - 1) \cdot e^{-kt}$$



- all parametrised by f_{min}, t_s

⇒ Can compute loss per process switch

- Use different $f(t)$ depending on timing reqirements, e.g.
  - "hard" real-time → use $f_{flood}(t)$
  - "soft" real-time → use $f_{lin}(t)$ or $f_{exp}(t)$

GOALS AND METHODS

- Question: How to find appropriate values for (t_s, f_{min}, etc.)?
- Empirical approach: Measure, 2 goals:
  1. Demonstrate/validate worst case behaviour (”flooding”)
  2. Determine realistic parameters
- Method:
  - Bring caches into a defined state (invalidate / read-fill / write-fill)
  - Read- or write-access data in a previously uncached memory region of configurable size (”working set”)
  - Measure: Time used for a given (variable) number of accesses

RESULTS

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

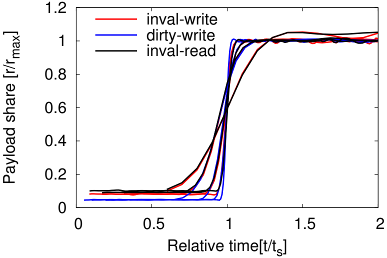
- MPC 5200 @ 400MHz: Simple, single-level cache, 16kB

Testcase	WSS	Cache	f_{min}	t_s
consec_wr	2k	dirty	0.05	17μs
consec_wr	4k	dirty	0.05	31μs
consec_wr	8k	dirty	0.05	59μs
consec_wr	16k	dirty	0.05	116μs
consec_wr	2k	invd	0.13	10μs
consec_wr	4k	invd	0.09	19μs
consec_wr	8k	invd	0.08	35μs
consec_wr	16k	invd	0.08	69μs
consec_rd	2k	invd	0.13	10μs
consec_rd	4k	invd	0.10	19μs
consec_rd	8k	invd	0.09	35μs
consec_rd	16k	invd	0.10	68μs

invd = cache invalidated, dirty = cache flood-filled

- $t_s \sim WSS$
- f_{min} between 5% und 13% independent of WSS

- Normalised values:



$\Rightarrow$ Matches model behaviour

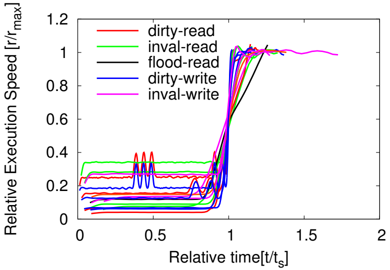
SOC EXAMPLES

Plat	**WSS**	**Cache**	f_{min}	t_s
i.MX6	16k	dirty-w	0.063	60μs
i.MX6	32k	dirty-w	0.068	119μs
i.MX6	64k	dirty-w	0.131	210μs
i.MX6	16k	inv-w	0.134	29μs
i.MX6	32k	inv-w	0.124	55μs
i.MX6	64k	inv-w	0.267	107μs
i.MX6	128k	inv-r	0.338	220μs
Exynos	32k	dirty-w	0.177	22μs
Exynos	64k	dirty-w	0.241	40μs
Exynos	16k	inv-w	0.161	13μs
Exynos	32k	inv-w	0.184	22μs
Exynos	64k	inv-w	0.238	40μs
KZM	16k	dirty-w	0.05	240μs
KZM	32k	dirty-w	≈0.5	470μs
KZM	64k	dirty-w	≈0.5	≈1050μs
KZM	16k	inv-w	0.11	114μs
KZM	32k	inv-w	≈0.18	208μs
KZM	64k	inv-w	≈0.18	312μs

Testcase: writing to adjacent cache lines

inv = Cache invalidated, dirty = cache flood-filled

- Normalised values (i.MX6):



- t_s grows with "working set"

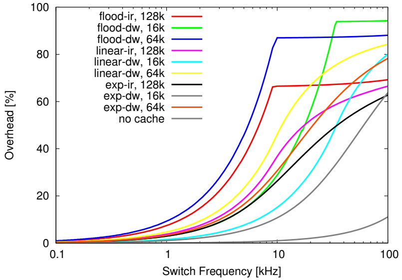
  (roughly proportional ..)
- f_{min} between (here) 5% und 34%, depending on WSS

  (due to 2-level cache)

⇒ Qualitatively: expected behaviour

SCHEDULER SIMULATION

- “Lowest lag first“ proportional share scheduler
- Cache load simulations: linear, exponential and flood
- Configured with f_{min}, t_s as measured on i.MX6



- 2 Tasks, 50% CPU each
- Assumed scheduler exec time: $1\mu s$

$\Rightarrow$ Shows trade off between continuity ↔ switch cost

$\Rightarrow$ In the given case (i.MX6), switch frequencies higher than $\approx$ 2-3 kHz lead to excessive overheads!

$\Rightarrow$ Similar results for other platforms [Kai08]

UPSHOT

- Processor performance can drop down to $\sim 5\%$ (worst case) if context switches occur frequently
- To compensate in such cases, budgets for critical real-time VMs must be increased accordingly
- Less critical tasks can use a less pessimistic payload share function, resulting in less overhead
- Since overheads and slowdowns are attributed to individual VMs, other VMs are not affected (except for the global admission test)

08

SUMMARY AND CONCLUSION

SUMMARY

Recommendations how to avoid or cope with ...

- Slowdown due to sharing of CPU with other VMs
  - For “slow“ real-time tasks ($\Delta p_{prox} \geq\sim 100ms$):
    - just use standard proportional scheduling
    - however, make sure sure budget can be guaranteed (e.g. static admission test at system configuration time)
  - For “fast“ real-time tasks ($\Delta p_{prox} <\sim 100ms$):
    - increase budget to compensate for switch costs
    - use time partitioning to enforce budgets
    - for time-triggered tasks: use “enclosing super schedule“
    - for event-triggered tasks: if possible, assign to core(s) different from event triggered tasks

SUMMARY

Recommendations how to avoid or cope with ...

- Slowdown due to pollution of shared caches / TLBs
  - (again,) increase budget to compensate for switch costs
  - partition caches (not covered here)
  - for security-sensitive tasks:
    - force cache flush (in the kernel) before switching contexts
    - enforce consumption of full budget for every job execution to avoid cache side channels

CONCLUSION

- Achieving hypervisor temporal determinism is possible!
- However, applicability of common hypervisors intended for server consolidation is limited:
    - Put significant effort into unneeded features, (thus increasing the amount of trusted code)
    - Fail to guarantee timely scheduling for “fast“ real-time workloads
- Classification and corresponding treatment of different workloads is necessary
- External requirements of real-time workloads can be computed from their task parameters

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

09
REFERENCES

REFERENCES

- Thomas Friebel and Sebastian Biemueller, *How to deal with lock holder preemption [extended abstract]*, 2008.
- Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer, *Temporal capabilities: Access control for time*, 2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017, 2017, pp. 56–67.
- Johannes Helander, *Unix under mach: The lites server*, 1994.
- Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter, *The performance of microkernel-based systems*, Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP) (St. Malo, France), October 5–8 1997.
- Robert Kaiser, *Estimating Context Switch Cost: A Practitioner's Approach*, OSPERT 2008 – Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (Praque, Czech Republic), UNC Technical Report, no. TR08-010, University of North Carolina, July 2008, pp. 73–82.
- ———, *Virtualisierung von Mehrprozessorsystemen mit Echtzeitanwendungen*, Ph.D. thesis, Universität Koblenz-Landau, June 2009.
- Robert Kaiser and Rudolf Fuchsen, *Method for distributing computing time in a computer system*, 2014, US Patent US8695004B2.
- Robert Kaiser and Dieter Zöbel, *Quantitative Analysis and Systematic Parametrization of a Two-Level Real-Time Scheduler*, ETFA 2009 – 14th IEEE International Conference on Emerging Technologies and Factory Automation, Mallorca, Spain, September 2009.
- Gerald J. Popek and Robert P. Goldberg, *Formal requirements for virtualizable third generation architectures*, Commun. ACM **7** (1974), no. 7, 412–421.